

Lean BIM-based communication and workflow during design phases

Lean BIM-basierte Kommunikation und Workflow in der Designphase

Marija Rakić

Master's Thesis in Informatics

Supervisor:

Prof. Dr.-Ing. Frank Petzold

Advisors:

Ata Zahedi, M.Sc.

Jimmy Abualdenien, M.Sc.

Submission Date:

15. 6. 2019



TUM Uhrenturm

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15. 6. 2019

Honest Student

Abstract

The final design of a building is a result of the joint efforts between many experts within the Architecture, Engineering and Construction (AEC) industry. Building Information Modeling (BIM) aims to streamline this collaboration while allowing all parties to access the relevant data. For example, if an engineer cannot perform analysis due to a missing value, he/she must wait for the responsible architect to specify it.

Different values get defined in different project stages. Level of Development (LOD) describes maturity of the BIM model. BIMForum defines six LOD levels and specifications of which information each level provides. However, it does not take into consideration the information fuzziness. Therefore, a research project EarlyBIM has developed a multi-LOD meta-model that “explicitly describes the LOD requirements of each individual building component type taking into consideration the possible uncertainties.” [Abualdenien and Borrman, 2019]

Due to the fragmented nature of AEC industry, there is great potential for a machine-readable communication protocol. BIM Collaboration Format is mostly used based on the textual comments, which the architect in the aforementioned example would be able to act upon. However, a computer could not understand this comment and analyze it for future references. With this in mind, our research group designed an adaptive minimized communication protocol and a ticketing system, in which only the computer-readable information on existing issues is exchanged. [Zahedi and Petzold, 2018]

This paper presents a plugin for Autodesk Revit using the principles of the aforesaid communication system and multi-LOD meta-model. The plugin compares a model to the requirements specified in the multi-LOD meta-model, visualizes all the missing values, and offers a way of integrating them into the building model. The changes to the model are logged and can be viewed using an external tool, such as a custom database combined with a web interface.

Contents

Abstract	i
Abbreviations	v
1 Introduction	1
1.1 An overview	1
1.2 Theoretical background	1
1.2.1 Building Information Modeling - BIM	1
1.2.2 Industry Foundation Classes - IFC	3
1.3 Literature review	5
1.3.1 One problems	5
1.3.2 Two suggested solutions	5
1.4 Objectives of the Thesis	6
1.5 Structure of the Thesis	6
2 Problem & Solution	7
2.1 Problem	7
2.2 Solution	7
3 Existing system	9
4 Implementation	11
4.1 An introduction to developing a Windows application	11
4.2 Framework and libraries used to develop the plug-in	12
4.3 Revit plug-in ecosystem	15
4.4 System setup	17
4.5 Graphical User Interface	18
4.5.1 Starting the plug-in for the first time	18
4.5.2 Home page for an architect	18
4.5.3 An architect requesting an analysis	19
4.5.4 Home page for an engineer	19
4.5.5 Analysis results	20
4.5.6 Creating a missing parameter	22
4.5.7 Data type of a missing value	23
4.5.8 Creating a missing value	25
4.5.9 Filtering tasks by type	26
4.5.10 Suggesting a missing value	27
4.5.11 Pending tasks	28
4.5.12 Solved tasks	30
4.5.13 Newly created value under Revit Properties	31
4.5.14 Tree view	32
4.5.15 Selecting an element	33
4.5.16 Charts	34
4.5.17 Changing the user	35
4.5.18 Successfully running the analysis	36

4.6	Communication within the system	37
4.7	Implementation	40
4.7.1	MarijaRakicMasterThesis	41
4.7.2	IExternalApplication	41
4.7.3	IExternalCommand	43
4.7.4	Singleton	44
4.7.5	Views	44
4.7.6	User Controls	47
4.7.7	Validation rules	50
4.7.8	ViewModels	51
4.7.9	Commands	55
4.7.10	ExternalEventHandler	56
4.7.11	Managers	59
4.7.12	Utils	60
4.7.13	MultiLODLib	62
4.7.14	MarijaRakicMasterThesis.DataTypes	66
4.7.15	MarijaRakicMasterThesis.Converters	70
4.7.16	com.server.api	71
5	Summary	75
5.1	Future work	75
	Bibliography	81

Abbreviations

2D - Two Dimensional

3D - Three Dimensional

AEC - Architecture, Engineering and Construction

BIM - Building Information Modeling

CAD - Computer-Aided Design

IFC - Industry Foundation Classes

WPF - Windows Presentation Foundation

XAML - Extensible Application Markup Language

UI - User Interface

JSON - JavaScript Object Notation

HTTP - Hypertext Transfer Protocol

XML - eXtensible Markup Language

HTTP - HyperText Transfer Protocol

1 Introduction

This chapter is organized as follows: first, an overview of the current state is given. It is followed by a short theoretical background on Building Information Modeling and Industry Foundation Classes. Next, a related work is discussed. Finally, the objectives of the thesis are presented.

1.1 An overview

To create a final version of a building model, many different experts from the Architecture, Engineering and Construction (AEC) industry have to work together. The model has to go through many iterations of feedback, testing and improvement in order to reach a stage where it is ready for construction. For these iterations to happen, data needs to be passed from one stakeholder to another, for example from an architect to a structural engineer. Unfortunately, this exchange is not always a seamless process. Architects have their own set of software tools they use; engineers of different specialization have different software applications for their field-specific analysis. Not less important are clients that have to maintain the building after the design and construction phases are done, since the maintenance is the longest part of the building's life cycle. The data exchanged needs to be understood by the side receiving it. In other words, a person who receives it should be able to import it into the tools they use. Prior to existence of a standard for exchange format and its wide use, a lot of data exchanged needed to be re-input manually. Just imagine an architect sending data, which an engineer needs to input manually before being able to run analysis, and then the engineer returning the results back to the architect who has to input them back to the software they use. With every exchange cycle, the probability that someone input incorrect data increases. After a significant number of iterations, the model is finalized, the construction happens, and then the model is handed over to the building owner, who has to manually input parameters into the maintenance system. If a problem occurs somewhere along the way and it is discovered late in the design phase or during the construction phase, it might be very expensive to fix and the estimated project duration might be exceeded.

Unlike the automotive industry, where processes are almost entirely digitalized and without a need for human interaction, the AEC industry is still highly dependent on human participation. In the AEC industry there is a huge number of small companies; in comparison, the automotive industry has a small number of big players. Due to its fragmented nature, no member is powerful enough to impose a workflow or software tools that everyone would have to use. Building Information Modeling (BIM) addresses this need for a unified workflow process that enhances the aforesaid collaboration in the AEC industry and specifies the use of a common exchange format - Industry Foundation Class (IFC). The next section gives a definition of those two and other related terms.

1.2 Theoretical background

This section explains a Building Information Modeling (BIM) and an Industry Foundation Classes (IFC) in more detail.

1.2.1 Building Information Modeling - BIM

Computers, their accessibility and growing performance speed up development in many fields, including the Architecture, Engineering and Construction (AEC) industry. Some ideas existed before, some appeared after computers, and some, like Building Information Modeling (BIM), originated and evolved in

parallel to the advancement of computers. While starting as an influential vision in 1962 [Engelbart, 2001], it was not until the 1970s that BIM as a process came to life and only in the 1980s, when technology could support its implementation, did it appear as a software package. However, at this time computers were not powerful enough for BIM to reach its full potential, and so it has taken until recent decades for it to become widely known.

Building Information Modeling (BIM) is a 3D model-based process. Since the final design of a building is a result of the joint efforts between many experts within the AEC industry, it is necessary to streamline this collaboration and ensure a smooth handover to the client at the end. This culminates in a model that can be used even for maintenance and renovation purposes after the construction phase has finished. Among other goals, BIM aims to save time and money by allowing for easy collaboration between different specialties as well as for a fast and early exchange of information. This increases productivity and decreases risks that can appear by manual re-entering of the data created in the previous stage, and shifts design decisions to an earlier stage of the project, at which it is cheaper to make adjustments than at later stages.

'B' in BIM represents not just buildings as objects, but everything that is the outcome of a building process. 'I' stands for information that is transported throughout the entire building life cycle, starting with a design and continuing through the construction phase, yet not ending there, but also lasting during the period of the building operation. BIM is so powerful due to the exchange of information in quick iterations. In the past, models were simply passed from one stage to another, from one stakeholder to another. In BIM, information is exchanged frequently, allowing all parties to have the data relevant to them, view it, suggest improvements, and spot possible problems early on, potentially saving a lot of money and time that could be wasted if the project went wrong.

It's not always easy to adopt something new, especially a big and complex workflow like BIM. Due to a steep learning curve needed, four levels of BIM have been defined (0, 1, 2 and 3) to describe the progress of BIM implementation. Level 0 refers to the 2D CAD models, when the exchange of the models is purely paper-based. Level 1 refers to the 3D models, when the exchange takes the form of sending and receiving individual digital files. Level 2 is where the AEC industry is at the moment. Everyone is still working on their own model, but the data is exchanged through the Common Data Environment (CDE). Level 3 is the full implementation of the BIM process for which there is one central model throughout the process.

BIM levels refer to a maturity of a BIM process and are not to be confused with BIM Levels of Development (LOD) or BIM dimensions, which both refer to BIM models.

Initially the abbreviation LOD referred to Level of Detail, a term which was not precise enough. Level of Development includes both information on the geometry of a model, usually called Level of Geometry (LOG) or Level of Detail, and information on the semantics of a model, called Level of Information (LOI). BIMForum, the US chapter of buildingSMART International, has defined six LOD levels and specifications of which information each level has to provide.

BIM models can be viewed in dimensions ranging from 2D to 6D. 2D and 3D provide the well-known two and three dimensions, respectively. Since a BIM model is more than just a geometrical representation of a building, time information can be added to a 3D model, which is then considered to be a 4D model. Next, a 5D adds cost information. Finally, 6D refers to the facility management.

The adoption rate of BIM is definitely growing, yet the question is whether it is faster, slower or as expected. Fragmentation of the AEC industry hinders growth. Experts from several disciplines use various modeling or analysis tools that produce output in different formats. If the model is created and exported in a closed format, it forces all the other stakeholders to adjust to that and use the same or compatible software. Therefore governments, as the largest clients, are trying to pull the industry in the direction of change, hoping that the industry will, in return, push the acceptance and utilization of the system [Walasek and Barszcz, 2017]. Some countries, like the United States, Singapore, South Korea, the United Kingdom and Scandinavian countries [Borrmann et al., 2015], are enforcing the use of BIM on all government projects. This means that all project and asset information, documentation and data need to exist in digital form. The United Kingdom is an example of a country that has systematically planned and imposed the use of BIM starting from 2016, announcing it in 2011, thus giving a 5 year period to prepare for its implementation [GOV.UK, 2011]. As a result, the BIM adoption rate has risen significantly in the past 8

years [Royal Institute of British Architects, 2018]. The percentage of companies that are “aware [of] and currently using” BIM has gone from only 13% in 2011 to 74% in 2018. At the same time, the percentage of “just aware” companies has dropped from 45% in 2011 to 25% in 2018, meaning that 99% of the companies are at least aware of BIM.

A distinction between openBIM and nativeBIM as two BIM workflows is made depending on the format of the exchange data. OpenBIM refers to using BIM with open standards. NativeBIM is not in contrast to, but rather closely related to openBIM. A process starts in a nativeBIM way, by creating a blueprint using proprietary software, and then exporting and sending it in an openBIM manner. A well-known format used to exchange models between various applications is Industry Foundation Classes (IFC), which is presented in the next section.

BIM Collaboration Format - BCF

BIM Collaboration Format (BCF) is an open standard for reporting problems or proposing a design change and tracking the progress. It separates communication from the model: only lite messages are exchanged, while the model is hosted on a Common Data Environment (CDE). Every issue has a unique identifier, information on a faulty unit or missing data, and eventually an assignee responsible for resolving it. Without using BCF, a person creating the report would take a screenshot of the problematic element, write a comment and share it with other stakeholders. When using BCF, upon opening the existing issue using the usual software tool, a specialist is presented with the error itself, without a need to look for it in the entire model.

In a BCF-based communication, parties in the AEC industry exchange an XML-like file containing information on a model flaws. This formatted file is edited after a flaw has been taken care of, and sent back. Since the file is potentially exchanged between many field experts at the same time, it can happen that older versions of the same file are passed around. This could be avoided if BCF was implemented as a cloud solution. In the first version created by Solibri, Inc.¹ and Tekla Corporation² in 2009, BCF consisted of topics that were connected to a model element using the element’s globally unique identifier (GUID), as well as view point and optionally screenshot. buildingSMART took over and the second version appeared in 2014, introducing BIM-Snippets, machine-readable topics. However, they are still not commonly used.

1.2.2 Industry Foundation Classes - IFC

Industry Foundation Classes (IFC) is a data model and a file format developed in order to ease interoperability in the Architecture, Engineering and Construction (AEC) industry. It was created by buildingSMART, an international association of companies sharing the same goal of improving the open exchange of data in the BIM workflow. IFC is vendor independent and defined by ISO specification. It firstly appeared in 1994 and has had multiple iterations since. The current version IFC4 is from 2016, although the previous version IFC2.3 is still widely spread. The file schema is defined in the header of the IFC document, as well as name, description and other information. IFC comes in three different formats: as a text (STEP) file, which is the most widely used, as an XML file (used less due to its large size), and as a ZIP file.

One can think of IFC as a PDF of the AEC industry [BIMconnect, 2017]. It ensures that everyone can view data being exchanged, regardless of which software was used to create a model.

Most of the time, a single stakeholder is not interested in the full IFC model. Also, exchanging an entire model would be too expensive, since all the tools importing it would need to implement several views of an IFC geometry representation, even when they are not needed for an expert analysis. Specialists care only about the part of the model that they need for their work. For example, an energy professional is interested only in part of IFC data needed to determine the level of energy efficiency of the building. For this reason, buildingSMART created Model View Definition (MVD), which defines various subsets of IFC model for different purposes.

¹<https://www.solibri.com>

²<https://www.tekla.com>

An example of a MVD model with a purpose to be handing over to a client after the construction is finished is called Construction Operations Building Information Exchange (COBie). It contains information needed for the building maintenance. It does not contain any graphical information, and can be viewed in a form of a spreadsheet, meaning no additional software is required, other than the one everyone is likely to already have. It is not yet widespread.

1.3 Literature review

In this section, I firstly review one paper that stresses out the importance of the early phases of the building design. After that, I go over two suggested solutions to the problem of insufficient information in those early phases.

1.3.1 One problems

The importance of the early phase: the case of construction and building projects

[Kolltveit and Grønhaug, 2004] examined different aspects of a project's early phase and their implications on the project's success. They started from a small number of research on the early phase of the project and its great importance on the project's overall performance, and wanted to know which aspects of this early phase are especially important. They presented results from a large-scale project, during which they conducted a large-scale research project focused on the early phase of the project. During the research, additional education was provided to the project stakeholders; the authors collected meeting notes, and conducted a series of interviews and questionnaires. The authors identified two aspects that have a major influence on the project's overall performance: uncertainty and the influence of project stakeholders. Uncertainty is the highest in the early stages of the project. At the same time, that is the time when the technical concept is developed. The authors pointed out that the influence of stakeholders is highest in the early phase, since the cost of introducing a change is lower than at any following stage. According to the results, stakeholders are aware of this, yet they do not seem to leverage it. Kolltveit and Grønhaug explained this by the claim that the construction and building industry is very conservative and slow in accepting changes.

1.3.2 Two suggested solutions

multi-lod meta-model

[Abualdenien and Borrmann, 2019] argue that LOD, explained in Section 1.2.1, does not provide enough information. For example, some building elements must be more detailed than others from the beginning of the project. A single LOD value that would describe the entire building model is not appropriate in this case. For this reason, the authors proposed a multi-LOD meta-model which specifies LOD, as defined by BIMForum, on a component type base and takes into consideration the information uncertainty. By doing so, they tried to address the following questions: how to have more than one LOD per building model, how to take into consideration fuzziness of the data, and how to manage different design variants, all with a special focus on the meta-model's influence on the design decisions made in the project's early phases. The multi-LOD meta-model introduces an instance and data-model level. For a single component type, multi-LOD meta-model separates geometrical and semantic requirements.

The authors utilized some well-known standardized concepts, such as IFC, explained in Section 1.2.2, its *PropertySet* mechanism which allows for a dynamic extension of the model, and *buildingSmart Data Dictionary*, which guarantees that everyone, regardless of the branch of the AEC industry they come from or the language they speak, uses the same term for the same IFC entity, therefore resulting in a smooth communication.

Next, Abualdenien and Borrmann evaluated the meta-model. For this purpose, the multi-LOD meta-model on the data-model level was implemented as a web service. They took a *Wall* as an example of a building component type and specified its multi-LOD meta-model requirements on multiple LOD levels.

Adaptive minimized communication protocol

[Zahedi and Petzold, 2018] proposed an *Adaptive minimized communication protocol*. This protocol would be a computer-readable set of messages that architects and engineers exchange during the process of assuring that the building model is efficient and complies to the regulations. The authors also tackled the

fact that it is difficult to make decisions in the early stages of the project due to insufficient information. They acknowledged the work of [Abualdenien and Borrmann, 2019] and the multi-LOD meta-model. Zahedi and Petzold proposed a *Feedback* function, which would be exchanged between an architect and an engineer communicating the building model flaws. This function accepts multiple input parameters, making it adaptive to different scenarios. The parameters are *actionType*, which specifies an action that should be taken with respect to the model and its existing flaw, *optionGroupID*, allowing for multiple suggestion values to be grouped together, *GUID*, uniquely identifying a building component, *aLODx*, *objectID*, *propertyID*, all referring to the multi-LOD meta-model, *value*, and *rating*. No building model or its parts would have to be exchanged, only small messages - therefore the minimized protocol.

1.4 Objectives of the Thesis

As part of my master thesis, I implemented an Autodesk Revit plugin, a showcase of how the communication protocol and the multi-LOD meta-model described in the previous section can be integrated into an application that is widely used in the AEC industry. The plugin is intended to be used by architect and engineers to assign tasks, which specify a building model's missing data.

1.5 Structure of the Thesis

This thesis is organized as follows:

Chapter 2 brings in the problem that we believe exists in the AEC industry nowadays and offers a potential solution.

Chapter 3 shortly presents the existing web service and a sequential database that my plug-in communicates with.

Chapter 4 introduces a reader to some basic programming concepts, with a focus on developing a Windows application. Next, a Revit plug-in ecosystem is presented. A system set-up is explained, followed by an explanation of the user interface and the features that the plug-in provides. Lastly, this chapter takes a deep dive into the implementation details.

Chapter 5 gives a summary of the thesis, as well as some ideas for the future improvements of the application.

2 Problem & Solution

2.1 Problem

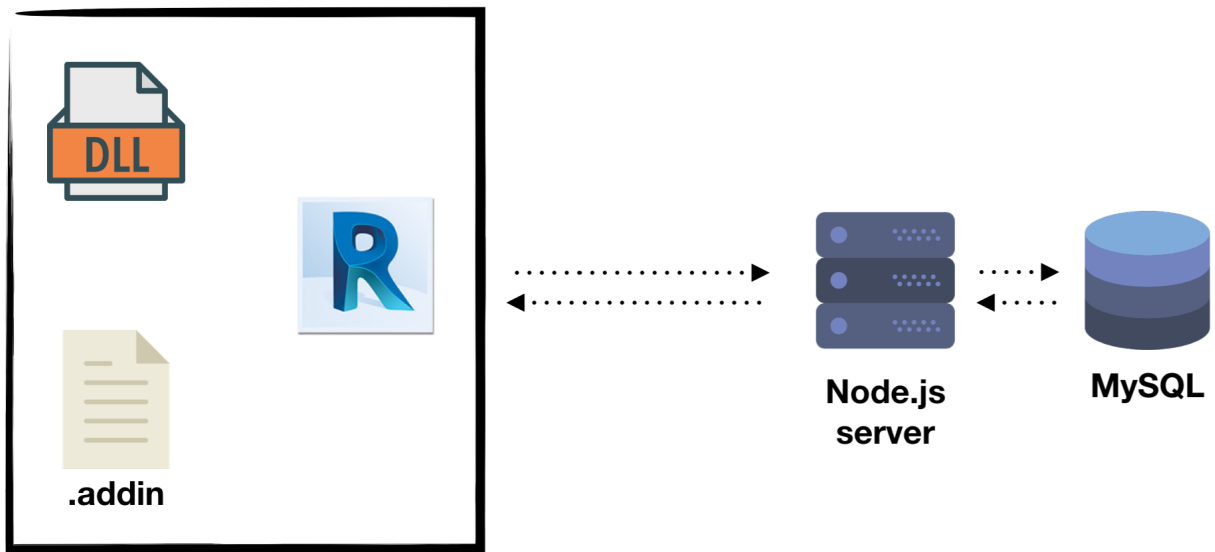
Let's take a close look at the communication that takes place in the Architecture, Engineering and Construction (AEC) industry. An architect starts by creating a building model using a BIM authoring tool, for example Autodesk Revit. After some time, the architect needs the feedback from an expert from a certain field, for example from a structural engineer. The architect sends the model to the engineer. The engineer receives a model and tries to perform the field-specific analysis.

A couple of different scenarios can occur at this point. Firstly, the engineer might not be able to run the analysis at all, due to the missing value. In this case, the engineer would have to report a missing value back to the architect and wait for them to input this value, before being able to run the analysis successfully. Secondly, some of the values might not be specific/precise enough. In this case, the model might appear finished, even though the values are not precise and might differentiate in the final version of the model. In both of these cases, the engineer needs to communicate those model flaws back to the architect who requested the analysis. At this point, a question arises of the analysis report format. Prior to existence of the standard communication protocols, an engineer would write down all the flaws in an unformatted report, maybe take some screen shots of the related building elements and attach them to the report. Then, the architect would have to manually track down each reported flaw and fix it. There would not be an option to automatically generate reports or track down the fixed and the remaining tasks. Another approach to this problem is using BIM Collaboration Format (BCF). BCF is based on the textual comments, which are human readable (the architect in the aforementioned example would be able to act upon), but not computer readable.

Multiple exchanges of these requests and reports are likely to happen between an architect and a single field specialist. Eventually, after all the required changes have been applied, the engineer runs the analysis successfully and reports positive results back to the architect. After that has been finished, the architect may need an estimate from a specialist in a different field. The aforementioned communication then repeats as many times as there are different field specialists required for the project.

2.2 Solution

We wanted to provide means which would provide this communication. I built a Revit plug-in which facilitates this repeated request/reply exchange between an architect and an engineer. There is a need for a server and a database, and so I used an existing Node server and an SQL database, which are explained in Section 3. The overall system architecture is as follows:



Designed by Smashicons and Swifticons from Flaticon

Figure 2.1 System architecture

A client application consists of a Revit application, a file with an .addin extension, further explained in Section 4.3, and a file with a .dll extension (a plug-in file), whose development is further explained in Section 4.7, as seen on the left side of Figure 2.1.

3 Existing system

Prior to the development of this application, a MySQL database and a simple web interface and HTTP server have been developed. I used this database and server to permanently persist data used by my application.

Schema of the database is as follows:

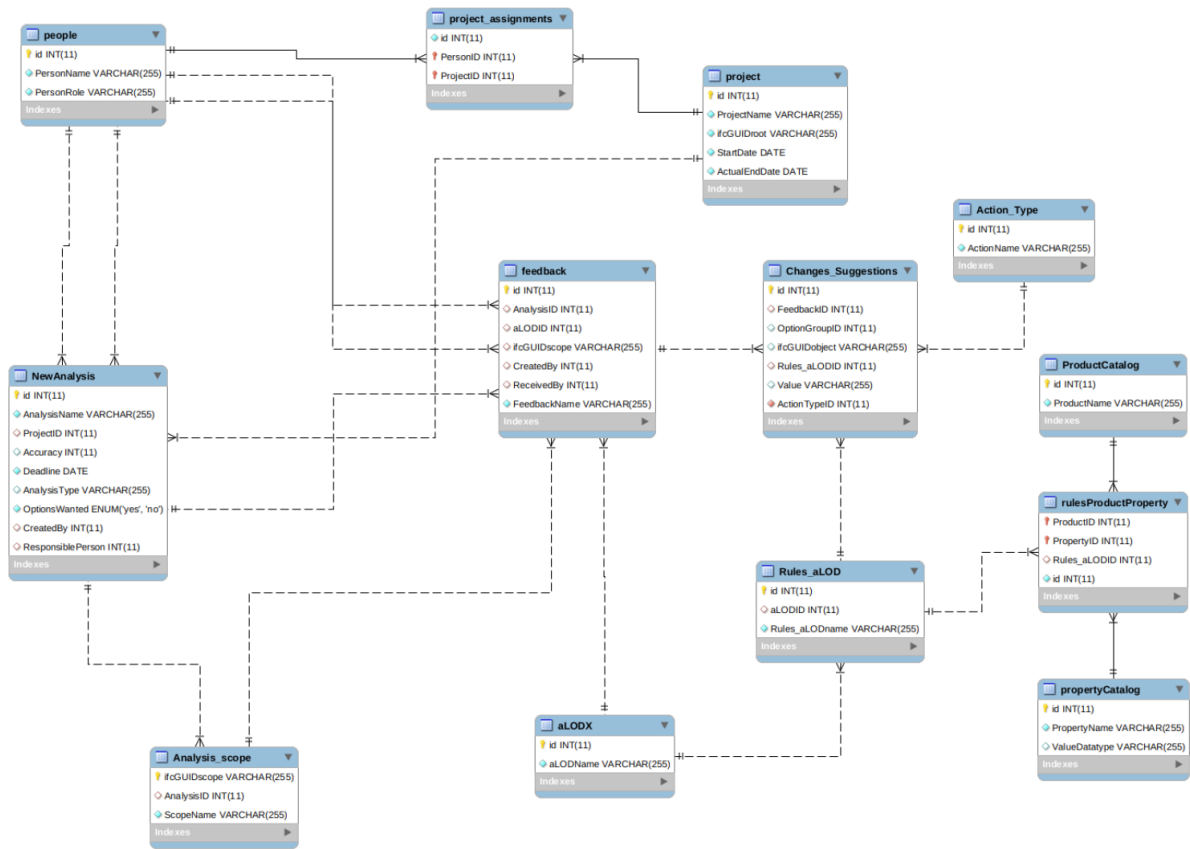


Figure 3.1 A database schema, by Christopher Onuoha, a part of an IDP project, TUM, December 2018

For every database table, the server provides REST API point to:

- insert an element,
- get an element by its ID,
- get all elements which contain a given keyword,
- get all elements from the table,
- update an element, and
- delete an element.

A client and a server applications exchange information. When an architect requests an analysis, an instance of a *New Analysis* is created in the database, and when an engineer runs the analysis,

an instance of a *Feedback* is created. Even though it has the same name as the function defined by [Zahedi and Petzold, 2018], it should not be confused with it. Feedback function is more comprehensive, whereas Feedback database table is not a top-level entity; it is dependent on other database tables, namely *NewAnalysis* and *Change_Suggestion*.

Each flaw reported to an architect is stored in the database as an instance of *Change_Suggestion* object. *Change_Suggestion* is an appropriate name for the table in the database, since it is very descriptive. Yet, it would be too long to be presented to a Revit user, and potentially ambiguous. If a model has a flaw, it means that the architect's task is fix that flaw. Hence, an engineer creates a Task and an architect's task is to solve them. Even though it introduces a second term for the same concept, i found it more appropriate. Hence, every time a model flaw is presented to a Revit user, it is referred to as a *Task*.

4 Implementation

Autodesk¹ Revit is a Building Information Modeling software available for Windows operating system. It provides a lot of built-in functionalities to architects and engineers. Nonetheless it allows this set of available features to be easily extendable by having a plugin-based architecture. Developing a plug-in would not be possible if Revit did not expose to developers the same classes and methods which the native commands use. A list of these classes and methods that are exposed, together with their names, input and output parameters and descriptions, is called Application Programming Interface (API). Revit API is available for .NET framework 4.0 compatible languages, such as Visual Basic .NET or Visual C#. With this in mind, I chose to develop the plug-in using C#, a general-purpose, multi-paradigm programming language developed by Microsoft.

Autodesk seems to be using terms "plug-in" and "add-in" interchangeably. In addition, Revit "sees" a plug-in as an external application, which will be explained in Section 4.3. Considering this, I will use words *plug-in* and *application* interchangeably in the remainder of this thesis.

This chapter is organized as follows: firstly, we have an introduction to developing a Windows application. Secondly, I shortly explain the third-party libraries that have been used in the development of the plug-in and why they were needed. This is followed by an overview of Revit plug-in ecosystem. Next, I talk about the system's predefined data and use it to explain the user interface of the application. Finally, the implementation of the plugin is described.

4.1 An introduction to developing a Windows application

Windows Presentation Foundation (WPF) is part of the .NET framework responsible for building a user interface for Windows desktop applications. To achieve this, WPF uses Extensible Application Markup Language (XAML). XAML is similar to eXtensible Markup Language (XML), but it adds additional features. For example, we can use a keyword *Binding* to bind a variable to a XAML control, so that if the variable (a data source) changes, the XAML control (a view) changes as well and vice versa.

Every XAML file is followed by a file with extension *.xaml.cs*. This file is called *code-behind*. It can be used to initialize some data which is presented in the XAML control or handle events, for example, a button click or an input text change.

WPF was developed with Model View ViewModel (MVVM), a software architectural pattern, in mind, but it is not mandatory to use it. In this pattern, a *Model* defines a format in which the data is to be stored and used. Very often it implements the *INotifyPropertyChanged* interface, to allow for the aforementioned data binding. The *Model* is not aware of either *View* or *ViewModel*. A *View* presents data to a user, but it does not store any data itself. However, the *View* holds information on its *DataContext* - a *ViewModel* that stores the data and implements logic. Except for the *DataContext*, the *View* should not hold any data or logic itself; its sole responsibility is to present data to the user. The *ViewModel* glues a *View* and a *Model*; it holds data, so it has information on the *Model*, but it should not know about the *View*.

Nowadays, there are almost no applications that work in offline mode only; the majority of applications exchange some data over Internet. In the next section, I shortly explain details of those exchanges.

HTTP communication protocol and its request methods

The Hypertext Transfer Protocol (HTTP) is a communication protocol in which a *client* sends a *request* to a server and the *server* sends back a *response*. An example of a *request* is when a person uses a web

¹<https://www.autodesk.com/>

browser (a *client*) to ask for a list of recent news from a newspapers website (a *server*), or when the same person posts a comment on the news they have read on the aforementioned website.

The most common HTTP request methods are:

- *POST*, which is used to create a new resource,
- *GET*, which is used to retrieve data from the server,
- *PUT*, which is used to update the specified resource, and
- *DELETE*, which is used to delete the specified resource.

This plug-in that i have developed is an example of an HTTP client, since it communicates with an existing server, which is a simple wrapper around an existing MySQL database.

4.2 Framework and libraries used to develop the plug-in

NuGet

NuGet is a package manager for .NET framework. It is used to install, update and deinstall third-party libraries and frameworks.

Json.NET

When data is exchanged over the internet, it is sent in a form of bits, i.e. zeros and ones. It is up to developers on both sides of this channel to decide on a communication exchange protocol, in order to be able to “pack” the data before sending and “unpack” data upon receiving it.

Json.NET² is a “popular high-performance JSON framework for .NET”, as per the authors’ definition. It is an open source library used to serialize and deserialize JSON data.

Json.NET developers claim to be “50% faster than DataContractJsonSerializer, and 250% faster than JavaScriptSerializer”, both classes being part of .NET framework. Json.NET is easy to integrate into the project using NuGet, described in Section 4.2. An example of the usage is deserialization of the data received from the server, that should be casted into the client-side model, e.g. an instance of the class `People`.

Listing 4.1 Deserializing data received from a server into a `People` object

```
var response = _client.Get(request);  
return JsonConvert.DeserializeObject<People>(response.Content);
```

More examples of the usage of this library can be found in Section 4.7.16.

RestSharp

RestSharp³ is “Simple REST and HTTP API Client for .NET”, as per the authors’ definition. It would be possible to establish this RESTful communication without a third-party library, since a native `System.Net.Http` provides an `HttpClient` class. Therefore, sometimes it is just a matter of personal preference whether to use a native or a third-party feature. RestSharp is easy to integrate into the project using NuGet, described in Section 4.2.

²<https://www.newtonsoft.com/json>

³<http://restsharp.org>

Extended WPF Toolkit™

Extended WPF Toolkit™⁴ is a free and open source collection of WPF controls, provided under the Microsoft Public License. It seems to still be popular, even though it is no longer actively developed. However, this could be due to the lack of the alternatives. Extended WPF Toolkit™ is easy to integrate into the project using NuGet, described in Section 4.2.

Material Design In XAML

Material Design In XAML⁵ is a UI library. Including the library gives the application a modern look and feel.

Material Design In XAML is easy to integrate into the project using NuGet, described in Section 4.2. However, using this plug-in in the Revit application is not a straightforward process, due to a specific way in which Revit loads frameworks and libraries its plug-ins require.

Even though the official *Getting started* tutorial states that it is as simple as creating the resource dictionary by merging four XAML files, and including it in the control where we want to use it, this was not enough. I defined the resource dictionary file in the following way:

Listing 4.2 Creating a resource dictionary by merging XAML files

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;
      component/Themes/MaterialDesignTheme.Light.xaml" />
    <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;
      component/Themes/MaterialDesignTheme.Defaults.xaml" />
    <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;
      component/Themes/Recommended/Primary/MaterialDesignColor.DeepPurple.xaml" />
    <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;
      component/Themes/Recommended/Accent/MaterialDesignColor.Lime.xaml" />
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

To make it available to the concrete XAML file, this resource needs to be specified in the following way:

Listing 4.3 Specifying a single Window's resource

```
<Window.Resources>
  <ResourceDictionary
    Source="/MarijaRakicMasterThesisAssembly;component/MaterialDesign.xaml" />
</Window.Resources>
```

MarijaRakicMasterThesisAssembly is a solution assembly file, and MaterialDesign.xaml is a name of the resource dictionary file.

In case the window needs to specify more than one resource, e.g. we want to include a converter and a Material Design resource dictionary, resources need to be merged in the following way:

Listing 4.4 Specifying multiple Window's resources

```
<Window.Resources>
  <ResourceDictionary>
    <BooleanToVisibilityConverter x:Key="Converter" />
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary
        Source="/MarijaRakicMasterThesisAssembly;component/MaterialDesign.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
```

⁴<https://github.com/xceedsoftware/wpftoolkit>

⁵<http://materialdesigninxaml.net/>

```

        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Window.Resources>

```

After taking all these steps, the plug-in should successfully use the Material Design library, but unfortunately it was not the case. I found no resources explaining why this problem occurs. One GitHub⁶ issue suggested the following solution, which turned out to work:

Listing 4.5 The hack to make the application work

```

ColorZoneAssist.SetMode(new GroupBox(), ColorZoneMode.Accent);
Hue hue = new Hue("name", System.Windows.Media.Color.FromArgb(1, 2, 3, 4),
                  System.Windows.Media.Color.FromArgb(1, 5, 6, 7));

```

A small presentation of how using Material Design makes a visual difference is shown in Figure 4.1. This was achieved by simply including a Material Design library. No changes to the code itself were made, no additional properties, attributes or different XAML elements have been used.

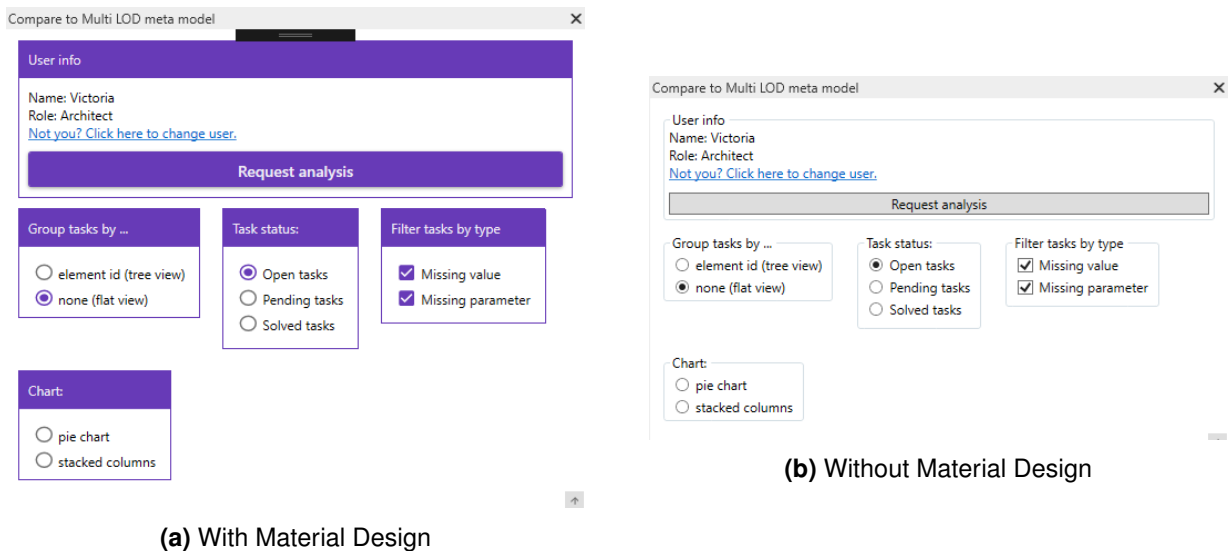


Figure 4.1 The same plugin page, with and without Material Design

Live Charts

Live Charts⁷ is a “Simple, flexible, interactive & powerful data visualization for .Net”, as per the authors’ definition. It is a free and open source library that offers many different charts out-of-the-box. This application utilizes only two of them, due to the limited information stored on the server, described in Section 3. The future versions of the application could easily include more graphs. Live Charts is easy to integrate into the project using NuGet, described in Section 4.2.

Similarly to including Material Design library, this library requires a hack to make it work in combination with Revit. The following code snippet comes from Pie chart code-behind, and it handles a Pie chart’s click event:

Listing 4.6 A hack to make the LiveCharts library work, code-behind

```

private void Chart_OnDataClick(object sender, ChartPoint chartpoint)
{
    // an empty method

```

⁶https://github.com/
⁷https://lvcharts.net/

}

Listing 4.7 A hack to make the LiveCharts library work, XAML click event

```
<PieChart DataClick="Chart_OnDataClick" />
```

4.3 Revit plug-in ecosystem

From a Revit user's point of view, the basic application should be as small as possible, so that the user is not overwhelmed with the functionality they do not need for their field of expertise. Nevertheless, each user can customize Revit with the plug-ins they will actually use. This makes the application faster and less memory demanding compared to the full packed software, without compromising the set of functionalities it offers. Autodesk offers an official marketplace of Revit plug-ins that contains a little more than one thousand plug-ins in June 2019.

A code of the plug-in is compiled into a Dynamic Link Library (DLL) file. A DLL file (and consequently the plug-in as well) cannot run standalone, it can only be used as part of another executable program that runs on a Windows platform. From a developer's point of view, a DLL provides modularity and memory efficiency, to name a few. The application can get new features easily, without modifying the application itself.

Three types of plug-ins

There are three ways to extend existing Revit functionalities: building a *Macro*, an *External Command*, which is executed on a button click, and an *External Application*, which starts and ends with Revit. I developed the plug-in in form of an *External Application*, since it provides with the most possibilities, as will be shown in the following section.

Ribbon tab, panel and button

An *External Application* allows us to customize ribbon tabs and ribbon panels. The implementation of this functionality is explained in Section 4.7.2. For now, we just take a look at the graphical outcome of the aforesaid implementation.

We navigate to the *Add-ins* tab, where we can see a panel named TUM.

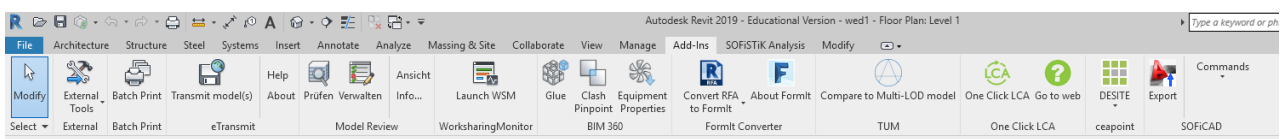


Figure 4.2 Add-ins tab with many panels, including the one called TUM

A panel can contain one or more buttons. Each button has an External command bound to it. Our panel contains only one button named *Compare to Multi-LOD model*, as shown in Figure 4.2.

.addin file

When we start developing a plug-in, we need to tell Revit about it. Revit requires different information in order to run our plug-in successfully, most notably the location of our assembly file (under the *Assembly* tag) and the entry point (under the *FullClassName* tag), i.e. an instance of the class that implements the *IExternalApplication* interface. All this information has to be stored in the XML format in a predefined location on a local system. The location is likely to be in the following format:

```
C:\ProgramData\Autodesk\Revit\Addins\{Revit version}
```

Listing 4.8 An example of the .addin file

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Application">
    <Name>
      MarijaRakicMasterThesis
    </Name>
    <FullClassName>
      TUM.MasterThesis.MarijaRakic.ExternalApplication
    </FullClassName>
    <Text>
      MarijaRakicMasterThesis
    </Text>
    <Description>
      Lean BIM-based communication and workflow during design phases
    </Description>
    <VisibilityMode>
      AlwaysVisible
    </VisibilityMode>
    <Assembly>
      C:\Users\ga83cay\Documents\Marija Rakic - Master Thesis\MarijaRakicMasterThesis\
      MarijaRakicMasterThesis\bin\Debug\MarijaRakicMasterThesisAsembly.dll
    </Assembly>
    <AddInId>
      502fe383-2648-4e98-adf8-5e6047f9dc34
    </AddInId>
    <VendorId>
      ADSK
    </VendorId>
    <VendorDescription>
      Autodesk, Inc, www.autodesk.com
    </VendorDescription>
  </AddIn>
</RevitAddIns>
```

4.4 System setup

In order to demonstrate how the plug-in works, some database tables had to be prefilled with values before running the application. Those will be listed later in this section.

In our system, a user can have one of three roles:

- *Architect*,
- *Life Cycle Analyst (LCA)*, and
- *Structural Engineer*.

Our system has three users:

- *Victoria*, who is an *Architect*,
- *Sarah*, who is a *LCA*, and
- *Jessie*, who is a *Structural Engineer*.

In our system, one project is defined:

- *MasterThesis*

Our system keeps information about which users are assigned to which projects. Each of our three users is assigned to the project.

There are six properties:

- *Height* of data type *Number*,
- *Width* of data type *Number*,
- *StructuralMaterial* of type *Material*,
- *IsExternal* of data type *YesNo*,
- *LoadBearing* of data type *YesNo*, and
- *ThermalTransmittance* of data type *Number*.

There is one product:

- *Wall*

There is a mapping between properties and products, and therefore every property is mapped with a *Wall*.

Our system defines the following action types:

- *MissingObject*,
- *MissingObjectProperty*,
- *CreateNewObject*,
- *DeleteObject*,
- *UpdateObjectProperty*, and
- *MissingPropertyValue*.

Although six action types are defined, only two are used at the moment: *MissingObject* and *MissingObjectProperty*. The rest of the actions could be implemented in some future versions of the application.

There is one aLODX:

- *MasterThesisaLODX*.

4.5 Graphical User Interface

4.5.1 Starting the plug-in for the first time

When a user starts the application for the first time, he/she is prompted with the list of existing users in the system. As explained in Section 4.4, our system has three registered users, one per each role. Information on the user's name and role are shown in Figure 4.3. The user chooses their identity by clicking on one of the rows.

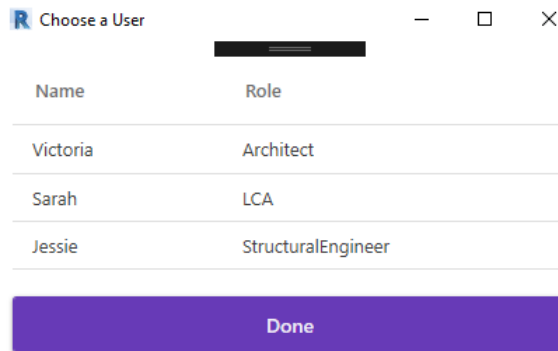


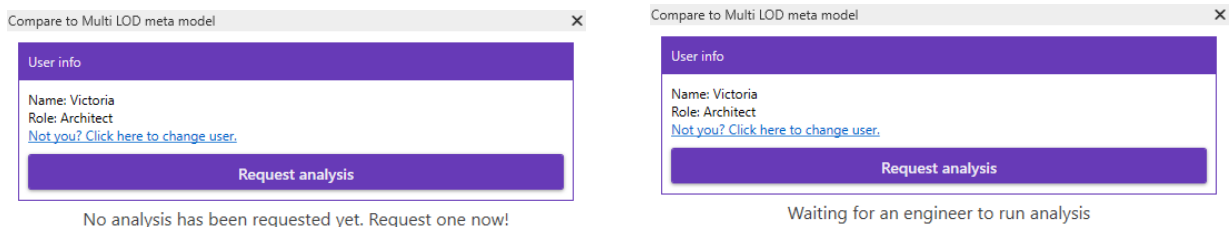
Figure 4.3 Choose a user

Since this plug-in is just a demo of what can be achieved, many features are simplified or completely omitted. The user log-in process is an example of a security feature that is omitted. The current implementation of the application does not require any password to confirm the user's identity. However, in a real life application, a user would have to confirm their identity by inputting a username and password.

Allowing the user to create an account, or at least create a request for an account, is another example of the omitted feature. At the moment, all users are predefined in the system, and a person who would like to join the system has no way of creating the account themselves. They would have to contact the owners of the application and request an account from them. This mimics the use case where an administrator or a responsible person from a company creates accounts for all their employees and assigns them appropriate roles.

4.5.2 Home page for an architect

Let's assume that the current user is our architect, Victoria. She creates a building model using Autodesk Revit. At some point, she needs to know what a structural engineer thinks of her current building model. She starts the plug-in by clicking on *Add-Ins* and then *Compare to Multi-LOD model* under *TUM* panel, as seen in Figure 4.2. Her home page looks like Figure 4.4a.



(a) Before an analysis has been requested

(b) After an analysis has been requested

Figure 4.4 Home page of the plug-in, as seen by an architect

First, information on the current user's name and role is presented in the upper left corner. Next, there is a message that *No analysis has been requested yet*. It also suggests that analysis can be requested at

any moment by clicking on a big *Request analysis* button. Upon clicking on this button, a window as seen in Figure 4.5a is presented.

4.5.3 An architect requesting an analysis

In order to request a new analysis of the current building model, our architect Victoria has to choose a person who will be responsible for running this analysis. She can choose between all engineers registered in the system. In our case, that is the choice between Sarah, a Life Cycle Analyst, or Jessie, a Structural Engineer. If there were more engineers registered in the system, they would be presented on this list as well. On the other hand, even if there were more architects registered in the system, they would not show up on this list.

Since our architect Victoria wants an opinion on the structural aspects of the building model first, she selects Jessie by clicking on her name, chooses whether a structural engineer should create options for the missing values or not, and a deadline for the analysis. The past dates are greyed out and cannot be selected, assuring that the user selects only a valid date for a deadline. After all these options have been chosen, Victoria clicks the *Request analysis* button, which permanently saves her request. All these elements can be seen in Figure 4.5a.

(a) An empty form

(b) A filled in form

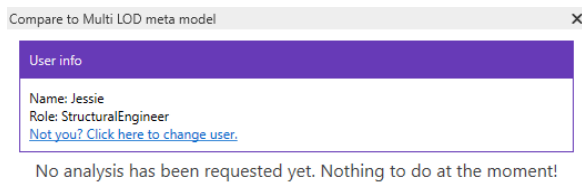
Figure 4.5 Form for requesting a new analysis, as seen by an architect

After an analysis has been requested, the home page that the architect sees slightly changes, as seen in Figure 4.4b. A new message is presented to the user, informing them that the analysis has been successfully requested and that they are now *Waiting for an engineer to run analysis*. The *Request analysis* button is still available, in the case the architect wants to request another type of analysis.

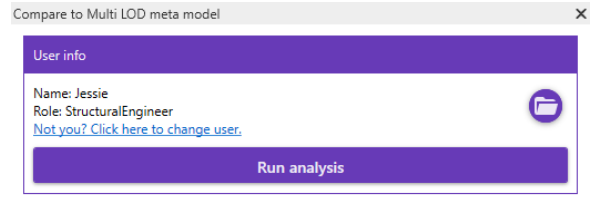
In the current implementation of the system, a responsible engineer becomes aware of the newly created request for an analysis only when starting the application. In future versions, it could be possible to implement a system where a user would be notified by a desktop notification once a new request exists, assuring no requests go unnoticed and they are processed as soon as possible.

4.5.4 Home page for an engineer

When Jessie, our structural engineer, starts the application, her home page looks as seen in Figure 4.6a. The name and role are presented in the upper left corner, in the same way as they are presented to an architect. Before any architect requests an analysis from her, she sees a message that *No analysis has been requested yet. Nothing to do at the moment!* But as soon as there is a request, a message will change to inform her that *analysis [has been] requested. Please run it* and the button *Run analysis* appears on the screen to make this action possible, as seen in Figure 4.6b.



(a) Before an analysis has been requested



(b) After an analysis has been requested

Figure 4.6 Home page of the plug-in, as seen by an engineer

When running the analysis for the first time, an engineer is prompted to choose a file that contains a list of requirements that the building model should comply with. Once this file has been chosen, this information is stored locally and the same file will be used for all the future analyses. If the engineer wants to choose another file at any later point, they can do that by clicking on the round directory button in the upper right corner, as seen in Figure 4.6b.

4.5.5 Analysis results

It can happen that the engineer is not able to decide on the building model's characteristics, because the model is missing information needed to run the analysis successfully. In this case, the output of the analysis is a list of tasks that the architect has to solve before the analysis could be performed.

A task can specify that the Revit element is missing a:

- *property*, e.g. a task that specifies that the *Wall* does not have its *Thermal Transmittance* property, as seen in Figure 4.7a, or
- *value*, in the situation when the parameter exists but does not have a value. An example is a task that specifies that the *Wall* is missing the value of its *Height* parameter, as seen in Figure 4.7b.

This list of tasks is presented in a table-like structure to both the engineer who ran the analysis and the architect who requested it. Every row holds information on a single task. There are four columns:

- *Element*, showing the Revit element's name,
- *Description*, showing whether the element is missing a parameter or a value,
- *Value*, showing a suggestion created by an engineer or value input by an architect, and
- *Action*, allowing an engineer to input a suggestion, or an architect to either input a value themselves, or accept or decline a value that has been suggested by the engineer.

In addition to this, every row starts with a circle that will be further explained in Section 4.5.15.

Figure 4.7 present the list of tasks the architect sees. There are three scenarios: all tasks specify a missing parameter, as seen in Figure 4.7a; tasks are a mixture between missing parameters and missing values, as seen in Figure 4.7b; or all tasks specify a missing value. The text on the action button indicates the action that the architect will take by clicking it.

Compare to Multi LOD meta model

User info
Name: Victoria
Role: Architect
[Not you? Click here to change user.](#)

Request analysis

Group tasks by ...
 element id (tree view)
 none (flat view)

Task status:
 Open tasks
 Pending tasks
 Solved tasks

Filter tasks by type
 Missing value
 Missing parameter

Chart:
 pie chart
 stacked columns

Element	Description	Value	Action
Generic - 200mm	Missing parameter Height		Add Paramete
Generic - 200mm	Missing parameter Width		Add Paramete
Generic - 200mm	Missing parameter StructuralMaterial		Add Paramete
Generic - 200mm	Missing parameter IsExternal		Add Paramete
Generic - 200mm	Missing parameter LoadBearing		Add Paramete
Generic - 200mm	Missing parameter ThermalTransmittance		Add Paramete
Generic - 200mm	Missing parameter Height		Add Paramete
Generic - 200mm	Missing parameter Width		Add Paramete

(a) Missing parameters

Compare to Multi LOD meta model

User info
Name: Victoria
Role: Architect
[Not you? Click here to change user.](#)

Request analysis

Group tasks by ...
 element id (tree view)
 none (flat view)

Task status:
 Open tasks
 Pending tasks
 Solved tasks

Filter tasks by type
 Missing value
 Missing parameter

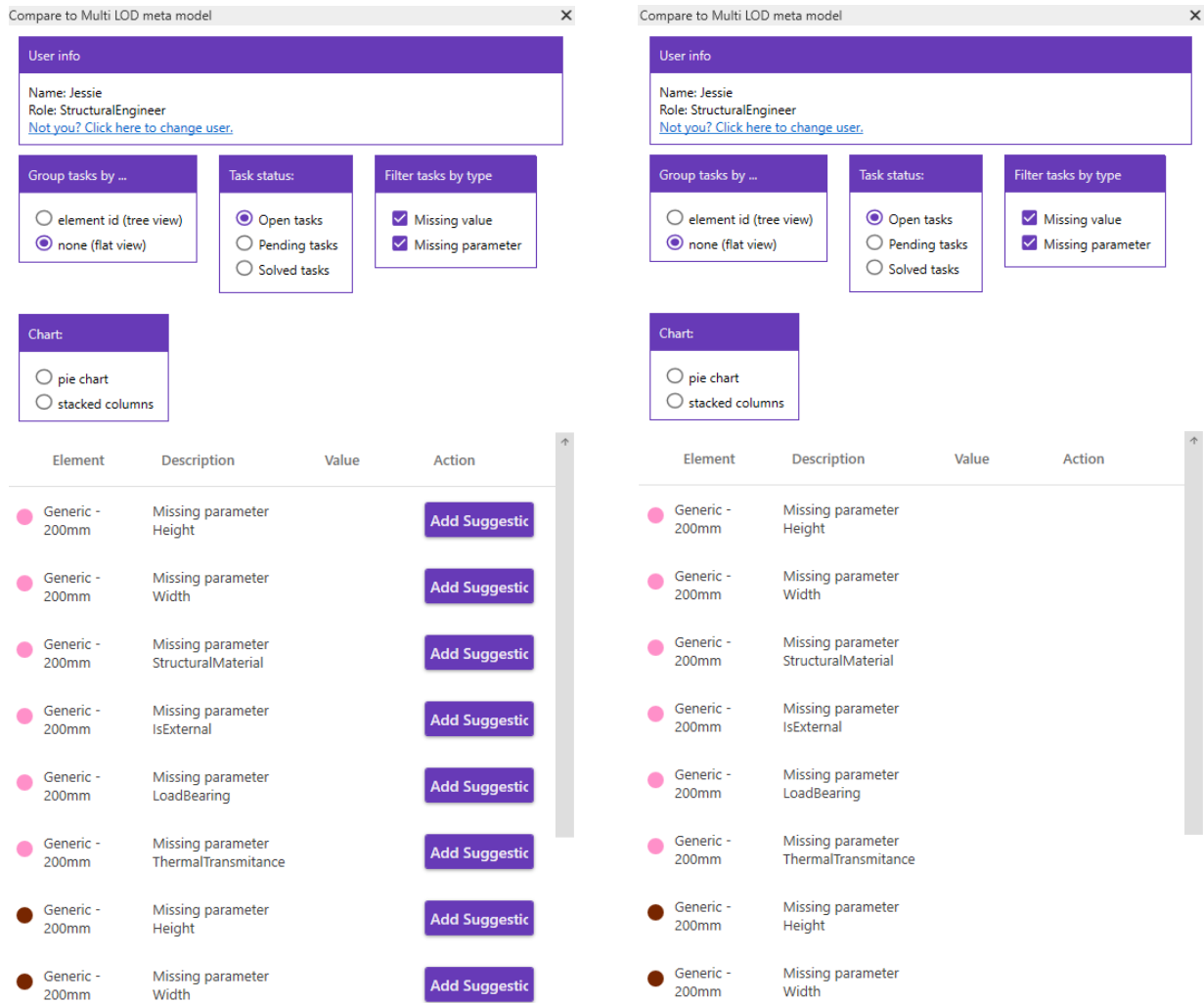
Chart:
 pie chart
 stacked columns

Element	Description	Value	Action
Generic - 200mm	Missing parameter Width		Add Paramete
Generic - 200mm	Missing parameter StructuralMaterial		Add Paramete
Generic - 200mm	Missing parameter IsExternal		Add Paramete
Generic - 200mm	Missing parameter LoadBearing		Add Paramete
Generic - 200mm	Missing parameter ThermalTransmittance		Add Paramete
Generic - 200mm	Missing value of Height		Add Value
Generic - 200mm	Missing parameter Width		Add Paramete
Generic - 200mm	Missing parameter StructuralMaterial		Add Paramete

(b) Missing parameters and values

Figure 4.7 Analysis results, as seen by the architect

If the architect requested suggestions for the missing values, the engineer can see a button to *Add suggestion* for every missing value, as in Figure 4.8a. If options were not requested, the engineer will not have this button, as seen in Figure 4.8b.



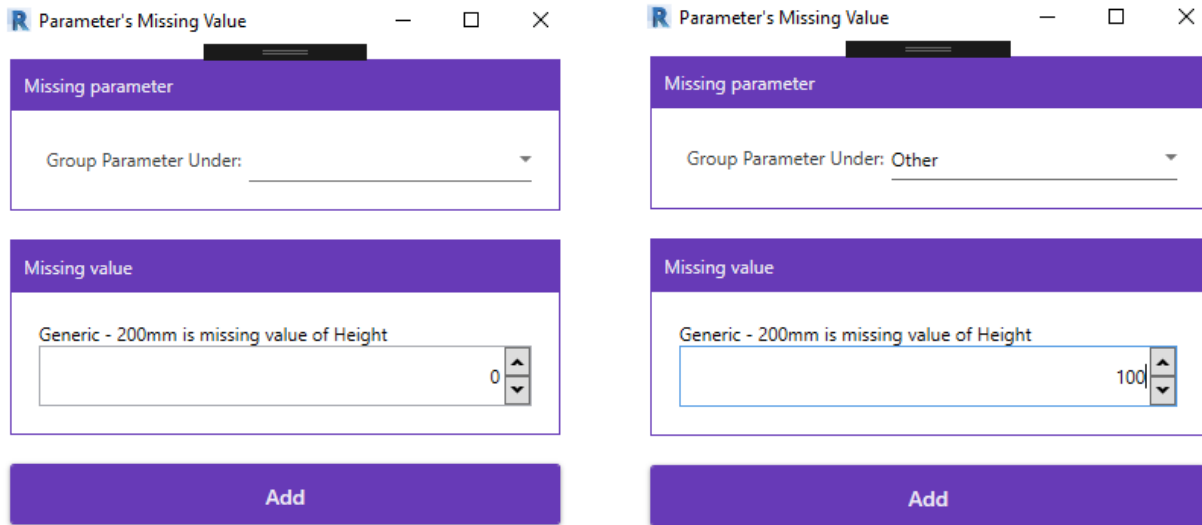
(a) Options were requested

(b) No options were requested

Figure 4.8 Analysis results, as seen by the engineer

4.5.6 Creating a missing parameter

Clicking either *Add Parameter* or *Add Value* buttons, a new window opens and asks the architect to input the value. For a missing parameter, the architect also has to decide which group to place the newly created parameter in. This is a mandatory step, so if this group is not specified, another window will appear and inform the user that they cannot proceed without defining this value.



(a) An empty form

(b) A filled in form

Figure 4.9 Creating a missing parameter, as seen by an architect

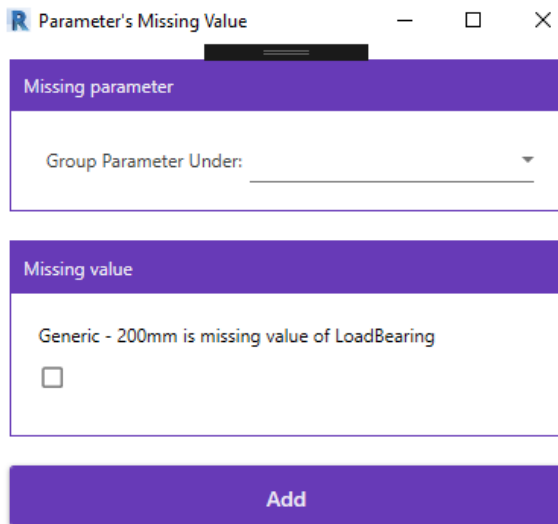
One important note when inputting decimal numbers using this `DecimalUpDown` control: the control infers decimal separator from the settings of the operating system. For example, in US-settings, the decimal number would be defined using a dot, e.g. 3.14. However, the same value would be defined as 3,14 in many different countries, for example Germany.

4.5.7 Data type of a missing value

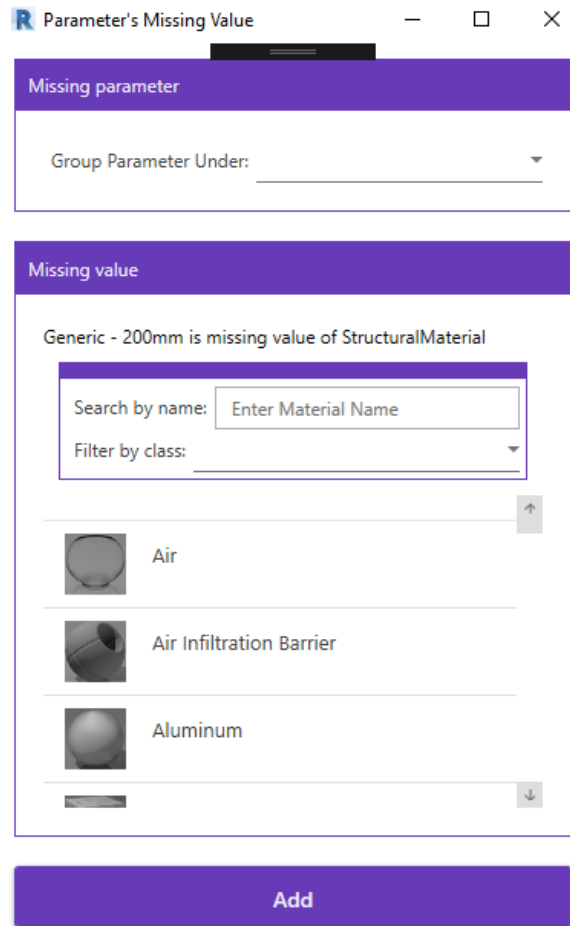
There are different types of missing values. In the current implementation of the application the following types are supported:

- numerical value,
- boolean value, and
- material.

Inputting each of these data types has a different interface. The interface for inputting a boolean value can be seen in Figure 4.10a, and for inputting a material in Figure 4.10b.



(a) A missing value is of type boolean



(b) A missing value is of type material

Figure 4.10 Creating a missing parameter of different data types, as seen by an architect

In the case of material, the user can simply scroll the list of available materials and select the desired material. The user also has an option to quickly access the desired material by either searching by its name or filtering all materials by class. Revit users are used to having these functionalities, since Revit offers the exact same search by name and filter by class features, as shown in Figure 4.11. I wanted to offer them the same user experience in our application.



Figure 4.11 A native Revit list of materials, with a search by name and filter by class features

4.5.8 Creating a missing value

If the parameter exists, but does not contain the value which the engineer needs to successfully run the analysis, the engineer creates a task for the architect to input this value. By clicking on the *Missing value* button as seen in Figure 4.7b, a new window is opened. Then, depending on the data type as explained in Section 4.5.7 a different field for inputting the missing value is presented, in the same way it is presented for the missing parameter, as seen in Figure 4.10. Figure 4.12 is an example of this window for creating a missing numerical value.

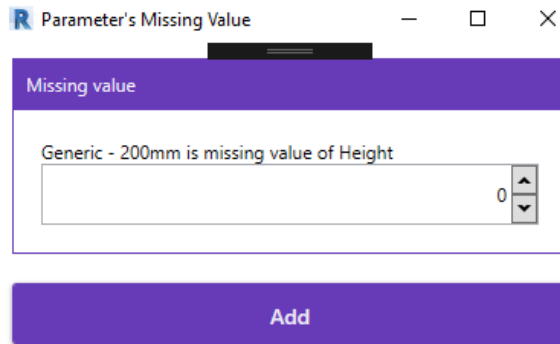


Figure 4.12 Creating a missing value

4.5.9 Filtering tasks by type

The user has an option to filter the list of tasks for only missing values or only missing parameters, by selecting the check boxes in the upper right rectangle named *Filter tasks by type*. Figure 4.13a shows results of filtering the list as seen in Figure 4.7b for only missing parameter results, while Figure 4.13b represents filtering the same list for only missing values.

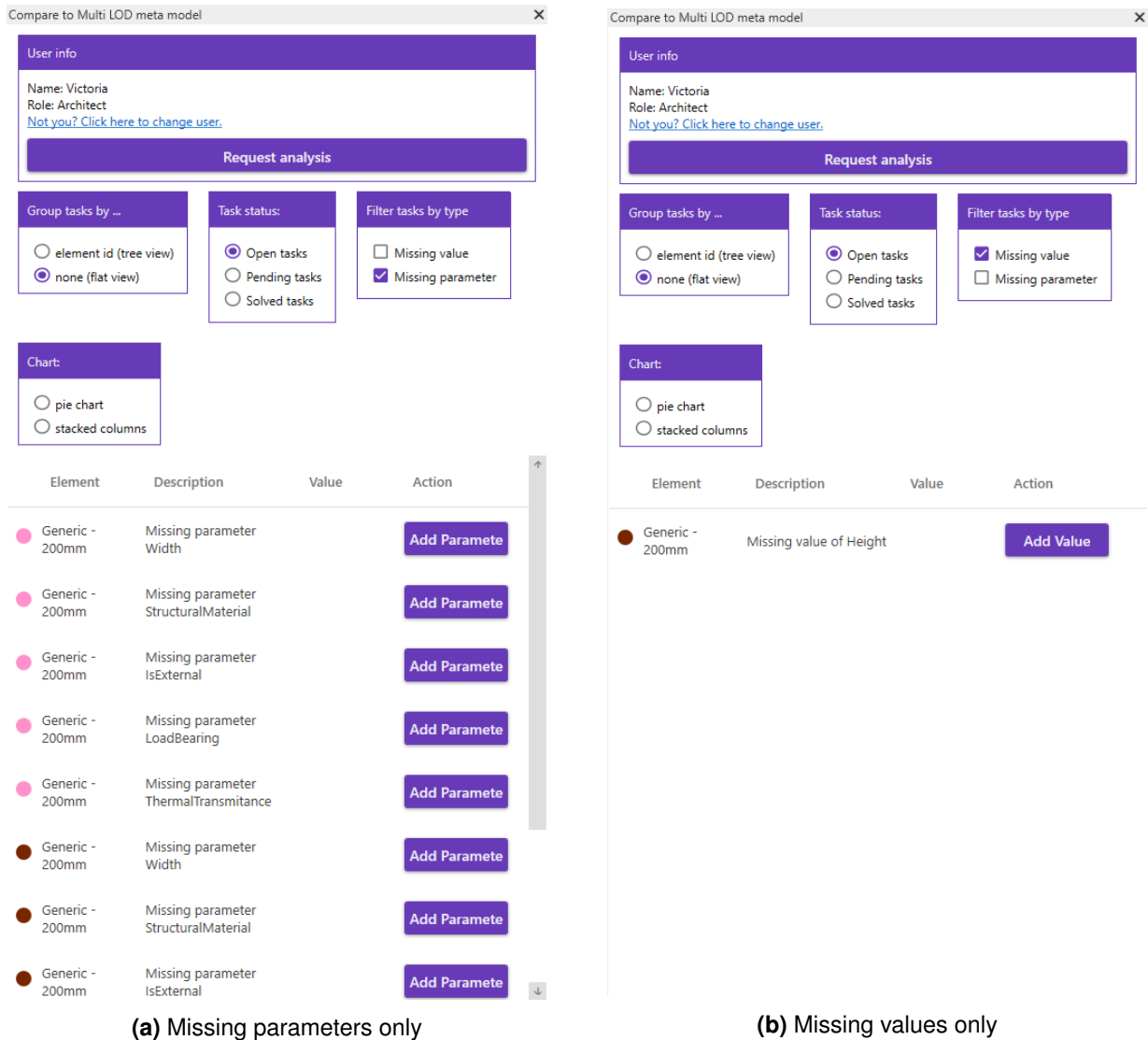


Figure 4.13 Filtering the list of tasks by type

4.5.10 Suggesting a missing value

Another way of creating a new value is when the architect asks the engineer to suggest one or multiple values for the missing element.

When the engineer clicks the button to add a suggestion, a new window opens, as seen in Figure 4.14a. If the engineer wants to suggest multiple possible values, they should select the box with a *Add another suggestion* label, as seen in Figure 4.14b.

When the engineer inputs a value and clicks *Add* button, the value is saved in the system and the window closes. However, if the checkbox *Add another suggestion* is selected, the value is saved in the system, yet the window remains open, giving the engineer an opportunity to input another possible value. When the engineer is done inputting values, they uncheck the *Add another suggestion* box by clicking on it again, and submit the last suggestion by clicking on the *Add* button.

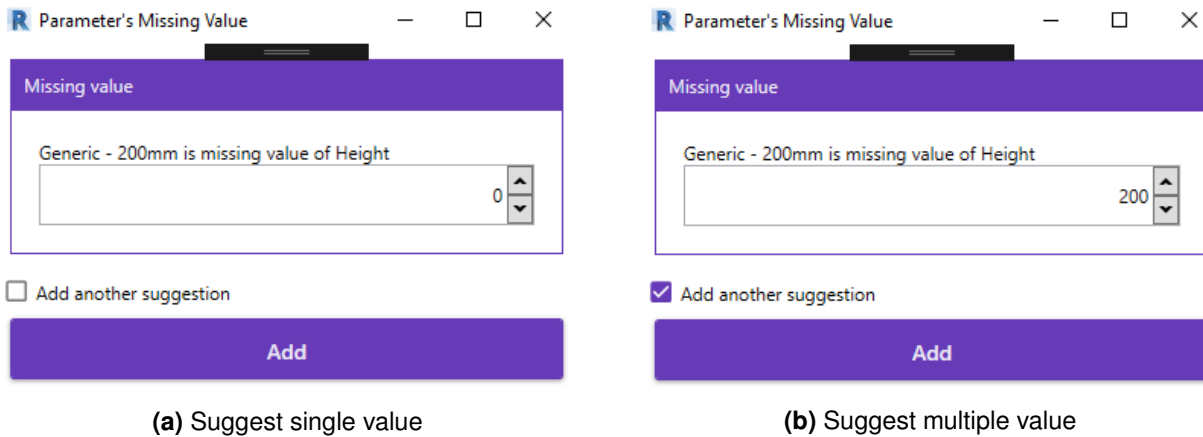


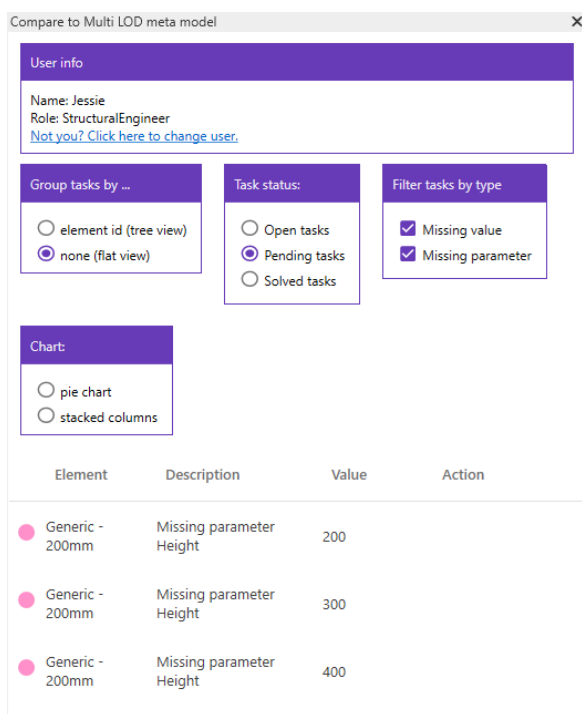
Figure 4.14 An engineer can suggest one or multiple values

This window is very similar to the window that the architect sees when creating a missing parameter, except for the box that defines the *Group parameter under* option. Since it is completely up to the architect how they will organize information on the building model within Revit, the engineer does not suggest this value.

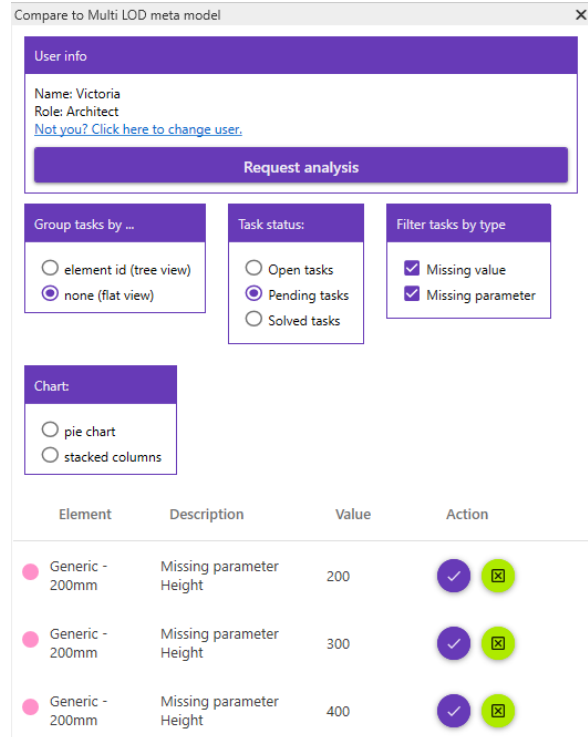
4.5.11 Pending tasks

After a suggestion has been created, it is removed from the list of open tasks and placed into the list of pending tasks, as seen in Figure 4.15a. After the engineer has created the suggestion, they can no longer take any action, but wait for the responsible architect to either approve or decline this value.

The architect sees the list of the pending tasks as shown in Figure 4.15b. They have an option of accepting or declining the suggested value by clicking on the purple circle with a check mark or a green circle with an x mark, respectively.



(a) As seen by the engineer



(b) As seen by the architect

Figure 4.15 A list of pending tasks

When the architect wants to accept one of the values suggested by the engineer, a new window as seen in Figure 4.16 opens and the value field is prefilled. If the parameter is missing, the architect still has to decide under which group to place it, therefore the upper box is yet to be filled in.

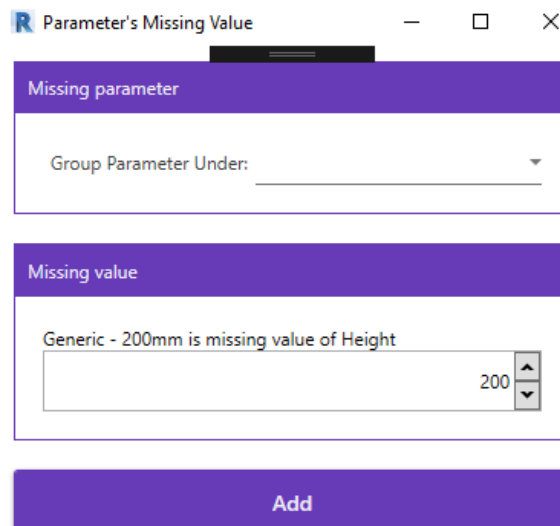


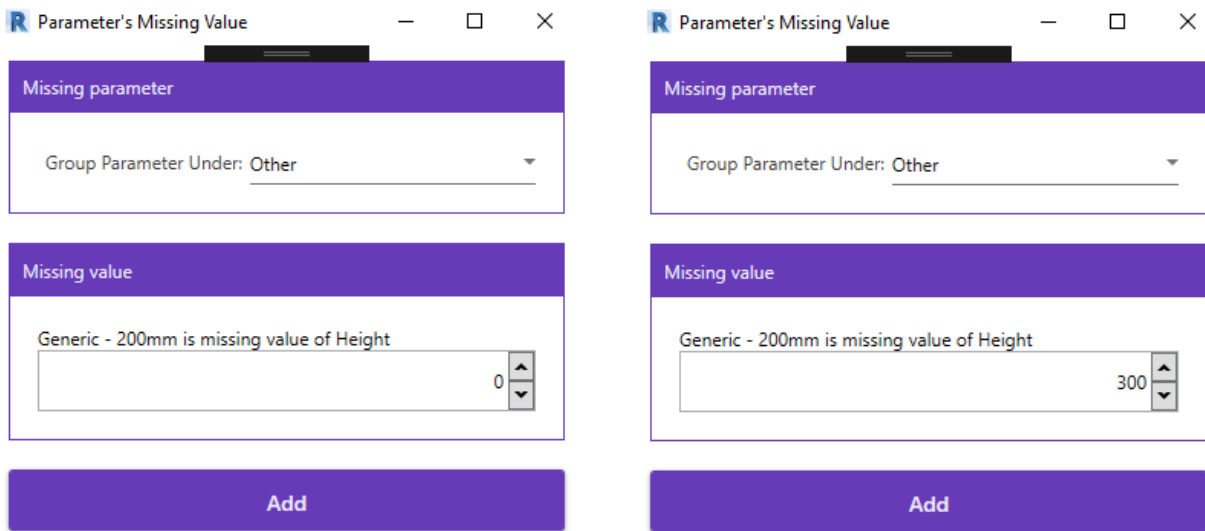
Figure 4.16 A value suggested by the engineer is automatically input in the value field

In the case of multiple suggestions per missing value, when the architect accepts one, all the others are automatically declined.

It can happen that the architect wants to create a missing parameter, and that the *Group parameter under* field is prefilled. This occurs in the following situation. An engineer runs the analysis and discovers that all *Walls* in the Revit model are missing a certain parameter, e.g. *IsExternal*. The engineer creates a task for each Revit element that holds this information. The architect receives the analysis report and starts

solving the problems one by one. The architect creates a missing *IsExternal* parameter for the randomly chosen *Wall*. By creating this shared parameter in order to assign it to the first *Wall*, the architect implicitly creates this parameter for all the other walls in the project. Yet this does not affect the remaining tasks that still specify the missing parameter for all the other *Walls*. Hence when the architect wants to solve another task of this kind, the *Group Parameter Under* data is prefilled, since Revit already has information on this now existing parameter.

Figure 4.17b shows the case when both shared parameter has already been defined, so the the *Group Parameter Under* data is prefilled, and the architect is about to accept the value suggested by the engineer, so the value field is also prefilled. The only thing left to the architect to do is to click *Add* button and the task is solved.



(a) Shared parameter exists, value is not suggested

(b) Shared parameter exists, value is suggested

Figure 4.17 Different combinations of existence of shared parameter and suggested value

4.5.12 Solved tasks

After a task has been solved, it is removed from the list of *Open* tasks and placed on the list of *Solved* tasks. Figure 4.18 shows that the architect has solved the task of missing parameter *Height* for the *Wall* of type *Generic - 200mm*, and this newly created parameter now has the value of 100.

To switch between those two lists, the user clicks the desired button under the *Task status* box.

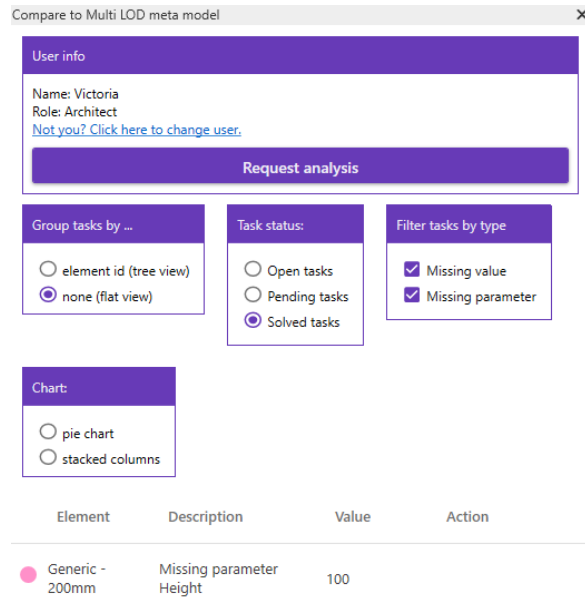


Figure 4.18 List of solved tasks

4.5.13 Newly created value under Revit Properties

Finally, after a value has been created, either a missing parameter or a missing value, it can be viewed in the Revit's *Properties* window. An example is our newly created *Height* parameter, grouped under *Other*, that has a value of 100, as seen in Figure 4.19.

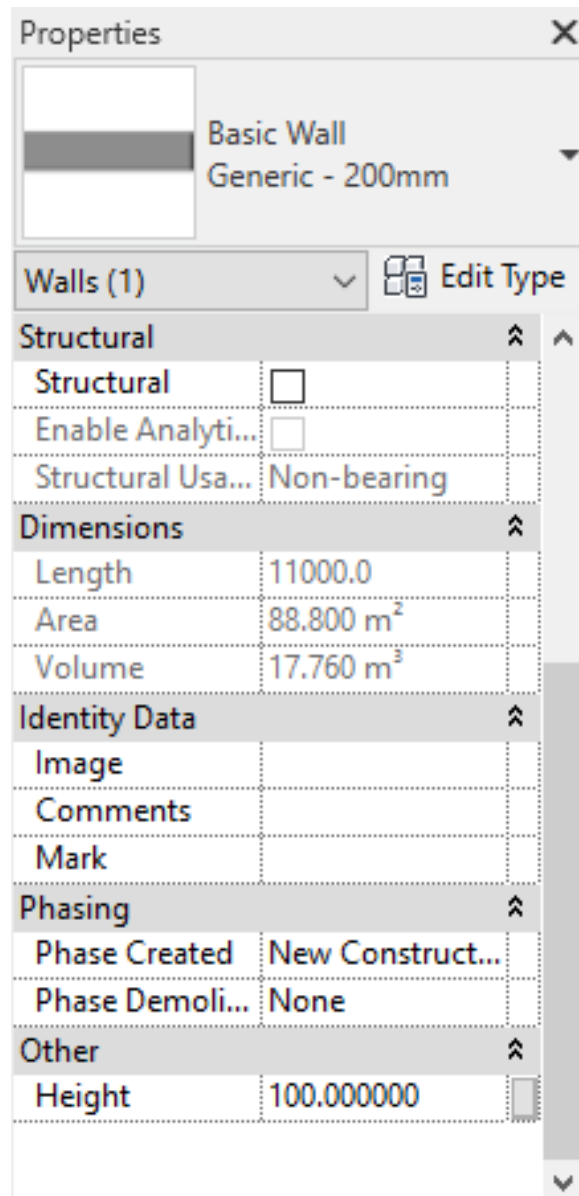
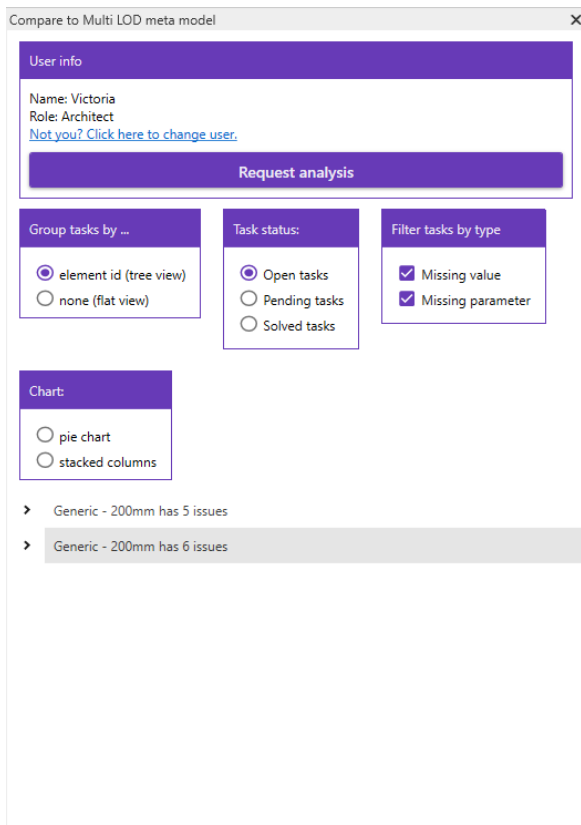


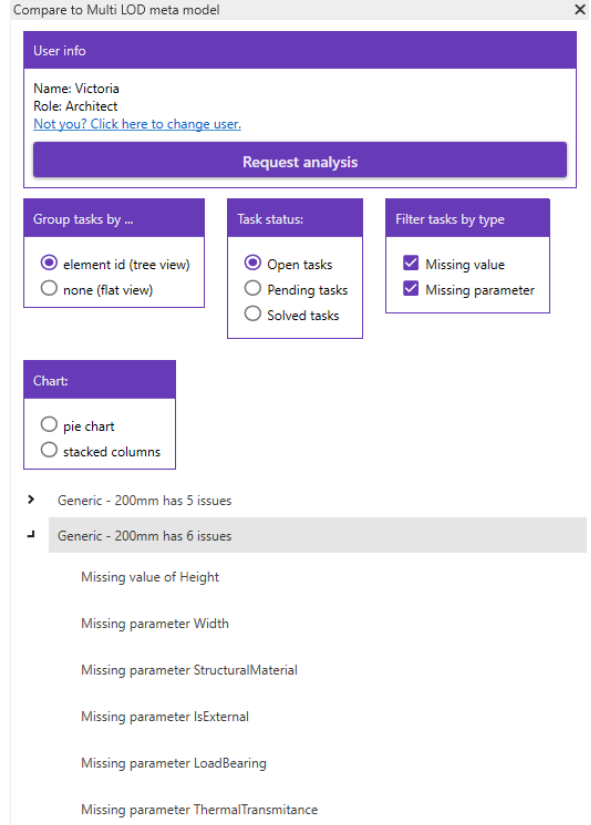
Figure 4.19 Newly created wall's parameter in the Properties view

4.5.14 Tree view

The list of tasks can be presented in a tree-like structure by grouping all the tasks that are related to a single Revit element, as shown in Figures 4.20a and 4.20b.



(a) A collapsed tree view



(b) An extended tree view

Figure 4.20 A tree view representation of tasks, grouped by Revit element

This representation of tasks is only a showcase that a flat list view with no grouping applied, as shown in previous sections, is not the only option. However, the tree view is there only to present data. Unlike the list view that groups tasks by their status (open, pending or solved) and allows for actions to be taken, the tree view does not offer any way of interacting with tasks. This is left to be implemented in some future versions of the application.

4.5.15 Selecting an element

The list of tasks can be very long and contain many tasks for many elements. To make it easier for the user to navigate and access the element to which the task refers, the application implements the following feature. Each row in the list of tasks starts with a small circle of a certain color. All tasks that relate to the same element have the the circle of the same color. When the users clicks the task, the element related to it changes color to the color of the small circle. This feature can be seen in Figure 4.21, which represents two different walls being colored when two different tasks are selected.

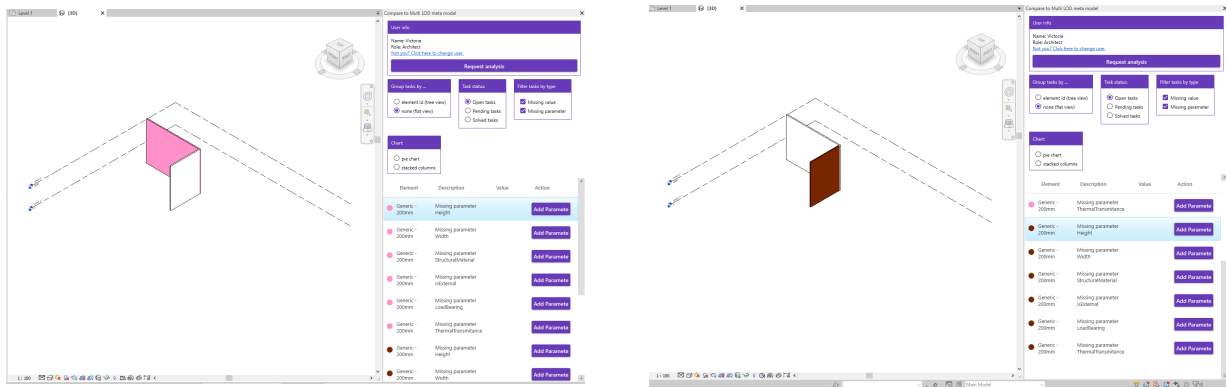


Figure 4.21 Wall is colored when selected

4.5.16 Charts

The user has an option to view some simple reports on the progress of solving tasks. The current version of the application implements two chart views:

- Pie chart, as seen in Figure 4.22a, and
- Stacked columns, as seen in Figure 4.22b.

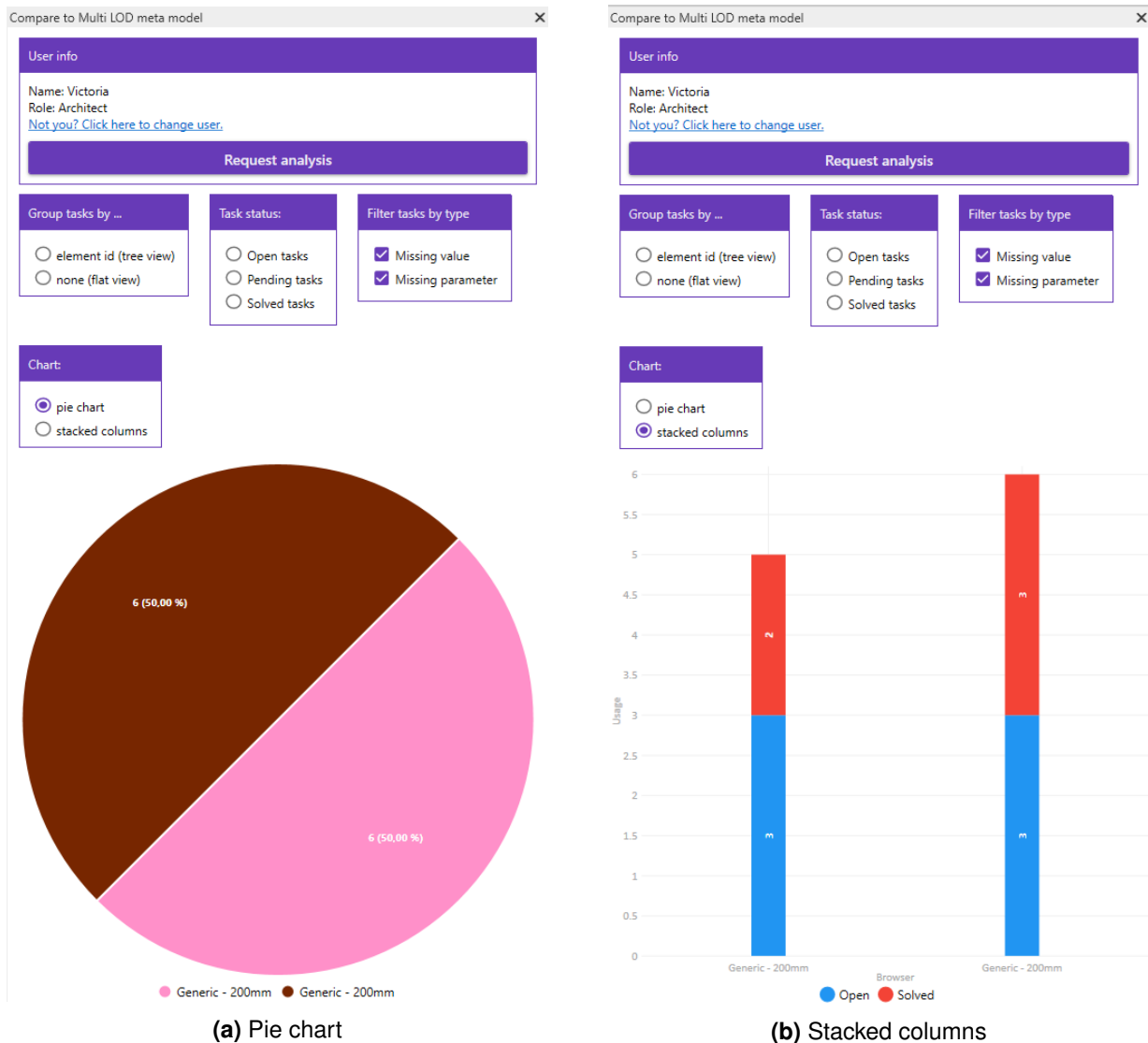


Figure 4.22 Available charts

Pie chart presents the ratio of the open tasks per a Revit element. An example shown in Figure 4.22a depicts a situation when all the open tasks refer to two instances of *Walls*, of type *Generic - 200 mm*. Both walls have six open tasks.

In stacked columns, each column represents a single Revit element. This single column further presents the ratio between open and solved tasks within a single Revit element. An example shown in Figure 4.22b depicts a situation when all the tasks refer to two instances of *Walls*, of type *Generic - 200 mm*. First one of them has three open and two solved tasks, while the other has three open and three solved tasks.

4.5.17 Changing the user

Information on the user who last used the application is stored locally, similar to *Stay logged in* feature that is very common in many mobile, web and desktop application. The next time the application is started, an assumption is made that the same person is using the application and so credentials are retrieved from the local storage. If this is not the case, the person who is currently using the application has an option to choose another identity by selecting one of the user currently defined in the system, excluding the currently logged in user, as seen in Figure 4.23.

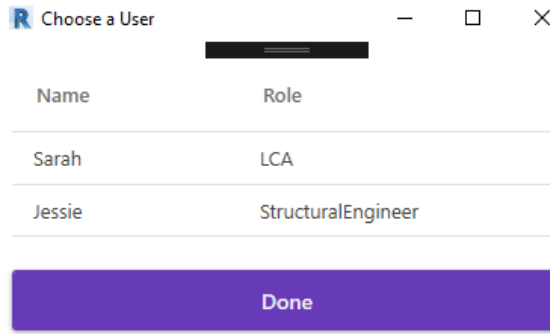


Figure 4.23 Changing the current user

4.5.18 Successfully running the analysis

After the architect has input all the values that the engineer has previously specified as needed, the architect informs the engineer of this event by requesting the analysis again.

Now the engineer can check that the Revit model contains all information they need to run the analysis successfully. After this is verified, they inform the architect that the analysis has been run successfully.

4.6 Communication within the system

This section gives a short overview of the communication that occurs between a client (a Revit plug-in) and a server (a Node server and a MySQL database) application.

A communication starts when an architect creates a request for a new analysis, as seen on Figure 4.24. A plug-in sends an HTTP request and receives an HTTP response that a new entity in the *NewAnalysis* table has been successfully created.

Next, an engineer runs *Revit* and starts our plug-in, which automatically checks whether someone requested a new analysis for the project that the engineer has opened. If there is, the engineer is informed about this in a form of a text message as seen on Figure 4.6b. Upon clicking *Run analysis* button, the Revit model is compared with the multi-LOD meta-model. Consequently, a new entity in the *Feedback* database table, as well as a list of *Change_Suggestions* entities in the database table with the same name are created. In case the architect has requested suggestions for missing parameters and values, the engineer can create them. They are automatically sent to the server, to be written to the *Change_Suggestion* database.

new analysis.pdf

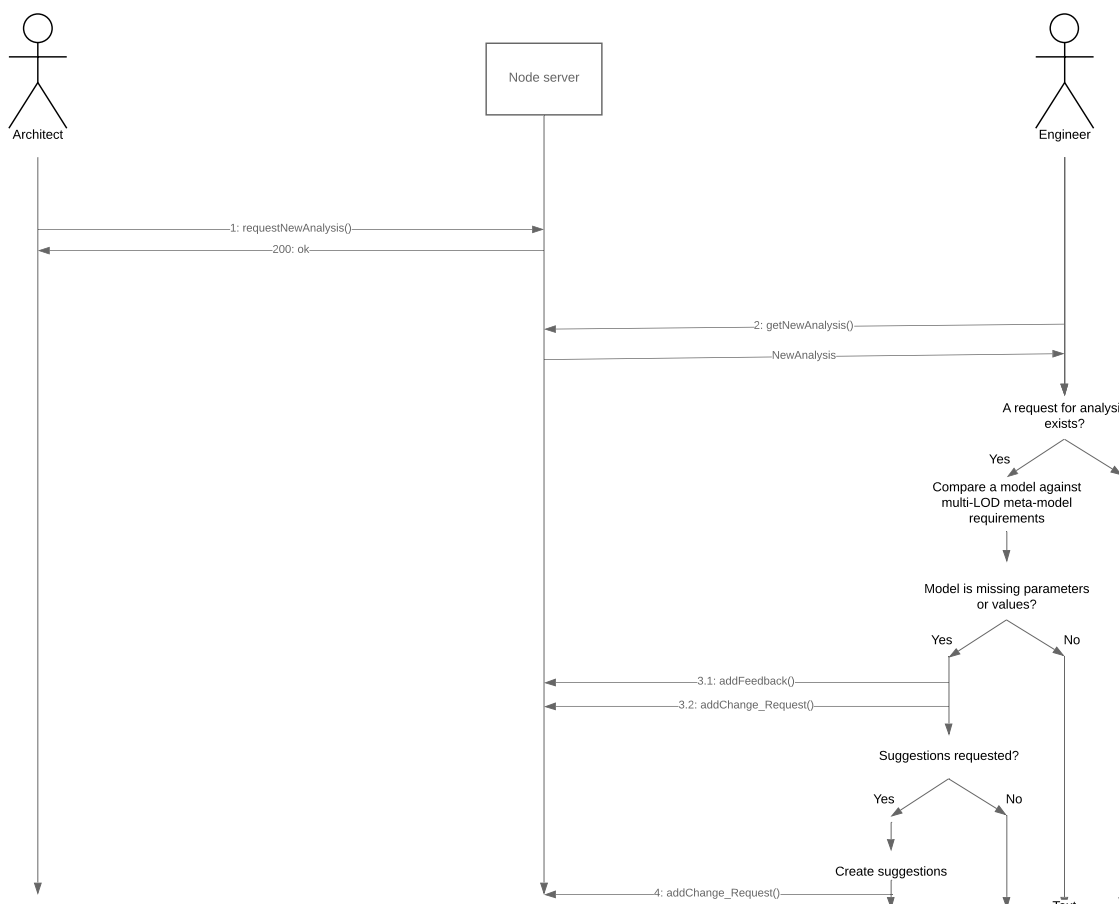


Figure 4.24 Requesting and running a new analysis

Next time the architect starts *Revit* and our plug-in, information about the project that the architect has opened is retrieved from the server. First, the plug-in checks whether there is a *NewAnalysis* request, as seen on Figure 4.25. If there is, an HTTP request for a related *Feedback* information is sent. Upon receiving it, all the related *Change_Suggestion* data is retrieved and a list of Tasks is created out of it, as

an extended version of *Change_Suggestion* data, with a purpose of being used locally, within the client application.

tasks-3.png

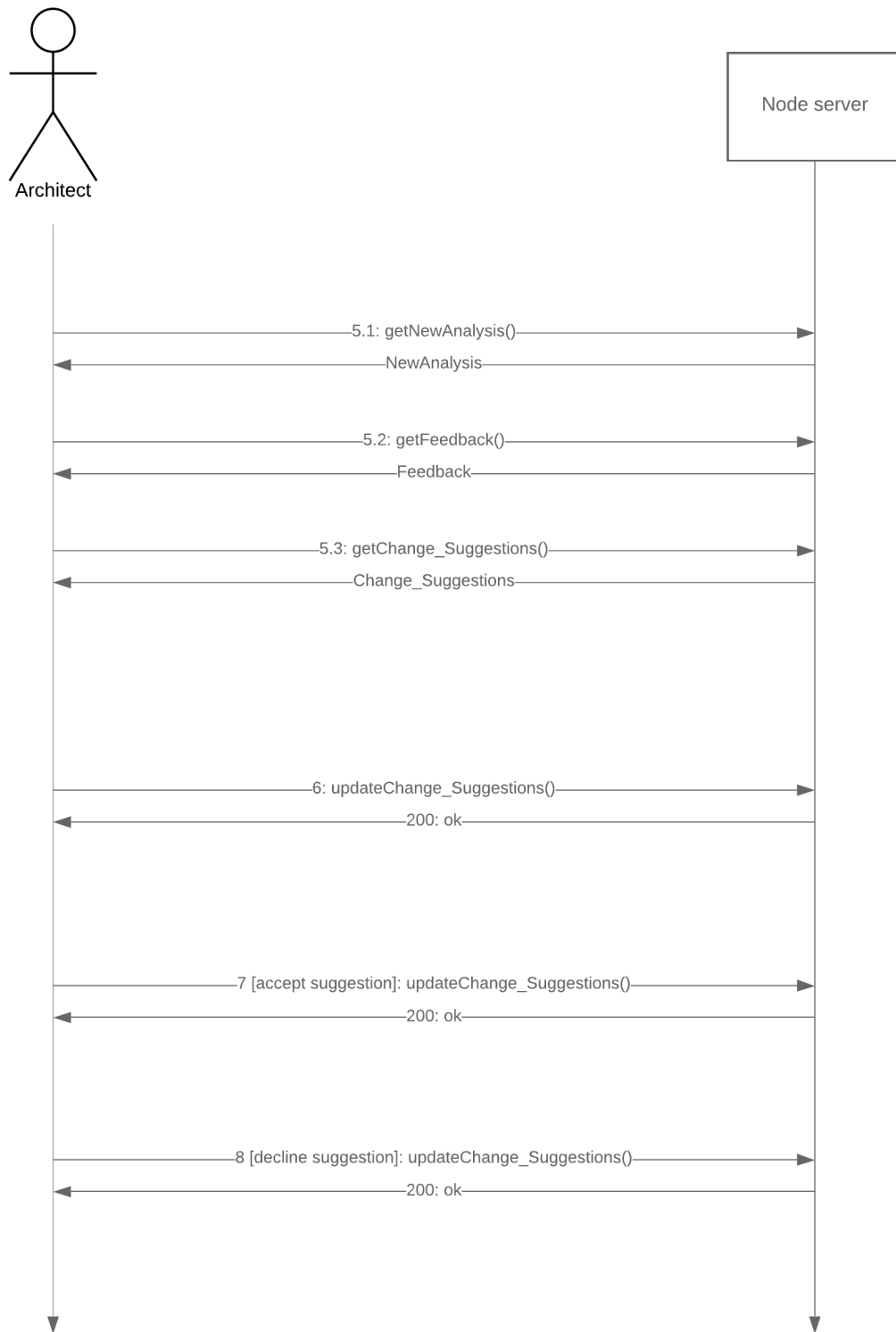


Figure 4.25 An architect solving tasks and accepting or declining suggested values

To either create a missing value or parameter or accept or decline an engineer's suggestion, the architect sends an HTTP request and receives a response after the value has been successfully updated in the database table.

4.7 Implementation

In C# development, a *project* groups classes, resources and various configuration files that logically belong together. Depending on a type, a *project* is compiled into a file with an .exe or a .dll extension, in other words into an executable application or a library.

Projects can be grouped under a *solution*. Even though it is theoretically possible to avoid using projects by putting everything into a single solution, it is usually a bad practise. Splitting a *solution* into multiple *projects* follows many good programming principles. For example, a project has a smaller code base than a solution, which makes it easier to maintain; also, it can be reused later. A solution should provide a high cohesion by grouping the related classes and methods into a single project, while allowing for loose coupling between projects.

The solution is split into five projects:

- MarijaRakicMasterThesis further described in section 4.7.1,
- MultiLODLib further described in section 4.7.13,
- MarijaRakicMasterThesis.DataTypes further described in section 4.7.14,
- MarijaRakicMasterThesis.Converters further described in section 4.7.15, and
- com.server.api further described in section 4.7.16.

Dependencies are defined per project. The aforementioned projects reference one another. Existing dependencies within this solution are shown in Figure 4.26.

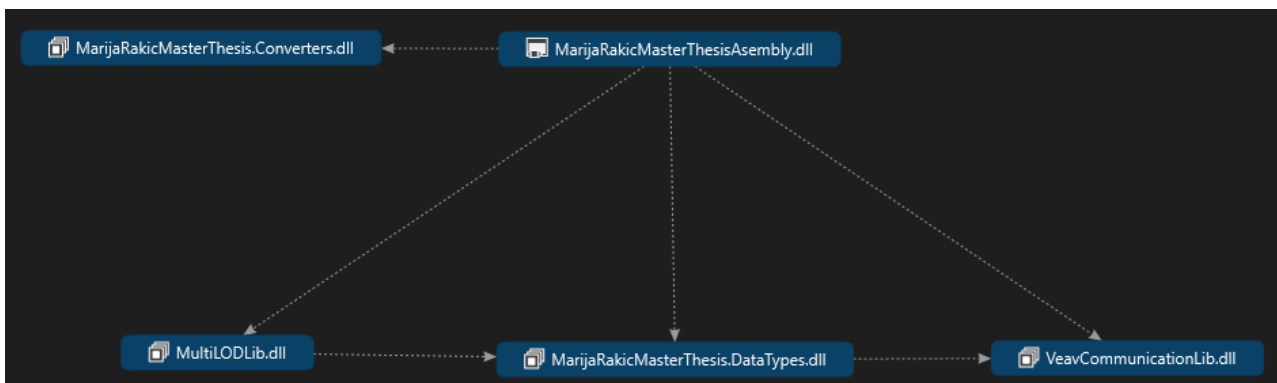


Figure 4.26 Solution dependencies

4.7.1 MarijaRakicMasterThesis

This project is split into several namespaces:

- `TUM.MasterThesis.MarijaRakic`, containing `ExternalApplication`, `ExternalCommand`, `RegisterPluginCommand` and `Singleton` classes, all further described in the upcoming sections,
- `TUM.MasterThesis.MarijaRakic.ExternalEventHandler`, further described in Section 4.7.10,
- `TUM.MasterThesis.MarijaRakic.Managers`, further described in Section 4.7.11,
- `TUM.MasterThesis.MarijaRakic.UserControl`, further described in Section 4.7.6,
- `TUM.MasterThesis.MarijaRakic.Utils`, further described in Section 4.7.12,
- `TUM.MasterThesis.MarijaRakic.ValidationRules`, further described in Section 4.7.7,
- `TUM.MasterThesis.MarijaRakic.View`, further described in Section 4.7.5,
- `TUM.MasterThesis.MarijaRakic.ViewModel`, further described in Section 4.7.8, and
- `TUM.MasterThesis.MarijaRakic.ViewModel.Commands`, further described in Section 4.7.9.

This project must reference two DLLs: `RevitAPI.dll` and `RevitAPIUI.dll`. The first one provides access to the general Revit API, and the second one provides access to the Revit API specialized in interacting with the Revit user interface. As a consequence, the application must be implemented using the 4.7 version of .NET framework 4.7, because Revit 2019 requires it.

Before doing a deep dive into the implementation specifics, I want to point out that I do not go into details of the entire code base. Also, code snippets that will be used in the remainder of this thesis will not be an exact copy of the running code; they will very often exclude the details that are not relevant to the point being explained at that moment.

4.7.2 IExternalApplication

An `ExternalApplication` class contains information on actions that are to be taken when Revit starts and shuts down. It implements the `IExternalApplication` interface, which is defined in the `Autodesk.Revit.UI` namespace and requires two methods to be provided: `OnStartup` and `OnShutdown`.

On Revit start, we want to create a new box under the *Add-ins* tab in which we will put a button that will start our plug-in. In Revit terms, this box is called `RibbonPanel` and we create it on the application object. When creating a panel, there are two options: a default one is to create a panel on the *Add-ins* tab, and the second one is to create a custom tab and add panel to it.

Listing 4.9 `OnStartup` method that creates a ribbon panel and a push button

```
public Result OnStartup(UIControlledApplication application)
{
    // Create a ribbon panel on a "Add-ins" tab and name it
    RibbonPanel ribbonPanel = application.CreateRibbonPanel(panelName);

    // Create a pushButton and add it to the ribbon panel
    CreatePushButtonOnRibbon(ribbonPanel);

    application.ControlledApplication.ApplicationInitialized += DockablePaneRegisters;

    return Result.Succeeded;
}
```

Next, we create a button that will run our plug-in when clicked. In Revit terms, this button is called `PushButton`. It is defined by its name, a description and a full name (with the namespace information) of the class that implements the `IExternalCommand` interface, as described in Section 4.7.3. We also provide a small and a large image for the newly created `PushButton`.

Listing 4.10 Creating of a push button on a given ribbon panel

```
private void CreatePushButtonOnRibbon(RibbonPanel ribbonPanel)
{
    // Create a pushButton and add it to the ribbon panel
    string assemblyName = Assembly.GetExecutingAssembly().Location;
    PushButton pushButton = ribbonPanel.AddItem(
        new PushButtonData(pushButtonDictionary["name"],
            pushButtonDictionary["text"],
            assemblyName,
            pushButtonDictionary["className"])) as PushButton;

    // Set the large image shown on button
    pushButton.LargeImage = ImageSourceForBitmap(Resources.push_button_32x32);

    // Set the small image which is used if command is moved to Quick Access Toolbar
    pushButton.Image = ImageSourceForBitmap(Resources.push_button_16x16);
}
```

Finally, we subscribe to the `ApplicationInitialized` event and when triggered, we invoke the `Execute` method of an instance of the `RegisterPluginCommand` class.

Listing 4.11 Execute method that runs when a user starts our plug-in

```
private void DockablePaneRegisters(object sender, ApplicationInitializedEventArgs args)
{
    RegisterPluginCommand registerPluginCommand = new RegisterPluginCommand();
    registerPluginCommand.Execute(
        new UIApplication(sender as Autodesk.Revit.ApplicationServices.Application));
}
```

This method is using an internal class `RegisterPluginCommand` that implements the `IExternalCommand` interface. Its main job is to set an instance of the `CompareToMultiLOD` class as the provider, i.e. a class that implements the `IDockablePaneProvider` interface, of the dockable pane.

Listing 4.12 A class that sets an instance of the `CompareToMultiLOD` class to be a provider of a dockable pane

```
internal class RegisterPluginCommand : IExternalCommand
{
    public Result Execute(ExternalCommandData commandData, ref string message, ElementSet
        elements)
    {
        return Execute(commandData.Application);
    }

    public Result Execute(UIApplication uiApplication)
    {
        CompareToMultiLOD managerPage = new CompareToMultiLOD();

        DockablePaneId dockablePaneID = new
            DockablePaneId(PaneIdentifiers.ManagerPaneIdentifier());
        uiApplication.RegisterDockablePane(dockablePaneID, "Compare to Multi LOD meta
            model", managerPage as IDockablePaneProvider);
    }
}
```

```

        return Result.Succeeded;
    }
}

```

4.7.3 IExternalCommand

Every Revit application has to have a class that implements the `IExternalCommand` interface. This interface defines an `Execute` method that needs to be provided, since it tells Revit what to do when a user starts a plug-in.

An input parameter `commandData` is “an `ExternalCommandData` object which contains reference to Application and View needed by external command”, as per the official documentation. We use it to extract the following objects and store them in `Singleton` for later use.

- `UIApplication` “represents an active session of the Autodesk Revit user interface, providing access to UI customization methods, events, the main window, and the active document”, as per the official documentation. We use `UIApplication` to e.g. dock a pane.
- `UIDocument` is “an object that represents an Autodesk Revit project opened in the Revit user interface”, as per the official documentation. We use `UIDocument` to e.g. color a selected Revit element in a given color.
- `Document` is “an object that represents an open Autodesk Revit project”, as per the official documentation. We use `Document` to e.g. fetch a `Material` or an `Element`, and to execute a `Transaction`.

Next, we open a dockable pane and let a user interact with it, as shown in Listing 4.13.

Listing 4.13 Execute method that runs when a user starts our plug-in

```

public Result Execute(ExternalCommandData commandData, ref string message, ElementSet
    elements)
{
    //Get application and document objects
    UIApplication uiapp = commandData.Application;
    Singleton.Instance.UIApplication = uiapp;
    Singleton.Instance.UIDocument = uiapp.ActiveUIDocument;
    Singleton.Instance.Document = uiapp.ActiveUIDocument.Document;

    // Open dockable pane
    DockablePaneManager.ShowOpenDockablePane(PaneIdentifiers.ManagerPaneIdentifier());

    return Result.Succeeded;
}

```

This method is also a good place to do some initial setup of the application, for example set a URL to the server, fetch information on the current user and the project, etc. This code is trivial, and therefore omitted from this code snippet.

To open a dockable pane, a static class `DockablePaneManager` implements one static method. This method accepts a pane’s unique identifier of type `Guid` as an input parameter and creates a `DockablePaneId` as an output. Finally, it creates an instance of a `DockablePane` and shows it to the user. Implementation of this class is as follows:

Listing 4.14 Creating and showing a dockable pane

```

public static class DockablePaneManager
{
    public static void ShowOpenDockablePane(Guid guid)

```

```

{
    DockablePaneId dockablePaneID = new DockablePaneId(guid);
    DockablePane dockablePane = UIApplication.GetDockablePane(dockablePaneID);
    dockablePane.Show();
}
}

```

4.7.4 Singleton

A singleton is a software design pattern. It provides assurance that there is only one instance of a class, which is achieved by internally invoking a private constructor the first time an instance of a singleton class is used.

It stores some values that are accessed from many different classes. For example, it stores information on the Document, UIDocument and UIApplication of the currently open Revit project. It also stores information on the current user and a Task the user is working on at the moment.

In addition, it caches some database tables, namely a list of ProductCatalog, a list of PropertyCatalog, a list of RulesProductProperty, a list of Rule_aLOD and a list of aLODX. The assumption is made that these values do not change frequently and that they will not change during a single use of our plug-in. Therefore, they are fetched from the database only once, and after that their local copies are used.

Unfortunately, not all the initial architectural design decisions turned out to be good ones. Very often, a communication between two views, and consequently ViewModels is needed. There are third-party frameworks, e.g. MVVM light⁸ and Prism⁹, which provide this functionality out-of-the-box. However, due to the lack of experience in the WPF development, i did not see the need for these frameworks right from the beginning of the development process. Later on, then the need was obvious, it was no longer easy to include them. The project had a significant code base that was modeled by the initial architectural design decisions, which was not compatible with the aforementioned frameworks. Hence, a way around the problem was to store information on an instance of the CompareToMultiLODViewModel class in Singleton and use it when it is needed to invoke its methods from another class. This is not an ideal solution, because it introduces unnecessary high coupling when compared to e.g. the publish-subscribe design pattern that the aforesaid third-party frameworks use.

4.7.5 Views

In this section, I describe the development of the user interface. First, I list some standard controls that have been heavily used, and then I go into some interesting details about each class.

The application is built using a lot of standard controls, such as:

- <Button />, which allows a user to take some action,
- <RadioButton />, which allow a user to choose one of multiple options,
- <CheckBox />, which allows a user to choose none or some of multiple options,
- <Grid />, which is used to organize other controls into rows and columns,
- <StackPanel />, which “Arranges child elements into a single line which can be oriented horizontally or vertically”, as per the official documentation,
- <GroupBox />, which “creates a container that has a border and a header for user interface (UI) content”, as per the official documentation,
- <Label />, which displays textual data,

⁸<http://www.mvvmlight.net>

⁹<http://prismlibrary.github.io>

- `<ListView />`, which displays a list of data items, and
- `<TreeView />`, which “displays hierarchical data in a tree structure that has items that can expand and collapse”.

The application consists of four independent windows:

- `CompareToMultiLOD`, our main page that inherits from the class `Page` and implements interfaces `IDisposable` and `IDockablePaneProvider`, and is shown in Figures 4.7 and 4.8,
- `AnalysisWindow`, which inherits from the class `Window` and implements the interface `ICloseable`, and is shown in Figure 4.5,
- `UserWindow`, which inherits from the class `Window` and implements the interface `ICloseable`, and is shown in Figure 4.3, and
- `ValueWindow`, which inherits from the class `Window`, and is shown in Figures 4.9, 4.10a, and 4.10b. This window uses a custom user control `MaterialsListView`, described in Section 4.7.6.

In the remainder of this section, I focus on the interesting aspects of each of these classes.

The home page and custom user controls

The home page utilizes four user controls that will be described in Section 4.7.6. They are used like any other native control, with opening and closing tags and an option to pass input parameters:

Listing 4.15 Usage of the custom user controls

```
<local:TasksListView MissingValues="{Binding IssuesCollectionViewSource.View}" />
<local:TasksTreeView MissingValues="{Binding MissingValuesPerElementList}" />
<local:PieChart />
<local:StackedColumns />
```

ValueWindow and xceed toolkit DecimalUpDown control

`ValueWindow` uses a third-party library `Extended WPF Toolkit™` explained in Section 4.2 to implement a field for inputting numerical values.

The library is included `xmlns:xctk="http://schemas.xceed.com/wpf/xaml/toolkit"` and used as follows:

Listing 4.16 Usage of the DecimalUpDown control

```
<xctk:DecimalUpDown Value="{Binding ValueDouble}"
    Name="myUpDownControl"
    Height="40"
    Visibility="{Binding Path=ShowReal, Converter={StaticResource Converter}}" />
```

We rarely work with just one XAML library per XAML file. Simply by creating a new `Window`, for example, *Microsoft Visual Studio* automatically includes four different namespaces for us:

Listing 4.17 Microsoft Visual Studio automatically includes some namespaces when creating a new Window

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

When including more than one library and consequently using more than one namespace per a single XAML file, names might conflict. To solve this problem, we assign a prefix to a library and use its controls with this prefix, which assure unique names. `xmlns` denotes an XML namespace and it is required to define a prefix for a library being included. In the previous example, we give a `xctk` prefix to the Extended WPF Toolkit™ library and we use a `DecimalUpDown` control with this prefix.

The first namespace not having a prefix makes the code significantly cleaner, since the most commonly used controls come from this namespace. We type e.g. `<Button>` instead of `<{prefix}:Button>`. We could use prefix as well, but there is no need for that.

AnalysisWindow and DatePicker

One interesting feature of the `AnalysisWindow` class is `DatePicker`. We specify all the past dates to be grayed out by defining the `CalendarDateRange` value of the `BlackoutDates` property. After the user has chosen the date, we apply the `ValidationRules`, further described in the 4.7.7, to make sure the user does not unintentionally input some incorrect value that is not a date.

Since only future dates are valid dates, `DatePicker` requires

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

to fetch information on the earliest possible date in the system as the start date and on the present date as the end date.

Listing 4.18 Usage of the DatePicker

```
<DatePicker x:Name="FutureDatePicker"
            Width="100"
            Margin="0, 0, 0, 20"
            materialDesign:HintAssist.Hint="Deadline">

    <DatePicker.BlackoutDates>
        <CalendarDateRange Start="{x:Static sys:DateTime.MinValue}"
                          End="{x:Static sys:DateTime.Today}" />
    </DatePicker.BlackoutDates>

    <DatePicker.SelectedDate>
        <Binding Path="Deadline"
                UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <domain:FutureDateValidationRule ValidatesOnTargetUpdated="True"
                xmlns:domain="clr-namespace:TUM.MasterThesis.MarijaRakic.ValidationRules" />
            </Binding.ValidationRules>
        </Binding>
    </DatePicker.SelectedDate>
</DatePicker>
```

Interface ICloseable

This namespace also contains one interface, called `ICloseable`, with the following definition:

Listing 4.19 Interface ICloseable definition

```
public interface ICloseable
{
    void Close();
}
```

The purpose of this interface is to allow for easy closure of `AnalysisWindow` and `UserWindow` after they have been shown on the home page.

We pass the Window instance from the XAML file as a command parameter in the following way:

Listing 4.20 Passing Window instance as a command parameter

```
<Window x:Name="MyWindow">

    <Button Command="{Binding MyCommand}"
            CommandParameter="{Binding ElementName=MyWindow}" />

</Window>
```

Then, we cast the method's argument to `ICloseable`:

Listing 4.21 Casting method's argument to `ICloseable`

```
public void Execute(object parameter)
{
    viewModel.ExecuteMyCommand(parameter as ICloseable);
}
```

Finally, we initiate closing the window within the `ExecuteMyCommand` method.

Listing 4.22 Invoking the `Close` method on an instance of `ICloseable`

```
internal void ExecuteMyCommand(ICloseable window)
{
    // execute some logic here; actual code irrelevant for this example

    CloseWindow(window);
}

private void CloseWindow(ICloseable window)
{
    if (window != null)
    {
        window.Close();
    }
}
```

4.7.6 User Controls

User controls provide a way to split the code needed to implement the graphical interface of the application into a couple of smaller classes, making the code easier to reuse and maintain.

In this application, there are five user controls:

- `MaterialsListView`, which groups the controls to show the list of materials, search it by name and filter it by material class, as shown in Figure 4.10b,
- `PieChart`, which is implemented using the `LiveCharts` library (explained in Section 4.2). It presents the ratio of open tasks per Revit element, as shown in Figure 4.22a,
- `StackedColumns`, which is implemented using the `LiveCharts` library (explained in Section 4.2). It presents the ratio of open and solved tasks per Revit element, for all Revit elements of the currently open model, as shown in Figure 4.22b,
- `TasksListView`, which presents the list of tasks in a table-like form, as shown in Figures 4.7 and 4.8, and

- `TasksTreeView`, which presents the list of tasks in a hierarchical way, grouped by the Revit element that they are related to, as seen in Figure 4.5.14.

All of them inherit from the `UserControl` class and have a very basic constructor where only the `InitializeComponent()` method is invoked.

Similar to Views described in Section 4.7.5, these user controls consist of many common XAML controls. In addition, `TasksListView` and `TasksTreeView` have some interesting features that will be further described in the remainder of this section.

Passing an argument to a custom user control

`TasksListView` and `TasksTreeView` accept an input parameter under the name `MissingValues`. By not hardcoding the concrete array of data to the user control, we get a loosely coupled component that is, consequently, easy to reuse in some future projects.

Data is bound to the user control in the following way:

Listing 4.23 Binding a value to the user control's `MissingValues` property

```
<local:TasksListView MissingValues="{Binding IssuesCollectionViewSource.View}" />
<local:TasksTreeView MissingValues="{Binding MissingValuesPerElementList}" />
```

To achieve this, the following property and fields are implemented in the `TasksTreeView`'s code-behind:

Listing 4.24 Implementation of the binding parameter to the custom user control

```
public ObservableCollection<MissingValuesPerElement> MissingValues
{
    get { return
        (ObservableCollection<MissingValuesPerElement>)GetValue(MissingValuesProperty); }
    set { SetValue(MissingValuesProperty, value); }
}
```

```
// Using a DependencyProperty as the backing store for MyProperty. This enables animation,
// styling, binding, etc...
```

```
public static readonly DependencyProperty MissingValuesProperty =
    DependencyProperty.Register("MissingValues",
        typeof(ObservableCollection<MissingValuesPerElement>),
        typeof(TasksTreeView),
        new PropertyMetadata(new ObservableCollection<MissingValuesPerElement>()));
```

`MissingValuesPerElement` models data presented in the tree form, and is further described in Section 4.7.14.

Implementation of the same feature in the `TasksListView` control is different only in the data type of the input value. Instead of `ObservableCollection<MissingValuesPerElement>`, `TasksListView` accepts input parameter of the data type `ICollectionView`.

Visibility toggling and boolean to visibility converters

There are multiple ways to change the visibility of the XAML element. A built-in method exists for the case when visibility is defined by a single variable. It should be included as a resource of a certain page, window or user control like this:

Listing 4.25 Including `BooleanToVisibilityConverter` as a user control resource

```
<UserControl.Resources>
    <BooleanToVisibilityConverter x:Key="Converter" />
</UserControl.Resources>
```

Later on, let's say that we want to change the visibility of a list view. The converter is used as follows:

Listing 4.26 Using BooleanToVisibilityConverter to set the visibility of a ListView

```
<ListView Visibility="{Binding Path=ShowListView, Converter={StaticResource Converter}}">
```

Sometimes we need to set visibility based on the result of some boolean expression that includes two or more variables. There are two ways to achieve this. One is to use MultiDataTrigger as shown in Listing 4.27, where we want to show StackPanel if the user is an architect AND they are viewing open tasks OR if the user is an engineer AND they are viewing open tasks AND the architect has requested options to be added to the results of the analysis:

Listing 4.27 Usage of MultiDataTrigger

```
<StackPanel.Style>
  <Style>
    <Setter Property="Control.Visibility"
      Value="Hidden" />
    <Style.Triggers>
      <MultiDataTrigger>
        <MultiDataTrigger.Conditions>
          <Condition Binding="{Binding ElementName=ListViewUC, Path=IsOpenTasks}"
            Value="true" />
          <Condition Binding="{Binding ElementName=ListViewUC, Path=IsArchitect}"
            Value="true" />
        </MultiDataTrigger.Conditions>
        <Setter Property="Control.Visibility"
          Value="Visible" />
      </MultiDataTrigger>
      <MultiDataTrigger>
        <MultiDataTrigger.Conditions>
          <Condition Binding="{Binding ElementName=ListViewUC, Path=IsOpenTasks}"
            Value="true" />
          <Condition Binding="{Binding ElementName=ListViewUC, Path=IsEngineer}"
            Value="true" />
          <Condition Binding="{Binding ElementName=ListViewUC, Path=OptionsWanted}"
            Value="true" />
        </MultiDataTrigger.Conditions>
        <Setter Property="Control.Visibility"
          Value="Visible" />
      </MultiDataTrigger>
    </Style.Triggers>
  </Style>
</StackPanel.Style>
```

Another way to achieve the same goal is to create a converter class that implements the IMultiValueConverter interface. An example is the MultiValueConverter class, whose implementation is explained in Section 4.7.15. This converter sets visibility of the XAML element when both values are true. In the following example, this converter sets StackPanel's visibility when both values IsCheckedPendingTasks and IsArchitect are true.

Listing 4.28 Usage of the MultiValueConverter class

```
<StackPanel.Visibility>
  <MultiBinding Converter="{StaticResource MultiValueConverter}">
    <Binding ElementName="ListViewUC"
      Path="IsCheckedPendingTasks" />
    <Binding ElementName="ListViewUC"
      Path="IsArchitect" />
  </MultiBinding>
</StackPanel.Visibility>
```

```
</MultiBinding>
</StackPanel.Visibility>
```

Similar to *LiveCharts* and *Material Design In XAML's* loading problem described in Section 4.2, there is a problem with loading of the `MultiValueConverter` class as well. To circumvent this, the following line of code should be added to the code-behind of the `TasksListView` class, in the constructor, right before initializing the component:

```
MultiValueConverter multiValueConverter = new MultiValueConverter();
```

4.7.7 Validation rules

This namespace consists of two classes that inherit from the `ValidationRule` class:

- `NotEmptyValidationRule`, which checks whether a user has input a mandatory value or not. Code of this class can be seen in Listing 4.29 and
- `FutureDateValidationRule`, which checks whether the date that has been input is a future date. Code of this class can be seen in Listing 4.30.

Listing 4.29 Implementation of the `NotEmptyValidationRule` class

```
public class NotEmptyValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        return string.IsNullOrEmpty((value ?? "").ToString())
            ? new ValidationResult(false, "Field is required.")
            : ValidationResult.ValidResult;
    }
}
```

Listing 4.30 Implementation of the `FutureDateValidationRule` class

```
public class FutureDateValidationRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo cultureInfo)
    {
        if (value == null)
        {
            return new ValidationResult(false, null);
        }

        if (!DateTime.TryParse((value ?? "").ToString(),
            CultureInfo.CurrentCulture,
            DateTimeStyles.AssumeLocal | DateTimeStyles.AllowWhiteSpaces,
            out DateTime time))
        {
            return new ValidationResult(false, "Invalid date");
        }

        return time.Date <= DateTime.Now.Date ?
            new ValidationResult(false, "Future date required")
            : ValidationResult.ValidResult;
    }
}
```

4.7.8 ViewModels

Our application consists of four ViewModel classes:

- `AnalysisViewModel`, which provides logic for an architect to request a new analysis from an engineer, as shown in Figure 4.5a,
- `CompareToMultiLODViewModel`, which provides majority of logic that enables a user to take action in a process of solving a tasks,
- `UserViewModel`, which provides logic to choose a user's account or change the current user, as shown in Figures 4.3 and 4.23 respectively, and
- `ValueViewModel`, which provides logic for an engineer to create suggestions and an architect to input values for the missing parameter or parameter's missing value, as shown in Figures 4.14a, 4.9a and 4.12, respectively.

All these classes implement the `INotifyPropertyChanged` interface in order to notify a view when the data changes. The related code is grouped under the `region` in each of these classes:

Listing 4.31 Implementation of the `INotifyPropertyChanged` interface

```
#region INotifyPropertyChanged

public event PropertyChangedEventHandler PropertyChanged;

private void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}

#endregion INotifyPropertyChanged
```

As with the View in Section 4.7.5 and User controls in Section 4.7.6, I describe some interesting features of these classes in the remainder of this section.

UserViewModel

This class is a `ViewModel` for the `UserWindow`. It is responsible for selecting a user from a list of predefined users in the system the first time a person runs the application, or changing the user later.

As explained in Section 4.5.17, information on the user who last used the application is stored locally. To achieve this, we permanently persist the user's name. The first step to achieve this is to access the project's settings by right clicking on the project file, and then choosing the last option called *Properties*. Then, a table with columns *Name*, *Type*, *Scope* and *Value* appears under the *Settings* tab. In the first empty row, we can define the name, type and scope of the variable we want to store.

In this project, two variables are defined and persisted: `MultiLODpath` and `Name`. Code snippet 4.32 shows how to store the `Name` variable.

Listing 4.32 Storing the `Name` variable in Settings

```
Properties.Settings.Default["Name"] = value;
Properties.Settings.Default.Save();
```

Accessing the variable's value from Settings is as easy as accessing an element of an array. Upon retrieving it, it should be casted to an appropriate data type. The following example shows retrieving the `Name` variable and casting it to string:

Listing 4.33 Retrieving and casting the Name variable

```
string name = (string)Properties.Settings.Default["Name"];
```

ValueViewModel

This class is a ViewModel for the ValueWindow. It is responsible for handling different input types of the missing value - numerical, boolean and material, as explained in Section 4.5.7, as well as for searching and filtering the list of materials, as explained in the same section. Behind the scenes, filtering an array of materials by name is implemented using a single command as shown in Listing 4.34. Assigning null value to the View.Filter variable means that the array will not be filtered, in other words all array elements will be presented to the user.

Filtering by material class is implemented in a similar way, since the same data model MaterialWithImage contains information on both name (used in the upper example) and material class information, accessible under the MaterialClass parameter.

Listing 4.34 Filtering an array by name

```
View.Filter = string.IsNullOrEmpty(value)
    ? null
    : new Predicate<object>(o => ((MaterialWithImage)o).Material.Name.Contains
        (value, StringComparison.OrdinalIgnoreCase));
```

ValueViewModel raises AddParametersMissingValueExternalEvent and

AddMissingParameterExternalEvent, further explained in Section 4.7.10, when an architect clicks the Add button.

Whenever possible, I tried offering the same options to the Revit users that they are used to having while working with the native Revit features. Therefore, I wanted to display the material's image next to the material's name, in the same way Revit does it. For this reason, ValueViewModel implements a search for the material's image. Even though it sounds like a basic feature, unfortunately the Revit API does not offer a direct way of accessing it. Hence, a code suggested on the Revit Autodesk forum¹⁰ was adjusted to meet the needs of the application and used to implement the aforementioned feature. Its final version is as follows:

Listing 4.35 Retrieving of the material's image

```
private string GetImagePath(ElementId appearanceAssetId)
{
    AppearanceAssetElement aae = Document.GetElement(appearanceAssetId) as
        AppearanceAssetElement;
    if (null != aae)
    {
        Asset asset = aae.GetRenderingAsset();
        if (null != asset)
        {
            if (asset[SchemaCommon.Thumbnail] is AssetPropertyString ap)
            {
                string path = ap.Value;

                if (path.StartsWith("Mats") || path.StartsWith("Maps")) // Revit default
                {
                    path = path.Insert(0, @"C:\Program Files (x86)\Common Files\Autodesk
                        Shared\Materials\2019\assetlibrary_base.fbm\ ");
                }
            }
        }
    }
}
```

¹⁰<https://forums.autodesk.com/t5/revit-api-forum/get-an-up-to-date-material-preview-image-file-path/td-p/7731827>

```

else if (path.StartsWith("material", true,
    System.Globalization.CultureInfo.CurrentCulture))
{
    path = path.Insert(0, @"%tmp%\ ");
}

if (!string.IsNullOrEmpty(path) && File.Exists(path))
{
    return path;
}
}
}
}
return null;
}

```

AnalysisViewModel

This class is a ViewModel for the AnalysisWindow. It is responsible for collecting all the information that the architect has specified, i.e. an engineer responsible for running the analysis, whether options are wanted, the deadline for running the analysis, and then sending this request to the server.

As explained in Sections 4.5.3 and 4.7.5, inputting the date that specifies the Deadline for running the analysis is mandatory and it must be a future date. This is achieved completely through the XAML validation rules `NotEmptyValidationRule` and `FutureDateValidationRule` described in Section 4.7.7, so no additional code is required in the ViewModel.

CompareToMultiLODViewModel

This class is a ViewModel for the CompareToMultiLOD, in other words the home page. Since it is responsible for many different features, it quickly grew in size, so it was divided into two partial classes in two different files: `CompareToMultiLODViewModel.cs` and `CompareToMultiLODViewModel.Commands.cs`. Unfortunately, this only masks the fact that the class is "crowded" with code and features, but it undoubtedly increases readability and makes the maintenance easier.

A very nice way of grouping those two files together is achieved by editing .csproj file:

Listing 4.36 Grouping two CompareToMultiLODViewModel partial classes

```

<Compile Include="ViewModel\CompareToMultiLODViewModel.Commands.cs">
  <DependentUpon>CompareToMultiLODViewModel.cs</DependentUpon>
</Compile>

```

The result of this action can be seen in Figure 4.27:

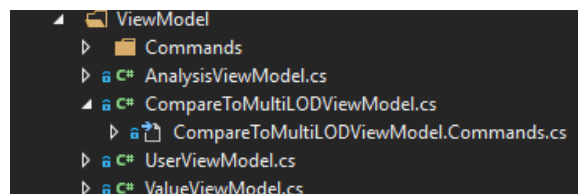


Figure 4.27 Grouping two CompareToMultiLODViewModel partial classes

In order to verify whether the building model has all the elements that are required to successfully run the analysis, an engineer invokes a `Run` method from the `Analyser` class of the `MultiLODLib` project (further described in Section 4.7.13) in the following way:

```

Analyser analyser = new Analyser(Properties.Settings.Default["MultiLODpath"].ToString(),
                                MappingFilePath, Document, CurrentUser);
openTasks = analyser.Run();

```

This ViewModel stores information on all tasks for the current project in a variable called `allTasks` declared as follows:

```

private ObservableCollection<Task> allTasks;
Open, pending and solved tasks are calculated by filtering allTasks by task's Status value:

```

Listing 4.37 example is here

```

openTasks = new ObservableCollection<Task>(
    allTasks.Where(x => x.Status == Status.Open));
pendingTasks = new ObservableCollection<Task>(
    allTasks.Where(x => x.Status == Status.Pending));
solvedTasks = new ObservableCollection<Task>(
    allTasks.Where(x => x.Status == Status.Solved));

```

Depending on the option chosen under the *Task status* box, a different array of tasks is assigned to the underlying `CollectionViewSource` as follows:

Listing 4.38 Assign a different subset of tasks to be presented to a user

```

switch (status)
{
    case Status.Open:
        IssuesCollectionViewSource.Source = openTasks;
        break;

    case Status.Pending:
        IssuesCollectionViewSource.Source = pendingTasks;
        break;

    case Status.Solved:
        IssuesCollectionViewSource.Source = solvedTasks;
        break;

    default:
        break;
}

```

`CompareToMultiLODViewModel` raises `SelectElementExternalEvent` and `DeselectElementExternalEvent` when a user clicks on a task in the list. This results in selecting and deselecting Revit elements, also recognized by coloring the selected element in a randomly chosen color assigned to that element, as shown in Figure 4.21.

To show a new window, the following code is used:

Listing 4.39 Showing a new window

```

// Workaround the fact that MainWindowHandle returns type 'IntPtr', but Owner requires type
// 'Window'
HwndSource hwndSource = HwndSource.FromHwnd(UIApplication.MainWindowHandle);
Window wnd = hwndSource.RootVisual as Window;
if (wnd != null)
{
    ValueWindow.Owner = wnd;
    //ValueWindow.ShowInTaskbar = false;
    ValueWindow.Show();
}

```



```
}
```

This code makes sure that the child window behaves in the same way as the parent window. If the parent window, in our case the *Revit* application, is minimized, then the child window is minimized as well. When *Revit* is put back into the focus, so is the child window. The line of code that is commented out specifies whether the child window has an independent icon in the Window's Taskbar or not. With the current setting of the application, it has the separate icon; uncommenting this line would result in only one icon existing.

When an engineer wants to choose a MultiLOD file, as explained in Section 4.5.4, they click the directory button in the upper right corner of the home page. Opening of the file from the local system is achieved in the following code snippet that was taken from the official Microsoft documentation on Dialog Boxes ¹¹:

Listing 4.40 Opening of the file from the local system

```
// Code taken from:
internal void ExecuteOpenFileDialogCommand()
{
    // Configure open file dialog box
    Microsoft.Win32.OpenFileDialog dlg = new Microsoft.Win32.OpenFileDialog();
    dlg.Filter = "JSON documents (.json)|*.json"; // Filter files by extension

    // Show open file dialog box
    bool? result = dlg.ShowDialog();

    // Process open file dialog box results
    if (result == true)
    {
        // Open document
        Properties.Settings.Default[MultiLODpath] = dlg.FileName;
        Properties.Settings.Default.Save(); // Saves settings in application configuration
        file
    }
}
```

4.7.9 Commands

Every button is associated with an instance of a class that implements the `ICommand` interface. These internal classes are part of the `TUM.MasterThesis.MarijaRakic.ViewModel.Commands` namespace. They all have a very similar code, since they do not hold any logic themselves; all the logic is contained in the associated `ViewModel`.

`AddValueCommand` is an example of these classes: it has one private field that contains information on the `ViewModel` that is passed as an argument of the constructor of the class. Next, we specify that the class `CanExecute` always. Finally, when a user clicks a button, the `Execute` method is invoked, where we pass information to the `ViewModel` for the further processing.

Listing 4.41 `AddValueCommand` as an example of the class that implements the `ICommand` interface

```
internal class AddValueCommand : ICommand
{
    private CompareToMultiLODViewModel viewModel;

    public AddValueCommand(CompareToMultiLODViewModel viewModel)
    {
        this.viewModel = viewModel;
    }
}
```

¹¹https://docs.microsoft.com/en-us/dotnet/framework/wpf/app-development/dialog-boxes-overview#Common_Dialogs

```

}

#region ICommand

public event EventHandler CanExecuteChanged;

public bool CanExecute(object parameter)
{
    return true;
}

public void Execute(object parameter)
{
    viewModel.ExecuteAddValueCommand(parameter);
}

#endregion ICommand
}

```

4.7.10 ExternalEventHandler

Every time a new modeless dialogue is presented to the user, we are no longer on the Revit thread, the only thread on which it is possible to make API calls. Obviously, we want to be able to make changes to the Revit model even from the modeless dialogues, and Revit API provides us with External Events for this purpose. In practise, this means raising an event from a non-Revit thread and implementing an event handler that will be executed on the Revit thread where changes to the building model are allowed.

This namespace contains four classes that make changes to the building model:

- AddMissingParameterExternalEventHandler,
- AddParametersMissingValueExternalEventHandler,
- DeselectElementExternalEventHandler, and
- SelectElementExternalEventHandler.

Each of these classes implements the `IExternalEventHandler` interface, and consequently the following two methods:

- `public void Execute(UIApplication app)`, which “is called to handle the external event”, as per the official documentation, and
- `public string GetName()`, which returns “String identification of the event handler”, as per the official documentation.

Common for all these classes is that they execute a transaction on the Revit element. When a transaction is committed, the associated document regenerates automatically.

SelectElementExternalEventHandler and DeselectElementExternalEventHandler

`SelectElementExternalEventHandler` and `DeselectElementExternalEventHandler` use the fairly similar code to color the selected Revit Element in a specified color, as shown in Listing 4.42 and clear that color when the element is deselected, as shown in Listing 4.43, respectively.

Listing 4.42 Set a color of the OverrideGraphicSettings

```
OverrideGraphicSettings ogs = new OverrideGraphicSettings();
System.Windows.Media.Color rgb =
    (System.Windows.Media.Color)ColorConverter.ConvertFromString(color);
Autodesk.Revit.DB.Color color = new Autodesk.Revit.DB.Color(rgb.R, rgb.G, rgb.B);
Element solidFill = new FilteredElementCollector(Document)
    .OfClass(typeof(FillPatternElement))
    .Where(q => q.Name.Contains("Solid"))
    .First();

ogs.SetSurfaceForegroundPatternId(solidFill.Id);
ogs.SetSurfaceForegroundPatternColor(color);
```

Listing 4.43 Clear a color of the OverrideGraphicSettings

```
OverrideGraphicSettings ogs = new OverrideGraphicSettings();
```

A final step is to execute a transaction on the current document. This step is completely the same for both selecting and deselecting a Revit element.

Listing 4.44 Execute a transaction

```
using (Transaction t = new Transaction(Document, transactionName))
{
    t.Start();

    try
    {
        // elementId = Id of element you wish to highlight
        UIDocument.ActiveView.SetElementOverrides(elementId, ogs);
    }
    catch (Exception ex)
    {
        TaskDialog.Show("Exception", ex.ToString());
    }
    UIDocument.RefreshActiveView();
    t.Commit();
}
```

AddParametersMissingValueExternalEventHandler

AddParametersMissingValueExternalEventHandler simply sets the value of the specified element's parameter. Information on the parameter whose value should be set is stored in the Singleton.Instance.Task.Parameter property. Since the Set() method accepts string, int, double or ElementId as input parameter, we first convert the string input parameter.

Listing 4.45 A transaction to set the parameter's value

```
using (Transaction transaction = new Transaction(Document, "update_property_value"))
{
    transaction.Start();
    try
    {
        // Set() accepts string, int, double or ElementId as input parameter
        switch (value)
        {
            case string stringValue:
```

```

        parameter.Set(stringValue);
        break;

    case int intValue:
        parameter.Set(intValue);
        break;

    case bool boolValue:
        parameter.Set(boolValue ? 1 : 0);
        break;

    case double doubleValue:
        parameter.Set(doubleValue);
        break;

    case ElementId elementIdValue:
        // Set Material id
        parameter.Set(elementIdValue);
        break;

    default:
        break;
}
}
catch (Exception ex)
{
    System.Diagnostics.Debug.WriteLine("An error occurred while executing a transaction
        update_property_value");
}
transaction.Commit();
}

```

AddMissingParameterExternalEventHandler

In case of the missing parameter, a situation is slightly more complicated. Implementation of this feature comes from the official *The Revit SDK samples (RoomSchedule)*, which was adapted to the needs of this plug-in.

First, we need to check whether a shared parameter with a given name exists by using the `SharedParameterExists` method from the `Utils` namespace, as described in Section 4.7.12.

If the shared parameter exists, we retrieve it by its name and continue with assigning a value to it by raising `AddParametersMissingValueExternalEvent`. If the parameter does not exist, we need to create it and execute a transaction to update the Revit model. For this, we use the following method:

Listing 4.46 A transaction to create a new shared parameter

```

// create shared parameter file
string modulePath = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
string paramFile = modulePath + "\\MultiLODSharedParameters.txt";
if (File.Exists(paramFile))
{
    File.Delete(paramFile);
}
FileStream fs = File.Create(paramFile);
fs.Close();

// cache application handle
Application revitApp = UIApplication.Application;

```

```

// prepare shared parameter file
UIApplication.Application.SharedParametersFilename = paramFile;

// open shared parameter file
DefinitionFile parafile = revitApp.OpenSharedParameterFile();

// create a group
DefinitionGroup apiGroup = parafile.Groups.Create("Compare to MultiLOD");

// create a visible {name specified in Singleton} of type {specified in Singleton}
ExternalDefinitionCreationOptions ExternalDefinitionCreationOptions =
    new ExternalDefinitionCreationOptions(
        Task.ProjectParameter.SharedParameter.Name,
        Task.ProjectParameter.SharedParameter.ParameterType);
Definition sharedParamDefinition =
    apiGroup.Definitions.Create(ExternalDefinitionCreationOptions);

// get category { specified in Singleton}
Category category = Document.Settings.Categories.get_Item(
    Task.ProjectParameter.SharedParameter.BuiltInCategory);
CategorySet categories = revitApp.Create.NewCategorySet();
categories.Insert(category);

// insert the new parameter
InstanceBinding binding = revitApp.Create.NewInstanceBinding(categories);

using (Transaction transaction = new Transaction(Document, "Create shared parameter"))
{
    transaction.Start();
    try
    {
        Document.ParameterBindings.Insert(sharedParamDefinition, binding,
            Task.ProjectParameter.BuiltInParameterGroup);
    }
    catch
    {
        System.Diagnostics.Debug.WriteLine("An error occurred while executing a transaction
            'Create shared parameter'");
    }
    transaction.Commit();
}

```

4.7.11 Managers

The `TUM.MasterThesis.MarijaRakic.Managers` namespace contains three classes. Their common characteristic is that they initialize some data in a very specific format required by a WPF control, e.g. a chart library. Those three classes are:

- `StackedColumnsManager`, a static class that transforms a list of `Tasks` to an instance of the `SeriesCollection` class, defined in the `LiveCharts` namespace. This class also returns an array of strings, each element of the array representing a Revit element's name to be a label assigned to a column, as shown in Figure 4.22b.
- `PieChartManager`, a static class that transforms an `IEnumerable` of type `MissingValuesPerElement` to an instance of the `SeriesCollection` class, defined in the `LiveCharts` namespace.

- GroupParameterUnderManager, a static class that initializes a list of GroupParameterUnder by mapping a list of existing BuiltInParameterGroup into their human-readable versions. As discussed in Section 4.7.14, we need a way to fetch a human-readable equivalent of the BuiltInParameterGroup, which is achieved in the following way:

Listing 4.47 Fetch a human-readable equivalent of all the BuiltInParameterGroup existing in the project

```

public static IList<GroupParameterUnder> InitializeGroupParametersUnder()
{
    IList<GroupParameterUnder> groupParametersUnder = new
        List<GroupParameterUnder>();

    // Get all families existing in the Document
    FilteredElementCollector collector = new FilteredElementCollector(Document);
    collector.OfClass(typeof(Family));

    // Loop through all families existing in the Document
    foreach (Family family in collector)
    {
        if (family.IsEditable)
        {
            // Get familyManager
            Document familyDocument = Document.EditFamily(family);
            FamilyManager familyManager = familyDocument.FamilyManager;

            // Loop through family's BuiltInParameterGroup values
            foreach (BuiltInParameterGroup item in
                (BuiltInParameterGroup[])Enum.GetValues(typeof(BuiltInParameterGroup)))
            {
                if (familyManager.IsUserAssignableParameterGroup(item))
                {
                    groupParametersUnder.Add(new
                        GroupParameterUnder(LabelUtils.GetLabelFor(item), item));
                }
            }
            // Note:
            // The following line of code is a quick&dirty solution, but atm i
            // didn't know how else to solve this.
            // Every following iteration will only add the same data to the list,
            // so it's "safe" to break the loop
            break;
        }
    }
    // Sort the list alphabetically
    groupParametersUnder = groupParametersUnder.OrderBy(x => x.Label).ToList();

    // Return list
    return groupParametersUnder;
}

```

Disclaimer: due to my limited understanding of many Revit and general architectural concepts, i am not convinced that this is the best solution, but it seems to work.

4.7.12 Utils

The TUM.MasterThesis.MarijaRakic.Utils namespace contains only one method that checks whether a shared parameter with a given name exists. If it does, the method returns information on its BuiltInParameterGroup

value. This value will further be used in combination with data contained in the `GroupParameterUnder` entity to show a human-readable version of the *Group parameter under* value, as shows in Figure 4.28b. The code is as follows:

Listing 4.48 Check whether a shared parameter with a given name exists

```
public static BuiltInParameterGroup? SharedParameterExists(string paramName)
{
    BindingMap bindingMap = Document.ParameterBindings;
    DefinitionBindingMapIterator iter = bindingMap.ForwardIterator();
    iter.Reset();

    while (iter.MoveNext())
    {
        Definition tempDefinition = iter.Key;

        // find the definition of which the name is the appointed one
        if (string.Compare(tempDefinition.Name, paramName) != 0)
        {
            continue;
        }

        // get the category which is bound
        ElementBinding binding = bindingMap.get_Item(tempDefinition) as ElementBinding;
        CategorySet bindCategories = binding.Categories;
        foreach (Category category in bindCategories)
        {
            if (category.Name == Document.Settings.Categories.get_Item(
                Task.ProjectParameter.SharedParameter.BuiltInCategory).Name)
            {
                return tempDefinition.ParameterGroup;
            }
        }
    }

    // return null if shared parameter doesn't exist
    return null;
}
```

4.7.13 MultiLODLib

This project consists of several internal data model classes and one public class called `Analyser` with one public method called `Run`. This class acts as a building model analyser by iterating through an array of requirements and verifying whether the building model complies with them. However, the current implementation of the application does not cover analyzing all existing building model elements, just walls. It is left for some future version of the plug-in to handle other elements.

The `Analyser` class has the following fields and properties:

- `MultiLODpath`, which contains a path to the local file containing a multi-LOD meta-model and `MultiLOD`, which contains data loaded from the previously mentioned file,
- `MappingFilePath`, which contains a path to the local file containing mapping between Revit and IFC properties, and `MappingFile`, which contains data loaded from the previously mentioned file,
- `Document`, which contains information on the currently open Revit project stored in an instance of the `Document` class, coming from the `Autodesk.Revit.DB` namespace, and
- `CurrentUser`, which contains information on the current user.

The output of the `Run` method, and therefore the `Analyser` class as well, is an `ObservableCollection` of type `Task`. Each element of this collection specifies one missing object's property or property's missing value.

This section will first describe the multi-LOD meta-model that stores requirements that the Revit model has to comply with, and then the `Run` method that analyzes the Revit model.

The multi-LOD meta-model

The multi-LOD meta-model is stored in a JSON file on the local system. This meta-model is still under development and at the moment of implementing this plug-in and writing this thesis, it has the following template:

Listing 4.49 An example of the multi-LOD meta-model

```
{
  "id":1,
  "level":"1",
  "description":"Building Development Level 1",
  "requirements":[
    {
      "id":21,
      "componentTypeId":1,
      "lodId":1,
      "lodLevel":"200",
      "componentTypeName":"Wall",
      "ifcType":"IfcWall",
      "requirements":[
        {
          "id":61,
          "name":"Pset_GenericWall",
          "componentType":1,
          "levelOfDevelopment":1,
          "geometryRepresentation":1,
          "properties":[
            {
              "id":110,
              "name":"Height",
              "isMandatory":true,
```



```

        "fuzzinessType":2,
        "fuzzinessPercentage":60,
        "isGeometric":true,
        "dataType":"Number",
    }
  ]
}

```

In order to easily load a meta-model from the file and use it in the code, several internal data model classes have been created in the `TUM.MasterThesis.MarijaRakic.MultiLODLib.Models` namespace. An *internal* class is a class that is visible only within the same assembly in which it is defined. In the case of these data models, an *internal* access modifier is appropriate, because these structures should not be accessible from outside of the `MultiLODLib` project; they are specific to this implementation of the `Analyser` class.

In the remainder of this section I shortly explain those data model classes. In addition, I focus on the properties that are used in the current implementation of the application.

A `MultiLOD` class is the top element consisting of an `id` property of type integer, a `label` and a `description` of type string, and an array of type `MultiLODRequirement`. In the current implementation of `Analyser`, only the array of `MultiLODRequirement` objects is used.

A `MultiLODRequirement` class consists of an `id` property of type integer, a `componentTypeId` of type integer and a `componentTypeName` of type string, a `lodId` of type integer and a `lodLevel` of type string, an `ifcType` of type string, and an array of type `MultiLODComponentRequirement`. In the current implementation of `Analyser`, only the array of `MultiLODComponentRequirement` objects is used. Since the current version of `Analyser` handles only missing parameters or parameter's value of walls, we do not use neither `componentTypeId`, `componentTypeName` nor `ifcType`. In the future versions, the `MappingFile` should be used to map information from the previously mentioned properties to a Revit element.

Further on, a `MultiLODComponentRequirement` class consists of an `id` property of type integer, a `name` of type string, a `componentType` of type integer, a `levelOfDevelopment` of type integer, a `geometryRepresentation` of type integer, and an array of type `MultiLODProperty`. Once again, only the array of `MultiLODProperty` objects is used at the moment.

Finally, a `MultiLODProperty` class consists of an `id` of type integer, a `name` of type string, an `isMandatory` of type boolean, a `fuzzinessType` of type integer, a `fuzzinessPercentage` of type integer, an `isGeometric` of type boolean, and a `dataType` of `ParameterType` type. `ParameterType` is an enum coming from the `Autodesk.Revit.DB` namespace and it is "An enumerated type listing all of the data type interpretation that Autodesk Revit supports", as per the official documentation.

In the current implementation of `Analyser`, only the following properties are used:

- `Name`, to store the name of the Revit parameter that we want to check whether exists and whether it has a value,
- `IsMandatory`, to specify whether the element has to contain a certain parameter, and
- `DataType`, to specify data format of the parameter with a previously given name. `DataType` exists in the same format that Revit expects it, so no additional mapping or casting of data types is needed.

Rest of the properties are yet to be utilized in some of the future versions of `Analyser`.

The Run method

The Run method works in the following way: first, it loads the multi-LOD object from a given local path and deserializes it into an instance of the MultiLOD class. For this we use Newtonsoft, a third-party library further described in Section 4.2.

Listing 4.50 Read multi-LOD meta-model from a file and deserialize it

```
// read file into a string and deserialize JSON to a type
MultiLOD multiLOD = JsonConvert.DeserializeObject<MultiLOD>(File.ReadAllText(path));
```

Next, MappingFile is initialized by reading the content of the local file that Revit uses to export a model to the IFC format. Each line of this file has the following format: *RevitCategory*, *IFCClassName* and *IFCType*, and so we map those information into the properties with the same names:

Listing 4.51 Private class MappingSchema

```
public string RevitCategory { get; }
public string IFCClassName { get; }
public string IFCType { get; }
```

After both of these properties are initialized, Run loops through all instances of MultiLODRequirement. MultiLODRequirement specifies a componentName, so we filter the currently open Document for all elements of this type. In our case, this property always has the value of *Wall*.

Listing 4.52 Get all walls from the current project

```
FilteredElementCollector walls =
    new FilteredElementCollector(Document).OfClass(typeof(Wall));
```

Next, we check whether each Revit element (in our case wall) has all parameters specified in the array of MultiLODProperty objects.

We get all parameters with a certain name of an element by invoking the following method:

Listing 4.53 Get all parameters of the wall that have a given name

```
List<Autodesk.Revit.DB.Parameter> parameters =
    wall.GetParameters(property.Name).ToList();
```

Cases that are of interest to us are if the parameter is specified to be mandatory, but:

- the list of parameters is empty, or
- a parameter does exist, but has no value.

In both cases we create a new Task. In the first case we store information on the missing shared parameter with a Name and a ParameterType:

Listing 4.54 Saving information on the SharedParameter that is yet to be created by an architect

```
SharedParameter sharedParameter = new SharedParameter()
{
    Name = property.Name,
    ParameterType = property.DataType,
    BuiltInCategory = BuiltInCategory.OST_Walls
};
ProjectParameter projectParameter = new ProjectParameter(sharedParameter);
```

In the latter, we store information on the existing Parameter that we fetched as shown in Listing 4.53.

Since the current plug-in version supports only walls, some information is hard-coded, e.g. a `BuiltInCategory` of a `SharedParameter`, as shown in Listing 4.54. In future versions, information from the multi-LOD meta-model should be combined with the mapping information from the `MappingFile` property to determine a correct `BuiltInCategory`.

One final note: since `Analyser` is a project, it could easily be replaced by a different analyser class, as long as it accepts the same input parameters and returns results in the same format.

4.7.14 MarijaRakicMasterThesis.DataTypes

This project contains data models. Classes are placed in the `TUM.MasterThesis.MarijaRakic.DataTypes.Models` namespace, whereas enumerations are placed in the `TUM.MasterThesis.MarijaRakic.DataTypes.Enums` namespace. It also contains a simple `Utilities` class that creates a random color for a Revit element.

Enumerations

Let's have a look at the enumeration types first:

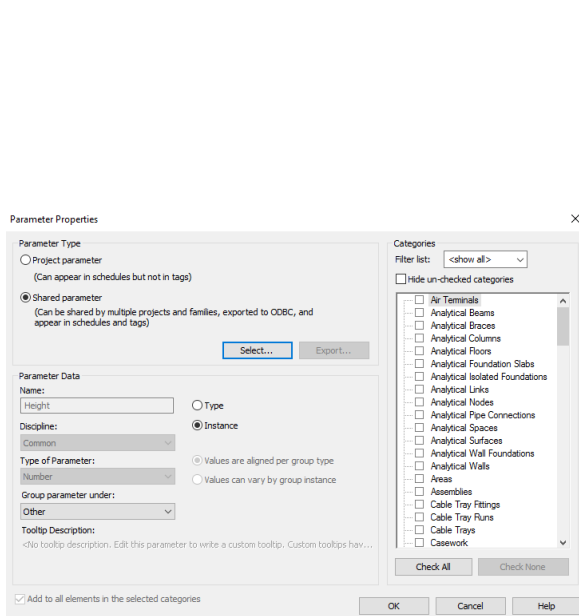
- `ActionType` specifies whether an element misses an entire parameter or only a parameter's value. It has two possible values: `MissingPropertyValue` and `MissingObjectProperty`.
- `ButtonContent` specifies the text on the button that can be seen in Figure 4.7a - *Add Parameter*, 4.7b - *Add Value*, and 4.8a - *Add Suggestion*. It has three possible values: `AddParameter`, `AddValue`, and `AddSuggestion`.
- `Chart` specifies the type of chart that is being presented to a user. It has two possible values: `PieChart` and `StackedColumns`.
- `IssuesView` specifies whether tasks will be presented to a user in a form of a list or a tree. It has two possible values: `ListView` and `TreeView`.
- `Role` specifies roles that exist in the system. At the moment of developing this plug-in, it is intended to be used either by an *Architect*, a *Life Cycle Analysts (LCA)* or a *Structural Engineer*. Defining a role as enumeration makes the system easily extensible. For example, introducing another type of an engineer to the application would require two easy steps: adding another value to the `Role` enumeration, and specifying whether a `User`, which will be explained in the upcoming section, belongs to the type of an architect or an engineer, since these two types of users have a very distinctive responsibilities. It would also be possible to retrieve this information from the server. In this case, application would be more easily extensible, since changing the existing roles would not require that a user updates the application.
- `Status` specifies the state of a `Task`. It has three possible values:
 - `Open`, meaning that an engineer has created the task, but the architect who requested the analysis has not solved it yet,
 - `Pending`, meaning that an architect has requested options from an engineer and the engineer has suggested options, but the architect is yet to accept or decline them, and
 - `Solved`, meaning that an architect has either created a solution themselves or that they have accepted the solution suggested by an engineer.

Classes

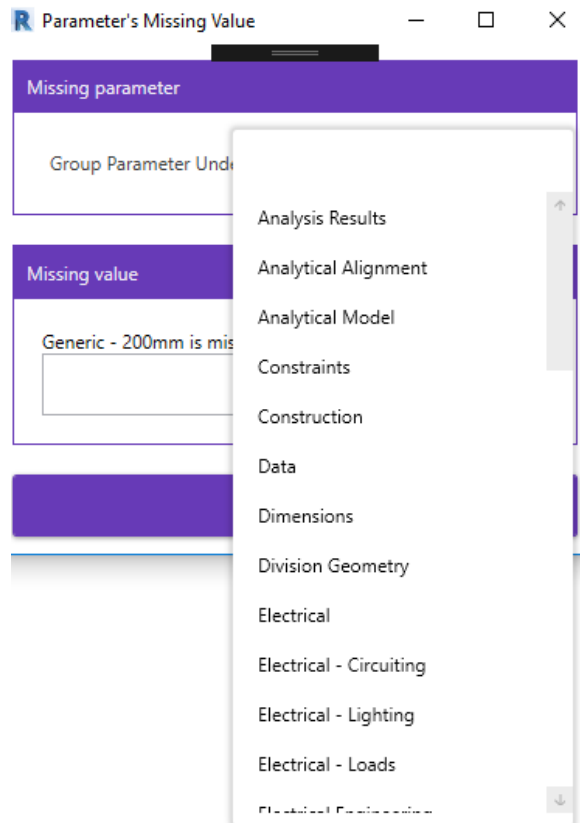
Classes that model the data used in this application are the following:

- `GroupParameterUnder` is used to display the `Group` parameter under information while an architect is creating a new parameter. It stores information in form of `BuiltInParameterGroup`, an enum defined in the `Autodesk.Revit.DB` namespace and described as "An enumerated type listing all of the built-in parameter groups supported by Autodesk Revit", as per the official documentation. This information is used by Revit when creating a new shared parameter. However, this is computer-readable data, that does not say much to the Revit user. Hence, we need to store the human-readable equivalent of this information, and for that we use a string property called `Label1`. Initializing of this field is described in more detail in Section 4.7.11.

- `MaterialWithImage` is used to group information on the `Material` class, defined in the `Autodesk.Revit.DB` namespace and that “Represents a material element within an Autodesk Revit project”, as per the official documentation; and its image that is shown to a user while creating a new parameter or a value of type `Material`, as seen in Figure 4.10b. As explained in Section 4.7.8, the Revit API does not offer a direct way of accessing a `Material`’s image property. With this in mind, this class was created to store this property once and easily access it every following time it is required.
- `MissingValuesPerElement` is used to group all `Tasks` that refer to the same Revit element. For this reason, this class stores a list of `Tasks` and an instance of the `ElementId` class, which represents “a unique identification for an element within a single project”, as per the official documentation; it is defined in the `Autodesk.Revit.DB` namespace. In addition, this class stores the Revit element’s name and color information. It is just an alias for the sake of a quick access, since the same data can be accessed through any element of the `Tasks` list as well.
- `ProjectParameter` is used to store information on the `SharedParameter`, described later in this section, and its `BuiltInParameterGroup`. It models information needed to create a new *Project parameter* out of an existing *Shared parameter*, in the same way a Revit user would do it using the native controls. Figures 4.28a and 4.28b depict creating a new project parameter using native Revit controls and our plug-in, respectively. An example of the usage of this class is shown in Section 4.7.13.



(a) A native Revit form for creating a project parameter



(b) Our plug-in’s ‘Group parameter under’ feature

Figure 4.28 Creating a new project parameter

- `RevitElement` is used to group information on the Revit element’s `ElementId` and `Name` properties. The `Name` parameter is very easy to access when we have an instance of the `Element`. However, that is not always the case, for example when an architect receives information from an engineer on the tasks that are to be completed, we only have information on the `ElementId`. Even though it is

possible to fetch a Revit Element by ElementId, we want to avoid doing this every time we want to print some message to the user, hence the need for this class.

- SharedParameter is used to store information on a shared parameter's Name, a ParameterType, defined in the Autodesk.Revit.DB namespace and described as "An enumerated type listing all of the data type interpretation that Autodesk Revit supports", as per the official documentation, and a BuiltInCategory, defined in the Autodesk.Revit.DB namespace and described as "A list of all the built in categories within Revit", as per the official documentation. It models information needed to create a new *Shared parameter*, in the same way a Revit user would do it using the native controls. A Revit user can relate all these fields to the form for creating a shared parameter shown in Figure 4.29:

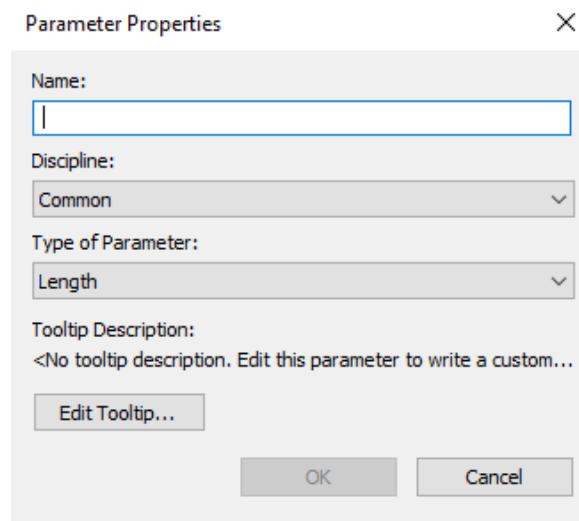


Figure 4.29 A native Revit form for creating a shared parameter

A note: a discipline will be inherited from ParameterType.

- Task is used to store information on the RevitElement that the task refers to, an instance of the Parameter class in case the parameter exists, but misses a value; Parameter is defined in the Autodesk.Revit.DB namespace and described as "The parameter object contains the value data assigned to that parameter". In case of the missing parameter, Task stores information on the ProjectParameter that is yet to be created. Next, Task store a Status of the parameter, an ActionType specifying whether the parameter or parameter's value is missing, a missing Value once it is created, and a Color assigned to the Revit element. Task also stores information on the time when the Task is solved. The current database schema does not support permanently persisting this value, but it is available for some future implementation of the system when, thanks to this information, it would be possible to show a lot of different charts, e.g. a burn down chart and many others that would model historical data.
- User is used to store information on the user's Name of type string and Role of type Role, as defined in the previous section. It also implements two methods that specify the type of a user, since information on the sub-type of an engineer is irrelevant to the current implementation of the system:

Listing 4.55 Methods to check whether the current user is an architect or an engineer

```
public bool IsEngineer()
{
    return Role == Role.LCA || Role == Role.StructuralEngineer;
}
```

```
public bool IsArchitect()
{
    return Role == Role.Architect;
}
```

4.7.15 MarijaRakicMasterThesis.Converters

This is a very simple project that consists of only one class `MultiValueConverter`. This class allows more than one boolean value to define, and consequently trigger the change of the display state of a XAML element. It does so by translating a conjunction of two boolean values to a value from the `Visibility` enum, whose possible states are `Visible`, `Collapsed` and `Hidden`.

`MultiValueConverter` class implements interface `IMultiValueConverter` coming from `System.Windows.Data` namespace. As per the official documentation, this interface “Provides a way to apply custom logic in a `System.Windows.Data.MultiBinding`.” This interface defines two methods to be implemented:

- `Convert`, which is described as “Converts source values to a value for the binding target. The data binding engine calls this method when it propagates the values from source bindings to the binding target” in the official method documentation, and
- `ConvertBack`, which “Converts a binding target value to the source binding values”, as per the official documentation. For the purposes of this plug-in the `ConvertBack` is not needed, hence it is not implemented.

The implementation of the `Convert` method is the following:

Listing 4.56 Method that converts a conjunction of two boolean values into `Visibility`

```
public object Convert(object[] values, Type targetType, object parameter,
    System.Globalization.CultureInfo culture)
{
    bool a = (bool)values[0];
    bool b = (bool)values[1];

    return a && b ? Visibility.Visible : Visibility.Collapsed;
}
```

This converter is imported to a XAML file as a resource in the following way:

Listing 4.57 Importing of the `MultiValueConverter`

```
<converters:MultiValueConverter x:Key="MultiValueConverter" />
```

and used to set the visibility of a XAML element, for example `StackPanel`, as following:

Listing 4.58 Usage of the `MultiValueConverter`

```
<StackPanel.Visibility>
    <MultiBinding Converter="{StaticResource MultiValueConverter}">
        <Binding ElementName="myElement"
            Path="IsCheckedPendingTasks" />
        <Binding ElementName="myElement"
            Path="IsArchitect" />
    </MultiBinding>
</StackPanel.Visibility>
```

Even though this is a fairly small class with only one class and two methods, it is still an independent project because of the reusability reasons: it makes integration into other solutions easy.

4.7.16 com.server.api

Disclaimer: this project was not implemented by the author of this thesis.

This project consists of data models that mimic the server database tables and a singleton class called `Communicator`. This class incorporates all methods for connecting to the existing API points. For this to work, `Communicator` utilizes third-party libraries `RestSharp` (described in Section 4.2) to make HTTP requests to the server and `Newtonsoft` (described in Section 4.2) to deserialize data received from the server.

By default, `Communicator` connects to the localhost, which is useful for testing purposes while the system is under development. Once the system is used in production, the default settings are no longer enough. Hence, when using `Communicator` to retrieve information from the production server, the first method that needs to be invoked is `UpdateServerName()`. As an input parameter, it accepts the server URL in string format. An example would be `UpdateServerName("http://10.195.1.44:3005")`, where "http://" specifies the communication protocol, 10.195.1.44 is an address of the server and 3005 is a port on which the application is available.

When it comes to data types, a server database consists of thirteen tables. Each one of them is mapped to a C# structure that is part of the `VeavCommunicationLib.Responses` namespace. The following structures exist:

- `ActionType`,
- `aLODX`,
- `AnalysisScope`,
- `ChangeSuggestion`,
- `Feedback`,
- `NewAnalysis`,
- `People`,
- `ProductCatalog`,
- `Project`,
- `ProjectAssignment`,
- `PropertyCatalog`,
- `Rule_aLOD`, and
- `RulesProductProperty`.

Previously defined terms will be further explained in the example of a database table and a C# structure `People` in the rest of this section. HTTP methods, as explained in Section 4.1, are used to send and receive data from the server:

- Code snippet 4.59 shows the `AddPeople` method that uses an HTTP POST request to create a user with a given name and role.
- Code snippet 4.60 shows three HTTP GET methods: `GetAllPeople`, `GetPeopleById` and `GetPeopleByName` to retrieve information on all users, a user with a given ID, and all users with a given name, respectively. It also shows deserialization of the HTTP response to a single value of type `Person` or to an array of values of type `Person`.
- Code snippet 4.61 shows the `UpdatePeople` method that uses an HTTP PUT method to update the user with a given ID.

- Code snippet 4.62 shows the RemovePeopleById method that uses an HTTP DELETE method to delete a user with a given ID.

Listing 4.59 AddPeople method as an example of the HTTP POST method

```
public bool AddPeople(string personName, string personRole)
{
    RestRequest request = new RestRequest("personsapi/");
    request.AddParameter("id", 0);
    request.AddParameter("PersonName", personName);
    request.AddParameter("PersonRole", personRole);

    var response = _client.Post(request);
    JObject parsedResponse = JsonConvert.DeserializeObject(response.Content) as JObject;

    if (parsedResponse == null) return false;
    if (!parsedResponse.ContainsKey("status")) return false;
    return parsedResponse["status"].ToString() == "Person Saved";
}
```

Listing 4.60 An examples of the HTTP GET methods and deserialization

```
public List<People> GetAllPeople()
{
    RestRequest request = new RestRequest("personsapi/");
    var response = _client.Get(request);
    return JsonConvert.DeserializeObject<List<People>>(response.Content);
}

public People GetPeopleById(int id)
{
    RestRequest request = new RestRequest("personsapi/{id}");
    request.AddUrlSegment("id", id);

    var response = _client.Get(request);
    return JsonConvert.DeserializeObject<People>(response.Content);
}

public List<People> GetPeopleByName(string name)
{
    RestRequest request = new RestRequest("personsapi/search/{keyword}");
    request.AddUrlSegment("keyword", name);
    var response = _client.Get(request);
    JObject parsedResponse = JsonConvert.DeserializeObject(response.Content) as JObject;

    if (parsedResponse == null) return null;
    if (!parsedResponse.ContainsKey("data")) return null;

    return parsedResponse["data"].Value<JArray>().ToObject<List<People>>();
}
```

Listing 4.61 UpdatePeople method as an example of the HTTP GET method

```
public bool UpdatePeople(People updatedPeople)
{
    RestRequest request = new RestRequest("personsapi/");
    request.AddParameter("id", 0);
    request.AddParameter("PersonName", updatedPeople.PersonName);
}
```

```

request.AddParameter("PersonRole", updatedPeople.PersonRole);

var response = _client.Put(request);
JsonObject parsedResponse = JsonConvert.DeserializeObject(response.Content) as JObject;

if (parsedResponse == null) return false;
if (!parsedResponse.ContainsKey("status")) return false;
return parsedResponse["status"].ToString() == "Person Updated";
}

```

Listing 4.62 RemovePeopleById method as an example of the HTTP DELETE method

```

public bool RemovePeopleById(int id)
{
    RestRequest request = new RestRequest("personsapi/{id}");
    request.AddUrlSegment("id", id);

    var response = _client.Delete(request);
    JsonObject parsedResponse = JsonConvert.DeserializeObject(response.Content) as JObject;

    if (parsedResponse == null) return false;
    if (!parsedResponse.ContainsKey("status")) return false;
    return parsedResponse["status"].ToString() == "person Deleted";
}

```

All the other methods from the `Communicator` class are implemented in a similar way, just invoking a different API point and deserializing response into a different data type.

The code of `Communicator` class is organized into regions to logically group multiple methods related to one database table, and therefore local C# structure as well. Consequently, there are *Action_Type Functions*, *aLOD Functions*, *Analysis_Scope Functions*, to name a few.

5 Summary

In the Architecture, Engineering and Construction (AEC) industry, different stakeholders need to communicate often. This implies frequent data exchange; at the same time it is crucial that no user has stale data. Another problem is that each party in this communication holds information in the format native to a software tool used in their field of expertise. Yet, to successfully exchange data, the person receiving it should be able to import it into their (different) set of tools. Building Information Modeling (BIM) works toward solving these problems. A desired outcome of a BIM process is an improved collaboration in this fragmented industry.

A Level of Development (LOD) defines how precise the entire building model is during each stage of the project duration. There are research attempts to define a more granular approach. The multi-LOD meta-model, for example, assumes that different elements can have different levels of detail, as per the EarlyBIM research project [Abualdenien and Borrmann, 2019]. This meta-model holds information on the model stored in an IFC format, an open source data model for achieving interoperability in the AEC industry; the meta-model itself is in a JSON format.

BIM Collaboration Format is an open file format for exchanging comments and reporting potential flaws of a building model. A TUM research group proposed a new way of communication, one which acts as an adaptive minimized communication protocol and a ticketing system. This is materialized in a form of an HTTP communication server and an existing relational database used to store relevant aspects of this communication.

I implemented a plug-in as a show case of the aforementioned communication protocol used directly within Revit, so that there is no need for a separate communication application that users would have to switch to. Since the admin interface is nonexistent, I acted as an administrator of the system and created data needed to run the system successfully, such as users, projects, etc directly in the database. Using C# and WPF, I created an application that allows an architect to request a new analysis from an engineer. After attempting to run this analysis, if it turns out that the engineer cannot evaluate the model because it is incomplete, this information is communicated back to the architect who requested the analysis with a list of tasks to be completed before the analysis can be run successfully.

To check whether the model is complete and ready for an analysis, the multi-LOD meta-model is used.

The architect can ask for value suggestions from an engineer, and the engineer can suggest one or multiple options. Only an architect can make changes to the building model, therefore they are responsible for accepting or declining a value(s) suggested by the engineer.

5.1 Future work

There are some features that were out of scope of my thesis and are, therefore, left as future work.

- An existing Revit model could be compared against more multi-LOD meta-model requirements, not just the one that specifies whether a parameter with a given name is mandatory.
- A communication protocol anticipates more possible actions, not just a missing parameter and a parameter's missing value. The entire element could be created, updated or marked for deletion through the use of the plug-in.
- At the moment, the plug-in checks whether walls comply with the multi-LOD meta-model requirements. Not all the Revit elements share the same characteristics; different Revit elements have different parameters. Therefore, the code is very "wall-specific" in some places. In the future, the

application could cover all the Revit elements. To achieve this, certain classes would have to implement parts of the checking process in a manner specific to the element type. Some of this code is already part of the code base, but is not invoked. For example, when retrieving Material information of an element, there are three different use cases:

- Elements of type *Wall*, *Floor*, *RoofBase* and *FootPrintRoof* have compound structures consisting of one or more layers. We fetch those structural materials by a layer's materialId property.
 - Elements of type *Beam*, *Column* and *Foundation* have another way to get their material: using their StructuralMaterialId property. This property returns an ElementId which identifies the material that defines the instance's structural analysis properties.
 - All other Revit elements, such as *Doors*, *Windows*, etc, have unique parameters that can, nonetheless, be accessed uniformly.
- The current implementation of the plug-in expects a Wall's material information only in a list of elements' parameters. However, a material could be part of the wall's layers as well. This code also exists, but is not used.
 - Improving graphical aspects of the application is a never-ending job. The application UI could use guidelines of an experienced UX expert. Regardless of this, one concrete feature could be added to the application. At the moment, a tree view is just a showcase that the list of tasks can be shown in many different ways, not just like as a simple table-like structure. It would be nice to provide the same options to the user that they have while working with tasks from a list view - a button to add a value, parameter or suggestion, an accept or decline button, information on the newly created value, and selection and deselection of a Revit element. On the level of the entire tree view, it could be possible to filter the tree view for open, pending and solved tasks.
 - If the database would be extended to store a date when a Task is created and a date when the Task is solved, the application could implement more charts. For example, a burn down chart could show the number of open tasks at the beginning of the project and the number of solved tasks per each day until all the tasks have been solved. It could also combine information on the open, pending and solved tasks in one interactive chart where the user could choose whether to show all these tasks at the same time or hide one or many of them.

List of Figures

2.1	System architecture	8
3.1	A database schema, by Christopher Onuoha, an part of an IDP project, TUM, December 2018	9
4.1	The same plugin page, with and without Material Design	14
4.2	Add-ins tab with many panels, including the one called TUM	15
4.3	Choose a user	18
4.4	Home page of the plug-in, as seen by an architect	18
4.5	Form for requesting a new analysis, as seen by an architect	19
4.6	Home page of the plug-in, as seen by an engineer	20
4.7	Analysis results, as seen by the architect	21
4.8	Analysis results, as seen by the engineer	22
4.9	Creating a missing parameter, as seen by an architect	23
4.10	Creating a missing parameter of different data types, as seen by an architect	24
4.11	A native Revit list of materials, with a search by name and filter by class features	25
4.12	Creating a missing value	26
4.13	Filtering the list of tasks by type	27
4.14	An engineer can suggest one or multiple values	28
4.15	A list of pending tasks	29
4.16	A value suggested by the engineer is automatically input in the value field	29
4.17	Different combinations of existence of shared parameter and suggested value	30
4.18	List of solved tasks	31
4.19	Newly created wall's parameter in the Properties view	32
4.20	A tree view representation of tasks, grouped by Revit element	33
4.21	Wall is colored when selected	34
4.22	Available charts	35
4.23	Changing the current user	36
4.24	Requesting and running a new analysis	37
4.25	An architect solving tasks and accepting or declining suggested values	38
4.26	Solution dependencies	40
4.27	Grouping two CompareToMultiLODViewModel partial classes	53
4.28	Creating a new project parameter	67
4.29	A native Revit form for creating a shared parameter	68

Listings

4.1	Deserializing data received from a server into a People object	12
4.2	Creating a resource dictionary by merging XAML files	13
4.3	Specifying a single Window's resource	13
4.4	Specifying multiple Window's resources	13
4.5	The hack to make the application work	14
4.6	A hack to make the LiveCharts library work, code-behind	14
4.7	A hack to make the LiveCharts library work, XAML click event	15
4.8	An example of the .addin file	16
4.9	OnStartup method that creates a ribbon panel and a push button	41
4.10	Creating of a push button on a given ribbon panel	42
4.11	Execute method that runs when a user starts our plug-in	42
4.12	A class that sets an instance of the CompareToMultiLOD class to be a provider of a dockable pane	42
4.13	Execute method that runs when a user starts our plug-in	43
4.14	Creating and showing a dockable pane	43
4.15	Usage of the custom user controls	45
4.16	Usage of the DecimalUpDown control	45
4.17	Microsoft Visual Studio automatically includes some namespaces when creating a new Window	45
4.18	Usage of the DatePicker	46
4.19	Interface ICloseable definition	46
4.20	Passing Window instance as a command parameter	47
4.21	Casting method's argument to ICloseable	47
4.22	Invoking the Close method on an instance of ICloseable	47
4.23	Binding a value to the user control's MissingValues property	48
4.24	Implementation of the binding parameter to the custom user control	48
4.25	Including BooleanToVisibilityConverter as a user control resource	48
4.26	Using BooleanToVisibilityConverter to set the visibility of a ListView	49
4.27	Usage of MultiDataTrigger	49
4.28	Usage of the MultiValueConverter class	49
4.29	Implementation of the NotEmptyValidationRule class	50
4.30	Implementation of the FutureDateValidationRule class	50
4.31	Implementation of the INotifyPropertyChanged interface	51
4.32	Storing the Name variable in Settings	51
4.33	Retrieving and casting the Name variable	52
4.34	Filtering an array by name	52
4.35	Retrieving of the material's image	52
4.36	Grouping two CompareToMultiLODViewModel partial classes	53
4.37	example is here	54
4.38	Assign a different subset of tasks to be presented to a user	54
4.39	Showing a new window	54
4.40	Opening of the file from the local system	55
4.41	AddValueCommand as an example of the class that implements the ICommand interface	55
4.42	Set a color of the OverrideGraphicSettings	57

4.43 Clear a color of the OverrideGraphicSettings	57
4.44 Execute a transaction	57
4.45 A transaction to set the parameter's value	57
4.46 A transaction to create a new shared parameter	58
4.47 Fetch a human-readable equivalent of all the BuiltInParameterGroup existing in the project	60
4.48 Check whether a shared parameter with a given name exists	61
4.49 An example of the multi-LOD meta-model	62
4.50 Read multi-LOD meta-model from a file and deserialize it	64
4.51 Private class MappingSchema	64
4.52 Get all walls from the current project	64
4.53 Get all parameters of the wall that have a given name	64
4.54 Saving information on the SharedParameter that is yet to be created by an architect	64
4.55 Methods to check whether the current user is an architect or an engineer	68
4.56 Method that converts a conjunction of two boolean values into Visibility	70
4.57 Importing of the MultiValueConverter	70
4.58 Usage of the MultiValueConverter	70
4.59 AddPeople method as an example of the HTTP POST method	72
4.60 An examples of the HTTP GET methods and deserialization	72
4.61 UpdatePeople method as an example of the HTTP GET method	72
4.62 RemovePeopleById method as an example of the HTTP DELETE method	73

Bibliography

- [Abualdenien and Borrmann, 2019] Abualdenien, J. and Borrmann, A. (2019). A meta-model approach for formal specification and consistent management of multi-lod building models. *Advanced Engineering Informatics*, 40:135–153.
- [BIMconnect, 2017] BIMconnect (2017). An analogy between ifc and pdf formats.
- [Borrmann et al., 2015] Borrmann, A., König, M., Koch, C., and Beetz, J. (2015). *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. Springer-Verlag.
- [Engelbart, 2001] Engelbart, D. C. (2001). Augmenting human intellect: a conceptual framework (1962). *PACKER, Randall and JORDAN, Ken. Multimedia. From Wagner to Virtual Reality*. New York: WW Norton & Company, pages 64–90.
- [GOV.UK, 2011] GOV.UK (2011). Policy paper: Government construction strategy.
- [Kolltveit and Grønhaug, 2004] Kolltveit, B. J. and Grønhaug, K. (2004). The importance of the early phase: the case of construction and building projects. *International Journal of Project Management*, 22(7):545–551.
- [Royal Institute of British Architects, 2018] Royal Institute of British Architects, RIBA, E. L. L. (2018). The national bim report 2018.
- [Walasek and Barszcz, 2017] Walasek, D. and Barszcz, A. (2017). Analysis of the adoption rate of building information modeling [bim] and its return on investment [roi]. *Procedia Engineering*, 172:1227 – 1234. Modern Building Materials, Structures and Techniques.
- [Zahedi and Petzold, 2018] Zahedi, A. and Petzold, F. (2018). Seamless integration of simulation and analysis in early design phases. In *Proceedings of the Sixth International Symposium on Life-Cycle Civil Engineering (IALCCE 2018)*.