NTNU
Norwegian University of
Science and Technology

Graphical User Interface for Petroleum Field Optimization Software (FieldOpt)

Development of Interface for Configuring
Driver Files

Karoline Louise Rykkelid

Master of Science in Engineering and ICT	
Submission date:	December 2016
Supervisor:	Jon Kleppe, IPT
Co-supervisor:	Mathias Bellout, IPT
	Einar Baumann, IPT

Norwegian University of Science and Technology
Department of Petroleum Engineering and Applied Geophysics

Abstract

This Master's thesis concerns the further development of FieldOpt, a petroleum field optimization software, in close collaboration with the Petroleum Cybernetics Group at NTNU. This particular part of the development has been about creating a graphical user interface(GUI) for configuring FieldOpt driver files. As the present way of configuring was insufficient, especially in terms of user-friendliness, we wanted to present a new and better solution that hopefully satisfy users on a whole different level.

The thesis idea originates from PhD student Einar Baumann's recommandation on further work for FieldOpt, from his master's thesis. The development is in addition based upon the structure of the JSON driver file created by Baumann.
We present the utilities like software, structure, and library, in addition to an introduction to user interfaces and design guidelines used during the development. This is followed by the established system requirements; what should be required and expected of the system together with the necessary functionalities. And then the GUI design and interface elements are presented. Based on this, we have designed and implemented an interface, and presented the result. We have evaluated the result by comparing the solution to the requirements and guidelines set for this GUI. Some reflection on the process and solution is also included in this report.

This thesis has resulted in a solution that has the potential to be the FieldOpt driver configuration interface, but is not a fully functioning interface. However, at the current state we think it has facilitated and laid the foundation for further development and finalization of the interface that will help users to easily configure FieldOpt driver files.

Sammendrag

Denne masteroppgaven tar for seg deler av den videre utviklingen av FieldOpt, et programvarerammeverk for optimaliseringsproblemer innen petroleumsfeltet. I tett samarbeid med *Petroleum Cybernetics Group* (på NTNU) har det blitt utviklet et grafisk brukergrensesnitt(GUI) for konfigurering av FieldOpt-driverfiler. Da dagens løsning virker mangelfull når det gjelder brukervennlighet, ønsker vi å presentere og lage en bedre løsning som tilfredsstiller brukerne på et høyere nivå.

Oppgaven baserer seg på Einar Baumanns anbefaling om videre arbeid med FieldOpt i hans masteroppgave. Brukergrensesnittet er i tillegg utviklet på grunnlag av driverfilen og dens data som Baumann har opprettet.

I oppgaven presenterer vi verktøyene og hjelpemidlene vi har brukt, som blant annet programvare, filstruktur og biblitotek. I tillegg introduseres brukergrensesnitt og flere retningslinjer for design som er brukt under utviklingen. Dette blir fulgt av spesifikasjonen til systemet. Hva kreves av systemet? Kravspesifikasjon er en stor del av systemutvikling og vi har kommet frem til hva som ønskes og burde forventes av grensesnittet sammen med de nødvendige funksjonene. Deretter blir GUI-design og grensesnittelementer presentert. Basert på bl.a. inneholdet av disse kapitlene, har vi designet og implementert et grensesnitt. Resultatet blir presentert og deretter evaluert og sammenliknet med hvilke krav og retningslinjer vi satte for det grafiske brukergrensesnittet. Refleksjon på prosessen og noen løsninger er også med i denne rapporten.

Denne oppgaven har resultert i en løsning som har potensial til å bli “FieldOpt driver configuration interface”, men er foreløpig ikke et fullverdig, velfungerende og ferdigutviklet grensesnitt. Uansett mener vi at med dette resultatet har vi tilrettelagt og lagt et godt grunnlag for videre utvikling og ferdigstilling av grensesnittet, som skal hjelpe brukere å konfigurere FieldOpts driverfiler på en enkel og brukervennlig måte.

Preface

This thesis is the result of work performed during the semester of 2016. It is a part of the Master's degree in *Engineering and ICT* with specialization in *Integrated Operations in the Petroleum Industry*, at the Department of Petroleum Engineering & Applied Geophysics at the Norwegian University of Science and Technology(NTNU). This thesis work was under the supervision of Prof. Jon Kleppe, Postdoc. Mathias R. Bellout and PhD stud. Einar J. M. Baumann. The idea for the master's thesis was introduced by Bellout and Baumann, members of Petroleum Cybernetics Group(PCG). It is based on the recommendation of further work found in Einar Baumann's Master's thesis [1].

The work processes were carried out in close cooperation with PCG. The grahical user interface developed during this project builds upon the JSON formatted FieldOpt driver file and the FieldOpt Utilities library created by Einar Baumann in the beginning of his PhD at NTNU.

It is assumened that the reader has knowledge of graphical user interface, programming, and understanding of terms within this field. However, some terms are still explained or defined directly in the text.

Acknowledgements

I would like to take this opportunity to give a special thanks to both my co-supervisors: Postdoc. Mathias Rodriguez Bellout and PhD stud. Einar Johan M. Baumann for their patience and assistance throughout the project. Bellout's enthusiasm for this project motivated me through some difficult times. Without the guidence, and help sessions from Baumann I would probably not have made it. The team meetings we had were really valuable to me.

I would also like to give a big thank you to my responsible supervisor Prof. Jon Kleppe at NTNU for guiding me towards the Petroleum Cybernetics group, but also for his support during the project. He helped me through the challenges by giving good advise and conversations.

In addition, I would like to express how grateful I am for the friendships I have gotten during the years in Trondheim. Thank you to all class mates and exchange students. There are so many great people out there that helped and inspired me to keep on going. You know who you are.

Finally, I will thank my family, my mother and father for raising me with great values, guiding and supporting me throughout the years. It has lead me to become the person I am today. I will also thank my sister and brother for being who you are to me. Thanks to my family and friends for always being there when I need you. I could not have done this without you, and your support. You have all played a huge part of this thesis. I am grateful.

Contents

List of Figures

Chapter 1

Introduction

This thesis concerns the development of a graphical user interface(GUI) for the field of petroleum. We are mainly looking at the specification and development aspect of software engineering with some software validation during the development. The focus is on usability since user interface design is about predicting what users might need to do, but also ensure that the interface has elements that are easy to access, understand, and use to carry out those actions.

In this chapter, we present the goal and motivation for the project, in addition to some background information about the group that needs a new tool for their software in development, as well as the document structure of this thesis.

1.1 Problem Description – Goal and Motivation

As with other reservoir simulators, the main way of configuration is through driver files. This master's thesis idea originates from Baumann's master's thesis, from the *Conclusions & Recommendations for Further Work* chapter, Chapter 5 – the paragraph about Driver files [1]. He presents that the simulator's driver files are to be written manually using a text editor, but that it "feels" outdated. For further work he suggests to develop a graphical user interface as a better alternative, and that is what we intend to do here.

The goal of this project is to create and design an interface for generating and configuring driver files of JSON[1] format for a petroleum field optimization software (named FieldOpt[2]). Petroleum Cybernetics group(PCG)[3] wants a user-friendly graphical inter-

[1]*JSON* – See Section 2.3.1
[2]*FieldOpt* – See Section 1.2
[3]*PCG* – See Section 1.2

face that replaces an alternative of direct manipulation of a text file(driver file), i.e GUI replacing a text editor. PCG wants a solution that let users, novices to experts, easily interact through the GUI regardless of their skills (as long as they are identified as target audience). The GUI should be based on the Utilities library[4] and the driver file example provided in Appendix A.1. The generated driver file should represent a meaningful configuration of FieldOpt, including settings for model, selected simulator and optimizer. With great focus on usability, the GUI aims to be an integral part of PCG's main product – FieldOpt.

The motivation for this project lies within the improvement the solution can give for users – the intentional target audience: students, professors, and people in the research community at NTNU. Manually generating a driver file requires hours of hours of training and reading of massive manuals, and finding errors may often be very time-consuming. It will maybe take years to master the composition. Neither learnability, recognizability, nor accessibility are qualities that stand out from this manually way of creating a driver file. We want to increase the learning pace, and decrease the time it takes for the transition from being a novice user to an intermittent, and then to an expert. And we believe that this can be done by creating a GUI. There is not yet set any quantitative goal for this.

We want a logical solution which you can use and understand without reading a manual – the pieces should come together just by thinking and exploring the GUI. We want students and researchers to have the best alternative that motivates and encourage to take work a step further in detail. Maybe they can skip using months on uninteresting material/work by using FieldOpt and the driver GUI instead. The long term goal is for the software and GUI to be a tool that the users long to use again.

A challenge of designing is to create features that are intuitive to the user. Users should feel reassured about how the system reacts to certain user interaction – it should be predictable. To help with getting there, we need to have the right tools to reach the goal. This is a crucial step of the design process. Another challenge is a combination of personal qualification and time. During the project, the personal "knowlegde database" needs to be expanded. This means that you have to teach yourself what you are suppose to do, and do it at the same time, on restricted time. To be both efficient and effective can be a great challenge, but a curious and exited mind keeps it going. To balance the

[4]Utilities library – See Section 2.2

work load based on the set of knowledge that you posesses and are able to learn, in addition to plan according to time available, are going to be very challenging.

1.2 About PCG and FieldOpt

FieldOpt is an open-source software framework in development, written mainly in C++[5]. The development of FieldOpt is a collaboration between two departments – called NTNU Petroleum Cybernetics Group(PCG). PCG wants to promote common research activities between the Department of Petroleum Engineering and Applied Geophysics(IPT) and Department of Engineering Cybernetics(ITK). And that is exactly what FieldOpt aims to be – a common platform for MSc. and Ph.D. students to conduct research.

We want FieldOpt, the petroleum field development optimization framework, to be a user-friendly and trusted tool that contributes positively in uniting the academic field of optimization and petroleum engineering in research. With its already built foundation, FieldOpt will simplify and enhance the development process of optimization algorithms to solve petroleum problems. It should make prototyping of new algorithms more efficient, and make it easier to test new techniquies. In addition, it should be easy to configure, adapt and execute any petroleum case for optimization purposes [2].
The FieldOpt source code can be found in its entirety on GitHub
(https://github.com/PetroleumCyberneticsGroup/FieldOpt).

1.3 Document Structure

In the next chapter, Chapter 2, implementation tool and language, structure, and library are presented as the main utilities in this GUI development.

Further, Chapter 3 defines graphical user interface, and introduces theory and guidelines for designing the GUI.

Some theory and description of the GUI development process, the implementation process, and challenges is being presented in Chapter 4.

What is required of the graphical user interface? In Chapter 5, the system requirements are presented and divided into functional and non-functional requirements. In addition, the GUI's quality attributes are presented.

Requirements are followed by GUI design in Chapter 6 that presents different features and interface elements used to create the current solution. A user analysis that

[5]C++ – See sub-section 2.1.1

discusses and defines the target audience is also included.

The results and the product of this project are being presented in Chapter 7, followed by evaluation and reflections in Chapter 8.

Finally, the last chapter, Chapter 9, summarizes the project and presents recommendation for further development of the FieldOpt driver configuration interface.

Chapter 2

Utilities

This chapter will give a brief description of the implementation tool and framework named Qt, an short introduction of C++, and a presentation of the JSON standard that is chosen as the driver file format. The driver file file will contain data and information that should be represented in the GUI. A customized library (Utilities by Einar Baumann) that is needed in the GUI development to take care of the reading and writing interaction between the GUI and the JSON file, is also introduced here.

2.1 Qt Application Framework

The Qt Company[1] is the developer of Qt, a cross-platform framework with C++ class Qt libaries. It has developed an integrated development environment(IDE) for open source (and commercial) use, called Qt Creator. The cross-platform IDE is stocked with features for easily creating UIs and applications [4]. The choice of framework was predecided, it was one of the conditions of this thesis, as the group, PCG, had already developed parts of the FieldOpt and gotten familiar with Qt. The library that was going to be included and used as a way to connect the GUI with the driver file, was also created using Qt. In previous steps of the FieldOpt development, a decision of using Qt was made based on what ResOpt(the beginning of it all) was written in, Qt. Choosing Qt would therefore ease the reuse of this code. Another advantage of Qt is that it is very portable when it comes to platforms/operating systems. It supports Windows, Mac OS, and Linux. In addition, Qt provides all kinds of features perfect for creating a GUI through the Qt Designer. It is integrated into the Qt Creator, and it can be launched through the Design mode.

[1]The company is responsible for everything regarding the Qt software; product development, and commercial and open source licensing, which is done together with the Qt Project(the Qt developers) [3].

The Qt Designer is the Qt Tool for building graphical user interfaces with Qt Widgets. There can be composed and customized windows and dialogs, with/in simple steps [5]. And just as easily, there can be added elements onto them. By drag and drop from the side library, objects can be placed onto the ui file. There is a flust of useful objects for displaying data, taking input, or adjusting the layout. Some examples are: labels, spin box, combo box, line/text edit, push button, radio button and (list) widgets with built-in functionality. There also exist other objects very commonly used in, in this case, desktop applications.
Another convenient property of the Designer, is to use Qt's *signals and slots* mechanism. It is used in the programming code to assign behavior to the graphical elements [5]. It can be set up by right clicking the object and "go to slot" and the function will automatically be generated.

The Qt Creator also has a very convenient way to organize the project files. It creates a folder structure by filtering them based on the file type – the .h files goes into *Headers*, .cpp to *Sources*, .ui into *Forms*, and .qrc into *Resources*. In that way, it is easy to find the files you want to work on.

2.1.1 C++

As mentioned, Qt uses standard C++ language – the Qt libraries are of C++ class. C++ is a general purpose programming language created with intention to be "close to the machine"(handle machine problems in a simple, but efficient way), but also to be able to create direct and concisely solutions. It is a language designed to support a wide variety of uses [6]. As stated by Bjarne Stroustrup (the designer and original implementer) in *The C++ Programming Language* [7] book, C++ is well suited for resource-constrained applications found in software infrastructures. With its flexibility and generality, it seems to be suitable in many solutions.
With these statements, C++ covers all aspects of FieldOpt, from writing algorithms to designing GUIs.

2.2 Utilities/Settings Library

The GUI should be based on the JSON[2] standard made driver file, and use the Field-Opt Utilities library made by Baumann. Utilities library consist of the *Settings* class

[2]JSON – See Section 2.3.1

and related filehandling classes. The original files can be found in the FieldOpt repository on GitHub, by following this link: GitHub - FieldOpt Utilities library [8]. The *Settings* class consists of *Model*, *Simulator*, *Optimizer*, and *Settings*. The Settings subclass takes the path to a driver file (JSON formatted) as input. This is parsed to get the general runtime settings, but also to create separate objects containing the settings and parameters for the three other sub-classes. So, it is through the use of this library – the *Settings* class, that the GUI interacts with the (JSON) driver file. In order to make the changes that the user wants, some modifications were done to the Settings classes. Setters and some getters for different fields of interest were added. Some other changes were made to fields by removing the *const*[3] qualifier where needed. More modifications will be done in the future. The modified and used version is found at https://github.com/karolinr/FieldOpt/.../Driver-file-gui/Utilities/settings.

2.3 Driver Configuration File

As being mentioned by Baumann in the end of his thesis and what is well known for specialists in the field, the main way of configuring most reservoir simulators are through a driver file. This will also be the case for FieldOpt.

A 'driver' acts like a translator, and communicates the important configurations to the simulator like: what data and grid to use, and defining how to run simulations and where to store the results. The driver file is often written manually using a text-editor, and can be a quite heavvy job. It takes a lot of experience and a thick manual to conquer the list of keywords and commands, which can be a complex and time-consuming task, especially when we include error searching. Instead, we want the user to be able to create and edit driver files through a GUI. The driver file should be in a machine-readable format so the complexity of code in conjunction with the file and GUI will be reduced significantly. To fulfill the needs, PCG decided to choose JSON as the driver file format.

2.3.1 JSON

JavaScript Object Notation, more known as JSON, is a way to store information in a certain organized way – a structure. As it is easy for humans to read and write it, and for computers to parse and write it, it fulfills the needs of the driver files format that is desired for FieldOpt. The complexity of "read/write driver files" code will most likely be reduced which will make it easier to add new configuration options in the future. JSON

[3]*const* – constant, not being able to change the marked field/instance/pointer etc.

is therefore chosen as the driver files format. In addition, the text format is language independent, and many programmers will be familiar with the coding and notation as it is frequently used in the “C-family”[4] languages. JSON has two structures, “a collection of name/value pairs” and “an ordered list of values”, (for many of us) respectively called ‘object’, and ‘array’ [9].

The driver file has four objects: Global, Optimizer, Simulator, and Model. They are present as sub-objects, and all of them must be defined. Objects may have an object as a field. The file consists of settings and parameters for FieldOpt like: ‘name’ used for deriving the output file names, specific parameters and settings for the selected optimization algorithm, what type of simulator is going to be used, path to a complete driver file for the model (with properties to generate the grid files), and definition of the model: reservoir and wells with related variables. There are some optional fields, but many fields are mandatory to define.

2.3.2 Structure Example from Driver File

Listing 2.1 is a small example of how the JSON file is structured. This is an eight lines clipping (part of the Optimizer object) from the 163 code lines long example driver file, attached in Appendix A.1. *Optimizer* contains specific settings and parameters dependent on the optimization algorithm chosen. It consists of these fields: Type, Mode, Parameters, Objective, and Constraints[], but the two last objects are cut from this structure example.

Listing 2.1: Part of the Optimizer object from *driver.json*

```
    "Optimizer": {
        "Type": "Compass",
        "Mode": "Maximize",
        "Parameters": {
            "MaxEvaluations": 10,
            "InitialStepLength": 50.0,
            "MinimumStepLength": 1.0
        },
```

Type denotes what optimizer algorithm is to be used. Currently, the only implemented algorithm is ‘Compass’. The Optimizer’s Objective function *Mode* is specified to be either ‘Maximize’ or ‘Minimize’. The *Parameters* object has three fields for the current

[4]F.ex. C, C++, Java, C#, and Python

choice of Optimizer Type ‘Compass’. They are quite self-explanatory. ‘MaxEvaluation’ takes an integer (*int*). ‘InitialStepLength’ and ‘MinimumStepLength’ require *float* number [10]. As seen here, there can be objects within objects, which can be of great use when creating the desired structure.

Chapter 3

An Introduction to User Interfaces and Design Guidelines

"The golden rule of design: Don't do to others what others have done to you. Remember the things you don't like in software interfaces you use. Then make sure you don't do the same things to users of interfaces you design and develop."

–Tracy Leonard (1996)

This chapter introduces the definition of a graphical user interface along with its role. It also introduces some important guidelines for designing a good graphical user interface, gathered and developed by different people with a lot of experience from the field. In addition to the guidelines and formed golden rules presented in this chapter, the quote of Leonard is something to keep in mind when designing an interface. Every detail matters, and every choice is crucial to whether the interface's application will be a success or not. Therefore, in addition to personal experience, we have tried to follow and use what is presented here in the design process as much as possible.

3.1 Graphical User Interface Definition and Role

A user interface(UI) establishes a dialog between users and computers. As they (users and computers) are not "speaking" the same language, the interface works as an interpreter, translating input and output back and fourth.

A graphical user interface(GUI) is a type of interface that enables a person to communicate/interact with a computer through the use of icons, symbols, and other visual elements and/or devices, rather than allowing interaction only by using text via the command line or a text menu.

A GUI has elements such as windows, tabs, menu bar, buttons, icons,and cursor. The FieldOpt driver file GUI, as an application, has the specific components: application window, dialogs, message box, text-box, buttons, radio-button, check-box, and combo-box. The UI is part of the software and is designed such a way that it is expected to provide the user insight of the software. There are more about the elements and controls in Section 6.2.

A graphical UI is important because the alternative is not adequate especially for novice and intermittent users. It requires even more knowledge to use a command-line based interface. And it is harder and more time-consuming to cross the threshold to become familiar and feel confident with the UI – a lot steeper learning curve for the text-based interface. A GUI is visually intuitive which is a huge advantage in terms of the learnability aspect of usability. The learnability plays a huge part to what defines a user-friendly interface, as is mentioned in Chapter 5, in Section 5.3.1 about the usability requirement.

3.2 Design Guidelines

Designing an interface is not an exact science because applications have so many varieties of amongst other: target audience, objectives, in addition to the development concerning different platforms. This makes it difficult to form a perfect recipe and solution for creating a perfect user interface. There may never be a definitive, but there exist many guidelines based on a lot of experience. Looking at them, there are similarities, and some recur more often than others. In addition, these rules are basically the same regardless of the interface and intended platform or device. In this section, we present guidelines and some design mistakes presented by Nielsen (and/or the NN group), Shneiderman et al., and Theo Mandel. They agree on many of the core guidelines, but have a few different rules. We have picked out those (guidelines) that seem to be relevant to this GUI.

3.2.1 Navigation

According to chapter two in *Designing the User Interface* [11], there need to be established some guidelines to make navigation as smooth as possible. The relevant guidelines for the FieldOpt GUI are considered. They are: to standarize task sequence, use unique and descriptive headings, and to use radio buttons for mutually exclusive choices[1]

3.2.2 Common Golden Rules of User Interface Design

In this sub-section, we have summed up the common guidelines among the mentioned authors (Jakob Nielsen and Schneiderman et al.). There seems to be a huge agreement regarding these “rules”:

Consistency is frequently mentioned. According to Schneiderman et al. and Nielsen, one should strive for consistency. Why is that? Consistency in the user interface is critical to efficient learning and use of the system. The system should be consistant in how the interface functions and reacts, but also in every design and layout element. It should exist in terminology, abbreviations, formats, icons, color, and capitalization. If there is consistency, the interface requires less (memory capacity) from the user throughout the system. For example, when the user has learned how to navigate in one tab/part of the system, it will automatically understand the next. The user do not need to turn its head around to understand, as it is already “stored”. (Some of these guidelines are somehow connected e.g. this is also a guideline that leads to “reduced memory load”.)

Standards have basically the same function. Users have become familiar and used to elements acting in a certain way through other interfaces over years of experience. On a daily/regular basis, and on different platforms, users deal with interfaces, so it is important to be consistant and keep the known elements and actions as they usually act/are.

Consistency seems to lead to easier learning because it will lead to for example fewer confusions and less errors than when inconsistancy occurs. Inconsistant interfaces may cause users to make errors, therefore we want similar procedures to be designed the same way. Also, it is important to use the same phrasing and skip synonyms. It can be confusing when different words are used to explain/show the same thing. The user may overthink it and end up using much time on the issue: “Do they mean the same thing? Does the same thing happen when I push the button?”, and in worst cases it leads to errors. These kinds of design mistakes are very unecessary.

Feedback should be provided by the system to keep the users informed about processes, progress etc. There should be customized feedbacks for different type of ac-

[1]This is backed up by Nielsen in the article *Check boxes vs. Radio buttons* [12].

tions; modest response for frequent and minor actions, while responses for infrequent and major actions should be of more substance. By making the system status visible for users they will get in more control, exactly what Mandel wants to achieve with his principles.

Simple error handling is what the system should offer to prevent the user from making serious errors. The error should be detected (before executed), and messages should indicate the problem in plain language. The messages should suggest a solution and recovery. Let the system be helpful. According to Mandel this will also help the users to be in control.

User control and freedom are topics that is about giving the user permit to easily revert an action. Schneiderman et al. claims that this feature relieves anxiety, because the user knows that he can make an error without bigger consequences – it can be undone. In this way, users will grow confidence and explore unknown functions. Undo and redo functions or the opportunity to make changes are measures to take.

Reduce short-term memory load Because of human's limited information processing in short-term memory, it should be required of the GUI designs to adjust to that area. A measure is to use recognition rather than recall. The user should not need to remember information from one part of the program to another. By keeping the layout simple, reducing/combining multiple pages into something solid, making objects, actions, and options visible, the user's memory load will be minimized.

Confirmation of the rules Jesse James Garrett is another author that builds up on that these are the guidelines that should be followed, which he does in his book: Elements of User Experience: User-Centered Design for the Web and Beyond (2nd Edition). He again confirms what we think of as the main guidlines for desiging interfaces.

3.2.3 Golden Rules of User Interface Design

The following rules are some of the golden rules for GUI design according to Shneiderman, Plaisant, Cohen, and Jacobs in their book, *Designing the User Interface* [11]. The most of them are gathered in the common guidelines section, but the authors had different focus on some of the main rules, so they are mentioned here.

Enable frequent users to use shortcuts As this system is not the biggest and most complex, we have not seen any need for the use of this guideline yet. Thus, we have thought of personal customization and therefore some hidden commands, but not especially due to frequent users, see Section 6.1.1.

3.2.4 Usability Heuristics for User Interface Design

The following list is some of Jakob Nielsen's 10 general principles for interaction design [13]. Even though according to Nielsen, "They are called "heuristics" because they are broad rules of thumb and not specific usability guidelines", we have used the term "guideline" for "heuristic" as we think of it as equals rather than strict "rule". Some of his principles are gathered under the common guidelines section, Section 3.2.2.

Match between system and the real world One should try to follow the real-world's "code". The system should communicate in the users' language and use familiar concepts and standards. This includes making information appear in a natural and logical order.
Error prevention "Even better than good error messages is a careful design which prevents a problem from occurring in the first place." [13, Nielsen, 1995]. This is something we really want to fulfill to the fullest. The design solutions should not let users be able to do anything wrong. If there have to exist such error-prone parts, one should try to reduce the risk of error as much as possible with thorougly descriptions, and one can present a confirmation option before they commit the action. But the best is to eliminate error-prone conditions.
Flexibility and efficienct of use will allow users to tailor frequent actions. This may speed the interaction for the expert, and let the user, inexperienced or experienced, be in control.
Aesthetic and minimalist design One must evaluate what kind of information the dialogs should be filled with. They should not contain information without relevance or something that is barely in use. All information (whether it's relevant or not) are competing for users' attention. We do not want users to lose focus from the important.
Help and documentation are something we want to avoid needing, but it may be necessary. Nielsen provides some tips e.g. this kind of information should be easy to

search for, the focus should be on user's task, and there should be a short and consise list of steps to follow.

3.2.5 Theo Mandel's

We have included these principles by Mandel because they are short and consise, and can be great to look-up during the GUI development. See chapter five in Mandel's paper [14] about principle that helps to place users in control, let the GUI:

1. Be modeless – use modes judiciously
2. Be flexible – allow users to use either the keyboard or mouse
3. Be interruptible – allow users to change focus
4. Be helpful – display descriptive messages and text
5. Be forgiving – provide immediate and reversible actions, and feedback
6. Be navigable – provide meaningful paths and exits
7. Be accessible – accommodate users with different skill levels
8. Be facilitative – Make the user interface transparent
9. Be adjustable – allow users to customize the interface (preferences)
10. Be interactive – allow users to directly manipulate interface objects.

3.2.6 Ten Application Design Mistakes

The following is a list of common design mistakes that we want to avoid. The ten mistakes are presented in a report by Whitenton et al. from 2012 [15]. They are presented as common mistakes based on many candidates, not from the 13 winners chosen for the Application Design Showcase report, but from submissions that the NN group judged. Some of these had rigid design and/or usability issues. Back to the quote of Leonard in the epigraph, we learn from more than what guidelines to follow. We can also learn from other's mistakes.

1. Steps in a task or process presented out of order on a page or screen
2. Missing simple paths
3. Invisible controls and cryptic icons
4. Hideous forms
5. Unclear differences when comparing choices

6. Indistinguashable current selection status
7. Branding feature or menu commands
8. Only the promise of content
9. Views and modes that trap people
10. Bad dialog.

Chapter 4

GUI Development & Implementation Process

In this chapter, the GUI development process is defined and presented including further description of the project work process. In addition, we shed some light on challenges and risk related to this project.

4.1 The Development Process

In this thesis' user interface development process, the iterative development model is used as a guide. According to *Tutorial Points*[1] a model for GUI implementation should consist of the following steps in a cycle [16].

- GUI Requirement Specification
- GUI User Analysis
- GUI Task Analysis
- GUI Design & Implementation
- GUI Testing

The design is developed iteratively – steps are repeated. Formality and detail is added during the development with a continuos correction of earlier solutions.

GUI Requirement Gathering The functionality of the GUI is defined here. The designers may want to have a list of all functional and non-functional requirements.

[1]A company with the mission of delivering "Simply Easy Learning" online.

This can be taken from user/customer demands. If they have any exisiting software solution, requirements (and "don'ts") can be drawn from this. All kinds of constraints are listed here.

User Analysis The designer needs to do a careful user analysis; defining and deciding whom the system will be of use for. This is an important part of the GUI development since the target audience are the ones the design details change according to. Knowledge and skills of the user are keys to what the GUI should look like.

Task Analysis Tasks should represent what the system should be able to do. After the designer analyzes and presents the tasks, the identified tasks are directly being converted to goals in the GUI presentation. The user should be able to perform those tasks through the interface. To simplify, (major) tasks may be divided into smaller sub-tasks. The flow of information between the sub-tasks often mirror the flow of GUI contents.

GUI Design and Implementation After carrying out the previous steps, the developer should have an overview and information about requirements, user environment, and goals for the GUI. The next step is to design and implement the GUI using the chosen software and/or programming language. It is then self-tested by the developers.

Testing GUI testing can be done in various ways. Continously desk-checking during the implementation, debugging, involvement of users for testing and validation, and release of beta version are a few of them. Testing may include learnability, operability, effectiveness, efficency, satisfaction etc.

4.2 The Work Process

4.2.1 Work Flow

The Petroleum Cybernetics group started with some thoughts and ideas of what the coming GUI could be. We had meetings where we discussed what the solution should do and be, and with the use of a whiteboard we started thinking of a design. The members of PCG can also be considered as potential users, so getting feedback from them could be an advantage in the early process. At the most, we had weekly meetings where we reviewed the work that was done, and we discussed the next steps and checked the progress of the work. From the beginning, we focused on the whole package with functional elements rather than just the interface design. We wanted to be able to run and test the application during the process. By openly sharing our thoughts, we are more sure of having the same understanding of the goal and vision of FieldOpt and of this driver interface. Bellout took control and clearified, as Morten T. Hansen in chapter four of "Collaboration", about unifying goal, teamwork, and inspiration and delegation. [17]

The weekly process was generally like this:

1. Feedback session: Consult with the team (co-supervisors). Go through the current solution. Reach an agreement on further work.
2. Do as discussed. Process, change, remove, and improve.
3. Expand to new solutions.
4. Repeat

Process when coding and working in Qt

1. Think of a solution to implement the idea.
2. Implement it in Qt.
3. Try to compile, and fix errors until it compiles.
4. Test the new solution in every way. And then ask some questions:
    - Does it handle all scenarios?
    - Is it intuitive and logical for users?
    - Is there no risk of user errors, or as low risk as possible?
    - Is the consistency kept throughout the design?

    Try to avoid any of the ten design mistakes presented in the theory – Section 3.2.6.
5. If good enough, move on to the next idea. If not, improve the solution until satisfied.

The work process/steps are repeated after coding small parts. These are small cycles within the development that have many iterations. [18]

4.2.2 Challenges

Technical competence There are risks and challenges related to a limited technical competence within the field of graphical user interfaces, software development, and programming. Without the expert developer with years and years of experience, the risk of a slow and time-consuming development are quite high. It can be very hard to plan and do time estimations on problem solving and programming. Unexpected problems will pop up throughout the project period. The problems may be small and easy to solve, or complex and time costly. In addition, it is very likely that lots of time will be spent on getting to know all the utilities, as well as learning GUI theory and how to practice it to be able to carry out this project.

Major and minor design changes

During the GUI implementation the JSON driver file structure/setup changes as well, as it is also under development. New fields or parameters are introduced, and others are removed. As a result, the GUI's layout will need to undergo changes in addition to the "normal" construction based on the initial driver file structure. Meaning there are a parallel development process that has to be taken into account – another factor in the GUI's process. The already very dynamic process can and will be less predictable, especially considering time. It can be hard to know how time-consuming it will be.

As mentioned above, design changes will happen. After starting discussing design in front of the whiteboard and drawing sketches, we are jumping right to the implementation part. We are planning to design further and evaluate during the implementation part – learning by doing. There exists a risk that this process may end up with several design changes. Minor ones does not need to affect the progress, but bigger changes in this phase will often be a very costly affair in terms of time. If we realize the current design cannot be implemented or there exist a much better solution, there may be a great challenge ahead.

4.2.3 Testing

During the implementation of the interface application, the code and program are tested continuously. After a function is written the code is checked for spelling mistakes or/and syntax errors, then ran to see if it compiles. An attempt to compile will reveal any typographical error or incorrect use of the programming language. Qt Creator will give feedback, messages, and hints about what seems to be the problem. The debugging starts, where we try to locate, detect, and remove all errors in the computer application. We want to get rid of every logic error, "bug", that is caused by incorrect use of control structures. We will also ask questions like: Does the GUI design respond like it should do? Is the positioning of elements logic? Is the naming right? Are elements intuitive? During the process, we may do some beta testing with real data (e.g. importing a .json driver file). For further testing, at a later stage when more of the functionality is in order, we can let users test the interface, get feedback, evaluate, and continue developing.

Chapter 5

System Requirements

This chapter contains system functionality, in addition to its constraints, the usability and portability quality attributes are presented. *Software Engineering* by Ian Sommerville [19] is used for guidance in most of this chapter.

A part of software engineering, requirement engineering, is to think of what is required of a system to be a good system. The system should deliver the required functionalities, but also in performance to the user, it should be maintainable, dependable, and usable. In this case, the system is defined as the GUI. “System” and “GUI” will be used interchangeably.

GUI specification is, in this case, another term for the process of understanding and defining what services(functionalities) that should be required of the system, and identifying the constraints on the system’s operation and development. In addition, the requirement engineering process aims to specify a system that is satisfying stakeholders/user requirements. In this engineering process, there is no feasibility study as there is no budget to think of and the developer(student) works for free. PCG can be looked at as the customer, and we have roles as (user), consultant, developer, and designer. Some requirements have already been decided for in other parts of the FieldOpt development, and those processes have led to some predetermined constraints for the driver GUI.

5.1 Functional Requirements

A functional requirement states what service the system should provide, in addition to how the system should behave to certain inputs and/or situations. What the system

should not do, may also be stated.
Each functional requirement is given a specific priority, categorized into three levels: Priority high(H), medium(M), or low(L). Priority high means that the requirement is necessary for the system to function, i.e. mandatory. Medium priority is for desirable requirements. Low priority is assigned for a requirement that is optional – to fine tune the system.

Nr.	Short label	Description	Priority
1	Read JSON file	GUI should be able to, through the use of the Utilities libary, read a JSON file.	H
2	Write JSON file	GUI should be able to, through the use of the Utilities library, write to a JSON file.	H
3	Data presentation	GUI should be able to present the data stored in the driver file in a human readable way.	H
4	User manipulation of data	GUI should enable the user to manipulate and change information/data.	H
5	Import from JSON file	GUI should enable the user to import data from an existing JSON file.	H
6	Save	GUI should enable the user to save the updated information (to that JSON file).	H
7	Save as	GUI should enable the user to save to a new JSON file (save as...)	H
8	Meaningful driver file	GUI should generate a meaningful configuration of FieldOpt as a driver file.	H
9	Valid input	GUI should only accept valid input from the user.	H
10	Learnability	GUI should be simple enough for users to easily use it without a manual.	H
11	Sense of progress	GUI should enable the user to get a sense of progress.	M
12	Navigation	GUI should enable the user to easily navigate through the menus, options, and settings.	H
13	Prevent mistakes/ avoid error	GUI should be designed in a way to prevent the user from doing typing mistakes. Create solutions for avoiding errors.	H
14	Diversity in users	GUI should facilitate options and solutions for the diversity of users.	L

Table 5.1: GUI Requirements – What the system should be able to do and provide.

These are requirements that we want the GUI to fulfill, then the user is able to perform certain tasks and the GUI is responding in a certain way.

Some kinds of GUI requirements can also be found in the chapter about the designing guidelines. Even though they are not specified in this table nor defined as rules, they may be as important to follow.

5.2 Non-Functional Requirements

Non-functional requirements apply mostly to the overall system. These result in constraints on the product being developed and services provided by the system, development process, and may also include time constraints and constraints in consequences of standards.

Non-functional requirements may be divided into three categories with even more subcategories, according to Figure 4.3 in *Software Engineering*. Considering our system, we have chosen to not include and consider external(ethical, regulatory, and legislative) requirements. Some organizational(environmental, operational, and development) requirements are thought of and presented here. We are also focusing on product requirements which constrain the behavior of the system. The product requirements are categorized into usability, efficiency, dependability, and security [20].

There is no focus on efficiency in terms of performance. Other parts of FieldOpt will and have considered this. Security requirements for FieldOpt have been considered, but at this stage FieldOpt seems to not need any special measures, also in regards to the GUI. As the system is free and open-source and available on GitHub, it is supposed to be open and accessible. It is therefore little need for protection, even though the openness comes with some vulnerabilities. At this stage, we cannot see any security threats for the system. Summed up, we are mainly looking at the usability sub-category, and some organizational requirements.

In this section, the constraints will be described. The constraints are due to the choice of components, technology, programming language, framework, layout language, and data structure. Some of the components are thoroughly described in Chapter 2.

5.2.1 Constraints

Our GUI will be created and implemented using the Qt Framework and Designer – a design decision made by the Petroleum Cybernetics Group. It constrains the system to be inside the frames of Qt with its graphical library. This was chosen so the code could be compatable with the rest of FieldOpt, as the most of the pre-existing code is written in C++. Therefore the programming language will be C++. And by using Qt, the GUI will also be available and recognizable for all platforms as it adapts and uses the different operating systems' layout. See Section 2.1 in Chapter 2 for a more detailed description of what Qt is.

Another decision that was already made was to use the Utilities/Settings library made by Einar Baumenn, see Section 2.2. The library is not yet complete, and it will be developed alongside the FieldOpt driver file GUI. Since the library is mostly constrained by C++ and Qt, and it is dynamic, it does not need to be seen as an additional constrain. The Utilities was already written to handle reading from a JSON file.

JSON is the data structure chosen for the driver/configuration file. The standard may come with some restriction to what can be done. See Section 2.3.1 for more information about that utility.

In terms of physical material, hardware should not be an issue. The GUI itself will not require any kind of special hardware. Other parts of FieldOpt may have requirements about the hardware, especially considering the simulations. The system is to be developed for desktops, so it is intentionally not created nor compatible for for example mobile devices. Neither security nor safety will be issues here, and the application will not be connected to a network.

A time constraint is given by NTNU. The project needs to be completed during a specific time period. Due to the limited time some GUI functionalities have not been implemented. If these assignments were handled by an expert in this field, time would not have been such a constraint for the progress of the GUI. Others with more competence would probably not need to use time to learn how to master Qt or use as much time to solve programming problems and come up with solutions.

5.3 Quality Attributes

"A quality attribute is a measurable or testable property of a system that is used to indicate how well the system satifies the needs of its stakeholders."

–Bass et al. [21]

Since we want our product to be of quality, this is something we need to focus on during the development. The product quality model is an important part of how to do quality evaluation of a system. It determines which quality characteristics will be considered during an evaluation of the properties of a software product. This is an ISO/IEC standard, called ISO/IEC FCD[1] 25010 (new in 2011). ISO, short for International Organization for Standardization, forms together with International Electrotechnical Commission(IEC) joint committees. They are representatives from various national standards organizations that promotes standards mainly to enhance and increase the number of products/services that are safe, reliable and of good quality [22] [23].

When deciding what qualities to choose, it is important to think of what you want the system to be focusing on. There are eight quality characteristics defined by the product quality model. We have chosen to focus on two attributes in the development. Implementation of the quality attributes will be required for the system. The usability and portability attributes are presented in the following sub-sections.

5.3.1 Usability

Probably the most important feature and attribute for a user interface is for it to be user-friendly, and naturally it applies to this GUI as well. The quality of usability is a requirement that needs to be fulfilled and reached to the fullest to deliver a quality GUI. Usability is defined as "The degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use." on page 193 in *Software Architecture in Pratice* [24], which is taken from the ISO standard [25].
According to the ISO/IEC FCD 25010 product quality standard, usability characteristic is composed of appropriate recognizability, learnability, operability, user interface aesthetics, and accessibility. These qualities are somehow all mentioned in the introduction chapter to user interfaces and design guidelines – Chapter 3. The guidelines and stan-

[1]Final Committee Draft(FCD).

dard are agreeing on what should be required of a user interface. We want our GUI to be lucid and recognizable. It should be easy to achieve goals of learning in an effective and efficient matter. The user should feel satisfied and pleased during and after using the interface. It should be designed in a way that makes it easy to operate and navigate. We want to understand the user and his behavior to be able to predict what and how he will use the system. Then the system can be built up and designed in a way that prevent errors and mistakes. The GUI should also be accessible for users with every kind of skillset and range of knowledge, within the target audience.

5.3.2 Portability

As mentioned before, we want FieldOpt to be used independently of operating system and available for different platforms. This requirement can be fulfilled by using a cross-platform framework and a GUI toolkit, then the code will be universal to all compilers. According to the ISO/IEC FCD 25010 product quality standard, the portability quality is composed of adaptability, installability, and replaceability [26]. We especially had the adaptability part of portability in mind when we were thinking about what kind of qualities FieldOpt and the GUI should have.
It is wanted for the GUI to replace direct textmanipulation of a configuration/driver file. Later, the GUI could be evaluated on the degree to which the GUI could replace the current or another existing solution for the same purpose in the same environment – evaluate the replaceability. In addition, an evaluation of how the system fulfills the requirements to be adaptable can be carried out.

Chapter 6

GUI Design

This chapter describes GUI design elements used in our system with application examples, and how they shall be used. They are tools in the process of meeting the requirements and following the selected guidelines. The first section defines the target audience of the system and gives an example from the system of how it can be customized. As the design details change according to the knowledge and competency level of the user, doing a user analysis is important.

6.1 User Analysis – Target audience

Understanding users is key to designing a customized UI. Before designing and implementing the GUI, we need to understand the users of the system, as the design depend heavily on who they are. It is about anticipating user behavior and ensuring that the solution has elements that are accessible, understandable, and that facilitate their wished actions.

As FieldOpt aims to be a tool for students, professors and others in the research community at the university (see section 1.2), so will the target audience for this GUI be. Users will mainly be M.Sc and PhD students within the field of petroleum or cybernetics that possess the corresponding knowledge and skillset. The field of FieldOpt will determine whom the user of this driver GUI will be. Even though people are different and may possess f.ex. different skills in computer application, we have a somewhat clear idea of what kind of diversity we have to deal with. Education level, motivation, goals, and skills they aquire will mostly be the same for every user by neglecting the exceptions. The users will be of both genders. The students have graduated from high school and have managed to get into NTNU by fulfilling NTNU's admission requirements and pass a certain grade limit, or they have graduated from other universities. The age of our users could range from 20 to close to 70, depending on the professors and if and when someone decide to take a PhD. In that way we somehow know what they are capable of. They are used to aquire new knowledge and have typically had experience with many different types of interfaces, but their skills is as diverse as the user group. Therefore there will be novice, intermittent, and experts amongst them. One of our goals is that people should be able to make a fast transition just by exploring the application.

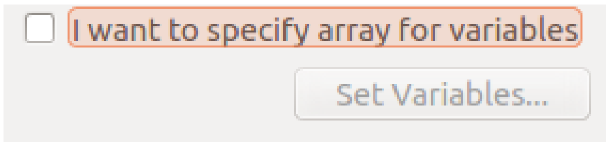
6.1.1 Customized Design

A good GUI should let the frequent users (soon to-be experts) tailor actions and use shortcuts (see Section 3.2). This is about the flexibility of the interface – a customized design. In today's solution, there are options that all categories of users may choose to use or not. With the use of a check-box it implies that the feature is optional to use. The option has different purposes; such as giving the user more parameters to specify, which can be seen as more advanced.
Unsecure or novice users will probably not choose to specify settings if they are unsure of what to enter. They have the option of skipping the step by keeping the check-box unchecked, or leaving it like the imported driver file may have specified it.

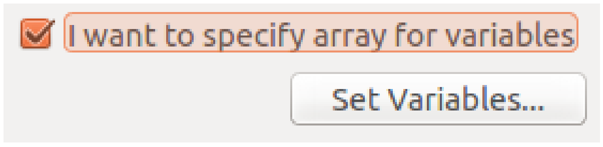
The other purpose is regardless of current type of user. There are times these parameters do not need to be specified because the experiment and simulation have to do with other parts of the model or the focus is on other details. Then the option to skip certain specifications can be of good use, and may be time saving and "mind" saving.

Except from the check-box expansion of settings, there are no differences in the GUI design for different users. We do not recognize needs for big differences in accessability/availability, as we do not want the GUI to be that complicated. We want to keep it simple enough for all type of users.

Figure 6.1 is a clipping and example of how our GUI have features customized for users. As explained above this feature has the purpose of giving an extra option to specify some variables, in addition to be a way to please all types of users and their need.



(a) Unchecked check-box – I do not want to specify variables, so the *Set Variables...*-button and dialog is unavailable.



(b) Checked check-box – I want to specify variables, so the *Set Variables...*-button and dialog is available.

Figure 6.1: Screenshot of a part of the *Model - Well* dialog – clipping of the *I want to specify array for variables* check-box. This is an example of a feature customized for user. The user can choose to specify array for variables or not by checking (or unchecking) the check box. The button change from an unavailable state to an available state, when checked. By clicking the button a new dialog opens with variables settings.

Another solution could be to make the user specify and choose what type of user they are, and then restricting their options. But if there are any uncertainties of what will be restricted, the user may be unsure of what to choose, and this may lead to unwanted confusion. Anyhow, we do not know what the different user would want to specify. It may not depend on how they determine themselves. Our solution keeps the options open. All type of users will have the the possibility to specify what they want to specify (of the voluntary options), but they are not forced to use them. Hopefully the system will contain solutions that will not frustrate the advanced users, nor the novice ones. We do not want any confused users on any part of the scale.

6.2 Interface Elements

Qt provides the familiar layout and element design to satisfy users. We use standard elements from the Qt libraries found in the 'Widget Box'. These elements have a certain standard look: a special form, function and style. The user should at ones be able to recognize e.g. a button and predict its function. If we have used the elements in the right way, the user should be able to predict actions allowed just by looking at the design.

By having no vague or misleading options that are impossible for the average user to understand, and as Jakob Nielsen says: by employing the design elements correctly enchance user's sense of mastery over technology [12], we may be able to satisfy the diversity of users.

When adopting these elements we have tried to keep in mind the guidelines presented in Chapter 3. How can we use these components to convey what we want from the user? How do we present what is accepted and what type of input we require? We need to use the available components to follow the guidelines.

The interface elements help us to direct, guide, and lead the user through the application, so they can execute actions, and enter correct/valid input, in order for us to fullfill the users' wish, and in the end getting the result of a configured JSON driver file.

6.2.1 Input Controls (Widgets and Buttons)

Through the use of these elements we get to retrieve user input, so we have enough information to generate a driver file for FieldOpt, and later execute a simulation, so the user can get their results.

As other elements, buttons and widgets should also have a familiar and clean design so it is easily recognizable as the element that it is. They should be describtive, and it should be possible to predict the behavior and action of the system after clicking/selecting the element.

For **push buttons**, the text is set as short and consise as possible, or the text is replaced by a descriptive icon as in Figure 6.2. The buttons change state when the pointer is above the button area, giving a hint that it is active and ready to be clicked. This can be seen by the shining or highlighted layout. Mostly, this applies to the other elements as well. By clicking the button, we expect the system to respond with an action. But some buttons express that there is no immediate action after, but that there is further choices beneath or a need for additional information. This is expressed through the use of an ellipsis, a set of three periods, in the descriptive text. This is illustrated by the examples from the GUI in Figure 6.3. It indicates that you can explore the software without any fear of something unforeseen happening.

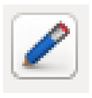


Figure 6.2: Screenshot of an edit button with an icon as the descriptive information. A pencil is a well known symbol for edit.

(a) A button with the text: "Browse..." used in the Global, Model, and Simulator tab.

(b) The button with the text: "Set Parameters..." used in the Optimizer tab.

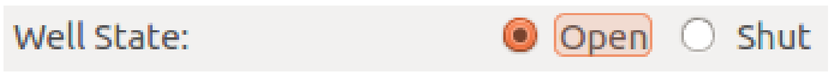
Figure 6.3: Screenshots of push buttons with descriptive text and an ellipsis; telling the user that the response of the click is not an immediate action. A new dialog will appear – another step where additional information is needed.

By clicking the button in 6.3a (in the application) you get into a browsing file dialog where you may choose a path to a predecided type of file, and you may search through your computer system directories for the file (or directory) you want to choose. Click 'Open' to take the further step to choose it.

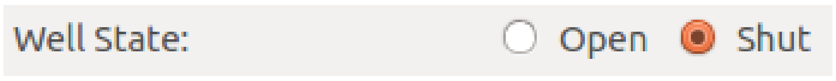
By clicking the button, 6.3b, you open a dialog where you need to set additional parameters, and then click 'Ok' to accept the new parameters.

Radio buttons are meant for mutually exclusive choices. These options cannot be chosen, be true, or exist at the same time. One option makes the other(s) unavailable for that state. The alternatives cannot be chosen more than one at a time, and one needs to be chosen. Not only does the radio buttons convey that these alternatives can only be used separately, but it also secures and prevents users from making a mistake (misunderstanding or mistyping) regarding the choice.
As the Figure 6.4 illustrate, radio buttons present the options openly – all options are permanently visible without forcing the user to click anything (to take any action). The user may easily/faster figure out which option applies to them. This has a lower cognitive load[1] compared to the design alternative *combo box* according to the Nielsen Norman Group [12]. A well state cannot be something other than open or shut, and therefore there is no question about preparing the design for any additional option or

[1]Cognitive load – "the amount of mental resources that is required to operate the system" [27]



(a) Well state “Open”.



(b) Well state “Shut”.

Figure 6.4: Screenshots from the model well controls dialog, where the well has a state of either open or shut, here represented by radio buttons. The subfigures 6.4a and 6.4b illustrates the two options.

change in further development. If that is the case, or there is limited space, a combo box may be the solution. The combo box can present more options in a tidier design, and can be more convenient to use as a solution when the alternatives are many. But unlike radio buttons, it does not show all the options without the user clicking the box, as you can see in Figure 6.5. By clicking the box, the menu rolls out, as seen in Figure 6.6. In other words, the options stays hidden until the user clicks it/takes an action.

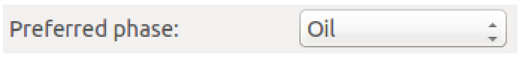


Figure 6.5: This is a screenshot from the Model’s well dialog. Only one option can be selected for that field. According to this state, the preferred phase of the current well is oil.

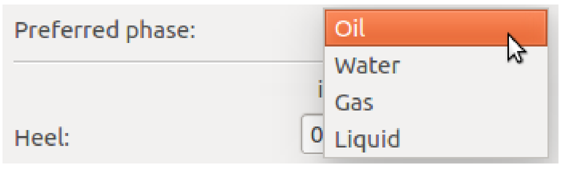


Figure 6.6: An example of an open combo box with all options visible. The preferred phase of this well can be the optional ‘Oil’, ‘Water’, ‘Gas’, or ‘Liquid’. This is a screenshot from the Model’s well dialog.

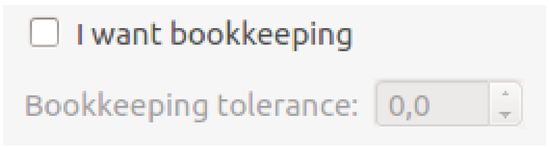
A drop down menu, or here referred to as a **combo box**, allows the user to select a single option out of (usually) a large number of items. As mentioned above, radio buttons may serve a similar need. Limited space usually forces the use of combo boxes. In the today’s solution you may argue that many of the combo boxes should be replaced by radio buttons, but this solution is chosen due to further development. We have focus on portability and the possibility to modify the application in the future. An expansion with more options (e.g. other types and modes) will be much easier with an existing combo-box than by adding additional radio buttons and demanding more space on the screen. Some examples are the Optimizer’s type and mode field that currently have only one and two options. We have chosen to automatically select an option for all combo box fields. We do not require the user to manually select it, especially not when there is only one option to choose from. We do not want to require the user to take any unnecessary actions that would not give you any additional information or make any difference. If we had not, that may have lead to frustrated users in the long run.

List boxes are boxes that can contain list of items. In the system, we use list boxes not as a list where you have to choose a specific element as an input, but as a way to organize components that have more information related to them. The user is able to add and remove items from the list, and set/change additional item information. The list is always visible in the specific dialog. Because of limited space, the box is set to a restricted size. When the list is filled with more items than the list can view, the box automatically gains a vertical scroll bar, making it possible to scroll through the whole list of items. The user may also maximize the dialog to view more items at a time. The list is designed to expand vertically. Single-selection list boxes are used since we are currenctly allowing the user to manipulate only one item's data at a time.

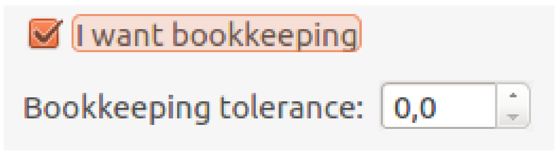
Generally for the GUI design, when the Settings class or the JSON driver file expect an array of a specific field, we have decided to use list boxes to represent the array. Related fields are represented by GUI elements beside the list, and are connected to items in the list. An item represent an array component. An example can be seen in the Optimizer's Objective dialog, Figure 7.14. The other dialogs with the same basis have been structured the same.

Figure 7.15 is a screenshot of the Optimizer's Constraints dialog. By selecting an item in the list box, the corresponding values for the related fields will show on the right. These can be changed dynamically. The idea is for them to be temporary saved after selecting other items, or clicking 'Ok'/accept. It will not be permanently saved until the user clicks 'Save' or 'Save as...' or use the shortkey 'Ctrl' + 's'. That functionality is not yet implemented. As mentioned, this solution has been implemented and used throughout the GUI, keeping it consistent.

A **check box** is used for a single option. The user switches through two different states, turning the option on or off. This is used when we want a typical boolean answer from the user. In regard to design, it is important to use this only when the reaction is clear for the user; what happens if it is checked, what happens if it is not? The user should feel safe to take an action. We do not want to make the 5th design mistake from Subsection 3.2.6. Figure 6.7 is an example of how the check box can be used and are used in this GUI. You either want to set a bookkeeping tolerance or you do not want bookkeeping. These are the two mutually exclusive options. The box can be toggled on and off. By having the ‘bookkeeping tolerance’ field visible at all time and using the enabling/disabling feature, it is easier for the user to understand what the consequences of checking/unchecking the check box are. Another quite similar example from the GUI is the Figure 6.1.



(a) Screenshot of a the unchecked bookkeeping check box.



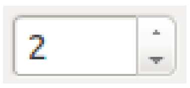
(b) Screenshot of the checked bookkeeping check box.

Figure 6.7: Screenshot from the Global tab, demonstrating how bookkeeping is an option to set or not. The subfigures a and b are a demonstration of what it looks like unchecked vs. checked; disabling and enabling setting the bookkeeping tolerance field.

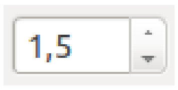
Line edit is a component where the user can enter text input, graphically extending in only one line. We try to set the graphical size of the element so it matches the reality, and we try to predict the length of the input. From there we set the horizontal length as practical as possible. The width (vertical length) is set appropriate to font size 27. If we had needed more text entered, the **text edit** would have been used. That is a text box of a chosen size with more lines (you could write paragraphs). A read only text edit were used for an informative purpose in the Simulator BASH Commands, see Figure 7.12.

In this solution, we have been forced to use the line edit as a way to require input when we cannot decide or predict the user input alternatives – and we are depending on individual input. When using edit elements in our design we may open up for user error, because we cannot control exactly what the input might be. Somehow we can keep control by allowing certain characters, and restricting length directly through the element. We can also set up controls and checks of content, and give feedback of if the entered input is accepted or not. But if we do not know what we are looking for, like the case is in the BASH dialog (read about it in Section 7.2.4, we cannot implement that solution. Anyway, no such checks have been implemented yet, but we try to guide the user as much as possible by providing information.

Spin box (Figure 6.8a) and **double spin box** (Figure 6.8b) are used to restrict input to be of type integer or float. When we demand numbers(integer or float) as input, we can use these specific elements for that purpose. A certain range, with minimum and maximum values, is set to avoid meaningless input. This is a measure that prevents user errors. The help arrows on the side can be set to a specific step length to give users, that would like to use the mouse as much as possible, a good experience.



(a) Spin box. Input: integer.



(b) Double spin box. Input: float.

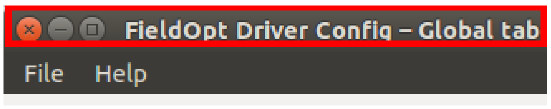
Figure 6.8: Example of the two types of spin boxes.

6.2.2 Navigational Components

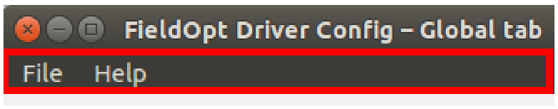
To be able to navigate through an application in a satisfying way and perform wanted actions we need universal navigation components. We have tried to follow advice of keeping consistency by standarize task sequences and group related tasks together, and structure those as logical as possible. The navigational components and measures used are title bar, menu bar, buttons, scroll bar, tabs, tab order, and modal window(dialog mode). Icons can also be mentioned as they can be a measure to guide the user to the right buttons and actions. See Subsection 6.2.3 (Informational components) for more about icons, and which icons that are being used in this application.

Generally, there are two kinds of windows/dialogs: modal and modeless dialog. A **modal window** is a child window that creates a mode so that the main and/or parent window cannot be used while itself is open. It is usually chosen to force or require the user to finish the interaction in that particular dialog before continuing in the main/-parent dialog window. It is also typically used in message boxes, in our case, for the information and question ones. In this interface, we use modal window as a feature to help the user to keep control of the dialogs. By restricting the amount of open dialogs to only the main window and the first level dialogs, there is a lower risk of the user getting confused in the potential sea of dialogs. The modal yields for the second level dialogs, seen in Figure 7.1. The rest of the GUI structure, the main window and first level dialogs, are modeless dialogs which implies they can be open at the same time. Qt has decided to keep the parent dialog as it is when we activate a child window. The title bar is greyed as when a dialog is not active. The rest is still visible, but the windows are disabled. It is not allowed to take any action (in the parent window or levels above) before closing the modal/child dialog, except for moving them around.

An application window has a certain possible layout where you can add standarized bars in addition to the very common title and menu bar, e.g. tool and status bar. We have chosen to not have a tool bar because it does not seem relevant or useful. A status bar may be a future solution to show progress (e.g. keeping count of the settings and entered input).



(a) The title bar area is marked by the red rectangle, and has the current title title: "FieldOpt Driver Config – Global tab", in addition to the standard exit, roll-up, and maximize/minimize buttons on the left.



(b) The menu bar area is marked by the red rectangle, with the current menu names: "File" and "Help".

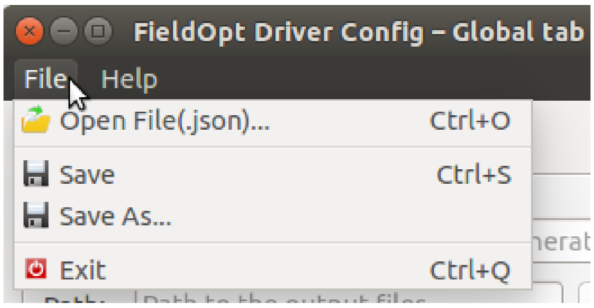
Figure 6.9: Screenshots of the application's main window's left corner – the title bar and menu bar. Their location is illustrated in the subfigures a and b by a red rectangle.

The **title bar** is used to display the name of the application or dialog window you are accessing. It lies horizontally on the top of the window dialog, as you can see in Figure 6.9a. It may and does also contain small buttons for minimizing, maximizing, exiting or rolling up the windows. Window and dialog title (title bar) is in addition an informative component, and could also have been mentioned in Subsection 6.2.3.

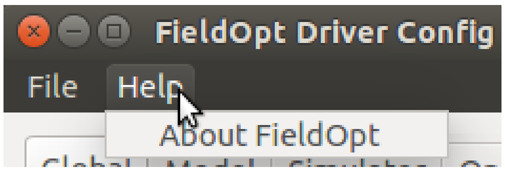
The **menu bar** is at the top level of the menu hierarchy, and consist of menus, so-called pop-up menus. Only pop-up windows can contain a menu bar, implying that child windows cannot. Currently, our GUI has two menu items or menu names: 'File' and 'Help' marked in Figure 6.9b. They are always visible in the main window in contrary to the menu that drop downs from these names only after being clicked. To help seperating the menus' menu items, a separator is used. This is a line that does not steal much attention, but does a solid job on grouping the items, which makes it easier for the user to select the wanted option. A similar separator, also a design element, has been used several places, horizontally and vertically, in the interface with the same intention of helping the user to group the content.

The Help item currently consists only of 'About FieldOpt' (see Figure 6.10b), when clicked by the user, opens a dialog with information about FieldOpt. It could have been replaced to a level up as there are no additional options, just an unecessary extra step to reach the only option. But this solution is under development and is therefore a temporary solution. Later, we may add more submenu items like a manual.

The File menu name, when clicked, activates a menu with the four menu command items, seen in Figure 6.10a. Well-known icons have been chosen along with ordinary informative names, and shortcut keys on the side.


FieldOpt Driver Config – Global tab
File
Help
Open File(.json)... Ctrl+O
Save Ctrl+S
Save As...
Exit Ctrl+Q

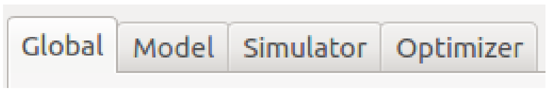
(a) A state where *File* is clicked showing the command items with icons and shortcut keys.



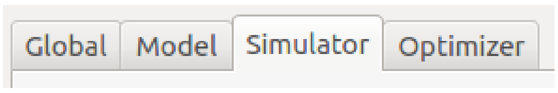
(b) A state where *Help* is clicked showing the 'About FieldOpt' item.

Figure 6.10: Screenshots of the Menu bar with menu items *File* and *Help*. The subfigures a and b demonstrates what the submenus looks like.

Tabs let the user move quickly between the equally important views. As these views will be used frequently, tab is a great tool. A tab has a title to descibe its content. In Figure 6.11 you can see what it looks like when a tab is active. The content and view change depending on the specific tab, as you can see when comparing Figure 7.2, 7.3, 7.4, and Figure 7.5 from GUI Development & Implementation Process, Chapter 4.



(a) A state where the *Global* tab is active.



(b) A state where the *Simulator* tab is active.

Figure 6.11: screenshots of the GUI's tabs in two different states. The subfigures a and b demonstrates what activation of different tabs look like.

Tab order is an order of importance especially for those who like to use the keyboard as much as possible. When navigating by pressing 'tab' the selection is expected, but also designed to move in a logical, practical, and predictable order. The selection order usually goes from top to bottom, and from left to right.

6.2.3 Informational Components

With the right combination of informational components in our application, we may have created a user-friendly informative solution.

The main way of describing what a specific ‘field’ is about is through the use of a display widget called ‘**label**’. The informative text is direct, usually short and consise, and in a normal font and size. By ’direct’ we mean that the information is clearly visible without requiring any special action from the user.

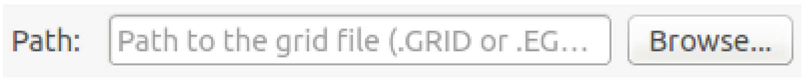
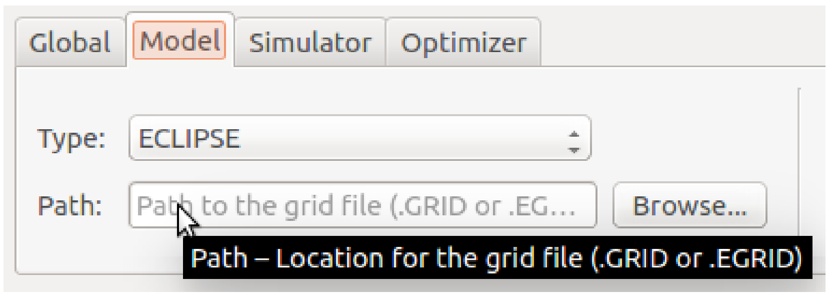


Figure 6.12: This is a screenshot from the Model tab of the path field that consist of: the label ‘Path’, the placeholder text ‘Path to the grid file (.GRID or .EGRID), and the ‘Browse...’ push button.

Placeholder text is another feature for guiding the user in a direct way. Information can be placed in line edits and text fields by the use of this feature. The font is of normal size and style. Contrary to labels, the text is grey and discreet. In this way, the text will not steal attention and focus away from other elements. In addition, it helps on keeping design simplicity. When the user starts writing, the new text replaces the placeholder text, exactly what the name of the feature hints about. With the use of these two components the user should be aware of and sure about what to put in that particular element. If more information is needed, a tool tip can be used, especially for elements that do not have the placeholder text feature available.

Figure 6.13: Screenshot of the Model’s Well name field’s line edit, with the placeholder text: “Well name”.

Tool tip is a feature that helps out on pinpointing what input a field requires, or gives information about other type of graphical elements on the screen. For very “obvious” fields, the tool tip will probably be exactly the same text as the label. It may be present only to confirm the user’s suspicion about the field, but also to keep consistency. For more complicated fields, extra information could clarify some uncertainties. The tip pop ups as a line of information when the user holds the pointer over a specific element for a second. The “box” has black fill and white text color. An example from the GUI can be seen in Figure 6.14. Every element placed in the application can have a tool tip, and we want to provide that in our GUI. Some other examples are: the Model tab’s browse button has the tip: “Browse – Find the path to the grid file (.GRID or .EGRID)” which is similar to the other browse buttons, and the plus button at Model’s control times says: “Add control time step (item) to the list”.



Path – Location for the grid file (.GRID or .EGRID)

Figure 6.14: Tool tip of the model grid path appears when the pointer is on the element. This is a screenshot of parts of the Model tab: the Model’s field Type and Path.

A graphical symbol/**icon** should have a clear, familiar and intuitive meaning. It is important to be keep consistency in using the icon. For example: it is well known that the floppy disc as an icon represent ‘save’ action. The floppy disc icon, Figure 6.15, chosen for this GUI is of Ubuntu/Linux design, and can be found in the menu bar together with the open/import file icon(Figure 6.16), and exit/quit icon(Figure 6.17). Figure 6.2 is a combined example of button and icon with the well known pensil for editing. There is also an icon attatched to the message box.

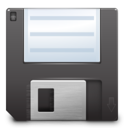


Figure 6.15: Save file (floppy disc) icon.

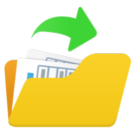


Figure 6.16: Open file (open document case and arrow) icon.

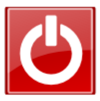


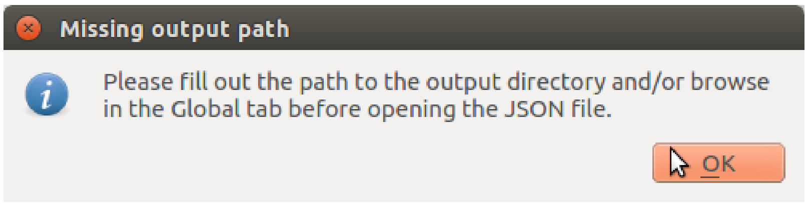
Figure 6.17: Exit icon, well known as the power sign (illustrated by a 1 and 0).

Message boxes are modal dialogs that pop ups to give the user a message – a feedback from the system. They represent different types of messages, and are well known for most people. The feedback is given as a respond to particular actions taken by the user. The “name” of the box/dialog together with a symbol represent the message we want to convey, e.g the degree of severity. The icon state the kind of feedback: i for information (see Figure 6.18, triangular sign with an exclamation mark for warning (warning sign), or a question mark (see Figure 6.19) for a question or request. Error message boxes with a red cross(x) can also be used. Those standard dialogs are set up with specific reply buttons.

Figure 6.18: Screenshot of the information icon that is attached to the information message box.

Figure 6.19: Screenshot of the question icon that is attached to the question message box.

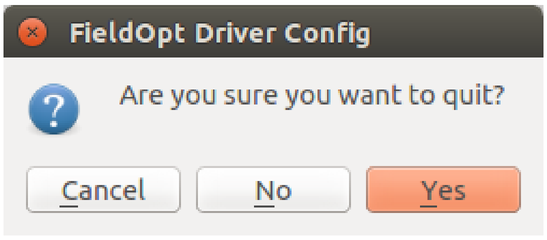
Figure 6.20 is an information message box that pop ups to guide the user to the right task order. If the user has tried to import a file without specifying the output file path, this dialog will appear. Figure 7.7 is another example, from the Model Tab. The Figure 6.21 is great example of a kind of safety net we provide, to make sure the user knows what is going on. We use this response only when the consequences can be of importance, i.e if the action that was taken was not done intentionally. Overuse will only result in irritated and frustrated users.



Missing output path

Please fill out the path to the output directory and/or browse in the Global tab before opening the JSON file.

Figure 6.20: The information message box that pop ups when the output path is missing and the user is trying to open/import a .json file.



FieldOpt Driver Config

Are you sure you want to quit?

Figure 6.21: The question message box that pop ups when the exit button in the title bar of the main window is released.

Chapter 7

Graphical User Interface for FieldOpt Driver Configuration

The Petroleum Cybernetics Group wanted to get a better and more user-friendly solution related to their petroleum optimization software in development, FieldOpt. The product, a graphical user interface, is the result of this Master's thesis work. We here presents the FieldOpt Driver Configuration interface with it's interface elements including description and screenshots of the different parts of the GUI. For the source code we refer to the GitHub repository https://github.com/karolinr/FieldOptDriverGUI.
The modified version of the Utilities library that was used can be found here:
https://github.com/karolinr/FieldOpt/.../Driver-file-gui/Utilities/settings. To better understand the GUI's fields we recommend to read through the Utilities/Settings README file. https://github.com/karolinr/FieldOpt/.../Utilities/settings/README.md. The user should be able to access every alternative and parameter that are supported by that library, through the GUI.

7.1 User Interface Structure

The current application dialog structure is illustrated in Figure 7.1. The figure provides an overview on how to operate the application. Everything can be accessed through the main application window.The main window consists of four tabs: Global, Model, Simulator, and Optimizer tab, and an overview (see Figure 7.2. It is through those views the first level dialogs can be opened and accessed. From the structure it can be seen which of the tabs that has to be active before the user can access the other dialogs. To get access to the next level's dialog, you need to click a button that triggers the application to open the intended dialog.

This structure can also give a better understanding of what the object hierarchy from the Utilities/Settings library and the structure of the JSON driver file look like. Since the GUI structure is made on the basis of these, it is easily recognizable when comparing this to the driver.json in the Appendix.

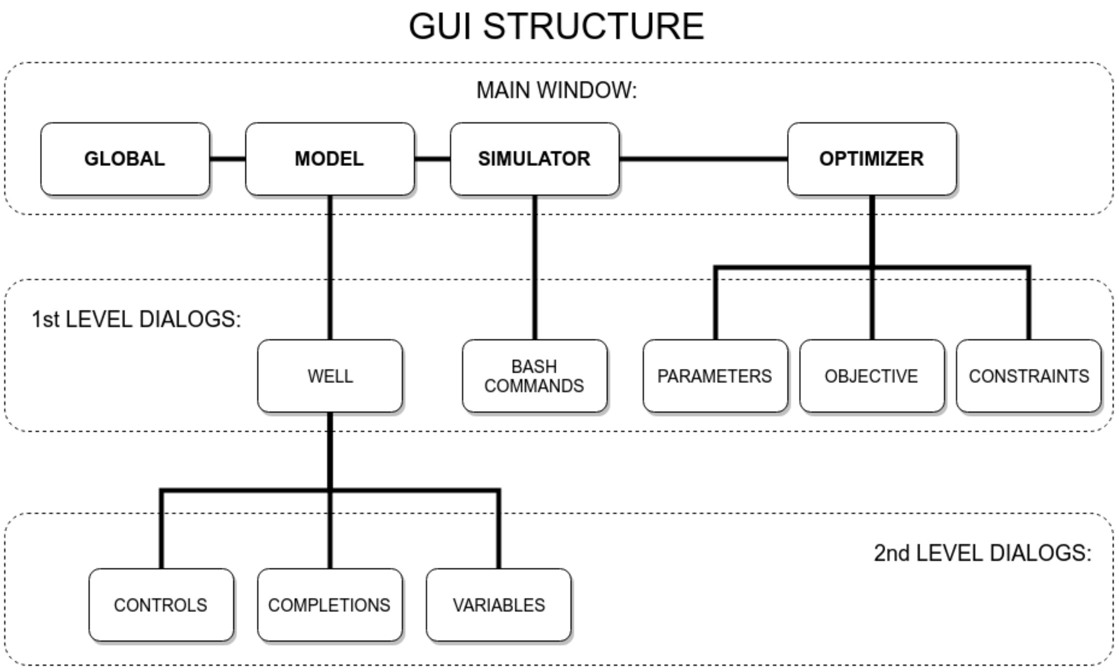


Figure 7.1: A simple dialog structure chart of the FieldOpt Driver Config interface.

All dialogs can be open at the same time, apart from the second level dialogs; the Model tab's well's child windows can only be open one at a time. Those are defined as modal dialogs. That means that the other "active" dialogs are blocked/disabled until the current second level dialog is closed. The other dialogs are of modeless type, opposite of modal, meaning that the user can switch focus between the main window and the first level dialogs (have them open).

7.2 The Application

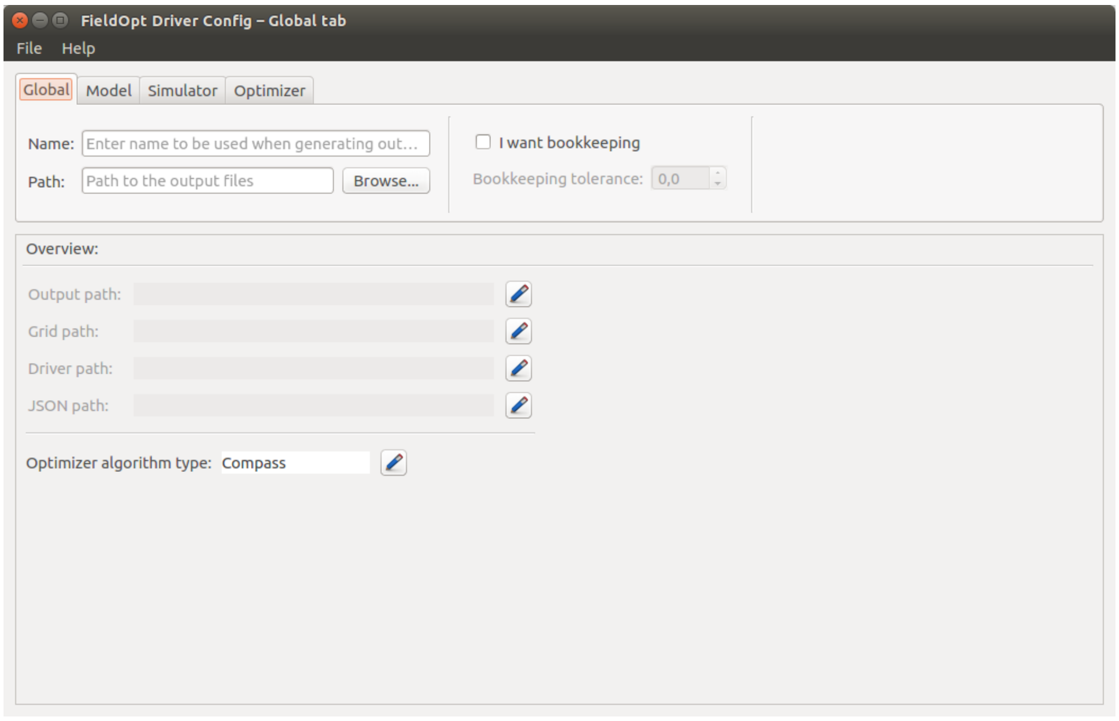
7.2.1 Main Window

The main window consist of the title bar, menu bar, tabs, and an overview. The menu bar has currently two menu names, as already explained in the previous chapter.

At this point, on the basis of the target audience, we have chosen to not focus and go for a solution with a start “page”. We think the application’s purpose is clear to most of those who will start it, and an extra navigational step will only be of irritation.

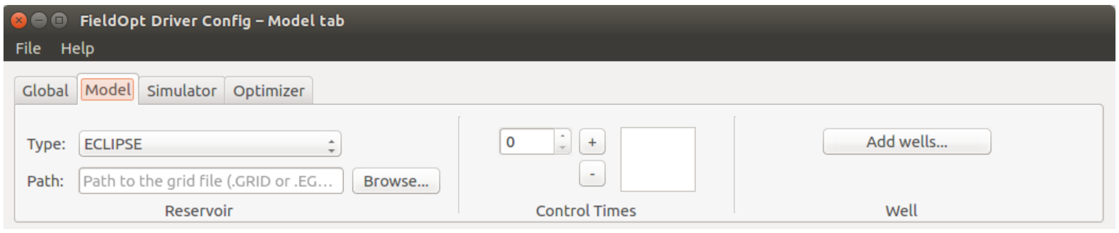
By executing the driver configuration application, the user also automatically starts on a new JSON file in order to create and edit driver file in the GUI. But it is not until the user clicks ’Save’ or ’Save As...’ that the temporary saved input should be tranformed into a configured driver file. These applications are not yet implemented.

As mentioned in the 6.2.2 section, tabs are a great tool to use when there are equally important views that will be used often. In general, this GUI does not have many forced orders or task flows. There are some inputs that depend on other, but we are planning to give feedback to the users if they have missed out on something. But we try discretly to make a natural order without locking the user to a specific flow. We hope the user will go through the tabs from left to right, in that way the user will be interrupted by messages about what to do next.



Optimizer algorithm type: Compass

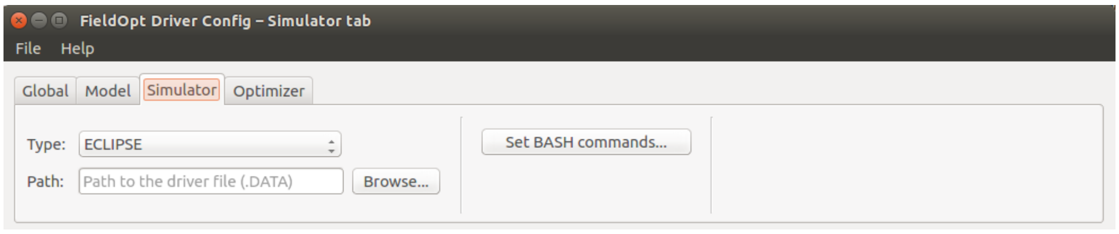
Figure 7.2: The main window with the Global tab active.



Type: ECLIPSE

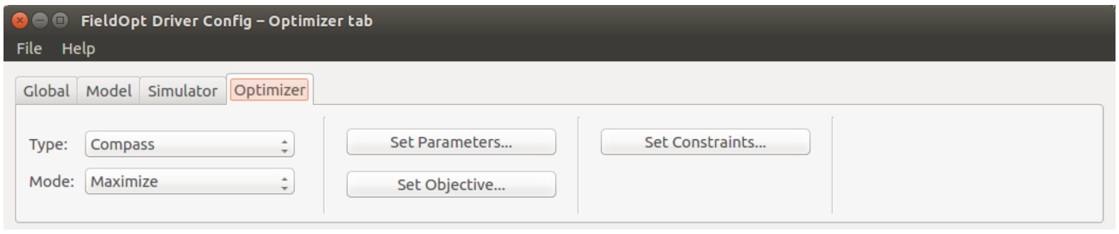
0

Figure 7.3: The main window's Model tab.



Type: ECLIPSE

Figure 7.4: The main window's Simulator tab.



Type: Compass

Mode: Maximize

Figure 7.5: The main window's Optimizer tab.

7.2.2 Global Tab

In the Global tab, see Figure 7.2, three inputs are required from the user – through the use of two line edits and a double spin box for the float input. “Name” is of string type and the input is used to derive the output file name(s). The user also gets to choose the path to a directory where the generated output files will be placed later. Through the well-known and simple browse button a standard browse file dialog opens that has been programmed to restrict the choice to the only valid choice, a directory.

The bookkeeper tolerance is used to set the tolerance for the case bookkeeper. The bookkeeper keeps track of which sets of variables that have been evaluated, to prevent unnecessary evaluation. The tolerance has to do with the limit for evaluating variables a “distance” to other already evaluated variables. In the design we have added a way to simplify the choice if you do not want bookkeeping. By not checking the “I want bookkeeping”, the bookkeeping field is disabled(not enabled) and the default value is set to 0.0. That value is what the driver file will store. The user only needs to check the box if he wants to set the tolerance, and then he may enter the desired value. For users that do not know what a bookkeeper is and what the tolerance may be, the choice of wanting it, is less confusing than to set it to manually to 0.0. This is also a field that should be considered to be included in the future manual, suggested in the Recommendation for Further Work and Development section.

The desired inputs are communicated to the user by label, the tool tip, and placeholder text, in addition communicated “discreetly” through the chosen GUI design element.

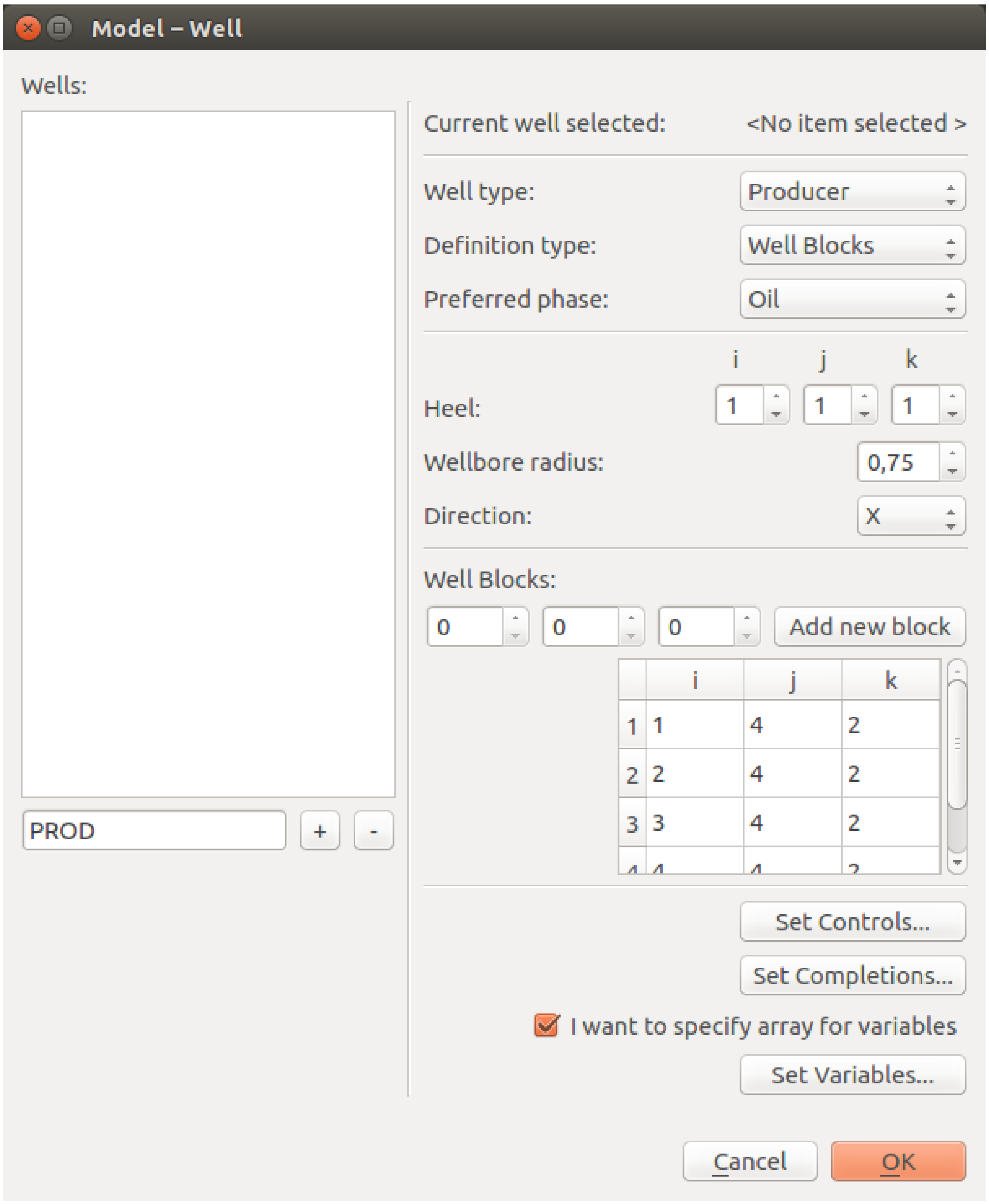
7.2.3 Model Tab

Through the Model tab, Figure 7.3, the user defines model related settings, i.e defining wells for the reservoir. The user needs to choose the type of reservoir model that is going to be used. The combo box restricts the use of models to the ones supported by the software – to avoid errors or misunderstandings. The only supported source, for now, is ‘ECLIPSE’, so the placeholder text and other help features hints about the path to a specific type of file. If others are added, the help text should dynamically change with the current selected model source. But for now, the GUI asks for the path to this reservoir grid file. Again, the user may browse for a path to the only valid file type, restricted to .GRID and .EGRID.

For the Control Times, we want to collect an array for all time steps at which any variable is allowed to vary. That means that the user needs to set all time steps that are going to be used further in the model. The GUI elements restrict the user to only enter integer numbers. Small add and remove buttons with the commonly known signs, + and -, shall be used to add/remove the desired/undesired time steps to the list box. The use of tool tips is quite valuable here to hopefully confirm the user's first impression and opinions about how to take action.

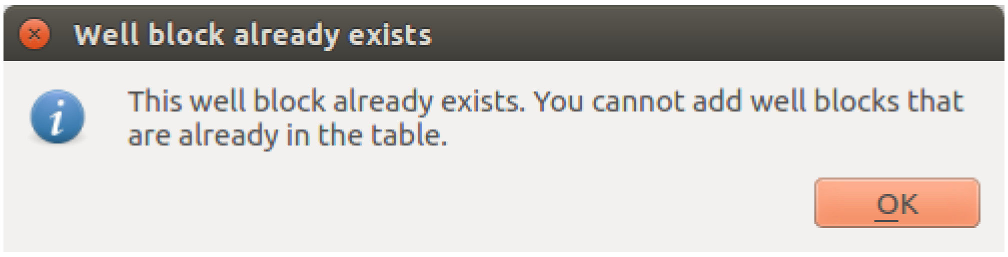
Wells can be defined and information about them can be specified in the "Model - Well" dialog, Figure 7.6, starting by clicking 'Add wells...' in the Model tab. From here the user should set well controls and completions; however, to set Variables are optional. See Figure 7.8, Figure 7.9, and Figure 7.10 to study the Model's second level dialogs further. If the user wants to specify array for variables, he can check the check box, and then the button for opening the dialog will be enabled.

Some elements in the dialogs are placed there depending on some choices made by the user, e.g. changing well definition type from 'Well Blocks' to 'Well Spline' changes the label's text, and spin boxes from integer to double/float type. If the user tries to add a well block that already exists, the response is that a message box pops up and informs the user (see Figure 7.7). It prevents the user from adding an identical block, and it explains why the normal response of clicking the button did not happen. Users should always be fully aware of what happens during interactions and why.



	i	j	k
1	1	4	2
2	2	4	2
3	3	4	2

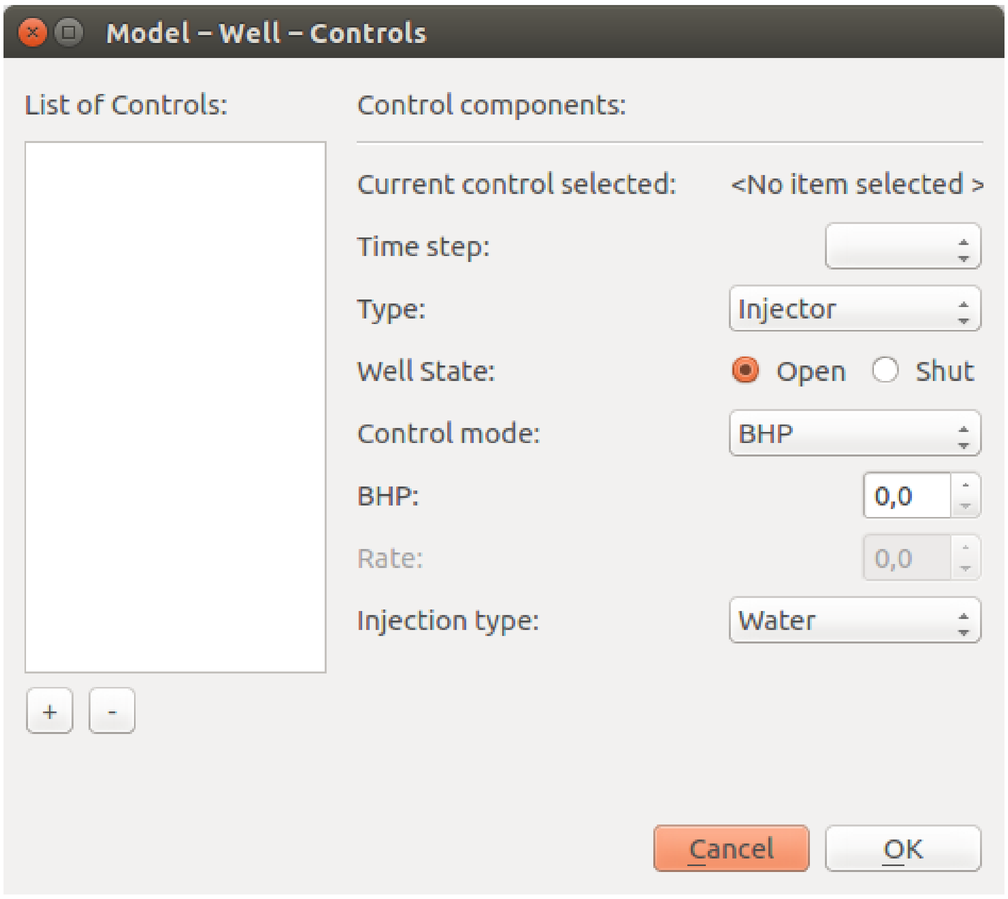
Figure 7.6: The Model's Well dialog with imported data from the driver.json file.


Well block already exists

This well block already exists. You cannot add well blocks that are already in the table.

Figure 7.7: The information message box that pop ups as a response to an action in the Model's Well dialog.

Again, GUI elements are specially chosen for a purpose of restricting user input to the valid values and types. This idea is applied in every dialog.


Model – Well – Controls

List of Controls:

Control components:

Current control selected: <No item selected >

Time step:

Type: Injector

Well State: [✓ Open] Shut

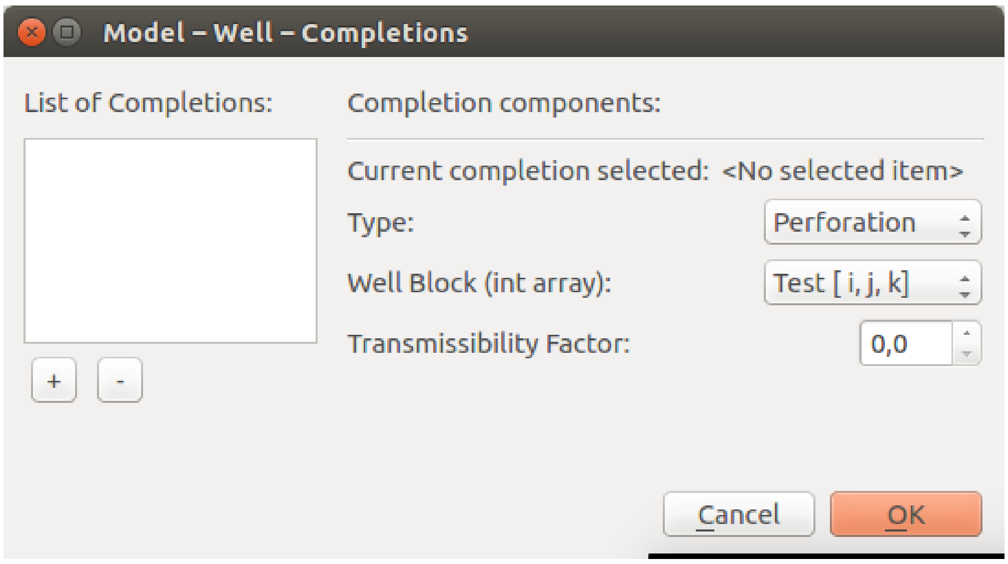
Control mode: BHP

BHP: 0,0

Rate: 0,0

Injection type: Water

Figure 7.8: The Model Well's Controls dialog with some imported data from the driver.json file



Completion components:

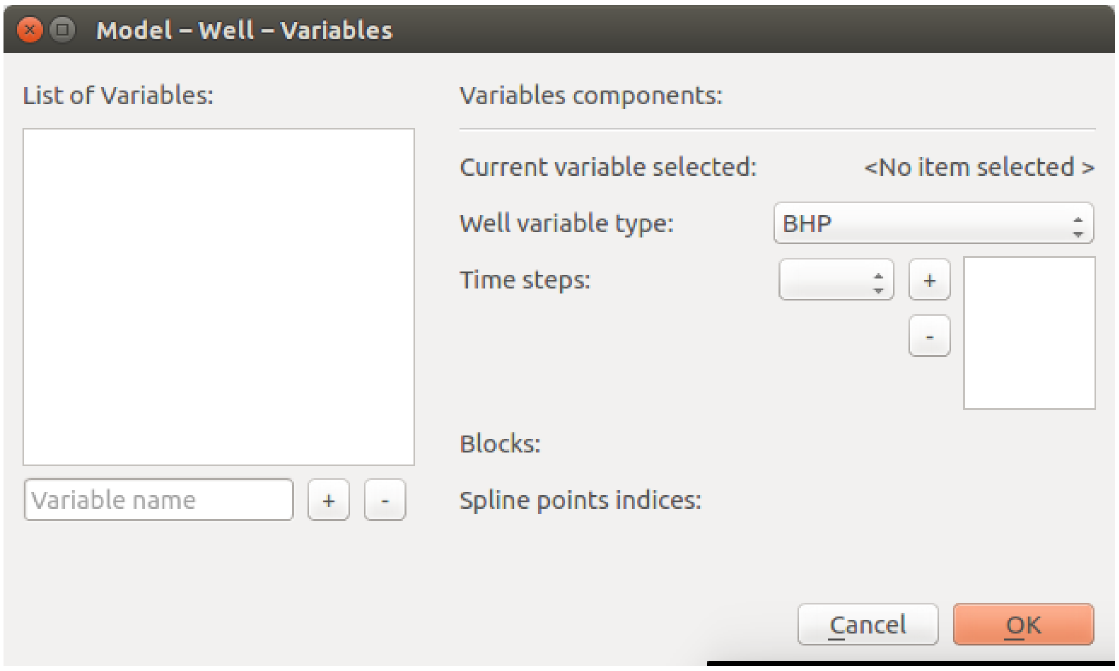
Current completion selected: <No selected item>

Type: Perforation

Well Block (int array): Test [i, j, k]

Transmissibility Factor: 0,0

Figure 7.9: The Model Well's Completions dialog.



Variables components:

Current variable selected: <No item selected >

Well variable type: BHP

Time steps:

Blocks:

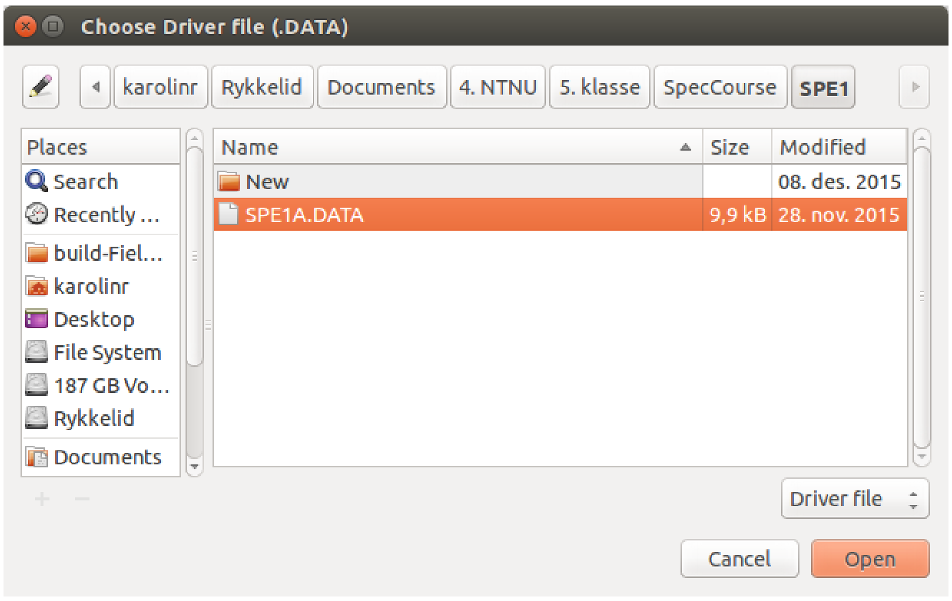
Spline points indices:

Figure 7.10: The Model Well's Variables dialog. Incomplete design of blocks/spline points.

7.2.4 Simulator Tab

The Simulator tab, Figure 7.4, contains fields so the user can define settings and parameters that is needed to launch the simulator. The field 'Type' denotes what simulator that is being used. The user can currently switch between two types in the combo box, either 'ECLIPSE' or 'ADGPRS'.

Again, the browsing is restricted to a specific file type. This time we want the user to choose the path to a complete driver file for the reservoir model, a .DATA file.

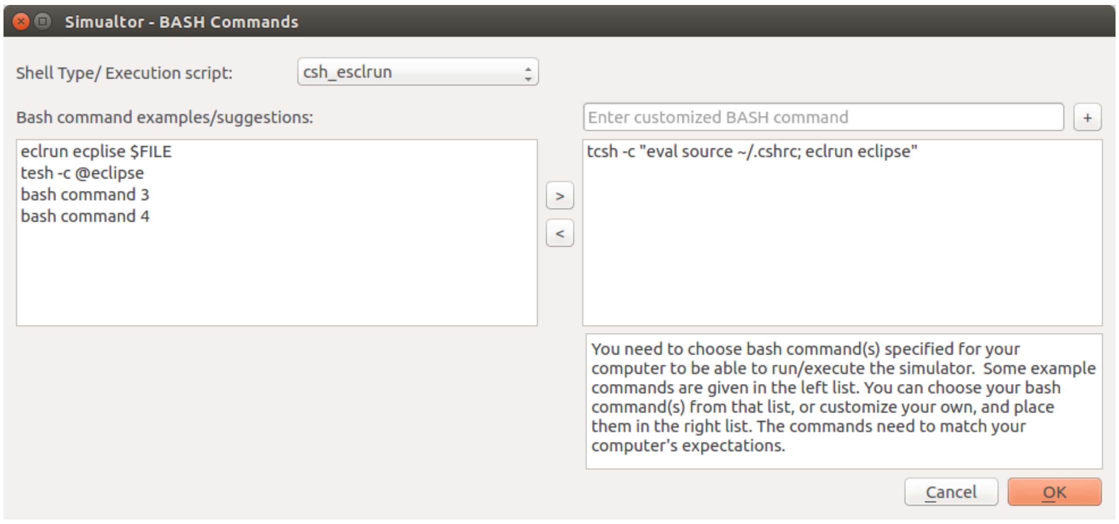


Name	Size	Modified
New		08. des. 2015
SPE1A.DATA	9,9 kB	28. nov. 2015

Figure 7.11: The Simulator's browse for a driver file dialog.

After clicking the 'Set BASH commands' button in the Simulator tab, the 'Simulator – BASH Commands' dialog opens. In this dialog, Figure 7.12, we want the user to enter and/or choose suited BASH command(s) for their computer and shell type, so the chosen simulator can be executed later. Users can also customize their own command if none of the examples match their need.

Risk of error The BASH command dialog is recognized as a specially error prone part of the interface, in terms of user error. It is probably the most demanding and difficult parameter to ask of the user, hence the most challenging part to keep error free. We are forced to accept whatever is being written in the line edit as we do not have a way to test or find the right bash commands for the specific user, because it is hard for us to know exactly the type of computer used. Our solution is to try to reduce the risk of error as much as possible. Our thoughts here are that we try to guide and help the user towards a correct input. There is a describtion text and example bash commands –



Shell Type/ Execution script: csh_esclrun

Bash command examples/suggestions:

```
eclrun ecplise $FILE
tesh -c @eclipse
bash command 3
bash command 4
```

```
tcsh -c "eval source ~/.cshrc; eclrun eclipse"
```

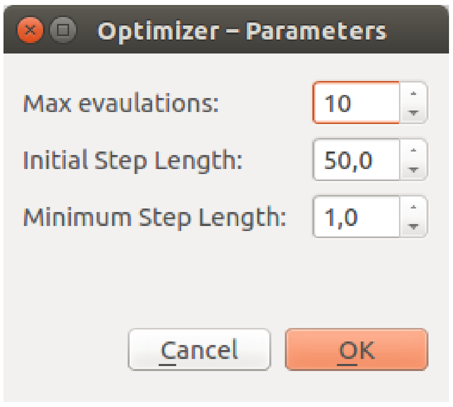
You need to choose bash command(s) specified for your computer to be able to run/execute the simulator. Some example commands are given in the left list. You can choose your bash command(s) from that list, or customize your own, and place them in the right list. The commands need to match your computer's expectations.

Figure 7.12: The Simulator's BASH dialog with imported data.

better and more examples will be provided along with tool tips.

7.2.5 Optimizer Tab

Through the Optimizer tab, Figure 7.5, we require the user to specify settings and parameters for the chosen optimization algorithm. The only algorithm type that can be chosen in the current solution is 'Compass'. The mode is chosen between either 'Maximize' or 'Minimize' the objective function. The Optimizer tab contains three sub-objects – first level dialogs. By clicking either of the buttons in Figure 7.5, a new dialog opens; either the parameters, objective, or constraints settings will be available for specification. The Parameters dialog has only three fields. The input is controlled through



Max evaulations: 10

Initial Step Length: 50,0

Minimum Step Length: 1,0

Figure 7.13: The Optimizer's Parameters dialog with imported data from driver.json.

the use of spin boxes. When the range of the different parameters are decided, they will be set in the GUI making sure only valid numbers are accepted as input. Max evaluations are set to be an integer number, initial and minimum step length can only be a float with one decimal.

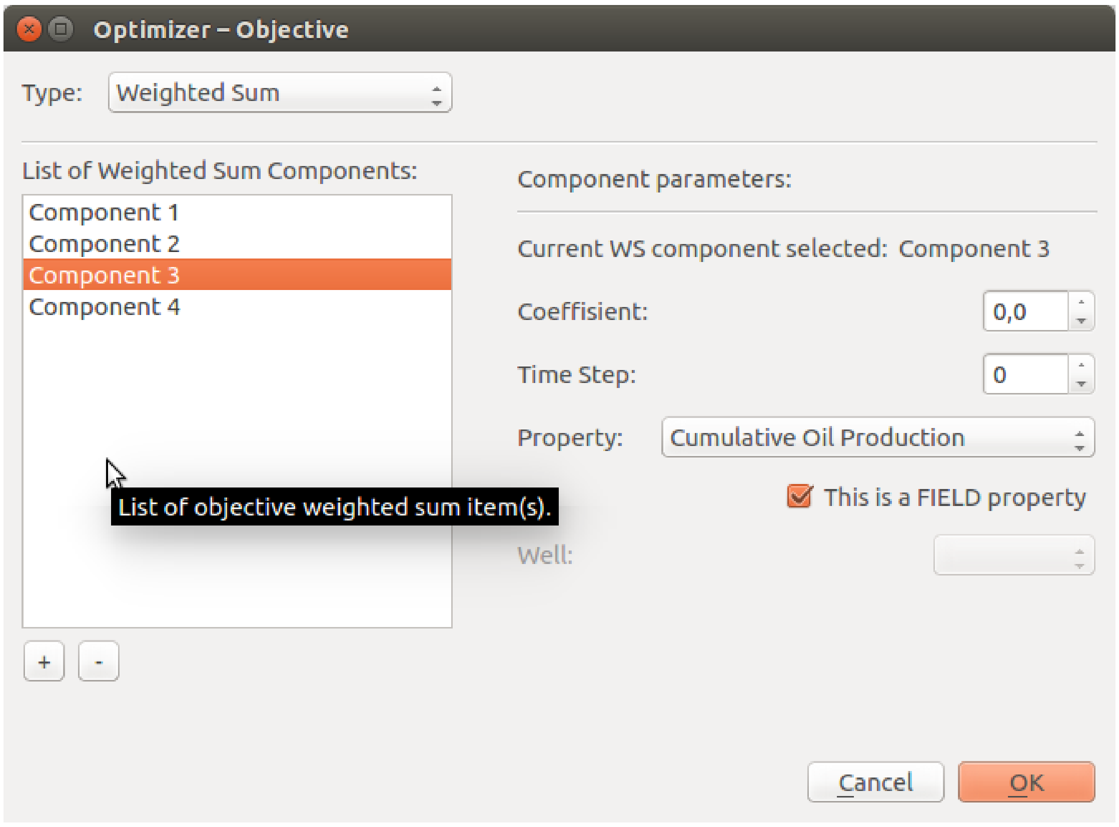


Figure 7.14: The Optimizer's Objective dialog

The user gets to set constraints on a well if there exists any. The combo box should be filled with well names as soon as the well items are being added in the Model section. If no wells have been added, the rest of the constraints dialog elements are disabled.

Figure 7.15 shows the Constraints dialog with imported data. The current constraints for the PROD well is of the BHP type, and that is the reason for why the Well Spline points elements are disabled. The GUI has the intention of 'PROD' in the Well name line edit to added as an item to the list. There is no reason to let the user have access to fields that are related to the other option, when they are mutually exlusive.

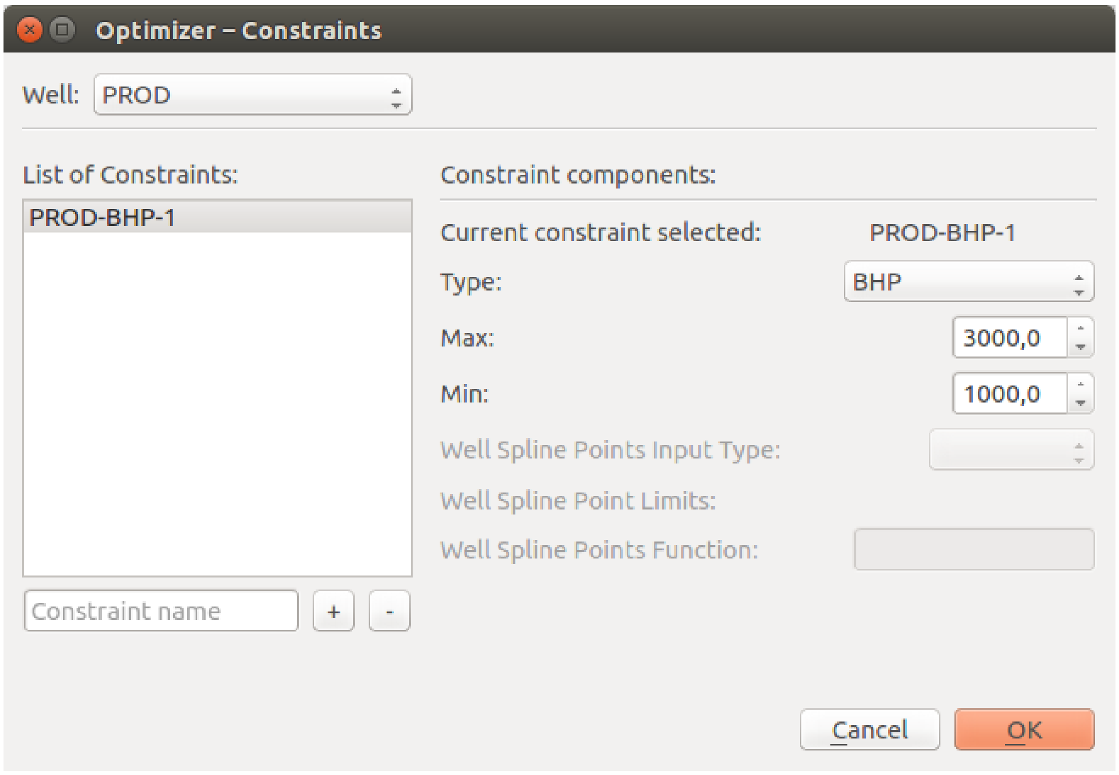


Figure 7.15: The Optimizer's Constraints dialog with imported data from driver.json.

7.2.6 Overview

The overview is a "static" view, in the meaning that it is available and visible independently of the active current tab – only It will probably be reorganized. Which parameters that will be included and summarized in the overview will be evaulated/considered closely. A user survey could also be of use to see what information most users would prefer to see in the summary section.

For example, the formula and result of the optimizer's objective input variables can be interesting. At the moment the only planned type(objective function) supported is weighted sum – meaning that the calculated weighted sum could have been added as data for the overview. The overview is probably going to be considered more in the last part of the development, when more of the application is functional and we can get users' advice.

If the overview needs to contain a lot more information, we can expand and adapt the initial size of the main window to the wanted displayed information. Anyway, it is important to include only relevant data.

Feature In the current solution, we have gathered the different source/file paths along with shortcut edit buttons. When the files are browsed for and selected, the path is changed, then the overview field will change accordingly. The edit button will take the user to the correlated tab and right into the ordinary way of selecting the file path. The text elements are disabled and less visible if the fields are empty – demanding less focus, as seen in Figure 7.2. The Optimizer algorithm type is also included in the overview – for now. After more review, maybe the only information about the Optimizer that is needed is a check list for the necessary input. Since the most of the information in the application is so easily accessable anyway, it does seem unnecessary to add the data to the overview as well. However, some information could be valuable anyway.

Intention of the overview is simple; to get the best overview of what parameters and variables the user needs to change or view before they can hit 'save' and have an ideal configured driver file. We want to give the user a sense of what is going on through the use of a simple summary of some of the input data. If there are some temporary results, these can also be shown here.

The idea is that the user can use the overview as a kind of check list. It may not be possible to include everything in the checklist, but at least it should give a clue of what the user needs to change/add to be able to run a simulation – all mandatory fields. We have considered the idea to involve the status bar available in the main window, to let the user keep track on the progress, but other solutions may be better.

7.3 Current Functionality

In this section, we have decided to briefly present a couple parts of the GUI and how the functional part of the application is and can be.

Dialogs Every dialog has a button box with an ‘Ok’ and ‘Cancel’ button present. To further explain to the user what the concequences of the click actions are, the tool tip is set. For most of the dialogs the text is something similar to this: “’Ok’ to accept new changes/values. ’Cancel’ to reject changes.”. The original settings are reinstated to the previously accepted input, if the ‘Cancel’ button is pushed. ‘Ok’ will provisional save the settings, so a ‘Cancel’ next time will lead to the settings’s previous state.

When maximizing the dialogs, almost every dialog keep their initial horizontal size. This design choice may be evaluated later, but we cannot see any reason it will give the user any advantage, at least in the way the dialogs are designed today. However, the dialogs expand vertically, mostly because there are list boxes present. If the user decides to add many items to those lists, it can be easier to keep track on the items by viewing more of them at the same time. The user may use the scroll bar to navigate through the list if there are more items than it is possible to view on the screen.

Open/import a JSON file First, make sure you have chosen a directory for your output files. Then click ‘File’ on the menu bar and click the ‘Open/File (.json)...’, a new dialog opens which let you browse to select the JSON file you want to open in FieldOpt Driver Config. If you have not set the output directory yet, an information message box will pop-up letting you know what you have to do before opening a file, see Figure 6.20. After following these instructions, the driver information should be visible in the application.

Response The current not complete version of the application does not handle every dialog signal yet. There is a lot of remaining coding to do before the interface is totally functioning. The program will unexpectedly finish in five of the dialogs when clicking ‘Ok’ or ‘Cancel’ in the current dialog, except from the Optimizer ones. However, that is only if you have not imported a JSON file. The coding part is in the middle of a state where functions are implemented to temporary save new changes done in a dialog. A Settings class object from the Utilities/Settings library is not yet created when starting from scratch on a driver file. Some functions assume there exists an instance and uses this kind of object. Therefore will errors occur when these functions are run, leading to the unexpected finish.

Chapter 8

Evaluation and Reflection

In this chapter, we have evaluated the project. We are looking at the degree of fulfillment, and we are comparing the result with the given requirements from Chapter 5, in addition to the design guidelines established in Section 3.2. The following bold text are points, rules, or requirements taken from those parts of the project report, marked to let the reader follow more easily. Further, we present some thoughts and reflection.

8.1 Design Guidelines

This section will briefly sum up how and if the golden rules and heuristics have been used and followed.

We have strived for **consistency** in every part of and throughout the interface, as far as we see it. We have designed the GUI with left-sided labels, we have consistently placed elements at the same starting point horizontally, on the x axes, and navigation is standardized. The GUI elements are generally used for their purpose and what they are known as for most users – fulfilling the recognizability quality. The GUI has the same structure for the same type of input collecting.

We have managed to follow the point about **error prevention**; how to prevent and avoid errors from Usability Heuristics for User Interface Design to the limit. Some error prone parts are recognized (Simulator class), and have been handled as far as we are capable. The risk of errors are assumed to have been reduced, however unfortunately not eliminated.

We know that some parts has the potential to make the user do something wrong, and therefore we have responses to those actions. We stop and help them by providing guided **feedback** for **simple error handling**, and we force them to handle in a certain task order.

We have kept the layout simple, **aesthetic and minimalist design**, so the **user's memory load** is minimized. We have not included information without relevance that competes for the user's attention. **User control and freedom** are given by letting the users undo changes by a click on the 'Cancel' button in dialogs and because it is easy to change back values manually. The GUI is quite simple and is therefore, in our eyes, not complex enough to have extra flexible solutions or have shortcuts(other than the File menu ones). We do not think there can exist more efficient GUI solutions. There are short steps from start to finish for most input. Though, there are some optionally fields, but that originates from the JSON driver file structure.

When it comes to **help and documentation** we have chosen to not focus on those. One of our goals is anyway to create a GUI that can be handled without any help option. Thus, there is created a menu item that can be expanded with such information.

We have managed to avoid most of the design mistakes that were presented in Chapter 3. Though, we do not have objective opinions from "real" users, opinions of the forms created, nor of if options are unclear or not etc. In addition, since the interface is not yet complete, there exist some **promise of content** that is not there. In that way, we cannot really conclude about the mistakes nor if we have succeeded in following the guidelines. Though, in the end, they are just guidelines. At least, we have tried to follow them to best of our knowledge. And it is not too late to change text, placement etc. to improve the graphical user interface.

8.2 GUI Requirements

After working through the Table 5.1 of the 14 functional GUI requirements, we can conclude that we have faciliated for many of these requirements, but that they are not necessarily functioning at the current state of the GUI. Some requirements are only partially fulfilled. We chose to prioritize all the listed requirements as high, except for two, sense of progress and diversity in users, that was defined as medium and low.

The application has included the **Utilities/Settings library**, as also written in the project description, in order to **read a JSON file**, so the GUI can enable the user to **import a JSON file**. Some programming and placement for a couple of elements remain before this requirement is more than partially fulfilled. As mentioned, this is not the only requirement that needs programming to be fulfilled.

The **write JSON file** requirement cannot be implemented as long as the functions in the Settings class are not implemented yet. These functions can be implemented in a similar way to what we have seen for the read file functions. This is also a huge part of why the GUI is not fully functioning.

Concering **data presentation** and **user manipulation of data**, the user is able to use GUI elements that presents all the data and manipulate it, with only a few exceptions in the Optimizer's Constraints and Model's Well's Variables dialog.

All functionality that concerns **valid input** has not yet been implemented, but through the use of interface elements with special purposes, and already implemented tests, we have met most of the requirement. The only thing that remains is to understand the petroleum fields: numbers and values, to be able to set the restrictions/range for the input directly on the element. To set them is an easy job. However, we have a weakness and risk of error mentioned earlier, in the Simulator's BASH Commands dialog. And since we cannot meet that criteria for all input, the requirement cannot be fulfilled to the fullest.

Currently, the user is being able to access and click the 'Save' and 'Save As...' item in the File menu, or use the shortcut for save, but the functionality is not implemented. Meaning, that the front-end of the requirement is fulfilled, but not the back-end.

PCG stated that they want the GUI to generate a meaningful driver file. Assuming the JSON driver file structure is meaningful, the GUI only accepts valid input, and the Settings class' functions are used correctly, this should also be fulfilled. At the current state, the GUI is not implemented with all its functionality, so for now, this requirement is not met.

Concering the requirement of **learnability**, we feel we cannot state much without conducting a usability test on a group of target audience. Nevertheless, we believe that by following all UI guidelines (that has the main focus of usability(which includes learnability)), we have facilitated and laid a good foundation for this to be true and fulfilled.

As already mentioned above, we have **prevented mistakes** to our best by only accepting valid input, but also by handling actions with important consequenses in a good way, through the use of message boxes.

We claim that the requirement of **navigation** is fulfilled, which implies that the user is able to easily navigate through the menus, options, and settings.

You may have noticed that we did not have the time to implement all the high priority requirements, and therefore, naturally we did not focus much on the **sense of progress** that was desireable either. The GUI can fully work without the overview.

Parts of the low prioritized **diversity in users** were fulfilled, but only as a consequence of optional fields.

In the Non-Functional Requirements section, the constraints were introduced, and they were taken into account during the development process. Developing the GUI using the Qt framework fulfills the portability quality requirement set for the system.

Concering usability, the quality attribute, we suggest based on what is written in the Section 5.3.1, that only appropriate user testing can evaluate those criterias. However, some of these have already been mentioned in other parts of this evaluation masked in other requirements. So on the basis of those we conclude that recognizability, learnability, operability, user interface aesthetics, and accessibility, are met to a certain degree.

8.3 Thoughts and Reflection

Even though the application is not functioning as we want yet, and the design still can be further developed. This GUI and Master's thesis will give ideas of how to continue developing the driver configuration GUI, and help and guide the PCG towards a good and better solution.

Great foundation As far as our knownledge reach, this is a GUI that should be easy to use and understand, even when the user have not used it before. We have tried our best to follow the golden rules set for the graphical user interface, and to avoid the common mistakes. There are some challenges with the GUI that still needs to be dealt with, but the GUI should be a great foundation for further development.

Although, we have not user-tested the GUI, some acceptance for the GUI's layout has come from potential users, especially in the early stages of the project. Some of the members of PCG have been a part of the feedback sessions, and they are categorized as target audience of the FieldOpt driver configuration interface. That gives us some confidence that the layout has the potential to be good.

This interface needs to be thoroughly tested for its user-friendliness/usability. Different use cases and scenarios can be created, then carried out by different test users. These users should be monitored, and surveys will be conducted to detect weaknesses and strengths – to see what parts of the interface are working as intended [Rykkelid, 2015].

Weaknesses The greatest weakness of this development has been the planning, the underestimating of the time constraint, and overestimating the expertise. The progress did not meet our expectations for this project. Those are not a good combination, and we are disappointed that we did not manage to end up with a more complete working application, but ending up with a not fully functioning GUI that has not been tested by users. That is certainly not ideal. In the most ideal world, we would have had a lot of user data to evaluate, then use this data to improve the GUI, and iterative again.

As mentioned earlier, one of the challenges and a weakness has been the technical competence of the developer. The lack of experience with this type of development has been a weakness for the result and process.

If we had predicted the time we had to use on the back-end implementation part of the application(the functionality), we could have switched focus to more of a total front-end development with the GUI layout and elements. Restricting the project could have been a solution to our challenges.

Large set of data/components. is a potential weakness of the GUI. List boxes are an attempt to handle objects with many components. But if there are very large sets of data, is the GUI designed to handle it? Is this a limitation of the design, or does the user get a good overview of the data without losing track of the components?

Design details Users notice every little design detail. If there is a disagreement between the developer and user, it will affect the use of the software/interface. Details that frustrate and irritates the user will affect their opinion about the interface, and every litle itch will count more than all the correct and positive design choices. If he gains negative attitude towards the interface, he will most likely spread this energy and thoughts about the GUI to other potential users. This will harm the software in the long run. That has to be avoided at all costs.

Even though, the project did not end up with having as good progress as first hoped for, it resulted in more than the visual results. The personal experience has great value.

In many ways, this project is successful.

Chapter 9

Summary and Further Work

In this chapter, we summarize the project and thesis, in addition we describe further recommendations and thoughts for the result's road ahead.

9.1 Summarization

This Master's thesis has concerned the development of a graphical user interface tailored for configuring driver files. These files are configured for the petroleum field optimization software, FieldOpt, that is currently being developed by the Petroleum Cybernetics Group at NTNU. The GUI was made as a replacement for the much simpler and inconvenient solution of direct text manipulation.

We have managed to follow the established design guidelines, partially fulfilled the system requirements, and we have exploited the GUI design elements, and carried out the development and implementation process.

Through the use of Qt Creator we have managed to create a GUI that could be used in the future to configure the JSON driver files. That interface will score high on usability as long as GUI golden rules are still being followed and thoroughly tested by target users in the future continuation of this project.

The current solution of the GUI, the application is not yet sufficient to take on any role in FieldOpt. We need to finish up the main functionalities and ideas of this GUI, before it can be accepted and realized as a true solution for PCG.

However, this thesis work has layed a solid foundation for the further development of the user-friendly FieldOpt driver configuration interface.

9.2 Recommendation for Further Work and Development

This section presents some thoughts and recommendation for further work on completion, expansion, and improvement of the interface.

Fully functional version There is still work left to do on the graphical user interface before it can be tested by potential users. To really test the GUI design the application needs to be fully functional and further developed from the already established foundation. Included in that finalization is the save function. Write-to-JSON-file functions have to be implemented in the Utilities/Settings library, and further used in the application. This can be done in a very similar matter to reading the JSON file.
To create a successful overview has the potential to be a very important detail of the interface. The development of the overview; structuring and placements of elements, and deciding what information to include, may play a huge part of if the FieldOpt Drive Config will have success or not.

Validation is an important part of figuring out if a creation is good enough. Other than the evaluation of the GUI and to what degree it fulfills the requirements, the GUI has not been evaluated. It has not been measured or assessed to what an individual, representing the target audience, experience in use. The current version has not been user-tested. The interface needs to be thoroughly tested for its user-friendliness/usability. We want to know if the GUI is usable, trustable, what the end-user and the PCG requires of the FiledOpt Driver Config GUI.

Software evolution As FieldOpt develops, even more parameters may need to be added to the GUI and driver file. The user needs to be able to configure these new parameters. The GUI have to be developed in parallel with FieldOpt. There may also be changes in operative systems and technology, which may lead developers to renew code. Hopefully, Qt will be developing along the way, and figure out ways to keep “old” code usable, at least easy to “convert”.

User manual In our curent solution, the only item of the Help menu is About FieldOpt. This menu could be expanded by adding access to a manual for guiding users. We would like to think that the interface should be simple and user-friendly enough so that the users will not have the need for any extra guidence or help. However, the manual may give users a comfort and answers when in doubt about how the GUI is working. This is something that can be explored while performing usability/user testing. Maybe the users feel the need to be able to search in a dictionary-like manual with explanations for all the parameters and some of the functions?

Bibliography

[1] E. J. M. Baumann, “Fieldopt: Enchanced software framework for petroleum field optimization,” Master’s thesis, Norwegian University of Science and Technology, June 2015.

[2] E. J. M. Baumann and M. R. Bellout, Personal communication, November 2015.

[3] The Qt Company, “Qt - about us.” (n.d.)., accessed: 02.2016. [Online]. Available: http://www.qt.io/about-us/

[4] The Qt company, “Qt - the ide.” (n.d.)., accessed: 02.2016. [Online]. Available: http://www.qt.io/ide

[5] The Qt Company, “Qt designer manual.” (n.d.)., accessed: 02.2016. [Online]. Available: http://doc.qt.io/qt-5/qtdesigner-manual.html

[6] B. Stroustrup, *The C++ Programming language*, 4th ed. Addison-Wesley Pearson, 2014, ch. 1.2, pp. 9–16. [Online]. Available: http://mazonka.com/shared/Straustrup4th.pdf

[7] B. Stroustrup., *The C++ Programming Language*, 4th ed. Addison-Wesley Pearson, 2014. [Online]. Available: http://mazonka.com/shared/Straustrup4th.pdf

[8] E. J. M. Baumann, “Fieldopt utilities/settings,” September 2015, accessed: 03.2016. [Online]. Available: https://github.com/PetroleumCyberneticsGroup/FieldOpt/tree/develop/FieldOpt/Utilities/settings

[9] JSON, “Introducing json,” (n.d.)., accessed: 03.2016. [Online]. Available: http://json.org/

[10] E. J. M. Baumann, “Settings readme.md,” September 2015, accessed: 04.2016. [Online]. Available: https://github.com/PetroleumCyberneticsGroup/FieldOpt/blob/develop/FieldOpt/Utilities/settings/README.md

[11] B. Shneiderman, Plaisant, Cohen, and Jacobs., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed. Pearson, 2014.

[12] J. Nielsen, “Checkboxes vs. radio buttons,” September 2004, accessed: 10.11.2015. [Online]. Available: http://www.nngroup.com/articles/checkboxes-vs-radio-buttons/

[13] ——, “10 usability heuristics for user interface design,” January 1995, accessed: 18.12.2015. [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/

[14] T. Mandel, *The Elements of User Interface Design*. John Wiley & Sons, 1997, ch. 5. [Online]. Available: http://theomandel.com/wp-content/uploads/2012/07/Mandel-GoldenRules.pdf

[15] K. Whitenton, K. Pernice, P. Caya, and J. Nielsen, “Application design showcase:2012,” *Nielsen Norman Group*, no. 02, pp. 229–299, 2012. [Online]. Available: https://media.nngroup.com/media/reports/free/Application_Design_Showcase_2nd_edition.pdf

[16] T. Point, “Software user interface design,” 2016, accessed: 05.2016. [Online]. Available: http://www.tutorialspoint.com/software_engineering/software_user_interface_design.htm

[17] M. Hansen, *Collaboration: How leaders avoid the traps, create unity, and reap big results.*, 3rd ed. Boston, Mass.: Harvard Business Press., 2013.

[18] B. K. Williams and S. C. Sawyer, *Using Information Technology: A Practical Introduction to Computers & Communication: Complete Version*, 9th ed. McGraw Hill, 2011, ch. 10, pp. 491–531.

[19] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, September 2011. [Online]. Available: http://faculty.mu.edu.sa/public/uploads/1429431793.203Software%20Engineering%20by%20Somerville.pdf

[20] ——, *Software Engineering*. Addison-Wesley, 2011, ch. 4, pp. 82–117, pdf: pp. 99-134. [Online]. Available: http://faculty.mu.edu.sa/public/uploads/1429431793.203Software%20Engineering%20by%20Somerville.pdf

[21] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Upper Saddle River, NJ: Addison-Wesley Pearson, 2013, ch. 4, pp. 63–65.

[22] International Organization for Standardization, “About iso,” (n.d.)., accessed: 22.05.2016. [Online]. Available: http://www.iso.org/iso/home/about.htm

[23] Wikipedia, “International organization for standardization,” May 2016, accessed: 10.05.2016. [Online]. Available: https://en.wikipedia.org/wiki/International_Organization_for_Standardization

[24] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Upper Saddle River, NJ: Addison-Wesley Pearson, 2013, ch. 12, pp. 185–200.

[25] International Organization for Standardization, “ISO/IEC 25010 – Security, Maintainability, Portability,” (n.d.)., accessed: 22.05.2016. [Online]. Available: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&start=3

[26] International Organization for standardization, “ISO/IEC 25010 – Compatability, Usability, Reliability,” (n.d.)., accessed: 22.05.2016. [Online]. Available: http://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&start=6

[27] K. Whitenton, “Minimize cognitive load to maximize usability,” *Nielsen Norman Group*, December 2013. [Online]. Available: https://www.nngroup.com/articles/minimize-cognitive-load/

[28] Microsoft, “About menus,” (n.d.). [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms647553(v=vs.85).aspx#bars_menus

Appendix A

A.1 JSON file - Example Configuration/driver file

Listing A.1: driver.json

```
{
    "Global": {
        "Name": "TestRun",
        "BookkeeperTolerance": 0.0
    },
    "Optimizer": {
        "Type": "Compass",
        "Mode": "Maximize",
        "Parameters": {
            "MaxEvaluations": 10,
            "InitialStepLength": 50.0,
            "MinimumStepLength": 1.0
        },
        "Objective": {
            "Type": "WeightedSum",
            "WeightedSumComponents": [
                {
                    "Coefficient": 1.0, "Property": "
    CumulativeOilProduction", "TimeStep": -1,
                    "IsWellProp": false
                },
                {
                    "Coefficient": -0.2, "Property": "
    CumulativeWellWaterProduction", "TimeStep": 10,
                    "IsWellProp": true, "Well": "PROD"
                }
            ]
        },
        "Constraints": [
            {
                "Name": "PROD-BHP-1",
                "Type": "BHP",
                "Well": "PROD",
                "Max": 3000.0,
                "Min": 1000.0
            },
            {
                "Name": "INJ-SplinePoints-1",
                "Type": "WellSplinePoints",
                "Well": "INJ",
                "WellSplinePointsInputType": "MaxMin",
                "WellSplinePointLimits": [
                    {
                        "Max": [40.0,10.0,40.0],
                        "Min": [0.0,0.0,0.0]
                    },
                    {
                        "Max": [40.0,40.0,40.0],
                        "Min": [20.0,40.0,20.0]
                    }
                ]
            }

        ]
    },
    "Simulator": {
        "Type": "ECLIPSE",
        "ExecutionScript": "csh_eclrun",
        "Commands": ["tcsh -c \"eval source ~/.cshrc; eclrun eclipse
   \""],
        "DriverPath": "../../examples/ECLIPSE/HORZWELL/HORZWELL.DATA"
    },
    "Model": {
        "ControlTimes": [0, 50, 100, 365],
        "Reservoir": {
            "Type": "ECLIPSE",
            "Path": "../../examples/ECLIPSE/HORZWELL/HORZWELL.EGRID"
        },
        "Wells": [
            {
                "Name": "PROD",
                "Type": "Producer",
                "DefinitionType": "WellBlocks",
                "PreferedPhase": "Oil",
                "Heel": [1,1,1],
                "WellboreRadius": 0.75,
                "Direction": "X",
                "WellBlocks": [
                    [1,4,2],
                    [2,4,2],
                    [3,4,2],
                    [4,4,2]
                ],
                "Completions": [
                    {
                        "Type": "Perforation",
                        "WellBlock": [2,4,2],
                        "TransmissibilityFactor": 1.0
                    },
                    {
                        "Type": "Perforation",
                        "WellBlock": [3,4,2],
                        "TransmissibilityFactor": 1.0
                    }
                ],
                "Controls": [
                    {
                        "TimeStep": 0,
                        "State": "Open",
                        "Mode": "BHP",
                        "BHP": 2000.0
                    },
                    {
                        "TimeStep": 50,
                        "State": "Open",
                        "Mode": "BHP",
                        "BHP": 2000.0
                    },
                    {
                        "TimeStep": 365,
                        "State": "Open",
                        "Mode": "BHP",
                        "BHP": 2000.0
                    }
```

```
            ],
            "Variables": [
                {
                    "Name": "PROD-BHP-1",
                    "Type": "BHP",
                    "TimeSteps": [0, 50, 365]
                },
                {
                    "Name": "PROD-TRANS-ALL",
                    "Type": "Transmissibility",
                    "Blocks": "WELL"
                },
                {
                    "Name": "PROD-WELLBLOCKS-ALL",
                    "Type": "WellBlockPosition",
                    "Blocks": "WELL"
                }
            ]
        },
        {
            "Name": "INJ",
            "Type": "Injector",
            "DefinitionType": "WellSpline",
            "PreferedPhase": "Water",
            "Heel": [2,1,2],
            "WellboreRadius": 0.75,
            "Direction": "X",
            "SplinePoints": [
                [20.0,0.0,20.0],
                [20.0,40.0,20.0]
            ],
            "Controls": [
                {
                    "TimeStep": 0,
                    "Type": "Water",
                    "State": "Open",
                    "Mode": "Rate",
                    "Rate": 1200.0
                }
            ],
            "Variables": [
                {
                    "Name": "INJ-SplinePoints-1",
                                    "Type": "SplinePoints",
                                    "VariableSplinePointIndices": [0,1],
                                    "TimeSteps": [0]
                                }
                            ]
                        }
                    ]
            }
}
```

Baumann,“https://github.com/.../Driver-file-gui/FieldOpt/.../driver.json”