Bachelor's Thesis

Turku University of Applied Sciences

Information and Communications Technology

2024

Eemeli Elgfors

Developing a Contract Test Preparation Application

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 48 sivua

Eemeli Elgfors

Sopimustestien valmistelusovelluksen kehittäminen

Tämän opinnäytetyön tarkoituksena oli luoda sovellus, joka pystyy socket-kommunikaatioon Lashmate-sovelluksen kanssa viestintäprotokollan kautta, ja siten pystyä etsimään ja esittämään virheitä Lashmaten suorittamissa laskelmissa. Sovelluksen ideana oli vähentää Lashmaten virheiden etsimiseen kuluvaa aikaa ja säästää Lashmate-kehittäjien aikaa.

Opinnäytetyön sovellus kehitettiin pääasiassa C#- ohjelmointikielellä. Sovellukseen vaaditut päätoiminnot jaettiin vaiheisiin, jotka oli suoritettava, jotta sovellus toimisi kunnolla. Sovelluksen testaus suoritettiin tarkkailemalla sovelluksen tuottamien raporttien tarkkuutta ja sovelluksen yleistä suorituskykyä. Lashmate-kehittäjien aikarajoituksista johtuen alkuperäinen sovelluksien kommunikaatiotapa ei ollut tässä opinnäytetyössä mahdollinen, joten opinnäytetyötä varten luotiin kommunikaatioprotokollasta imitaatio. Tästä johtuen opinnäytetyön raporttien tarkkuustestaus testaa vain, että sovellus tulostaa imitaatioprotokollan esiasetetus tiedot oikealla tavalla.

Sovelluksen testauksesta kerätyt tiedot olivat pääosin odotuksen mukaisia. Opinnäytetyösovellus onnistui tuottamaan oikeat arvot imitaatioprotokollasta ja sovelluksen ohjelmointitoimintojen suorituskyky oli riittävä. Tiedon määrän vuoksi sovelluksen käyttöliittymä kohtasi vakavia suorituskykyongelmia.

Vaikka lopullinen sovellusversio ei pystynyt muodostamaan oikeanlaista socket-kommunikaatiota Lashmaten kanssa, pystyy opinnäytettyösovellus silti tuottamaan haluttuja tietoja, joita tarvitaan sovelluksen jatkokehittämisessä.

Asiasanat:

Virheraportti, Ohjelmavirhe, Konttikuljetus, Ohjelmisto

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and Communications Technology

2024 | 48 pages

Eemeli Elgfors

Developing a Contract Test Preparation Application

The objective of this thesis was to create an application capable of socket communication with the Lashmate application through a communication protocol, and thus be able to seek out and display errors in the calculations Lashmate performs. The purpose of the application was to decrease the time it takes to seek out errors in Lashmate to save the Lashmate developers’ time.

The application was developed using mainly C# programming language. The required core functionalities of the application were divided into steps that had to be completed for the application to run properly. The testing of the application was to be carried out by observing the accuracy of the reports that the application output, and by the overall performance of the application. Due to the Lashmate developer’s time constraints, the original intended method of application-to-application communication was not possible in this thesis, so an imitation of the communication was created for this thesis specifically. Due to this, this accuracy testing of the thesis only tests that the application is properly outputting the pre-set data from the imitation protocol.

The data collected from the testing of the application was mostly as expected. The thesis application was successful in outputting the correct values from the imitation protocol, and the overall performance of the methods inside the application was sufficient. However, due to the amount of data output, the user interface encountered serious performance issues.

Although the final application version was unable to establish proper socket communication with Lashmate, the development of the contract test preparation application outputs the desired data needed for further development.

Keywords:

Bug report, Error, Container shipping, Software

Contents

Figures

Tables

List of Abbreviations

API	Application Programming Interface
CFBF	Compound File Binary Format
DLL	Dynamic-link Library
IDE	Integrated Development Environment
LMUD	Stowage Plan File
LMUI	Container Ship Definition File
MSI	Microsoft Installer
TEU	Twenty-foot Equivalent Unit
UI	User Interface
VS	Visual Studio
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

1 Introduction

A software bug is an error, flaw, or unintended behavior in a computer program or software application. Software bugs manifest themselves in many ways and are often a detriment to the application they are found in. In larger applications the amount of code can exceed tens of thousands of lines, so the specific cause for a bug can be hard to find. Therefore, a large amount of software developers' time is spent in finding the cause for the software bugs.

Software bug issues are unavoidable in software development and maintenance (Zhou 2020). In software development bugs can appear in multiple separate phases of the development process, with the most common phases being development or coding. Harmful bugs are fixed before the release of the application to the consumers, but sometimes more detrimental bugs can pass through the testing process due to time constriction or because of an oversight from the testers. Updates on the application carry out this same risk as well. Depending on the severity of the bug this can have a large monetary impact on the development company, so the time it takes to fix the bug is better kept at minimum.

Software testing can take up to half of the resources of the development of new software (Arcuri 2008). Manual bug finding consumes more time compared to automated testing. An automated test environment removes the manual aspects around bug finding, and it ensures that all the processes involved in and around software development testing are seamless, joined up, and fully integrated. The benefit of automated testing is the consistent results as well as saved time for the developers.

1.1 Objective of Thesis

The thesis' objective is to create a working contract test preparation application for MacGregor, which is part of the Cargotec Oyj company. The test application works in tandem with the company's own application Lashmate. Communication between the contract test preparation application and Lashmate is conducted through socket communication. Socket communication is essentially a mechanism of communication between processes (Haiping, 2019).

The development of this application is important to Macgregor as it saves time on finding the bugs manually. Lashmate is an ongoing project in MacGregor, and it is constantly being updated due to the consumer demand of the Lashmate application. These updates may cause unforeseen bugs in the application that hinder the usage of the application to some consumers. Finding the bugs as fast as possible is particularly important.

This thesis's application will be developed mainly using Visual Studio. Visual Studio will be used to create the application, with C# as the programming language. The application also consists of multiple DLL's that are connected to C++ programming language snippets that allow communication between thesis's application and Lashmate. There are two main parts in the script, one is to assert the connection between the applications and the other is to process the information collected through the connection and present it to the user in an easily readable format.

1.2 Scope and Deliverables

The contract test preparation applications scope is too broad for this thesis, so the thesis scope has been scaled down to a more appropriate level. Creating proper bug test reports would entail an automatic installation and proper handling of older Lashmate versions which cannot be driven programmatically. This process would consist of significantly more work than is possible to deliver in this thesis. The output of this thesis is to deliver a working script that creates a file

that compares results from unchanged internal test cases to newly performed test cases.

Figure 1 shows the flow map of a working thesis application. The dark green area entitles Contract Test Preparation Application covers the scope of this thesis. The light green elements depict the next steps of the application development process. The Lashmate printer driver client and Lashmate proxy CONSORT protocol are already developed protocols that the thesis application will utilize.

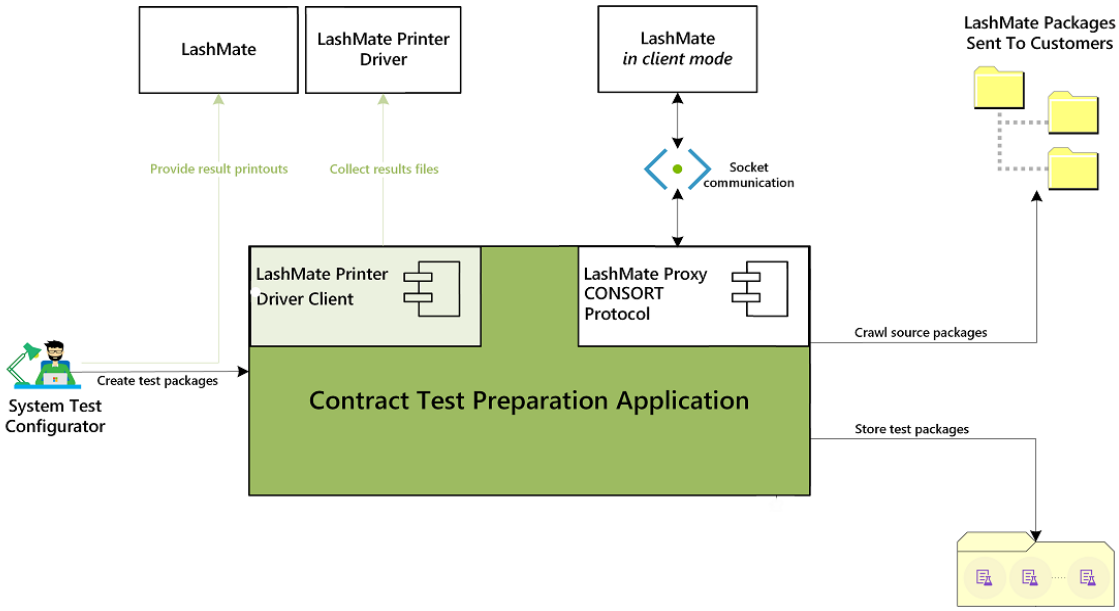

Figure 1. Contract Test Preparation Application flow map.

1.3 Structure of Thesis

This thesis consists of 6 main chapters. The first chapter is an introduction chapter meant to introduce the reader to the thesis subject, objective, and scope. The second chapter includes the bulk of the theory relating to the thesis. This chapter contains basic information about container shipping, containers, appropriate tools of the industry relating to the thesis, and the physical forces that containers are subjected to. The chapter also explains some essential key words relating to container positioning on the ship. Relevant technologies and tools used

in the thesis are also briefly described in this chapter, although some of them will be touched upon in later chapters.

The third chapter aims at explaining the methodologies and practices used in the thesis application’s development. This chapter explains the philosophies of user interface design decisions and goes through the intended functionality pipeline of the application and its main functions and methods.

The fourth chapter includes the methodologies used in testing the thesis applications functionality and performance. The fifth chapter intends to highlight the test results from the application's testing and discuss the results.

The last chapter is the conclusion of the thesis. This chapter summarizes the key findings of the study, describes the limitations and problems that the testing methodologies caused, and recommends improvements for the testing method.

2 Theory

This chapter contains basic information about container shipping, containers, appropriate tools of the industry relating to the thesis, and the physical forces that containers are subjected to. The chapter also explains some essential key words relating to container positioning on the ship. Relevant technologies and tools used in the thesis are also briefly described in this chapter, although some of them will be touched upon in later chapters.

2.1 Container Ships, containers, and lashing equipment

MacGregor specializes in equipment in cargo and load handling on container ships. As the thesis application is related to the Lashmate application used in cargo handling, a review of basic container ships and container information is required.

Container ships have grown significantly in size in over the last 20 years, in 2002 a large container ship would be able to carry approximately 6,500 TEU, today the largest containerships can now transport 24,000 TEU (International Chamber of Shipping s.a.). For such rapid growth in the industry, specialized equipment used to load and transport such substantial amounts of cargo is needed. The equipment often consists of twistlocks used in combination with turnbuckles, [lashing] rods (Andersson 1997).

A twistlock (Figure 2) is a mechanical locking device attached to the corner of a container. Twistlocks and latchlocks locate and secure containers either to each other, withing a stack, or to the transport mode (Andersson 1997). A twistlock can be locked either manually or automatically, depending on the twistlock type being used.

Figure 2. C5AM-DF semi-automatic twistlock produced by MacGregor (MacGregor container securing systems product catalogue 2016).

Lashing bars (Figure 3) is one of equipment safety used for fastening container when cruise which will be connected with turnbuckle (Australian Standard, 2001). A lashing bar is a solid steel rod that is inserted into lashing points on containers, and they connect multiple containers to the ship's deck. Turnbuckle is a tensioning device which in one end fits to the bottom part of a lashing rod and in the other end to the structure of the transport mode (Andersson 1997). Turnbuckle is used to adjust and maintain the required tension needed for the lashing bars to secure the containers properly. Turnbuckle's central frame can be turned, applying, or relieving tension in the securing system.

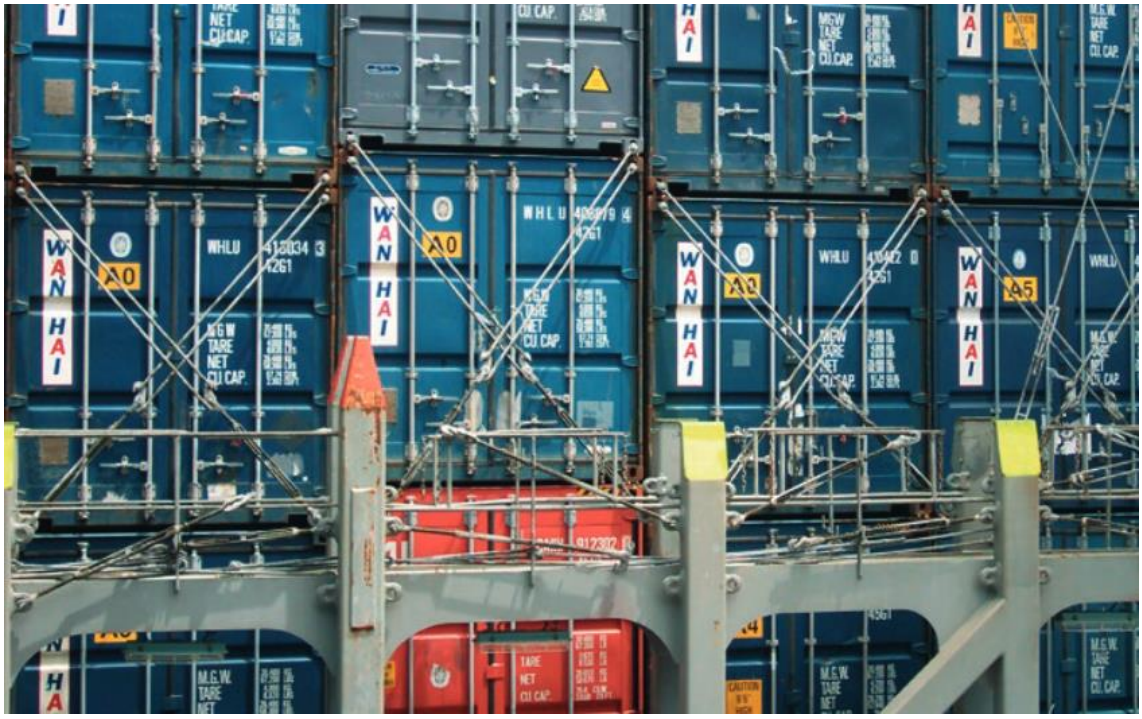

Figure 3. Lashing bars and turnbuckles used to secure container stacks into a lashing bridge (MacGregor container securing systems product catalogue, 2016).

2.2 Forces applied to containers

Lashing systems and equipment are necessary, because of the forces that are applied to the containers and other cargo while the container ship travels through the seas. Safe sea transport of containers stowed on weather deck requires that containers and their lashing gear withstand extreme forces caused by gravity, ship motions in waves, and wind-induced pressure loads acting on outer container stacks (Wolf & Rathje, 2021).

As stated by the German Insurance Associaton in their Container handbook (2002), a container ship exhibits two types of dynamic motions, linear motion, and rotational motion. Linear motion includes ship's motion along longitudinal axis called surging, motion along the transverse axis called swaying, and motion along the vertical axis called heaving. Rotational motion includes ship's motion around

the longitudinal axis called rolling, motion around the transverse axis called pitching, and motion around the vertical axis called yawing.

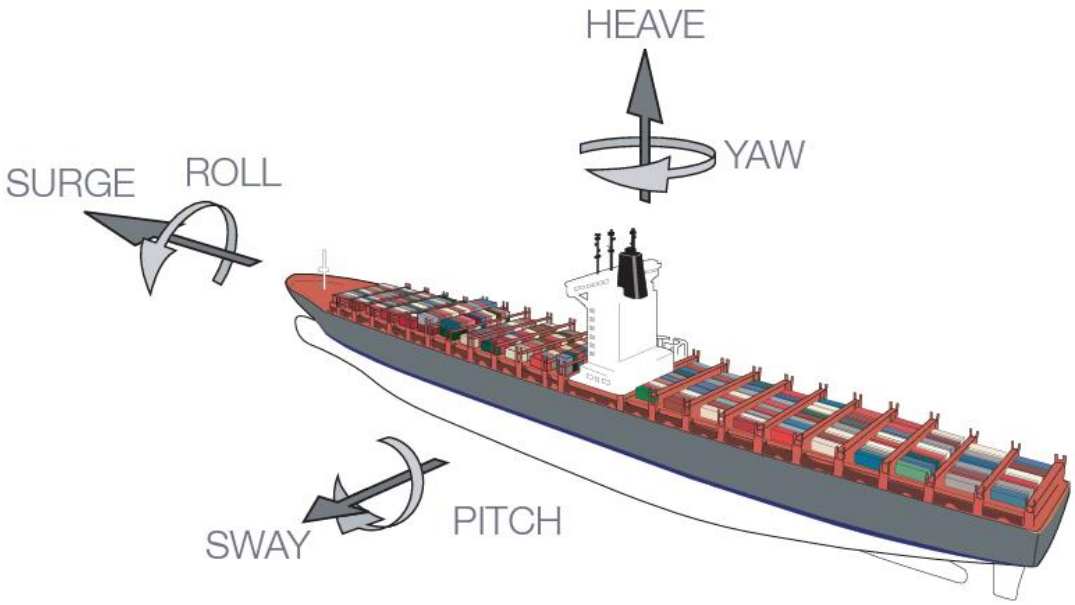

Figure 4. Container ship's linear and rotational motion depicted visually (MacGregor container securing systems product catalogue 2016).

Other forces affecting the container ships also include the container ship's acceleration and deceleration, torsion forces created by the ship's linear and dynamic motions, as well as environmental forces applied to the containers. All ships experience air and wind resistance while under way at sea (Andersen 2012).

2.3 Container Placement Measurements

The methodical stacking and placing of containers on a vessel are an important aspect of cargo management. The suitable arrangement of containers is directly related to their placement in the vessel (Firooz 2022). The stowage position on board container ships is generally documented according to the bay-row-tier system (COP for Packing of CTU 2012). The "tier" represents the vertical stacking

of containers on top of each other. The tiers are the layers of containers, numbered from the bottom and up. The “bay” refers to the longitudinal sections of the vessel, with containers arranged side by side within a bay. Bays run perpendicular to the ship's length and are typically numbered from bow to stern. Lastly, the “row” refers to the alignment of containers along the length of the ship. Rows are numbered from the middle of the ship outwards, with even numbers on the port side and odd numbers on the starboard side. The precise coordination of these tier, bay, and row measurements optimizes cargo distribution, enhances stability, and facilitates efficient loading and unloading operations on container ships.

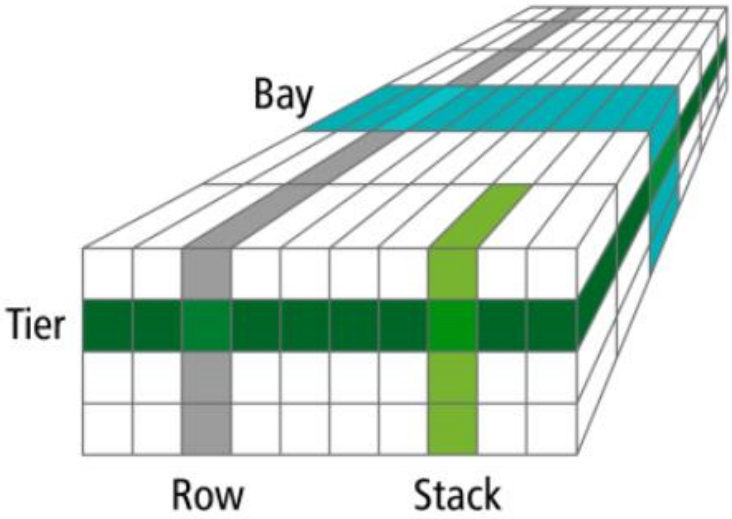

Figure 5. The primary measurement for container placement (Grundmeier 2016).

2.4 Lashmate

As explained in the previous chapter, containers on top of an actively moving container ship are being constantly affected by various forces of different power and direction. Pre-emptively measuring, calculating, and predicting these various forces and their breaking points is especially important for shipping companies across the world.

A stowage plan is made for every single voyage a container ship takes moving cargo across the globe. A stowage plan is one where the size, weight, and destination of the containers are considered for how they are placed inside or on top of the container ship. Objectives for an optimal stowage plan is to maximize the available capacity of the ship to bring in as much cargo as possible, to protect the ship and the cargo, to provide for rapid and systematic discharging and loading of the cargo, and to adhere by global standards and regulations of cargo shipping.

Modern stowage plans are executed by computer programs using mathematical calculations (Azevedo et al. 2013). MacGregor's stowage plan calculation software is called Lashmate, which is the software that the contract test preparation application supports. Lashmate is one of many commercial stowage plan calculation software options in the world. Lashmate works by using container ships profiles and stowage plan export files, which are special files created and used internally by MacGregor. A new container ship definition profile is created for each individual ship using the Lashmate program. This ship definition profile is called LMUI, and it contains the specific ship's size, model, name, and container slots inside container ship's cargo holds and on top of the deck. The stowage plan export file is called LMUD, and it is a digital version of the stowage plan for a specific voyage. Lashmate utilizes both LMUI and LMUD to calculate the lashing forces for the entire ship's lashing system and actual loading cases of the cargo. The software gives out warnings if excessive forces on the containers or lashings are detected. These excess forces are calculated by using different shipping classification standards that the ships must abide by. Lashmate can also suggest optional stack distribution methods, if applicable for the situation.

2.5 Relevant technologies

This chapter discusses and explains the development environment used to create the thesis application, as well as the relevant methods, protocols, and coding languages used in it. Only a generic overview of the code is given in this chapter, as the specifics of the code are discussed more thoroughly in later chapters.

2.5.1 Visual Studio

Contract test preparation application is developed and programmed inside the Visual Studio. Visual Studio is an integrated development environment (IDE) that can be used to write, edit, debug, and build code (Microsoft Learn 2023a). developed by Microsoft and widely used for computer program development. Visual Studio can be used for free with a community license, but the thesis application is developed using a business license. Visual Studio supports development with many different programming languages, which the thesis application utilizes.

To allow the usage of multiple programming languages in one project, Visual Studio uses Dynamic-link Libaries, also known as DLL. A DLL is a library that contains code and data that can be used by more than one program at the same time (Microsoft Learn 2024). DLLs are used to share functions and methods from one program to another for a more streamlined and performance-oriented program. DLLs can use a different programming language from the main program, with the integration between the two languages being handled by Visual Studio.

2.5.2 Programming languages

Contract test preparation application uses multiple different programming languages for its different methods and protocols. The thesis application is developed by using C#, while more memory-heavy functions are handled with C++. Extensible Application Markup Language (XAML) is used to create the application's UI.

The main programming language used in the thesis application is C#. C# is an object-oriented high-level programming language that runs on the .NET framework (Microsoft Learn 2023b). In the contract test preparation application C# is used on the creation of basic functionalities in the UI, file crawling, data collection, bug test comparison, and test report output. Most of the functions and methods used in the application are written in C#, as the intuitive language design and automatic memory allocation makes the language versatile.

WPF is used to create the thesis application's UI. WPF framework is implemented inside the .NET framework, and so is well suited for the thesis application. WPF uses its own programming language for UI building called XAML. XAML is a declarative language (Microsoft Learn 2023c) based on XML and is structurally like the JavaScript programming language. The thesis application uses XAML to create its controls, graphics, and different data binds between UI elements and the C# code.

C++ is a high-level programming language with manual data allocation. C++ is used in the thesis to crawl, open, decrypt, and finally parse LMUI files. The method of reading LMUI files proved to be too complex with C#, so C++ was used for this reason. The C++ application was shared to the main application via DLL.

2.5.3 Compound File Binary Format

CFBF, also called Compound File, is a compound document file format for storing numerous files and streams within a single file on a disk (Microsoft Corporation 2008). Key features of CFBF include its structured file storage model (Microsoft Learn, 2021), data streams that can contain several types of data, and property sets, an attachment file that can store information about the data it attached to. CFBF is a memory-efficient way to store important data of applications, and in the case of the thesis application, LMUI files are CFBF files.

2.5.4 CONSORT

CONSORT is a communication protocol that is used to connect Lashmate with the contract test preparation application. To achieve communication between applications, Lashmate must first be installed via MSI administrative installation. After this the CONSORT protocol runs Lashmate in the CONSORT server mode, which drives the application programmatically. With Lashmate in this mode, CONSORT protocol provides communication between the applications through socket communication. Bidirectional stream socket communication is used to control applications and for data exchange. This process allows contract test preparation application to query test results from Lashmate using the LMUD and LMUI files. These results can then be accessed by the thesis application and the comparison between Lashmate results and thesis applications expected results can be performed.

3 Development

This chapter delves into the designing, programming, and development of the contract test preparation application. The codebase and the methods and functions used will be explained more deeply.

3.1 Designing the Application

Before starting the application’s programming, a well-thought-out design is recommended. The design of an intuitive and easy-to-use application requires a good UI design and a streamlined use process. The application must be as easy to understand as possible without further instructions but also must consider the intended userbase of the application. The contract test preparation application will only be used internally in the company, so the overall design can be more lenient in its operation explanations.

3.1.1 Functionality Design

The first step in designing the contract test preparation application is to define the application's functionality. Without first defining the operations the application is supposed to perform the UI design and programming cannot be optimally performed.

The contract test preparation application has several core operations that it must carry out to satisfy its functionality requirements. These core operations have been requested by the company where this thesis’ application is going to be used. These are the core functionalities that are required to be in a working condition for the application to be usable in its first design:

1. The user must be able to select the Lashmate installation folder for processing.

2. Lashmate installation folder must contain the required LMUI and LMUD files, as well as the executive file for Lashmate installation. The application must check if the folder meets these requirements.
3. For installation folders with multiple LMUI and LMUD files: an option to mark the files as pairs for further processing must be present.
4. Ability to start the processing after all is set up correctly. The user interface must be responsive during lengthy operations and the abort processing command must be available.
5. Processing status must be reported. Both successful and failed processes. There must be an option to save reports to a file.

With these operations the application can run its main goal of displaying the results of the tests to the user. The operations listed above are large generalizations, and their specific functions and procedures are to be explored more in the coming chapters.

3.1.2 User Interface Design in the Application

The success of any computer application is dependent on it providing appropriate facilities for the task at hand in a manner that enables users to exploit them effectively (Dillon 2006). This aspect of software development is encapsulated in user interface design, which serves as the bridge between software’s back-end functionality and the users who interact with it.

UI is the point of interaction between humans and technology. It consists of the visual and functional elements of an application, granting the user an ability to navigate and interact with the said application. At its core, UI design is the art of presenting complex functionality in a way that is intuitive, accessible, and aesthetically pleasing.

Main principles that should be considered in good UI design are clarity, consistency, simplicity, feedback, and flexibility. A well-designed UI enhances the application, as it can affect many aspects of the application's usability. Good UI increases user experience, making interactions with the application smooth,

enjoyable, and frustration-free. Intuitive UI design streamlines the user's workflow, increasing productivity and efficiency. A well-thought-out UI also minimizes user errors in the application's operation process, since creating a simple and easy-to-understand UI layout reduces the possibility of misunderstandings and mistakes. Providing the user with feedback regarding their actions helps them understand the weight of their interactions with the application.

The contract test preparation application takes into account the principles of good UI design. The application lacks the complexity of larger commercial applications, so clarity and simplicity are easy to apply. The thesis application's UI design also takes reference from other applications used internally by MacGregor, as the relative familiarity of the UI will enhance the user experience and reduce the learning curve associated with the application. As the application is only to be used internally at the company, the UI is more focused on functionality over visual appeal.

Figure 6 shows the main window view of the thesis application. The main window is empty, aside from the "File" and "Communication" dropdown menus on the top-left corner of the window. Inside the File dropdown menu is only the "Exit" input, which closes the application. The Communication dropdown menu is populated by the "Load Ship," "Match Ships and Containers," "Query Results," and "Quit" inputs. Having these dropdown functions in this way is identical to another application used internally at the company called "Lashmate socket communication client test application." This similarity between the two applications will help the intended users navigate the thesis application more easily.

The Communication dropdown menu holds the application's main functionality. The operations are placed in the dropdown menu in a way that the user must activate each operation in a top-to-bottom order to complete the application's functionality.

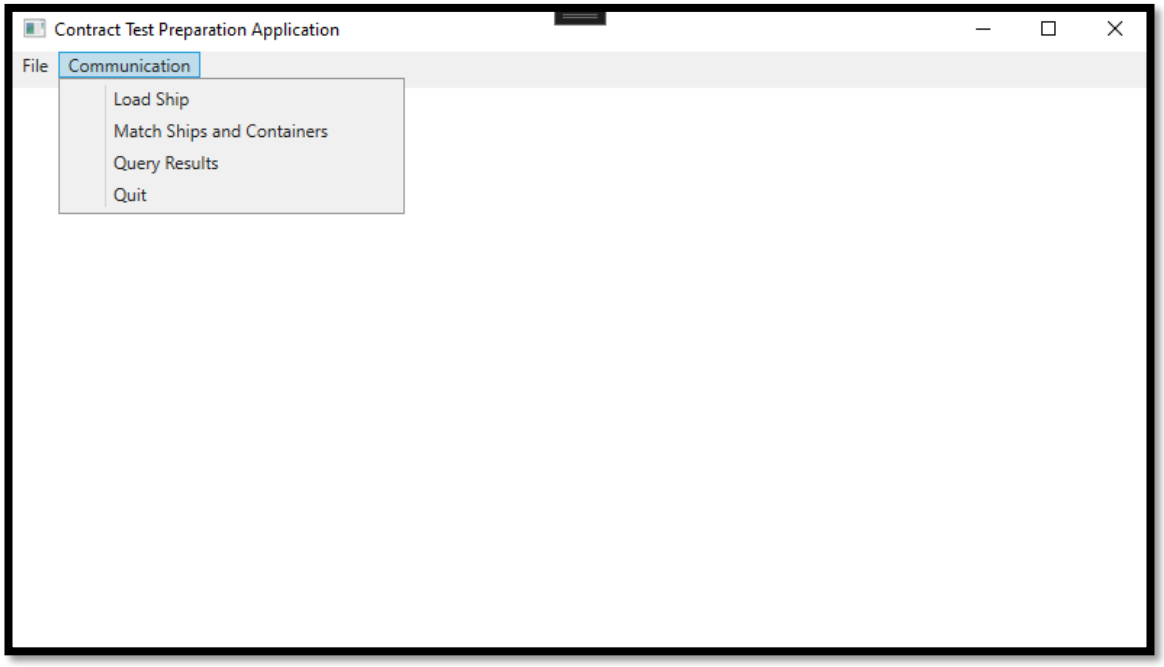

Figure 6. Main window view of the contract test preparation application.

Load Ship input (Figure 7) is the first step that the user must take in the application. Pressing the Load Ship input in the Communication dropdown menu opens a separate, smaller window in the middle of the main window. In this separate window the user must open the file explorer to find the appropriate Lashmate installation folder, which holds the necessary folders and files to execute the application's main function. The Load Ship window is a small window with clear and simple buttons. After choosing the folder through the file explorer, the folder's path will be shown in the thin, gray rectangle on the top-left area of the window. This works as feedback for the user, as the printed folder location indicates that the application has confirmed the folder's existence. After the folder has been found, the user presses either "Apply" to apply the chosen folder for the next operation, or press "Cancel" to cancel the chosen folder and close the Load Ship window.

Figure 7. Load Ship window view.

The next step in the application is to enter the “Match Ships and Containers” input in the Communication dropdown menu. The window contains three different list elements, with two of them next to each other at the top of the screen, and the third one at the bottom. The top two lists contain the LMUI and LMUD files from the folder chosen in the Load Ship input step, while the bottom list appears empty. The user must select one LMUI and one LMUD file from the top lists by clicking on them, and then press the “Add Pair” button to add the two files together. This created pair will appear on the bottom list. The user can also choose to delete an added pair by clicking on it and pressing “Delete Pair” button instead. To ensure that the user is aware of a clicked element, the window highlights the chosen element by boldening the font of the chosen element. This method works as more feedback for the user, so the risk of creating unwanted file pairs is reduced. The “Apply” and “Cancel” buttons are also like the ones from the previous Load Ship window to create a consistent UI experience for the user. After the user has created the wanted pairs, the user can either press “Apply” to apply the chosen file pairs for the next operation, or press “Cancel” to cancel the chosen file pairs and close the open window

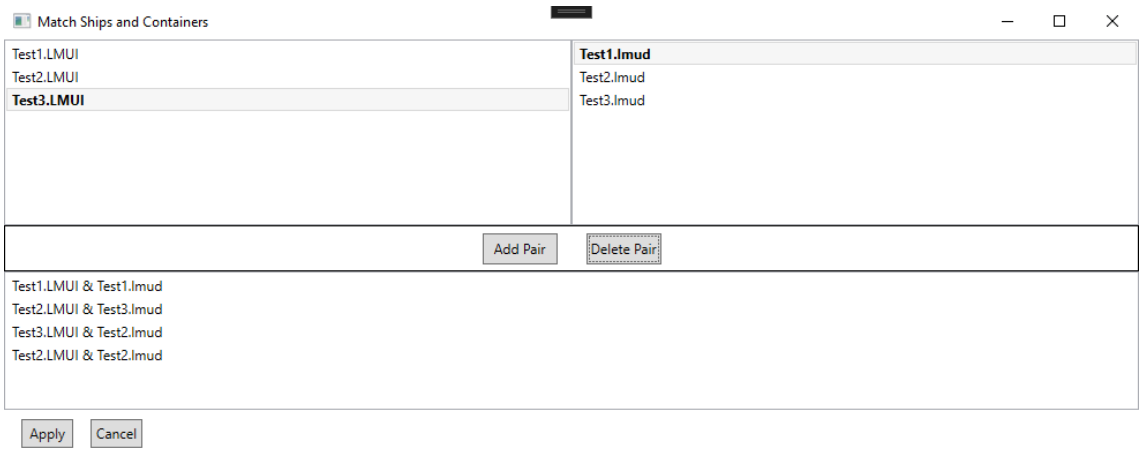

Figure 8. Match Ships and Containers window view.

The remaining two inputs inside the Communication dropdown menu do not consist of their own separate window views. “Query Results” input can only be operated successfully once the user has completed the two previous steps without complications. The Query results input will start the CONSORT protocol to temporarily install Lashmate application and create the socket communication with Lashmate and contract test preparation application. Results and comparisons of the test cases inside the chosen Lashmate installation folder will be automatically calculated and shown in the application’s main window’s previously empty area. The process of the query is shown for the user in lengthier processes to provide feedback. After the query has been completed successfully, the user will be given a possibility to save the results to a separate text file. The “Quit” input’s function deletes the previously selected folder and pair information from the application’s temporary memory, as well as the result information showing in the main window view. After this the user may start another operation cycle in the application.

To ensure consistent, quick, and intuitive application utilization, multiple error handling functions have been developed. There are multiple different error messages that can appear once the user tries to apply the chosen folder in the Load Ship window. In Macgregor, the Lashmate installation folder has an internally standardized composition, with specific folder names and file locations. Because of this, error messages about different missing files and folders are plentiful. In the “Match Ships and Containers” window the user cannot add the

same pair twice into the third list, as an error pop-up window will appear to warn the user of the mistake. In the Query Results step of the application if the two previous steps have yet not been completed, a pop-up error window will appear in front of the main window view to tell the user so. The application utilizes pop-up error windows to inform the user of a mistake in the operation process.

3.2 Programming and Operational Design

This subchapter will discuss the programming of the thesis application, its distinct functions, and the application's code structure. The subchapter will also touch on the CFBF and its implementation in the application more thoroughly than described before.

3.2.1 Code Structure

The thesis application project is done in Visual Studio, using a WPF app template with .NET framework. This template utilizes the use of C# programming language and is used to create Windows desktop applications. Contract test preparation application's directory structure follows the WPF app template's primary hierarchical structure, with UI-defining XAML files nestling the XAML.cs files that hold the main functions and methods of the application. The various files have been inserted into differently named folders to improve readability and structure of the directory. Further programming files that are not tied to an XAML file are placed separately.

As seen in figure 9, the XAML files have been placed in the "View" folder, which contains the "UserControls" and "Windows" folders. "UserControls" folder stores the application's dropdown menu bar's XAML and XAML.cs files, while "Windows" folder holds the appropriate files for the functionality of the windows' buttons. "ShipDefinitionReader.cs" and "LogicHandler.cs" are programming files, that consist of the application's function code, excluding buttons' click functions. LogicHandler.cs contains the function that works in collaboration with

OpenLmuiFolder.dll to correctly open and read the LMUI files needed. Most of the application's main functionality was sought to be kept in one programming file to keep the data types of the application from traversing through different programming files, which complicates the code unnecessarily. However, some functions could not all be implemented into the LogicHandler.cs due to the dependencies they had on the specific file types.

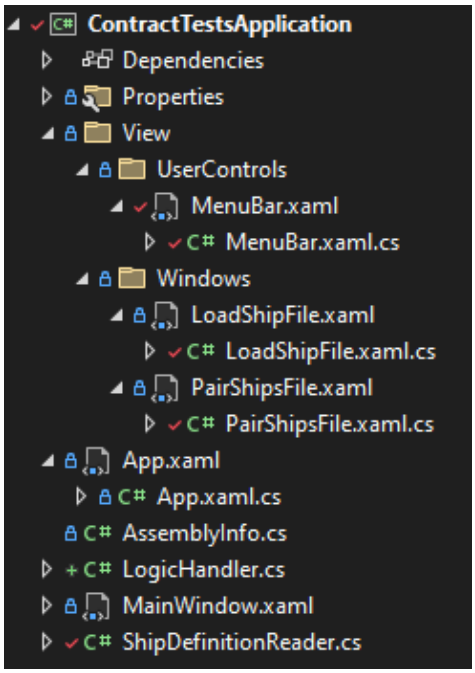

Figure 9. Directory hierarchy of the ContractTestApplication project.

Naming conventions in the code of the application follow the basic and widely used principles, with classes and variables utilizing capitalization in every word of the name, and functions having the first word be lowercase. Code documentation has been used to help the code's readability.

3.2.2 Functionality Implementation

In paragraph 3.2.2 the code of the main functions of the contract test preparation application will be explored. Due to the large amount of code, only the most significant and essential functions and methods shall be depicted and analyzed. These functionalities will be explored in the same order as they are operated in

the application's standard operation order. The code of some of the functionalities, for example: CONSORT and LMUDReader, will not be shown and analyzed in the thesis, as they have been imported into the project as a DLL file.

3.2.2.1 LoadShipFile

LoadShipFile.xaml.cs and LogicHandler.cs contain the functionalities to complete the steps 1 and 2 from the core functionality list in 4.1.1. In this file the selection and checking of the Lashmate installation folder is performed. File exploration is done by using "System.Windows.Forms" namespace. This namespace contains a "FolderBrowserDialog" class, which can perform the necessary folder exploration and selection. "DialogResult" method from the namespace confirms, that the user has selected a folder. After the confirmation, the selected folder path is saved into a default string variable for further use. The explained program operation is shown in figure 10.

```
1 reference | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void fileExplorer(object sender, RoutedEventArgs e)
{
    WinForms.FolderBrowserDialog dialog = new WinForms.FolderBrowserDialog();
    dialog.InitialDirectory = "C:\\Users\\User\\Documents\\Test Lashmate Folder";
    WinForms.DialogResult result = dialog.ShowDialog();

    if(result == WinForms.DialogResult.OK)
    {
        folderPath = dialog.SelectedPath;
        fileText.Text = folderPath;
    }
}
```

Figure 10. "fileExplorer" function used to select the installation folder.

The folder path selected by the "fileExplorer" function is next checked by "lookForValidFiles" function. This function crawls through the selected folder, looking for specific folder names inside it. As mentioned in one of the previous chapters, Lashmate installation folders are internally standardized, so they include the same names for files and folders. Inside the selected folder "lookForValidFiles" searches for folder named "Installer," "SHIP DEFINITION,"

and “TEST FILES.” These three folders are further explored for .msi, LMUI, LMUD files. For each valid folder and file found the function changes the values of two Boolean data type arrays called “foldersNotEmpty” and “filesFound.” These Boolean arrays’ pre-designed values are set to “false” and for every correct folder and file found a specific Boolean value is changed to “true.” These Boolean arrays are later used to tell the user of missing folders and files. The explained program operation is shown in figure 11. If no missing files and folders are detected the program saves the locations of all LMUI and LMUD file location paths into two arrays for later use.

The Boolean arrays are used in functions “checkFolders,” “checkFileBool,” and “CheckFolderBool.” These functions create custom error messages depending on which folders and/or files are missing in the Lashmate installation folder. An example error message is shown in figure 12.

```
1 reference | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void lookForValidFiles()
{
    var folders = Directory.GetDirectories(folderPath, "*", SearchOption.AllDirectories);

    foreach (var folder in folders)
    {
        if (folder.EndsWith("Installer")) {
            foldersNotEmpty[0] = true;
            var msiFile = Directory.GetFiles(folder,"*.msi");
            if (msiFile.Length > 0)
            {
                filesFound[0] = true;
            }
        } else if (folder.EndsWith("SHIP DEFINITION")) {
            foldersNotEmpty[1] = true;
            var lmuiFile = Directory.GetFiles(folder,"*.LMUI");
            if (lmuiFile.Length > 0)
            {
                filesFound[1] = true;
                lmuiArray = lmuiFile;
            }
        } else if (folder.EndsWith("TEST FILES")) {
            foldersNotEmpty[2] = true;
            var lmudFile = Directory.GetFiles(folder, "*.lmud");
            if (lmudFile.Length > 0)
            {
                filesFound[2] = true;
                lmudArray = lmudFile;
            }
        }
    }
}
```

Figure 11. “lookForvalidFiles” function.

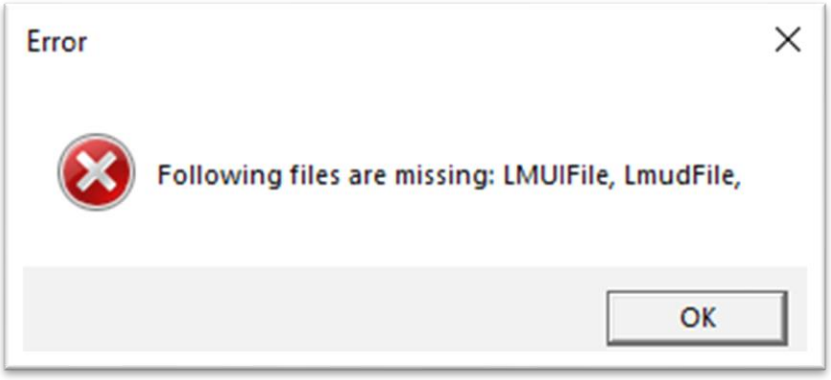

Figure 12. Example of an error from missing files in the Lashmate installation folder.

3.2.2.2 PairShipFiles

Next functionality to be implemented was the pairing operation of LMUI and LMUD files. This was carried out in the “PairShipFiles.xaml.cs” file (Figure 13). This program takes the before-saved LMUI and LMUD file location paths from the arrays and prints them out in a shortened version for the user to select. Upon selecting two different file paths and adding them as a pair, the code simply adds the two shortened file path names together in a sentence, printing out the pair in the pair list. The pair selection data is saved in an array list called “pairIndexList,” that stores the index of both file paths from the arrays they were first acquired from. From here the paired file paths will be opened and get their data extracted by two data parsing programs.

Due to the high number of buttons and click functions in the file pairing section of the application operation, the functions inside the PairShipFiles.xaml.cs could not be immigrated inside the LogicHandler.cs. The data typer, however, are still called from the LogicHandler.cs, which simply copy the values that the user sets in the PairShipsFile.xaml window that was shown in figure 8.

```
1 reference | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void PairAndAddItemsToList(string shipItem, string contItem)
{
    var pairedItem = shipItem + " & " + contItem;
    pairListView.Items.Add(pairedItem);
}

1 reference | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void PairItemIndex(int shipIndex, int contIndex)
{
    int[] ints = { shipIndex, contIndex };
    pairIndexList.Add(ints);
}

2 references | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private string ShortenPathNames(string pathName)
{
    string shortName = Path.GetFileName(pathName);
    return shortName;
}

4 references | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void NullValues()
{
    pairListView.SelectedItem = null;
    shipListView.SelectedItem = null;
    containerListView.SelectedItem = null;
}

1 reference | Eemeli Elgfors, 1 day ago | 1 author, 1 change
private void AddButton_Click(object sender, RoutedEventArgs e)
{
    if (selectedShip != null && selectedCont != null)
    {
        bool pairExists = pairIndexList.Exists(pair => pair[0] == selectedShipIndex && pair[1] == selectedContIndex);
        if (pairExists)
        {
            MesBox.Show("Selected files are already paired.", "Error", MessageBoxButton.OK, MessageBoxImage.Error);
        }
        else
        {
            PairAndAddItemsToList(selectedShip, selectedCont);
            PairItemIndex(selectedShipIndex, selectedContIndex);
            NullValues();
        }
    }
}
```

Figure 13. “PairShipsFile.xaml.cs” main functions.

3.2.2.3 OpenLmuiFolder

The data parsing of the LMUI folder was done in another programming language due to the difficulty of the implementation in C#. “OpenLmuiFolder.sln” is a common language runtime (CLR) template from Visual Studio that uses C++ as its programming language. CLR runs on the .NET framework, and it allows the C++ programming functions to work inside the C# based contract test preparation application via DLL.

As stated previously, LMUI files are CFPF files, also known as compound files. These files are structured storages with standardized file structure and incremental file accessing. The functions and methods to create, parse, and access these files are implemented inside an API called Component Object Model (COM). This API is not accessible in C# and implementing them separately

would be a difficult and time-consuming task, thus another programming language was used for data parsing the compound files. Inside the OpenLmuiFolder.sln are four functions:

- OpenAndExploreCompoundFiles
- ExploreStorage
- ReadAndPrintStreamContents
- ReturnData

OpenAndExploreCompoundFiles simply takes the compound file location, initiates the exploration of the chosen compound file, and handles the possible error checking that comes with the initialization of this operation. This function also initializes COM with the “CoInitialize” method, as well as opening the compound storage using the “StgOpenStorage” method. At the end, the function also releases the memory and resources used by the operation and uninitializes COM. ExploreStorage function works on exploring the different folders and files of the selected compound file. The function runs through the compound file and uses the next function, ReadAndPrintStreamContents, to read the files. The reading function selects the necessary files by name and parses them through as a single byte array to the last function called in the OpenLmuiFolder.sln, the ReturnData-function. ReturnData takes the created byte array and turns it into a data type that is usable in the main application.

Back at the contract test preparation application, a function called “decodeLmui” can be found inside the LogicHandler.cs file. This function receives the byte array from the ReturnData function. Currently, the byte array is unreadable, so the function must decrypt it first. After the decryption, the function parses the decrypted byte array into a “MemoryStream” method that is provided by a “System.IO” namespace. This method turns the byte array into a readable string data type.

3.2.2.4 ShipDefinitionReader

The information from the OpenLmuiFolder.sln, decrypted and parsed by decodeLmui function, is used to extract essential information about the container ship that the LMUI file was created for. This necessary information contains items like ship name, International Maritime Organization (IMO) code, ship builder company name, hull number, and the ship classification society name. These specific data are picked from the created string by a function called “lmuiValueReader” that is located inside the ShipDefinitionReader.cs programming file. The function uses regular expressions to search the string for the required information. Due to the structure of the string, the information right before and after the required data is known, so the regular expressions target the data between the known values. The wanted data is then placed into a list to be used on by the “reportMaker” function. This function takes the list of created data and places it in report format to be later shown to the user.

3.2.2.5 Result Query

The last operation to be ran in the contract test preparation application is the “QueryResults.” This function controls the dropdown menu click function of the same name. QueryResults consists of four calls to distinct functions: previously mentioned “decodeLmui,” as well as three other ones, “containerReader,” “consortStackTest,” and “consortContainerTest.” containerReader is a function that utilizes the LMUDReader DLL provided by the MacGregor programmers. The function decodes the LMUD files and parses through the container information of the file. This container information consists of container’s specific weight, code, and height, and containers position in the ship, comprising row, bay, and tier data. This data and the LMUI data are then used in the consort test function. To put it simply, consort test functions are test functions meant to imitate the real CONSORT protocol. The consort tests require both LMUD and LMUI data collected from previous functions to operate properly. Once the “QueryResults” function is activated, the application runs the programs connected to the function, carries out the necessary calculations, and finally displays the results in a report format for the user in the main window of the application.

4 Application Testing

This chapter will explore the testing functions and the methodology used in testing the thesis application. The chapter will also touch upon the imitation CONSORT protocol used in the testing, as the real CONSORT protocol is unavailable for use in the application testing. The imitation protocols return values are also listed for context.

The contract test preparation application's testing is handled by the consort test functions, as stated in the previous chapter. Due to the application using simple test functions instead of the real CONSORT protocol, the results of the tests are known beforehand the testing. The test functions utilize a DLL called "ConsortInterface," which is provided by the MacGregor programmers. ConsortInterface imitates real methods used in CONSORT protocol but returns only pre-determined values from the methods. Parameters required by ConsortInterface methods are also accurate to the real protocol, so if the consort test functions in the thesis application work correctly with ConsortInterface then it will also work with the real CONSORT protocol. The testing will determine if the application's operations and their respective functions and methods work properly to output the already-known values provided by the ConsortInterface DLL.

ConsortInterface returns a number of different values through various methods. Some of the more important methods are called "queryStackResults" and "queryContainerResults." These two methods return values such a container stack weights, container stack's vertical center of gravity, individual container's position in the bays, rows, and tiers, and each container's corner coordinate position and container's general direction. Each container also includes ten different values for various forces applied to them, such as compression, lashing, and tension.

As an imitation protocol, ConsortInterface returns only the same values for every container, stack of containers, and other CONSORT values. These values exceed over 20,000 depending on the size of the LMUI and LMUD files used, which would greatly affect the application's performance. With the imitation protocol in use instead of the real CONSORT protocol, the displaying of error calculation values is not possible. Thus, only basic container stack values of one LMUI and LMUD pair are going to be printed out in a user report format, and the rest will be displayed in Visual Studio's Solution Explorer window. In the practical usage of the application, these values would only be presented to the user if the values differ from pre-determined test values created for specific LMUI files, but in this test scenario the values are shown to test the application's output and performance.

The performance of the application shall be tested via using "MethodTimer.Fody" tool acquired from NuGet, which is a package manager for the .NET framework. This tool adds a way to measure the time it takes to complete methods inside the application code. With the measured time of different methods, performance of the application can be analyzed with average time values calculated from various runs of the application and detect bottleneck methods in the code that slow the performance down significantly. Due to the simplicity of MethodTimer.Fody, only the most data intensive methods shall be calculated, focusing on the methods dealing with the data from LMUI and LMUD files. The test will be carried out with ten of the main methods of the application, with the test being repeated ten times.

Additionally, the performance of the UI will be tested by utilizing a test tool native to the Visual Studio called "Performance Profiler". From this tool a graph of the UI thread utilization can be displayed, which shows the highest performance-taxing processes are in the user interface. This test shall include one full application run time, which lasted approximately thirty seconds.

5 Testing Results

This chapter will introduce the testing results, including the accuracy of the returned values and the application's performance. The results will be further discussed, and some limitations and recommendations will also be presented.

5.1 Report Output

The application’s report output was as expected. The LMUI file was read correctly, and the necessary information about the ship was printed out at the start of the report. This information can be seen in Figure 14 below, with the actual outputs blurred out for security reasons. After the LMUI file’s information comes the stack results. The positions of the individual stacks are read from the LMUD file provided for the application, which are afterwards run through the queryStackResults method. This outputs the necessary information about each stack in numerical order, starting with the stack on the deck of the ship with the lowest bay and row number. In the case of the test LMUD, this stack was in bay 1 and row 0. Some relevant information about the stack was also presented after the location of the stack. These include the overall weight of the stack and the vertical center of gravity related to the stack. These values were static in the test scenario, which was expected.

In Figure 15 the output of the queryContainerResults method can be observed. The number of containers in this test scenario was 11,133, with each container containing the values of the containers position, direction, and forces applied. In the figure one such container with its inner values can be seen. This output was omitted from the printed-out report due to the vast number of values, and thus was decided to be only observed through the Visual Studio Solution Explorer. By reviewing the query's values, it can be determined that its output was sufficient and contained the expected values.

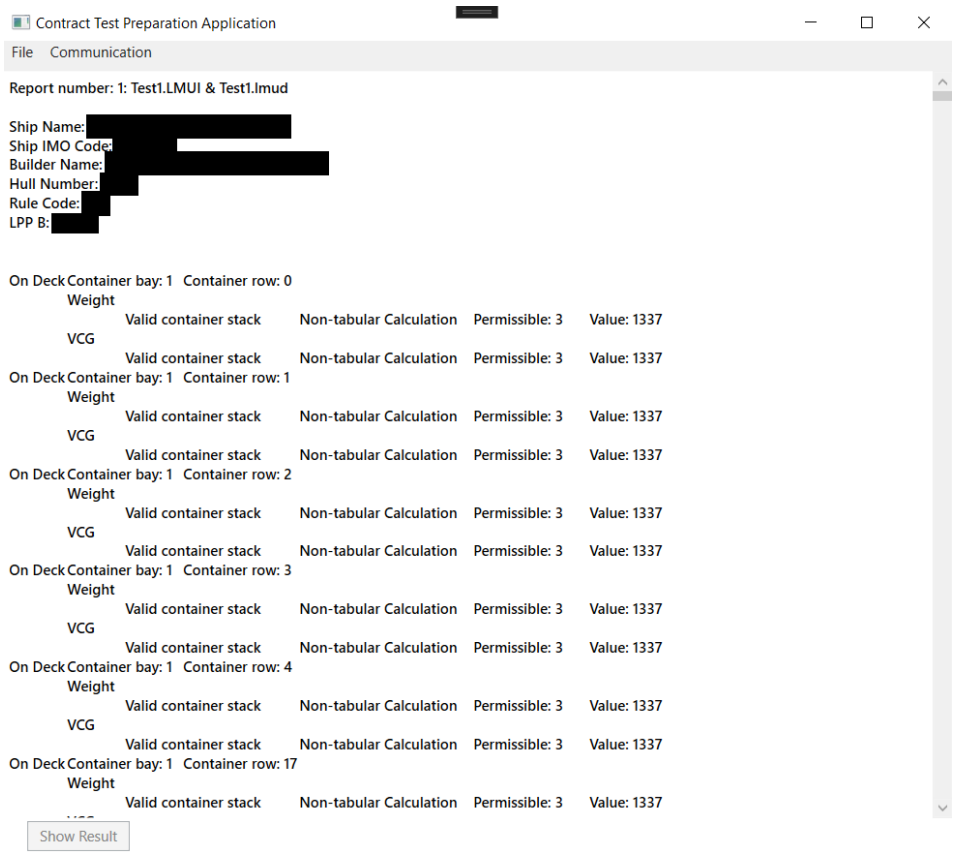

Figure 14. Report output in the thesis application.

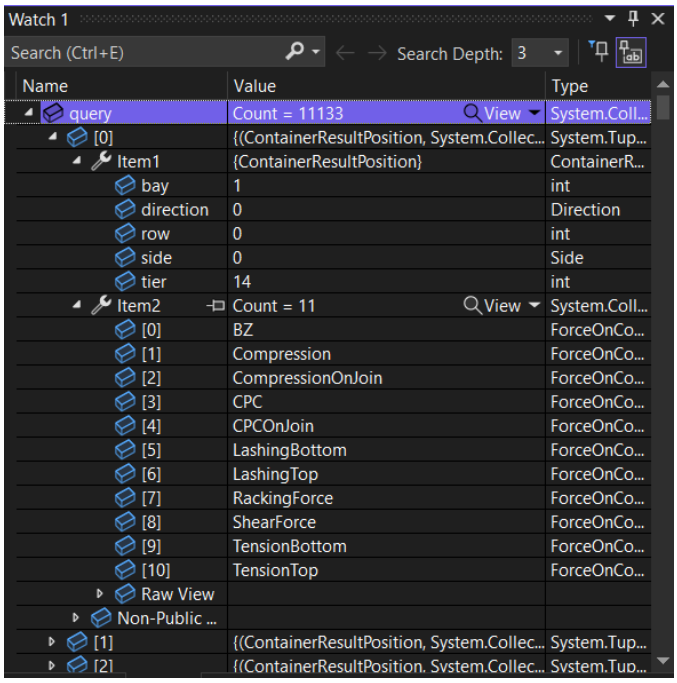

Figure 15. The output of the queryConstainerResults.

5.2 Performance

Table 1 below shows results of the performance tests of the selected main methods. The performance of the methods was measured in milliseconds. From the table it can be seen that most of the methods perform in well under a second after the initiation of the methods.

The significant outlier of the group was the fileExploration method that is used to select the Lashmate installation folder for the application to process further. As mentioned in chapter 4.2.2.1, this method uses the FolderBrowserDialog class to run the folder browsing. This class is derived from a namespace created by Microsoft. The next method, lookForValidFiles, took an insignificant amount of time to verify the chosen folder.

The next two methods, decodeLmui and containerReader, both work towards reading and storing the data from the LMUI and LMUD files found inside the chosen folder. Both methods took on average 50 ms together to complete. The consort test methods consortStackTest and consortContainerTest both completed on average faster than expected.

The heaviest methods of the application, excluding the folder browser method, were the methods executing the report making and printing. The “stackResultToString” method arranges the data from queryStackResults into a more readable format, which is passed on to the “reportMaker” method, which completes the reports by adding the necessary LMUI information. Finally, the reports are presented to the user via the “ShowResults_Click” function. The methods used in creating the reports and presenting them to the user were on average the second longest functionalities in the application.

Table 1. Performance test results.

Method Name	Number of Performance Tests (ms)										
	1	2	3	4	5	6	7	8	9	10	Average
fileExploration	3989	2404	2493	2864	2432	2602	2826	2387	2421	2406	2682.4
lookForValidFiles	5	3	3	2	3	3	3	2	2	2	2.8
decodeLmui	31	21	19	19	21	19	19	19	20	21	20.9
containerReader	50	28	43	27	33	26	26	35	28	27	32.2
consortStackTest	9	7	21	8	7	8	8	7	10	7	9.2
consortContainerTest	31	37	48	32	34	29	29	30	34	37	34.1
lmuiValueReader	0	0	0	0	0	0	0	0	0	0	0
stackResultToString	432	408	531	391	388	381	381	391	377	499	417.9
reportMaker	470	446	568	426	428	412	412	421	411	535	452.9
ShowResults_Click	560	512	629	514	495	497	497	491	481	603	527.9

From the performance profiler testing tool shown in figure 16 it can be seen that the heaviest processes taxing the performance of the application's UI are the application code and the Layout. Parsing also causes a significant spike in performance utilization at the start of the application run time. This is due to the fileExploration method mentioned in Section 4.2.2.1. While the first ten seconds of the application run time shows a notable UI utilization growth due to the application code, this did not affect the performance of the UI in a serious manner. However, at the end of the application run time, between twenty and thirty seconds, a very notable performance issue was spotted due to the Layout process. This process is responsible of the large report being printed and shown to the user. The UI experienced a significant decrease in framerate and responsiveness in the basic functionalities of the application and the scrolling of the report was notably delayed and lacked smoothness.

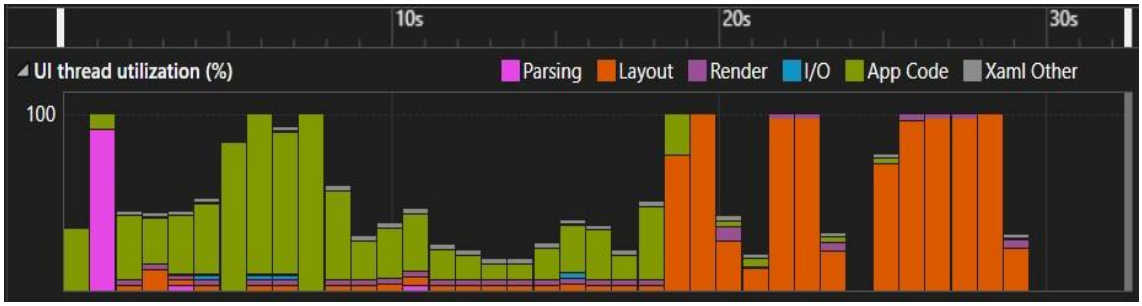

Figure 16. UI thread Utilization Graph.

5.3 Result Discussion

The results of this test scenario provided valuable insights into the thesis application's functionality and performance. One of the larger anomalies of the results was the fileExploration method. Due to the FolderBJurowserDialog class being made and provided by a third party, it can be argued that the notable performance difference between the fileExploration method and the other methods was due to the suboptimization of the FolderBrowserDialog class. The reduction in performance indicated that the WPF TextBlock control used in the report display could not handle a report with a lot of characters.

In the test cases a lot more data was being processed and output than would be in a normal CONSORT protocol communication, so the values gotten from the tests display the sheer output potential of the methods. The length of the full report used in the test scenario was 277,309 characters, which was created, on average, in about 1.5 seconds. The length of the full report also corresponds with the methods decodeLmui and containerReader. The time taken to accomplish these methods will vary depending on the size and number of the LMUI and LMUD files paired up during the Load Ship input function, but overall, the difference in time would not be significant enough to affect performance.

In the case of the lmuiValueReader method, in every performance test the process time came out as 0, which might be due to the timer methods inability to

show decimals past a millisecond. The possibility of lmuivalueReader coming out from the test with a result of 0 is not due to an error in the code, as the reports would not be output in the manner highlighted in the figure 14 if the method was not functioning as intended.

The other interesting point of discussion is the difference between the consort test methods consortStackTest and consortContainerTest. Container test took on average 3.7 times longer than the stack test did. This was due to the difference in amount of data the two test methods must have processed through. The other method that outperformed its expectations was the lmuiValueReader method.

Although the time of creating and printing out the report's averages out on couple of seconds, the real performance problem was observed after the reports were displayed for the user. It can be assumed that this is due to the high number of characters shown to the user, and/or due to the XAML element used to display the report not being optimized enough to present the report without performance issues.

6 Conclusion

This thesis set out to create an application that is capable of socket communication with MacGregor's commercially used application Lashmate, and by doing so is also capable of seeking out bugs and errors in the multiple calculations Lashmate performs throughout its normal functions. The thesis application and its functions were to be completed via a C# program created in Visual Studio, using WPF for the user interface creation.

The original idea was to use the already-created CONSORT protocol to achieve communication between the applications, but unfortunately due to time constraints, the protocol could not be modified for the thesis application's use. Because of this, an imitation test protocol was created that mimics the real protocol by outputting set values.

The testing scenario for the contract test preparation application encountered some challenges and limitations during its testing. The initial difficulty of testing the functionality of the produced reports was the unavailability of the real CONSORT protocol in the thesis application. While the test protocol outputs identical values and gives access to similar methods to the real CONSORT protocol, it cannot accurately imitate the authentic socket communication that would occur between the applications. Thus, the imitation protocol was used as a substitute. Since the imitation protocol does not communicate between applications to calculate its output values, but only outputs set values, the original idea of comparing Lashmate's calculated values to contract test preparation application's pre-determined values could not be recreated for the thesis test scenario. Because of this, an alternative way of measuring application's capabilities was used, which involved the output of the imitation protocol's set values. However, even with this limitation the application is still able to perform the communication with the real CONSORT protocol due to the connecting functions with the imitation protocol.

For the testing of performance, the limitations and challenges were less severe than with the application functionality. The greatest limitation was the lack of a method to accurately assess the reason for the performance issues of the UI elements of the application. A significant difference in performance was visible as evidenced by data, but the VS performance profiler lacked the means to expand on the UI thread utilization further. Another limitation of the testing was the lack of documentation on the MethodTimer.Fody method used to measure time. The accuracy of the time measurements is based on the MethodTimer's inner methodologies, which is unknown for this test scenario. There is also a possibility of the MethodTimer affecting the overall measurements. The small number of tests can also skew the test results due to the small sample size. The performance tests were completed manually, so as a timesaving method a small sample size was decided upon. The small sample size also meant that abnormalities in the test results change the average test results significantly more than in a larger sample size. This can be observed in test number 1, where some of the method's results are higher than normal compared to the other tests. This can mainly be seen on fileExploration, decodeLmui, and containerReader.

Improvements in the testing scenario methodology are highly recommended. A genuine communication between contract test preparation application and Lashmate via the CONSORT protocol is needed for more realistic test results relating to the functionality of the application. On the performance side, a method for automatically running the performance test is recommended. This would allow for a higher sample size, creating more accurate data. A method for visualizing performance data relating to the user interface of the application is also needed. It is recommended to create or acquire a tool to monitor the framerate of the application's main window to measure the performance of the UI.

Further development of the thesis application is already planned in MacGregor. The contract test preparation application must firstly establish a proper connection with the Lashmate application for further other improvements and updates. After this, the next step will be the creation of a system test process, which will create test packages for the thesis application to use as the baseline values for the Lashmate calculations. Further updates will include the application’s assimilation to the general Lashmate build pipeline and the possibility of handling older versions of Lashmate.

References

Andersen, I. M. V. 2012, Wind Forces on Container Ships In: Mercator, Marts.

Andersson, L. 1997, SP Report, Container Lashing, Swedish National Testing And Research Institute

Arcuri, A., 2008. On the automation of fixing software bugs. Proceedings of the International Conference on Software Engineering, pp.1003-1006. doi:10.1145/1370175.1370223.

Australian Standard 2001, Townley Forging Australian Quality, Sydney: Australia.

Dillon, A., 2006. User Interface Design. doi:10.1002/0470018860.s00054.

German Insurance Association 2002, In: Container Handbook: Cargo loss prevention information from German marine insurers, Volume 1. In: https://www.containerhandbuch.de/chb_e/stra/index.html?/chb_e/stra/stra_vo.html (Accessed: 26 March 2024).

Group of Experts for the revision of the IMO/ILO/UNECE 2012 ,Guidelines for Packing of Cargo Transport Units (CTUs), UNECE. In: https://unece.org/transport/intermodal-transport/imoilounece-code-practice-packing-cargo-transport-units-ctu-code (Accessed: 26 March 2024).

Grundmeier, N. 2016, Simulationsbasierte Energiebedarfsprognose in Seehafen Container-Terminals, Oldenburg.

Hakimi Firooz, K. & Lee, M. & Tavakoli, M. 2022, Arrangement and placement of containers in a container terminal. doi:10.13140/RG.2.2.16346.62404.

International Chamber of Shipping a.t., Container Ships, In: https://www.ics-shipping.org/explaining/ships-ops/container-ships/ (Accessed: 20 March 2024).

MacGregor container securing systems product catalogue 2016, Container securing systems, In: https://www.macgregor.com/globalassets/picturepark/imported-assets/65120.pdf/ (Accessed: 20 March 2024).

Microsoft Learn 2021, Containers: Compound Files, In: https://learn.microsoft.com/en-us/cpp/mfc/containers-compound-files?view=msvc-170 (Accessed: 20 March 2024).

Microsoft Learn 2023a, What is Visual Studio?, In: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (Accessed: 20 March 2024).

Microsoft Learn 2023b, A tour of the C# language, In: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (Accessed: 20 March 2024).

Microsoft Learn 2023c, XAML overview (WPF .NET), In: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/?view=netdesktop-8.0 (Accessed: 20 March 2024).

Microsoft Learn 2024,What is a DLL, In: https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library (Accessed: 20 March 2024).

Si H., Sun, C., Chen, B., Shi, L. and Qiao, H. 2019."Analysis of Socket Communication Technology Based on Machine Learning Algorithms Under TCP/IP Protocol in Network Virtual Laboratory System," in *IEEE Access*, vol. 7, pp. 80453-80464, doi:10.1109/ACCESS.2019.2923052.

Tavares de Azevedo, A., Fernandes de Arruda, E., Leduino de Salles Neto, L., Au-gusto Chaves, A. &; Carlos Moretti, A. 2013, “Solution of the 3D Stochastic Stowage Planning for Container Ships through Representation by Rules”. doi:10.2991/.2013.15

Wolf, V., and H. Rathje. "Motion Simulation of Container Stacks on Deck." Paper presented at the SNAME Maritime Convention, Providence, Rhode Island, USA, October 2009. doi:https://doi.org/10.5957/SMC-2009-014.

Zhou C., Li B., Sun X. 2020, Improving software bug-specific named entity recognition with deep neural network, In: Journal of Systems and Software, Volume 165, 110572. doi:https://doi.org/10.1016/j.jss.2020.110572.