Tabular data import in web-based Enterprise Resource Planning systems

Mats Jalas

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 20.5.2016

Thesis supervisor:

Prof. Jukka Manner

Thesis advisors:

M.Sc. (Tech.) Mikko Halttunen

M.Sc. (Tech.) Antti Tuomi

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Mats Jalas

Title: Tabular data import in web-based Enterprise Resource Planning systems

Date: 20.5.2016 Language: English Number of pages: 9+74

Department of Communications and Networking

Professorship: Data Networks

Supervisor: Prof. Jukka Manner

Advisors: M.Sc. (Tech.) Mikko Halttunen, M.Sc. (Tech.) Antti Tuomi

Every year new software systems are developed ranging from small programs to large enterprise systems. The requirements of these systems have a tendency to change over time, which requires changes to be made. Making changes to a system according to changed requirements is part of software maintenance. The costs of software maintenance can be between 40-70% of a system's costs during its life-cycle. Systems with high maintainability need less maintenance effort, which means lower maintenance costs.

This thesis studies different aspects of software maintainability, with the purpose of redesigning Sapphire Build's data import feature. Sapphire Build is an Enterprise Management Suite developed by Kova Solutions, Inc. A proof-of-concept prototype is implemented based on the design. The prototype is evaluated based on the design's requirements and how well it can handle future changes.

The evaluation results indicate that the maintainability of a system depends both on its design and how it is implemented. Design documents provide important guidelines for developers but might not cover all the necessary details needed to implement it as planned. The results suggest that following the SOLID principles can greatly improve maintainability when developing software using an object-oriented approach.

Keywords: data import, software maintainability, prepare for change, design patterns, SOLID principles

AALTO-UNIVERSITETET
HÖGSKOLAN FÖR ELEKTROTEKNIK

SAMMANDRAG AV
DIPLOMARBETET

Författare: Mats Jalas

Titel: Importering av data i tabell form i webbaserade affärssystem

Datum: 20.5.2016 Språk: Engelska Sidantal: 9+74

Department of Communications and Networking

Professur: Datanät

Övervakare: Prof. Jukka Manner

Handledare: Diplomingenjör Mikko Halttunen, Diplomingenjör Antti Tuomi

Nya mjukvaru system utvecklas årligen, varierande från små program till stora affärssystem. Dessa systems krav har en tendens att förändras över tiden, vilket innebär ett behov att göra förändringar i systemen. Mjukvaru systems kostnader under dess livstid kan till 40-70% bestå av underhålls kostnader. System med hög underhållbarhet kräver mindre ansträngning vid underhålls arbeten, vilket minskar underhålls kostnaderna.

Denna avhandling studerar mjukvaras olika underhållbarhets aspekter. Målet är att omforma Sapphire Builds data importerings egenskap. Sapphire Build är ett affärssystem utvecklad av Kova Solutions Inc. En prototyp utvecklas på basis av omformnings designen för att bevisa omformningens duglighet. Prototypen evaluaras på basis av design kraven. Dessutom evalueras prototypen på basis av hur bra den kan hantera framtida förändringar.

Resultaten indikerar att underhållbarheten av ett system beror både på dess design och hur den är implementerad. Design dokument förser utvecklaren med viktiga riktlinjer, men täcker nödvändigtvis inte alla detaljer som behövs för implementering. Resultaten föreslår att mjukvaras underhållbarhet kan förbättras markant genom att följa SOLID principerna, när mjukvara utvecklas på ett object orienterad sätt.

Nyckelord: data importering, mjukvarans underhållbarhet, förberedning för förändring, design mönster, SOLID principerna

Preface

Over the years software development has become one of my passions, because it always presents new exciting challenges. The last couple of years I have been striving to learn as many new aspects of software development as possible to increase the quality of the software I develop. Writing this thesis has been an exciting opportunity to learn about the different aspects of software maintainability. I wish to thank Jouko Väkiparta, CEO of Kova Finland Oy, for the opportunity to write this thesis and for the chance to be part of the Kova Finland team. I also want to thank my supervisor Professor Jukka Manner for his advice and guidance in writing this thesis.

I want to thank both of my instructors M.Sc. (Tech) Mikko Halttunen and M.Sc. (Tech) Antti Tuomi for all of the interesting discussions and valuable feedback. I am also grateful to Mikko for presenting the topic of this thesis and Antti for deepening my knowledge in software architecture, generic programming, and the SOLID principles. I want to thank the whole Kova Finland team for making the office a great learning environment and a fun place to work at.

I want to thank all my friends that have supported me over the years and have enabled me to gain new insights into software development and life in general. Finally, I want to thank my parents Ari and Susanne Jalas for supporting me during all these years and teaching me to think outside the box to find alternative solutions to problems.

Otaniemi, 20.5.2016

Mats Jalas

Contents

Abbreviations

SOLID	Five object-oriented principles
SRP	Single responsibility prinicple
OCP	Open-Closed principle principle
LSP	Liskov substitution principle
IS	Interface segregation principle
DI	Dependency inversion principle
DLL	Dynamic Link Library
JSON	JavaScript Object Notation
MI	Maintainability Index
HV	Halstead Volume
CC	Cyclomatic Complexity
LOC	Average number of lines of code per module
COM	Percentage of comment lines per module
REST	Representational State Transfer
API	Application Programming Interface

List of Tables

List of Figures

1 Introduction

Software is part of everyday life, either we know it or not. It is included in all sorts of devices and systems, ranging from small sensors and mobiles phones to cars and airplanes. Software is used to solve a lot of different business problems. Requirements have a tendency to change over time, which makes it important to be able to change software according to the new demands. Implementing changes in software are seen as software maintenance. The higher the maintainability of the software, the less maintenance effort is required.

The purpose of this thesis is to develop a solution that enhances the maintainability of Sapphire Build's data import functionality. Sapphire Build is developed by Kova Solutions Inc. The goals of this thesis are presented in the next section. All code examples presented in this thesis are written in C# since this is the chosen implementation language of Sapphire Build.

1.1 Goals of this thesis

The Sapphire Build data import feature has been in use for several years. The purpose of this thesis is to produce a solution to the current maintainability problems in the data import feature's implementation. The proposed solution should be able to answer two particular research questions, in order for the solution to provide benefits for Kova Solutions Inc.

The first research question covers the future needs to support other data management interface than Excel spreadsheet. The current implementation only enables its users to import data through Excel spreadsheets. How should a generic data import system be implementend to extend usage beyond Excel spreadsheets?

The second research question covers continuous changes in requirements. There is a continuous flow of change requests from users that need modifications to be made to the Excel data upload files, which are used to import data to the system. How can the data import functionality be implemented reliably so it can handle changing data requirements?

1.2 Methods and scope

The method used for finding answers to the research questions and to the maintainability problems consist of a literature research on different software maintainability viewpoints. The requirements for a potential solution are gathered by analyzing the requirements and the problems of the current implementation. A design for the solution is developed based on the found requirements. The design is validated by developing a proof-of-concept prototype, which is evaluated based on design pattern,

object-oriented principles and measurement metrics. The prototype is also evaluated based on the research questions and requirements.

The scope of this thesis is mostly limited to maintainability aspects of the proposed design. Performance aspects that have significant impacts on the solution will be taken into consideration and shortly discussed. There are many other aspects that should be taken into consideration when implementing a data import feature to a software system, e.g. security, data integrity and reliability aspects. However, these aspects are outside the scope of this thesis.

1.3 Results

The evaluation of the prototype suggests that the proposed data import design will provide the means to for developing an extendable and maintainable data import functionality to Sapphire Build. The design documents provide developers with the guidelines needed to develop the data import functionality. However, to support reliable and easy maintenance, the results also clearly indicate that the maintainability of the data import functionality heavily depends on how it is implemented. The SOLID principles and the knowledge documented in design patterns provide the means to significantly increase the maintainability of the prototype implementation, which means that an implementation with low quality can greatly reduce the maintainability even if the design is good.

The maintainability measurements objectively shows clear maintainability improvements in the prototype compared to the old implementation. Unfortunately, the lack of cohesion measurements and small width of the coupling measurement does not provide any indications of reusability improvements. However, further measurements are needed to make any accurate conclusions regarding the reusability aspect.

Even though the focus of this thesis is not on automated testing, the results do show that automated tests provide important feedback about potential future maintainability issues before they become real issues. The results of this thesis also gives indications that the level of design and implementation knowledge of a developer or a development team correlates with future maintenance costs and issues.

1.4 Structure of the Thesis

Chapter 2 gives and introduction to the home building industry in the US and the Sapphire Build software suite developed by Kova Finland Oy. This chapter also includes an overview of how the data import is currently implemented in Sapphire Build, which can be found in Section 2.3. This section also provides background information and the base knowledge for the library requirements discussed in Section 4.2.

Chapter 3 is a literature review on current knowledge about software maintenance and software maintainability. The chapter provides the base knowledge for designing maintainable software, by discussing topics like how to prepare for future changes, design patterns and the five principles of object-oriented design. The knowledge gain from the literature review is used when developing the proposed design in Chapter 4 and when evaluating the design based on the implemented proof-of-concept prototype in Chapter 5

Chapter 4 begins with a discussion on why there is a need for a redesign of the data import functionality. The chapter also presents the requirements, the problem domain and the proposed redesign of the data import feature.

Chapter 5 discusses the evaluation of the proposed design based on an implementation of the design. The prototype is evaluated based on the knowledge gain in Chapter 3, the research questions presented in Section 1.1 and the requirements specified in Section 4.2.

Chapter 6 discusses the results and conclusions made from the design evaluation in Chapter 5 and the literature review in Chapter 3.

2 Home building software

The purpose of this chapter is to provide background information about the home building industry in the USA and the Sapphire Build software suite. First, a short introduction about how the home building industry in the US looks like. This is given in Section 2.1. Then the Sapphire Build software suite is shortly presented in Section 2.2. The purpose of the Sapphire Build software suite introduction is to give a general idea of what services the software suite can provide its users. Lastly, Sapphire Builds data import feature is presented in Section 2.3. The purpose of the data import section is to shortly describe the current implementation and the problems that the implementation is causing, focusing especially on its maintainability issues.

2.1 Home building industry in the US

There are thousands of home building companies in the US, ranging from small businesses that build only a couple of homes per year up to publicly traded companies with multi-billion revenues. The focus group of buyers varies between the top homebuilders. Homebuilders build homes for either a single type of buyers or a combination of different types. In the single buyer type segment, some homebuilders focus on providing luxury homes in the affluent market. Others instead focus on buyers who rely on securing affordable financing, which is referred to as the entry-level category. Most homebuilders focus on neither of these two categories. Instead, they focus on "first-time" buyers. The definition of a "first-time" buyer can be found in [1]. The average price of a home in the entry-level category is modestly lower than a home in the first-time category. [2]

2.1.1 Homebuilder types

The US home building industry includes three types of builders: production builders, building manufacturers, and custom builders. When put together the production builders build the largest number of units per year, even though the majority of the industry are custom builders. [3]

Production builders
Production builders build homes on land they own. There are two types of home categories that production builders build: single-family and multi-family homes. Some production builders build homes of both categories; others focus just on one category. Production builders usually buy large areas of land. Then they take a piece of the land, on which they then plan and build a community. Only a limited number of models are offered in a community which allows the builder to benefit from economies of scale. [3][4]

Building manufacturers
Houses built by building manufacturers are built in factories [3][5]. Among other things, this protects the materials from exposure to vandalism and weather [5][6]. According to Hyväri, there are two types of building manufacturers: mobile and modular home builders [3]. Mobile homes are homes on wheels that are placed on non-permanent foundations [7]. Modular homes are built as almost complete modules in factories and are put together at build-site on a conventional basement or crawl space foundations [7].

Custom builders
Custom builders provide home buyers the opportunity to buy a home customized to their needs. Production builders and building manufacturers usually provide different options for their customers, but do not provide fully customized solutions as custom builders do. The negative side of being able to customize can be seen on the price tag since custom builders e.g. can not buy materials in bulk and each home need to be designed separately. Since each home differs, custom builders build fewer homes per year compared to production builders and building manufacturers. [3]

2.1.2 Community Life Cycle

The community life cycle is the process starting from the builder buying a piece of land to selling the last home built on the land. The community life cycle begins with a home builder buying a piece of land and creating a layout for a community. The community is divided into lots, which each will accommodate one home. After this, the builder applies for building permits and hires contractors for building the infrastructure of the community when the they have been accepted. When the infrastructure is done, then between two to five model homes are built on chosen lots. One of the model homes is used as a sales office for the community. The available lots are sold to the customers, who select the lot and the model with options that will be built on said lot. Finally, when all lots are sold, then the model homes are sold. [3]

This process includes a lot of data which is processed by the Sapphire Build software suite. The next section introduces the Sapphire Build software suite and describes some details of the software suite that will affect the data import library design and prototype implementation.

2.2 Sapphire Build

The purpose of this section is to give an overview of the Sapphire Build software suite developed by Kova Solutions Inc and its technologies.

2.2.1 Overview

The Sapphire Build software suites real name is Sapphire Build Enterprise Management Suite, but in this thesis, it will be referred to as Sapphire Build. Sapphire Build is "an Internet-based, integrated, collaborative management and information distribution system for building companies" [8]. Sapphire Build provides six different modules:

- Product Development
- Sales & Marketing
- Land Development
- Production
- Purchasing
- Customer Service

Each of the six modules is integrated together, which makes company-wide data management much easier. Since Sapphire Build processes a lot of data, requires it to be able to import large data amounts. Section 2.3 discusses more on data import. [8]

Business processes can be implemented in Sapphire Build because of its workflow-based approach. The implemented business processes provide "a structured approach to management of product information, product changes, vendor management, scheduling, finances and customer information" [8]. Sapphire build also enables user customization of all workflows and provide vendors, builder associates, and customers information through the web interface. The role-based security system ensures that data is only accessible by users with sufficient permissions. [8]

2.2.2 Technologies

Sapphire Build is built on top of Microsoft's technologies and platforms for web-based applications. The application is implemented based on the C#.NET framework and Microsoft SQL Servers holds all application data. The backend runs on Microsoft Windows Server.

2.2.3 CodeGen

GodeGen is an inhouse code generation tool that used by developers to make sure the architecture of the whole system does not differ between business objects. GodeGen generates business object code ranging from C# and ASP.NET WebForms pages to SQL scripts. Object models are defined in custom formated text files, from which CodeGen then generates the code. [9]

2.3 Data import

This section begins by giving a short introduction to the data management interface used by users to manage a large amount of their data in Section 2.3.1. Then an overview of the data import feature implementation is presented in Section 2.3.2. Finally, Section 2.3.3 discusses problems with the current implementation, one of which is maintainability.

2.3.1 Data management interface

Sapphire build provides data import functionality through Excel files. Builders are familiar with using Microsoft Excel to manage their data, which has been the main reason for using Excel spreadsheets as a data management interface. Users can modify their data in an Excel spreadsheet. Users data is then imported into Sapphire Build's database by uploading an Excel file to the system. To be able to represent business objects in an Excel spreadsheet has lead to the usage of a particular spreadsheet layout, which divides a spreadsheet into 'sections'. Each 'section' holds business objects belonging to a particular type. Figure 1 displays an example upload file showing two sections: community and section data.

Community Data

Key	Sort	ID	Name	Description	Parent B/U	City	Zip Code	County	State	Status
Community	1	C1	CommunityOne	CommunityOne	CommunityOne	Example City	12345	Example County	ES	Open

Section Data

Key	Sort	ID	Name	Description	Community	Status
Section	1	C1	CommunityOne, Secti	CommunityOne, Section 1	C1	Open

Figure 1: Part of an example community data upload Excel file.

Communities are divided into sections. A 'section' in a upload spreadsheet has one header row and one or many object rows. An object row holds data belonging to one business object. There are a few special cases where data belonging to a business object is divided onto more than one row. The reason behind this decision was to make the upload spreadsheets more user-friendly. However, all object rows must at least include a 'Key' and an 'ID' column. The 'Key' column is used to identify the business object type in the data parsing code. The 'ID' column represents the ID belonging to the particular business object. If the business object already exists in the database, then the value in the 'ID' column is the key to finding that particular business object.

The purpose of the example Excel file in Figure 1 is only to give a peek of the structure of an Excel data upload file. Real-life data upload files uploaded to Sapphire Build

by users can consist of thousands of rows of data. The next section gives an overview of how the data import feature has been implemented.

2.3.2 Current implementation

The current data import implementation includes six different logical parts:

- Web UI
- UploadData folder
- DataBuilders
- ExcelDataBuilder
- ExcelImports database table
- BOPersistor

Excel upload files are uploaded to Sapphire Build through the Web UI and stored in an UploadData folder. A user can then start the actual data import process through the UI, which calls a DataBuilder object. There exists one DataBuilder object per each type of Excel upload file. A DataBuilder then loads the content of the Excel file into a temporary database table called ExcelImports. When the data has been stored in ExcelImports table, the user can start the data building process through the UI. The same DataBuilder delegates the actual data building work to an ExcelDataBuilder, which then creates a business object from an Excel row. The ExcelDataBuilder then calls a BOPersistor to do one out of three options: update or delete the existing object in the database, or store this new object to the database. The term BOPersistor in this context refers to the parts of the system that handles data storing. Finally, the BOPersistor returns a so-called result message to through the call chain back to the user. The data import process data flow can be seen in Figure 2.

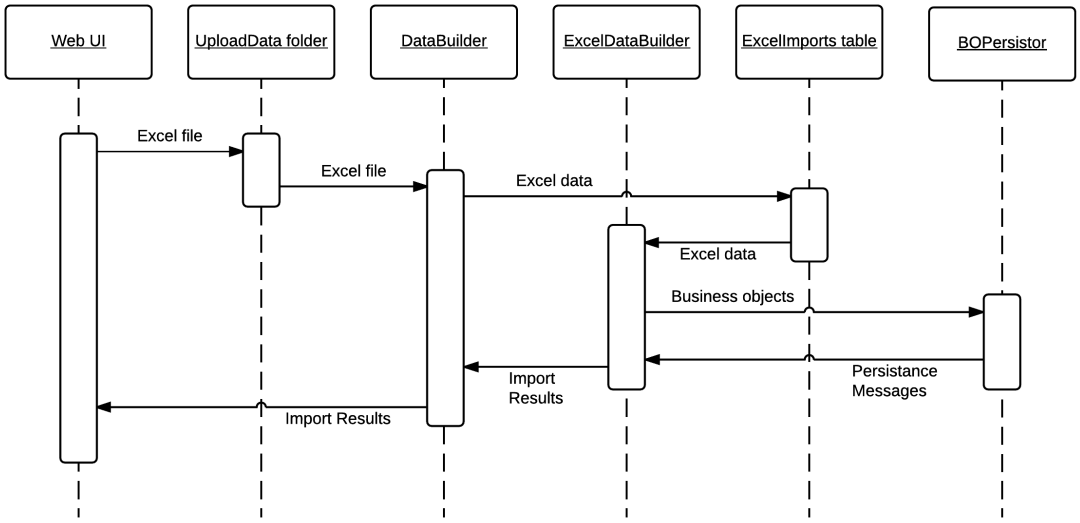

Figure 2: A sequence diagram of the current data upload process.

2.3.3 Current maintainability problems

The current solution has been used for several years, but over time, the logical parts seem to end up unsynched and not working as expected. The main reason behind this is because components in the system that should change for the same reason are separated into different parts of the system. The logical parts of the data import feature can be seen in Figure 3. The figure also shows the dependencies between the logical parts. The maintenance problem will be described in an example of a change request to the Community data upload Excel spreadsheet shown in Figure 1. Let us say a user wants to add a new column to Section object's rows. Then there are at least three different logical parts in the system that may need to change. A change would definitely have to be made in the Excel template related to community data upload. Most likely changes would also have to be made in the Excel parsing code since the template has changed. If the added column is a new property for Section objects, then a change would also have to be made in the data persistence code and the database. Three different locations do not sound like a massive problem. However, Figure 3 only shows high-level elements representing both the parsing code and the data persistence code. These elements consist of hundreds of lines of code, which means that there might be several more places that should change. All of this adds up to a potential maintenance nightmare. Also, there is a good chance of forgetting to change one of the parts. This problem has occurred with the current solution and is one of the reasons for this thesis.

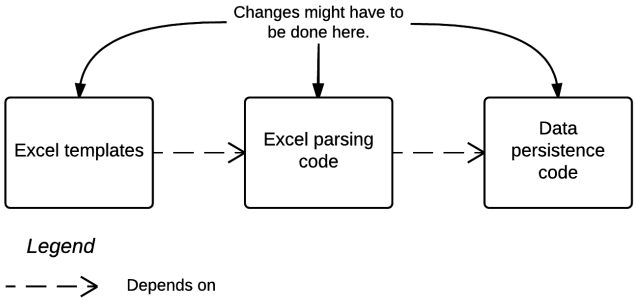

Figure 3: Fragmentation of excel upload logic.

Another maintenance problem with the current solution is a lot of code duplication. Since a lot of business object related code is generated with CodeGen, it has also lead to a lot of code duplication. Code generation in itself does not produce code duplication. Instead parts of the CodeGen design causes the code duplication. CodeGen in itself is out of the scope of this thesis, but the design issues in CodeGen goes back to the early days of the .NET framework and its limitations.

There is also a performance problem with the current implementation. The implementation loads a whole Excel spreadsheet into memory twice. Once when reading the file into the temporary database table, and a second time when the data is read from the temporary table to be parsed. First of all, this causes unnecessary IO calls. Secondly, if the Excel spreadsheet happens to be huge, then it will require a lot of memory by the system. Lastly, the current implementation does not provide any means of scaling the data processing in order to increase performance.

The solution for these problems is presented in Chapter 4.

2.4 Summary

This chapter has provided an introduction on the home building industry in the US and the Sapphire Build software suite. This chapter has also shortly presented Sapphire Builds data import feature and its current implementation problems, mainly focusing on maintainability. The next chapter will describe different aspects of software maintainability, which gives the base knowledge required for the design and implementation face. The next chapter also discusses techniques on how to develop maintainable software and how to measure software maintainability.

3 Software Maintainability

Software systems are practically never error-free when released, which requires maintenance work to be done to enable continuous usage of the software. Thomas and Hunt agree with this when they state that no one in the history of computing has been able to write perfect software [10]. In order to deal with errors, maintenance is required. Also, systems never stay the same during their life cycles [11]. Jacobson et al. argue that "this must be borne in mind when developing systems expected to last longer than the first version (i.e. practically all systems)" [11]. User's needs also change over time, which leads to changes in requirements. This can either be a need for new functionalities or need to improve an existing functionality. All the extra work done after a release refers to maintenance. This chapter starts with shortly introducing two maintenance perspectives: business perspective and developer perspective. These two perspectives are discussed in Section 3.1, and should give reasons for why software maintainability is important. Section 3.2 introduce different change types, maintenance categories, and tools that help in preparing for future changes. Section 3.3 discuss ideas on how to design for maintainability. Sections 3.5 and 3.4 introduce techniques that are useful for designing and implementing maintainable software. To be able to know how maintainable a software is, it should be measured. Maintainability and reusability metrics are discussed in Section 5.1 in the evaluation chapter.

3.1 Overview

Maintenance can be analyzed from many different perspectives. However, this thesis focuses on two perspectives that felt the most important regarding the purpose of this thesis. First, the business perspective on maintenance will be presented, since most software is developed to solve a business problem. Second, the developer perspective is discussed, because they have the greatest impact on how maintainable a system will be when it is released. However, it is important to define maintainability before discussing the two perspectives. Tortorella defines maintainability as "the ability of a system to be repaired and restored" [12]. Grubb and Takang define maintainability from a maintenance perspective: "the ease with which maintenance can be carried out" [13]. The term maintenance is defined as "the act of keeping an entity in an existing state of repair, efficiency, or validity: to preserve from failure or decline". Software maintainability is defined in the ISO/IEC 14764 standard as "the capability of the software product to be modified" [14]. The ISO/IEC 25010:2011 standard defines maintainability as "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [15]. All four definitions say that maintainability refers to how modifiable a software is. Low maintainability means that software is difficult to change while high maintainability means that software is easy to change. Sections 3.1.1 and 3.1.2 discusses importance of maintainability from business and developer perspectives.

3.1.1 Business perspective

Software is developed to solve problems [16]. Business software is developed to address a business problem. Software should create value for its users, in other words, a software system should be useful. If users find a software unusable, they will not use it [13]. A company can only profit from software that its staff or customers use. To keep a software system useful for its user's errors should be fixed. Also, new features should be added to increase the usefulness of the software. As defined earlier, these activities are software maintenance activities.

Swanson and Dans argue that systems in some organizations outlive the time maintenance personnel works at the organization [17]. McColm Smith states, in his book Elemental Design Patterns, that "software has a peculiar trait of living long past its expected lifetime" [18]. Some software systems were developed as a solution to a particular domain problem, but has been found so useful that it has been used in entirely different areas than originally intended. WordPress is an excellent example of this. WordPress was originally developed as a blogging system, but has been used as a content management system and also for many other purposes [19]. Software systems found useful in other problem domain, can also increase the software life-cycle. The increased life-cycle combined with an increased user base can add to maintenance costs, due to a potentially increased amount of new feature requests. Also, the level of complexity in software can also affect maintenance costs. Rajiv et al. found in their study that increasing complexity of a system also increased maintenance costs [20].

Anda argues that even though the cost of maintaining software varies, it can often be expensive [21]. Maintenance costs can be between 40-70% of the software's costs during its life-cycle [13]. Král and Žemlička agrees that maintenance costs are higher than development costs, approximately two or three times greater [22]. For small software systems with short life spans, this might not be a huge cost. On the other hand, large software systems can have huge costs and with maintenance having approximately a 40-70% cut of all costs, should already by itself motivate to develop maintainable systems. Still a lot of unmaintainable or hard-to-maintain systems are developed [23]. All the additional costs should give an incentive that keeping a system maintainable is important from a business perspective. Maintainability should be kept in mind already in design and development stages, and not wait until the software is released.

3.1.2 Developer perspective

Maintenance is not something many developers like to perform [24]. Developers would much rather implement something new. Kurt Vonnegut, Jr. once said "Another flaw in the human character is that everybody wants to build and nobody wants to do maintenance", which would explain why developers prefer implementing something new than to work on maintenance tasks [14]. Higgins strengthened this with the

following statement [13]:

"...programmers... tend to think of program development as a form of puzzle solving, and it is reassuring to their ego when they manage to successfully complete a difficult section of code. Software maintenance, on the other hand, entails very little new creation and is therefore categorized as dull, unexciting detective work."

Another reason why developers prefer implementing something new is that writing code is easier than reading code [25]. Adding new features, which is considered as maintenance, usually means writing at least partly new code. However, implementing new features can be very difficult or even impossible in legacy systems with much technical debt [26]. Also, According to Grubb and Takang, maintenance engineers have previously not had as high a status as developers [13]. However, they also stated that maintainers perceived value has increased and that their status is aligning with developers [13]. Because most developers do not like maintenance work, it should motivate developers to develop maintainable software in the first place. This way, maintenance task would be less dull and tedious.

Both the cost aspect of maintenance and the amount overhead work maintenance engineers have to complete, should give the incentive to develop software prepared for future changes in the first place. By preparing for future changes, maintenance costs can be reduced, and maintenance work would be a less dull and tedious task for both developers and maintenance personnel. Section 3.2 will discuss ways of preparing for the future requirements and maintenance tasks. The concept of separation-of-concern and the power of introducing higher lever abstraction, to help design for maintainability, is discussed in Section 3.3. Sections 3.4 and 3.5 describes techniques that can be used to improve the design and maintainability of a software system.

3.2 Prepare for change

Changes have to be made to software systems in order to keep them useful for users. Even more than half of changes made to a system is done after first release [27]. By preparing for future changes will decrease both the effort needed to make the changes and the cost of making changes [27]. Feathers state that there are at least four primary reasons to make changes to software: adding a feature, fixing a bug, improving design, and optimizing resource usage [28]. In order to better understand why and how to make changes, this section starts with identifying different types of change and how they relate to maintenance activities. Section 3.2.2 discuss the importance of feedback when developing software, which helps developers to know when changes should be made. Section 3.2.3 shortly introduce different sources of feedback that help developers develop better software. Section 3.2.4 discuss techniques which help in keeping the code base maintainable. These topics combined should give ideas on how to prepare for future requirement changes and where to get more information about each topic.

3.2.1 Change types

A requirement for being able to prepare for changes is to identify the different types of changes that are made in a software system. Grubb and Takang present, in their book Software Maintenance, four different types of changes that are to software systems [13]: adaptive, corrective, perfective, and preventive.

Adaptive change
Adaptive changes are made when there is a need for a system to adapt to changes in the environment it operates in. Grub and Takang argue that these environment changes may come from several different sources, e.g. changes in business rules, enabling usage of a system on a new platform, or even from government policy changes [13].

Corrective change
A corrective change refers to fixing a defect found in software. Grub and Takang state that a defect in software originates from one of three different errors: design, logical or code errors [13].

Perfective change
Perfective changes are made to a system with the intention of enhancing the system. Perfective changes include adding new functionality, improving existing functionalities, and removing functionalities that have become useless or have not been accepted by users [13].

Preventive change
Grubb and Takang state that to prevent future malfunctions and to improve maintainability preventive changes have to be made [13]. A preventive change is initiated when e.g. there is a need to improve the structure of the code, optimize code usage and update documentation [13]. By regularly making preventive changes to software systems, the changes are usually smaller and there should be less of a risk of having to rewrite large parts of a system.

Tripathy and Naik, instead of discussing types of change discuss changes from a maintenance viewpoint. Tripathy and Naik describe three different classification viewpoints of maintenance activities: intention-based, activity-based, and evidence-based [14]. In this thesis, the focus will be on the intention-based classification viewpoint since it has some correlations with the types of change presented earlier in this section. Tripathy and Naik divide the intention-based viewpoint into four different categories: adaptive, corrective, perfective and preventive [14]. The names of these four categories have some resemblance to change types. Both adaptive and corrective maintenance activities described the same changes as the adaptive and corrective changes.

However, there are some differences with perfective change and maintenance and preventive change and maintenance. Tripathy and Naik say that improving maintainability is a perfective maintenance activity [14]. Grubb and Takang classified maintainability improvements as a preventive change [13], which when put into a

maintenance activity context would imply that preventive changes can be made during perfective maintenance activities. On the other hand, restructuring code to improve coding style is, by Tripathy and Naik, seen as a preventive maintenance activity [14]. This would imply that preventive changes are made both in perfective and preventive maintenance activities. More information about change types and maintenance activities can be found from the 'Software Evolution and Maintenance' book by Tripathy and Naik, and from the 'Software Maintenance' book by Grubb and Takang. Both books can refer to more resources regarding the topic discussed.

3.2.2 Importance of feedback

Creating something that no one want to use can be seen as waste [29]. Creating something from an idea that requires a lot of resources can be risky. Sometimes the result might be a success, but there is always a chance of costly failures. Gathering feedback helps in making informed choices, and improves the odds for creating a useful product [30]. Over the history of software development many software projects have failed, and many have resulted in huge additional costs [31]. There can be several reasons for why a project fails or ends up being many times expensive than planned. Implementing a project without keeping high code quality, will most likely at some point cause problems [32]. By continuously getting feedback about the quality of the code and how it performs after changes have been made, can prevent future problems. Beck argue that when requirements continuously change, it results in an increased need for feedback [33].

The form feedback comes in varies [33]. Examples of feedback forms given by Beck [33]: "

- Opinions about an idea
- How the code looks when you implement the idea
- Whether the tests were easy to write
- Whether the tests run
- How the idea works once it has been deployed

"

Extreme Programming (XP) practices have a tendency to produce a lot of feedback [33]. Beck argue that by getting feedback as soon as possible the sooner teams can adapt and make changes [33]. However, the examples shown are just to give some practical examples of forms feedback comes in, but the more important thing is how the feedback is used.

One way feedback is used in systems is by adding feedback control that detects changes in system output [34]. By adding feedback control that detects changes in the output of a system, enables a system to respond to detected changes [34]. Developers can also add feedback controls into the development process in order to

gather important feedback about the software system under development or under maintenance. This is why the following Section 3.2.3 shortly presents different feedback sources especially useful for developers regarding code quality.

3.2.3 Code Quality feedback sources

Feeback sources for making an informed decision varies depending on what decisions should be made. The following feedback sources presented in this thesis are useful for making an informed decision about how to keep system maintainability high, which is most useful for developers and maintainers. The topics below, however, are not covered in dept but focus on what feedback each source can give developers. The are many sources for more information about each topic, and each reference is a good starting point for that specific topic.

Automated testing

Testing provides feedback about how a system functions when given different inputs. Tests can be done either manually or they can be automated. Automated tests provide several benefits over manual testing, including speed, efficiency, accuracy, and precision [35]. Manual testing, in the other hand, is very useful for exploratory testing [36]. Automated tests might require more effort when tests have to be formalized. Baxter argue that automation depends on formalization [37]. However, Baxter also belives that "productivity advances in software construction and maintenance depend on automation" [37].

According to Pryce and Freeman, there are three different levels of automated tests [38]:

- Acceptance tests
- Integration tests
- Unit tests

Each level give valuable feedback about the system. Acceptance tests should specify functional requirements [36]. When acceptance tests pass it gives feedback to developers that the required functionality has been implemented. Acceptance tests should also give feedback about how the system works. Integration tests give feedback about how a code developed by a team works with 3rd party code that can not be modified by the team. Unit tests give feedback about how objects in a system work. Unit tests also give feedback about how easy an object is to use. [38]

As a side note, Martin argues that passing acceptance tests alone should not be used as a definition of done [36]. When all levels of tests pass and stakeholders and QA have accepted the developed functionalities, then the criteria of done is met [36].

Code coverage
There are some potential drawbacks if solely relying on writing automated tests as a measure of working code. How does a developer know if a test suite exercises all the code that a change may affect? How is the effectiveness of a test suite measured? Passing tests may give developers that are making code changes false confidence about the success of the changes [39]. This can be solved by gathering code coverage data when running automated tests. By writing tests without having information about the code coverage of the tests written, it could be seen as developers writing tests blindly [39]. Wloka et al. found from their research that the effectiveness of automated tests is both sensible and practical to measure with code coverage [39]. Because of this, test coverage information should be seen as an important source for feedback information relating to the effectiveness of automated test suites. Also, as a side note, research studies have been made to discover the correlation between code coverage and software reliability [40][41][42]. These research studies found that higher code coverage lead to higher software reliability.

Continuous Integration
Continuous integration refers to continuously integrating code developed by separate developers inside a team. Feedback about how each developers code integrates with each other is important for a team in order to know if the system that is being built is going to have any changes of working. When continuous integration is implemented as an automated process and runs several times a day, it will give developers the opportunity to get rapid feedback about how new code integrates with existing code. [43]

Pair programming
Beck defines pair programming as two people working together side-by-side in front of one machine. There are many benefits of pair programming regarding the amount and how fast feedback can be received. Pair programming gives the opportunity to clarify ideas when explaining them to the pair. Getting answers to questions are potentially faster since questions can always be directed to the pair. Pair programming also leads to a practice of always having code being analyzed by at least two developers. [33]

Code review
The term code review refers to reviewing source code. Peer code review refers to reviewing code written by others [44]. By including peer code reviews in daily work, pieces of code is always read by at least one other developer than the author. This can provide very useful feedback for developers. Milanesio gives a good example of how e.g. a Junior developer adding a comment to a piece of code stating that he could not understand it, which would give important feedback that the code probably should be simplified [45]. By simplifying the code so that every one on the team can understand it, also improves the maintainability of the code since everyone on

the team can maintain it [45]. Different perspectives from other developers can lead to better informed design decisions [45]. Peer code reviews also add the benefit of knowledge transfer in a developer team [44].

3.2.4 Continuously improve the code base

Another way to prepare for future changes is to continuously improve the code base. Pryce and Freeman argue that the time developers spend on writing code is far less than the time spent on reading code [38]. Because simpler code is easier to read than complex code, effort should be made to keep the code as simple as possible [38]. Also, bad code slows down development, since developers most likely have to put more effort into understanding what some piece of code does [26]. Bad code also reduces software's internal quality [46]. Developers should always strive to continuously improve the code base they are working on [26][47]. By refactoring pieces of code in order to make it easier to add a new feature also improves the quality of the code structure. Fowler defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [48]. More information about refactoring can found from the book 'Refactoring: Improving the Design of Existing Code' [48]. Understanding object oriented principles and patterns also makes it easier to write maintainable code [49], which is discussed further in Sections 3.4 and 3.5. However, Martin argues that principles and patterns are not enough and that it also takes attention, discipline and "a passion for creating beauty" [49].

The next section discusses higher level techniques on how to design for maintainability, which makes it easier to keep the code base clean.

3.3 Design for maintainability

The purpose of this section is to give a short introduction on how to design maintainable software. Pryce and Freeman suggests the usage of two principal heuristics to help design maintainable software: separation of concerns and higher levels of abstractions.

One of the fundamental principles of software engineering is the principle of separation-of-concerns [50][51]. The principle was originally introduced, by Djikstra and Parnas [50], to solve the problem of controlling the complexity of growing programs. By separating concerns and encapsulating them into modules, the complexity of the system is divided into more manageable and comprehensible parts [51]. Pryce and Freeman argue that it is much easier to make changes in a system when all code related to a behavior is located in the same place [38].

According to Pryce and Freeman, "the only way for humans to deal with complexity is to avoid it, by working at higher levels of abstraction" [38]. By creating abstractions for a module or group of modules, makes it easier deal with. Encapsulating lower level

details into abstractions makes it easier to focus on different levels of an application, e.g. when defining different business rules the way different objects are persisted might not be very importing to be aware of at that time. Pryce and Freeman clarifies this by a simple example: “most people order food from a menu in terms of dishes, rather than detail the recipies used to create them” [38].

Over the years, separating concerns into modules has led to the idea of object-orientation [50]. Object-oriented languages help developers deal with complexities through encapsulation and information hiding. Software evolution is less costly when concerns are encapsulated [51]. When the changes are localized to modules the impact is also smaller [51]. However, D. Kung et al. stated in 1994 [52] that the object-oriented paradigm introduced new problems for software maintenance. To avoid some of the problems, the next two sections will discuss techniques to solve object-oriented design problems: Design patterns and SOLID principles. Section 3.4 introduces the concept of design patterns and describes patterns that will be useful for designing and implementing a prototype for the data import library. Section 3.5 introduces five object-oriented principles, when used properly help in the effort of developing maintainable software.

3.4 Design patterns

Solving complex problems is hard. Solutions that try to solve large problem domains easily become complex themselves. Designing software solutions, especially object-oriented approaches, are no different in this aspect [53]. Previous sections in this chapter have already introduced the topic of maintainability and some techniques on how to prepare for future needs. This section begins by describing the concept of design pattern, and why design patterns are useful for creating maintainable software. Then the desing pattern template created by Gamma et al. [53] is presented in Section 3.4.2. The design pattern template is discussed to provide base knowledge needed to be able to read documented design pattern. Any particular design patterns are not discussed in this section, because so many design patterns have been documented over the years since Gamma et al started discussing about design patterns in the software community.

3.4.1 Overview

Design patterns were introduced to the software community by four authors who wrote the book Design Patterns: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. The four authors have since been commonly known as the Gang of Four (GoF) [18][54]. The idea of design patterns came from the work done by the civil engineer and architect Christopher Alexander’s in the 1960s. Alexander worked on finding similarities in buildings and towns, which he then called patterns [54]. Gamma et al. argue that Alexanders findings are also true in object-oriented design patterns [53]. [18][53][54]

Design patterns are abstract solutions to reoccurring design problems [53]. Gamma et al. argue that reusing successful designs and architectures is made easier by design patterns [53]. McColm Smith even goes as far stating that "design patterns are one of the most successful advances in software engineering" [18]. Design patterns reduce the need for developers to re-invent the wheel each time when solving a problem.

GoF presented 23 patterns they had identified through their research [54]. Gamma et al. used two criteria to divide the patterns they presented: purpose and scope [53]. The purpose of a design pattern can be either creational, structural or behavioral. Creational patterns are patterns that describe how object creation problems can be solved. Structural patterns describe how to deal with class compositions. The third, behavioral patterns, deals with classes or objects interactions, and how responsibilities should be distributed. The scope criteria define if a design pattern should be primarily applied to classes or to objects. [53]

Gamma et al. argue that there are four essential elements in a design pattern: pattern name, problem, solution, and consequences [53]. The pattern name is the pattern's identifier, which makes it easier to communicate with others about a pattern. The problem describes situations where the pattern can be applied and the solution describes the building blocks needed for implementing the solution for the problem. A pattern's consequences describe what the trade-offs and results are. There is rarely only one solution to a problem, which is why a pattern's consequences can be very helpful when choosing between different alternatives [53].

McColm Smith stresses that design patterns are design concepts, which do not depend on a specific programming language. Certain design concepts are easier to implement in some languages and harder in others. Some programming languages even have design patterns built in as features of the language. McColm Smith states that unfortunately many developers believe that a design pattern expressed in two different languages differ depending on the language. McColm Smith argues that this is not true and that only the implementation differs not the basic underlying concept. [18]

Over the years since the release of a lot of design patterns have been documented [18]. This thesis only focuses on the design patterns found most useful for the prototype design and implementation. However, more information about design patterns can be found from several books [55] and websites [56][57].

All design patterns presented in the Design Pattern book follow a specific template created by Gamma et al. This template is presented in the following subsection. The rest of this section continues with presenting the most relevant design patterns for the design and implementation of the prototype for the data import library.

3.4.2 Design pattern template

Gamma et al. argue that describing design patterns with graphical notations is not sufficient enough since graphical notations only describe relationships between classes and objects [53]. They state that it is important that a design pattern also describes decisions, alternatives, and trade-offs that led to that specific design [53].

By following the template to document design experience, enables sharing of knowledge with other developers. Writing design patterns following the template structure makes them easier to learn, compare and use [53]. This makes design patterns relevant to software maintainability, since implemented solutions, decision and trade-offs can be documented, which can be useful for maintainers of the software. The template consists of 13 sections defined by Gamma et al.[53]:

Pattern name and classification
: The name of the pattern, which should be chosen wisely. If the pattern is accepted by the team or software community, it will become part the vocabulary used when refering to the design. [53]

Intent
: According to Gamma et al. the intent should answer the following questions: "What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address" [53].

Also known as
: Also known as section lists other names by which the pattern is known as [53].

Motivation
: Gamma et al. state that the motivation section should describe "a scenario that illustrates a design and how the class and object structures in the pattern solve the problem" [53].

Applicability
: Applicability section should give answers to the following questions: What kind of situations could the design pattern be applied to and how are these situations recognized? What type of poor design examples can the pattern solve? [53]

Structure
: Gamma et al. state that the structure of the pattern should be given as a graphical representation of included classes [53].

Participants
: Participants section of a design pattern describes the classes and/or objects that participates in the design pattern [53].

Collaboration
: Collaboration section describes how the responsibilities are carried out by the participants in collaboration [53].

Consequences
Trade-offs of the design pattern are described in the consequnces section. It should also describe its dependencies in the system structure and how the objectives of the design pattern is supported. [53]

Implementation
Important things to describe in the implementation section are pitfalls of implementing the pattern and what techniques are useful for implementing the pattern. Also, it is good to mention about any language-specific issues regarding implementation of the pattern. [53]

Sample code
According to Gamma et al. sample code of an implementation of the pattern should be in C++ or Smalltalk [53]. However, the language of the sample should not be a reason for not documenting a design experience, so the chosen language should probably be one of the most used langauge in that time era or same as used by the team involved.

Known uses
Known uses would be examples of where the pattern has been used [53].

Related patterns
Related patterns section should describe patterns that closey relates to the documented pattern. Also, patterns that should be used in combination with the pattern should be described in this section. [53]

3.5 SOLID principles

Thomas and Hunt already stated in the end of 1990's that changes occur in a near-frantic pace[10]. Thomas and Hunt argue that code must be flexible and loosely coupled in order to keep up with the pace changes occur [10]. The SOLID principles consist of techniques on how to implement flexible and loosely coupled software. The principles were first described by Robert C. Martin in his first version of the book: "Agile Software Development, Principles, Patterns, and Practices". [58][59]. SOLID consist of five principles:

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation
- Dependency inversion(/injection)

Each principle will be discussed in their own subsections. Also, a couple of ways of implementing the principles in C# will be presented. The SOLID principles are included in the principles of object oriented design [49].

3.5.1 Single responsibility principle

The single responsibility principle (SRP) is a way to achieve good system design. Robert C. Martin created the definition of SRP based on the principle described by Tom DeMarco and Meilir Page-Jones, which they called *cohesion*. Martin's definition for SRP: "A class should have only one reason to change". Martin defines a *reason for change* as a responsibility. From this follows, that for a class to have only one reason to change, it should have only one responsibility. A class with several reasons to change has several responsibilities. Classes having multiple responsibilities, causes responsibilities to be coupled. This leads to fragile design, where changes to one responsibility can cause others to stop working properly. [49][58]

SRP is one of the most difficult to get right, even though it is one of the simplest of the SOLID principles [49]. To give a better understanding of SRP, it will be demonstrated through a simple example. The example scenario has the starting point seen in Figure 4. Temperature parts of House implementation can be seen in Listing 1. The House class is responsible for managing both the temperature and the front door. If there will be no reasons for the responsibilities to change at different times or at all, then there would be no need to apply SRP [49].

However, for the purpose of this example, changes will occur to both responsibilities at different times in the future. This will cause recompilation and redeployment of the House class each time either responsibility receives changes, e.g. new features or logical improvements. To avoid unnecessary compilations, we will first separate the responsibilities. Then we will introduce interfaces to avoid the unnecessary compilations of the House class. [49]

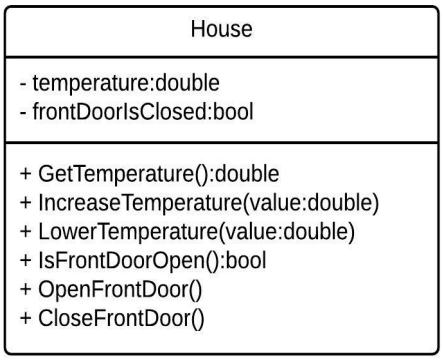

Figure 4: A House class with two responsibilities.

```
public class House{
        private double _temperature;

        ...

        public double GetTemperature(){ return _temperature; }
        public void IncreaseTemperature(double value){ _temperature += value; }
        public void LowerTemperature(double value){ _temperature -= value;}

        ...
}
```

Listing 1: Temperature parts of the House class implementation.

McLean Hall states that "through a process of delegation and abstraction, a class that contains too many reasons to change should delegate one or more responsibilities to other classes" [58]. To follow his statement, we will create two new classes: TemperatureControl and Door. Then, by changing the House class to use the TemperatureControl and Door classes, the responsibilities are delegated to the new classes. The design change can be seen in Figure 5, and example code change of the House class *IncreaseTemperature* method can be seen in Listing 2.

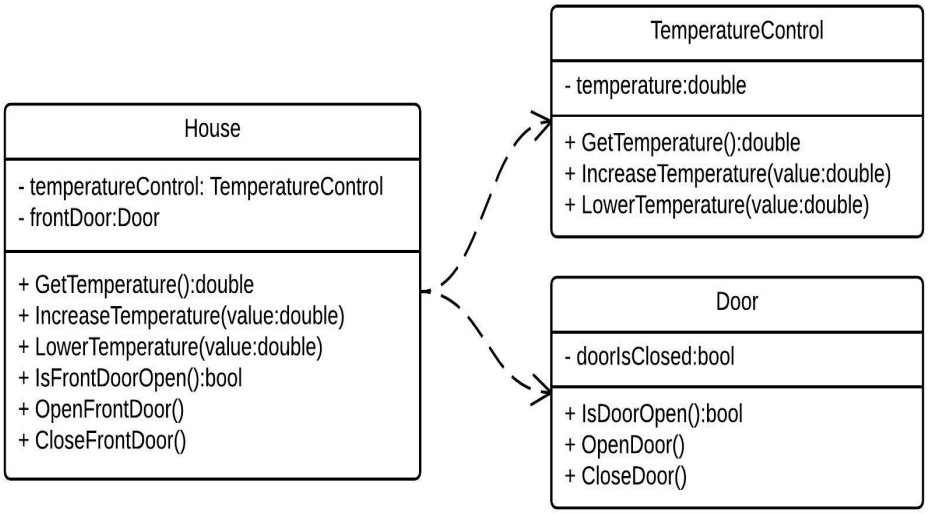

Figure 5: Temperature and door responsibilities in separated classes.

```
public class House{
        private TemperatureControl _temperatureControl;
        ...

        public void IncreaseTemperature(double value)
        {
                _temperatureControl.IncreaseTemperature(value);
        }

        ...
}
```

Listing 2: Changes to IncreaseTemperature method in House class.

Now that the responsibilities have been separated, there is still one issue with the House class regarding compilation. The House class is directly depending on both the TemperatureControl class and the Door class. Because of this, each time either TemperatureControl or Door changes, a recompilation is needed. The House class would also have to be recompiled. Also, if there would be a need to change the Door class for another implementation of the door functionality, changes would have to be made to the House class. Solutions to these issues will be discussed in the following sections.

SRP has a great positive impact on code adaptability, and it also improves clarity and order of a software [58]. One main part of software design is finding and separating responsibilities [49]. This is why SRP can be seen in one way or the other in the examples presented when the rest of the SOLID principles are discussed.

3.5.2 Open/Closed principle

The idea for the Open/Closed principle (OCP) was introduced by Bertrand Meyer in his book in 1988. Meyer introduced two concepts: open modules and closed modules. Meyer defines a module that can be extended to be open, and a module that can be used by other modules to be closed. Because requirements change over time, it would require that a module would have to be kept open until completed. However, for the module to be useful it should be available for other modules, which according to Meyers definition requires the module to be closed. The problem has been solved in classical design approaches by closing the module when it seems to be done, and then reopened it when modifications are needed. Unfortunately, this leads to a new problem when a module has been modified. All modules depending on the modified module also has to be reopened, so that required changes can be made to comply with the changes done to the modified module. Object-oriented design with inheritance enables a solution for the problem, which will be discussed shortly. [60]

Since Meyer introduced the open and closed definitions, two definitions have evolved according to McLean Hall. The first definition quoted in McLean Hall's book [58] is

the Bertrand definition of OCP:

"Software entities should be open for extension but closed for modification"

The second is a more elaborated definition by Robert C. Martin:

"Open for extension: This means that the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does."

"Closed for modification: Extending the behavior of a module does not result in changes to the source or binary code of the module. The binary executable version of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched."

The solution to extending a module, without opening the module and make modifications to the source code, is through abstraction. These abstractions are abstract base classes and inherited classes that represent the possible group of behaviors. This creates a fix abstraction that closes the module from modifications and allows behaviors to be added through inheritance, which makes the module open for extensions. There are at least two patterns for creating a structure that follows OCP: Strategy pattern and Template method pattern. Both patterns will be further discussed in the Design Pattern section. [49]

McLean Hall argues that a module that is open for extension can be extended through extension points. An extension point is a point where new behavior can be added by hooking future functionality into the existing code. McLean specifies three extension points in his book: virtual methods, abstract methods, and interface inheritance. The first two extension points are possible through class inheritance. Methods that are allowed to be overridden in child classes must include the 'virtual' keyword in the parents method signature in C# [61]. [58]

McLean Hall states that the preferable extension point is the interface inheritance. Both abstract and virtual methods are available through implementation inheritance. This leads to a subclass, which provides the extension, depending on the base class implementation. Instead of reducing dependencies, more will be added with each new subclass. The original client still depends on the base class, and the subclasses then also depend on their base classes. This is why only the interface inheritance extension point will be demonstrated in this section. However, subclassing will be discussed in the next section on Liskov substitution principle. Examples of abstract and virtual method extension points can be found in McLean Hall's book 'Adaptive Code via C#'. [58]

To demonstrate the interface inheritance extension point, a solution for the recompilation issue that was discussed in the SRP example will be presented. The starting state of the example can be seen in the previous section in Figure 5. Because of the

House class has a direct dependency to Door and TemperatureControl classes, it depends directly on the specific implementations. If there would be a reason to switch one of the classes, e.g. switching the Door class for WindowedDoor class, then it would require opening the House class for modifications. This would not comply with OCP because it brakes the rule of a module being closed for modifications. One way to solve the problem is to introduce interfaces between the classes [49]. To decouple Door class from House, an IDoor interface will be introduced. The interface specifies the same method signatures as Door class, which specifies the contract between the implementation and the client module. The same changes will be made with the TemperatureControl class. The design changes can be seen in Figure 6.

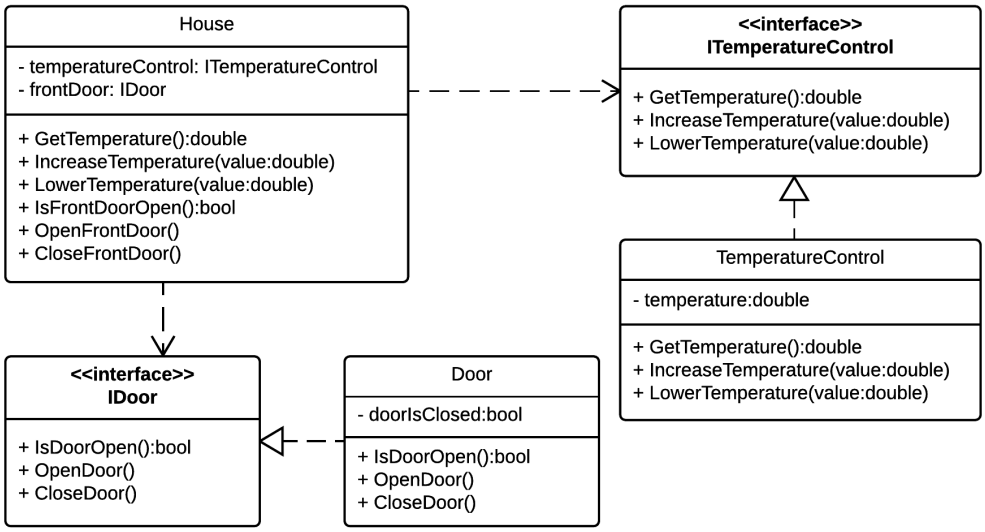

Figure 6: Final example solution.

Now that the House class depends on two interfaces, instead of on two implementations, it is closed for modifications and open for extensions. However, this only applies as long as no changes are required inside of the House class. As long as only the front door or the temperature control functionalities need to be extended, it can be done by introducing new implementations that follow the contracts specified by the interfaces. However, even though interfaces were added to the design in this example, it might not always be the best solution. Both McLean Hall and Martin keeps reminding that abstractions should be added when they are needed, and not just because it is possible to add more abstractions. When it is likely that a user requirement will change in the future, then interfaces are a good tool for preparing the architecture for the change and not breaking OCP. [49][58]

3.5.3 The Liskov substitution principle

The Liskov substitution principle (LSP) was introduced by Barbara Liskov in a conference keynote in 1987, titled 'Data abstraction and hierarchy'. Liskov presented the following substitution property: "If for each object o1 of type S there is an object o2 of type T such that or all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T. " [62]. This statement describes the idea of a structure in a class where some logical part's implementation can be replaced with another implementation, without the need to modify the class itself. LSP gives guidelines on how to create inheritance hierarchies so that subtypes of a base type are substitutable [49]. By following LSP, it is possible to create interfaces that clients can follow without being dependent on the actual implementation [59]. A scenario where OCP must be violated to make a change in the implementation, it is mostly likely also a sign that LSP is violated [49]. LSP makes it possible to achieve OCP, because when parts of an implementation can be substituted there is no need to brake OCP.

To get a better grasp of the principle, it will be demonstrated through an example starting from the scenario shown in Figure 7. The example scenario starts with a House class that depends on a Door class. To keep this example simple, a House only has one door. The example implementation of House and Door classes can be seen in Listing 3.

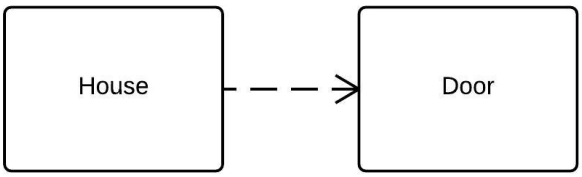

Figure 7: Starting scenario with a House and a Door class.

```
public class Door{
        ...

        public Door(double width, double height, double thickness){
                this.width = width;
                this.height = height;
                this.thickness = thickness;
        }

        public void open(){
                this.doorIsOpen = true;
        }

        public void close(){
                this.doorIsOpen = false;
        }
}

public class House{
        Door door;

        public House(Door door){
                this.door = door;
        }
        ...
}
```

Listing 3: Implementation of the Door and House class.

If no changes will occur in the future, then there are no problems with the structure. However, to demonstrate the Liskov principle, there is a request from a user that she wants a door with a window. To comply with the request the Door class has to be substituted to a WindowDoor class, which represents a door with a window. The new scenario can be seen in Figure 8. This new requirement forces the implementation of House to be opened and modified. Example modification of House class that violates OCP can be seen in Listing 4.

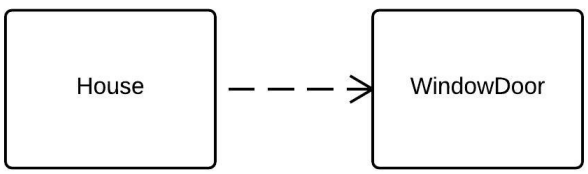

Figure 8: Door class is replaced by WindowDoor.

```
public class WindowDoor{
        ...
        private Window window;

        public WindowDoor(double width, double height,
                          double thickness, Window window){
                this.width = width;
                this.height = height;
                this.thickness = thickness;
        }

        public void open(){
                this.doorIsOpen = true;
        }

        public void close(){
                this.doorIsOpen = false;
        }
}

public class House{
        WindowDoor door;

        public House(WindowDoor door){
                this.door = door;
        }
        ...
}
```

Listing 4: Modified House class and new WindowDoor class implementations.

This modification forces House objects to depend on WindowDoor type doors. This is a good time to introduce LSP into the implementation. According to McLean Hall, there are three ingredients that relate to LSP: a base type, a subtype, and the context [58]. The base type is the type clients should depend on. The subtype should inherit from the base type so that clients does not know which subtype is actually being used. The context refers to the way a client interacts with a client. As discussed earlier in the section, the supertype and subtype Liskov discuss in her article [62] are the same as the base type and subtype McLean Hall present in his book [58].

McLean Hall also discusses LSP rules in his book, which he splits into contract and variance rule categories [58]. These rules are only shortly discussed at the end of this section because the type substitution is still possible without implementing the contracts.

To follow LSP, a base class is needed in the example. Let us add an abstract BaseDoor class, which the House class depends on. This can be seen in Listing 5.

```
public abstract BaseDoor{
        ...

        public BaseDoor(double width, double height, double thickness){
                this.width = width;
                this.height = height;
                this.thickness = thickness;
        }

        public void open(){
                this.doorIsOpen = true;
        }

        public void close(){
                this.doorIsOpen = false;
        }
}

public class House{
        BaseDoor door;

        public House(BaseDoor door){
                this.door = door;
        }
}
```

Listing 5: Abstract BaseDoor class and modified House class.

Now any door class that inherits the BaseDoor class can be given as an argument to the House constructor. Let us modify both Door and WindowDoor classes to inherit the BaseDoor class, so that they can be used as doors for House objects. The modification can be seen in Listing 6.

The new structure of the example can be seen in Figure 9. The House class only depends on the BaseDoor class, and any child class of BaseDoor can be added to the House class. Now the structure allows substituting the door for any door that inherits the BaseDoor, which follows the substitution property presented by Liskov.

However, the example solution does not force any rules on the implementation of the child classes. McLean Hall presents in his book [58] two rule categories that enforce inherited classes to comply with LSP: contract rules and variance rules. Contract rules deal with expectations of classes and variance rules deals with type substitution. [58]

Contract rules

Contract rules consist of three types: preconditions, postcondition, and data invariants. Preconditions describe conditions which a method should fulfill. An example of

```
public class Door : BaseDoor{

        public Door(double width, double height, double thickness)
        : base(width, height, thickness){
        }
}

public class WindowDoor : BaseDoor{

        private Window window;

        public WindowDoor(double width, double height, double thickness,
                          Window window) : base(width, height, thickness){
                this.window = window;
        }
}
```

Listing 6: Door and WindowDoor classes inherits abstract BaseDoor class.

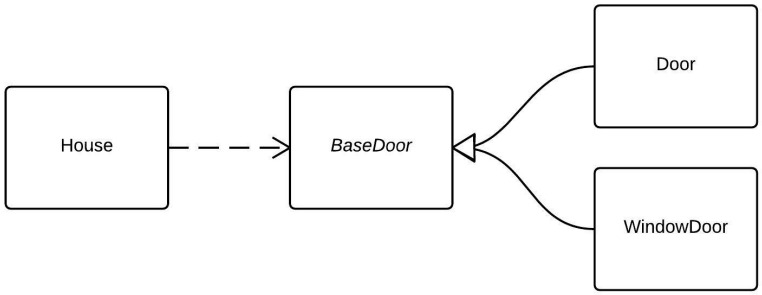

Figure 9: Starting scenario with a House and a Door class.

a precondition could be to always check that if only positive integers are allowed as arguments, then it is also checked and enforced or the method fails. Postconditions relate to a condition that e.g. an object should fulfill after it is modified and before the method returns the object. If the condition fails, the method should fail. Data invariant refers to a condition where data should always fulfill some criteria. An example of a data invariant could be the height of building represented as a double. A simple condition for the height property would be that it should always be positive. An invariant contract could require that the height is always kept positive when changing the value. [58]

McLean Hall introduces three variance rules: contravariant method arguments in subtypes, covariant return types in subtypes and subtypes cannot throw new exceptions unless they are subtypes of exceptions used in base class. Covariant return types allow the return type of a method to be substituted to a subtype in an inherited class. With contravariant method arguments, the supertype-subtype hierarchy is turned around. A method that takes supertype as an argument can be

given a subtype instance. However, in C# contravariance and covariance can only be used with generic types. [58]

3.5.4 Interface segregation

The interface segregation principle (ISP) deals with interfaces that are non-cohesive and too large, in other words "fat" [49]. One disadvantage of a "fat" interface, is that clients implementing the interfaces are required to implement all the methods even though they might not be needed. The principle will be clarified through an example, which starts from the scenario shown in Figure 10.

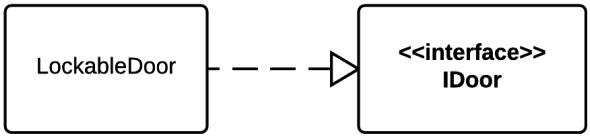

Figure 10: LockableDoor class depending on IDoor interface.

The example consists of a LockableDoor class and an IDoor interface. Listing 7 display the implementation of the IDoor interface. A lockable door can be opened and closed, and also locked and unlocked. The methods defined in IDoor seem to make sense for this scenario.

```
public interface IDoor{
        void lock();
        void unlock();
        void open();
        void close();
}
```

Listing 7: "Fat" non-cohesive interface.

However, doors do not always include a lock, so a Door class is also needed. It should make sense that the Door class implements the IDoor interface, which can be seen in Figure 11. Now if we look again at Listing 7, it clearly states that the new Door class must implement both the 'lock()' and 'unlock()' methods. The door does not have locking and unlocking capabilities, so it does not make any sense that it should implement the method.

The conclusion that can be drawn from this scenario is that IDoor might define more functionalities than it should. Let us move the locking functionality methods to a separate interface ILockable. Then LockableDoor class can implement the IDoor interface to get the door functionalities, and also implement the ILockable interface to get the locking functionality. This can be seen in Figure 12.

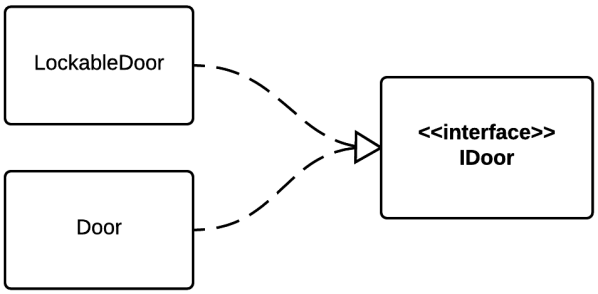

Figure 11: Door class added to architecture.

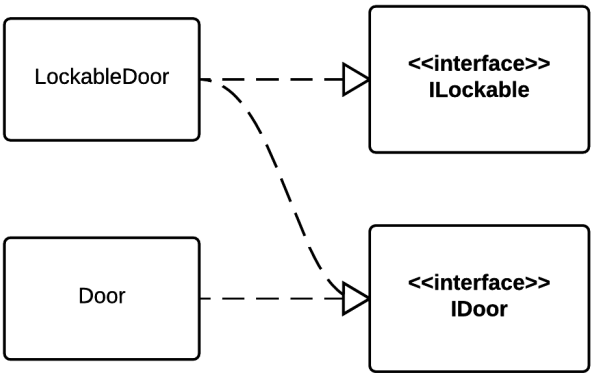

Figure 12: ILockable interface introduced.

Implementations of the two interfaces can be seen in Listing 8. IDoor and ILockable are now both cohesive, and also handles one responsibility each. This makes it possible for classes to implement only the functionality they really need, instead of being forced to implement methods that do not belong to a specific class. If a class is required to implement several interfaces, an abstract base class that implements all the interfaces could be created as Martin suggests in his book [49].

```
public interface IDoor
{
        void open();
        void close();
}

public interface ILockable
{
        void lock();
        void unlock();
}
```

Listing 8: Cohesive interfaces.

3.5.5 Dependency inversion

The last principle in SOLID principles is dependency inversion (DIP), which is called dependency injection in Halls book [58]. In traditional software development methods, according to Martin [49], software tends to be created in a way were high-level modules depend on low-level modules and policies depend on details. This is usually the case with procedural programming [49]. Martin also states that the dependency structure is inverted in object-oriented programming and that often clients are the ones owning the service interfaces [49]. But this does require good object-oriented design, otherwise, the dependency structure might still have a procedural structure.

To better understand dependency inversion, it will be discussed through an example. The example demonstrates a simple dependency between two classes, Object Creator and JSONBuilder. The dependency relationship can be seen in Figure 13. A simple implementation of the example scenario can be seen in Listing 9. By looking at both the dependency relationship and the code snippet, it can be seen that the higher-level ObjectCreator class depends on the lower-level class JSONBuilder.

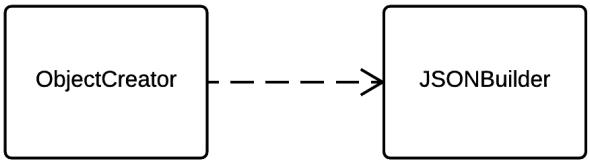

Figure 13: Higher-level ObjectCreator depending on lower-level JSONBuilder.

The code snippet in Listing 9 demonstrates a simple implementation of the classes displayed in Figure 13:

```
public class ObjectCreator{
        private JSONBuilder _jsonBuilder;

        public ObjectCreator(){
                _jsonBuilder = new JSONBuilder();
        }

        public object Create(string json){
                object obj = jsonBuilder.Build(json);
                return object;
        }
}
```

Listing 9: Simple hidden dependency example.

From the code snippet, there can also be seen that the dependency is hidden to clients using the ObjectCreator, due to the fact that there is no way for the client to know about the dependency unless the client has access to the source code or documentation describing the dependency [58]. Also, when there is a need to change JSONBuilder to some other builder class, the modification cannot be made internally in the class without breaking OCP. This means that a client has no control over which builder class to use. However, in Java, it would be possible to inherit the ObjectCreator and override the Create method, but this is only possible in C# when the parent method is declared as virtual [61]. However, it possible to hide the inherited 'Create' method with the 'new' keyword in C# [63], but this can cause other problems which are not discussed in this thesis. Hidden dependencies, like the one present in the example, also makes it difficult to create unit tests [58].

According to Martin [49] "dependency inversion can be applied wherever one class sends a message to another". In the example in Listing 9 the JSON string is forwarded to the builder class, which can be seen as a message to the builder. This is, at least, one reason why DIP can be applied to the example problem. To solve the problem, first the abstractions have to be separated from the details [49]. ObjectCreator depends directly on the details of the JSONBuilder. To separate the abstraction from the detail, the _jsonBuilder variable should depend on an interface instead of concrete class. Secondly, instead of initializing a JSONBuilder instance internally, it should be initialized externally and given to the ObjectCreators constructor method. The code snippet in Listing 10 shows how the code can be refactored follow DIP.

```
public interface IBuilder{
        object Build(string data);
}

public class ObjectCreator{
        private IBuilder _builder;

        public ObjectCreator(IBuilder builder){
                _builder = builder;
        }

        public object Create(string json){
                object obj = _builder.Build(json);
                return object;
        }
}
```

Listing 10: Refactored class that follows DIP.

In Figure 14 the same change is displayed as a simple schema. Now the ObjectCreator depends on the interface instead of the implemented builder, and JSONBuilder also depends on the interface. In the future when the builder might have to be changed to another builder, the ObjectCreator class does not require any modifications as long as the new builder class implements the IBuilder interface.

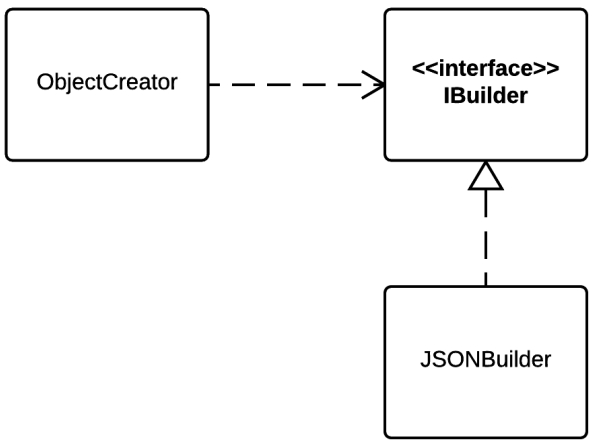

Figure 14: Higher-level ObjectCreator depending on lower-level JSONBuilder.

3.6 Summary

This chapter presented different views on how to develop software that is easier to maintain in the future. Preparing for future changes combined with designing for

maintainability, future maintenance costs may be reduced and less effort overall will be needed to adjust to changing requirements. This chapter also discussed two techniques that can help both in the design and implementation work. Design patterns provide techniques to document design experience and create reusable software modules. The SOLID principles gives ideas on how to design object-oriented software systems and techniques on how to implement maintainable object-oriented software.

4 Library design

The main purpose of this thesis is to design a solution that solves the problems described in Section 2.3.3. The chapter begins with a motivation for why redesign was necessary, in Section 4.1, followed by a description of the requirements for the redesign in Section 4.2. Then the chapter continues with a short introduction of the problem domain in Section 4.3. Finally, the architectural views of the redesign and a rationale are presented in Section 4.4. The IEEE 42010:2011 standard [64] was partly used as a conceptual framework for this chapter.

4.1 Need for a new design

The preferred way of solving the current maintainability problem was to refactor the implementation to minimize the risk of losing domain knowledge. Refactoring removed a lot of code duplication. However, this did not solve all of the problems discussed in Section 2.3.3. The refactored code did reduce the number of potential change locations, but the logic was still fragmented into separate parts of the data import subsystem. Refactoring only changes the code structure and not the behaviour of a module. The performance problems mentioned in Section 2.3.3, excess IO usage, and scaling could not be solved only by refactoring the existing implementation. To solve these problems a redesign of the data import functionality was necessary.

4.2 Requirements

This section focuses on identifying the stakeholders and requirements for the library. The library stakeholders consist of the users, developers, maintainers and the core of Sapphire Build. The users are data managers working for the customer companies. The developers refer to the people implementing the design and integrating the library into Sapphire Build. Maintainer refers to individuals that do maintenance work on the library. The implementers of the library are also the most likely maintainers of the library. Since the library will end up as a module of Sapphire Build, which leads parts of Sapphire Build to depend on the library. Because of this, Sapphire Build can be seen as a stakeholder. Stakeholders are shown in the use case diagram in Figure 15.

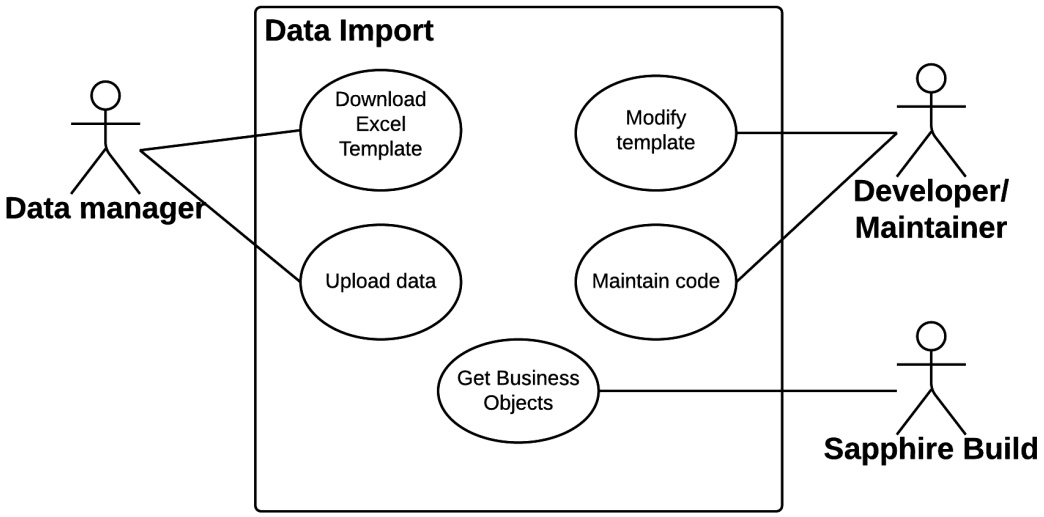

Figure 15: Library's use case diagram.

The purpose of the current implementation is to provide data import functionality to customer's data managers. Data managers need to be able to download the latest version of an Excel template file, fill in the data into the template and then upload the Excel file to the system. These requirements are also part of the new data import library.

Developers and maintainers need to be able to make changes to data upload templates and maintain the code. The library is easier to maintain when the number of change points is minimal. Compared to the fragmented separation of the data import logic seen in Figure 3 the redesign should try to centralize the metadata that will change over time.

The purpose of the library is to provide the business logic for Sapphire Build's data import functionality. The library should not depend on any particular user interface. That way, any changes in a user interface does not affect the library. Decoupling the library from any potential user interface also means it can be used with different user interfaces. Also, maintainers of the library should not have to deal with the logic that handles business objects persistence. Instead, persistence implementation should be provided by Sapphire Build according to library's needs.

Since the library should be easy to maintain, it should also be testable. When a maintainer makes a change, he or she should be confident that the change has not caused any ripple effects in other parts of the code base. To reduce the risk of unwanted ripple effects a safety net should be used [28]. An automated test suite can work as a safety net that provides early feedback about a change breaking other parts of the system [28]. The library should not only be testable, but should also include a safety net for future maintainers.

The last stakeholder, Sapphire Build, should receive business objects from the library and persist them. Sapphire Build should not depend on the library's business

logic implementation. Also, the library should not depend on the business object persistence implementation. The communication between the library and Sapphire Build should be through an interface.

The next section will shortly describe some additional requirements, after which Section 4.3 discusses the problem domain.

Additional requirements
Performance issues were also discussed in Section 2.3.3. Since the focus of this thesis is on maintainability, the performance issues are not as important as the maintainability issues in the solution. However, the performance issues should be taken into consideration in the library design, so that it is possible to improve the performance in the future.

4.3 Problem domain

This section shortly describes the problem domain of the data import library. A visualization of the problem domain is shown in the domain model in Figure 16. The purpose of the domain model is to clarify the problem domain [65]. The domain model describes the concepts of the problem domain and their associations and cardinalities. Concept's attributes are also specified in the domain model. The definition of each concept is outlined in the concept glossary in Table 1. Attribute definitions are defined separately to ease readability and are shown in Tables 2 - 6.

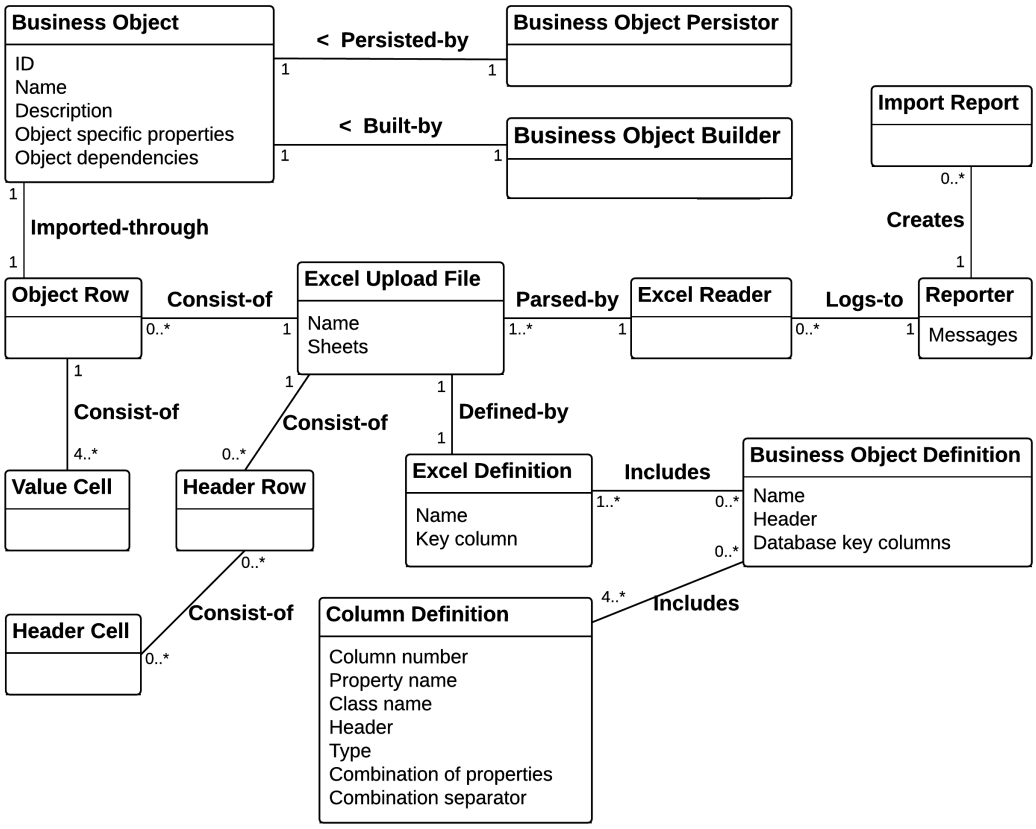

Figure 16: Domain model for the Excel import library.

Concept	Definition
Business Object	A class in Sapphire Build that describes a business object, like a community, a lot, etc.
Business Object Persistor	A component in the system that stores business objects in the database
Business Object Builder	A component that builds a business object from data representing the business objects properties.
Excel Reader	A component that parses the content of an Excel upload file.
Reporter	Enables components to log process events in the system by sending messages to the Reporter.
Import Report	A report created by a Reporter, which inlcudes all received messages. Displays the success of the import process.
Excel Upload File	An Excel file with business object data that should be imported to Sapphire Build
Object Row	A row in an Excel Upload file that represents one business object's data.
Value Cell	A column in an Object Row, which represents the value of a business object property.
Header Row	A row in an Excel Upload file that represents a header for an Object Row.
Header Cell	A column in a Header Row.
Excel Definition	A file which described metadata about an Excel Upload file.
Business Object Definition	A file which describes metadata about a business object in a Excel Definition file.
Column Definition	Defines metadata about a column in an Excel Upload file.

Table 1: Concept glossary for the domain model in Figure 16.

Attribute	Definition
ID	The business object identifier.
Name	The business object name.
Description	Describes the business object.
Object specific properties	Properties that are specific for a business object type.
Object dependencies	Describes which other business objects the business object depends on.

Table 2: Business Object's attribute definitions.

Attribute	Definition
Name	Name of the Excel file, excluding the suffix.
Sheets	Sheet names in an Excel file.

Table 3: Excel Upload File attribute definitions.

Attribute	Definition
Name	Describes the name of the Excel file. Should match the name of an Excel Upload File
Key Column	Describes the Excel column that should hold the type of a business object represented on an Object Row.

Table 4: Excel Definition attribute definitions.

Attribute	Definition
Name	Describes the business object type name.
Header	Describes header for section in an Excel Upload File.
Database key columns	Describes the columns in an Object Row that should be used as a key to retreive the business object from the database if it exists.

Table 5: Business Object Definition attribute definitions.

Attribute	Definition
Column number	Describes which column in a Excel row this definition defines.
Property name	Describes which business object property this column belongs to. Can be empty in case the column is a combination of multiple properties.
Class name	Describes the type name of the business object, which property this column describes. The name also matches a table in the database.
Type	Describes the type used in Sapphire Build for the column value.
Combination of properties	Is used to describe which properties are combined in this column.
Combination separator	Describes the character used in a Value Cell to separate the properties..

Table 6: Column Definition attribute definitions.

In many cases, the concepts described in a domain model can be used as a base for finding the objects needed in a solution. The concepts in the domain model in Figure 16 was used as a base for naming objects in the solution. The domain model is a great communication tool when communicating with a team or customers since it reduces concept misinterpretations when they are clearly defined.

The domain model and the concept glossary should give an overview of the problem domain and the base knowledge to make it easier to understand the architecture views in Section 4.4. The the next section discusses different views of the library architecture.

4.4 Architecture

The purpose of this section is to present the design of the solution. First, the centralization of changes view is shown in Section 4.4.1. Second, the structural view is discussed in Section 4.4.2. The interaction view of the library is covered in Section 4.4.3. Section 4.4.4 presents how the metadata should be stored. Finally, the decisions behind the architecture are discussed in Section 4.4.5.

4.4.1 Centralization of changes

The current fragmented approach for the data upload logic was described in Section 2.3.3. The new design takes a more centralized approach regarding where changes need to be made in the data upload logic. Instead of having to remember to make changes in three different locations as in Figure 3, changes are instead focused to metadata files. An overview of this new approach can be seen in Figure 17.

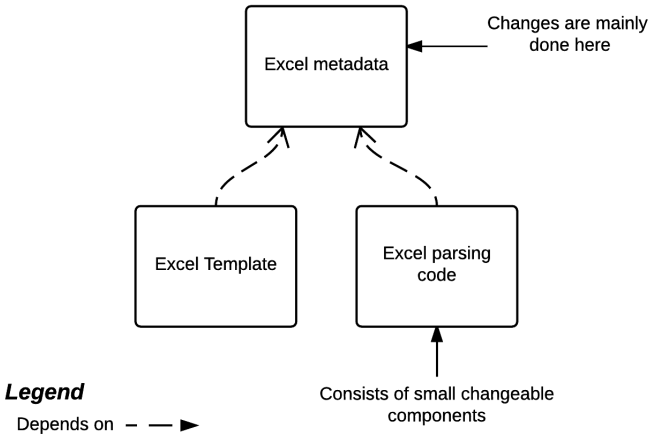

Figure 17: New centralized approach for the excel upload logic.

The Excel Metadata part in Figure 17 describes metadata files, which are described further in Section 4.4.4. In this new design, Excel templates are generated from

Excel metadata files, which means that users that want to upload data should first download the latest Excel template in order to avoid errors. However, clients might already have data stored in older versions of an Excel upload template that has not been previously uploaded to the system which can cause future problems that must be sorted out. However, the versioning issue is outside of this thesis scope and needs to be considered in future work. The Excel parsing code part refers to the actual code of the library. Even though the metadata files are considered as a separate part, it is still in the heart of the library and is represented in code in the parsing part of this design. The parsing code part describes the logic of the library, which is further discussed in Section 4.4.2.

4.4.2 Library components

The library consist of six different components, each providing one interface for communication purposes. Each component has one single responsibility in the data import process. The purpose of each component is described in the component glossary in Table 7. The interfaces are described in the interface glossary in Table 8.

Component	Purpose
Excel Importer	The entry point component of the library.
Reporter	Receives messages from components and builds a report on how the data import succeeded or why it failed.
Template Generator	Generates Excel Upload templates that users can fill with their data that they want to import.
Excel Reader	This component focus on reading an Excel file.
Row Handler	The Row Handler receives a row, checks the business object type and routes the row forward to a data collector.
Business Object Data Collector	Collects the property values from a Data Row.
Business Object Reader	A Sapphire Build component that reads data from a database.
Business Object Builder	Builds business objects from the collected properties.
Business Object Persistor	A Sapphire Build component that handles data persistence.

Table 7: Component glossary for the component view in Figure 18.

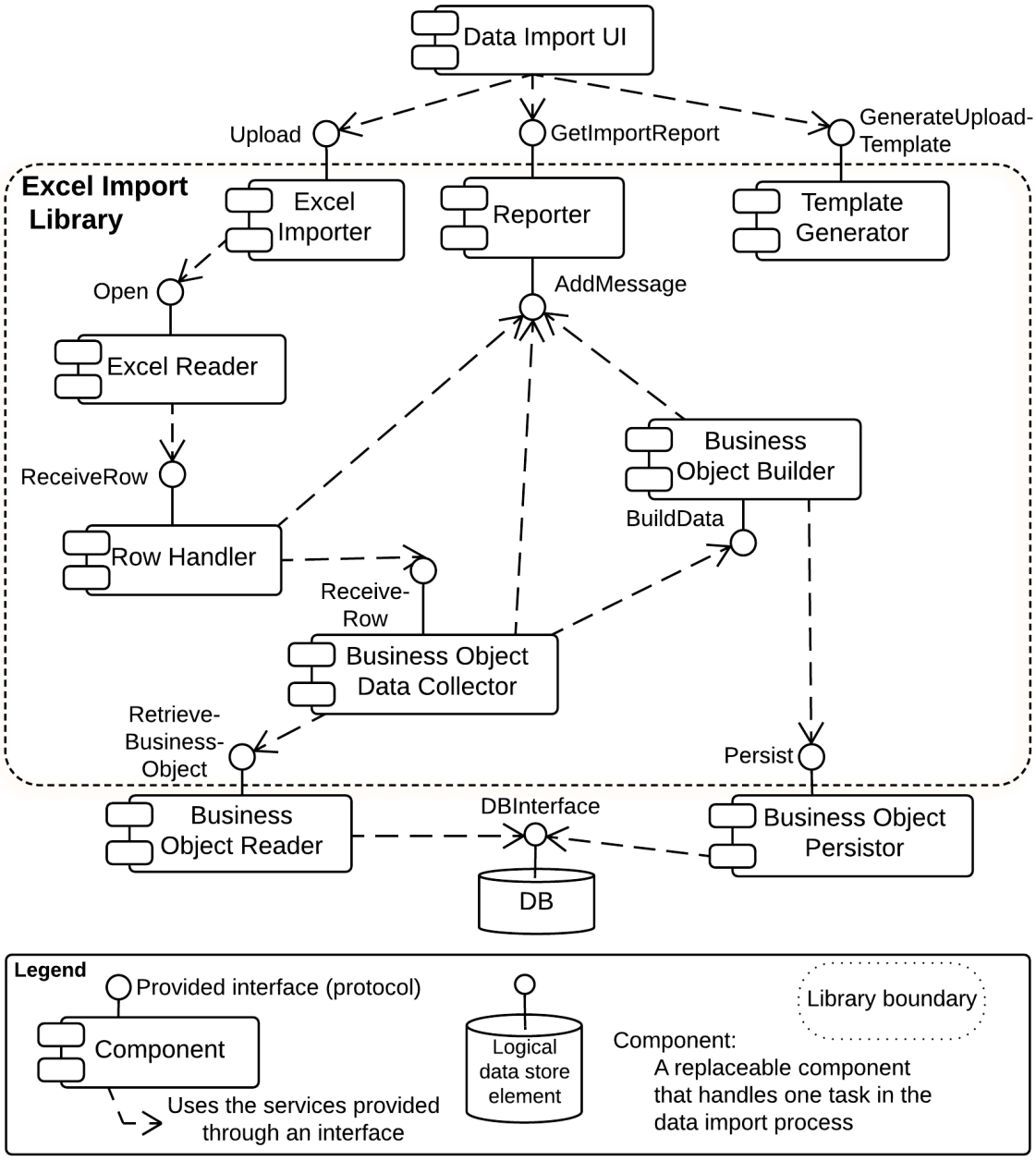

Figure 18: Components of the library.

The connections between the components are shown in Figure 18. Four of the concepts described in the domain model in Section 4.3 provided names for four of the components in the architecture: Excel Reader, Reporter, Business Object Builder and Business Object Persistor.

Components communicate with each other through the described interfaces, which will decouple different component implementations from each other. Decoupled implementations make it easy to replace one component implementation with a new implementation, as long as it implements the exact same interface as the old

implementation. Also, by making sure that the components are loosely coupled, will make it easier to switch components old implementations to new implementations. Replaceable components makes it easier to apply changes, which increases the maintainability of the library.

Interface	Purpose
Upload	Provides the gateway for the user interface to import data from an Excel file.
GetImportReport	Provides the report from the data import process.
AddMessage	Provides logging functionality to the functional elements.
GenerateUploadTemplate	Provides Excel Upload templates to the user interface for users to download.
Open	Starts the Excel reading process.
ReceiveRow (Row Handler)	The purpose of the interface is to enable a Row Handler to receive Data Rows
ReceiveRow (Data Collector)	The purpose of the interface is to provide a channel for Row Handler to transport a Data Row to a Data Collector for processing.
RetreiveBusinessObject	Provides Data Collector the ability to retreive existing business objects from the database.
BuildData	Enables Business Object Builder to receive a collection of business object properties that should be built into a business object.
Persist	Provides business object persistance functionality to Business Object Builder.

Table 8: Interface glossary for the component view in Figure 18.

4.4.3 Component interactions

One of the biggest differences in the new design compared to the old implementation, is that the new design does not require that a whole Excel file is read into memory and then stored in a temporary database table after which the data is loaded back into memory for parsing and updating existing objects or adding new objects. Instead of the old approach, the library is designed as a pipeline consisting of components that handles received data and pushes the result forward to the next component [66][67]. The designed data flow can be seen in the sequence diagram in Figure 19.

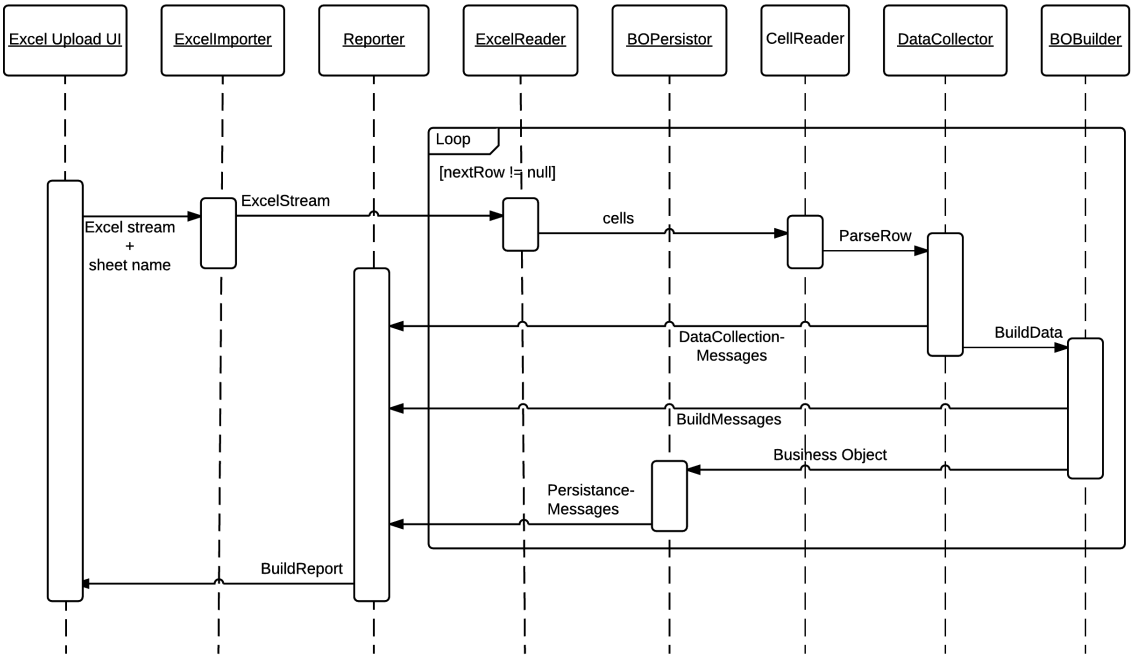

Figure 19: A sequence diagram of the new data upload process.

In this design the pipeline takes an Excel upload file as an input and produces a report on how the upload succeeded. The design also improves reporting capabilities for the data import feature, since components are injected with a Reporter type object that receives messages from each stage and builds a report in the end of the process.

4.4.4 Metadata file structure

The purpose of the metadata files is to have a centralized way of storing metadata related to specific Excel upload files.

The metadata is located in several different locations in the current data upload implementation, which requires changes to be made in more than one place. The fragmentation of the metadata has caused parts to end up unsynchronized over time. The centralized approach should prevent this problem in the future. Excel upload template files are generated by the Template Generator component based on these metadata files.

The metadata is stored in JSON format, since it makes it relatively easy for both humans and machines to read the files [68][69]. Also, JSON is very lightweight [69] and there is an option to validate JSON with a schema if needed [70]. JSON is also well supported in the .NET framework. Metadata for an Excel upload file is stored in a minimum of two JSON files: one ‘Excel definition’ file and one or many ‘Business object definition’ files. The names match to of the concepts described in the domain model in Section 4.3.

Excel definition files

The metadata about the base structure of an Excel upload file is stored in an 'Excel definition' file. Figure 11 shows an example of a community data upload Excel definition file. The definition file describes the name of the Excel file and the key column that is used to identify the object type of an Excel row. The definition file also describes the file names where the Business Object Definitions are stored, which was not included as an attribute in the 'Excel Definition' concept presented in the domain model in Section 4.3.

```
{
    "name": "6.1.1_UploadCommunityData",
    "keyColumn": 3,
    "businessObjectReferences": [
        "community.json",
        "section.json",
        "phase.json",
        "lot.json"
    ]
}
```

Listing 11: Example Excel definition file.

Business Object definition files

Business objects metadata are defined in so-called 'Business object definition' files. An example of the metadata related to Section business objects can be seen in Figure 12. A business object definition file starts with describing the name of the business object and which header should be displayed for the object rows in an Excel file. The name of the object builder class and database key columns are also defined in the definition file. The database key column describes which columns hold values that are used as a key for updating a certain business object that is stored in the database. Lastly, metadata about each column in an object row is described in the 'columns' list. Each object in the 'columns' list is defined according to the 'Column Definition' concept described in the domain model in Section 4.3. The definitions for each key-value pair in the column objects can be seen in Table 6. The example shown in Figure 12 does not include the last two attributes of a 'Column Definition'.

```
{
    "name": "Section",
    "header": "Section",
    "builder": "GenericBOBuilder",
    "databaseKeyColumns": [6],
    "columns": [
        {
          "n": 5,
          "class": null,
          "propertyName": "SectionID",
          "header": "Section ID",
          "type": "string"
        },
        {
          "n": 6,
          "class": null,
          "propertyName": "Name",
          "header": "Name",
          "type": "string",
        },
        ...
        {
          "n": 8,
          "class": "Community",
          "propertyName": "CommunityID",
          "header": "Community ID",
          "type": "string"
        },
        {
          "n": 9,
          "class": "Community",
          "propertyName": "Status",
          "header": "Status",
          "type": "string"
        }
    ]
}
```

Listing 12: Example business object definition file.

4.4.5 Rationale

The architecturally significant requirements included the maintainability requirements and the performance requirements discussed in Section 4.2. The driving reason behind the pipeline architecture came from the performance requirements, because of its scalability benefits. It should be relatively easy to run several data collectors and data builder components in parallel to gain data processing benefits. Also, the

performance requirements guided to the decision to read one row at a time and push it to the next component in the pipeline. The maintenance issues in the current data import implementation were one of the primary drivers for this work, which made the maintainability requirements critical. Since the pipeline supports separation of responsibilities very well, which supports the effort to design a maintainable system. The reason the performance requirements had a bigger impact on the design decisions than initially intended, was because making changes to the system to increase performance at a later stage could end up very costly.

The architecture could have been presented by using several different views. The IEEE 42010:2011 standard does not specify any particular view that should be used [64]. The centralization of changes view was chosen to give and overview of how the new architecture should differ from the old design. The component view was chosen, since it presents the main components of the system and their interfaces. The interfaces parameters were not specified, because they may wary depending on how they are implemented. A class diagram was also not presented, because of the same reason. The purpose of each interface was specified to give guidelines on what they should provide, but not how they should be implemented.

The interaction view added details about how the components should interact with each other. It also describes the part of the system that is executed several times during a single Excel upload, which may give hints on where to scale to get performance boosts in the future. Even though the presented metadata file structure is in its early stage, it does provide guidelines on where it is heading. The motivation for having different files for Excel definitions and Business Object definitions was due to maintainability and reusability reasons. The maintainability aspect of the decision was to reduce the information scope of each file. The reusability aspect of the decision was to enable easy customization of Excel uploads.

4.5 Summary

This chapter presented the design of the data import library. The chapter discussed the different requirements for the library design, especially focusing on the maintainability requirements. The problem domain was presented with the help of a domain model describing the concepts and their associations. The concepts were also defined in separate tables to increase readability. Finally, the architecture was presented through four different view: centralization of changes, library components, component interactions and metadata file structure.

5 Design Evaluation and Implementation

The purpose of this chapter is to discuss the evaluation of the design. The design was evaluated by implementing a proof-of-concept prototype based on the proposed architecture. According to Varma this is a common way of validating an architecture [71]. However, the only objective validation the prototype implementation provides is if it works or not. To gain a better understanding of the prototypes attributes, some sort of measurements are needed. This thesis focus on the maintainability attribute of a system, which is why only maintainability measurements are covered in this thesis. The next section will shortly introduce the most common way of measuring maintainability in the industry and present the different measurement metrics. Then, the design is evaluated based on the implementation in Section 5.2. The maintainability aspect of the prototype is first evaluated through the SOLID principles and design patterns, which works as an easy and daily way for developers to evaluating what has been produced. Then, the prototype is evaluated based on measurement metrics to give a more objective and accurate evaluation. Second, the design is evaluated based on the research question presented in Section 1.1. Finally, the design is evaluated based on the requirements discussed in Section 4.2.

5.1 Measuring maintainability and reusability

This section will shortly discuss metrics used for measuring maintainability and reusability. First, metrics for measuring maintainability is discussed. Second, metrics for measuring reusability is discussed. Some of the metrics presented in these sections are used when measuring the implemented prototype in Section 5.3. Techniques on how to measure maintainability and software in general is outside the scope of this thesis. More about measuring software and software metrics can be found in the 'Software Metrics A Rigorous & Practical Approach' book [72].

5.1.1 Maintainability metrics

According to Jones "measurement is the basis of all science, engineering, and business" [73]. DeMarco once said that "you cannot control what you cannot measure" [72]. Both of these statements clearly indicate that system attributes should be measured to gain necessary insight. Maintainability is one of many attributes of a system, which means that it should also be measured. There are multiple ways of measuring maintainability. Riaz et al. conducted a systematic review of software maintainability prediction and metrics in 2009 [74]. The results of their study did not suggest any maintainability prediction model that were an obvious choice. According to their study, however, the most common used metrics for maintainability predictors were: application size, complexity, and coupling. These three metrics are gathered from source code.

The Maintainability Index (MI) is one of the commonly used maintainability metrics. According to Heitlager et al. MI was proposed by Oman et al. in 1994 [75]. MI was suggested as a metric for determining the maintainability of a system in an objective manner [75]. The metric is calculated from the source code of a system and is based on three mandatory metrics and one optional metric. The three mandatory metrics are the Halstead Volume (HV) metric, the Cyclomatic Complexity (CC) metric, and the average number of lines of code per module (LOC). The optional metric is the percentage of comment lines per module (COM). A description of the Halstead Volume is found in [76] and the Cyclomatic Complexity is described in [77]. MI is not discussed in depth in this thesis, but the formula for calculating the index can be seen in Equation 1 [75]. More information about the Maintainability Index is found in [78].

$$MI = 171 - 5.2\ln(HV) - 0.23CC - 16.2\ln(LOC) + 50.0\sin(\sqrt{2.46 * COM}) \quad (1)$$

Heitlager et al. had identified limitations in MI through their experience of using the metric. The software engineering community has not widely accepted the Halstead Volume metric, because of the difficulties to define and compute the metric. Some comment lines are simply lines of code that have been commented out, which can make it difficult to calculate the number of comment lines per module. The probably most significant limitation with MI is that when it shows a result of low maintainability, it does not give any clues of where the maintainability problems might be. Although MI has its limitations, Heitlager et al. considers that it does provide more information about the state of a system than not having any measurements at all. [75]

The limitations in MI lead Heitlager et al. to formulate an alternative maintainability model. The model maps source code measures, chosen by Heitlager et al., onto maintainability characteristics defined in ISO 9126. The mapping is shown in Figure 9. Heitlager et al. state that the model has been tested and refined when used in dozens of software assessment projects. The alternative model is not discussed in depth in this thesis; the research paper 'A Practical Model for Measuring Maintainability' written by Heitlager et al. provides more information about the topic. However, some of these metrics will be used to gain more insight in maintainability attribute of the data import library. The chosen metrics for the prototype measurements are further discussed in Section 5.3. [75]

Source code properties	Maintainability characteristics			
	Analysability	**Changeability**	**Stability**	**Testability**
Volume	x			
Unit complexity		x		x
Duplication	x	x		
Unit size	x			x
Unit testing	x		x	x

Table 9: Source code properties correlation to maintainability characteristics presented by Heitlager et al. [75].

5.1.2 Reusability metrics

According to Poulin software reusability can be measured from different viewpoints, like the level of reuse in an organization, adherence to formatting standards and style guidelines, and existence of integration instructions and design documentation [79]. All of these viewpoints give indications about how reusable a software system or component is. Reusability metrics can also be measured from source code. Gui and Scott have studied coupling and cohesion measures as a way of evaluating component reusability, which they initially released a research paper in 2006 [80] and an extended version in 2008 [81]. They separately compared different existing cohesion and coupling metrics and two metrics proposed by them.

Coupling is measured by looking at how classes act upon others. The desirable result would be to have loosely coupled classes. Making a change in a class that is highly interdependent might cause unwanted side effects on other classes, which was already discussed in Section 3.5.1. [80]

Cohesion is measured by assessing whether similar set of instance variables are accessed by methods of a class. If the methods of a class access several different sets of instance variables, then the class is not cohesive. The single responsibility principle, discussed in Section 3.5.1, tries to increase the cohesion of classes since the characteristic of a well-designed subcomponent is high cohesion. [80]

This thesis will not go into details on how to collect and calculate these two metrics. More information about coupling and cohesion metrics calculations can be found in [81].

5.2 Evaluation based on implementation

The prototype was implemented as a own project separate from the Sapphire Build project, so that Sapphire Build would not introduce any overhead to the prototype. All external dependencies were either simulated or separately imported into the prototype. A simple in-memory database was implemented to simulate persistence functionalities that Sapphire Build would provide in production. The simple in-memory database

was developed to provide full control of the persistence functionalities, so that an external database would not cause any overhead or unecessary complexity to the proof-of-concept prototype. Because prototype performance measurements is outside the scope of this thesis, this does not affect the outcomes of the evaluation.

Automated tests were used to provide feedback about the maintainability and testability aspects of the prototype. Writing automated tests provided valueable feedback about potential class dependency issues. If a class was difficult or tedious to initialize in a test case it gave indications of potential dependency issues. Tests also produced feedback about classes responsibilities. The number of responsibilites for a class correlated with the number of different test cases that had to be written to prove the basic correctness of the class. Section 3.2 presented automated tests as one of the feedback sources on code quality. The prototype development process proved this to be true, since automated testing provided almost instant feedback about the code structure, complexity and quality.

The following two sections will present parts of the prototype implementation and discuss how the the SOLID principles and design patterns provided maintainability benefits to the implementation.

5.2.1 Maintainability through SOLID

One of the design goals was to prepare the library for future changes. The SOLID principles presentend in Section 3.5 were found very useful in developing a code structure that welcomes future changes. Each component, found in the component view in Figure 18, was implemented by creating one main class and a varying amount of helper classes. The number of helper classes depended on the complexity of the responsibility of a component. The Liskov principle, presented in Section 3.5.3, was used to decouple the main classes implementations from each other. By creating interfaces between the classes forced the main classes to depend on interfaces instead of implementations, which means that an implementation can easily be replaced without requireing changes in other classes. An example of how the classes were organized in the Business Object Builder component can be seen in Figure 20.

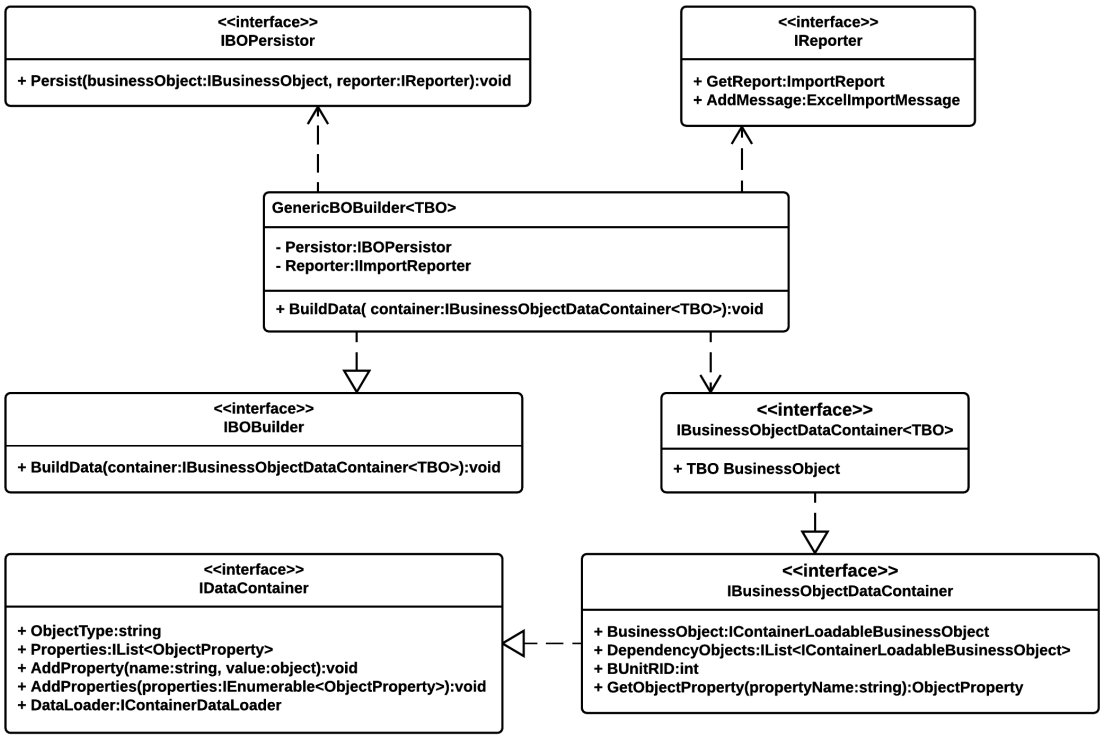

Figure 20: Class diagram of the Business Object Builder component.

The GenericBOBuilder is implemented with the help of generics so that it can handle different business objects, which means that there is not a need to implement a builder class for each business object type. The implementation that is currently is developed with one builder class per business object, which caused unecessary code duplication. The builder class implements the IBOBuilder interface, from which follows that the data collector component can depend on the IBOBuilder interface instead of depending on the implementation. As mentioned earlier in this section, the interface allows a builder implementation to be changed in the future when there is a need for it and that makes it easy to maintain. As is shown in Figure 20, the GenericBOBuilder only depends on interfaces. Because the GenericBOBuilder does not care about which implementation is used, which enables easy implementation replacement without requiring changes in the GenericBOBuilder. The rest of the implemented components follows these same principles, which makes each component easy to maintain in separation from the other components.

The private properties in the GenericBOBuilder are injected into the class through its constructor method according to the dependency inversion principle described in Section 3.5.5. Because the Persistor and the Reporter are initialized outside of the constructor method, they can be referred to as explicit dependencies. Because both of the dependencies are interfaces, it makes the class easier to test. Any persistor class that implements the IBOPersistor interface can be injected into the

GenericBOBuilder, which means that a test double can easily be used instead of a real persistor implementation. Test doubles remove unnecessary overhead and make tests focus on the code under test. Tests are also easier to write since the tests do not have to cover the functionalities of the dependencies and can instead only test the component or unit under test. When tests are easy to write, it lowers the threshold for maintainers to write a test case for the scenario that potentially caused the component to fail in production. Writing a test case to cover the failing scenario can potentially reduce the code scope to focus on for a maintainer, which should lower the required maintenance effort. The required maintenance effort correlates with the maintainability of the overall implementation, which means that by reducing the required maintenance effort causes the maintainability to increase.

The single responsibility principle, discussed in Section 3.5.1, had an apparent positive effect on the size of the classes and methods. By only allowing a class or method to have one responsibility, kept the classes and methods small. The measured lines of code for classes and methods can be seen in Table 12. Smaller classes and methods combined with semantical names significantly increased code readability. The higher readability makes the code easier to maintain.

The open-closed principle was kept in mind to some extent for the future, but several changes were made to the classes during the prototype development. It would not have made any sense to follow the closed for modification rule blindly since the none of the classes were considered as done before finishing the prototype. However, new functionality was added to the definition classes through extension methods. Extension methods in C# enable developers to add functionality to any object type without the need to modify the existing type [82]. Extension methods are called in the same way as instance methods but are implemented as static methods inside a static class. Usually, the name of an extension class begins with the name of the original type and ends with an "Extension" postfix. The static class is never visually used. It is possible to create extension methods for several different types inside the same static class, which could make sense when gathering all extension methods for e.g. a 3rd-party library into one location. However, this might potentially cause the static class to explode in size, which would reduce the maintainability and readability of the class. One potential negative side effect of using extension methods is that if the classes are poorly named, it can be difficult to find all extension methods belonging to a particular class. Also, care should be taken when adding extension methods to existing classes to avoid unwanted usage outside of the intended scope, e.g. outside of a module, namespace or even a library.

5.2.2 Maintainability through design patterns

The Factory pattern provide a convenient way of initializing classes for its clients. Because the ExcelImport components main class will most likely be initialized by injecting different objects into its constructor. By defining an interface for Factory classes, different Factory implementations can be developed for the different purposes.

The code that utilizes a Factory implementation can depend on the interface, which mean that old Factory implementations can easily be changed as discussed in the previous section about the Liskov principle. The Factory interface would belong to the library, since it should control how the Factory classes should behave. The implementations are added to the library's client modules, because the library does not care who the clients are. The Factory pattern was also used inside the library, when there was a need to initialize a component in different ways. Library maintainers can add new Factory classes in the future when there is a need initialize a class in new ways. More about the Factory pattern can be found from [53] and [49].

The Facade pattern is used in Sapphire Build as a gateway for getting e.g. database reader and writer classes for a particular business object type. There is a business object metadata class for each business object type, that provides e.g. the related reader and writer classes. Business object metadata classes were implemented, following the same rules as in Sapphire Build, for the simulated business object types created in the prototype implementation. These business object metadata classes provide the Persistor class for a business object as in Sapphire Build, but they also provide the data collector and builder classes for the particular business object type. The Facade pattern might not been used for this purpose in the library if it would have been for Sapphire Build. However, the business object metadata classes does remove the need for the library to figure out which component to use when uploading data for a particular business object type, which reduce the complexity of the library and increases the maintainability. More about the Facade pattern can be read from [53] and [49].

5.3 Evaluation based on measurements

This section presents the measurement results and evaluates the prototype based on the measured metrics. The focus of this thesis is not gain a deep insight in the maintainability attribute of the prototype, which has lead to the decision that the metrics that Visual Studio 2015 Enterprise can provide will suffice. Visual Studio 2015 Enterprise has a tool that calculates values for MI, CC, Depth of Inheritance, Class Coupling and Lines of Code. This section is divided into subsections, each covering one or two measured metrics. To be able to make more objective conclusions about the prototype, one class of the old implementation is measured and used as a comparison. It would take too long to measure all the classes of the old implementation, which is why only one is measured. The name of the measured class is UrlLinkExcelDataBuilder, which does most of the job that the data collector and builder classes do in the new design. Most of the tables that present the measured values for the prototype includes a row that shows the same metric value for the UrlLinkExcelDataBuilder as a comparison.

5.3.1 Maintainability Index

The MI reference values presented on the code metrics value page [83] of the Visual Studio pages is shown in Table 10 . The interval indicating good maintainability is remarkably large, which raise the question of why there is such a large gap between the minimum and maximum values that represent good maintainability? However, the evaluation based on the MI value provided by Visual Studio will follow the presented reference table despite the unanswered question. More knowledge about the details of these reference values would probably be needed to gain deeper insight.

Rating	MI
Good	20-100
Moderate	10-19
Bad	0-9

Table 10: Reference values for the MI metric.

The MI values are shown in Table 11 are calculated as an average of the MI values for all classes included in a particular component. According to these values, the maintainability of the prototype is relatively high and way over the lowest acceptable value for good maintainability. The MI value for the UrlLinkExcelDataBuilder is also inside the range of good maintainability, but is significantly lower than the MI values of the prototype classes. Even though the old class according to the MI value has a good maintainability rating, clear improvements have been made when implementing the prototype.

The reason for the 'all classes', which includes all of the prototype classes, MI value being much higher is because it includes all the interfaces. All of the interfaces had MI value of 100, which clearly has a significant impact on the 'all classes' MI value. Interfaces effect on the 'all classes' MI value shows that assessing maintainability based on MI should be done with caution. In Section 5.1.1, it was mentioned that MI does not demonstrate where the root-cause for a small MI value is located. However, when measuring MI with Visual Studio, it measures the MI value for all the methods in all the classes. The width of the measurement does provide some sense of direction of where potential problems might be.

Pipeline components	
ExcelReader	77
RowHandler	81
BusinessObjectDataCollector	79
GenericBOBuilder	84
All classes	92
UrlLinkExcelDataBuilder	51

Table 11: MI results.

5.3.2 Lines of Code

The minimum and maximum Lines of Code values for both methods and classes are shown in Table 12. These low values are a result of following the SOLID principles and refactoring the code several times. As a comparison, the largest method in the UrlLinkExcelDataBuilder had 60 lines of code, which is significantly larger than the largest method in the prototype. The difference in class size is almost 30 lines of code. These results indicate that the classes and methods in the prototype are more focused on a single task compared to the old implementation. More measurements would be needed to verify the case accurately, but theses results do give indications about what a more thorough measurement would look like.

Lines of Code for prototype methods	
Minimum	2
Maximum	13
UrlLinkExcelDataBuilder maximum	60
Lines of Code for prototype classes	
Minimum	5
Maximum	91
UrlLinkExcelDataBuilder	119
Prototype total	525

Table 12: Lines of Code measurement results.

An effort was made into creating semantical method names, which combined with the small sizes should make them easier to maintain. The same goes for prototype classes. The downside with having small classes and methods is that the overall class and method count increases, which might reduce maintainability according to some developers and maintainers. However, this was not the case for the prototype developer.

The total Lines of Code value for the prototype is relatively low because the prototype only handles the basic data import scenarios required by Sapphire Build. The basic scenarios were enough to prove the potential of the design. Also, to cover all of the individual cases would not have been possible in a sensible way without directly integrating the prototype into Sapphire Build. The small number does indicate a good starting point regarding the size of a future production ready data import library.

5.3.3 Cyclomatic Complexity and Class Coupling

Visual Studio's measurement tool provided values for both Cyclomatic Complexity and Class Coupling. Unfortunately, no reference values were found that would tell if the

values are good or bad. However, the values provided by the UrlLinkExcelDataBuilder measurement will be used to evaluate the measured results. Table 13 shows the minimal and maximal Cyclomatic Complexity values for the prototype classes and the value for the UrlLinkExcelDataBuilder class. The results clearly state that the prototype classes have a lower Cyclomatic Complexity, which indicates that the prototype code has a higher maintainability.

Prototype classes	
Minimum	1
Maximum	37
UrlLinkExcelDataBuilder	52

Table 13: Classes Cyclomatic Complexity measurement results.

The Class Coupling measurement results can be seen in Table 14. The difference between the prototype class with the highest coupling value and the UrlLinkExcelDataBuilder is small, but noticeable. However, the UrlLinkExcelDataBuilder is one of the simplest classes in the old implementation, so further measurements might change the coupling analysis results in favor of the prototype. At least from these measurements, a good conclusion regarding reusability can not be made. Since the Visual Studio did not provide any metric for cohesion, accurate conclusions cannot be made regarding the cohesion of the prototype classes. The Lines of Code measurement might, however, give some indications that the prototype might have a higher cohesion than the old implementation.

Prototype classes	
Minimum	2
Maximum	32
UrlLinkExcelDataBuilder	29

Table 14: Classes Cyclomatic Complexity measurement results.

5.3.4 Code coverage

Code coverage was not discussed in the section presenting maintainability metrics, but unit testing was included in Table 9. The percentages presented in Table 15 gives some sort of direction about the width of the safety net provided by the automated tests written for the prototype. Even though no goal was set for code coverage for the prototype, these values does show that the prototype is at least fairly testable. Since the total code coverage is only approximately 73%, it may only slightly increase the overall maintainability of the prototype. However, the old implementation has zero code coverage, so compared to that the prototype has at least a significantly wider safety net for refactoring tasks.

Prototype classes	
Minimum	61.36%
Maximum	100.00%
Total	72.58%
UrlLinkExcelDataBuilder	0.00%

Table 15: Code coverage provided by automated tests.

5.4 Evaluation based on research questions

The first research question, presented in Section 1.1, asks how a generic data import system should be implemented so that it can be used beyond Excel spreadsheets? The main goal of the design was to propose a solution for this question. The design consist of replaceable components, which mean that the component that reads data from an Excel spreadsheet can be replaced with a component that reads data from other sources. Instead of using an Excel spreadsheet as a data source, a Google spreadsheet could be used as a source. By implementing a component that can read data from the Google spreadsheet and gathers object related data into an object row it can be included in the data processing pipeline. By creating a Factory class that initializes an ExcelImport component and injects the new data source reader component into it, then the data import feature supports Google spreadsheets as a data source. However, the design can only give guidelines on how it should be implemented, but does not provide enough details that assures that the components are implemented as reusable components.

The second research question focus on how the data import functionality can be implemented reliably so that it can handle changing data requirements? Replacable components does provide some parts of the solution, but the maintainability aspect of the proposed design should cover the rest of the question. The centralization of metadata through the definition files makes it easier to make changes according to new requirements. The Excel template files should be generated based on the definition file, which should remove the scenarios of Excel template files being unsynced from the rest of the data import library. The business object metadata classes, which were shortly presented in the previous section, control which data collector and builder components that should be used for each business object type. Because the business object metadata classes control some of the data import metadata, it could cause potential issues that might affect how reliably the data import feature can adapt to changing data requirements.

5.5 Evaluation based on requirements

Data managers should be able to download Excel templates and upload data to Sapphire Build. The design includes a Template Generator component which provides

Sapphire Build's user interface with the Excel templates that users can download. The architecture views, presented in Section 4.4, described the pipeline architecture. The pipeline architecture enables data to be read from an Excel file and built into business objects that can be persisted by Sapphire Build, which should satisfy both data managers requirements and Sapphire Builds requirements.

Developers and maintainers requirements consisted of two parts: to be able to modify Excel templates and maintain the code base. The metadata file structure presented in Section 4.4.4 enables maintainers to modify the Excel templates, since the Excel templates are generated based on the metadata files. The proposed design can only partly provide a solution for maintainers to maintain the code base. The designed pipeline architecture does specify that the components should be loosely coupled, so that they can easily be replaced. The replacable components should make maintenace less tedious. However, as discussed in Section 5.2 the way the design is implemented can have a huge impact on how maintainable the code base is. Never the less, design patterns and the SOLID principles were used in the prototype implementation, which greatly improved the maintainability of the prototype's code base.

6 Conclusions

The purpose of this chapter is to summarize the results from the evaluation in the previous chapter and to discuss the knowledge gain from this work. The last section in this chapter discusses future work.

6.1 Results

The main purpose of this thesis was to develop a solution to the current maintainability problems in the data import feature's implementation. The design should also provide reliable means to adjust to changing data requirements. The result is a redesign of Sapphire Build's data import feature. Overall, the proposed design provides the means to extend the data import functionality to new data sources, solves the maintainability issues and enable reliable maintenance.

Because there are no objective measurements that can state whether a module has low coupling or high cohesion, no objective conclusions can be made regarding reusability. The coupling comparison between the prototype classes and the class of the current implementation did not show any improvements regarding coupling. However, as stated in the evaluation, further measurements are needed to make any accurate conclusions.

The results indicate that the way a design is implemented can have a significant impact on the maintainability of the system. The results also show that the measured maintainability metrics have improved when comparing the prototype with the old implementation.

6.2 Discussion

Design patterns and object-oriented principles are very useful tools for implementing maintainable software. By following the SOLID principles the prototype implementation consist of only minimal code duplication, and dependencies were manageable. Continuasly improving the code when possible had a positive impact both on code readability and maintainability. The Clean Code book by Martin [26], provided great guidelines on how to keep the code clean and readable. No extra comments were added to the code, but instead effort was put into crafting method names that explain what the method does.

Even though Maintainability Index only provided partial information about the maintainability of the implemented prototype, it can still be useful. By calculating the MI value both before and after changes are made, it could provide some level of information about where the maintainability of a system or component is heading. Unfortunately, the time frame of this thesis did not provide the means to pursue any further measurements to gain more objective insight regarding the system.

This thesis did not focus on automated testing, but during the development of the prototype, writing automated tests were also a great aid in developing maintainable code. If a test was hard to write, it usually provided feedback about a class being difficult or tedious to initialize or about potential dependency issues.

Design documents provide important guidelines to developers on what the solution should do and how it should look like. Software does not only include the written code, but also all the documentation about the software. Since documentation is part of a software, it also means that documentation has an impact on the maintainability factor of the software. Even though this thesis did not focus on how to maintain software documentation, it should not be neglected.

Specifying the requirements is critical for developing the right system, which means the system that is useful for its users. Maintaining the requirements specifications will most likely help developers in continuing to develop the right system. The domain model is a very useful tool to specify the concepts of the problem domain, and it can help keep all members of a development team in sync during the development stage. The domain model can also provide parts of the domain knowledge needed to maintain the system. A domain model can increase the maintainability of a software by reducing the guesswork about the meaning of the concepts. The architecture views work as important guidelines to developers during the development stage, and as useful documentation of the system for maintainers. Keeping the most important architectural views up to date also increases the maintainability of the system. If maintainers have a good overview of the system and how different parts of the system should be kept separated, it can also potentially reduce the risk of reducing cohesion in the system.

During the prototype implementation stage object-oriented programming proved again and again to be non-trivial. The knowledge gained from reading the literature referred to in Chapter 3 did make the task easier in some sense. However, the there are so many different ways software programs can be written. It requires years of experience and practice to get it right. During the implementation stage, the different feedback sources described in Section 3.2.3 proved over and over again to be very valuable for getting early warning that the implementation was heading to wrong direction. The faster the feedback is received, the faster corrective measure can be taken. The same applies to software maintenance. The faster feedback is received from the system and different stakeholders, the faster corrective measures can be made.

6.3 Future work

The proof-of-concept prototype proved that the design could solve maintainability and performance issues of the old implementation. The next step is to make small changes to the prototype so that it is possible to start integrating it to Sapphire Build, because the prototype as such can not handle all business objects out of the box.

However, since both the design and the prototype was developed with maintainability in mind it welcomes changes with open arms. After the library has been integrated into Sapphire Build and is usable for importing all of Sapphire Build's business object types, other data sources than Excel spreadsheets can be looked into. One purpose for redesigning the data import functionality was to be able to support new data sources. The Excel Reader can be switched out to a component capable of reading other tabular data sources, so the task would be to implement new data reader components. For the library to be able to import data from other than tabular data sources, e.g. a REST API, the usage of data rows would need to be changed. The metadata files should, however, provide the necessary information even though it includes extra data that most likely would not be needed when used with a REST API.

Since no clear objective conclusions could be made regarding reusability, other measuring tools besides the tools provided by Visual Studio could be tried in the future. Also, either a literature study or a research study could be conducted to find out if there are any correlation between the amount of extension points in a class or module and its reusability. Extension points can facilitate reuse of at least some parts of the code in a base class through inheritance, which will be the case when the integration process is started.

This thesis focused on only on the data import functionality of Sapphire Build. However, Sapphire Build does also include a data export feature. The data export feature also suffers from maintainability problems which should be dealt with. The redesigned data import library can be used as a basis for redesigning the data export feature. The metadata files could be extended to fully support data export, since most of the existing data stored in the metadata files are usable for data export purposes.

Git was used as the version control system for the prototype, which provided versioning for the source code. Git could potentially be used as a tool for documenting implementation decisions. Instead of writing a comment describing a decision in the source code, it could be documented into a Git commit. Using Git commits to document implementation decision is something that could be looked into in the future, since it could potentially provide a easy way for keeping track of implementation decisions that could be beneficial for maintainers in the future.

References

[1] *First-Time Homebuyers*. http://archives.hud.gov/offices/hsg/sfh/ref/sfhp3-02.cfm. Accessed: 2016-04-19.

[2] Barnes Peter. *Home, sweet home: Your guide to the homebuilding industry*. http://marketrealist.com/2015/02/home-sweet-home-homebuilding-industry-shapes-economy. Accessed: 2016-04-19.

[3] Petri Hyväri. "3D Visualization of Configured Homes". MA thesis. Department of Computer Science and Engineering, Helsinki Univeristy of Technology, 2007.

[4] Craven Jackie. *What Is a Production Home Builder?* http://architecture.about.com/cs/buildyourhouse/g/production.htm. Accessed: 2016-04-20.

[5] *Manufactured Homes*. http://www.championhomes.com/home-plans-and-photos/manufactured--homes. Accessed: 2016-04-20.

[6] *Core Principles*. https://www.excelhomes.com/excel-homes/core-principles. Accessed: 2016-04-20.

[7] *FACTORY AND SITE-BUILT HOUSING A COMPARISON FOR THE 21ST CENTURY*. https://www.huduser.gov/portal//Publications/pdf/factory.pdf. Accessed: 2016-04-20.

[8] *Sapphire Build Overview*. http://www.kovasolutions.com/ProductModules/SapphireBuildOverview.aspx. Accessed: 2016-04-21.

[9] Mikko Halttunen. "Centralized management of a distributed PDM portal for home builders". MA thesis. Department of Computer Science, Engineering, Aalto University School of Science, and Technology, 2010.

[10] David Thomas and Andrew Hunt. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, Oct. 1999. ISBN: 020161622X.

[11] Magnus Christenson, Ivar Jacobson, and Larry L. Constantine. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley, July 1992. ISBN: 0-201-54435-0.

[12] Michael Tortorella. *Reliability, Maintainability, and Supportability: Best Practices for Systems Engineers*. John Wiley & Sons, Mar. 2015. ISBN: 9781118858882.

[13] Penny Grubb and Armstrong A Takang. *SOFTWARE MAINTENANCE Concepts and Practice*. World Scientific Publishing Co, 2003. ISBN: 981-238-425-1.

[14] Priyadarshi Tripathy and Kshirasagar Naik. *Software Evolution and Maintenance*. JOhn Wiley & Sons, Nov. 2014. ISBN: 9780470603413.

[15] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)– System and software quality models*. Standard. International Organization for Standardization/International Electrotechnical Commission, Mar. 2011.

[16] *What is the purpose of Software?* https://davidlongstreet.wordpress.com/2010/04/30/what-is-the-purpose-of-software/. Accessed: 2016-04-08.

[17] E. Burton Swanson and Enrique Dans. "System life expectancy and the maintenance effort: exploring their equilibration". In: *MIS Quarterly* 24.2 (June 2000), pp. 277–297. URL: http://profesores.ie.edu/enrique_dans/download/misq.pdf.

[18] Jason McColm Smith. *Elemental Design Patterns*. Addison-Wesley Professional, Mar. 2012. ISBN: 9780321711922.

[19] *About WordPress*. https://wordpress.org/about/. Accessed: 2016-03-31.

[20] Rajiv D. Banker et al. "Software Complexity and Maintenance Costs". In: *Commun. ACM* 36.11 (Nov. 1993), pp. 81–94. ISSN: 0001-0782. DOI: 10.1145/163359.163375. URL: http://doi.acm.org/10.1145/163359.163375.

[21] Bente Anda. "Assessing Software System Maintainability using Structural Measures amd Expert Assessments". In: *Software Maintenance, 2007. ICSM 2007* (2007).

[22] Michal Král Jaroslav and Žemlička. "Computational Science and Its Applications – ICCSA 2014: 14th International Conference, Guimarães, Portugal, June 30 – July 3, 2014, Proceedings, Part V". In: ed. by Beniamino Murgante et al. Cham: Springer International Publishing, 2014. Chap. Simplifying Maintenance by Application of Architectural Services, pp. 476–491. ISBN: 978-3-319-09156-3. DOI: 10.1007/978-3-319-09156-3_34. URL: http://dx.doi.org/10.1007/978-3-319-09156-3_34.

[23] *How to develop unmaintainable software*. http://typicalprogrammer.com/how-to-develop-unmaintainable-software/. Accessed: 2016-03-31.

[24] Alain April and Alain Abran. *Software Maintenance Management: Evaluation and Continuous Improvement*. John Wiley & Sons, Apr. 2012. ISBN: 978-0-470-25802-6.

[25] *Things You Should Never Do, Part I*. http://www.joelonsoftware.com/articles/fog0000000069.html. Accessed: 2016-04-09.

[26] Robert C. Martin. *Clean code*. Prentice Hall, Aug. 2008. ISBN: 9780136083238.

[27] Tom Poppendieck and Mary Poppendieck. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional, Sept. 2006. ISBN: 9780321437389.

[28] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, Sept. 2004. ISBN: 0131177052.

[29] Tom Poppendieck and Mary Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, May 2003. ISBN: 9780321150783.

[30] Dan Olsen. *The Lean Product Playbook: How to Innovate with Minimum Viable Products and Rapid Customer Feedback*. John Wiley & Sons, June 2015. ISBN: 9781118960875.

[31] *The Long, Dismal History of Software Project Failure*. http://blog.codinghorror.com/the-long-dismal-history-of-software-project-failure. Accessed: 2016-04-11.

[32] Steve McConnell. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, July 1996. ISBN: 9780735634725.

[33] Kent Beck. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley Professional, Nov. 2004. ISBN: 9780321278654.

[34] Philipp K. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., Oct. 2013. ISBN: 9781449361693.

[35] Ron Patton. *Software Testing, Second Edition*. Sams, July 2005. ISBN: 9780672327988.

[36] Robert C. Martin. *The Clean Coder*. Prentice Hall, May 2011. ISBN: 9780132542913.

[37] Ira D. Baxter. "Design Maintenance Systems". In: *Commun. ACM* 35.4 (Apr. 1992), pp. 73–89. ISSN: 0001-0782. DOI: 10.1145/129852.129859. URL: http://doi.acm.org/10.1145/129852.129859.

[38] Nat Pryce and Steve Freeman. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, Oct. 2009. ISBN: 9780321503626.

[39] J. Wloka, B. G. Ryder, and F. Tip. "JUnitMX - A change-aware unit testing tool". In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. May 2009, pp. 567–570. DOI: 10.1109/ICSE.2009.5070557.

[40] F. Del Frate et al. "On the correlation between code coverage and software reliability". In: *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. Oct. 1995, pp. 124–132. DOI: 10.1109/ISSRE.1995.497650.

[41] P. Garg. "Investigating coverage-reliability relationship and sensitivity of reliability to errors in the operational profile". In: *Software Testing, Reliability and Quality Assurance, 1994. Conference Proceedings., First International Conference on*. Dec. 1994, pp. 21–35. DOI: 10.1109/STRQA.1994.526380.

[42] Mei-Hwa Chen, M. R. Lyu, and W. E. Wong. "An empirical study of the correlation between code coverage and reliability estimation". In: *Software Metrics Symposium, 1996., Proceedings of the 3rd International*. Mar. 1996, pp. 133–141. DOI: 10.1109/METRIC.1996.492450.

[43] Andrew Glover, Steve Matyas, and Paul M. Duvall. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, June 2007. ISBN: 9780321630148.

[44] Alberto Bacchelli and Christian Bird. "Expectations, Outcomes, and Challenges of Modern Code Review". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 712–721. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486882.

[45] Luca Milanesio. *Learning Gerrit Code Review*. Packt Publishing, Sept. 2013. ISBN: 9781783289479.

[46] Sebastian Bergman and Stefan Priebsch. *Real-World Solutions for Developing High-Quality PHP Frameworks and Applications*. John Wiley & Sons, Apr. 2011. ISBN: 9780470872499.

[47] Sandro Mancuso. *The Software Craftsman: Professionalism, Pragmatism, Pride*. Prentice Hall, Dec. 2014. ISBN: 978-0-13-405250-2.

[48] Don Roberts et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999. ISBN: 9780201485677.

[49] Robert C. Martin and Micah Martin. *Agile Principle, Patterns, and Practices in C#*. Prentice Hall, July 2006. ISBN: 9780131857254.

[50] Bart De Win et al. "On the importance of the separation-of-concerns principle in secure software engineering". In: (2002).

[51] Harold Ossher and Peri Tarr. "Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software". In: *Commun. ACM* 44.10 (Oct. 2001), pp. 43–50. ISSN: 0001-0782. DOI: 10.1145/383845.383856. URL: http://doi.acm.org/10.1145/383845.383856.

[52] D. Kung et al. "Change Impact Identification in Object Oriented Software Maintenance". In: *Software Maintenance, 1994. Proceedings., International Conference on* (1994).

[53] Erich Gamma et al. *Design Patterns: Element of Reusable Object-Oriented Software*. Addison-Wesley Professional, Oct. 1994. ISBN: 0201633612.

[54] Alan Shalloway and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition*. Addison-Wesley Professional, Oct. 2004. ISBN: 9780321247148.

[55] *Books*. http://handbookofsoftwarearchitecture.com/?genres=patterns&orderby=title&order=asc. Accessed: 2016-04-05.

[56] *Design Patterns*. http://www.oodesign.com/. Accessed: 2016-04-05.

[57] *Design Patterns*. https://sourcemaking.com/design_patterns. Accessed: 2016-04-05.

[58] Gary McLean Hall. *Adaptive Code via C#: Agile coding with design patterns and SOLID principles*. Microsoft Press, Oct. 2014. ISBN: 9780133979749.

[59] James W. Grenning. *Test Driven Development for Embedded C*. Pragmatic Bookshelf, Apr. 2011. ISBN: 9781934356623.

[60] Meyer Bertrand. *Object-oriented software construction*. Prentice Hall, 1988. ISBN: 0-13-629031-0.

[61] *virtual (C# Reference)*. https://msdn.microsoft.com/en-us/library/9fkccyh4.aspx. Accessed: 2015-08-28.

[62] Barbara Liskov. "Data Abstraction and Hierarchy". In: *OOPSLA '87 Addendum to the proceedings on Object-oriented programming systems, languages and applications* (1987).

[63] *new Modifier (C# Reference)*. https://msdn.microsoft.com/en-us/library/435f1dw2.aspx. Accessed: 2015-08-28.

[64] "ISO/IEC/IEEE Systems and software engineering – Architecture description". In: *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (Dec. 2011), pp. 1–46. DOI: 10.1109/IEEESTD.2011.6129467.

[65] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition*. Prentice Hall, July 2001. ISBN: 9780130925695.

[66] Bobby Woolf and Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, Oct. 2003. ISBN: 0321200683.

[67] *Pipes and Filters Pattern*. https://msdn.microsoft.com/en-us/library/dn568100.aspx. Accessed: 2016-05-05.

[68] *JSON: The Fat-Free Alternative to XML*. http://www.json.org/xml.html. Accessed: 2016-04-22.

[69] Sai Srinivas Sriparasa. *JavaScript and JSON Essentials*. Packt Publishing, Oct. 2013. ISBN: 9781783286034.

[70] *The home of JSON Schema*. http://json-schema.org. Accessed: 2016-04-22.

[71] Vasudeva Varma. *Software Architecture: A Case Based Approach*. Pearson Education India, Mar. 2009. ISBN: 9788131707494.

[72] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics A Rigorous & Practical Approach, Second Edition*. PWS Publishing Company, 1997. ISBN: 053495425-1.

[73] Caper Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill, Apr. 2008. ISBN: 9780071502443.

[74] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. "A Systematic Review of Software Maintainability Prediction and Metrics". In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 367–377. ISBN: 978-1-4244-4842-5. DOI: 10.1109/ESEM.2009.5314233. URL: http://dx.doi.org/10.1109/ESEM.2009.5314233.

[75] I. Heitlager, T. Kuipers, and J. Visser. "A Practical Model for Measuring Maintainability". In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. Sept. 2007, pp. 30–39. DOI: 10.1109/QUATIC.2007.8.

[76] Maurice H. Halstead. *Elements of Software Science*. Vol. 7. Elsevier, 1977.

[77] M. J. P. v. d. Meulen and M. A. Revilla. "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs". In: *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*. Nov. 2007, pp. 203–208. DOI: 10.1109/ISSRE.2007.12.

[78] D. Coleman et al. "Using metrics to evaluate software system maintainability". In: *Computer* 27.8 (Aug. 1994), pp. 44–49. ISSN: 0018-9162. DOI: 10.1109/2.303623.

[79] J. S. Poulin. "Measuring software reusability". In: *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*. Nov. 1994, pp. 126–138. DOI: 10.1109/ICSR.1994.365803.

[80] G. Gui and P. D. Scott. "Coupling and Cohesion Measures for Evaluation of Component Reusability". In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR '06. Shanghai, China: ACM, 2006, pp. 18–21. ISBN: 1-59593-397-2. DOI: 10.1145/1137983.1137989. URL: http://doi.acm.org/10.1145/1137983.1137989.

[81] G. Gui and P. D. Scott. "New Coupling and Cohesion Metrics for Evaluation of Software Component Reusability". In: *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*. Nov. 2008, pp. 1181–1186. DOI: 10.1109/ICYCS.2008.270.

[82] *Extension Methods (C# Programming Guide)*. https://msdn.microsoft.com/en-us/library/bb383977(v=vs.140).aspx. Accessed: 2015-08-26.

[83] *Code Metrics Values*. https://msdn.microsoft.com/en-us/library/bb385914.aspx. Accessed: 2016-05-17.