Engineering Degree Project

Simulator improvements and scenario testing

Author: Lukas Gunnarsson
Supervisor: Kevin Järnbo
Lnu Supervisor: Jonas Lundberg
Semester: Spring 2023
Subject: Computer Science
Abstract

The usage of a graphical user interface (GUI) in software often make up for a great experience for the user and is often not an issue, until the only way to run a program is through a GUI. Such a dependency will make development of a project very hard as the only way to perform tests is to execute them manually. This is the case for a simulator that the company Creone uses and it is where we will perform our work. Creone works with smart key management systems and cabinets that allow for a safe and convenient way to store and handle keys. Registered users can open the cabinets with a pin code that is entered on the dial on the cabinet door. Keys are assigned to users and what keys that a user can take from a cabinet is seen on the display above the dialpad. We are to create a new core implementation that will remove the GUI dependency and allow the simulator to perform automated tests to some extent.

Keywords: software testing, test-driven development, test effectiveness, test automation, simulator, keybox
1 Introduction
1.1 Background ........................................... 1
1.2 Related work .................................... 2
1.3 Problem formulation ......................... 3
1.4 Motivation ..................................... 3
1.5 Scope/Limitation ............................... 4
1.6 Target group .................................. 5

2 Theory
2.1 Software Testing Principles ...................... 6
2.2 Design Patterns ................................ 7
2.3 Qt ........................................... 7

3 Method
3.1 Research Project ................................. 9
  3.1.1 Design Science .......................... 9

4 Implementation .................................... 13
4.1 Application .................................. 13
4.2 Classes ...................................... 14
4.3 Design ...................................... 15
4.4 CoreObject .................................. 15
4.5 Widget ....................................... 16
4.6 CoreObject and Widget ....................... 17
4.7 Functions and Macros ......................... 17
  4.7.1 GET_WIDGET .................................. 17
  4.7.2 CreateWidget ............................... 18
4.8 Communication between CoreObject and Widget 18
4.9 CoreApplication ............................... 19
4.10 Window ..................................... 19
4.11 CreoneApplication .......................... 20
4.12 Keyboard .................................... 20
4.13 CLI .......................................... 20
  4.13.1 Menu ...................................... 20
  4.13.2 Main Menu ................................. 21
  4.13.3 Test Menu .................................. 21
4.14 Test class .................................... 22
4.15 Creating tests ................................ 23
  4.15.1 UserOpenCabinet ......................... 23
4.16 Running tests ................................ 23

5 Experimental Setup, Results and Analysis .......... 24
5.1 Open cabinet with service pincode ............ 24
5.2 Registered user opens cabinet ................ 24
5.3 Unregistered user tries to open cabinet .......... 25
5.4 User opens cabinet, takes a key and closes the cabinet 26
6 Discussion  
6.1 Selection system ............................................. 28  
6.2 Test implementation ........................................... 28  
   6.2.1 Test implementation complications .................... 28  
   6.2.2 No fully automated testing ............................... 28  
6.3 States ............................................................ 28  
6.4 Singleton abuse ............................................... 29  

7 Conclusion  
7.1 Future work ..................................................... 30  
   7.1.1 Testing frameworks ....................................... 31  
   7.1.2 Custom renderer ........................................... 31  
   7.1.3 Automated builds and tests .............................. 31  

References  


1 Introduction

Testing plays a crucial role in software development, ensuring that desired quality standards are met. It helps identify defects and improve reliability which results in better performing software. Without testing, there are room for bugs and errors that can lie dormant for ages and as the project grows larger this room for error expands as well. When the issues and errors starts to pile up in the development process, the resources required to deal with them also grows larger. Good testing practices reduces the risk of having to having to place time and focus trying to fix old faulty software later in a project. Creone is a company that produces electronic key cabinets to manage keys in a safe way. To test their software they have a simulator which can be used to test new features added to their systems. The simulator cannot be run without manual input because of its dependency on a graphical user interface (GUI) and thus the ability to test their software is affected. To increase the effectivity of testing the GUI dependency is to be removed and new ways of running the simulator without manual input is to be added.

1.1 Background

As of why software testing plays such an important role in all types of different projects might be hard to see if one only is looking for the positives. To really see why software testing is required, one should start by looking at failed projects that lacked good testing practices and the damage that they caused for both involving companies and the general public. It is often when numerous bad cases have occurred that good practices is taken more seriously. Considerably one of the worst cases of where a software bug turned deadly was with the Therac-25. The Therac-25 was a linear accelerator that provided cancer patients with radiation treatment. Due to bugs in the software, six accidents [1] took place that exposed patients to extreme amounts of radiation. A few patients died of radiation sickness [2] as a result from the deadly doses of radiation being incorrectly emitted from the machine. Lack of testing and safety features was deemed one of main causes to why the accidents took place. Further issues were also present, one being how the manufacturer of the Therac-25 handled reported accidents from inside the company. Another incident where a small problem grew to a large one is with the Mars Climate Orbiter and its measurement bug [3]. The software systems on the spacecraft used different units of measurement which in turn led to its destruction because of a miscalculated trajectory.

The internet has lots of vulnerabilities due to its nature and because of this security is of utter importance. In 2014 there was a discovery of a vulnerability in the OpenSSL cryptographic library that allowed attackers to read the memory of protected systems without leaving a trace [4]. The vulnerability took advantage of OpenSSL’s implementation of the TLS/DTLS heartbeat extension. Leaks between the client and the server allows the attacker to gain chunks of memory with potentially sensitive data.

There are reasons for why downtime is such a negative thing for large companies that rely on IT services. Especially if your IT service is something that others rely on as well. In 2012 the trading firm Knight Capital Group lost $440 million dollars because of a bug in their system [5]. It should be noted that this bug was not entirely software related as there was a technician that initially caused the error, but the lack of checks and other variations of testing let this error go undetected. The bug caused Knight Capital to send millions of orders to about 150 stocks in under 1 hour. After just a little while the purchased shares were sold resulting in a loss of about $440 million dollars.
One key point to take away from all previously mentioned catastrophic events is the lack of testing or safety protocols.

Combitech is a technology consulting company owned completely by Saab [6] and has services in cybersecurity, engineering and lots of other technology areas in Sweden, Norway and Finland. The company has many clients because of its nature in consulting practices and one of those clients is Creone. Creone’s electronic key cabinets are used by big companies such as Gina Tricot, Heineken and Bosch [7]. Their simulator uses their keybox software together with Qt to simulate a real cabinet. Because the simulator is dependent on a graphical user interface (GUI) it is not possible to automate testing or any usage of the application other than with manual input via the GUI. To make testing viable there must exist other ways of automating input and data flow.

1.2 Related work

Test-Driven Development (TDD) is a development practice that puts focus on writing tests before writing any other code. By doing this the code will be of better quality and more easily maintainable. When testing is begun early in the development process, defects become easier to identify and fix. This results in less bugs and defects being present in software that reaches production. What makes TDD work is what is called the TDD-cycle. It can be summarized in three steps, fail, pass and refactor. A test is failing, implementation takes place and the test passes, when the test has passed the written implementation is looked at and if needed it is refactored. This cycle is repeated with more and more features being continuously added. The cycle is sometimes called by its three phases, namely the "Red-Green-Refactor" cycle. Another benefit of TDD that is sometimes overseen is the documentation that follows along as the cycles are iterated. The focus on separate features that are implemented makes it easier to add documentation on why the implementations exist and the role they play. The documentation is also written at the same time as the cycle iterations which can be helpful.

In a study by Janzen and Saiedian there were comparisons on traditional development practices versus TDD [8]. Three groups were created labeled "Test-First", "Test-Last" and "No-Tests" and they were tasked to implement a HTML pretty print system, a system that takes a HTML file as an input and outputs the same file in a more readable format. The "Test-First" group managed to implement 12 features, "Test-Last" implemented six and "No-Tests" implemented five. In addition to the large difference in implemented features, the "Test-First" group managed to complete a task in building a GUI that no other group were able to complete. The study concludes that the "Test-First" group spent less effort per line of code and feature than any other group and demonstrated that the practitioners of TDD produced code with fewer defects. An observation made in the study was that one potential issue with TDD is its reliance on the programmer’s discipline and motivation to go about to practice TDD seriously.

In addition to fewer defects, TDD is helpful whenever adaptability is necessary because of frequent changes or updates to requirements [9]. Because tests are written first, developers think more about the functionality of the code which results in it being of a higher quality.

Another study that makes use of comparisons is by Rod Hilton that shows that TDD is
effective at improving code quality [10]. As with the study made by Janzen and Saiedian [8], comparisons were made between projects that were written with or without TDD. This was to determine how code quality, cohesion, coupling and complexity was affected. Improvements could be seen in all three previously mentioned areas. In comparisons between the code written with TDD and code written by the traditional test-last approach, an overall improvement of 21% was found in favor of the code written with TDD. An alleged contributing factor to this improvement is due to the fact that TDD puts lots of focus on refactoring as it is a big part of the TDD development cycle. This would explain the decrease in code complexity.

With the insights from the research studies, testing and TDD can be beneficial and of use for software projects. The studies previously mentioned show evidence for higher code quality and maintainability thanks to testing and TDD.

1.3 Problem formulation

The project aims to see how feasible it is to try to update an already existing GUI-dependent application with automated testing combined with CLI functionality. One of the goals is to see how the testing aspect of the application is affected by updated code and being applied a development practice such as TDD. To what extend and the usefulness will be determined by evaluating the end results and the final implementations accompanied by tests to show that the newly added functionality can indeed be tested. Two questions help formulate the problem.

Research Question 1: What are the benefits of removing the GUI dependency from the simulator?
In order to run automated tests there must not be a GUI dependency for input manipulation. The goal is to be able to perform tests and the detachment of the GUI layer is simply just something that is preventing that from happening and thus it needs to be dealt with. The starting version of the simulator requires manual input with the use of a mouse and/or keyboard. There are several positive outcomes that will be gained from removing the GUI dependency. One is the ability to test the system autonomously and another one is a healthier codebase to allow for a more reliable way of development in the future. As the code gets less coupled and is able to expand further, the system will be more reliable and future additions to it will be easier as the entire structure of the application will be in a better state.

Research Question 2: Can the implementation of a set of runnable tests serve as enough proof to demonstrate the feasibility of the added features to the simulator?
If tests can be run without a GUI then this shows that the simulator no longer requires a GUI but can still use it if desired. The feasibility of the work done will get evaluated by analyzing the project results whilst also taking the limitations and experience gained from the project into consideration. The added features will not determine the feasibility alone. The runnable tests are a way of showing that the simulator no longer needs manual input and can be run without it.

1.4 Motivation

Removing the GUI dependency will allow for a wider range of features to be implemented because the code is not as tightly coupled and the software can get tested properly. How
testing will be conducted and how well such an adjusted system can be tested is also of
great interest and a motivator. A large application that has lots of features and functionality can not only do lots of things, it can also break in extremely many places in the code. Even if the application can have its GUI dependency removed, if the test implementation somehow suffers because of this it might not be ideal to go through with it. By evaluating the project process and how testing works in the updated simulator, the optimal ways of actually testing the system will be something to look out for. Some ways will always be better than others and finding these will be a big priority.

A simulator application without a GUI dependency will allow Creone to test out new features without being too concerned with the state of the application breaking in ways not foreseen. It is not only beneficial for the company but it also helps the users/customers of the key cabinets in areas such as user experience and safety. A group of people that could benefit from the project are those who themselves are to face a deal with a similar problem in the future.

The effect of how well a development practice can be applied to an already existing large project is also a motivator. Finding out the preferred ways to work and spotting the pitfalls.

1.5 Scope/Limitation

The scope of the project is to remove the GUI dependency in the simulator application by creating a core implementation that is separated from the GUI. The simulator is by no means a simple application, it has lots of code and functionality behind it which introduces limitations to consider.

What will be done:

1. Create a core implementation that separates core and GUI logic
2. Add a command line interface
3. Implement ways to run tests
4. Evaluate the process of removing a GUI dependency and how it affects testing

What won’t be done:

1. Create a perfect CLI implementation to the simulator
2. Testing on the GUI for things such as incorrect alignments for widgets and other GUI-related inconsistencies
3. Tests on every added feature

Some limitations:

1. No external libraries are allowed
2. Time constraints
3. No prior experience with Qt
As the simulator will get updated and new implementations take place, there will be subjectivity about how useful or necessary they are. Everyone has their own way of seeing things therefore it is very important that added software has documentation with motivation for why it exists and what it accomplishes. This also applies to code that is removed and deemed useless. At the end of the project it will be possible to go through the code and see the good and the bad parts. The newly added code will mature over the span of the project but some code will only stick around solely because of the time constraints.

1.6 Target group

With automated testing and documentation of the removal of a GUI dependency, the project will not only benefit Creone but also other developers who aim to achieve the same goal. The first CLI implementation will probably not be one that is used directly but is more to be seen as a first draft that will shine light on areas that need more attention and focus than others when removing the GUI dependency. Another group would be those who want to see the best way to test a similar system as the simulator application in this project.
2 Theory

The following sections contain information about software testing and software design in general.

2.1 Software Testing Principles

Testing of software projects are crucial to keep consistency and reliability and should be something that is continuously expanded as the project codebase grows. According to ASTQB there are seven testing principles [11] as follows:

1. Exhaustive testing is not possible
   It is impossible to cover all errors with tests and thus risk assessment is required to determine what tests are needed and where.

2. Defect clustering
   The pareto distribution is a principle and phenomena stating that 80% of outcomes are caused by 20% of the causes [12]. This also applies to software testing, 80% of errors and defects are caused by 20% of the code which indicates that a fraction of the code is causing most errors. Taking note of this is a great way to know where your attention should be targeted during testing.

3. Early testing
   Testing should be something that is done very early in a project lifecycle. This reduces the risk of having to fix defects that are hard to get rid of in the late stages of a project when they could be fixed in the earlier stages instead.

4. Testing is context dependent
   All projects differ in one way or another and the testing required and how it should be implemented will naturally also be different.

5. Pesticide paradox
   If one uses the same pesticides to kill insects, given enought time the insects will gradually develop a resistance towards the pesticide and it will not be as effective anymore. If a codebase is increasing in size and complexity, the already existing tests needs to be updated to follow the wave of change. Same tests will detect the same kind of defects and may miss out on new potential defects.

6. Tests detect defects but not their absence
   Tests shine a light on bugs and defects that might be present in the code but lots of defects go undetected because of the reasoning behind why a test is implemented. Project requirements will shift the testing attention towards specific areas of use cases and thus defects in other parts of the codebase still lie dormant.

7. Absence of errors fallacy
   Even if most bugs and defects are removed from an application and is considered almost bug-free it can still crash because of an error that somehow wasn’t detected. This makes the software unusable in the context of shipping a product or having a customer use it. This can happen if one ignores early testing and/or has been focused on testing the wrong part of the codebase.
2.2 Design Patterns

Several design patterns are used but the most obvious ones are the following.

1. Model-View-Controller
   The Model-View-Controller pattern or MVC for short focuses on separating logic into three components [13]:
   - Model, main component and is responsible for managing data and logic
   - View, responsible for display and representation of information
   - Controller, handles user input and converts it to executable commands for either the model or the view

2. Composite
   Pattern that describes a group of objects uniformly and thus treats them all as if they were of one instance of the same type of object [14]

3. Facade
   The Facade pattern is used as a front-facing interface to allow general access to functionality that is abstracted away. This makes it so that objects that use the facade does not need to know any of the complex underlying logic and can instead just make use of it [14].

4. State
   The State pattern make it so that an object can alter its behaviour whenever its state is changed or updated [14].

5. Chain of Responsibility
   Pattern to achieve loose coupling by not making a single object or class having responsibility in lots of different areas but instead to have objects do their respective part and then pass remaining commands to another object [14].

6. Observer
   Messaging pattern that allow for callbacks and other events to take place if so is desired. Works by having objects subscribe to another object that publishes messages to each of its assigned observers. The pattern can also be called the Publish-Subscribe pattern [14].

7. Command
   The Command pattern is a behavioural pattern to encapsulate the necessary data to perform a command or an action. The object that invokes or calls the command does not need to know anything about the command other than the command interface to trigger execution [14].

2.3 Qt

Qt is a cross-platform framework that provides a comprehensive set of tools and libraries to make use of [15] and it is mostly used for development of graphical user interfaces. Because of its support for multiple platforms lots of different kinds of applications can be built and run on different platforms without modifying the application itself too much. This is one of the many reasons why Qt is used as much as it is. Qt offers developers lots of standard GUI elements to make use of to create their GUI application. The elements
include text boxes, buttons and themes to name a few. The base class of all object classes in Qt is QObject [16]. It allows for the use of signals and slots which is very similar to how events and callback functions work. For GUI programming the class QWidget [17] is used which inherits from QObject. QWidget contains properties and functions that are necessary for GUI interactability, such as screen positioning, geometry sizes, coloring and visibility to name a few. The classes that Qt brings to the table can be utilized to create sophisticated applications with rich user interfaces.
3 Method

The following sections explain in detail on what methodology is used for the project.

3.1 Research Project

The research to be done will investigate the feasibility of applying testing on an already existing system. As the already existing simulator application is to be used as a starting point the method for conducting the project work is design science.

3.1.1 Design Science

Commonly used in computer science and engineering fields, design science is a research method that is used for creating solutions to problems that exist in the real world [18]. The method makes use of a set of guidelines to help dissect the problems at hand.

Problem Identification: The starting simulator application cannot operate without the use of a GUI and thus its testing ability is severely hindered. In fact, there are no tests at all. The initial simulator application suffers from two problems that intertwine, the dependency on a GUI and the lack of tests and testing functionality. The GUI dependency makes it very hard if not impossible to implement automated tests. Even if tests somehow were available, they would’ve had to spoof the GUI in one way or another which would lead to workarounds and inefficient development. The initial simulator implementation can be seen as a visual representation in Figure 3.1.

![Figure 3.1: Initial layer stack](image)

By resolving the issue with the GUI dependency, this would make the application applicable for testing in a broader range and with less constraints. The core layer is to replace the GUI layer in terms of executing simulator logic which can be seen in Figure 3.2.
Introducing a new layer that works on the same level as the GUI layer is the CLI layer. The CLI and GUI layer both act as mediums for transmitting input to the simulator and do not necessarily need to replace one another. Both layers communicate with the core layer below it which means that the simulator works the same with either one of the layers, see Figure 3.3.

The simulator is executing the exact same logic with the only difference being how the execution started, depending on how the input got transmitted in to the application. First and foremost, the GUI dependency needs to be dealt with and the introduction of the CLI will follow after that. An application with CLI support allows for automation. It helps tremendously in debugging and testing. Not only is it excellent for these purposes but it
can also make it certain that the software is indeed not faulty if it is executed on a customers’ or clients’ end if proper testing functionality is implemented. Lots of time and effort can be saved if it can be determined that the customer is actually using the software incorrectly instead of the software being broken.

Objectives: One of the primary objectives is to find out if implementing testing features late in to an existing application yields results that can motivate such an implementation for other applications and projects. To find out if it is worth the time or if it is better to simply start over from scratch. How well the existing application respond to the new changes is also of great interest. Applying TDD to a project late as it is in this case will not come without its challenges and seeing how the application respond will help tremendously in evaluating the procedure and its worth and feasability. To start implementing testing features the GUI dependency needs to be removed, therefore this is another primary objective.

Artifact Design: The major designed artifact for the project is a revamped and updated version of the initial simulator. The artifact is to include all legacy functionality of the simulator whilst also introducing new features that will allow the application to be run and tested without a GUI. The application is to be split in to two parts whereas one part takes care of core logic and the other one the GUI. The core part follows an object oriented style to make sense out of interactions and scenarios that are to be able to be run without a GUI. The CLI implementation is necessary in the core part as to make sense of input and do something with it.

Scenario tests will be created and run to showcase new added functionality and for the sake of development using TDD. An example of a scenario test and how it plays out can be seen in the following:

Example Scenario: User accessing cabinet and takes a key

1. Enters PIN code, authentication takes place
2. Opens the cabinet door
3. Picks up a key
4. Closes the cabinet door

All the simulator functionality that is present in the initial version should be present at the end of the project with the updated version.

Implementation: The implementation of the artifact is to be done in a serial manner. First and foremost the codebase is in need of refactorization which will be the first implementing step. An addition of lots of new classes that follow the object oriented paradigm to make it possible to manipulate the simulator without manual input. Existing classes with excess responsibilities are split up in to new ones to apply the single-responsibility principle [19]. Secondly, a CLI is to be added that can take input and transform it to executable commands to be passed onto the simulator. The third major implementation step is test functionality and the support for it. It is crucial that the two other steps are done before the third step is looked at. The testing part of the application should make use of the different
parts and components of the simulator to not only test the single components but rather the components in communication with other ones. Please note that the third step focuses solely on the actual implementation details of tests. Writing tests with pseudo-code is a great way to map out flaws in and absent functionality and that practice is not tied to this step.

The work done was as aligned with the methodology of design science as best could. With help of employees and engineers that had worked on the project and system previously, culprits and issues were pointed out for us. There were set goals to try to achieve in time whilst also delivering results. The start of the project was going through and reading the codebase trying to get a grasp of how the system behaved. After adding logging capabilities, system flow was easier to understand when placing trace messages in different spots in the code. After one or two weeks, the information gained was to be transformed into something concrete. The artifact design took place mostly during the refactoring phase with the way forward not obvious for us at the time being. Bugs and bad code would bring us back from the implementation to the artifact design many times during the project.
4 Implementation

The following sections explain in depth about implementation details of the project.

4.1 Application

The simulator application has been split into two major classes, CoreApplication and Window. The CoreApplication class is responsible for all logic concerning the simulator, including the communication with the Keybox software. The class diagram and structure for the application can be seen in Figure 4.4.

The refactorization of the simulator moved the GUI code to Window and kept all remaining logic in CoreApplication. It is therefore possible to run the simulator application with or without a window. The Window class can be interpreted as an extension on top of the core implementation to visualize the runtime of the application and to allow input with mouse and/or keyboard.

Figure 4.4: Partial class diagram of the simulator application
4.2 Classes

The simulator is using a couple of classes to simulate real life use of a cabinet. A class diagram that display the classes and their relations can be seen in Figure 4.5.

Driver classes communicate with the keybox software to make the simulation work as to
replicate how it works in the real world. There are lots of drivers in the application but the drivers that are of most interest are the CAN driver, keyboard driver, door lock driver and the door sensor driver. Not too much time was invested in updating the driver classes as they were working as intended.

The core implementation can itself also be divided into two parts. One part focuses on objects that are required to simulate a keybox cabinet such as the classes Cabinet, KeyList, KeySlot and KeyItem. The other part is more about manipulating these objects to trigger events such as taking or returning a key to a cabinet. The classes that are more focused on the input and manipulation part is ControlPanel, Display, Keyboard, Numpad and Button.

4.3 Design

With no prior experience with the simulator nor Qt and taking the time constraints into account, creating an entirely new simulator wasn’t possible. A core implementation that makes use of the simulator logic that already exists in the codebase whilst refactoring the GUI code was the way to go.

Initially, lots and lots of code was refactored to new classes and/or updated or removed. The different components that were run in the initial simulator were not entirely removed but most had their GUI logic refactored to new classes. The simulator was working at the beginning of the project and thus all components to make it actually simulate a real cabinet were already present in the codebase.

The initial classes got split up into two types, core and widget. The two types are paired up to work together to make it possible to run the simulator with or without a GUI. Meaning that the widget class is to be interpreted as a visual representation of the core class and its state.

4.4 CoreObject

Even though CoreObject have implementation details that use the word Widget, there is no GUI dependency. The code for the CoreObject class can be seen in Listing 1.

```cpp
class CoreObject : public QObject
{
    Q_OBJECT

public:
    virtual ~CoreObject() {} 

    virtual void CreateWidget() {}
    QObject* GetWidget() const { return *m_widget; }
    void DeleteWidget() { delete *m_widget; m_widget = nullptr; }

protected:
    std::shared_ptr<QObject*> m_widget = nullptr;
};
```

Listing 1: Implementation of the CoreObject class
An extraordinarily important observation is to be made whilst looking at the CoreObject’s member called m_widget. The member pointer type is a raw QObject to make it possible to have a relation to a Widget object without introducing any GUI dependency. To allow this, each time the Widget object is fetched from a CoreObject, it needs to be casted to its appropriate type which solely depends on the CoreObject in question. This makes it possible to always retrieve the correct widget of a CoreObject whilst also making sure that the core implementation does not introduce any GUI logic. E.g if the CoreObject is a KeyItem then the fetched Widget object needs to be cast from a QObject to a KeyItemWidget. The class names follow this pattern by appending ‘Widget’ to the CoreObject class name.

<table>
<thead>
<tr>
<th>Core class</th>
<th>GUI class</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyItem</td>
<td>KeyItemWidget</td>
</tr>
<tr>
<td>KeySlot</td>
<td>KeySlotWidget</td>
</tr>
<tr>
<td>CabinetController</td>
<td>CabinetControllerWidget</td>
</tr>
</tbody>
</table>

Table 4.1: Examples of how classes are named

All CoreObject and Widgets are named like this. A few examples of how this look like can be seen in Table 4.1.

4.5 Widget

Everything GUI related resides inside a CoreObject’s Widget class. Same as with CoreObject’s GetWidget, Widget can make use of the function GetCoreObject to retrieve its CoreObject. Code for the partial implementation of the Widget class can be seen in Listing 2.

```cpp
class Widget : public QWidget
{
    Q_OBJECT

public:
    virtual ~Widget();

    virtual CoreObject* GetCoreObject() = 0;

    template<typename WIDGET, typename FONT>
    static void SetFontSize(WIDGET widget, FONT size);
    static void SetBackgroundColor(QWidget* widget, Color color);
};
```

Listing 2: Partial implementation of the Widget class

The class has functions to modify the widget appearance, both font size and its color. One key difference between the CoreObject and Widget class is the member pointer. The Widget class does not in its base implementation define a member variable that points to a CoreObject, instead each class that derives from Widget takes care of that implementation themselves. If the GUI is enabled then there are no constraints on the code in terms of including GUI related code, so it would not make any sense to use a basic type such as QObject in the CoreObject implementation as seen in Listing 1. This means that a KeyItemWidget will have a KeyItem pointer member, a KeyListWidget will have a KeyList pointer member and so forth.
4.6 CoreObject and Widget

The two classes work together in pairs. Whenever a Widget object is needed the CoreObject can be used to retrieve it without the risk of somehow getting the wrong one. Because of the pair implementation, the way a Widget is updated is dependent on its CoreObject. The changes are made with the help of emitted signals from the CoreObject. Signals are emitted at certain events that can take place at interactions with an actual cabinet. Opening a cabinet, closing a cabinet, taking a key, returning a key, all of these interactions change the cabinet state somehow and updates are therefore required.

How and when the updates take place differs from each individual implementation. One example is whenever a key is removed from a key slot in a cabinet. In terms of code this means that a KeyItem object is removed from a KeySlot object which requires the GUI needs to accommodate for the change.

This is done by removing the icon representing a key from the KeySlot widget whilst also moving the actual KeyItem object to the KeyStash and update the KeyStashWidget to contain it. The chain of events that follow do all start with the use of a signal. In the previous example with a key being removed from a key slot, the signal KeyRemovedFromKeySlot would be emitted and the slot function OnKeyRemovedFromKeySlot would respond to it.

Every single derived implementation of the Widget class has their CoreObject member pointer named to m_assignedTo.

```cpp
std::shared_ptr<KeyItem*> m_assignedTo = nullptr;
```

This convention is done to easily make use of the CoreObject in the code when needed. The only thing that differs between the Widget implementations regarding this member is the CoreObject class that it points to which will vary.

4.7 Functions and Macros

Some functions and macros are used more than others and the most important ones are featured here.

4.7.1 GET_WIDGET

The GET_WIDGET macro works by casting this pointer of the QObject pointer type to the CoreObject’s widget counterpart class.

```cpp
#define GET_WIDGET(CORE_CLASS_OBJECT, WIDGET_TYPE) \ 
    dynamic_cast<WIDGET_TYPE>(CORE_CLASS_OBJECT->GetWidget())
```

The macro makes use of a CoreObject to retrieve the Widget object as aQObject. This is done by invoking the function GetWidget followed by a cast to the Widget class that has to be passed as the second argument in the call.
An example of how the macro is used to retrieve the Widget object of a KeyItem instance can be seen in Listing 3.

KeyItem* key;
...
KeyItemWidget* keyWidget = GET_WIDGET(key, KeyItemWidget*);

Listing 3: Usage of the GET_WIDGET macro on a KeyItem object

The macro is only used in the GUI-part of the code where a Widget object is using another Widget object. Which means that the macro is mostly present in classes that act as some kind of container, e.g KeyStashWidget or KeyListWidget.

4.7.2 CreateWidget

The general implementation of the function CreateWidget can be seen in Listing 4. Ifdefs are used for each of its implementations to remove any GUI related code when the GUI is disabled.

```cpp
void ExampleCoreObject::CreateWidget()
{
    #ifdef GUI_ENABLED
        if (m_widget == nullptr)
            m_widget = std::make_shared<QObject*>(new ExampleWidget(this));
    #endif
}
```

Listing 4: General implementation of the CreateWidget function

With this implementation there is no GUI dependency as the CreateWidget function will simply become an empty void function if the preprocessor statement GUI_ENABLED is not defined.

4.8 Communication between CoreObject and Widget

There are no other code inside the CoreObject classes that use Widgets other than in the function CreateWidget. The pair relation is initiated because of the CoreObject that invokes the Widget creation passes itself as an argument in the Widget constructor. That is the reason why this is used in the CreateWidget function as seen in Listing 4.

This implementation allows for Widgets to get updated in the GUI whenever a CoreObject changes whilst also abstracting away GUI details. One visual representation of how this is done can be seen in Figure 4.6. The Widget object is created by the CoreObject and the Widget attaches hooks onto the CoreObject and listens to it and responds to its different states and transitions to these states. The passed CoreObject in the Widget constructor is used with Qt’sQObject connect statements to make it so that the Widget takes appropriate action for different kinds of signals emitted by the CoreObject by executing specified slot functions for the signals.
4.9 CoreApplication

The class that is responsible for all core functionality is CoreApplication. It makes use of all the CoreObjects in the simulator and is a QObject implementation to make use of slots and signals. The Keybox software is run on a separate thread which makes it so that the simulator is merely observing the keybox logic and can apply input to it but is not blocking its execution. It responds to the different states available and takes appropriate action by delegating tasks to the core classes. The CoreApplication class also makes sure everything is in sync before executing new commands. Some of the checker functions that makes this possible can be seen in Listing 5.

```cpp
bool IsUserLoggedIn() const;
bool IsCabinetDoorOpen(int cabinetDoor) const;
bool IsCabinetLocked(int cabinetDoor) const;
bool IsKeyInCabinet(QString keyName);
bool IsKeyInStash(QString keyName);
bool IsTestRunning() const { return m_isTestRunning; }
```

Listing 5: Some of the checker functions in CoreApplication

The checker functions are used in the start of some core functions to make sure the requested change is actually needed and that the simulator is in a valid state to execute the request if it is possible to do so. The CoreApplication class doesn’t contain any GUI related functionality and is thus free of GUI dependencies. The CLI is managed from CoreApplication which makes it possible to run the simulator without a GUI.

4.10 Window

All GUI related code that is responsible for taking care of Widgets resides in the Window class. Even this class applies the same philosophy as CoreObject and Widget do with their pair implementation but instead it does it with CoreApplication. CoreApplication
manages CoreObjects and the Window class manages Widgets. It holds the necessary
code to create a window and renders it using Qt and it also sets up the CoreObject widgets
in its initialization as seen in Listing 6.

```cpp
for (const auto& coreObject : coreObjects)
    coreObject->CreateWidget();
```

Listing 6: Creation of CoreObjects Widget objects during the initialization of Window

The CoreObjects that have been created during the initialization stage of the simulator are
stored inside a QList which Window makes use of to invoke the CreateWidget function
for each individual CoreObject to create its Widget object.

### 4.11 CreoneApplication

The main simulator application. It has CoreApplication as a member variable and is the
creator of Window if the GUI is enabled. The class itself derives from QObject to allow it
to make use of signals to notify either CoreApplication or Window about certain required
updates or other forms of events. It does not introduce any new functionality but instead
it manages the core and GUI implementations.

### 4.12 Keyboard

To replicate input events and to path the use of the CLI and the GUI through the same
implementation the Keyboard class is used. It manipulates the Numpad class to be able to
enter pin codes and input to the simulator and this input is in turn forwarded to the driver
class CKeyboardDriver. The simulator running in either CLI or GUI mode will use this
class exactly the same.

### 4.13 CLI

The command-line interface is built around three components.

1. Input handler
2. Main menu
3. Test menu

The CLI takes input and translates it in to a command that is passed on to CoreApplication. Some of its functions can be seen in Listing 7. The functions are also present in CoreApplication and thus the CLI acts a caller or invoker of certain simulator commands. This is very similar to the command pattern as earlier described in chapter 2.

New code that the CLI introduces is not tied to the simulator but is required for it to function. Input handling for navigating the menus and simulator output in the terminal is two of the biggest features that the CLI brings to the table.

#### 4.13.1 Menu

The abstract class Menu which can be seen in Listing 8 is used to create a solid foundation
for input to be entered through the CLI and translated in to actual executable commands.
The menu uses a class called MenuItem to act as the option that can be selected. The
MenuItem class has a string for its name and a void function to be used as callback for
void OpenCabinet(int cabinetID, QString pincode);
void CloseCabinet(int cabinetID);
void TakeKey(KeyItem* key);
void ReturnKey(KeyItem* key);
void InsertKey(KeyItem* key, int keylistIdx, int slotIdx);
void SelectKeyInStash(int idx);
void GenerateNewKeys(int keys = 1);

Listing 7: CLI functions to manipulate the simulator

class Menu : public QObject
{
    Q_OBJECT

public:
    virtual ~Menu() = default;

    void AddMenuItem(const std::string& itemName,
                     const std::function<void()>& callback);
    virtual void ShowMenu() = 0;

protected:
    QList<MenuItem*> m_menuItems{};
};

Listing 8: Partial implementation of the abstract Menu class

whenever the item is selected in a menu. A menu is to be used as a way for input to
be handled by the simulator in a manner that is structured and reduces the risk of errors.
It also allows for extension of the CLI by making it possible to add more menues that
operate under the same interface. All menues act as invokers or callers to other core logic
in the simulator.

4.13.2 Main Menu

The basic functionality of the simulator such as opening a cabinet with a pincode, closing
a cabinet and locking it or taking a key is all placed inside the class MainMenu. The menu
itself can handle prompts for tasks such as selecting a key, taking a key or returning a key
to a specific keyslot position. All available menu selections in the MainMenu make use
of functions from CoreApplication.

4.13.3 Test Menu

Everything related to testing in the CLI is found inside the class TestMenu.
It operates the same as MainMenu but is more oriented towards testing, featuring basic
test runner functionality. Tests get executed after being selected as a menu alternative. A
display of how this looks like can be seen in Figure 4.8.
### 4.14 Test class

The Test class is an abstract class that requires each class inheriting it to define a function called `Execution` that is the start signal to begin running a test. The test itself needs to account for all the steps that are required to complete it. It also has to evaluate itself, if the task/action that was performed was executed successfully or not. The implementation details for the class can be found in Listing 9.

```cpp
class Test {
public:
    virtual ~Test() = default;

    virtual void Execute() = 0;
    virtual std::string GetName() const = 0;
};
```

Listing 9: Partial implementation of the abstract Test class
4.15 Creating tests

To add new tests, one must register it using the `RegisterTest` function inside `TestMenu`.

\[
\text{RegisterTest<TEST\_NAME>(TEST\_ID, \text{EXPECTED\_RESULT}, \text{OPTIONAL\_ARGS});}
\]

All tests take an ID, expected result in the form of a boolean and optional arguments. The optional arguments passed on in the registration call is entirely dependent on what test is being created and how that particular test’s constructor looks like.

The actual tests are series of checks before and after a simulator action has been executed. E.g., if a test is run to open a cabinet door then there are some checks before and after the function `OpenCabinet` from `CoreApplication`. If the cabinet door is already unlocked and open, then there is no need to call the function to open the door again.

If this is the desired behaviour, that is fully dependent on the test itself and what the expected result should be. The actual simulator functionality can be tested in terms of executing ‘real’ actions such as opening a cabinet door or closing it by the use of such a test. The code for testing the actual simulator implementation itself can also be tested with the same kind of test.

4.15.1 UserOpenCabinet

A cabinet door is to open, the test `UserOpenCabinet` is executed which uses the core function `OpenCabinet`. There are two different states that a cabinet can have in terms of its door:

1. Door is closed, enter pincode and check to see if cabinet door is opened
2. Door is open, no need to enter pincode

The second alternative where the door already is open, how that case is handled can also be a test on the implementation details alone rather than just how the keybox system responds to an action triggered by the simulator.

Using a test in such a manner could answer questions such as:

- How will the simulator react when it is told to open an already open door?
- Are the checker functions used correctly if the simulator is told to do an unnecessary task (such as opening an already open door)?

4.16 Running tests

Tests are run by calling the function `RunTest` and passing the string test ID as the argument. This function iterates through all the registered tests and when a match is found with the given test ID, the test is executed by using invoking its `Execute` function. The simulator is not explicitly aware whenever a test is running or not. It does what it is told to and is using the core implementation the same way as the CLI and GUI. The issue with running tests like this is discussed in Chapter 6.
5 Experimental Setup, Results and Analysis

The development of the new version of the simulator allows for use with a command-line as well as a graphical user interface. The added CoreApplication and CoreObjects makes it so that the GUI is using the same tools such as the CLI. Instead of the GUI directly manipulating the simulator logic it is now a view of the application and its input is now triggering core functionality. As the need for manual input is removed, automated tests can be added. At the end of the project there exist four types of tests. The tests include the following scenarios:

1. Open cabinet with service pincode
2. Open cabinet with registered users pincode
3. Unregistered user tries to open cabinet
4. Open cabinet and take a key and close the cabinet

More can be added but because of the time constraints, this could not be done in time. Simulators are very context-dependent and do all have their niche styles and implementations and this can make it difficult to find an entry to where and to what to start testing. As the tests are built on top of the simulator, the tests themselves were all run on the same machine under the same circumstances.

5.1 Open cabinet with service pincode

Expected outcome: Service list is shown

![Image](example.png)

Figure 5.9: Using the service pincode to enter service list on cabinet

The service pincode does not open the cabinet the same way as a registered users pincode does. It is used for service purposes which doesn’t involve getting access to keys. This means that the expected result for the test to be deemed successful is to be in the ServiceList state whenever the test is evaluated.

5.2 Registered user opens cabinet

Expected outcome: Cabinet is unlocked and door is open

The test will get the currently selected cabinet and try to open it with a registered users pincode. The passed pincode is that of a registered user so the cabinet is expected to be unlocked and open.
5.3 Unregistered user tries to open cabinet

Expected outcome: Cabinet stays locked

An unregistered user is equivalent to ALL the pincodes that do not have a user assigned to them. Therefore an unregistered user is also interpreted as an incorrectly entered pincode by the simulator. The reason for this is because a user is identified by their pincode and there is also no way of selecting a user and then trying to log in.
5.4 User opens cabinet, takes a key and closes the cabinet

Expected outcome: Targeted key is taken from the cabinet and the cabinet is locked

The test successfully opens the cabinet door, takes the targeted key and closes the cabinet and locks it.
6 Discussion

Most of the core functionality has been covered but the work done is not perfect. There are some parts that we would like to change or remove which be covered later in this chapter. One aspect of the project that was hard to cover was the use of TDD. As mentioned before, applying TDD is one thing if its on a project that is new or if it has been using it before. Applying TDD on a large codebase such as what we’ve worked with is a challenge to say the least. With no experience of the practice it was even harder. To use and get use of a development practice it is crucial that everyone that is working together on a project are all in one way or another applying the practice. If one person is using TDD and the rest is writing production code the usual way, the expected results should not be too high. The reason for why it is used is because you want high quality code that is testable. The untested code and the code that wasn’t written with TDD then becomes your problem in a way if you really want to apply TDD. If you constantly are rewriting code others have produced then you are limiting yourself and what you can bring forth in a project. Now if TDD is applied by all participants on a project, instead of looking backwards you are now looking forwards and can work to solve the actual problems that are of your concern regarding your project.

In our experience, the use of TDD was very limited and constrained. Even if we would’ve had experience with TDD before the project it wouldn’t have made a very big difference as the limitations would still be there because of the existing codebase. As seen with the study by Rod Hilton [10] the projects used to decide whether TDD was effective or not were not of the same nature as our project. The projects where TDD was applied started with the practice from the beginning. There were no large projects that started using TDD in all of a sudden. In our estimation to get the best out of TDD and maximize its benefits, it is required to start with it from the beginning of the project.

With that said, it was mostly when writing the CLI and the test menu that a workflow that resembles TDD could be somewhat seen. The tests that were written highlighted required parts of the simulator that didn’t exist and the implementation took place in a fashion very similar to the TDD cycle. In the industrial case study published by Henrike Barkmann, Rüdiger Lincke and Welf Löwe [20] it is mentioned that there are cases where TDD simply is not suitable. One example from the paper is when it is technically impossible/extremely difficult to test the code using unit tests. Another example is when the time required creating a test is not in relation with manual verification of a feature, in respect to the complexity of the implemented feature. Testing GUI features is incredibly hard and thus it is often the case that manual verification is the way to go when dealing with such features. The use of TDD can increase the initial required development time as seen in a study by Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat and Laurie Williams [21]. Their study was conducted at three teams at Microsoft and at another team at IBM. What they found was an increase between 15-35% in the initial development time spent. A large and complex codebase can therefore be a problem that TDD might not be able to solve. A problem that we encountered during our work. Please note that the teams at Microsoft and IBM are of high skill and competence and of a level of knowledge that we haven’t reached yet. For us the time increase would be larger than 15-35%.
6.1 Selection system

The core application has a selection feature that is used a bit in the GUI as well. Its intended purpose was to manipulate CoreObjects in the simulator that could be interacted with in the same manner as a mouse does when it clicks on widgets. The core application was to be able to point at certain CoreObjects before performing an action on them. The reason being that the selection system was to mimic that of a mouse and thus being able to be used by the mouse in the GUI as well as being used in the CLI. Even if it is used in CoreApplication its implementation is not that great and its quite obfuscated.

6.2 Test implementation

As no external testing framework was used for the testing aspect of the new implementation, tests are run internally in the simulator. As we’ve never used any C++ testing frameworks before and given the restricted time, using a framework such as Cucumber or GoogleTest was ruled out.

It only took one single scenario test to realize that the way tests are handled is not that straightforward. Logging and constantly checking states makes it very hard to write a simple test for a single action such as opening a cabinet and closing it. The code is very cluttered and because if statements are needed to check for states, the indentation makes things even harder to follow.

Most of the trivial work to be carried out could be removed by using a framework. Not only would it simplify the process but the available testing frameworks are well developed and have lots of functionality baked in to them. By not using one there are some issues that are problematic. The reason for why testing turned out this way was because there had to be something that could show that it is possible to test the application.

6.2.1 Test implementation complications

Because the tests are run in the simulator, if a test would fail so miserably that the entire application crashes, the test itself would not be evaluated because the application running the test is no longer running. If multiple tests are set to run then there could be lots of problems if the software is in a very early development stage where lots of bugs are present. Such behaviour should not be allowed to occur. If a test causes a crash, it should be marked as failed and the testing should move on to the next.

6.2.2 No fully automated testing

Tests can be run but there wasn’t enough time to add functionality to have the simulator parse arguments passed through the command line and to run tests based on the parsed data. Not having this feature implemented kind of defeats the purpose removing the GUI in the first place because input is still required to trigger the tests. There is a very light implementation of this kind of test running in the codebase but there were some issues with the threads and the keybox software being fully initialized before tests were executed.

6.3 States

The keybox application uses lots of states. The simulator does not require the use of all off the available states but only some of them, at least as of now. Thus there is a set
of states in the CoreApplication that mimics the selected keybox states that are of use. Therefore the states need to be in sync for the simulator to work as intended and this can be problematic. A better solution would be to have the CoreApplication be an observer of the class that is responsible for delegating and notifying new states. The need for syncing states would no longer be required and the simulator would be easier to expand in terms of implementations regarding states and the potentially necessary actions that are associated with them. Requiring syncing of states at multiple locations at once is something that should be avoided at all costs.

6.4 Singleton abuse

The singleton pattern is used excessively in both the simulator and keybox codebase. As the project had to start somewhere and given the time constraints, keeping momentum was of high priority. Changing too much of the existing code and already existing design would hinder progress and it would also be incredibly difficult to make things work. It is not necessarily a bad thing to use singletons but it is when its use exceed a certain threshold that one might start to suspect that there is some flaws in the system design. There are places in the code that makes use of CoreApplication in a way that just feels counterintuitive. This is somewhat more of an observation of the entirety of the whole application without having any real solution to the problem itself. One might argue that if it works then it is not that problematic but it is something worth taking note of.
7 Conclusion

The goal of the project was to remove the GUI dependency and implement testing whilst also evaluating the best way to go about testing an application such as the simulator. The new version of the simulator assigns responsibility to each individual core object to take care of itself and its widget. By doing this, updates on certain components will never get missed by the GUI if it was to be used whilst also allowing the CLI to be run completely without any dependency on graphics. The philosophy of having a CoreObject and a Widget could be applied to other projects in the future that are to deal with the same types of problems as faced here. This is especially relevant for similar projects that also use Qt.

The TDD approach was not entirely straightforward to apply to the project as the existing application was not designed using TDD or any other similar testing practice. A majority of the simulator code had to be refactored and updated before the application could be deemed ready for development using TDD. It was not until late in the project that tests were used to pave the way for what was necessary in terms of functionality.

To make testing efficient for a similar application one should make use of a testing framework such as GoogleTest or Cucumber instead of writing the test implementation entirely by your own. The project’s scenario test implementations are quite hard to follow and the creation of new types of tests results in lots of duplicate code. If a test causes a crash then the test should be marked as failed and the next one should start to get executed. That is not the case here as the tests are run entirely by the simulator application itself. The testing environment should be isolated. The application must also be in such a state that a development practice can be applied to it. If the project is entirely new and fresh then there are no restrictions, but with an already existing application that has its flaws and dependencies, then it starts to get more difficult. From our experience, we would advice to begin with TDD at the start of the project because as the project matures it will get increasingly more difficult to start using it. If the base implementation isn’t aligned with the development practice and philosophy then there is a high risk that future implementations are not reaching their full potential. There is also risk of confusion if everyone is working differently.

As the simulator wasn’t created with a core implementation initially, the work done in the project is far from perfect. After spending lots of time with the simulator there is one major key point to take away, write tests early in the project. Writing tests helps very much in covering areas in the codebase that are missing functionality. The tests will shine a light on where there is a need for further implementation and they help in keeping focus on adding the required necessities to the application. Having runnable tests early in to a project is also terrific, creating a stable structure for the application to be continuously developed upon. Even if the entire project doesn’t apply TDD as its development practice, having tests that cover most functionality is very beneficial. Both for ensuring that the application works as intended and that future implementations follow the same structure as the previous ones.

7.1 Future work

Some of the things we would’ve like to have done if given enough time would be:
7.1.1 Testing frameworks

The way testing works is something that we wish we would’ve gotten more time to try out things with. With the current implementation, tests do work but there is no way to create tests other than internally in the codebase. No scripts, no external framework such as Cucumber or anything remotely similar exist to create tests. A simple way to allow tests to be created and evaluated is something that we would’ve liked to see being done. The current implementation is more somewhat of a proof of concept that something can be done with tests but a better solution would be to use a well developed and maintained testing framework to evaluate tests. More information about the test implementation and its flaws can be found in chapter 6.

7.1.2 Custom renderer

When starting the project, one of the first things that we experienced was the fans going 100% when running the simulator. This is probably because of inefficient rendering and constant polling that forces the GUI to render each frame. A custom renderer that encapsulates all render functionality and provides an easy to use API to draw things in the GUI or trigger popups or whatnot would abstract some of the render implementation away from the rest of the code. If a well designed renderer is written this could also prove beneficial in terms of performance with draw calls being optimized. The renderer would not necessarily be placed in the simulator but rather in the Keybox system itself. Doing this would allow functionality to be implemented to totally remove all draw calls being submitted when running the simulator in CLI or headless mode. The benefits would not only be performance and more loosely coupled code, but also help in setting up automated builds and testing on the cloud using CI/CD pipelines by completely having the render element be removed.

7.1.3 Automated builds and tests

As mentioned before, an improved renderer would allow for an easier setup to run automated builds and tests on the cloud with CI/CD pipelines. Not too much time has been dedicated for build systems combined with testing rather than just running the simulator locally but having a headless simulator be run on a CI/CD pipeline would be great. Adding such a feature is not within the scope of the project and it would alone take a tremendous amount of time and effort.
References


