Niko Hokkanen

Modularization of a monolithic software application and analysis of effects for development and testing

Helsinki Metropolia University of Applied Sciences

Bachelor's Degree

Information Technology

Bachelor's Thesis

2019

Helsinki
Metropolia
University of Applied Sciences

Author Title Number of Pages Date	Niko Hokkanen Modularization of a monolithic software application 59 pages + 12 appendices 28 August 2019
Degree	Bachelor’s Degree
Degree Programme	Information Technology
Specialisation option	Software engineering
Instructors	Mikko Tylli, M.Sc., Lead software engineer Janne Salonen, Head of ICT, Principal Lecturer

This Bachelor’s thesis is about the decomposition of a monolithic software system into multiple components for increases in application testability, readability and maintainability. The business problem being the slow development speed caused by the monolithic software architecture, where the code base resides within a single executable file.

The method for the modularization relied on the identification of the system’s components, defining their dependency hierarchies and multiple refactorings based on the SOLID principles and dependency injection. For the analysis of modularization effects, multiple key performance indicators were benchmarked and compared between the modularized and original versions of the application.

The analysis of modularization effects revealed minor decrease in compilation times, huge increase in testability due to the ability to run unit-tests specifically against the produced modules and minor decrease in memory consumption.

With the produced, modularized version of the case-study application, the case company has a proof-of-concept module on how the modularization of the application may be done.

Keywords	OOP, SOLID, Application decomposition, Modularization, Software architecture

Helsinki Metropolia University of Applied Sciences

Author Title Number of Pages Date	Niko Hokkanen Modularization of a monolithic software application 59 pages + 12 appendices 28 August 2019
Degree	Bachelor's Degree
Degree Programme	Information Technology
Specialisation option	Software engineering
Instructors	Mikko Tylli, M.Sc., Lead software engineer Janne Salonen, Head of ICT, Principal Lecturer

Tämä insinöörityö käsittelee monoliittisen ohjelmistojärjestelmän jakamista komponenteiksi sen testattavuuden, luettavuuden ja ylläpidettävyyden parantamiseksi. Liiketoimintaongelma ollen monoliittinen ohjelmistoarkkitehtuuri, jossa koodi sijaitsee yhden suoritettavan tiedoston alla.

Työtapa modularisaatiolle perustui järjestelmän komponenttien tunnistamiseen, niiden riippuvaisuushierarkioiden määrittämiseen ja useisiin refaktorointeihin SOLID periaatteisiin ja riippuvaisuus-injektioon perustuen. Useita avain-suorituskyky indikaattoreita vertailtiin alkuperäisen ja modularisoidun sovelluksen välillä modularisointityön analysointia varten.

Modularisointityön analyysi paljasti pientä käännösnopeuden nopeutumista, suurta parannusta testattavuudessa, johtuen kyvystä testata yksikkötesteillä moduuleja monoliittisen ohjelmiston sijaan ja pientä muistin käytön vähenemistä.

Opinnäytetyön tuottamalla modularisoidulla versiolla ohjelmistosta, asiakasyrityksellä on konseptintodistus siitä, miten modularisointi voitaisiin suorittaa ja miten se vaikuttaa testattavuuteen ja kehitykseen.

Keywords	OOP, SOLID, Application decomposition, Modularization, Software architecture

Abstract
List of Abbreviations

1 Introduction 1

2 Methods and material 3

2.1 Meetings 4
2.1.1 Project kick-off 4
2.1.2 Modularization workshop with development team 5
2.2 Application decomposition procedure 6
2.2.1 Decomposition in detail 8
2.3 Outlining methods by which modularization results are analysed 10
2.3.1 KPI selection 10
2.4 Material 12
2.5 Software testing 14

3 Studies of modular system prerequisites 16

3.1 Software artefact dependencies 16
3.2 Defining module, modularity and modularization 18
3.3 Software architecture 19
3.4 Monolithic applications 20
3.4.1 Architectural patterns generally 20
3.4.2 Model-View-View model 21
3.5 Object-oriented programming 22
3.6 Creating loosely coupled code 23
3.6.1 Dependency injection 23
3.6.2 Principle of least knowledge 25
3.6.3 SOLID 26
3.7 TruckTool 28
3.8 Dependency mapping 30
3.8.1 Case study application's software architecture 30
3.8.2 Detailed look into the data model 32

4 Results and analysis 37

4.1 Current state analysis 37
4.2 Application modularization 41
4.3 Analysis of modularization effects 46
4.3.1 Unit test running times 46
4.3.2 Compilation times 49

4.3.3 Application RAM-usage on run-time 51
4.3.4 Communication speed 54

5 Discussion and conclusions 57

5.1 Summary 57
5.2 Next steps 58
5.3 Objective vs. Results 58
5.4 Final Words 59

References

Appendices
Robot script Import CAN
Robot script Import TCP/IP
Robot script Import Serial
TCP/IP import times in the sequence of running. Original version.
TCP/IP import times in the sequence of running. Modularized version.
Serial import times. Original version
Serial import times. Modularized version
CAN import times. Original version
CAN import times. Modularized version
Robot script for TCP/IP batch import
Robot script for CAN batch import
Robot script for Serial batch import

List of Abbreviations

CAN	Controller Area Network
CI	Continuous Integration
CSA	Current State Analysis
DIP	Dependency Inversion Principle
DLL	Dynamically Linked Library
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
ISP	Interface Segregation Principle
KPI	Key Performance Indicator
LoD	Law of Demeter
LSP	Liskov's Substitution Principle
MVC	Model View Controller
MVVM	Model-View-View model
OCP	Open/Closed Principle
OOP	Object Oriented Programming
RAM	Random Access Memory
SoC	Separation of Concerns
SOLID	SRP – OCP – LSP – ISP - DIP
SRP	Single Responsibility Principle
TCP	Transmission Control Protocol
TT	TruckTool
UML	Unified Modelling Language

1 Introduction

Modularity of system's components is an important feature in software design. It allows for their reuse and reconfiguration separately from the system and makes it easier for developers to understand the system and how it functions. It facilitates reuse of the good components within an application and allows for redesign and replacement of inadequate ones. It enables clear separation of concerns and establishes set dependency hierarchies.

Despite the benefits of modularity, applications with monolithic software architecture are common in the field. Their user interface, data manipulation and data access functionalities lie within the same code base, advocating an interconnected and interdependent system composition. For these kind of systems, clear design documentation is of utmost importance to avoid design drift and architectural erosion. These are forces that prey on monolithic solutions, as developers work with these easily corruptible systems.

Although decomposition of a monolithic system into micro services has been the topic of much research in the past few years, the decomposition into modules on the same platform has received less attention. Yet, micro services architecture is not applicable to every software application, namely to the ones where an internet connection cannot be a requirement imposed on the user. Additionally, the decomposition of a system into the central, integral parts is not something the micro service-related research touches upon, but rather advocates service-specific decomposition. These services are what the main components of the application enable, not what they are in entirety.

The objective of the thesis is to gather information on how to decompose a monolithic software application into modules and to apply the information for the procurement of one module from the case study application's solution. Additionally, the effects of the decomposition in terms of code maintainability, readability and testability are to be determined. To these ends, the thesis explores concepts and techniques that aid in the production of largely self-contained components out of an industry level software application with multiple years of development under its belt. The research question is: "How should a monolithic application's code base be decomposed into shared libraries to improve its testability, readability and maintainability?"

The case-study application for which modularization techniques are applied to is designed for the modification and visualization of various operational parameters of lift-trucks. The software is developed by Rocla Oy to which this thesis carried out for. The application is a sizable software project, consisting of over 100 thousand lines of code and has been developed utilizing mostly .NET Framework and C# programming language. The application is one monolithic solution, meaning, that it is composed of one single unit of executable code.

The thesis outline is the following: firstly to go over the methods by which both the modularization and the analysis of its effects is done. Then give background information relating to the terminology of the thesis, Client Company and the decomposable software application. Afterward go over some principles related to refactoring and Object-oriented programming (OOP), which are applied during the modularization work. The results section will give an overview of the current implementation of the system, the performed modularization work and benchmarked key performance indicator (KPI) results and their analysis. Finally, the conclusions section will examine the work done and its ramifications for the software development of the case company.

2 Methods and material

This section will detail the methods by which the objectives of the thesis are to be accomplished.

Information gathering will follow material discussed in development team meetings and interviews. Source material is chosen based on them and based on the material in them. Chosen material to follow in the following sections.

Modularization will target code base section as instructed by the software project management. Functionality specific to application's communication with lift-trucks is given as an assignment for the modularization. To achieve this, an examination of the application's architecture is performed. As there is only few documents pertaining to the architecture of the application and them being rather outdated, some architectural recovery is done as well. Therein the documentation of the application is done with unified modelling language (UML).
The decoupling of software artefacts is done case-by-case bases. Objective therein is to sever inappropriate dependencies, so that the software component honors the intended dependency hierarchy. This hierarchy is recovered through the utilization of UML diagrams, meetings with the development team and analysis of the application's code base.

Modularization results analysis is performed by comparisons between the modularized application and the original. KPI are build- and unit test running speeds, code complexity and more.

This thesis targets .NET C# language-programmed application and as such the terminology used reflects concepts defined within it.

2.1 Meetings

For data gathering and informing the development team of the thesis work's progress, several meetings were arranged during the thesis work. These are referenced in the data gathering table. The most notable meetings are the project kick-off, development team group-interview for current state analysis, workshop for the implications revealed by the current state analysis (CSA) and final interview for the feedback on the work.

2.1.1 Project kick-off

Project kick-off meeting was held with the lead developer, project manager and the thesis worker. Names, titles and dates available from data gathering table. The goal of the meeting was to define the objective of the work, reasons for its necessity and details about its implementation.

Within the meeting the objective of the project was defined: **to compile information regarding modularization in general, procure a singular module from the applications code base and analyse results in terms of at least maintainability and testability**.

Unit tests were also specified to form modules based on the module they were to test. Whereas before in the applications architecture they, like all other components, resided within the one monolithic application with mostly undefined architectural borders. Now for every module that was modularized, a module of the associated unit tests was to be created as well.

The necessity of the work was specified to be an **increased maintainability, testability and speed of the application's development**. This was to result from clearer architecture brought forth by the new modules. Testability would increase due to having well defined, self-contained modules.

2.1.2 Modularization workshop with development team

A workshop concerning the modularization project was held with part of the case-study application's development team. The objective of the meeting was to spread awareness of the work, to gain insight into the existing software architecture of the application as well as to formulate a high-level understanding of the procedure for modularization.

As a result of the meeting, an initial, high-level procedure for the modularization was agreed upon. Thesis work was also specified to include a CSA, wherein a reasonably comprehensive examination into current state of the application architecture was to be conducted. The content of the modularization proof of concept (POC) module was specified encompass all the functionality relating to application's communication functionality with the lift-trucks.

This meeting detailed the focus of the work to be the physical separation of all functionality relating to the application's communication with the lift-truck into its own assembly. To this new assembly would the main project then form a project reference, enabling it to utilize the module but from which a reference back would be impossible due to a formation of circular dependency. This forces the module to be independent of the business logic of the application, which increases testability of the application due to faster unit test running times.

As the communication functionalities namespace was presumed to be quite entangled with circular dependencies, a helper library was designated to be used as well. To this library could some dependencies be moved into if breaking them should introduce a refactoring work thought too demanding or if the dependency would actually be a valid one.

A method for the modularization was specified to be created during this work, so that it might later be used for further modularization, if deemed necessary. Additionally, an analysis towards what kind of modules should the case study application be further separated into could be conducted as well.

Testability of the application during all the stages of the thesis work was emphasised. Functioning set of unit-tests is often pressed to be the foundation for solid refactoring work [4] and so this was set as a requirement for the work over all.

2.2 Application decomposition procedure

A high-level procedure for modularization was formulated during the workshop meeting conducted with the development team. This procedure was further developed during the project and is detailed within this section.

1. Identify the system components by inspecting system documentation, dependency mappings and the code base.
2. Produce component dependency mappings utilizing Visual Studio 2019 Enterprise edition.
3. Identify intended dependency hierarchy between the components. Use system documentation and the development team.
4. Pick the software artefact to be moved to the new assembly.
5. Break its dependencies to the components it should not be dependent on.
6. Move the dependencies it should have to another library.
7. Move the artefact to the Truck communication functionality assembly.
8. Test the changes.

Steps 4 to 8 form an iterative process which loops until the decomposable component is moved into its own project in entirety. See figure 1.

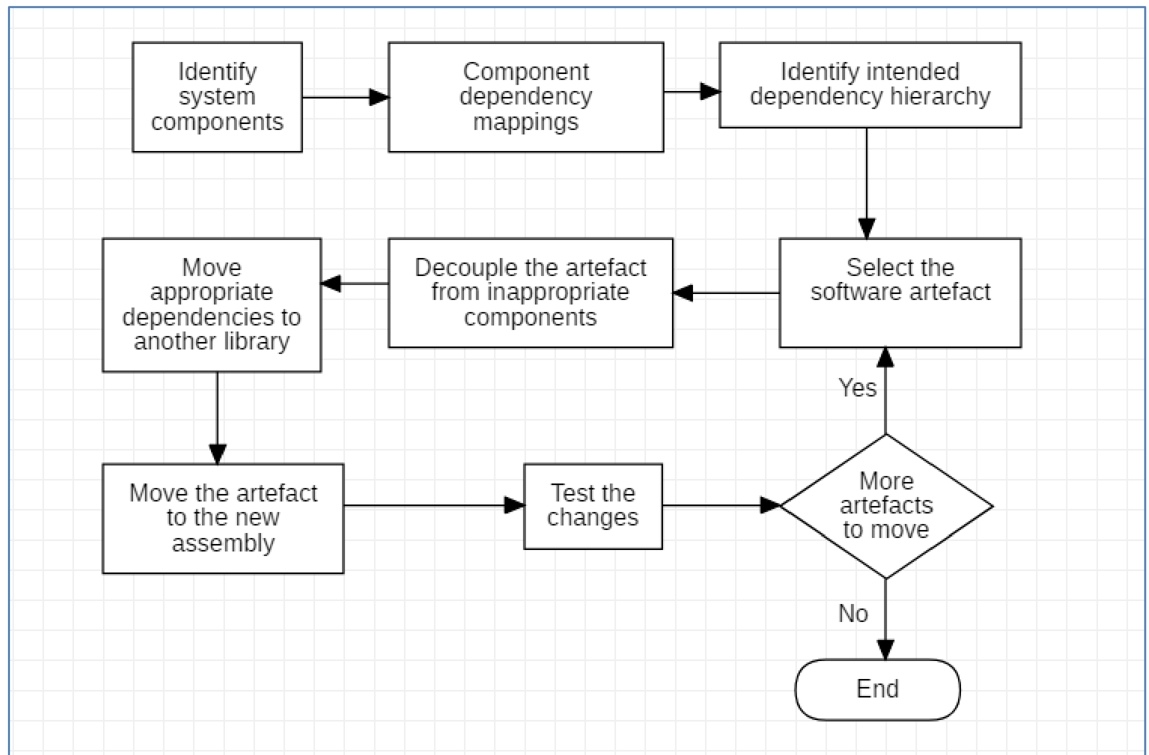

Figure 1. Modularization procedure. Notation: informal

2.2.1 Decomposition in detail

The process starts with the identification of system's components. This is to be done by examining the UML diagrams of the application, by meeting with the application's developers and by code base examinations.

Component dependency mapping is done by abstracting relevant information out of code maps produced with Visual Studio 2019 Enterprise Edition.
These let a developer see of which other classes and interfaces a software artefact, a namespace or an assembly is dependent on without direct examination of the code base although this method is to be used as well. The product of this step is a set of class and package diagrams of UML notation, which help understand the current implementation of the system concerning dependencies.

The process continues with the identification of component dependency hierarchy. Developers and UML diagrams are once more utilized for this step. Obviously code base examinations might not give accurate information, as the system implementation might differ from its design.
This is an important step, as the decomposition relies on the knowledge of whether an artefact's dependency is appropriate or not. Inappropriate dependency is a one that a component should not have. The product of this step is an UML package diagram, which defines dependency hierarchy of the system's components.

Step 4 begins the iterative process of moving the software artefacts from the main solution into the new library. It begins with the selection of the class or an interface to be moved. This should be started from the easiest artefacts – those that have the least dependencies, as they themselves will be ones that are depended on. Thus moving them first will clear the way in a sense for moving the rest of the artefacts. These first software entities are often enumerators, data structures and interfaces, which usually have less dependencies than concrete classes.

Once the artefact to be moved is selected, its inappropriate dependencies should be refactored out. These might be easy or hard, depending on the level of architectural degeneration and quality of system's design.

After inappropriate dependencies of a software artefact are refactored out, its appropriate dependencies should be evaluated in terms of whether they, left in-place, introduce circular dependencies. If no, one can move onto next step and if yes, those dependencies should either be refactored out or they should be moved onto an assembly of their own.

After dependencies have been taken care of for the software artefact, it can be moved onto its new assembly. This should be an easy step, however, one should take care to update the project and namespace references based on the changes made.

Testing of the changes is the final step of the iterative process. Therein changes are validated by testing methods defined in chapter 2.5: “Software testing”.

2.3 Outlining methods by which modularization results are analysed

To determine the results of the modularization, an analysis of its effects is to be conducted. This section details which KPIs are to be examined and how they will be benchmarked between the modularized version of the application and the original.

The decomposition of the application into shared libraries will be conducted on its own feature version control branch. Once the decomposition is completed, its product shall be compared with the original version to determine any effects, positive or negative, to the testability, compilation times and resource usage of the application. Additionally, surplus notes of effects outside the indicators are to be documented and examined also. These are more involved with theoretical benefits and negative effects of the modularization work, which cannot within the thesis work's scope be benchmarked. They are educated guesses based on the research material and modularization product.

2.3.1 KPI selection

The KPIs were chosen based on the meetings with the case-study application's development team on feedback regarding the supposed areas of performance malleability due to the modularization. Two areas of importance were identified: application development and application usage.

- **Unit test running times**
  Development is greatly affected by unit-test running times. They are used when coding to test the changes and they are ran at application deployment automatically. Unit test running times is therefore chosen as a KPI for the analysis.
- **Compilation times**
  Compilation of the application is done to test the changes made. Like with the unit tests, it also affects development during the making of the changes to the code base and at the deployment of the application. Compilation times are therefore used as a KPI for the analysis.
- **RAM usage**
  Benchmarking the random access memory (RAM) usage and communication speeds are determinants of changes caused by the modularization work to the end-users. They're an indication of the effects caused by splitting the code base

into multiple dynamically linked libraries (DLL)s and are as such important factors to take into account. The memory usage benchmarking can also reveal any memory leaks introduced by the refactorings associated with the modularization work.

- **Communication speed**
  Memory usage is recorded with the performance analyser of Visual Studio. The actions executed during that benchmarking were automated with Robot Framework. Three versions of the executable actions were configured to test three different communication formats the case-study application supports. The used robot scripts as appendices 1-3.

Robot Framework is a platform independent UI-automating toolset for acceptance test development [23]. Extendable by test libraries written in Python, it should serve well in automating the UI actions performed during performance benchmarking.

As a summary, the KPIs that are to be benchmarked in this work are the following:

1. Unit test running time
2. Compilation time
3. Application's RAM - usage on run-time
4. Communication speed

2.4 Material

This section details the most relevant source material regarding the thesis work. It is chosen based on the meetings and informal talks with the development team. The concepts of “dependency”, “module”, “software component modularity”, “monolithic structure vs n-tiered one” guided the material search process so, that the following books and resources were chosen.

Although no material concerning modularization specifically was identified, vast amount of material concerning software architecture was available. For software architecture references, “Software architecture in practice” [3] was chosen.

Based on the code analysis and design documentation, it was noted that the case application was designed with “Model-View-View model” (MVVM) architecture pattern. Material regarding to it was sought out specifically but only little was found. However, the creator of the pattern, John Gossman, lays out its main principles in relation to the more common pattern “Model-View-Controller” (MVC) in his introduction to the MVVM [12]. This information is to be used for determining how the components of the case study application should relate to each other.

Modularization as a concept is essentially about changing the already existing code so, that its behaviour does not change, to isolate well-defined modules. This definition makes it a code refactoring task whose objective it is to produce modules out of the solution. Therefore material regarding refactoring was sought. To this end, “Refactoring: Improving the design of existing code” [1] was chosen as one of the guides for the thesis work.

In the effort of finding ways to control the dependencies a module might have, the following tools for producing loosely coupled code were identified: SOLID principles and dependency injection (DI). Following books were chosen as a resource concerning these ideas: “Agile principles, patterns and practices in C#” [4] (SOLID) and ”Dependency injection in .NET” [5] (dependency injection).

In addition to the material concerning class level refactorings like SOLID and Dependency injection, material at the component level was sought out as well. Namely to inform the modularization work regarding how the components should related to each other. For this purpose, Robert C. Martin’s “Clean architecture: a craftsman’s guide to software

architecture” [8] was identified to touch upon the subject in the form of component principles.

2.5 Software testing

The case study application development follows some CI [10] methodologies involving a self-testing, commit-following, automated application build-event. Additionally, pull requests are used for code reviews which the code changes have to pass before being merged into the development branch of the version control. 1 to several team members take part in these reviews for improved code quality.

As constantly testable solution was specified as a requirement and it being an important part of any refactoring work [1, p.3], these already established CI methodologies were utilized. This meant, that for every pull request, a code review was done and automated build and the project's unit tests were ran for build validation. See figure 2 for technical implementation of CI.

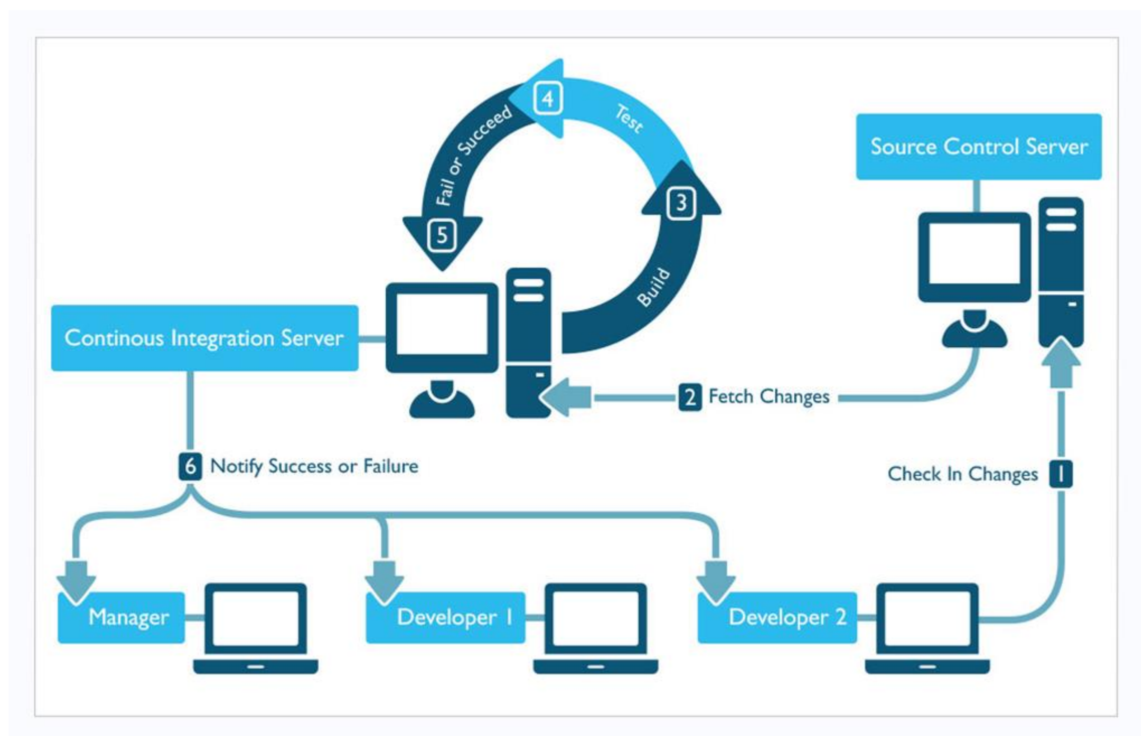

Figure 2. Technical implementation of CI [11]

Although not part of the build validation, the project had an encompassing library of automated regression tests. These could be used to test the changes at system level, whereby the test automation script runs a set of user interface (UI) actions consisting to a specific application's use case.

Manual testing is to be done as well. For these, a virtual truck configuration and several lift-truck simulation boards are utilized. These act as stand-ins for the actual lift-trucks, to which access is limited. There are two different kinds of virtual truck configurations. One for the serial and the other for controller area network (CAN) open vehicles. Both are implemented by software only. The testing boards on the other hand simulate the trucks with hardware. They are configurations of the vehicle's controllers wired together to simulate the vehicles for testing purposes.

3 Studies of modular system prerequisites

This section details research pertaining to the methods outlined in the previous chapter. In this section the background relating to the research space, case-company, the application to be decomposed and .NET framework is given. The research starts with the definitions of key terminology relating to the modularization work, goes on to introduce the case-study application before moving on to software architecture and one of its patterns of relevance concerning the application. Additionally, multiple OOP-related design principles are examined in relation to producing modular components.

3.1 Software artefact dependencies

As the concept of software dependency is of such integral relevance to the modularization work described and investigated by this thesis, the term is briefly explained in this section and discussed in relation to the concepts of coupling and cohesion.

Coupling is defined as: "*a measure of the interdependence among modules in a computer program*" [24, 2-3]. The definition explains coupling to be synonymous with the level of dependency a module has to the surrounding system. When a module has low coupling, it is self-sufficient and has a high cohesion. This often indicates a good separation of concerns (SoC) and that the module's functions and methods work cohesively together to fulfil the one responsibility designed for it.

[19, p.64] Defines loose coupling as one of the desirable design characteristics. It also defines it to occur between two software entities, which are connected solely by interfaces. As discussed earlier, interface usage hides the details concerning how classes execute their procedures. This in contrast to tight coupling, where interfaces are not used, and the interoperating modules are aware of the implementation details of the other.

While having dependencies and coupling is by no means inherently good or bad, they have a significant effect on the realised software architecture and testability of the application, especially when considering more sizable projects: unit testing is hard to do

properly, when the testable unit depends on five other units. Reuse of application components is difficult as well, when the component is wired to the rest of the application. See table 1 for more benefits gained from loose coupling.

Table 1. Benefits of loose coupling. [5]

Benefit	Description	When is it valuable?
Late binding	Services can be swapped with other services.	Valuable in standard software, but perhaps less so in enterprise applications where the runtime environment tends to be well-defined
Extensibility	Code can be extended and reused in ways not explicitly planned for.	Always valuable
Parallel development	Code can be developed in parallel.	Valuable in large, complex applications; not so much in small, simple applications
Maintainability	Classes with clearly defined responsibilities are easier to maintain.	Always valuable
Testability	Classes can be unit tested.	Only valuable if you unit test (which you really, really should)

3.2 Defining module, modularity and modularization

The term "module" in this thesis refers to a project of an application, which by way of compilation produces either an exe or DLL. These modules are then used by the business logic to extend the functionality of the application by some very specific way. The modules are decoupled from the main application so, that module has no dependencies towards the main application but the main application may depend on the modules. This ground-rule alone will dictate a certain structure to the high-level composition of the system: the main application will act as the controller for calling the functionalities of the modules and will have references to them based on the need for their services. The modules' functionalities are not dependent on the logic within the main application.

The communication between the module and the main application happens through well-defined interfaces. The modules may be exchanged behind the interfaces without breaking the application. Consequently, the modules should be usable by any other module through the interfaces they offer.

Modularity is the attribute which indicates the degree of interdependence of a component or a module from the surrounding system. For the purposes of this thesis, component with high level of modularity is called a module. Given the definition for coupling in the previous section, a module has a low coupling to the surrounding system.

The act of modularization or decomposition is defined as: "a large *software is divided into a number of smaller named components having well-defined interfaces that describe component interactions. Usually the goal is to place different functionalities and responsibilities in different components*" [24, 2-3]. This describes the objective of the thesis work quite well. The components being projects which produce a .NET DLL.
The term "modularization" within this thesis could also be described as "code restructuring" or "software retrofitting". It is the act of increasing the modularity of the system components by refactoring into them looser coupling.

3.3 Software architecture

Understanding the system components and their relationships is of critical importance when analysing a system. For this purpose one should be acquainted with the notion of software architecture.

[3, p.2] defines the term "software architecture" in the following way: "*The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them*".
The definition determines, that computer applications are systems comprised of software elements and that architecture is the definition of those elements and their relationships.

[9, p.156] defines three purposes for architecture:

1. It serves as a means of education. For the new members of the development team it can help understand the structure of the system, their part in its development and what the system does.
2. It serves as a communication vehicle for project stakeholders.
3. It serves as the basis for system analysis. As it is the ultimate abstraction of the system, it can help in the analysis of whether the code base reflects the design.

Software architecture of an application often takes a form of a pattern or patterns. The most common ones seem to be the "layered pattern", "micro services pattern" and MVC. [6, p.1] They offer a framework, by which to organize the application's code base. This has multiple positive effects: the components relate to each other in a known way and the design is easier to communicate, when it can be talked about with a well-defined pattern.

3.4 Monolithic applications

Monolithic system architecture is defined to consist of a "*single application layer that supports the user interface, the business rules, and manipulation of the data all in one*" [7]. The Microsoft document goes on to specify, that should the application consist of multiple assemblies like DLL's, it should still be counted as a monolith. Therefore, even though the assemblies would not share a common application solution (group of projects) but would still come together at run-time on a single hardware, the software system would still be a monolithic one.

As a monolithic system is built for one platform to host, display and control, the control between the all the components happen with function calls. They are fast but facilitate also interdependent systems: passing complicated dependencies within them is easy. This in contrast to web applications for example, where the services are called via Hyper Text Transfer Protocol (HTTP).

Without a forced SoC, monolithic systems tend to evolve into big balls of mud: a composition of interdependent components, where a change in one alters or often breaks another [13]. This degeneration of the code base is often hastened by a lack of design documentation in the lines of component dependency hierarchy.

3.4.1 Architectural patterns generally

Though monolithic, an application is free to follow multitudes of architectural patterns. In essence, they are core ideas of solutions to general problems one has when designing a system. As the systems have various demands like performance, testability, platform independence, the design solutions have to accommodate them – patterns address different compositions of these factors.

Architectural patterns offer a common framework for organizing the code base and its components so, that the developers may have get a good idea of the application's mode of operations by only knowing the pattern.

3.4.2 Model-View-View model

As noted in the section “Case-study application’s software architecture”, the case-study application is designed with MVVM architecture pattern. This section details its most prominent features and strengths.

MVVM is a variation of a more traditional MVC pattern. It is designed to separate the logic and the UI from each other. This enables a more graphically oriented designer to work on the view instead of a requirement for a developer, which might not be that concerned of the visual side of the application. It also increases testability of the traditionally more hard to test UI logic by decoupling it from the UI controls. [14]

The pattern defines three entities the application should consist of:

1. **Model**
2. **View**
3. **View model**

The model is as defined by the MVC pattern: the business logic and data model of the different entities of the software application. It is completely UI independent [12].

View consists of the style, layout and content of what the user sees on the screen.

View model is an abstraction of model to be used by the view to display model details.

The three components of the MVVM pattern form a dependency hierarchy: View should know about its view model and the view model should know about the model. Model however should not know about the other two and the view model should not depend on any specific view that utilize it. See figure 3.

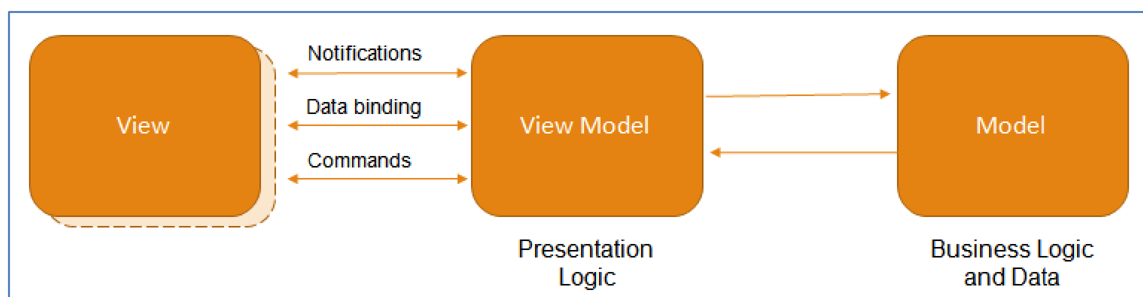

Figure 3. Model/View/View Model interaction diagram [15]. Notation: informal

3.5 Object-oriented programming

As the case study application is mostly written in C#; an object oriented programming language, this section delves into that concept. This information is applied for the refactorings reported in the analysis and results section.

OOP is often said to rely on 3 pillars: inheritance, polymorphism and encapsulation. [25] Understanding them is a requirement for any meaningful analysis of an application written in C# and as such they are discussed within this section.

Inheritance is about creating hierarchies of classes. Meaning, that a class can inherit properties and methods of its base class. When this happens, the child class forms an “is a” relationship to its base class. Multiple different classes can inherit the mentioned base class while implementing its functionalities in differing ways. [17, p.4] That brings us to polymorphism.

Polymorphism is about having a datatype which can behave differently depending on how it is implemented. A developer can have a wide variety of different data types but as long as they share the same base class – when they inherit it – they can all be treated as that base class. This means that they can share a list of the base class data type and the list can then be iterated over. Calling a method declared in the base class can then result to different behaviour depending on how it is defined in the derived classes.
Another way to produce polymorphic behaviour is by utilization of interception: object behaviour can be altered at run-time with various DI containers by intercepting method calls and altering them based on the data within. Unlike the polymorphic behaviour achieved by inheritance and encapsulation alone, this method alters behaviour of even objects of the same data type.

Encapsulation is a technique concerning implementation detail hiding by way of access restriction utilization. C# and Java, among others, let developers do this explicitly via the usage of *private* and *public* keywords on function-level. Encapsulation can be used on higher levels as well: interface usage in C# hides the actual implementation behind them, allowing for looser coupling between software artefacts. Their utilization can be used to hide the implementation details of a whole component. [17, p.4]

Interface in C# is a common language library (CLR) reference type, which lists method signatures. This list is a contract, which obligates the interface implementing classes to

implement every method, property, event and indexer declared within the interface. The instances of classes which implement the interface can be passed to a client code as the interface datatype. Thusly the client receives only the method signatures but the implementation details are hidden from it. [26] This means, that the client is no longer dependent on the implementation.

3.6 Creating loosely coupled code

This section examines various sets of tools and principles for producing loosely coupled and modular components: DI, Law of Demeter and SOLID principles. Techniques such as these have to be employed to produce general use libraries, such as ones to be created as a result of this thesis work.

Though not strict rules to live by, they offer a set of guidelines which help apply encapsulation, inheritance and polymorphism and help understand different concepts relating to object-oriented design. Additionally, they are designed to help with producing modular code and are, therefore, important for the thesis work.

3.6.1 Dependency injection

As DI is a concept about which one can write whole books about, this section will just define the concept broadly and examines how it could be used by the modularization work of the thesis.

The definition given by Mark Seeman in his “Dependency Injection in .NET” for the concept is the following: “*Dependency injection is a set of software design principles and patterns that enable us to develop loosely coupled code*” [5, p.4]. Daniel Baharestani defines it in the following terms in his: “Mastering Ninject for Dependency Injection”: “*Dependency injection is one of the techniques in software engineering which improves the maintainability of a software application by managing the dependent components*” [20, p.35].

From the definitions one can gather that it's about making the code more maintainable by limiting strong coupling between objects. How could its principles be applied in practice to achieve this?

DI specifies 3 elements of responsibility: object composition, lifetime management and interception [5, p.7]. These, it states, should be handled by objects dedicated for them specifically. This to uphold the SRP.

Object composition as defined by DI is about composing objects for client-specifically. This is an act that enables separation of concerns between the calling code and the object parameters given to the server. Normally they would be composed by the caller and even though they might be given to the server in the form of an interface, the associated dependencies are with the client. DI proposes a class for the composition specifically – a composer.
At its simplest, DI can be done by just passing the responsibility for the object composition higher in the call stack.

Object lifetime handling is closely associated with the object composition. As an object gives away the control over the composition of the object, it also gives away the control over its lifetime. This is due to .NET's garbage collection, which is invoked when an object loses all references. This reference is ultimately to the one entity that instantiated the object and a responsibility for the DI composer.

Interception in DI refers to the act of intercepting consumer calls before they reach the called service. An application of Decorator pattern, it enables the modification of object behaviour at run-time for polymorphic behaviour

3.6.2 Principle of least knowledge

Often summarized to: "*Talk only to your immediate friends*" [19, p.134] [20], the principle states that objects should only ever interact with other objects closest to them. Also known as the "Law of Demeter" (LOD), it discourages wiring into the objects knowledge of the internal structure of the system. This is often violated by creating chains of object calls to the style of:

Controller.Sensors.TemperatureSensor.GetTemperature();

In the example, the calling object has inbuilt knowledge (a dependency) of this chain to get the temperature reading from a class called "TemperatureSensor". It knows it can be reached through "Controller" properties' "Sensor" property. If the dependency chain would be broken, the calling class would break as well.

The violation of the principle damage the testability of the calling class, as a failure in unit-tests no longer indicates a problem with the class directly, but may be an indicative of otherwise broken artefact along the dependency chains it utilizes.

Now, assuming that the temperature would be a desirable property for the calling class to be aware of, the application of LOD would make the call something in the lines of:

Controller.GetTemperature();

The actual refactoring associated with the change could be something as simple as the creation of a method for the Controller to return the Temperature. The main point being, that even if an object could use chained calls, they should not to.

As a brief summary: principle of least knowledge advocates for keeping dependencies to a minimum and supports the creation of loose coupling between the components of the system.

3.6.3 SOLID

SOLID is an acronym of principles which aim to make code more extendable, maintainable and flexible. This section will investigate the principles, so that they might further in the thesis be used as part of the analysis when discussing ways for increasing the modularity of a software application.

Single responsibility principle (SRP)
“A class should have only one reason to change” [18, p.15]
A principle advocating for single responsibility per class. A formation of the old adage of splitting a problem into small parts, while making sure no two separate problems are tackled by a same class. The opposite of following this principle would be a god class implementation, which has numerous responsibilities and does everything.

Open/Closed principle (OCP)
“*Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification*”
This principle states simply, that once a software artefact is created and its methods and properties are used across the system, the modification of that behaviour forces the updating of the users as well. Therefore, modification of the depended, existing behaviour should be prevented. Instead, software entity behaviour should be extended with new methods and/or properties. [18, p.15]

Liskov’s substitution principle (LSP)
“Subtypes must be substitutable for their base types.”
This principle advocates for the correct use of inheritance. The obvious case of breaking the principle is when a subtype leaves an inherited method empty for its unsuitableness to the subtype (also sometimes referred to as “refused bequest” code smell) [18, p.15].

Interface segregation principle (ISP)
“Clients should not be forced to depend on methods they do not use.”
This principle advocates interface creation client specifically. [18, p.16]
Massive interfaces often force unnecessary dependencies to clients, advocating interdependent systems and less cohesive implementations. [19, p.133]
Admittedly, the creation of multiple interfaces introduce complexity to the system, and care should be shown that the SRP for the existing classes is not violated.

Dependency inversion principle (DIP)

"High level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions." [19, p.131]

The word "abstraction" used in the quote refers to interfaces and abstract classes while the word "details" to the concrete classes and implementers and inheritors of the abstractions.

This principle advocates the use of abstractions instead of concrete classes for loose coupling between software entities. It leans on OOP concepts of polymorphism and encapsulation to provide clients an abstraction of the server, which enables polymorphic behaviour from the point-of-view of the clients. In other words: usage of interfaces and abstract classes enable a collection of objects to exhibit differing behaviour of each other. Also depends upon DI in sense that for a class to use an abstraction, it needs to be given for it. Should the class (a client) instantiate an object (a server) itself, all the dependencies of that object transfer to the class as well.

3.7 TruckTool

Case study application for this thesis work is a software application called TruckTool (TT). It is a software application created for the maintenance operations and error diagnoses of lift-trucks. It is used to visualize their various active events, sensor values and operational parameter values and names.

The application is developed by Rocla Oy and is localized to 8 different languages. It supports multiple different brands of 5 distinct types of lift-trucks with numerous models under them. The types of the trucks are automated guided vehicles (AGV's), pallet trucks, reach trucks, internal combustion trucks and electric-counter balance trucks.

TT's main features include the following: wizards for calibration and parameter setting, maintenance check lists, visualizing operational parameter values and enabling their alteration, visualising signal values and enabling their recording, exporting a pre-set parameter configurations, importing the parameter configurations from the truck to save a snapshot of the vehicles operational state and more. See figure 4 for the truck model view of the application.

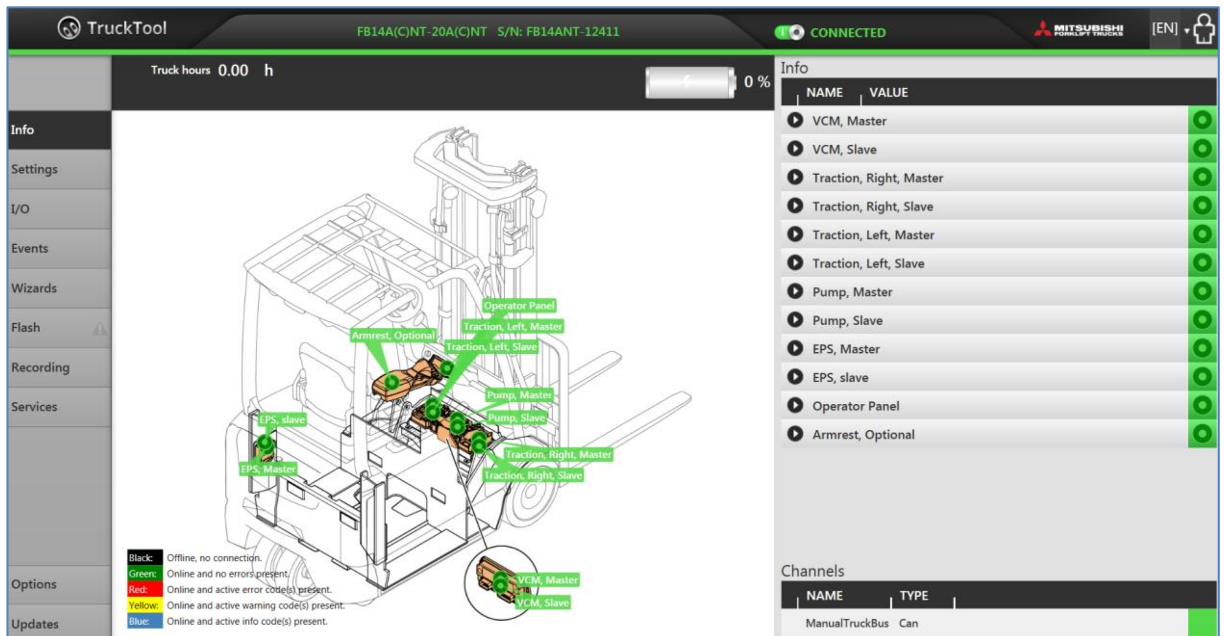

Figure 4. TruckTool model view.

The application's code base is around 130 000 lines of code according to SonarCube statistics. It's been developed for ten years and is used by the service personnel in Finland and abroad for daily maintenance operations of the vehicles.

It is a desktop application, connected to the lift-trucks with a cable or a wireless local area network (WLAN). It is capable of utilizing a serial port for communication or a CAN open implementations of CAN protocol. For the AGV's, a specific transmission control protocol / internet protocol (TCP/IP) interface is used for communications.

As the application is a commercial product, the most comprehensive and detailed UML diagrams are omitted from this paper. This will direct the thesis to concern itself more with general nature of the application's software architecture and very specific cases of architecture pattern and principle application (see section 3.5: "mapping the application" and 4.1: "Current state analysis").

3.8 Dependency mapping

Refactoring work necessitates understanding of the part of the code to be altered. For this reason, and for the sake of the CSA, an overview of the current state of the application architecture is within this section detailed.

Visual Studio 2019's Enterprise edition was used to produce code maps relating to the software architecture of TT. These maps offered a way to visualise different aspects of the application without going through all the code manually. This was important, as there was very little documentation regarding TT's software architecture and also for the sheer size of the application.

Here the attempt is to not include the whole application architecture, but merely the parts relevant to the component relocation outside the monolithic solution as its own assembly. Additionally, the section will contain mostly observations of general nature based on the notes made during the work. This in order to limit the amount of redactions for the public version of this paper.

3.8.1 Case study application's software architecture

This section will list the important parts of TT's software architecture as described by its design documentation. Many of the UML diagrams are quite old, and the current implementation conflicts with it by some measure. It is, however, important to understand the intention behind the original design to understand the current implementation.

The application is designed with 5 main components:

1. TruckToolController
2. UI
3. ProductCategoryDataModel
4. TruckDataModel
5. TruckCommunication

"TruckToolController" is designed to be responsible of the lifetime of other system components as indicated by the UML-defined "composition" lines with filled diamonds. This dictates as well, that the controller is high within the component dependency hierarchy.

Components are otherwise defined to have general dependencies to 1 or 2 other components. "TruckDataModel" being generally dependent on both the "UI" and the "TruckCommunication" components. See figure 5 for a component diagram of the setup.

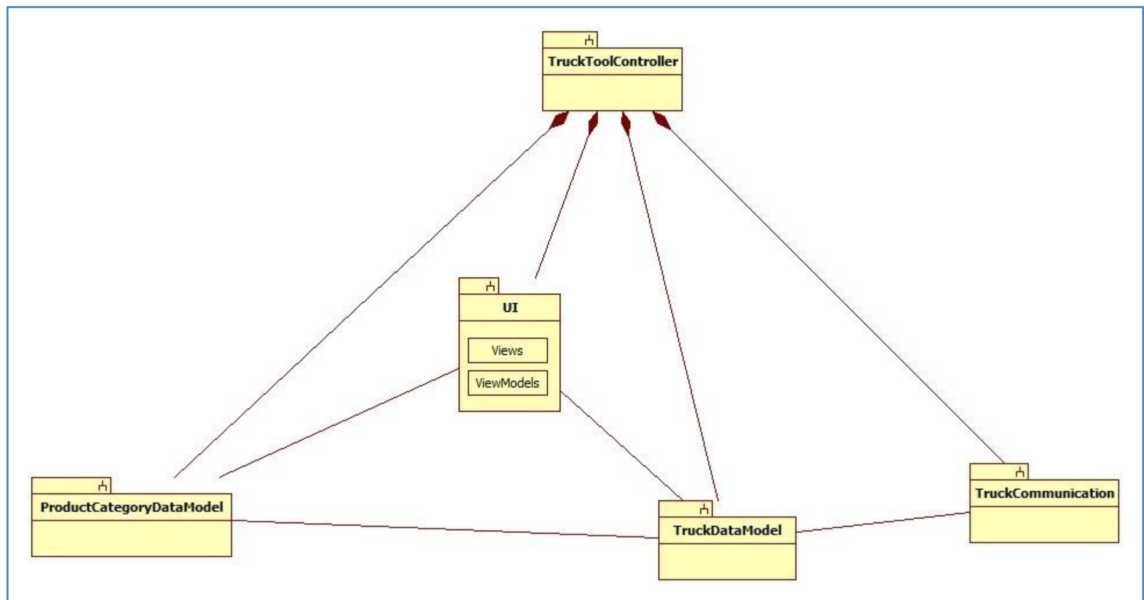

Figure 5. TT’s components as described in the original design documentation. Notation: UML component diagram

The application seems to be designed with the MVVM architecture pattern. See figure 6. More on the pattern in the section: “Model-View-View Model”.

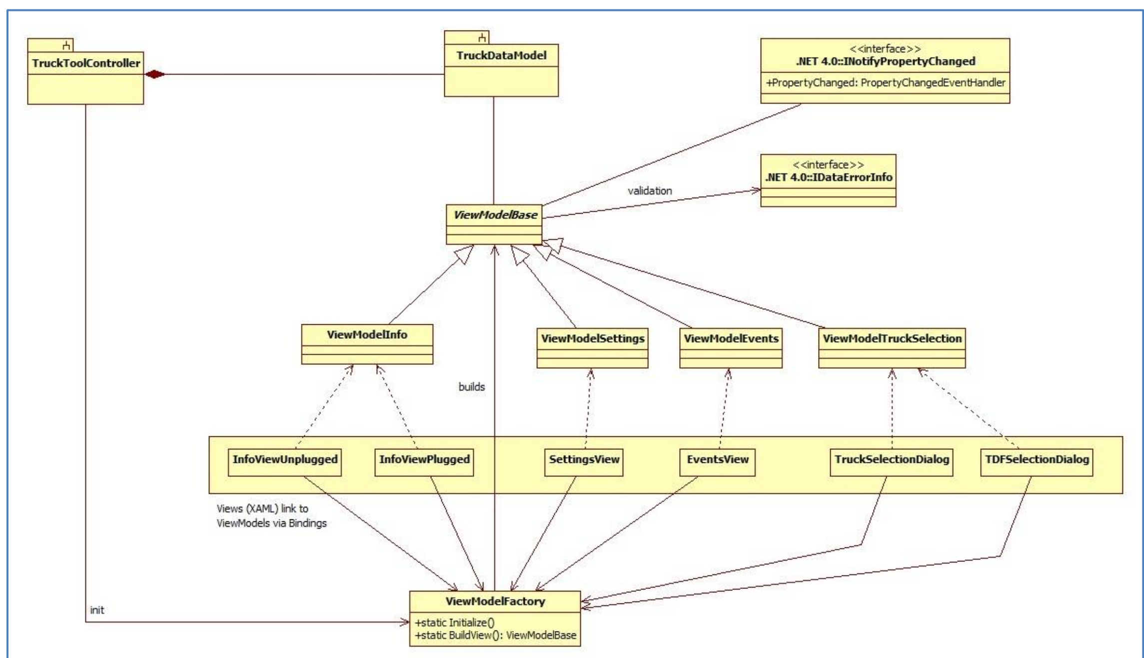

Figure 6. MVVM architecture of TruckTool. Notation: UML component diagram

3.8.2 Detailed look into the data model

As described by the earlier section and diagram 5, many components lack in-depth definition of the dependency hierarchy or are left intentionally general. This section aims to map out the relevant dependencies concerning “TruckCommunication” and “TruckDataModel” components, to get a scope of the decomposition work to be done. Notes made within this section is further referred to from the “**Current state analysis**” section.

3.8.2.1 Truck Communication functions

True to its name, the “TruckCommunication” namespace envelops the functionality required for the application’s communication with the various trucks the application supports. This section lists its crucial components, how they interoperate and their substantial dependencies.

The namespace consists of 132 files, out of which 130 are ones containing different software artefacts. It is split into several namespaces according to the communication format and generality. The communication formats are separated based on the requirements imposed by the type of connection used. There are 3 of these **formats: TCP/IP, serial, and CAN**. General functionality includes connection state management, IP address setters, truck auto detecting and truck communication, which utilizes the communication format-specific functionalities to do its bidding.

“TruckCommunicator” is an entry point to the communications with the vehicles. It is the first object created by the application controller for that purpose and to it is given as a parameter “ITruck”, a “TruckDataModel” component’s interface to “Truck” object. Based on the data of the “Truck” object, the communicator creates channel communicators, to which all the vehicles’ operational parameter information are given through “TruckDataModel” interface “IDataObject”. The communicator is responsible for the life-cycle of these channel communicators and of their functionalities’ invocation and halting.

Channel communicators are implemented for each of the connection formats individually and differ in notable ways. They share some common features however, namely the “Connect” method, which in all implementations in one way or another attach the “ICommunicables” to their handlers and “Disconnect” method for detaching them.

The communicable handlers are another set of “TruckCommunication” component’s classes and meant for storing the pending write and read operations. They are stored as, and manipulated through the “ICommunicable” interface.

“TruckConnectionStateMachine” is used by the application controller to determine the state of the connection between the truck and itself. The connection is checked through the utilization of “ITruck” interface’s “truck type” and “Status” enumerators and by subscribing to its “PropertyChanged” event, which it inherits from the .NET’s in-built “INotifyPropertyChanged” interface.

IPAddressSetter sets the network adapter’s IP when connecting to a lift-truck utilizing TCP/IP interface, ergo when Ethernet cable is used. It does this utilizing the application defined dependency container, which is quite comprehensive package of various application components. It is used only for enqueueing a pop up message for the application controller’s dialog service relating to a failure of IP setting.

3.8.2.2 Application data model

A namespace for the general data structures concerning the application and central abstractions like the truck, its controllers and parameters are defined within the data model. It includes very little business logic and is designed to be used by the majority of the application’s other components. If objects can be defined as “data with behaviour”, the data model, in general, houses objects with relatively little of the latter.

Concerning the most relevant data types the data model specifies in relation to the modularization work: the truck abstraction and its controllers and especially parameters stand out. The whole of communication functionalities is almost completely built around few of the types within the parameter inheritance hierarchy.

The truck portion of the model derives from a data object container interface, which is a “read only” collection of data type “DataObject”. This is inherited by the interface “IUITruck”, which is an abstraction of truck. It adds more of read only collections in the form of interfaces of truck controllers and data objects. Additionally it holds a collection of the wizards that are supported for the specific truck model. The “IUITruck” has multiple dependencies of “TruckToolController”, “UI” and “wizards” namespaces. The UI truck is further inherited by an “ITruck” data type, which is an interface that adds methods for

setting the user level for the trucks accessibility, for setting the trucks UI mode, and for getting the Device data types of the truck. See the visualisation of the hierarchy in figure 7.

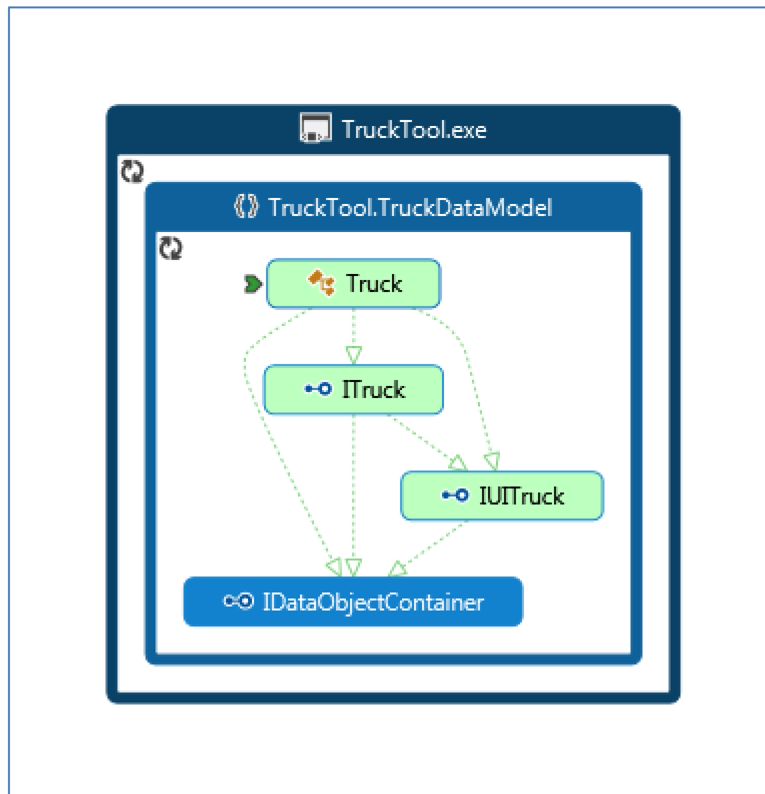

Figure 7. The inheritance hierarchy of “Truck” datatype. Produced with Visual Studio 2019 Enterprise Edition. Notation: informal

The base data type of what constitutes the truck's controller is an "IDevice" interface. It does inherit an IEquatable of "Device" however, which instils the implementers to implement "equals" method for "Device" comparisons. Makes also the interface dependent on its own implementation (more on this in CSA section). The IDevice interface declares a get method for the controller specific wizards like the flashing and controller change wizards. It declares also a get method for the controllers' data objects, firmware file info list, error history, status and its parent "Truck" data type. The "IDevice" is implemented by "Device" class, which is the data type used to describe the controller of the truck in the data model of TruckTool. See the visualisation of the hierarchy in figure 8.

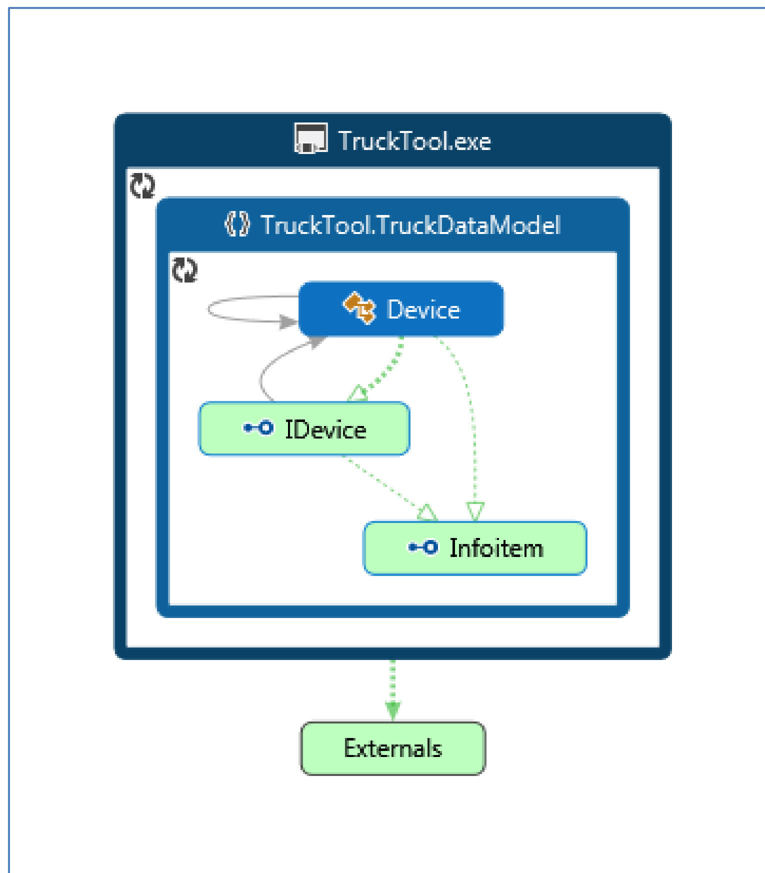

Figure 8. Device inheritance hierarchy. Produced with Visual Studio 2019 Enterprise Edition. Notation: informal

Operational parameter of the truck's controllers are described at their base level by the data type "IRegisterable", which declares just the get methods for the object's identifier and a list of communication formats. The latter declares channel's ID and settings. The "IRegisterable" is inherited by "ICommunicable" which declares a hefty amount of functionality for the subsequent implementers and inheritors. It's most notable contributions are the declaration of get and set methods for the parent controller in the form of "IDevice", parameter's raw value and polling priority. It is further inherited by "IDataObject" interface, which details methods for getting and setting the range of allowed values, image sources and paths, visibility and accessibility settings. This IDataObject is inherited by an abstract class "DataObject", of which the multiple different specialisations are derived. See figure 9 for the relevant portion of the inheritance hierarchy of the datatype.

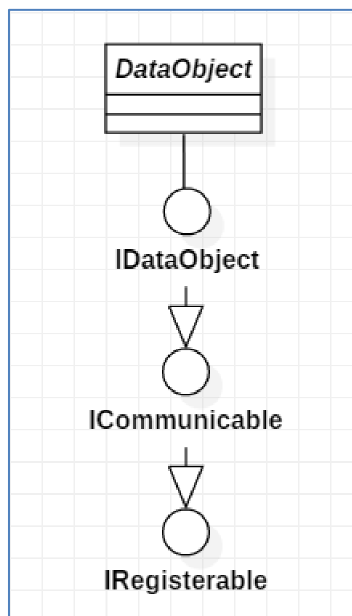

Figure 9. Inheritance hierarchy of "DataObject". Notation: UML class diagram.

4 Results and analysis

4.1 Current state analysis

This section details notes, findings and conclusions of the current state of the TT's monolithic architecture. It is based on the held meetings with the development team and the conducted survey of the application's code base (see section 3.8: dependency mapping). This section should not be taken as an assessment of the overall design of the system, but rather as a focused overlook on the composition of the system in relation to the decomposition of the vehicle communication functionalities assigned as one of the thesis work's objectives.

For the modularization work, defining the dependency hierarchy of the components is essential. Artefacts inside two namespaces within a single project may depend on each other but the same cannot hold true, when those namespaces reside within two separate projects. As described before, this would make the projects cyclically dependent on each other, thus nullifying whatever benefits they would bring to the table in terms of modularity.

TT is designed to utilize MVVM architecture pattern. See figure 6. Acknowledging this is important, as the system composition and component hierarchy should then adhere to it to some level at least. Another defining factor to take into the account as well is the application controller, which seems not to follow the pattern but seems to stand outside of it to control the setup of the system, view to be shown and access the file structure. To do this, it has references to each of the MVVM's components. Due to the inclusion of the controller, the application could be said to implement a "MVVMC": "Model-View-View model-Controller". Although the acronym seems to not be mentioned anywhere, figure 5 indicates this to be the case, as the three components of MVVM are designated by aggregation to exist due to the controller.

TT's main project is divided under 6 main namespaces. They are the following: "TruckCommunication", "TruckToolController", "TruckDataModel", "ProductCategoryDataModel", "UI" and "Wizards". These, with the exception of wizards, are illustrated in the component diagram in figure: 151. The diagram leaves all dependency-definitions generic, except for the ones with TruckToolController. For example, TruckDataModel and TruckCommunication are defined to have a generic dependency, although, according to

the meetings with the development team, the communication functionality should depend on the model and not the other way around. These are important definitions when one does modularization work such as the one in this thesis, as enforced dependency hierarchy by way of shared libraries should not form cyclical dependencies. They are important for the developers as well, as they guide the development so that the systems components remain modular and not interdependent.

Interdependency between the system's components makes running unit tests a slow process, as the majority of the time sinks into the compilation of the one massive interdependent system. Unit test execution times of over 5 minutes were clocked during the work, where approximately 2 minutes went into running the tests and the rest into the compilation process.

Communication functions are tightly coupled to the application's data model through the abstractions related to the truck, its controllers and parameters. In fact, the whole communications namespace relies on the data model's truck entity, which is given as a parameter to the main communicator. This is problematic from the viewpoint of modularization, as the truck entities' dependencies sprawl to the rest of the application and ultimately to the UI level. However, a generic dependency between the data model and the UI is defined within the figure 5 and so this concern of tight coupling should not be considered a fault in the implementation but rather a design related specification issue.

Communication functions are also somewhat coupled to the application's controller namespace, although not as much as to the data model. The controller namespace, however, is specified by the figure 6 to have dependency over the whole application. This, unlike the dependencies towards the data model, present refactoring work, as communication functionalities should not depend on the controller's namespace and controller's functionalities are not to be decomposed into their own library within the scope of the thesis work.
Most of these dependencies come in the form of "Translation" namespace invocations. The often-most occurrence seems to be for giving the translated message for communication result to the view models in charge of the current process within the application. These are problematic, however, as fetching them requires file structure access given by the controller.
Additionally, communication functionalities namespace includes an IP address setter, which accesses the application controller to request modification of the used internet

adapter's IP address when connecting the computer to an AGV. This presents a challenge in a same manner as the translations, since the application controller namespace is used to access the file structure for application configuration settings.

The interfaces used to access "Truck" functionality derive from a list of parameter abstraction which has numerous UI, and application controller related dependencies (more on this in "Dependency mapping"). This means, that every functionality using the Truck interfaces is coupled to those namespaces.

Communication functionalities use the truck abstraction for the communications, but due to its inheritance hierarchy, it is coupled to the UI elements of the application. Communications are this way coupled to the UI components of the application.

Truck data model depends on the "wizards" namespace through at least the "IDeviceWizard" interface. These present a problem to the modularization, as the namespace includes wizard-specific view models to which data model should not depend on.

There are some instances, where an interface depends on its own implementer. Although not strictly forbidden, any class that uses the interface becomes dependent also on the referenced implementation and its dependencies. This also couples the interface and its implementation to each other which means, that the system is that much less modular. Instances of this are the following: "IDevice" inheriting an IEquatable of "Device". "IDeviceStatus" having as a property one of its implementor's inner class instance, "IZapiFlashVersion1Settings" defines an enumerator property, which is defined inside its implementer. "IMNSerialFlashData" depends on its own implementer's inner class "MNSerialFlashBlock".

TT's main application targets .NET Framework version 4.5.2. This makes it incompatible with .NET Standard 2.0, which is targeted by some projects instructed to be used for the modularization. To utilize the .Net Standard 2.0, the projects have to be updated at least .NET Framework version 4.6.1 [21].

As a summarization of the CSA:

- The application is designed with MVVM architecture pattern with an application controller to instantiate the other components.
- The communications component is very dependent on the data model of the application which reflects the original design. It is very hard to separate the two.
- The data model of the application has dependencies towards the UI component through the “truck” datatype directly and indirectly by its dependency of “wizards” component.
- The communications component is dependent on some of the application controller's functions but refactoring these out should pose no major challenges.

4.2 Application modularization

This section details how the decomposition work was done given the findings of the CSA.

The decomposition method (see section 2.2) outlined dependency hierarchy definition as a requirement for the work, as it was needed for determining whether dependencies should be decoupled or just moved to a separate project. The component diagram in figure 10 was produced to depict this hierarchy.

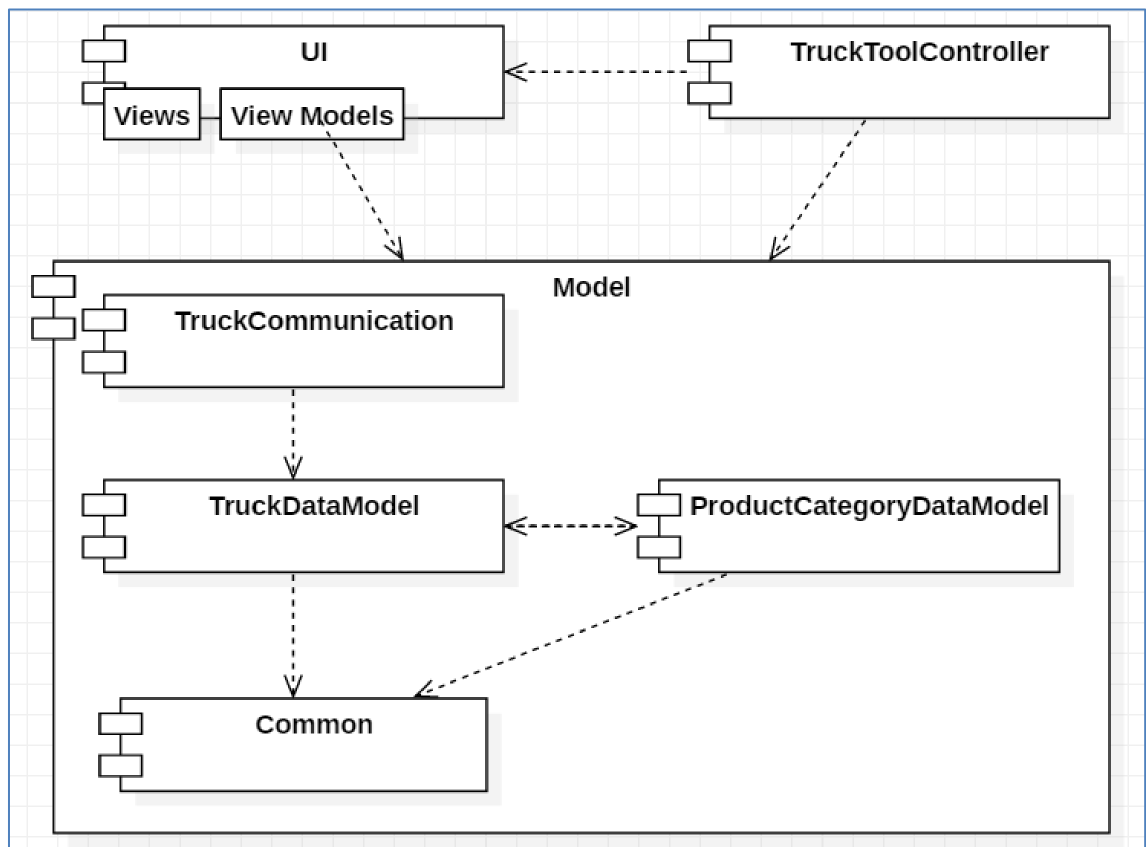

Figure 10. Mapping of the system component dependency hierarchy. Notation: informal

The dependency hierarchy diagram of figure 10 could be used to verify, which dependencies of the implemented system were valid. Valid dependencies could either be refactored out or moved to their own separate project. Although the creation of new modules outside the communication functionalities one was not a requirement, it was deemed necessary for limiting the work to reasonable limits. Therefore, as depicted by the diagram, “TruckDataModel” and “Common” namespaces were in parts moved to separate projects, of which DLL’s are compiled.

Conversely, any dependencies that the truck communication functionalities had towards application controller or the UI had to be refactored out, as they were dependencies that should not exist.

“TruckDataModelLite” and “Common” projects were created to target .NET Standard 2.0 framework. As .NET Framework 4.5 does not support the utilization of the framework version, the main application had to be updated. Although v.4.6.1 would have sufficed, .NET Framework 4.7.2 was chosen. This has introduced no major issues at the writing of this thesis, but the “Parse” method calls of the .NET class “Double” had to be updated to include culture specification involving decimal separator.

As mentioned in the CSA, the namespace “TruckCommunication”, which was to be refactored out of the monolithic solution was tightly coupled to the “DataModel” namespace. That namespace itself is intricately coupled to the rest of the application so to just make it onto an assembly of its own as it existed was not a sufficient resolution to the problem. What was done essentially was that two other modules or shared libraries to support the new TruckCommunication module had to be created as well: “TruckDataModelLite” and “Common”. Both existed within the monolithic solution already and were in limited capacity relocated outside of it to these pre-existing projects. See figure 11.

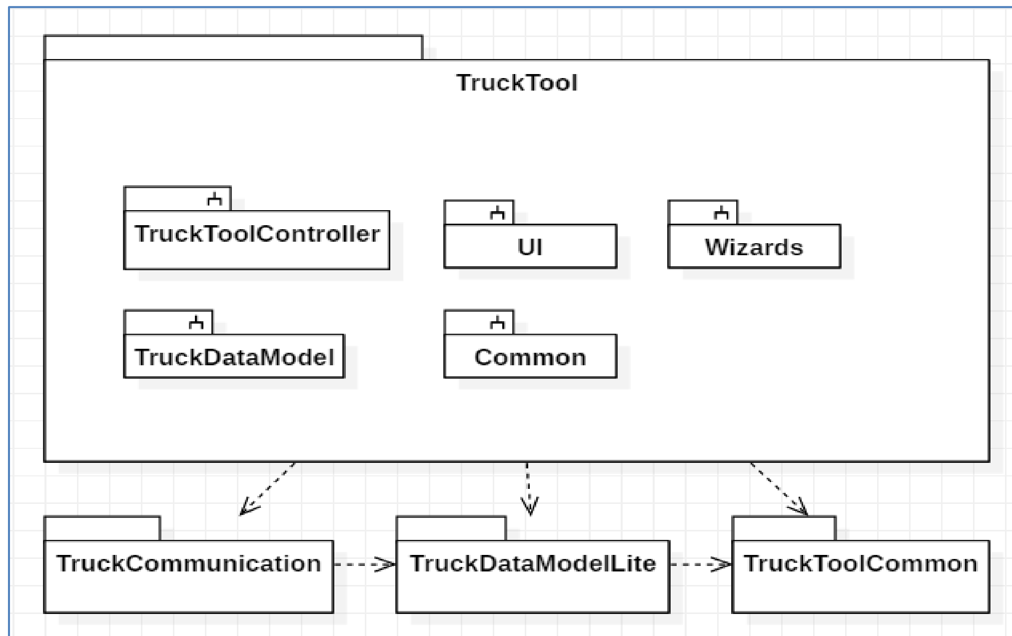

Figure 11. Application projects after the modularization. The three packages below the main project “TruckTool” were created as a result of the modularization work. Notation: UML package diagram.

From the viewpoint of the modularization work, the data model interfaces used within the vehicle communications component were too extensive. Especially the truck and data types representing its controllers and parameters tied it to the code base quite intricately. To remedy this, the ISP was applied to produce interfaces more suited for the component. Specifically, the data types of the controllers and the truck demanded a new version for the communications module. To this end, “ICommunicableDevice” and “ICommunicableTruck” interfaces were created to be implemented by the pre-existing “Device” and “Truck” classes respectively. They had only the barebones-dependencies needed by the communication functionalities.
The data type for parameter had a communications-intended version already: the “ICommunicable”, but minor alteration had to be performed to limit its coupling to the data model. Namely, the new interfaces had to be declared and implemented as properties for the ICommunicable in the place of the old ones. The old “Device” property of the “ICommunicable” was a dependency which could not be injected into the TruckCommunication component, and was pushed down to the “IDataObject” level. In its place, a “CommunicableDevice” property was placed. It was then implemented by the “DataObject” to return the old “Device” instance but downcast as the new “CommunicableDevice”. This is an example of OCP: the DataObject behaviour was not changed but extended to give out its parent Device as a datatype client specifically. More generally related to

OOP: it serves as an example of inheritance, as the property given out is downcast to a more general datatype and as an example of encapsulation by giving out an interface. Both are concepts by which loose coupling is produced. See diagram 12 for the old inheritance hierarchies and 13 for the new.

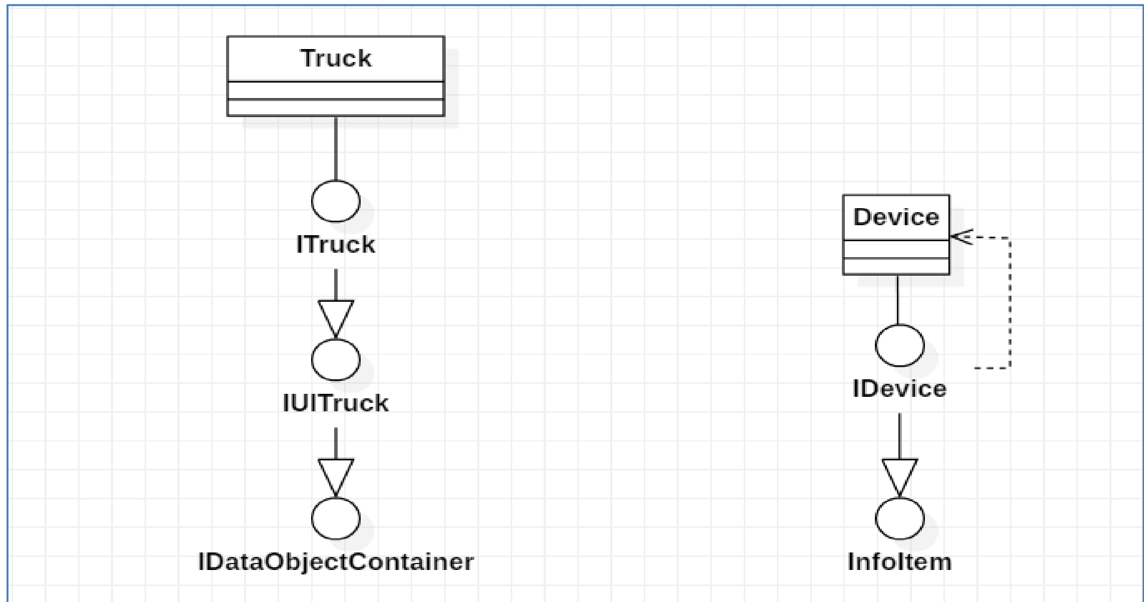

Figure 12. Old inheritance hierarchy of the data model classes “Truck” and “Device”. Notation: UML class diagram

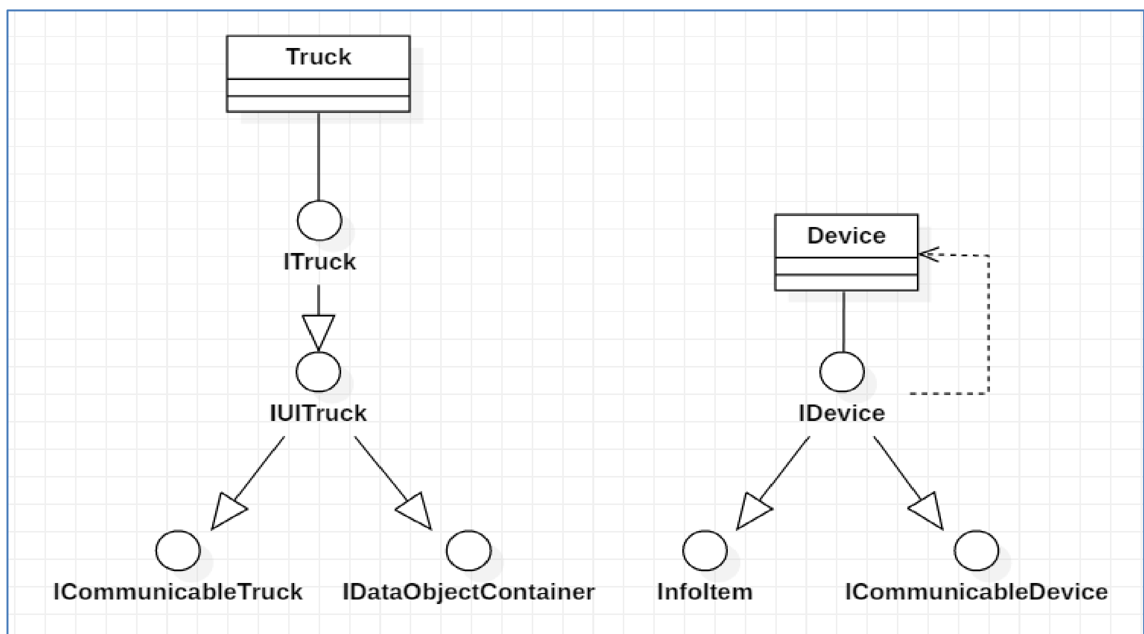

Figure 13. New inheritance hierarchy of the data model classes “Truck” and “Device”. Notation: UML class diagram

Truck communications namespace had multiple instances of Translations namespace functionality invocations. As discussed in the CSA, that namespace is part of the application controller, and should not be used by the “model” part of the application’s code base. Those were refactored out by moving the concerning texts to be translated elsewhere. Some were moved to the application controller, others to subwizard view models and a single instance of translation was removed as obsolete. The view model would seem to be the correct place to perform the translation, as it should generally hold the presentation logic for the views in MVVM pattern. In practice the translations were moved by returning the translation ID from the original place and utilizing it then in view model or application controller.

As a summarization of the modularization work:

- The contents of the “TruckCommunication” namespace were decomposed out of the monolithic project to its own in its entirety with the exception of two classes, which were specified to belong under another namespace.
- “TruckCommunication” had been designed and implemented to depend upon the data model of the application. The relevant portion of data model was relocated into another project to accommodate this, as the communication component could not depend on the main project wherein the data model was situated.
- The refactoring done to enable only partial relocation of the data model included the creation of communications specific interfaces for some of the used classes.
- To remove dependencies the communications component had on the filesystem of the application, the UI translations were refactored to be handled in the view models and the application controller.
- Communications related unit tests were moved into a project of their own.

4.3 Analysis of modularization effects

This section will examine the effects of the decomposition work done within the thesis. Specifically relating to the modularization of the vehicle communications functionalities into their own project. Here the focus will be on the KPIs introduced in the methods section:

1. Unit test running time
2. Compilation time
3. Application RAM-usage on run-time
4. Communication speed

The benchmarking results and analysis to follow in the order of the list above.

As a summarization of the differences of the modularized and the original application:

- Original runs on one monolithic project which includes the *user interface, the business rules, and manipulation of the data all in one.* The modularized version has split the whole communications section into its own project.
- The original is using .NET Framework 4.5 whereas the Modularized .NET Framework 4.7.2.

4.3.1 Unit test running times

Of the case-study application's unit test suite, approximately one fourth targeted the communication functionalities. They were moved on to a project of their own with the project references to only the ones created during the modularization work. These were the project which now encompassed the vehicle communications functions, a project with part of the data model and a project with some helper classes. Like stated in the CSA, the unit tests took a considerable time to even start being executed due to the large, monolithic solution that had to be compiled beforehand. As a result of the modularization work, however, the communication functionalities have been separated from the monolithic solution, and is now considerably lighter to compile.

Setup for benchmarking the unit test running times:

- For both the original and the modularized version, the run unit tests are the same and pass.
- 5 unit test runs for each version are recorded.
- Modularized version's previously assembled DLL's are cleaned between the test runs to make the compilation equal to the original.
- MSBuild's parallel builds configuration is set to 4.

Table 2 includes the unit test running times for the original and the modularized version of the application. The recorded times include unit test running times specifically and with the associated build procedure, where the tests library and the associated projects are compiled.

Table 2, row 2 shows a 3 second (2%) increase in unit-test runtime: 2:23.12 - 02:26.45. This could indicate minor decrease in computational efficiency associated with the increased overhead of DLL usage.
Row 3 confirms a small increase in communications specific unit-test runtime of ~1 second.

Table 2, row 3 shows a 23 second (6.5%) decrease in unit-test runtime **with** the associated build: 06:18.56 - 05:54.97. This was most likely due to the utilization of the MSbuild's parallel project building. As a sizable portion of the application's code was decomposed into its own project, the method could now be utilized for it for minor decreases in compilation time.

Table 2, row 5 shows a 226 second (72%) decrease in communications specific unit-test runtime **with** the associated build. This was an expected benefit of the modularization due to the division of the main application into smaller, more accurately targetable units.

Overall, table 2 demonstrates a clear improvement between the times of unit test execution for communications component, while the execution of all tests showed minor increases in total time. Decrease in test-associated build time was a surprise, which was due to the opened possibility of building more projects in parallel.

Table 2. Unit test running times.

	Original TruckTool	**Original Mean time**	**Modularized TruckTool**	**Modularized Mean time**
All unit tests	02:32.19 02:19.60 02:14,79 02:25,89 02:23,17	02:23.12	02:30.76 02:33,35 02:28,57 02:34,50 02:20,83	02:26.45
All unit tests + build	06:25.41 06:35.95 05:56.70 06:15.59 06:19.11	06:18.56	05:59.60 06:09.03 05:58.52 05:48.61 05:39.10	05:54.97
Communication related unit tests	01:23.08 01:20,42 01:22,16 01:25,53 01:21,19	01:22.62	01:19.91 01:19,57 01:23,11 01:26,14 01:27,18	01:23.32
Communication related unit tests + build	05:12.36 05:09.25 05:10.52 05:15.17 05:26.70	05:14.80	01:27.07 01:26.74 01:26.25 01:30.02 01:30.61	01:28.13

4.3.2 Compilation times

Application compilation is part of the CI-pipeline and as such an important factor to consider when assessing deployment time. As a complete rebuild is done for every deployment of a new version, it is done for this benchmarking as well.
This section lists the compilation times of the modularized and the original version of TT.

Benchmarking setup:

- Clean applied to each project associated with the deployment of the application.
- Microsoft's Log Viewer was used for logging the builds and MSbuild build--machine for the building.
- 10 timings for each version
- Entries are the reported values of the build machine

Looking at the data of table 3, one can observe slight variation of the data samples between entries. For the original, the fastest compilation executed in 181908 and slowest in 209207 – a difference of ~13%. For the modularized version, the variation can be observed to inhabit approximately the same range: 190649 – 219842.

An increase of 4.3% in compilation time can be calculated from the mean times. This was a surprise, as the compilation times together with the unit tests produced results to the contrary – therein the compilation times had decreased on average due to parallel project building. However, the compilation process was slightly different, as there the used test adapter decided the projects that were built. Since the build process was logged with "MSBuild structured log viewer", the processes could be analysed to see the cause for this.

Table 3. Rebuild times. Units in milliseconds.

	Original TruckTool	Mean Original time	Modularized TruckTool	Mean Modularized time
Compilation times	219207 185303 181908 193545 195285 193724 199906 196125 193785 189408	193820	190649 205275 219824 196557 191280 195861 205117 205347 208277 203276	202148

Upon investigating the build logs, it was noted that the build did not utilize parallel project building – the factor deemed to have caused the decrease of compilation time for the unit-test associated builds. This was due to builds being ran from the MSBuild Structured Log Viewer, which had disabled the feature on default. The compilation times were then benchmarked again to verify.

Benchmarking setup:

- Microsoft's Log Viewer was used for logging the builds and MSbuild build--machine for the building.
- Build optimization configured for 4 processors.
- 10 timings for each version.
- Entries are the reported values of the build machine

Table 4 shows the rebuild times with the parallel project building. Here one can observe considerable reduction from the mean values reported in table 3 for both the original and modularized version: 193820 – 177914 ms (8.2%) and 202148 - 177765 (13.1%) respectfully. Parallel project building can be said to have a significant effect on the compilation times for both, although for the modularized version the effect is greater. So much so in fact, that it on average has a 2.1 seconds (1.1%) faster compilation time compared to the original version of the application.

Table 4. Rebuild times with **build optimization.** Units in milliseconds

	Original TruckTool	Mean Original time	Modularized TruckTool	Mean Modularized time
Compilation times	178302 177014 176764 178241 178850 176928 173425 185477 178753 175382	1779136	174513 184279 172062 174109 179985 175822 173951 174720 174459 173660	175756

4.3.3 Application RAM-usage on run-time

For benchmarking the memory usage, a pre-set sequence of actions were devised to invoke the use cases of the application. For automating the actions, Robot Framework was utilized.

The use-case to be acted out was chosen to be the importation of vehicle's parameter values.

Benchmarking setup:

- 40 runs for each.
- No disconnect of the application from the testing board is done between the imports.
- Computer reboot was issued before each run of the set
- One lift-truck model for each of three communication formats is chosen.

In the two runs displayed by the figures 13 and 14, both versions seem to allocate the same amount of memory. Very minor yet steady increase of memory allocation can be observed in both. Approximately 20MBs of memory is allocated during the 40 imports.

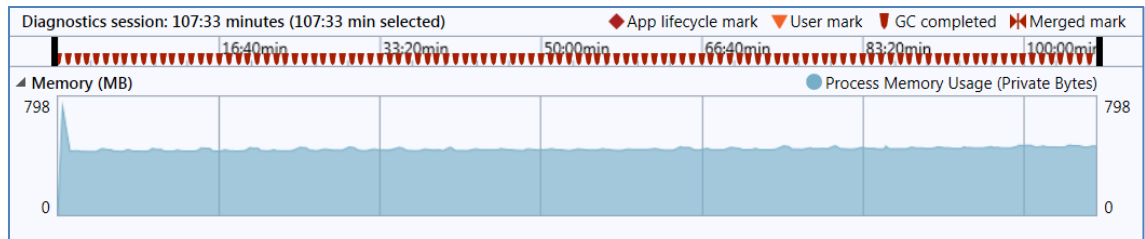

Figure 14. Memory usage of the application during the importing of the parameter values with the original version and with CAN communication format.

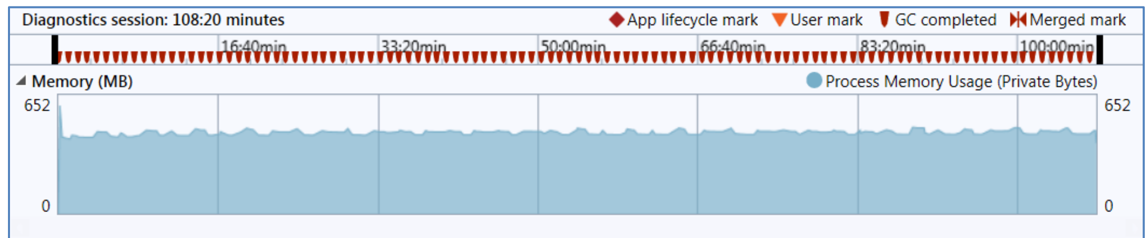

Figure 15. Memory usage of the application during the importing of the parameter values with the modularized version and with CAN communication format.

When importing the parameters with TCP/IP, a substantial and steady increase in the memory allocation for both versions seems to occur during the script run-time of 45 minutes. An increase of 400MB to 1GB in allocated memory can be observed with the over-all memory usage being similar between the versions. As the issue affects both versions, the cause for the seemingly unintentional memory allocation is left un-investigated for the thesis.

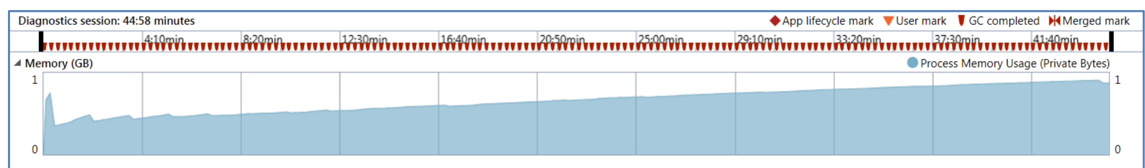

Figure 16. Memory usage of the application during the importing of the parameter values with the original version and TCP/IP communication format.

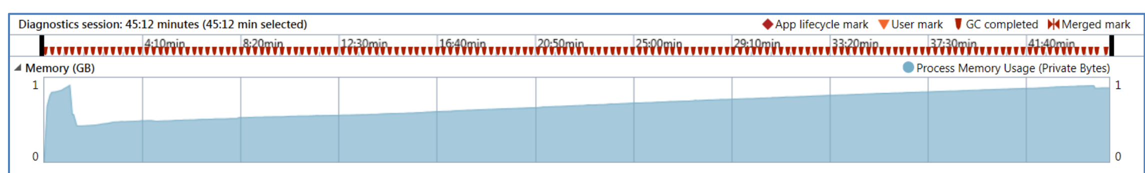

Figure 17. Memory usage of the application during the importing of the parameter values with the modularized version TCP/IP communication format.

Import runs utilizing serial communication format are displayed in the figures 17 and 18, the original version would seem to use less memory on average but more at its maximum. Like with the CAN import run, a minor but steady memory allocation can be observed for both. An average of 20MB is allocated during the 30 minutes of import taking.

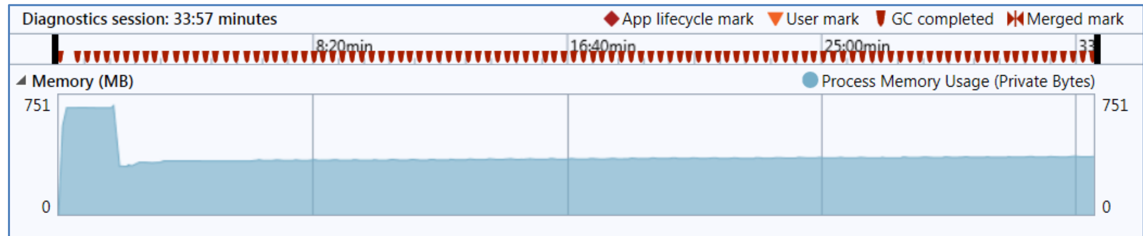

Figure 18. Memory usage of the application during the importing of the parameter values with original version of the application and serial communication format.

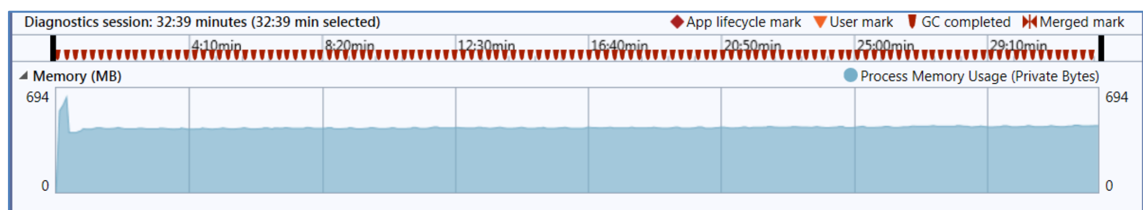

Figure 19. Memory usage of the application during the importing of the parameter values with the modularized version of the application and serial communication format..

Overall, the modularization work seems to have caused less increase in memory usage than anticipated and in one case even lessened it. Since the modularization work updated the application's .NET Framework to version 4.7.2, the improvements to the garbage collection [22] may be the source of the more efficient memory management. However, this is left to be investigated outside the thesis work.

Summarizing the benchmarking results for the memory usage:

- No signs of unintended memory allocation caused by the modularization work could be found.
- Original and the modularized version seem to allocate the same amount of memory.
- A possible bug concerning memory allocation when using TCP/IP communication format may be present for the modularized and the original versions of the application.

4.3.4 Communication speed

As the modularization work separated the code into multiple dynamic libraries and since their usage entails some overhead, it is probable that some changes to the communication speeds with the vehicles occurred. This section compares changes to the speeds of communications by listing benchmarked times of taking an import of the parameter values of the lift-truck with the application.

Setup for the benchmarking:

- One lift-truck model for each communication format is chosen.
- Lift-truck testing boards are used.
- The timing was automated so, that it starts at the method call of the import functionality and stops when the call has run its course.
- Import time reports are to be automatically produced by an embedded script in the application code.
- The import is run for 40 times consequently without disconnecting the application and the testing board in between.
- Robot script is used for automating the UI actions. See appendices 10 - 12.
- Units are shown in milliseconds.

See appendices 4 – 9 for the results of the import time benchmarking.

The average import times displayed in table 5 show very minor increases (1%) in communication speeds on average. Minor decreases can be observed for serial and CAN communication formats and minor increase (3.5%) for TCP/IP.

Table 5. Average of the benchmarked import times. Units in milliseconds.

	Original TruckTool	Modularized TruckTool
Import - CAN	139457,5165	140082,0514
Import – Serial	18506,7	18684,4
Import - TCP/IP	5254,363	5058,465

Figure 20 depicts a general tendency of the import time increasing for both application versions when importing parameters with TCP/IP communication format. Both versions

start at approximately 2.5 seconds and keep increasing to over 6 seconds. As the increase affects both of the versions, its cause is left to be investigated outside this thesis.

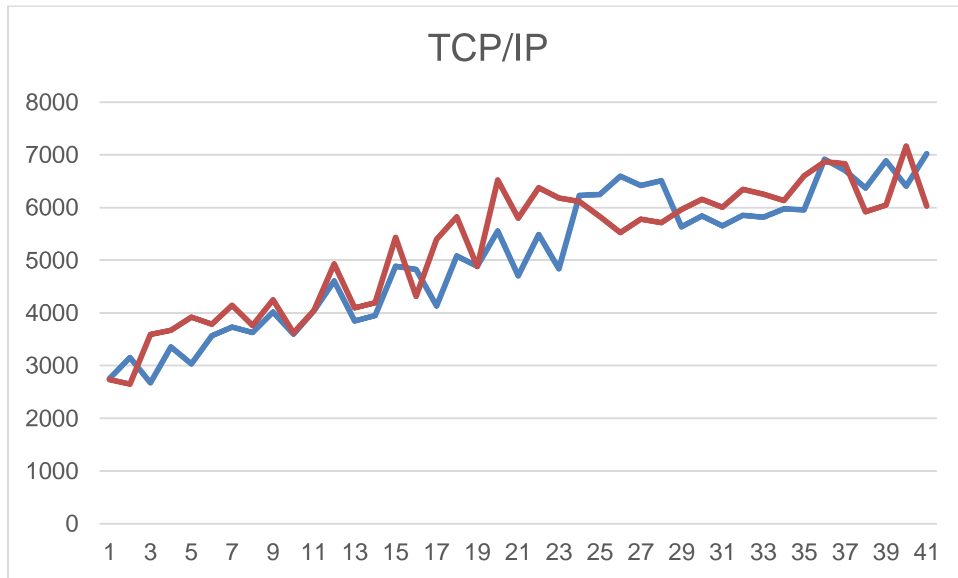

Figure 20. Graph of import times with TCP/IP communication format. Red line indicates the original version and blue the modularized one. Vertical axis indicates time of import and horizontal the number of import.

Figure 21 depicts the import times with serial communication format. Both application versions maintain a steady import time of approximately 18 seconds.

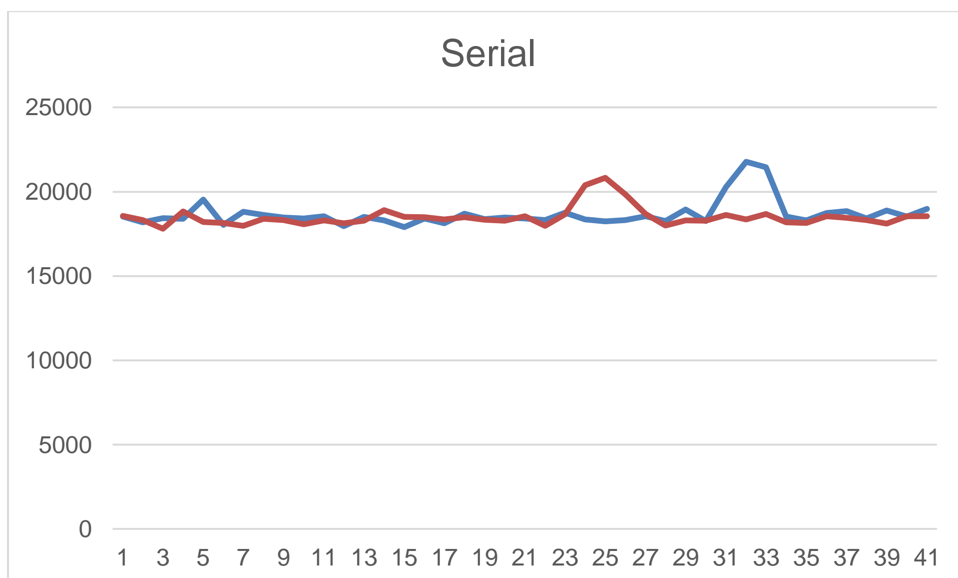

Figure 21. Graph of import times with serial communication format. Red line indicates the original version and blue the modularized one. Vertical axis indicates time of import and horizontal the number of import.

The import with CAN communication format takes longest of the three. No increase of import time can be observed from the sample size. See figure 22 for graph of import times taken.

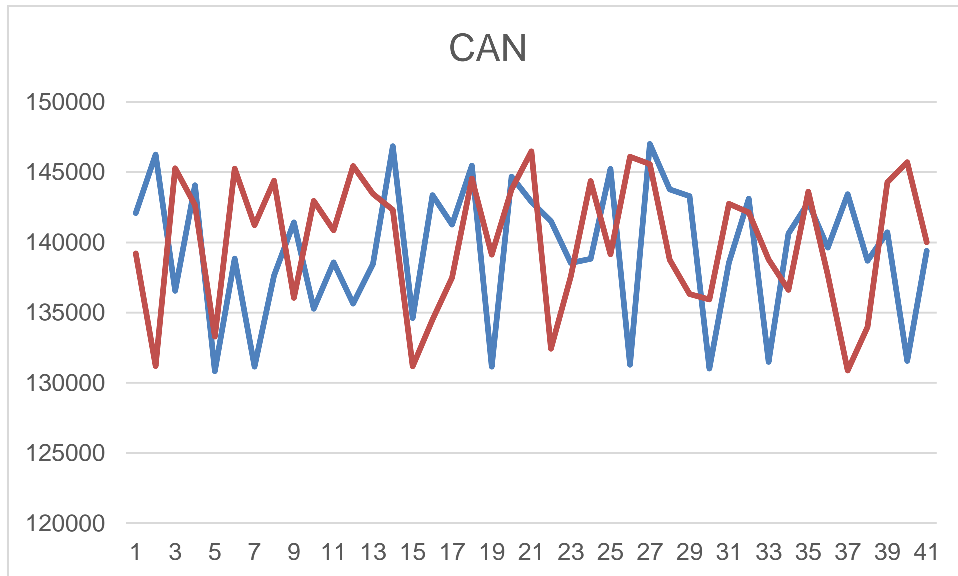

Figure 22. Graph of import times with CAN communication format utilization. Blue line indicates the original version and red the modularized one. Vertical axis indicates time of import and horizontal the number of import.

5 Discussion and conclusions

5.1 Summary

The objective of the thesis was to research how a monolithic software application could be decomposed into multiple libraries. POC module was to be procured using this research and then analyse how its use would affect the testability, maintainability and readability of the application.

The key part of the work was to produce the procedure for the modularization. It was developed with the development team of the case study-application and included steps concerning UML diagram examinations, design definitions, refactoring and testing. Research was conducted to understand how the application should be structured and refactored. For the composition of the system, the architectural patterns that its development followed were examined and discussed. The refactoring work was conducted together with the development team and by the utilization of OOP-related principles like SOLID and DI.

The project work identified and updated design definitions of the system as it relates to the component dependency hierarchies. The proceeding modularization of the system followed these definitions to decompose a shared library project from the monolithic solution. Analysis of the modularization's effects showed major improvements in the testability of the application through the capability of running unit-tests against the decomposed project in isolation from the rest of the system. Build optimization could be further taken advantage of to build the new communication component in parallel to the main application resulting into moderate decrease in overall compilation times. Additionally, minor decreases in memory usage were observed. However these were most likely due to improvements relating to .NET Framework update.

The key findings of the work were the realisation of the level of inter-dependency of the case-study system's components and the absolute importance of encapsulation by interface utilization when creating modular systems.
It was generally noted during code analysis, that DIP and "code to an interface" principles were widely used in the case-study application's code base. LOD violations could not be

identified either. However, many interfaces in very central position of application's functionality were quite extensive and used extensively. Examples of these are the Truck, Controller and Parameter types, to which ISP could be applied.

5.2 Next steps

With the modularized version of TruckTool, analysis of its effects and this paper, the software management at Rocla may assess whether the work produced the modularized version of the application that was hoped for, and whether the original goals were met. If so, the feature version branch where the modularization work was conducted may be used in the development. If not, then the work has at least shed light into the composition of the software system and to the level of its components' inter-dependencies, which is critical information for the future work in regards to making the system more maintainable and testable, while more features are added.

5.3 Objective vs. Results

This section examines the level by which the set objectives met the results of the work.

The introduction laid out the objectives as the:

1. "gathering of information for the decomposition of the system into loosely coupled modules"
2. "procure a POC module out of the application's code base"
3. "analyse the effects of modularization"

For the information gathering, multiple different OOP and OOD-related principles were examined. SOLID, LoD and dependency injection were identified as principles which help in creation of loosely coupled code. These were examined and applied during the work. The concepts of cohesion and coupling were also touched upon, which aid in the identification of components and their dependency hierarchies – crucial information regarding decomposition of a system. Additionally, procedure for the modularization was formulated together with the development team.

As to the POC module: one was created with the employment of the information gathered and together with the development team. It may be used in separation of the parent

application, something that cannot be said for the original version's communications component. Notable specific use case for this being the capability to test it independently.

Analysis of the modularizations effects were conducted based on development team feedback on the interesting KPI's to evaluate.

Overall, the objectives can be said to have been met.

5.4 Final Words

Retrofitting structure into a monolithic software application can be very challenging. While this paper mainly examined principles for building modularity into individual software artefacts, less attention was shown towards the methods by which to define the components of an interconnected code base. This lack of attention was made possible by the cohesiveness of the contents of the namespace containing the communication functionalities, which were to be decomposed as their own assembly – it enabled the selection of software artefacts to be done mainly based on the namespace. However, should the work continue, much more of an analysis would have to be conducted to understand how the application should be divided into different components.

Utilization of projects within an application solution seems to be a viable way to force a certain architectural design. This has clear advantages on multiple fronts: faster system understanding, increased testability and component reusability, build optimization, component specific framework configurations, easier responsibility delegation across teams and more. The methods and principles examined within this paper enable these benefits, which are crucial for creating sustainable, long living software projects.

References

1. Beck, Kent; Roberts, Don; Brant John; Opdyke, William; Fowler, Martin. “Refactoring: Improving the design of existing code”. 1999. Addison-Wesley Professional
2. Löwy, Juval. “Programming .NET components” 2005. Sebastopol CA O’Reilly. ISBN: 0-596-10207-0
3. Bass, Len; Clements, Paul; Kazman, Rick. “Software architecture in practice”. 2005. Boston, MA: Addison Wesley
4. Martin, Robert. “Agile Software Development: Principles, Patterns, and Practices”. Pearson Higher Education. International edition. 2013
5. Seemann, Mark. “Dependency Injection in .NET”. 2012. Manning publications Co. ISBN: 978-1-9351-8250-4
6. Richards, Mark. “Software Architecture Patterns”; 2015. O’Reilly Media, Inc.
7. docs.Microsoft. “Three-tier Application model” [Internet] available from: https://docs.microsoft.com/en-us/previous-versions/office/developer/server-technologies/aa480455(v=msdn.10). accessed 22.7.2019
8. Martin, Robert. “Clean Architecture: A craftsman’s guide to software structure and design”. 2017. Prentice Hall
9. Plakosh, Daniel; Seacord, Robert C; Grace A. Lewis. “Modernizing legacy systems: software technologies, engineering processes, and business practices”. Addison-Wesley Professional. 2003
10. Fowler, Martin. “Continuous Integration” [internet]; available from: https://www.martinfowler.com/articles/continuousIntegration.html. Accessed 26.7.2019
11. PepGoTesting. “Continuous integration” [internet]. available from: http://www.pepgotesting.com/continuous-integration/. Accessed 26.7.2019
12. Gossmann, John, “Introduction to Model/View/Viewmodel pattern for building WPF apps” [internet], available from: https://blogs.msdn.microsoft.com/john-gossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/. Accessed 28.7.2019
13. Foote, Brian; Yoder, Joseph. “Big ball of mud”. [Internet] 1999. Department of Computer Science University of Illinois. Available from: http://www.laputan.org/mud/. Accessed 4.7.2019

14. docs.Microsoft. “The MVVM pattern” [Internet] available from: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10). accessed 30.7.2019
15. documentation.devexpress; “MVVM-Framework” [Internet] available from: https://documentation.devexpress.com/WPF/15112/MVVM-Framework. accessed 30.7.2019
16. docs.Microsoft. “C# 6.0 draft specification” [Internet] available from: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction; accessed 1.8.2019
17. Clark, Dan. “Beginning C# Object-Oriented Programming - Second Edition”- Springer science + Business media. 2013; ISBN 978-1-4302-4936-8
18. Joshi, Bipin. “Beginning SOLID principles and design patterns for ASP.NET developers”. Springer science + Business media. ISBN 978-1-4842-1848-8
19. Dooley, John. “Software Development and Professional Practice”. Appress. Berkeley, CA. 2011. ISBN 978-1-4302-3802-7
20. Lieberherr, Karl. “Law of Demeter: Principle of Least Knowledge” [Internet]. Available from: http://www.ccs.neu.edu/home/lieber/LoD.html
21. docs.Microsoft. “net-standard” [Internet] available from: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. accessed 20.8.2019
22. docs.Microsoft. “what’s new in net framework 4.7.1”. [Internet] available from: https://docs.microsoft.com/en-us/dotnet/framework/whats-new/#whats-new-in-net-framework-471. accessed 20.8.2019
23. Robot Framework, “RobotFramework”. [Internet] available from: https://robotframework.org. accessed 24.8.2019
24. International organization for standardization; International electrotechnical commission. ISO/IEC TR 19759:2015 – “Software engineering – guide to the software engineering body of work (SWEBOK)” [Internet]. 2015. accessed 24.8.2019. Available from: https://standards.iso.org/ittf/PubliclyAvailableStandards/c067604_ISO_IEC_TR_19759_2015.zip
25. docs.Microsoft. “Inheritance”. [Internet] available from: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance. accessed 24.8.2019
26. docs.Microsoft. “Interfaces”. [Internet] available from: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/index. accessed 24.8.2019

Robot script Import CAN

```
Import From CAN
    [Tags]  ImportCAN
    Take Control Of Running Application
    Use Search Function                     ${CAN}
    Sleep  2s
    Click Element                           automation_id=${CAN_button}
    Sleep  3s
    Click Element                           automation_id=${connec-
tion_switch_button}
    Sleep  2s
    Input Testing Serial Numbers            ${testing_serial_number}
${testing_mast_serial_number}
    Sleep  1s
    Click Element                           automation_id=ConfirmYesButton
    Sleep  1s
    Click Element                           automation_id=SettingsTabButton
    Sleep  2s
    Click App Element  NoButton
    Click Element                           automation_id=ImportFromTruck-
Button
    Sleep  1s
    Click Element By Title  Browse...  TextBlock
    Sleep  3s
    Click Element By Title  File name:  Edit
    enter text element identified by title  File name:  test  Edit
    Click Element By Title  Save  Button
    Click App Element                      automation_id=ContinueWizardBut-
ton
    Wait Element                           automation_id=FinishWizardButton
status=ready    timeout=180
    Click App Element  FinishWizardButton
    Sleep  1s
    Click Element                           automation_id=${connec-
tion_switch_button}
```

Robot script Import TCP/IP

```
Import From TCP/IP
    [Tags]  ImportTCP
    Take Control Of Running Application
    Use Search Function                 ${rocla_agv}
    Sleep  2s
    Click Element                   automation_id=TruckImage_agv
    Sleep  3s
    Click Element                   automation_id=${connection_switch_button}
    Sleep  2s
    Input Testing Serial Numbers        ${testing_serial_number
    Sleep  1s
    Click Element                   automation_id=SettingsTabButton
    Sleep  2s
    Click App Element                 NoButton
    Click Element                   automation_id=ImportFromTruckButton
    Sleep  1s
    Click Element By Title  Browse...  TextBlock
    Sleep  3s
    Click Element By Title  OK  Button
    Click App Element                 automation_id=ContinueWizardButton
    Wait Element                    automation_id=FinishWizardButton    status=ready
timeout=180
    Click App Element  FinishWizardButton
    Sleep  1s
    Click Element                   automation_id=${connection_switch_button}
```

Robot script Import Serial

```
Import From Serial
    [Tags]  ImportSerial
    Take Control Of Running Application
    Use Search Function                    ${serial}
    Sleep  2s
    Click Element                          automation_id=${serial_truck_im-
age}
    Sleep  3s
    Click Element                          automation_id=${connec-
tion_switch_button}
    Sleep  2s
    Input Testing Serial Numbers           ${testing_serial_number
    Sleep  1s
    Click Element                          automation_id=SettingsTabButton
    Sleep  2s
    Click App Element                      NoButton
    Click Element                          automation_id=ImportFromTruck-
Button
    Sleep  1s
    Click Element By Title                 Browse...  TextBlock
    Sleep  3s
    Click Element By Title                 File name:  Edit
    enter text element identified by title  File name:  test  Edit
    Click Element By Title                 Save  Button
    Click App Element                      automation_id=ContinueWizardBut-
ton
    Wait Element                           automation_id=FinishWizardButton
status=ready    timeout=180
    Click App Element                      FinishWizardButton
    Sleep  1s
    Click Element                          automation_id=${connec-
tion_switch_button}
```

TCP/IP import times in the sequence of running. Original version.

Units in milliseconds

2750.4983
3155.1385
2674.0605
3353.613
3029.778
3567.5684
3733.9408
3627.8166
4018.1598
3596.362
4039.8005
4609.4825
3848.4753
3949.4737
4885.4334
4827.6702
4134.0343
5082.1531
4888.8479
5556.789
4704.2608
5490.6135
4838.8423
6232.6905
6249.0354
6593.6638
6417.9456
6509.5018
5633.1518
5842.7071
5654.7237
5852.7585

5817.2242
5974.2614
5955.4406
6916.9947
6702.931
6368.9763
6885.5918
6407.2025
7019.4429
6833.1978
7176.7109
7228.2465
6082.1045
6038.063
5788.032
6486.8976
6746.0749
7033.0431
7476.6199
8013.0518
7853.3017
8042.8973
8156.6354
7961.7856
8639.7524
8881.503
8348.8621
7416.4756
7839.7454
7407.5422
7929.3552
7269.0216
9077.8723
9946.0304
10086.0574

10046.1119

10415.4896

7470.7043

9066.2509

10681.3176

11209.788

10872.5551

11917.1085

10631.235

8562.5121

10912.5594

11454.6149

9645.6051

11583.0427

9984.0085

11771.1454

10558.4597

12985.0333

11738.0328

13725.5633

12217.5701

11222.7461

11147.9324

13083.6954

12066.3205

11309.4282

11715.4886

14363.9449

13975.8242

12307.6366

11741.6234

TCP/IP import times in the sequence of running. Modularized version.

Units in milliseconds

2750.4983
3155.1385
2674.0605
3353.613
3029.778
3567.5684
3733.9408
3627.8166
4018.1598
3596.362
4039.8005
4609.4825
3848.4753
3949.4737
4885.4334
4827.6702
4134.0343
5082.1531
4888.8479
5556.789
4704.2608
5490.6135
4838.8423
6232.6905
6249.0354
6593.6638
6417.9456
6509.5018
5633.1518
5842.7071
5654.7237

5852.7585
5817.2242
5974.2614
5955.4406
6916.9947
6702.931
6368.9763
6885.5918
6407.2025
7019.4429
6833.1978
7176.7109
7228.2465
6082.1045
6038.063
5788.032
6486.8976
6746.0749
7033.0431
7476.6199
8013.0518
7853.3017
8042.8973
8156.6354
7961.7856
8639.7524
8881.503
8348.8621
7416.4756
7839.7454
7407.5422
7929.3552
7269.0216
9077.8723
9946.0304

10086.0574

10046.1119

10415.4896

7470.7043

9066.2509

10681.3176

11209.788

10872.5551

11917.1085

10631.235

8562.5121

10912.5594

11454.6149

9645.6051

11583.0427

9984.0085

11771.1454

10558.4597

12985.0333

11738.0328

13725.5633

12217.5701

11222.7461

11147.9324

13083.6954

12066.3205

11309.4282

11715.4886

14363.9449

13975.8242

12307.6366

Serial import times. Original version

Units in milliseconds

18561.5178
18316.9156
17801.9005
18830.0984
18209.0385
18143.3467
17984.0298
18402.993
18321.2722
18079.0526
18294.2417
18123.0697
18287.9607
18907.6944
18520.7402
18500.3012
18354.1733
18486.4604
18343.1583
18288.9527
18548.6726
17973.0477
18668.5119
20391.6715
20828.3996
19841.9971
18680.2154
18003.892
18295.0307
18275.8792
18619.4725
18362.5085

18677.4564

18180.4155

18145.9209

18542.5054

18448.2265

18326.6102

18113.5666

18546.275

18547.8904

Serial import times. Modularized version

Units in milliseconds

18535.688
18183.2182
18437.9587
18403.4041
19546.3039
18036.2892
18820.3135
18621.376
18471.396
18422.4753
18541.9255
17961.9543
18507.4491
18305.2236
17904.8369
18424.212
18140.2529
18700.7227
18375.0348
18477.6608
18425.4552
18314.5595
18762.3729
18358.7558
18252.182
18323.9215
18553.2616
18256.0607
18949.4144
18265.0241
20271.9118
21787.9167

21462.8846

18525.2838

18309.794

18748.3417

18852.3291

18425.8759

18893.4285

18524.9923

18978.3763

CAN import times. Original version.

Units in milliseconds

142090.5302

146257.0524

136548.3339

144084.9363

130824.6104

138843.9024

131141.4004

137650.4589

141423.3574

135276.9256

138581.2599

135643.312

138465.3641

146848.4372

134616.1628

143369.1988

141272.0542

145459.9365

131154.577

144685.7008

142883.9805

141524.1574

138543.5699

138822.4814

145249.2146

131297.5809

147007.6332

143776.6802

143296.2262

131017.6442

138555.2449

143116.4283

131500.4227

140628.5102

142846.3762

139623.6589

143447.4749

138701.4262

140719.6722

131560.7606

CAN import times. Modularized version.

Units in milliseconds

139223.7362
131198.0094
145293.0058
142664.8525
133290.3545
145259.8018
141222.2227
144385.5081
136052.5153
142947.5743
140868.5413
145438.8772
143457.2104
142322.0881
131163.3493
134490.8521
137488.6876
144562.5995
139124.4519
143768.3556
146485.7432
132421.5272
137580.175
144372.4354
139148.0698
146102.5003
145568.895
138757.6444
136313.7785
135930.6956
142754.5729
142127.4915

138779.1844

136618.3503

143630.3809

137664.335

130879.0063

133986.4377

144277.5425

145721.4962

Robot script for TCP/IP batch import

```
Import 40 Times TCP/IP
    [Documentation]  Imports currently selected AGV truck's parameters 40 
times. Starting position is expected to be the settings view
    [Tags]  40ImportsCPI2
    :FOR  ${INDEX}  IN RANGE  1  40
        \    Take Control Of Running Application
        \    Click Element                        automation_id=Import-
FromTruckButton
        \    Sleep  1s
        \    Click Element By Title  Browse...  TextBlock
        \    Sleep  3s
        \    Click Element By Title  OK  Button
        \    Click App Element                   automation_id=Continue-
WizardButton
        \    Wait Element                        automation_id=FinishWiz-
ardButton    status=ready    timeout=180
        \    Click App Element  FinishWizardButton
        \    Sleep  1s
```

Robot script CAN batch import

```
Import 40 Times CAN
    [Documentation]  Imports currently selected CAN or Serial truck's pa-
rameters 40 times. Starting position is expected to be the settings view
    [Tags]  40Imports
    :FOR  ${INDEX}  IN RANGE  1  40
    \    ${index_str}  Convert To String  ${INDEX}
    \    Take Control Of Running Application
    \    Click Element                              automation_id=Import-
FromTruckButton
    \    Sleep  1s
    \    Click Element By Title                     Browse...  TextBlock
    \    Sleep  3s
    \    Click Element By Title                     File name:  Edit
    \    enter text element identified by title  File name:  ${index_str}
Edit
    \    Click Element By Title                     Save  Button
    \    Click App Element                          automation_id=ContinueWiz-
ardButton
    \    Wait Element                               automation_id=FinishWiz-
ardButton    status=ready    timeout=360
    \    Click App Element                          FinishWizardButton
    \    Sleep  1s
```

Robot script Serial batch import

```
Import 40 Times Serial
    [Documentation]  Imports currently selected Serial truck's parameters 
40 times. Starting position is expected to be the settings view
    [Tags]  40ImportsSerial
    :FOR  ${INDEX}  IN RANGE  1  40
    \    ${index_str}  Convert To String  ${INDEX}
    \    Take Control Of Running Application
    \    Click Element                         automation_id=Import-
FromTruckButton
    \    Sleep  1s
    \    Click Element By Title                Browse...  TextBlock
    \    Sleep  3s
    \    Click Element By Title                File name:  Edit
    \    enter text element identified by title  File name:  ${index_str}  
Edit
    \    Click Element By Title                Save  Button
    \    Click App Element                     automation_id=ContinueWiz-
ardButton
    \    Sleep  1s
    \    Wait Element                          automation_id=ContinueWiz-
ardButton    status=ready    timeout=360
    \    Click App Element                     automation_id=ContinueWiz-
ardButton
    \    Click App Element                     FinishWizardButton
    \    Sleep  1s
```