UPPSALA
UNIVERSITET

UPTEC STS22 004
Examensarbete 30 hp
Februari 2022

Designing an application to improve cloning of BIM drawings - a feasibility study

Alexander Fors

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Designing an application to improve cloning of BIM drawings - a feasibility study

Alexander Fors

Construction drawings have gotten a lot more sophisticated through the years. Most companies today use Building Information Modeling (BIM) which is the process of creating a digital representation of physical places. These models save time and reduce the risk of errors during the construction process, but are time-consuming to create. Companies are therefore interested in improving this process using software solutions. Tekla Structures is a BIM-software that enables the creation of 3D-models. These models can in turn be used to generate a 2D-drawing using a built-in cloning algorithm. This method saves a considerable amount of time when creating construction drawings, but the algorithm does not perform well when used for more complex tasks.

This thesis is written in collaboration with the architecture and engineering consultancy Sweco. The study aims to analyse the feasibility of solutions for improving or replacing the Tekla Structures cloning algorithm. An elementary application is developed to answer the research question. During development methods such as architectural design and prototyping were employed as construction tools and used to identify constraints. In the later stages of development Human-computer interaction (HCI) guidelines are followed to create more user-friendly software. Tekla Open API has been used as an interface to interact with Tekla Structures, providing useful methods and classes. This study emphasises the complexity of creating these kinds of applications. While it is true that similar tools can and should be developed in the future, it is suggested that the focus should be on creating custom solutions using the built-in Tekla cloning algorithm.

Handledare: Filip Kristiansson
Ämnesgranskare: Amin Allalou
Examinator: Elísabet Andrésdóttir
ISSN: 1650-8319, UPTEC STS22 004

Sammanfattning

Konstruktionsritningar existerar idag i digital form och har blivit allt mer sofistikerade. Byggnadsinformationsmodellering (BIM) är en modelleringsprocess där verkliga objekt avbildas med hjälp av olika 3D-modeller. Det är vanligt att använda BIM vid större byggprojekt används BIM för att reducera risker och spara tid. Mjukvaran Tekla Structures används för att skapa dessa modeller. Användare kan med hjälp av en 3D-modell i Tekla skapa 2D-ritningar i ett PDF-format som är användbart för bland annat distribution av byggnadsritningar till underleverantörer.

Tekla Structures har en inbyggd funktion för att klona ritningar, där en färdig ritning används som mall för en ny. Denna metod att kopiera delar av en tidigare skapad ritning sparar mycket tid och resurser för företag som är specialister på att skapa BIM-ritningar. Desto mer den klonade ritningen skiljer sig åt från den ritning användaren vill skapa, desto mer fel uppstår vid användandet av Teklas kloningsfunktion. Sweco är ett konsultföretag inom arkitektur och teknik och vill veta om processen att klona en ritning kan förbättras.

Målet med denna uppsats är att utforska olika mjukvarulösningar, som alternativ- eller komplement till Teklas kloningsfunktion. En applikation har sedan utvecklats för att visa att en av dessa lösningar att genomförbar. Tre prototyper har skapats under utvecklingen av den färdiga applikationen för att utforska olika lösningar, men även för att identifiera systemets uppbyggnad och begränsningar. Ett enkelt användargränssnitt har skapats i enlighet med riktlinjer för god användarvänlighet. Ett centralt verktyg i utvecklingen av applikationen är Tekla Open API. De klasser och metoder som är tillgängliga gör det möjligt för utvecklare att skriva program som kommunicerar direkt med Tekla Structures.

I slutsatsen av studien betonas komplexiteten i att skapa denna typ av applikationer. Rekommendationen är att i framtiden inte utveckla alternativ till Teklas kloningsfunktion, utan skapa programvara som komplimenterar denna funktion. Utvecklare bör inte försöka skapa en universallösning, utan fokusera på ett specifikt problem och för en viss typ av 3D-ritningar.

Contents

Introduction

Programmable automation is commonly used worldwide but may be underused in the construction industry. Construction projects share similar processes and a lot of manual work is constituted of repetitive tasks. Today most large construction projects utilize some kind of building information modeling software to create comprehensive 3D-models of the building, structure or infrastructure. This model is used as a drawing for the building, including every bit of information to complete the building project.

> Building Information Modeling (BIM) is a digital representation of physical and functional characteristics of a facility. A BIM is a shared knowledge resource for information about a facility forming a reliable basis for decisions during its life-cycle; defined as existing from earliest conception to demolition.

In contrast to a regular 3D-model the BIM-model contains information referring to the attributes of every part in the model. Therefore a model can be broken down into smaller parts, comfortably modified and extracted.

> All construction software models and drawings representing a building are not BIM, for example, models that contain only visual 3D data but no object attributes, or those that allow changes to dimensions in one view but do not automatically reflect those changes in other views. These examples miss the above-mentioned data for supporting the construction, fabrication, and procurement. (Trimble 2021)

BIM can save time in later stages of the project, however, creating these models is time-consuming. Most construction projects employ several subcontractors responsible for example manufacturing a specific part. Therefore it is unwarranted sharing the complete 3D-model with every contractor. The desired approach is to create a 2D-drawing from the complete 3D-model and emphasize the information relevant to the subcontractor. As a result,t companies investigate ways to automate the process of creating mentioned models.

1.1 Tekla Structures

Tekla Structures is used to create 3D-models of a structure and its components. Structural drafters and engineers are using Tekla Structures to model residential buildings, factories, bridges and skyscrapers (Eastman et al. 2008). The application employs object-based parametric modeling. An assembly typically contains several objects e.g. a railing contains beams, bolts and a plate used as a base for the railing. Properties like the shape of an object can be defined and controlled according to a hierarchy of parameters at assembly, sub-assembly or individual object level. In parametric design when creating e.g. a door the object has to be an instance of a class of doors, constraining the parameters of the generated object. Classes are also used to define parameters of relations of objects e.g. attached to, parallel to and distance from. Consequently, the parameters of an instance of a class can vary depending on its parameters and contextual relations to other objects. (Eastman et al. 2008)

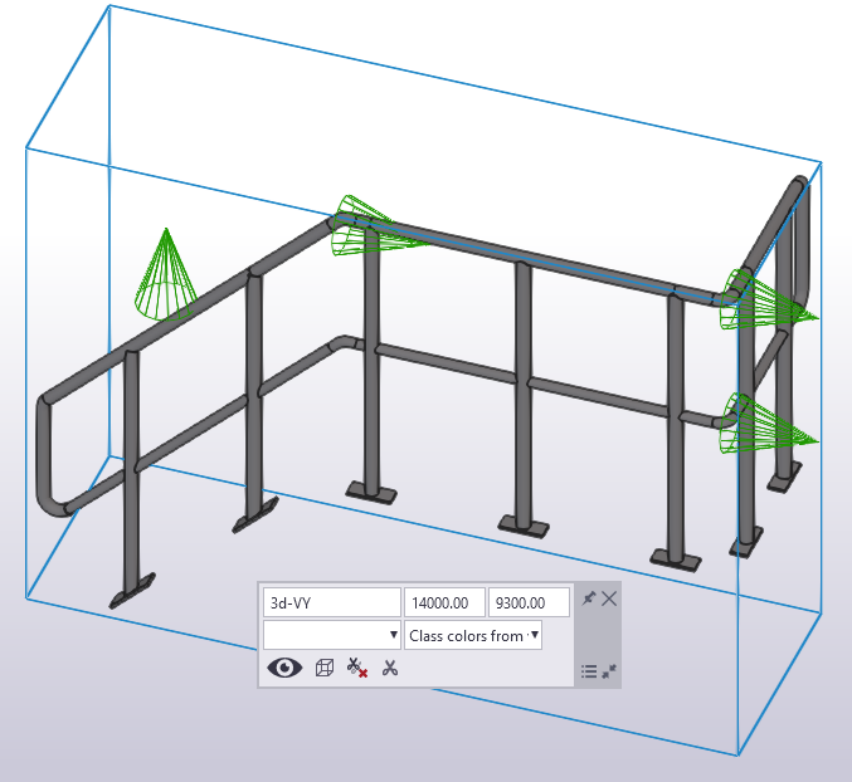

Figure 1.1: 3D-model of a railing in Tekla Structures

1.1.1 Tekla coordinate system

Tekla uses global and local coordinate systems. The global system has an X, Y, and Z-axis and has its origin in 0 for every axis. The local coordinate system is often called the work plane and most operations in Tekla Structures are dependent on this system. Users can change the work plane to make it easier to perform operations in the 3D-model. This system is defined by the XY-plane and the Z-axis is determined using the right-hand rule. In the figure below, the global coordinate system is represented by the green box and the work plane is represented by the red arrows. (Coordinates 2021)

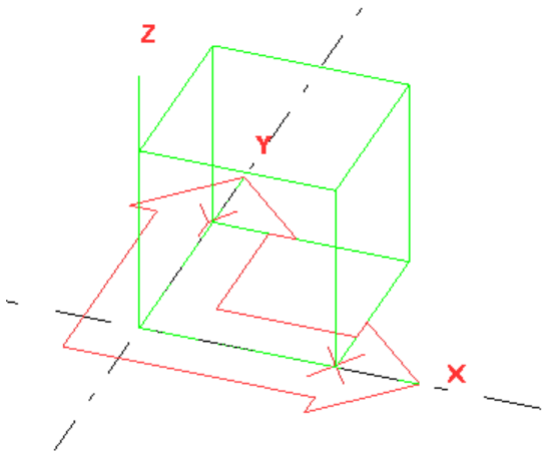

Figure 1.2: Global and local coordinate systems in Tekla Structures

1.1.2 Tekla Structures Drawings

Tekla Structures can generate a 2D-drawing from the selected assembly. This can be especially useful when working with subcontractors on a project. A structural drafter or engineer can generate a drawing from the comprehensive 3D-model excluding any unnecessary information and highlighting essential details, lowering the risk of errors and saving time. A Tekla drawing can be modified after it is generated and is structured according to object-based parametric modeling principles. The drawing is then converted to a PDF file and shared with appropriate stakeholders.

1.2 Understanding the scenario

Tekla has a cloning function that enables users to generate a drawing to a corresponding assembly using another drawing as a reference. When Tekla generates a drawing an estimation is made of how many and what type of views should be created for the selected assembly. A view is the selected part displayed from different angles. When the process is not successful the user then has to manually delete and create new Views. For more complex assemblies the Tekla drawing generation algorithm is insufficient in selecting the correct Parts for the DetailView. This is the most common source of errors for Views and a laborious process for the structural drafters and engineers (Jonsson 2021). The dimensions are often drawn in an undesirable way, reaching far beyond the sheet and not capturing the right measurements as seen in Figure 1.3.

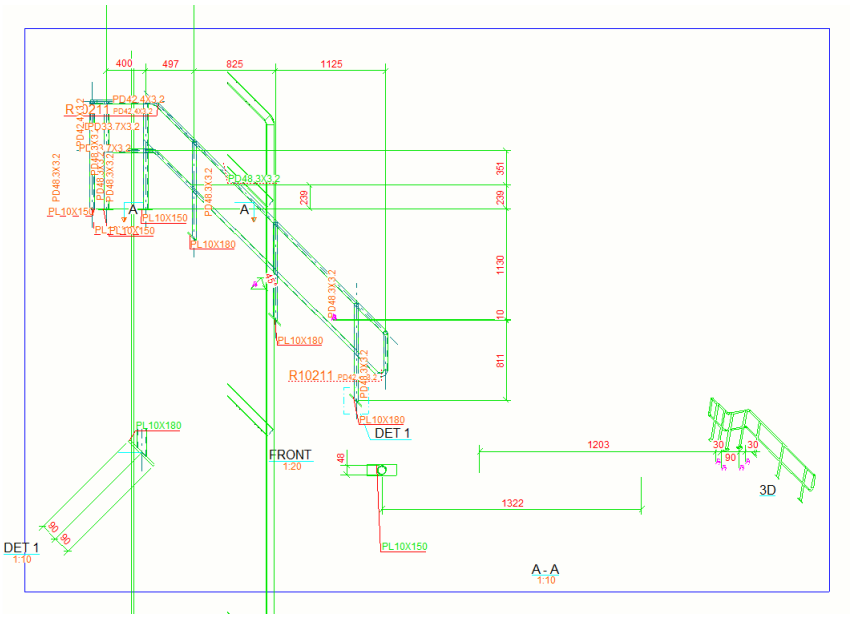

Figure 1.3: Failed cloning of a Tekla drawing

The more similar two assemblies are when cloning the better the result. The most desirable solution is for this process to just work at a click of a button, but we understand it is not that simple (Jonsson 2021). The solution is maybe more of an exploratory problem than a process of getting a solution as close to the desired outcome as possible. Are there any possible solutions to the problem and if so, which is closest to the desired outcome? Figure 1.4 shows a correct drawing for comparison. Every number is readable and displays a meaningful value.

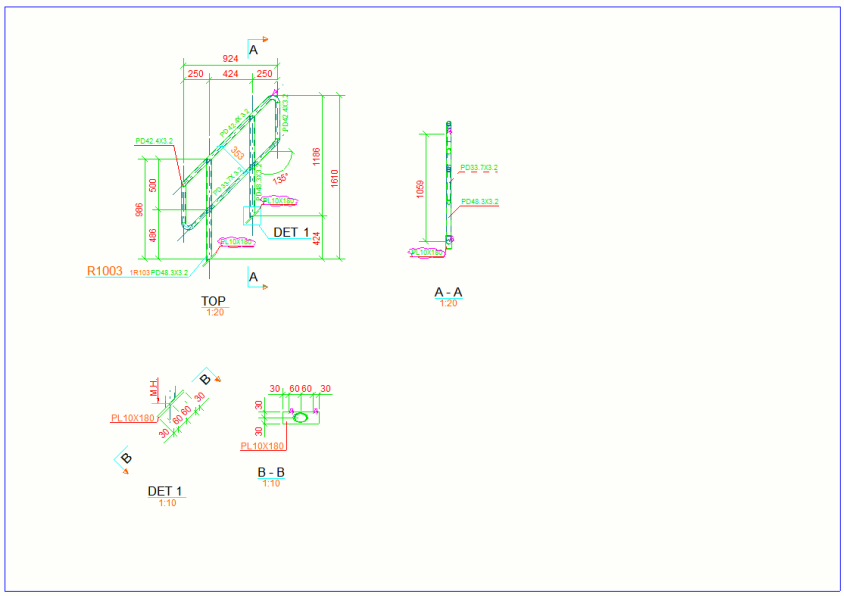

Figure 1.4: Example of a correct Tekla Structures drawing

1.3 Purpose

The generation of 2D-drawings in Tekla Structures works well for simple assemblies with few parts, but when generating drawings from more complex assemblies the result is according to the users far from desirable. Manually editing bad drawings takes a lot of time and it is a repetitive task for employees at Sweco (Kristiansson 2021). Mimicking this process seems plausible using some kind of application or script. A new system has to interact with Tekla Structures in some way since the generated drawings are of the type Tekla drawings. Since the application Tekla Structures has to be used, the architecture of the application also has to be understood to develop a solution to the problem. One goal of the study is to explore the feasibility of different solutions to the problem. Understanding the system is a central part of this study and requirement engineering helps in this regard by guiding development by clarifying the development process.

1. Explore the feasibility of different solutions. Make a qualified decision based on a theoretical framework. What is this the best solution and why?

2. Design an application, considering the requirements of the project.

1.4 Limitations

It is not expected for this thesis to present a complete solution to the problem. This is uncharted territory, the goal is to explore an idea using an application as proof for the feasibility of the solution. Less focus is dedicated to the later stages of the development process. Validation and evolution do have lower priority due to time constraints, but also because of how the goals of the study are formulated. Exploring the feasibility of different solutions and identifying requirements is done early in the development process. Some testing was still performed on the application but was restricted to railings. The application is therefore not applicable with other types of assemblies.

Theory

This thesis aims to develop an application. To do this effectively and satisfactorily several theories concerning software development has to be employed. These techniques are presented in this chapter. Theories directly concerned with the software process is used as a guide for how this thesis is presented and organized. Other theories like proof of concept prototyping are used more directly as a tool to develop the application.

2.1 The software process

Roger S. Pressman describes the importance of a balanced focus on the software process: "If the process is weak, the end product will undoubtedly suffer, but an obsessive over-reliance on the process is also dangerous". (Pressman 2001) Four activities are frequently used in the development process. Business software is often developed by modifying existing systems or by configuring and integrating custom software. A software product can also be created more or less independent of other systems utilizing a common programming language like Java or C#. Depending on what kind of system is being developed these can vary depending on the specific requirements of the system. The activities are as follows:

1. Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.
2. Software development, where the software is designed and programmed.
3. Software validation, where the software is checked to ensure that it is what the customer requires
4. Software evolution, where the software is modified to reflect changing customer and market requirements.

2.2 Incremental development

The incremental development model utilizes several versions of the software to have a more flexible development process. With every new version, the specifications of the project are updated and the validation activities are interleaved with the development of the software as can be seen in Figure 2.1. Every new increment of the system incorporates functionality specified by the customer. The most urgent or important functions are usually prioritized and focused on in earlier versions of the project. The customer can evaluate the system at an early stage of the project. The system can then be changed or new functionalities can be specified for the next increment. Changing customer requirements is therefore easier to manage with the incremental development model. The customer can be more flexible and change functionality requirements as the development process progresses. Customers can get value from the product before all the functionality has been developed. (Sommerville 2011)

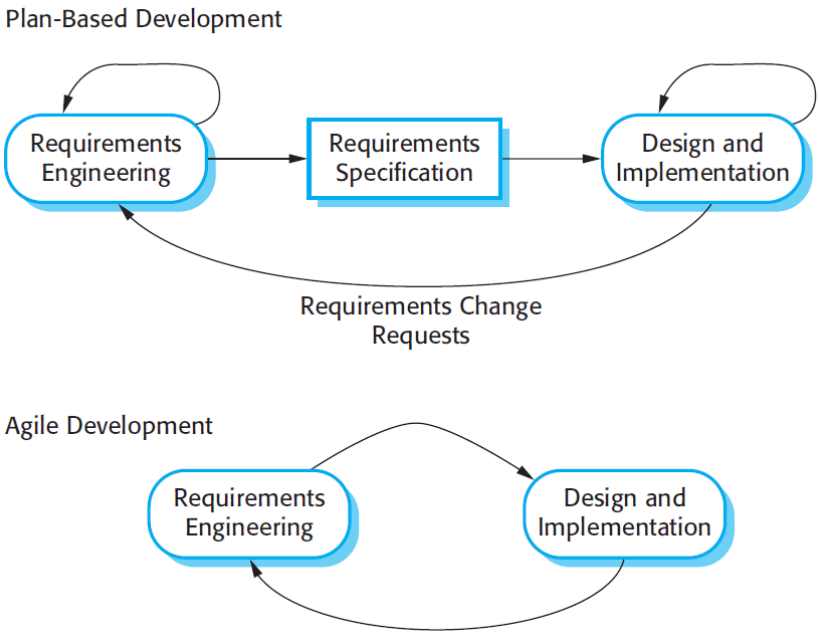

Figure 2.1: Plan-based and agile development (Sommerville 2011))

One of the problems with the incremental development process is the lack of viability, it is hard for product owners to measure progress and heavily rely on deliverables. The structure of the system usually degrades with every increment resulting in a more difficult and costly development process in the later stages of the project. The downsides of agile development are more acute for more complex and large systems with different teams working on separate parts of the system. (Sommerville 2011)

2.3 Requirements engineering

Requirements can be descriptions of how a system should behave as well as constraints of the development process. Requirements are guidelines for the development choices influencing the final product (Sommerville 2011). It is often unnecessary or impossible to fully specify all requirements before starting product development. It is common to take an agile approach refining and redefining requirements alongside the product development process. Learning just enough about the requirements of the system before starting the next iteration based on the updated requirement specification. (Wiegers and Beatty 2013)

2.3.1 Requirements elicitation

In requirement,s engineering elicitation has a central role. Elicitation is the process of collecting, discovering, extracting and defining requirements. Many types of information can be gathered employing different elicitation techniques (Sommerville 2011). Activities in the requirement process are not simply carried out in a strict sequence. Requirements can be defined after the elicitation phase which is especially true for agile projects. Establishing the system requirements can be done by observing the current system, but requires a good understanding of that system. Sharing knowledge with stakeholders and users of the system is often a good approach. Different models and prototypes of the system can be used to understand a real-world system. (Sommerville 2011)

Document analysis can be a good starting point when eliciting potential software requirements. The developer analyses all available documentation which includes current system requirement specifications, lessons-learned collections and user manuals. When a developed solution is reliant on other systems it can be useful to define what user requirements are inherited between systems. Document analysis can provide information users themself do not provide in for example interviews. Sometimes users do not think of mentioning some information or they are not aware of it. By being more prepared doing document analysis can save time when performing other elicitation techniques. It is important to keep in mind the risk of using outdated documentation, requirement documentation may not have changed in pace with the functionality

of the system. (Wiegers and Beatty 2013)

Maybe the most obvious way to gather system requirements from a user is to simply ask them. Interviewing is one of the most common techniques for performing requirement elicitation. When new to an application domain interviews with experts are great ways to get accommodated with a system quickly. Interviews are also suitable for eliciting requirements from executives who have a limited amount of time to spare. When conducting interviews as a developer it is often good practice to suggest ideas. Rather than simply transcribing what the user says take a more active role. Sometimes users do not understand what the developer is capable of and the interviewer can therefore share suggestions for new functionality. This could lead to the user opening up and viewing the system in a more open-minded way. (Wiegers and Beatty 2013) Interviews can be closed or open, but in reality, it is often a mix between the two. Closed interviews have a set of pre-defined questions and are usually more structured. Open interviews have no pre-defined agenda and the interviewer explores multiple subjects. The interviewee is allowed to a larger extent to dictate the pace and topics of the interview. (Sommerville 2011)

In some cases, it may be difficult to get users to describe their role in working with a system. This is usually because of two reasons the first being that it is a complex system. The other reason is that the user is so accustomed to working with the system it is hard to describe the tasks involved. Observing the user in their work is therefore a good approach and can offer a more detailed look into the tasks of the user. Since observations are time-consuming it is a good idea to break down the sessions into specific tasks. In agile projects, these tasks are often related to the next iteration of the project. Observations can be used to identify potential interview topics and problems with the current system. Observations can be silent or interactive where silent observations are more appropriate in situations where users work demands a lot of focus. When conducting an interactive observation it is good practice to clarify the reasoning behind the decisions of the user. (Wiegers and Beatty 2013)

2.3.2 Requirement classification

To create a successful software system it has to fit the user's needs and the environment in which the system operates (Ecma 2017). Software system requirements are often bundled into two categories:

1. *Functional requirements* - These requirements are descriptions of what the system should provide to the user. It could include how the system should respond to different inputs in detail. In some cases, these requirements also specify what the system should not do.

2. *Non-functional requirements* - Describes the constraints on the services and functions provided by the system. These requirements often affect the system as a whole rather than a specific function or service. Examples of Non-functional requirements are timing constraints, development process constraints and security. (Sommerville 2011)

The lines between these types of requirements are not always clear. Non-functional requirements may generate functional requirements when implemented. A requirement could impose constraints on other requirements. An example of this is a non-functional security requirement stating a system only allow access to authorized users. This imposes constraints on the system as a whole, generating functional requirements such as the system should perform user authentication. (Sommerville 2011)

2.4 System modeling

To structure the system requirements it is helpful to better understand the system architecture. In most cases, a system has to interact with existing systems which impose requirements on the new system. Different graphical models can be utilized to represent a software system. Non-functional requirements can be represented using the structure from a system model. (Sommerville 2011)

The most important aspect of a system model is to leave out some levels of detail. A model is a representation of a more complex system. A good model reduces complexity and maintains all the information from parts of the represented system. The reason for employing a system representation using a model is different for every project. Sommerville (2011) describes three different ways in which graphical models are used:

1. Facilitating discussion about a system. Used as a way to discuss design choices during the development process. The model does not have to be complete as long as it is useful. This approach is often used in an agile development process.

2. Documenting an existing system. The model can be used to document parts of a larger system. Therefore there is no need for the model to be complete, meaning representing the whole system. The model does however need to be a correct description of the system.

3. A detailed system description is used to generate a system implementation. When using a model as a basis to generate code the model needs to be complete not excluding any essential complexity of the real system. The developed program also has to be correct, meaning it has to be an accurate depiction of the real system.

2.4.1 Architectural design

Architectural design and requirement processes often overlap and an architectural decomposition is often necessary when organizing a requirement specification (Sommerville 2011). A clear representation of how the system functions are helpful when explaining requirements. The system architecture can be designed with different perspectives.

1. Architecture in the small is concerned with individual programs and how they function. The programs can then be divided into components and their features.

2. Architecture in the large is concerned with complex enterprise systems. It includes many different systems, programs and components. These systems may be situated on different computers managed by different companies. (Sommerville 2011)

For the architectural design to be understandable the level of abstraction has to be regulated with respect to the real system and the purpose of the architectural design. Block diagrams are a common way of displaying the architectural design of a system. Architectural design is a creative process and considerably affects the way a system will operate. According to Sommerville (2011) it is recommended to consider the following questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?

2. How will the system be distributed across several cores or processors?

3. What architectural patterns or styles might be used?

4. What will be the fundamental approach used to structure the system?

5. How will the structural components in the system be decomposed into subcomponents?

6. What strategy will be used to control the operation of the components in the system?

7. What architectural organization is best for delivering the non-functional requirements of the system?

8. How will the architectural design be evaluated?

9. How should the architecture of the system be documented?

When developing a new software the architecture often reflect the architectural design of the pre-existing environment. Decisions about how many cores or what classes to use fundamentally affect the application. A developer has to decide how many of these core concepts to reuse.

2.5 Prototyping

Sometimes requirements are not technically feasible due to a lack of understanding of the limitations of the operating environment. Having developers perform requirement elicitation and creating proof-of-concept prototypes may be a possible solution. (Wiegers and Beatty 2013)

> A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. (Sommerville 2011)

A prototype is an initial version of a product and there are several reasons to create a software prototype. It can be used to explore design alternatives allowing stakeholders to evaluate different user interaction methods. Used as a construction tool a prototype can work as a functional part of the final product where new parts are developed in increments. The prototype can also function as a requirements tool, obtaining and evaluating the quality of the requirements of the project. The developer averts wasting time focusing on unclear or unfeasible requirements. Using a prototype this way is especially helpful when working with risky or complex parts of a system. It is often unnecessary to prototype an entire system, but instead, focus on high-risk areas. (Wiegers and Beatty 2013) As can be seen in Figure 2.2 Wiegers and Beatty (2013) describes four categories of prototypes.

A mock-up prototype can demonstrate functionality without actually implementing it. This is common when evaluating possible user interface alternatives. It can give the user a feeling of what the program is supposed to do and how it should look. The goal is to refine requirements to save time in development. It is a good idea to make the mock-up seem as real as possible. Buttons, layout and controls should be visible but do not have to function. The information displayed in the mock-up can be hardcoded and constant to further save time and make the prototype seem more realistic. (Wiegers and Beatty 2013)

A proof of concept prototype is working the way a real system is supposed to work, focusing on parts of the system. Opposite to a mock-up, it touches all levels of the system implementation. It can be used to determine the feasibility of a solution, optimize algorithms or evaluate a database schema. For this prototype to be meaningful it has to be developed using the same tools and environment as the final product. (Wiegers and Beatty 2013) As an example, a mock-up prototype can be used to clarify requirements before completing the interface design, a proof of concept prototype validate core algorithms and the system architecture. (Wiegers and Beatty 2013)

A throwaway prototype is exploratory and will not be part of the finished product. The purpose being improving requirements quality and eliminating uncertainties. The prototype is to be discarded after use and developers try to create a working program as fast as possible. Maintaining coding standards and solid software techniques is not a priority since the product will not be expanded or maintained in the future. This type of prototype is useful when developers face uncertain, incomplete or vague requirements. (Wiegers and Beatty 2013)

An evolutionary prototype is a more robust product and the idea is to reuse some or all of the code making up the prototype in later iterations. Therefore it is important for developers to employ solid design principles, in other words, no shortcuts. The idea is to create a prototype that can grow into a finished product. The prototype is meant to satisfy a subset of the requirements. In agile projects, stakeholders and developers review the prototype and update requirements for the next iteration. A prototype at the end of an iteration can become the final product. (Wiegers and Beatty 2013)

	Throwaway	Evolutionary
Mock-up	■ Clarify and refine user and functional requirements. ■ Identify missing functionality. ■ Explore user interface approaches.	■ Implement core user requirements. ■ Implement additional user requirements based on priority. ■ Implement and refine websites. ■ Adapt system to rapidly changing business needs.
Proof of concept	■ Demonstrate technical feasibility. ■ Evaluate performance. ■ Acquire knowledge to improve estimates for construction.	■ Implement and grow core multi-tier functionality and communication layers. ■ Implement and optimize core algorithms. ■ Test and tune performance.

Figure 2.2: Different applications of software prototyping (Wiegers and Beatty 2013)

2.6 Usability

Why is user interface design important? Pressman (2001) writes:

> If the software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. Because it molds a user's perception of the software, the interface has to be right. (Pressman 2001)

Software may offer valuable functions, but without good interface design, users may not understand the application and could stop using it entirely.

2.6.1 Design principles

There are many different approaches to creating an excellent design. Human-computer interaction (HCI) incorporates several design aims that may not be obvious when considering an application interface design. A system should be accessible to different types of users. When choosing a design it is important to consider users special needs and abilities. It is easy to develop an application with maybe just one type of user in mind. It could also be a good idea to ensure the application design comply with company guidelines. A more evident aim in designing an interactive system is usability. It should be easy and efficient to perform the work to be accomplished. Therefore only appropriate information should be presented to the user. If something is unclear it should be explained to the user. Conclusively the user has to accept the software and want to use it. Acceptability does not have to do with the usability of the software but with other factors like how convenient it is to use in a specific situation. There could also be political and cultural reasons users may not accept the application. (Benyon 2014) Employees have a different attitude towards monitoring in the workplace for example. In the book Designing interactive systems by Benyon (2014) twelve design principles are presented to help guide the developer in the design process. 1 - 4, *Helping users access, learn and remember*. 5 - 7, *Giving a sense of being in control and knowing what to do*. 8 - 9, *In a safe and secure way*. 10 - 12, *Suiting the user*.

1. *Visibility* - It should be clear to the user all the functions of the system and what the system is supposed to do. It may not be feasible to make all functions visible, consider using other senses to communicate with the user. People have an easier time recognizing things than recalling them, therefore it is good to use visuals to help the user remember the system.

2. *Consistency* - Be consistent when designing a system, comparable features should function and look the same. If an application uses a progress bar for all time-consuming operations except one it may confuse the user.

3. *Familiarity* - It is a good idea to use symbols and language the user is familiar with. This could save time when navigating and learning the system. If some features can not be explained in a familiar manner try providing a suitable metaphor to improve user understanding.

4. *Affordance* - Try making functions reflect their use, making it obvious what they are for. One example is making buttons look more clickable by making them look raised from the background. It is widely known a big X represents "Close", but it is important to realize these affiliations are culturally determined.

5. *Navigation* - Make it easier to navigate the application, maps and signs can be helpful to guide the user.

6. *Control* - Clearly state the relationship between the user and the system. The system does some functions by itself, while other functions require user input. It should be obvious what effects the controls have on the system. These principles also apply to the relationship between the system and the world around it.

7. *Feedback* - The system should provide feedback to the user so they know the outcome of their actions. This will give an increased sense of control.

8. *Recovery* - Recovery from mistakes and errors should be swift and effective. Minimize the time it takes for the user to correct their mistake.

9. *Constraints* - Some actions are inappropriate and should be avoided. Design the application in a way that prevents the user from making serious errors. Introduce restrictions of unwanted actions and require confirmation when performing dangerous operations.

10. *Flexibility* - The system should appeal to users with different levels of experience. Add multiple ways of doing things and provide the user with the option to personalize the system.

11. *Style* - Design of the system should be polished and attractive.

12. *Conviviality* - Interactions from the system to the user should be polite and friendly. For users to like the system it should respond as a likeable person would. (Benyon 2014)

Tools

The tools employed in this thesis are presented in this section. Visual Studio is used as IDE for developing, testing and visualizing graphical changes to the application. Tekla Open API handles the connection to Tekla Structures and is used to extract model data and modify Tekla drawings. The API is based on the .NET framework. The main programming language used is C#.

3.1 Visual Studio

An integrated development environment (IDE) is a program used to create software. It always includes a shell where the developer writes the code and features which facilitate the coding process. (Bitesize 2021) Microsoft Visual Studio is a popular IDE and is used with the Microsoft operating system. In software development, it functions as a platform for writing, editing, debugging and building code. Applications can also be packaged and published using Visual Studio. It supports quality of life features like code cleanup, looking up definitions and suggesting code fixes. (Studio 2021)

3.2 Windows Forms

Windows Forms is a UI framework used for building Windows applications. It is integrated with Visual Studio and intuitive to use. Changes in the code can be seen updated in the form almost instantly. A form is a visual surface on which the programmer can display information to the user and let the user interact with the application. Controls can be added to the form and are discrete UI elements that display data or take data input. (.NET 2021) Controls and the appearance of the application can be modified by drag-and-drop. Every control has properties that determine attributes like size, name and events. Events handle any user input and how a specific control should act to it. The documentation on Tekla Open API classes and methods is limited, therefore the development of the application relies on a few guiding code examples. All of the example code utilizes the Windows Forms graphical class library and it seemed like a natural design choice for this application as well.

3.3 .NET framework

The .NET framework is a technology that supports running and building Windows applications and web services. It provides a consistent, object-oriented environment and builds communications on industry standards. Common language runtime (CLR) is the foundation of .NET. It is the virtual machine component of the framework and manages the execution of programs. Just-in-time compilation converts compiled intermediate language code into machine instructions. Meaning code is compiled during the execution of a program and source code is transformed before converting it to machine code. Resulting in more efficient program execution. CLR also provides memory management, type safety, exception handling, garbage collection, security and thread management. The .NET framework can be used with several high-level languages because a common type system (CTS) specifies how data types and specific values are represented in the computer memory. This enables code generated in C# to interact with for example code generated in F# and so on. A large collection of libraries are included, which support different workloads. They provide different functionality from file input and output, string manipulation, XML parsing, web application frameworks and Windows forms GUI. (.NET 2021)

3.4 C#

C# was developed by Microsoft and released in July 2000 and is intended to be suitable for large sophisticated systems as well as embedded systems with specific functions. It is an object-oriented programming language, programmers define types and their behaviour. The language is efficient concerning memory and processing power but can not compare to the performance of C or assembly languages. (Ecma 2017) It is type-safe and therefore prevent type errors, but not statically type-safe so errors may occur at run-time. C# run on the .NET architecture and include some features to help create robust and durable applications. Garbage collection reclaims memory from unreachable, unused objects. Nullable types protect the program from crashing when variables refer to unallocated objects. Exception handling provides a mechanism to handle error detection and recovery. C# also supports asynchronous operations which allow developers to use threading and create distributed programs. (.NET 2021)

Types are an essential concept of C#. A type defines the structure and behaviour of any data in C#. A variable is an instance of a type and inherits the structure and behaviour of that specific type. C# has two kinds of types, value types and reference types. Variables of a value type contain the data. Variables of a reference type only reference to the data. The data of a reference type is known as an object. Since a reference type does not contain the data, two variables can perform operations on the same object. C# is unified meaning a value of any type can be treated as an object. Every type does derive from the object class type. Values of value types can be treated as objects by performing boxing and unboxing operations. An example is given below where an int value is converted to object and back again to int. (.NET 2021)

```
int i = 123;
object o = i;      // Boxing
int j = (int)o;    // Unboxing
```

An example of a value type is the enumeration type (enum type). It is defined by a set of named constants of the integral numeric type. The integers start at zero by default and increase by one. Each integer then has an associated value. The System. Enum type provides several methods for interacting with a specific enum. The values of an enumerator can not be changed. Data can however be read and converted to a different data type. (.NET 2021)

A class is an example of a reference type. It has to be defined using a unique identifier name. An instance of a class is called an object. Classes are commonly used to define a type of object. The class does not contain the data of the object, but only instructions of how the object is to be defined. Behaviour and data are defined using methods, properties and fields and are also referred to as class members. (.NET 2021)

A method works similar to functions in other programming languages. It is a way of executing a block of code with a required set of arguments. Every executed instruction in C# is performed in the context of a method. A field is simply a variable declaration inside a class. It is generally used when variables should have restricted access. (.NET 2021)

Properties provide a more flexible way of reading, writing or computing the value of a private field. They can be used to specify the way data is accessed outside a class. This is achieved using special methods called accessors. Properties can be read-write meaning they have both a *get* and a *set* accessor. Read-only meaning they have a *get* accessor but no *set* accessor. Finally, they can be write-only meaning they have a *set* accessor but no *get* accessor. Properties are commonly used to restrict access to sensitive data. (.NET 2021)

Classes support inheritance meaning classes can inherit data and behaviour from other classes. This is accomplished by declaring a base class from which another class inherits all its members except for the constructors. (.NET 2021)

A constructor is a method that defines the creation of new instances of a class. It is signified by having the same method name as its type and does not have a return type. (.NET 2021) Instances of a class are referred to as objects. A class does not need to have a constructor method.

3.5 Tekla Open API

In this section some of the most essential methods and classes of this project are presented. Tekla Open API is an Application Programming Interface, developed using the .NET Framework which allows the user to create applications to communicate within the Tekla Structures modeling and drawing environment (Structures 2021). Tekla Open API allows the user to extract model data using a set of enumerators with a logical hierarchical structure. Any change to the model database is mostly done with the methods:

- *Select()* for object selection
- *Insert()* for creating a new object instance
- *Modify()* for modifying an object
- *Delete()* for deleting an object (Structures 2021)

To use Tekla Open API Tekla Structures has to be running in the background. The user can then retrieve information from the 3D-model using the enumeration interface. Every object in the 3D-model is a member of a class, often with subclasses. The hierarchical structure of the classes is modelled as a logical translation of the Tekla Structures graphical interface. For example, if a user of Tekla Structures modifies the colour of a beam by highlighting the beam and changing a value in a text field, using the API this is done by changing the beam object property colour.

3.5.1 Tekla.Structures.Model Namespace

This namespace includes functionality to select, insert, modify or delete objects inside 3D-model. It is also possible to access different kinds of data from the current model like accessing information about currently selected objects. (Model 2021)

The assembly class used when creating assembly drawings is described as follows:

> The Assembly class defines a single manufacturing unit: objects that are bolted or welded together in the workshop. An assembly has a main part and secondary parts attached to it. The number of secondaries is limited to 2048. Hierarchical assemblies can also have sub-assemblies attached to them and they can have a parent assembly instance. (Model 2021)

Some methods are available to extract data from the assembly object using the assembly properties. It is also possible to add and remove parts of the assembly. The assembly class is a subclass to the modelobject class. It inherits most properties and identifiers from this class. Most classes inherit some properties from the systemobject- and database object class as can be seen in Figure 3.1.

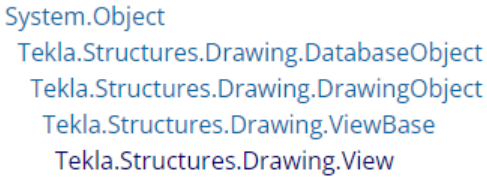

Figure 3.1: Inheritence structure of the view class

3.5.2 Tekla.Structures.Drawing Namespace

A Tekla drawing is generally made up of a few main classes. The "blank paper" of the drawing is called the Sheet which constitutes the size of the drawing. Placed on the Sheet are the Views displaying the assembly from different angles and perspectives. Figure 3.2 is a railing and consists of beams, steel plates and bolts which all belong to the class Parts. The measurements (Dimensions) are displayed as rulers and specify the length and angles of the Parts. Marks are used to identifying each Part and are inherited from the assembly. Every part in the drawing inherits most properties from the modelobject class. The modelobject class contain identifiers to keep track of what object in the drawing belongs to what object in the 3D-model.

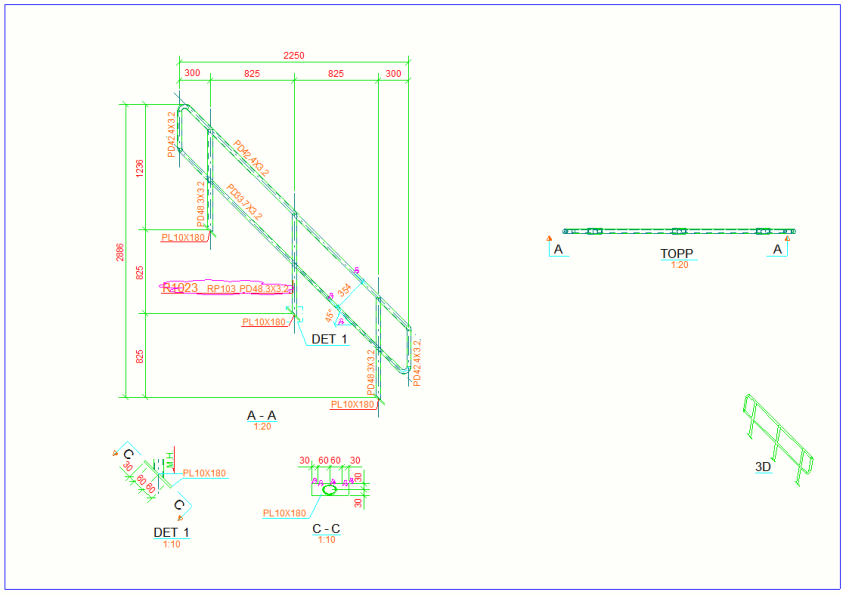

Figure 3.2: Tekla Structures drawing example

3.5.3 Tekla.Structures.Drawing.View

Views are essentially a 2D-representation of a collection of objects, seen from different angles. The ViewType property can be used to keep track from what side a collection of objects are being displayed in a view. Views can also have different scales which work as a zoom tool. This is used to reduce the number of objects seen in the view and put more focus on certain objects. Every view has a Viewtype property that cannot be changed after the view is created and can have one of ten values:

1. *UnknownViewType* - The view type is not known.
2. *FrontView* - Front side view of the part.
3. *TopView* - Top side view of the part.
4. *BackView* - Back side view of the part.
5. *BottomView* - Bottom side view of the part.
6. *EndView* - End side view of the part.
7. *SectionView* - Section view of the part.
8. *ModelView* - Model view of the part.
9. *DetailView* - Detail view of the part.
10. *_3DView* - 3D view of the part. (Drawings 2021)

Most ViewTypes are self-explanatory TopView is the part seen from the top based on the local coordinate system in the 3D-model (Orientation 2021). All views have a 2D perspective when representing the different parts except for the _3DView. The parts in this view are rotated to give the impression of a 3D perspective and are mostly used to better understand the composition of the parts displayed in the drawing. The last ViewType worth mentioning is the DetailView which is a cropped view of certain parts in another view. This is used to augment details of certain parts in that view. In Figure 3.2 A-A is a front view, TOPP is a top view, DET 1 is a detail view, 3D is a 3D-view and C-C is a section view of the assembly. Detail view functions in this case as a zoom tool aggregating some information in a specific area of the assembly. The section view has the same purpose but shows a cross-section of the selected parts. 3D-view is only used to give an overview of the shape of the assembly.

3.5.4 Tekla.Structures.Drawing.Part

Every *Part* is a visual representation of a modelobject in the Tekla Structures 3D-model. Each Part has an ID and this can be used to retrieve additional data from the Tekla drawing. A Part is a subclass of ModelObject and each instance of the part class can also be a member of seven subclasses:

1. *Beam* - The Beam class represents a single beam in the model. A beam has a single start and endpoint.
2. *BentPlate* - A class for the bent plate.
3. *Brep* - The Brep class represents a single brep in the model. A brep has a single start and endpoint.
4. *ContourPlate* - The ContourPlate class represents a part made with a contour, such as a concrete slab.
5. *LoftedPlate* - This class represents a lofted plate.
6. *PolyBeam* - The PolyBeam class represents a continuous beam with a contour as input.
7. *SpiralBeam* - A class for the spiral beam part. (Drawings 2021)

A typical railing is generally made up of several Beams, some kind of BentPlate and bolts to fasten the railing. BoltGroup is modelled as a subclass to ModelObject and is therefore not a member of the Part class. The seven subclasses all have unique properties. If for example a part class object is converted to a beam class object the properties StartPoint and EndPoint may be accessed. Consider a beam which is essentially a cylinder with a length and width. The properties StartPoint and EndPoind define the length of the object. This is not true for the BentPlate class where these properties would be confusing to use. Instead, this type of part has geometry and thickness properties, which is a better way to describe the look of a BentPlate.

3.5.5 Tekla.Structures.Geometry3D

The Geometry3D namespace contains geometric classes used by Tekla Structures. Euclidean space and distances between objects are defined using these classes. Some methods and classes are available to make it easier to perform operations like rotations and coordinate system transformations.

Points and vectors in Tekla are represented by the Point and Vector classes. These classes function exactly as any point or vector in euclidean space. Some convenient methods include *GetNormal()* which returns a normalized vector of the current vector, *Dot()* method which returns the dot product of two vectors and *Cross()* method which returns the cross product of two vectors. (**geometry3D**)

3.5.6 Tekla.Structures.Geometry3D.CoordinateSystem

The CoordinateSystem class defines a coordinate system in space. The class constructor takes in three arguments. First, a Point specifies the origin of the system, then a vector defining the X-axis and another vector defining the Y-axis. No vector designating the Z-axis is needed since it is defined as the cross product of the X-axis and the Y-axis. (**geometry3D**)

3.5.7 Tekla.Structures.Geometry3D.Matrix

Tekla uses matrices for coordinate transformations. The matrix class in Tekla Open API represents a 4x3 matrix. Every item in the matrix object can be viewed or changed using the Item property. A point can be converted to a different coordinate system using the *Transform()* method. (**geometry3D**)

The MatrixFactory class gives a convenient way of generating matrices for different coordinate system transformations. The *ToCoordinateSystem()* method returns a transformation matrix of the given coordinate system. This method is a fast way to transform points into a new coordinate system. A convenient way of rotating objects is using the *Rotate()* method to transform coordinates. The method takes in two arguments, a number defining the angle to be rotated and a vector specifying the rotation axis. (**geometry3D**)

3.5.8 Tekla.Structures.Drawing.StraightDimension

Dimensions in Tekla drawings display distances between two points. It is a subclass to the DimensionBase class which is a base class for all different types of dimensions. Other dimensions like AngleDimension measures an angle between two vectors or three points, but most commonly used is the StraightDimension class. (Drawings 2021)

New instances of the StraightDimension class can be created using the class constructor and can be seen in Figure 3.3. The method first argument is the target view for the dimension, defining what view it should be created in. Arguments two and three are the start- and endpoint for the dimension, defining between what points the dimension should be drawn. The fourth argument defines the up direction and it is a vector. It functions as a reference line for the start- and endpoints. The last argument defines the height of the dimension. Height is used to force a distance between a part in the drawing and the dimension associated with that part, with height set to 0 the dimension would be drawn on top of the part resulting in a cluttered drawing (Drawings 2021). When the object is created it can be saved to the Tekla database using the *Insert()* method. To change an existing dimension the *Modify()* method is used and they are removed using the *Delete()* method.

C# VB

```
public StraightDimension(
        ViewBase targetView,
        Point startPoint,
        Point endPoint,
        Vector upDirection,
        double distance
)
```

Figure 3.3: StraightDimension constructor in C#

3.5.9 Tekla.Structures.Drawing.Mark

The Mark class is a subclass to the MarkBase class. Just as the StraightDimension class it is a drawingobject. The Mark class inherits most properties from the MarkBase class. In Tekla, this class represents a text field that can contain information about for example the ID of a part. (Drawings 2021)

Every mark has to be associated with a modelobject in the 3D-model. Since this is the case the Mark constructor only takes in one argument namely modelobject. The mark will by default be positioned in the middle of that object. The properties InsertionPoint and Placing define the positioning of the Mark object. The attributes property defines the look and content of the text field. When adding text to a Mark a new TextElement has to be added which holds the string to be displayed in the Mark object. Marks updated to the database the same way as dimensions. (Drawings 2021)

Development

In this chapter, the steps of the development process are presented. In early development, defining the system architecture and its requirements were the main focus. During development, these requirements were continually updated as the understanding of the system increased. Different prototypes were created to better understand the system and visualize the final product.
In later stages of development, the code generated from the prototypes was reused later in the project and the final product. Exploring new solutions was a lower priority and more testing was performed on existing components. Evaluating the design and functionality of the application was then done to ensure requirements were being met.

4.1 Choosing methods

Tekla Structures is a comprehensive and complex software. To interact with the system a user has to have some basic knowledge of the logic of the system architecture. As proposed by Swecos representatives a flexible approach of observations and interviews was adapted (Jonsson 2021). Where software specialists would demonstrate a task in Tekla Structures and describe every step in detail using the share screen function in Microsoft Teams. The demonstration could be paused at any time for clarification. Open interviews were interwoven between observations to elicit requirements and discuss design choices. Observing is a potent strategy to extract information that is obvious or intuitive to the participant.

The function of the first and second prototype is as a requirements tool, establishing new requirements. They are also used as a fast way to determine the feasibility of a solution. The third prototype is more used as a construction tool with code being reused in the final version and parts being created in increments. The fourth prototype is used as a way to evaluate user interaction methods.

4.2 Understanding the system

Several meetings are conducted with a Tekla Structures expert at Sweco to better understand the system. These meetings consist of demonstrations using the video communication software Zoom. One example of these sessions is a demonstration of the drawing cloning process, with both defective and correctly cloned drawings. When comparing the generated drawing to a manually edited drawing the differences are clear. We would like to explore the possibility of generating a drawing that resembles the manually edited drawing as closely as possible (Kristiansson 2021). Two completed drawings were to be used as a template for a successfully generated drawing.

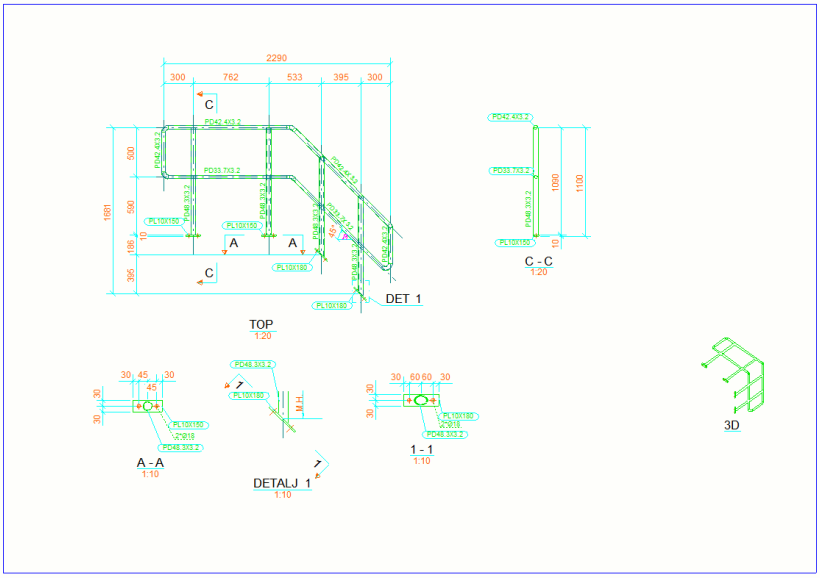

Figure 4.1: One of two Tekla Structures drawings template

The Tekla algorithm which creates the 2D-drawing from a 3D-model is essentially a black box. Since Tekla Structures is a comprehensive and complex piece of software, it is generally a good idea to work with simple assemblies. When selecting an assembly in the 3D-model, it is done using the cursor, selecting the desired parts in the model and clicking them. However, when clicking, a filter is applied to the selection, resulting in the user being unable to clone. It is also important to double-check if the selected parts make up an assembly object before trying to clone, which can be difficult to do without the necessary experience using Tekla Structures. Other factors which could prevent a user from cloning are freezing and locking the drawing.

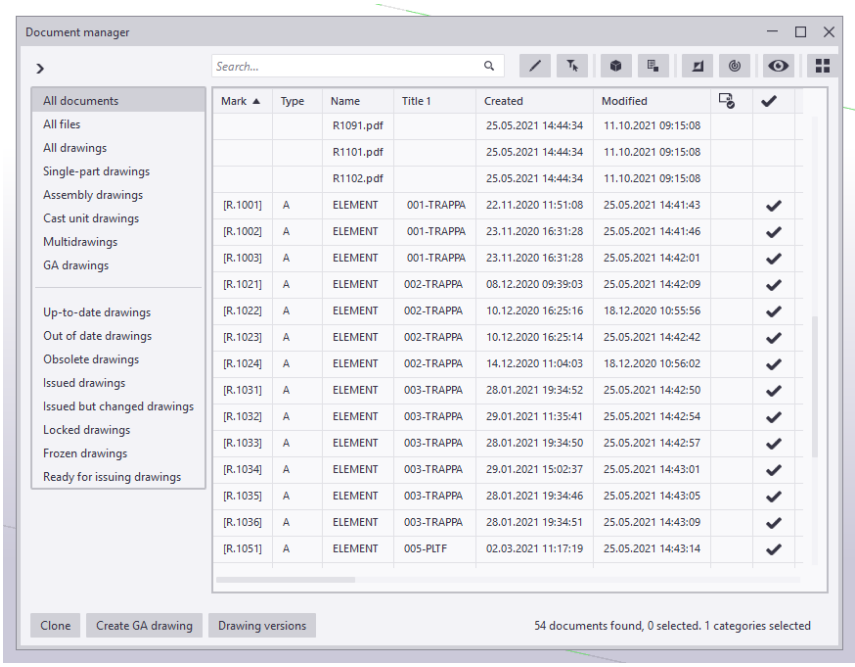

Figure 4.2: Tekla Structure Document manager

Two selections have to be made before cloning a drawing. The first is the selection in the 3D-model to choose what parts to display in the drawing. The second selection is to tell Tekla which drawing to use as a temple for the generated drawing. This is generally done by choosing a drawing in the document manager in Tekla Structures. The document manager is a window containing a list of all available drawings of the

project. It is used for accessing, modifying and deleting drawings. The document manager gives the user the functionality to lock and freeze drawings. Locking a drawing is used to notify users that the drawing should not be opened, edited or deleted and it stops people from doing this by accident. Freeze is used to preventing unwanted changes in associative objects on top of the drawing views. It also prevents changes in annotation objects like dimensions, marks and view properties (Structures 2021). Another function that can prevent the user from cloning is numbering. All the parts must have updated numbering for a clone to be created.

> Numbering is the key to the production output, for example, drawings, reports, and NC files. Numbers are also needed when you export models. Part numbers are vital in the fabrication, shipping, and erection stages of construction. Tekla Structures assigns a mark to each part and assembly/cast unit in a model. The mark includes part or assembly prefix and position number, and other elements, such as profile or material grade. It is useful to identify the parts with numbers to see which parts are similar and which different. Identical parts within a numbering series have the same number, which makes the planning of the production easier. (Structures 2021)

An ID is associated with every assembly. Copying and modifying assemblies in the 3D-model could result in a duplication of assembly ID, which also prevents the user from cloning.

4.3 First prototype

A proof of concept prototype is created to explore the many functions of Tekla Open API. Nearly all documentation of the API is found on Teklas main website. The Tekla Structures namespace is organized hierarchically, encompassing the relationship between classes, objects, methods and properties. Code examples and snippets are also provided for some of the methods.

The purpose of the application is to reduce time spent on manual editing when generating a drawing. This occurs when the assembly is too complex or when there is too much difference between the assembly and the cloned assembly. The result is that the Tekla Structures cloning algorithm performs in an undesirable way. The natural approach is to explore if Tekla Open API has the functionality to change the way the cloning algorithm performs in any way.

The idea is to mimic the process of cloning in Tekla Structures as closely as possible but tuning the cloning algorithm with an external application. First, the user has to select what assembly to create a drawing from in the 3D-model, then the drawing to be used as a template in the document manager in Tekla. This is the same process as when cloning the conventional way. Instead of clicking clone in Tekla, the user clicks on a clone in the application, replacing the last step in the cloning process. Since Tekla Open API uses the .NET framework and the application was to be as simple as possible Windows Forms is used as a UI framework.

Tekla Open API has the functionality to get the selected assembly from the 3D-model as an enumerator of all parts selected utilizing the IEnumerator Interface. Note that the API treats the selected parts as a list of objects and not an assembly object. The application used the currently opened drawing instead of checking for which drawing the user selected in the document manager, this was a faster solution.

The reason for creating a proof of concept prototype was to explore the API. No indication of functionality connected to modifying the way Tekla clones drawing was found. However, once a drawing is created the API provides more leeway in terms of modifying the objects of that drawing.

When creating the prototype a few requirements came to light. Both Tekla Structures and Visual Studio has to be opened. This could be because Tekla requires a licence and this is possibly a way to provide means of authentication for both Tekla Structures and Tekla Open API. Visual Studio having to be open has to do with the way a coding project is made and can be disregarded once an executable is created. As mentioned a licence has to be purchased before opening any Tekla projects and can be regarded as a requirement of this application as well. Tekla Open API is installed in Visual Studio using a NuGet package.

> Put simply, a NuGet package is a single ZIP file with the .nupkg extension that contains compiled code (DLLs), other files related to that code, and a descriptive manifest that includes information like the package's version number. (NuGet 2021)

The package is installed fitting a specific Tekla Structures version, meaning the developer has to choose which version to create the application for. Since Tekla Structures has to be open for any of the APIs methods to work, choosing the right version is essential.

4.4 Choosing the right Drawing

After creating the first prototype and starting to understand the scope of the project a decision was made to make contact with a frequent user of Tekla Structures. This person is a structural drafter at Sweco and has in-depth knowledge of the Tekla work process. Some tasks are very similar and take up a lot of time. The assemblies used have minor differences, but the cloning still fails, leading to a lot of manual editing (sweco2). To clarify the goals and requirements of the project it was decided to focus on just one type of drawing. The type of drawing used had to be useful, meaning the type of drawing had to be frequently used in assignments by the users of the application. Secondly, the drawing had to be clear, load fast and easy to test.

A decision to work with railings is made. A railing is commonly used by the users, it is also clear in the sense that a railing consists of simple geometrical shapes and is therefore easy to grasp how it should look like. A railing is also a relatively small assembly, meaning the program has fewer objects to iterate through when testing. Seeing that the railing consists of a few part types it would be easy to test, with one exception. A drawing view displays the assembly from different angles. Imagine a railing fastened on the side of some straight stairs. When standing at the top of the stairs it is very difficult distinguishing every individual part of that railing. Therefore to test the application several views have to be present in the drawing.

4.5 Second prototype

Tekla Open API has a lot more functions related to manually editing a drawing once it is created. The second approach was to generate a faulty Drawing in Tekla Structures using the cloning algorithm. This involves manually selecting an assembly from where to create the new drawing in the 3D-model, then selecting a Drawing to clone from in the document manager and clicking clone in the document manager. Using Tekla Open API objects in the drawing can then be deleted, modified or new ones created. This works well if, for example, only the dimensions are faulty. If it is feasible and saves a lot of time for the user this would be a good design choice. According to Sweco Kristiansson (2021) Modifying Dimensions requires a lot of manual editing and is therefore time-consuming. Designing an application focusing on this specific task seemed therefore like a good strategy.

A proof of concept prototype was created to explore the functionalities of the API to modify a pre-existent drawing. Every object in a drawing can be accessed with the drawing namespace. They are available using a set of enumerators. Every object is treated as a drawing object but can be selected by type. It is, therefore, possible to select only modelobjects of type dimension. The selected objects can then be converted to dimension objects with methods and properties related only to that object group available. Dimensions can easily be deleted this way with the *Delete()* method, interacting with the Tekla database. For the changes to be visible to the user the drawing has to be updated. This is done by saving the drawing, closing and opening it again.

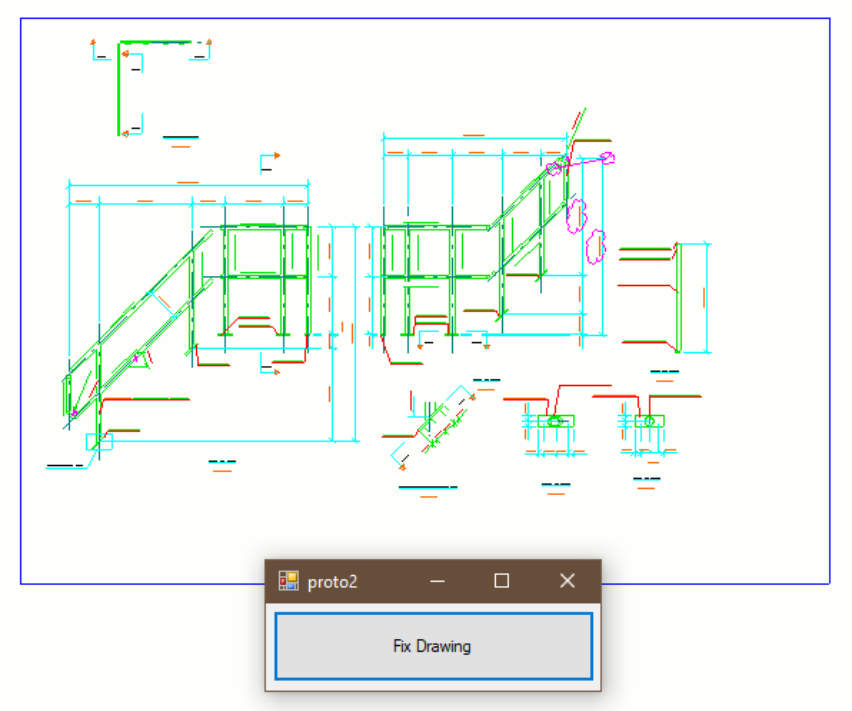

Figure 4.3: Second prototype with a Tekla Structures drawing in the background

The next step is to create new dimensions. There are different kinds of dimensions, the most commonly used when working with railings is the straight dimension. A straight dimension object is created by first specifying which view the object should exist in, a startpoint, an endpoint, a vector indicating what is considered the up direction and the height of the dimension. The height is used to enforce a distance between a part and the dimension belonging to that part. Both points are coordinates with X, Y and Z components. The vector has the same three components. The newly created dimension is then saved to the database using the *Insert()* method. For the user to acknowledge the changes the drawing has to be saved, closed and opened again using methods related to these actions in the API.

Dimensions start- and the endpoint has to be related to a part in the drawing for it to have meaning. The purpose of a dimension is to display critical information to the user about the measurements of a specific part. Since the points of the dimension can be set when creating a new dimension the next step is to map the coordinates of the points to the coordinates to the parts in the drawing. The 3D-model and the 2D-drawing share coordinates. The views simply show the parts from different angles, but the parts maintain their coordinates. A railing largely consists of beams which is an underclass to the modelpart class. Every modelobject can be selected using an enumerator and parts can be sorted out. Once converted to parts only beams can be selected. The upside to the beam class is that it has a start- and endpoint property, meaning it is possible to map a dimension to the start- and endpoint of every beam.

This prototype proved it is possible to modify a pre-existing drawing and modify its dimensions in a meaningful way. Important to note is that for excellent dimensions to be created, suitable data of the parts used in the drawing is required.

4.6 Stakeholder requirements

Stakeholder requirements are the functional requirements of this project but have to be broken down to be more understandable. The creation of prototypes made it easier to understand the system, but also realize the goals of the stakeholder Sweco. A better understanding of the system was achieved by extracting knowledge with the help of prototypes and asking more and more specific questions about the system as the project progressed.

During the first meeting with Jonsson (2021) at Sweco, they were uncertain of the feasibility of the aims of this project. If it is possible then we would like you to do it (Jonsson 2021). The question then becomes how to translate the stakeholder goals to functional requirements. The approach used was to look at the goals and deconstruct them. The template drawings are the goal and also the output of a Tekla Structure process. The drawing also has to look a certain way to satisfy the stakeholders. This is achieved using the Tekla cloning algorithm. The requirements are:

1. The system shall clone a drawing.
2. The cloned drawing should resemble the template drawings.

By activity observing and interviewing a Tekla user and learning more about the system through prototyping these requirements could be improved upon. After evaluating the second prototype it was decided to explore the feasibility of the program also create a new drawing. By using prototyping the process of cloning drawings is now clearer. The Tekla cloning algorithm cannot be changed as learned from the first prototype. It is, therefore, necessary to change the requirements. It was learned from prototype two that new drawing objects can be created in a meaningful way. The objects created in prototype two are of the class Straight Dimension, the same as in the template drawings. The dimensions created does not resemble the dimensions in the template drawings. Style, colour and positioning of the dimensions have to change for the creation of dimensions to be useful. The objects created in the drawings has to reduce the amount of manual editing compared to a drawing cloned using the Tekla cloning algorithm. A great deal was learned from watching users at Sweco edit drawings. Manual editing as a process involves using the mouse to create new objects or modifying objects in the drawing. The positioning of dimensions takes a long time to correct (Kristiansson 2021). Every object class also has unique properties which means the user sometimes have to open another window to change these properties using a set of lists and boxes. Creating, repositioning and modifying drawing objects are the most time-consuming tasks. By reducing these actions the user saves time in manual editing. Using the application has to be faster than the Tekla cloning algorithm. This involves the manual editing associated with using the application. The requirements of the system can now be broken down into the following:

1. The system shall create a drawing.
2. The system shall create the right type of objects in the drawing.
3. The system shall create the right number (or close to) of objects of each class.
4. The system shall reduce the need for repositioning objects.
5. The system shall reduce the need for modifying object properties.

The term "right type" or "right number" is subjective and has to be evaluated based on the type of drawing being created. The evaluation is done by comparing the created drawing to the template drawings.

4.7 Modeling cloning of a drawing

A model of the cloning process is created to later clarify and define the non-functional requirements of the system. The model is read from left to right and starts with the action of opening the 3D-model. Every box in the model is called an element and every element is an action performed in the Tekla Structure environment. The user now has to select an assembly in the 3D-model, then select a cloning template in the document manager. Some options are available, dimensions can for example be excluded from the cloning. Then the user clicks clone and the drawing is opened. It is common for the user to perform some editing before verifying if the generated drawing is acceptable. Manual editing takes considerably more time than the other activities.

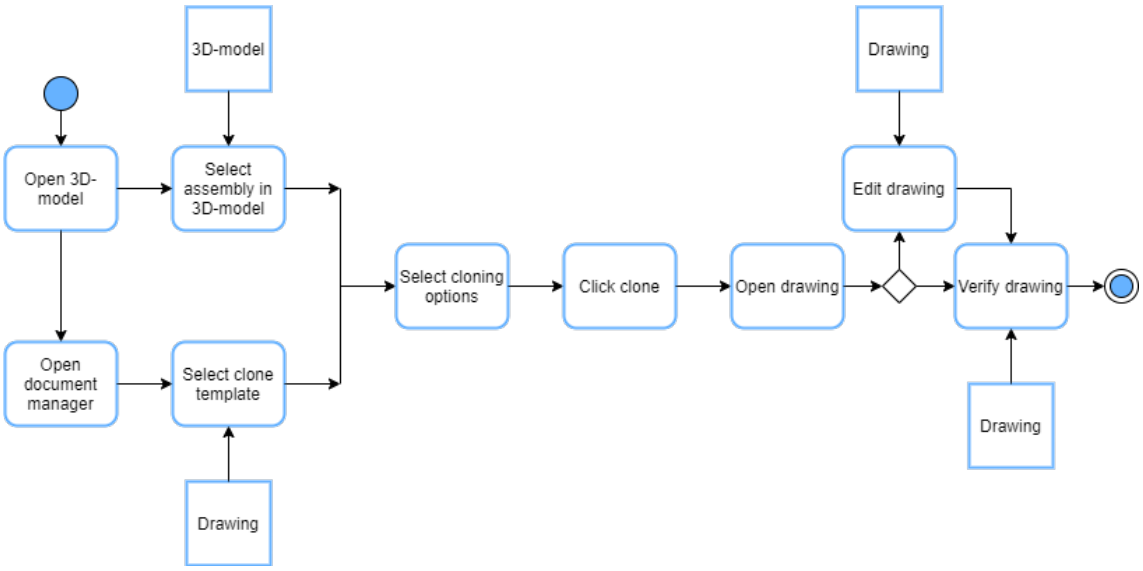

Figure 4.4: Model of cloning process in Tekla Structures

From observing users perform the task of cloning and creating two prototypes non-functional requirements of the system can now be established. The model is a simplification of the Tekla cloning process, there is a chance some information is lost. Nonetheless, some kind of structure is needed to categorise the non-functional requirements of this system. The application will have to interact with this system and in one way have to abide by these requirements.

Starting at the beginning a 3D-model has to be active in Tekla which means Tekla Structures also has to be opened. There also has to exist a drawing to use as a cloning template. When selecting a cloning template drawing the document manager has to be opened first. The cloning template drawing has to be closed before performing any operations on the drawing. Locking and freezing a drawing does not affect the ability to use it as a cloning template.

The last prerequisite for the cloning options to be available for the user is the selection of an assembly in the 3D-model. The selected parts have to be of the same type as the template drawing used. Since the application should clone assembly drawings, the selection has to be an assembly and have an ID. As learned from prototyping numbering has to be performed before cloning is allowed by Tekla Structures.

Select cloning options, clicking clone and opening drawing are all made in the document manager. In the model, we assume the drawing always has to be verified for the process to be completed. Editing is treated as an alternative and not mandatory for the cloning process, but in practice, this is always needed. The reason behind this modeling decision is the ambition to eliminate the need for editing entirely as expressed by Jonsson (2021). The cloning process as a whole takes minutes or seconds to complete without editing, but considerably longer with editing.

It is important to keep in mind the implications of a skilled user performing the operations in this model. Tekla Structures is a comprehensive and complex software as mentioned before. There is not a lot of hand-holding features in this application and the difference in speed performing certain tasks is considerable. Plainly the ability to quickly edit a drawing makes a great difference, but other elements cannot be overlooked. The ability to understand why some operations are not allowed by Tekla can save a lot of time and can be hard to learn without proper experience. Also choosing the right drawing template and cloning options can be a time-saver and reduce the amount of editing needed.

4.8 Requirements of the application

The requirements affecting the application are considered. By reviewing the documentation on the Tekla website, they provide developers with the means to interact with Tekla drawings using Tekla Open API.

Since creating a Tekla Structures drawing without the use of the application Tekla is impractical, using the API should be a requirement. All API documentation is provided in C# developed using Visual Studio. No requirements concerning timings and the speed of the program are expressed from the stakeholders. But for the program to be useful, it has to be an alternative to the Tekla cloning process. For it to be an alternative it has to be faster than this process, it is assumed the difference in speed between common programming languages is negligible. Sweco uses Windows as OS on their computers used with Tekla. It should therefore be a requirement for the program to use C# and Visual Studio. As learned through prototyping Tekla Structures has to be opened to use the API which is a requirement on the application side. It is also important to note Tekla Open API is sensitive to what version of Tekla Structures the user is running. Lastly, a license is required to use the Tekla Structures program and should therefore also be a requirement for the application.

By using the model in the previous section it is possible to classify a set of non-functional requirements. In this thesis, they are called Tekla environment requirements. Examples of these types of requirements are the need to select an assembly in the 3D-model and part numbering. The process of developing the application could eliminate some of these requirements as knowledge of the system increases.
Tekla Open API is not as dependent on some elements in the cloning process. For example, the document manager does not have to be opened to access drawings. From prototyping, it was learned it is necessary to use some key methods to update a drawing and objects associated with this drawing. These are constraints on the application, concerning Tekla Open API. These types of constraints are classified as operational API constraints, specifying how the API has to be used. Other constraints concerned with the API are classified as feature API constraints. One example of these types of requirements is the inability to change the way the Tekla cloning algorithm works, the feature is missing in the API.

The user has a central role in the creation of drawings. The way a user behaves affects the speed of creating and the quality of a drawing. It should therefore be a requirement for the application to be easy to use. So far the following non-functional requirements are:

1. The application has to work with Windows operating system.
2. The application has to be developed using C#, Visual Studio and Tekla Open API.
3. A license is needed to use the application.
4. Using the application cannot be slower than using the Tekla cloning process.
5. The application has to abide by Tekla environmental requirements.
6. The application has to abide by operational API constraints.
7. The application has to abide to feature API constraints.
8. The application has to be easy to use.

The reason for defining non-functional requirements and writing them down is to use them as a guide when developing the application. It is easier to make decisions about the application architecture with fewer possible design paths to take.

4.9 Architectural decisions

Some kind of graphical user interface or GUI is needed since ease of use is a requirement of the application. From the functional requirements, the application has to create a new drawing. The new drawing also has to mimic some methods used in the Tekla cloning algorithm like creating views and copying view properties. A second drawing has to be used to copy these properties.

The approach used when deciding on the architecture of the application is to integrate some of the elements in the cloning process model into the application. Using this approach makes it easier to abide by Tekla environmental requirements since the application can maintain the structure of the cloning process. The user has to interact with the application and Tekla Structures to perform the creation of a new drawing. Some elements will never be integrated into the application. A drawing should be created after the selection of an assembly and cloning template, to follow the structure of the cloning process. Therefore it seems natural to integrate the following elements, namely selection of cloning options, clicking clone and opening of the

drawing. What cloning options to include depends on how well the program performs. Providing more options could save time in editing, but can be confusing and overwhelming to the user. Providing more options could be easier to implement since there are fewer functions to automate. All communication with Tekla Structures is done with the use of Tekla Open API or user input. The architecture is illustrated below:

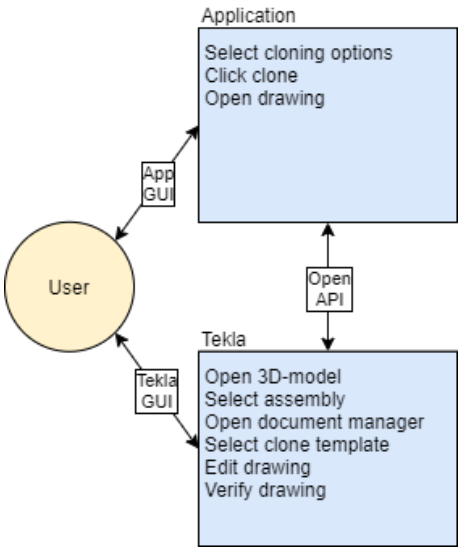

Figure 4.5: Suggested architectural design of application

More elements can be integrated later in the development process. Some elements like verification most likely have to be done in the Tekla environment since it is a Tekla drawing being verified.

4.10 Third prototype

The third prototype is an evolutionary, proof of concept variant and is expected to be reused. It is used as a construction tool to help expand the application with new functionality and a way to measure progress. According to Kristiansson (2021) the application should create drawings from scratch. It saves a lot of time not having to manually clone a drawing. This will be the first requirement to handle since all other functionality has to derive from the created drawing. The program also has to be built in a way to simplify testing. This is achieved with unit testing. Creating, saving, closing and opening drawings slows down the program as learned from the second prototype. When testing it is therefore sensible to minimize these tasks. Since parts of the code are to be reused in later stages of the project it is important to keep the code clean. This includes explanatory and unambiguous methods- and variable names and keeps the code design consistent.

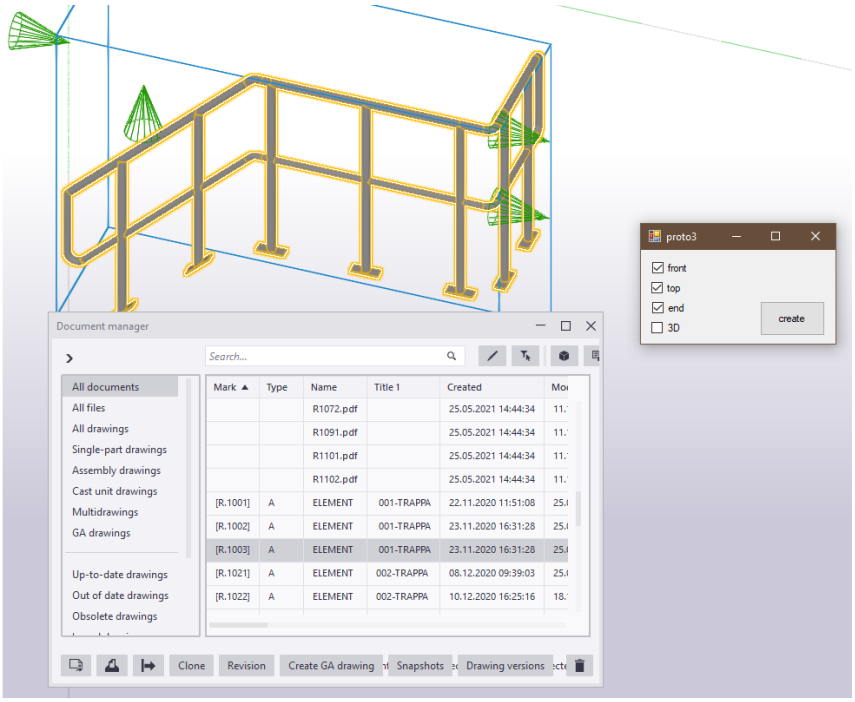

Figure 4.6: Third prototype with Tekla Structures 3D-model and document manager to the left

In Tekla Structures every assembly has to have an ID, in Tekla Open API this is called an identifier. The assembly drawing class constructor takes a minimum of one argument called assemblyIdentifier. This is the assembly ID from the 3D-model in Tekla Structures and specifies what assembly to create a new drawing for. This process fails if there exist two or more identical IDs:s in the 3D-model or the ID does not belong to an assembly. The application has to check if these two conditions are met before proceeding. A sheet object is created with the drawing on which views can be placed. A default view is also created, but can easily be deleted. As seen in the template drawings the most commonly used views are top, front, end, 3D and detail view. It was decided to focus on the first four views since they take in similar arguments. The view class takes in a minimum of four arguments. A ContainerView which in this case is the sheet of the drawing on which the views are to be placed. A CoordinateSystem specifies the boundaries of the views. A second CoordinateSystem determine how the objects in the view are placed. This is used to rotate objects and view them from different angles resulting in different views. Lastly, an ArrayList is a collection of the model objects displayed in the view. The model objects in this case are obtained from the manual selection in the 3D-model. A selection of an assembly in the 3D-model has to be made before the program is run.

After updating the drawing all the views are inserted but overlapping. This is solved with the method *PlaceViews()* which intelligently determines a distance between the views and arrange them on the sheet. The template drawings have different names, colours and fonts for each view. A default name change is made for each view. Another drawing is used as a drawing clone template from which to copy some key characteristics. Since the colours and fonts of the views in the Swecos template drawing are the same any view in the drawing clone template can be used. The view class has a property called attributes which contains definitions for font and colour. This property can simply be copied and used in the newly created drawing. Some attributes are not applicable to transfer from one drawing to another, therefore only font and colour are transferred.

Dimensions are created for every view except the 3D-view. They are created using the straight dimension class constructor. Startpoint and endpoint of the dimensions are determined by the startpoint and endpoint of the beam class from the 3D-model. A constant distance is specified to each dimension to prevent them from being placed on top of each part. A dimension object from the template drawing is used to copy properties to change the colour and style of the dimensions. Each dimension is then inserted and the drawing is updated.

During the development of the third prototype, one method was especially helpful. The *Model.CommitChanges()* method commits the changes made to the database so far. When creating new objects and inserting them it is required to call the method at times. Tekla Open API will output an arbitrary exception if too many

inserts are done in succession without calling the method.

Part marks are created for each beam in the drawing. They are placed between the startpoint and endpoint. Part marks are used to identify each part in the drawing. The text of each part mark is copied from the class modelobject which has the modelIdentifier property. A mark is copied from the template drawing and the properties like colour, type and arrowhead are cloned. The arrowhead property is the style of the arrow pointing to the part.

The prototype has a small GUI with some essential options available to the user. The goal of the application is to save time in the process of cloning drawings. Reducing the number of clicks the user has to perform saves time. This is notably true for this application where some tasks are performed repeatedly. As can be seen in the drawing templates and by observing users perform the process of cloning most drawings use at least three different types of views. It is therefore decided to pre-check all the textboxes to reduce the number of clicks the user has to perform.

Familiarity is an important design principle to consider when creating a GUI. It helps the user navigate the system. Even though the prototype design is fairly simple since this is an evolutionary prototype it is advisable to be more thorough when considering design choices. Working with windows forms have several advantages in this aspect. The intended users of the application are working within a Windows environment. The architecture of a window with all control elements placed on it should therefore not be a novel concept. The symbols for minimizing, maximize and close are also familiar to Windows users. Checkboxes are frequently used in both Windows and Tekla and the simple design is somewhat self-explanatory.

4.11 Preparing the 3D-model for testing

Several 3D-models are available for testing the application. Every 3D-model has a related drawing. The 3D-models in this case are several railing assemblies. A 3D-model is chosen and copied, then the copy is used to generate a new drawing using the prototype. The more the generated drawing resembles the original 3D-models drawing the better the performance of the program. A similar drawing is used as a cloning template and object properties are copied from this drawing.

In the beginning simpler 3D-models of railings were used to speed up testing and minimize time spent on error correction. After the initial tests larger and more complex railings were used. Railings could for example be longer and curved.

4.12 Integrating document manager

To simplify the process of creating a drawing the document manager was integrated into the application. The design principles state the importance of control and navigation. With an integrated document manager it is more clear the user has to specify a cloning template drawing. The alternative would be to describe to the user how to select a drawing in Tekla Structures and guide the user in the right direction. This step is now simplified by using an explanatory text above the document manager. The need for good navigation is thereby reduced by changing a design choice. One final advantage of this design is the ability to pre-select drawings in the list. This could save time when using the same drawing as the cloning template repeatedly.

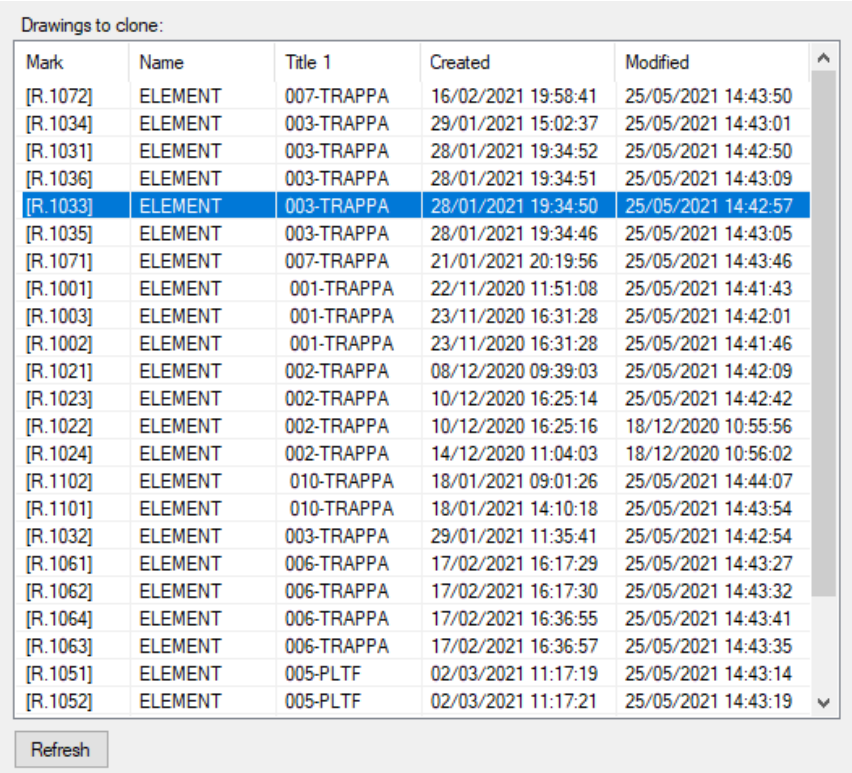

Figure 4.7: ListView element with explanatory text and refresh button

Windows Forms has the ListView class providing a control that displays a collection of items. The users of the application are going to have some experience using the document manager in Tekla Structures. It seems therefore reasonable for the ListView to resemble the document manager while at the same time remembering the consistency design principle. The category names are the same as in the document manager, the same applies to the overall structure of the window. What is changed from the document manager are the colours, fonts and style of the windows to improve consistency with the rest of the Windows Forms window. The option to open the created drawing was also integrated according to the suggested architecture.

4.13 Handling exceptions

This section will describe the development of exception handling. An exception is generated when the program encounters an unplanned event. Most of the time this is due to an invalid input, for example trying to perform an undefined operation on an object. Understanding how tasks in Tekla Structures are performed greatly helps when troubleshooting the application. When getting an exception from working with Tekla Open API, the explanations are often short or non-existent. Sometimes the most meaningful information from an exception is learned from looking at the different exception class names. The exception generated from trying to create a view with unnumbered part objects generate a *CannotPerformOperationNumbering-NotUpToDate* class exception.

Handling exceptions is important for the program not to crash and can be avoided with proper exception handling. This is done by specifying a set of operations to be performed in case of an exception. Handling exceptions when working with Tekla Open API can be done when performing sensitive operations like creating new dimensions. Exception handling can also be improved by utilizing the different exception classes. Using exception classes can give both user and developer more explanatory feedback.

The program can encounter unplanned events without an exception being thrown. Depending on how the program is designed this could result in nothing happening and or the program freezing. Handling exception is therefore not only dealing with exception classes but in code expressing a set of operations for every potential path. In this application, it is a good idea to look at the cloning process model for a guide to good exception handling. Some pre-requisites are needed for cloning to take place. First Tekla Structures

has to be opened to use the methods of the API. This is done with using the DrawingHandler class and the *GetConnectionStatus()* method. Since an assembly drawing is to be created the selected parts all have to belong to the same assembly and only one assembly. The program also has to check if the user selected the cloning template in the document manager. Lastly, the program has to deal with the numbering exception mentioned before in this section. Once a drawing is created the methods for creating objects in the drawing are dependent on a drawing to be open. Since these operations take some time it is possible for a user to accidentally close the drawing. The program, therefore, checks if the correct drawing is opened at the start of each method involving the insertion of objects in the drawing. This is done with the DrawingHandler class *GetActiveDrawing()* method.

Reflecting on exception handling is not only relevant to program performance, but the usability of the program. According to the ninth usability principle, some actions should be avoided. Some actions could be harmful to the user, application or other systems. Examples of this include causing a crash of Tekla Structures, corrupting the 3D-model or drawings used. The .NET framework and Tekla Open API prevents some harmful actions from being made through a controlled runtime environment and restricted file access. It is however possible to delete weeks of manual editing with one line of code using Tekla Open API which is good to keep in mind when constraining user access.

Feedback and recovery are two important design principles to be mindful of when working with exception handling. Feedback is self-explanatory, the system should provide feedback to the user so they know the outcome of their actions. Exception handling should therefore come with some kind of message, explaining what the user did wrong. It should be obvious what action led to the message being displayed to the user. Recovery means resolving errors should be swift and effective. In practice, this means when the user does something wrong time lost should be kept at a minimum. Reducing the number of tasks the user has to redo when making a mistake is desirable. For example, do not force the user to restart the application when getting an error message. It is also important to allow the user to redo all tasks necessary, so as not to force the user to redo the same mistake again resulting in a soft loop.

4.14 Improving usability

The GUI appearance is inspired by the document manager. Users of the application are already familiar with Tekla Structures and should therefore swiftly get accustomed to this design. Some explanatory text is still needed to clarify what the ListView element is used for. The checkboxes from the third prototype are reused, but the text is modified to be more clear. The design principle of conviviality is considered when producing text directed at the user. The language should be polite and friendly. It is assumed the user know the different ViewTypes referenced in the checkbox texts. The most important element is the create an assembly drawing button which is made in a way to improve the affordance principle. It looks clickable, raised from the foreground, it is also a bit bigger to signal its importance. Elements are placed on the window in a certain way. People in the western world read from the upper left to the lower right. Elements are placed on the window in the same manner. The user is supposed to use the elements in the upper left area of the application before clicking the button in the lower right.

A refresh button is also included for the user to be able to open the application before the document manager can be accessed. The application tries to refresh the list without the user needing to push the button when the application is run. This is to save the user some time since selecting a drawing in the ListView element is a mandatory operation. The button is also placed close to the ListView element to signify what the button does.

As mentioned before it is important to constrain the user from making errors. Unpredictable errors can occur when operations are cancelled, resulting in an unintended action. When the user clicks the create assembly drawing button it takes several seconds to generate a drawing and open it. Changing options in the window or closing it can result in unexpected errors. Two mechanisms are implemented to prevent this. When the button creates assembly drawings is clicked every control element on the window is greyed out, inactivating the use of them. The user can still close the application using the X in the upper right corner. Not allowing the user to close the application seems like a bad idea since it would be frustrating and the operation of creating a drawing cannot be cancelled any other way. At the same time, it is unwanted for the user to accidentally close the application. This is prevented by the second mechanism, a progress bar indicating a drawing is being made. The progress-bar replaces the create assembly drawing button to clearly state it has a relation to the drawing being made. Considering the design principle of control the relationship between

the user and the application should be clearly stated. The user cannot do anything until the progress-bar is done processing. It also gives necessary feedback on what is happening. This will reduce the risk of the user closing the program prematurely thinking it froze or thinking a drawing already was created.

4.15 Faster testing

Many procedures were streamlined by reducing unnecessary tasks and automating some manual work. A set of boolean variables were used as breakpoints to control what parts of the program to test. It made it possible to just insert dimensions without cloning by changing one boolean. This was a useful way to isolate errors in the code.

The application involves some manual tasks using the interface, minimizing these tasks are also made possible with a few boolean variables. Firstly the generated drawing is deleted every time the program is run. Secondly, the views created is predetermined and the drawing is created without using the interface. Thirdly the drawing used as the cloning template is predefined and reused every time the program is run. Every procedure to minimize execution time is valuable since drawing creation often exceeds ten seconds.

4.16 Created drawings

In this section, the drawings generated from the program are presented. The order a drawing is generated is by first creating a drawing, adding views, cloning views, creating dimensions, cloning dimensions, cloning parts, creating marks and cloning marks. The objects are inserted or updated in the order they are presented and can be seen in Figures 4.8-4-14. Functionality is added to the program in increments. Developing a method for creating dimensions had higher priority and these objects are therefore created before marks. When create assembly drawing is clicked the drawing created is opened. The user can see every object inserted and modified in the drawing while the drawing is being created. The methods *Delete()*, *Modify()*, *Insert()* and *Model.CommitChanges()* are used to update the changes to the Tekla database.

4.16.1 Create Drawing and Views

An assembly drawing object is created, by default a view is inserted with every assembly type drawing. This view is then deleted and the views checked in the GUI are inserted instead. The views created are using default view properties and are arranged on the drawing sheet using the *PlaceViews()* method.

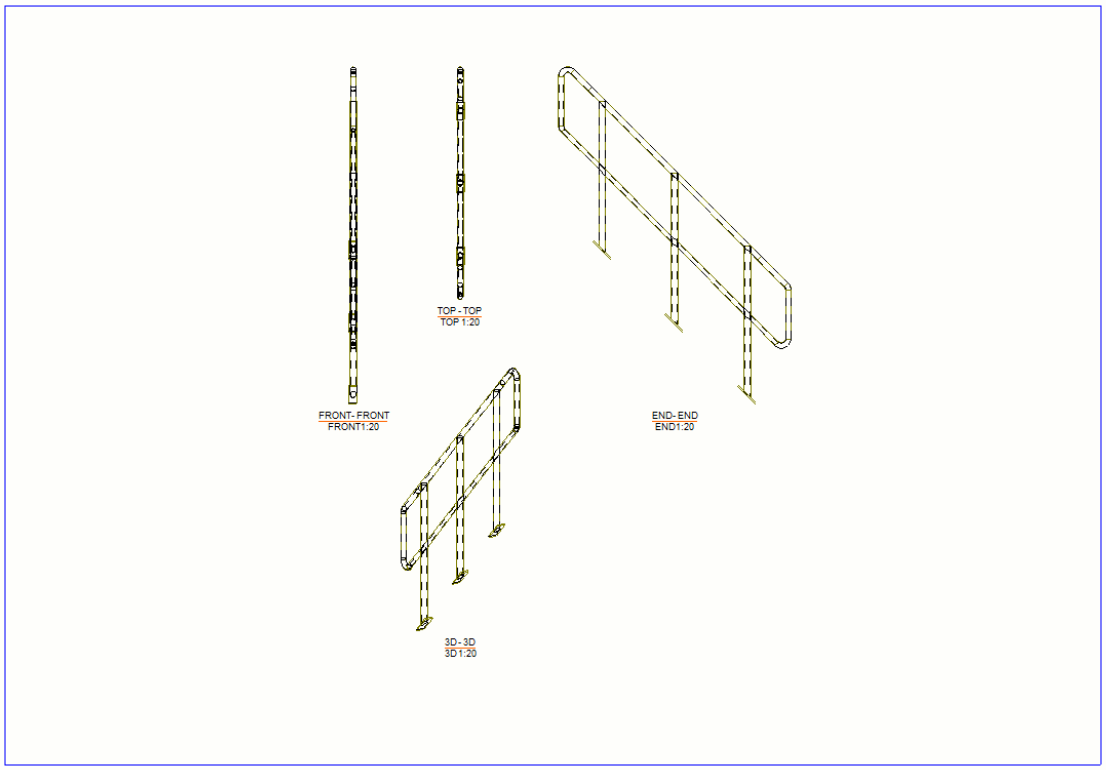

Figure 4.8: Created drawing with views

4.16.2 Clone Views

View properties change the appearance and content of the text under each view. These properties are copied from the selected drawing in the GUI. Every view is then updated with new properties.

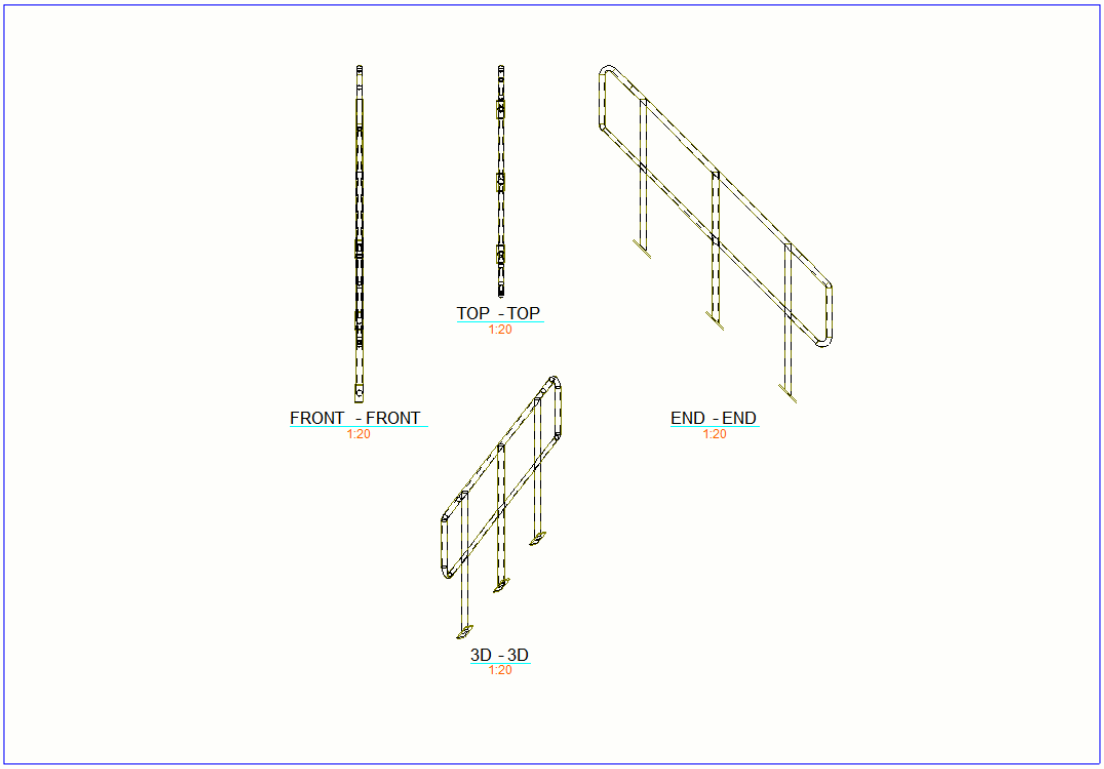

Figure 4.9: Cloned Views

4.16.3 Create Dimensions

Dimensions are created for each beam object. A longer dimension is inserted to show the length of all vertical and horizontal dimensions. A static distance property is set for every dimension. No dimensions are created for the 3D-view.

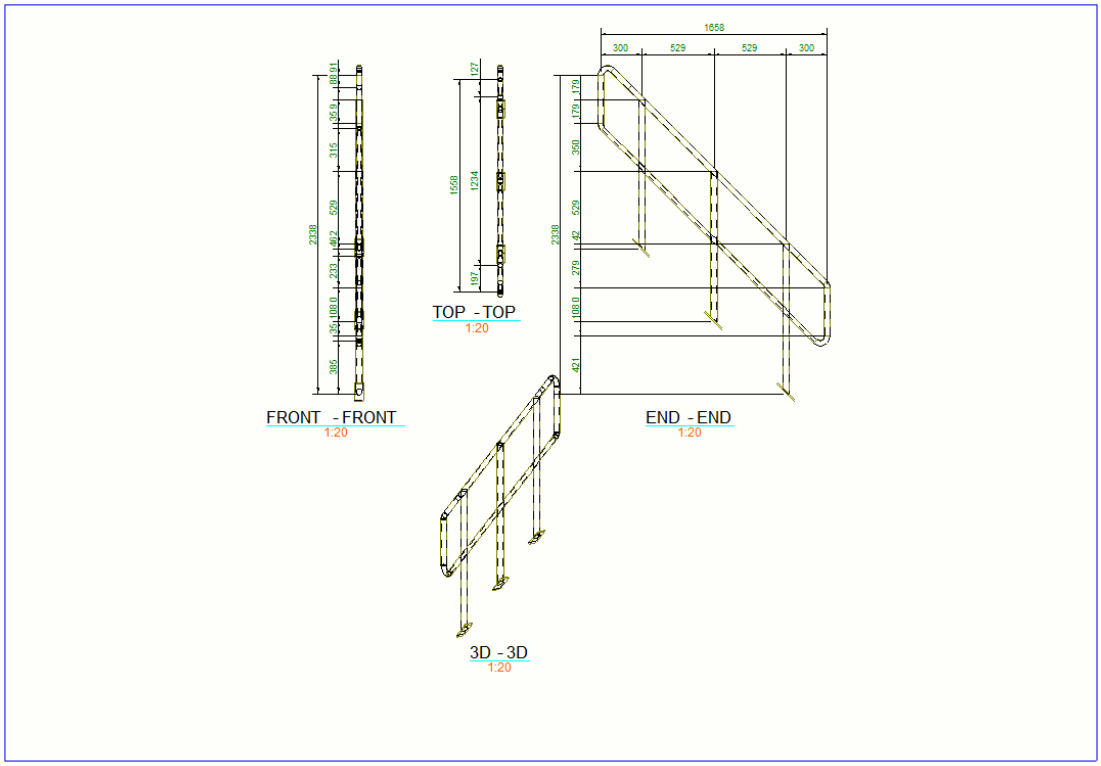

Figure 4.10: Created Dimensions

4.16.4 Clone Dimensions

Dimension properties are copied from the selected drawing in the GUI. Both style and color is updated for every dimension object.

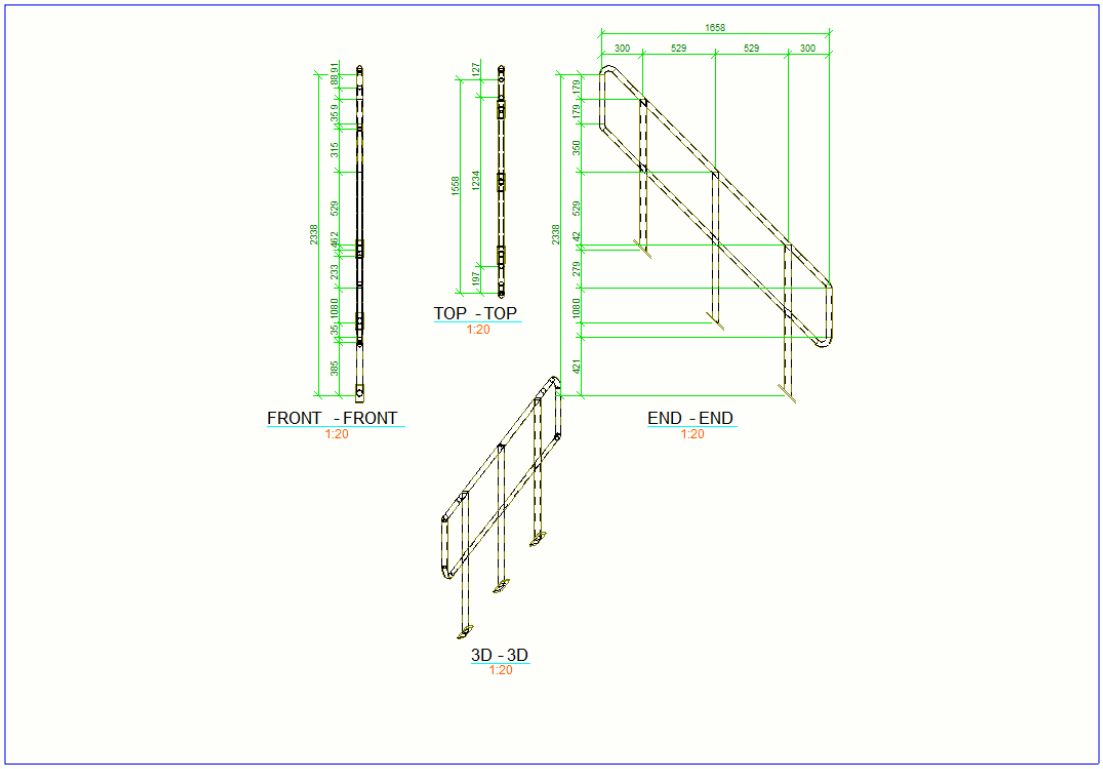

Figure 4.11: Cloned Dimensions

4.16.5 Clone Parts

Drawing part objects are copied from the drawing selected in the GUI. Every part object is updated for every view.

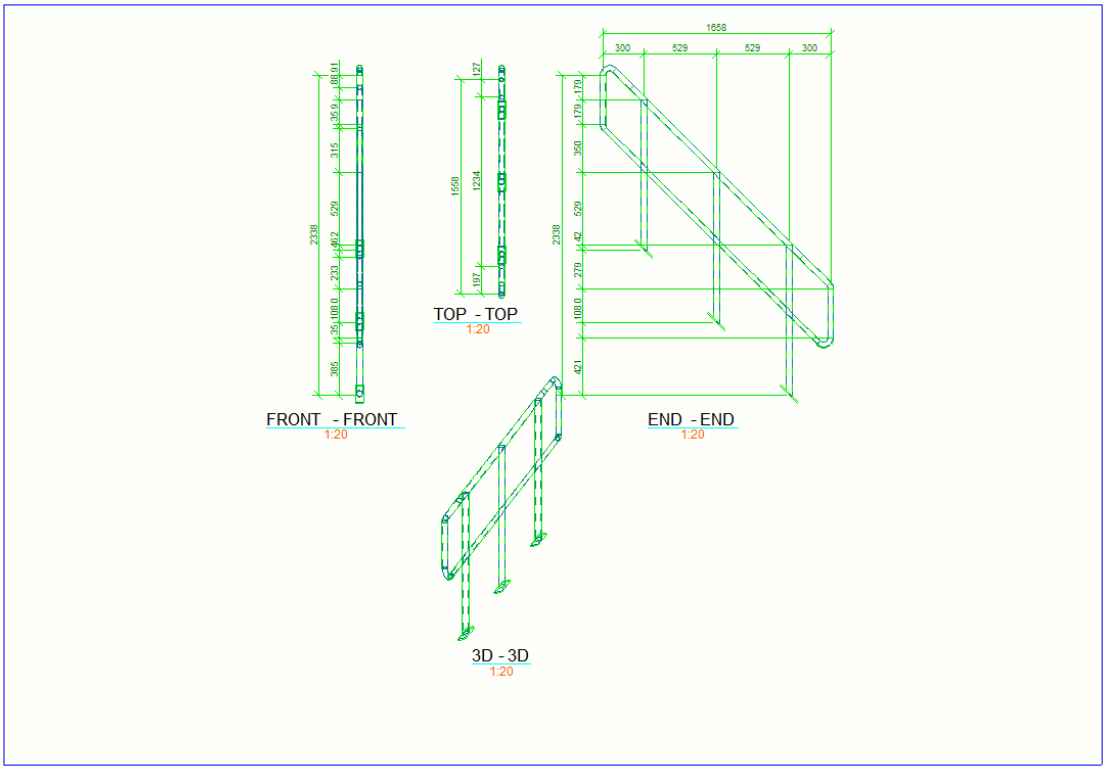

Figure 4.12: Cloned Parts

4.16.6 Create Marks

Part marks are inserted in the middle of every part in every view except the 3D-view.

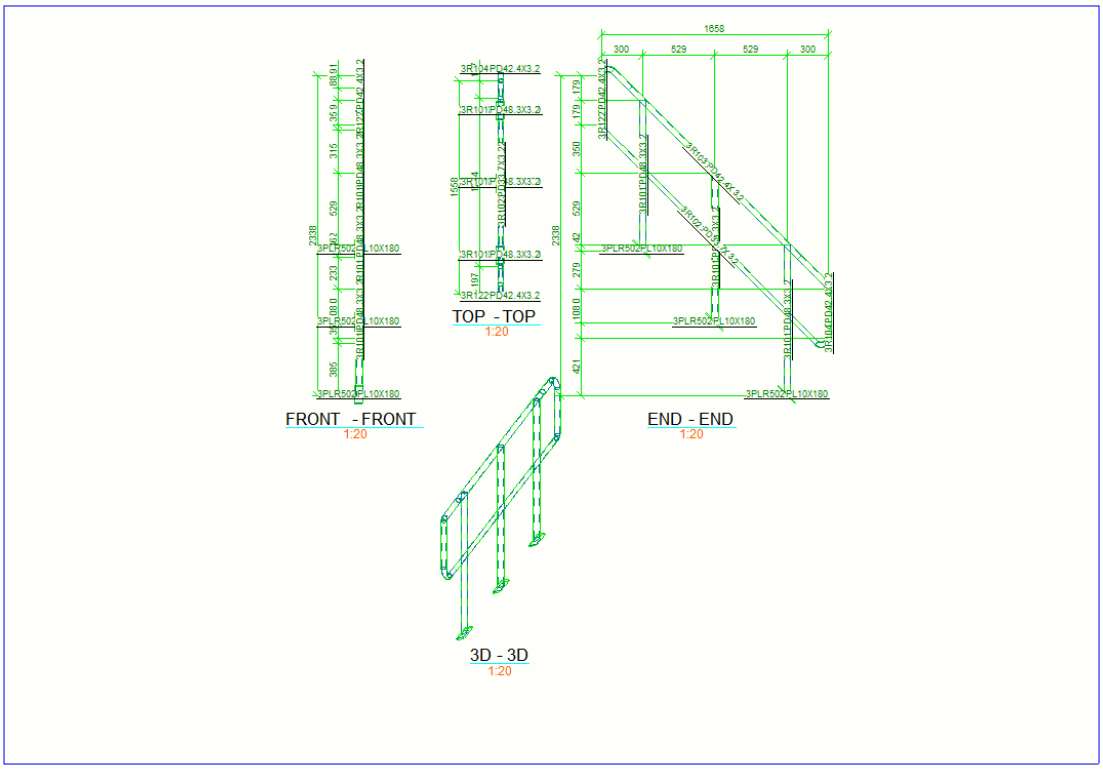

Figure 4.13: Created Marks

4.16.7 Clone Marks

Marks are cloned by copying mark properties from the drawing selected in the GUI.

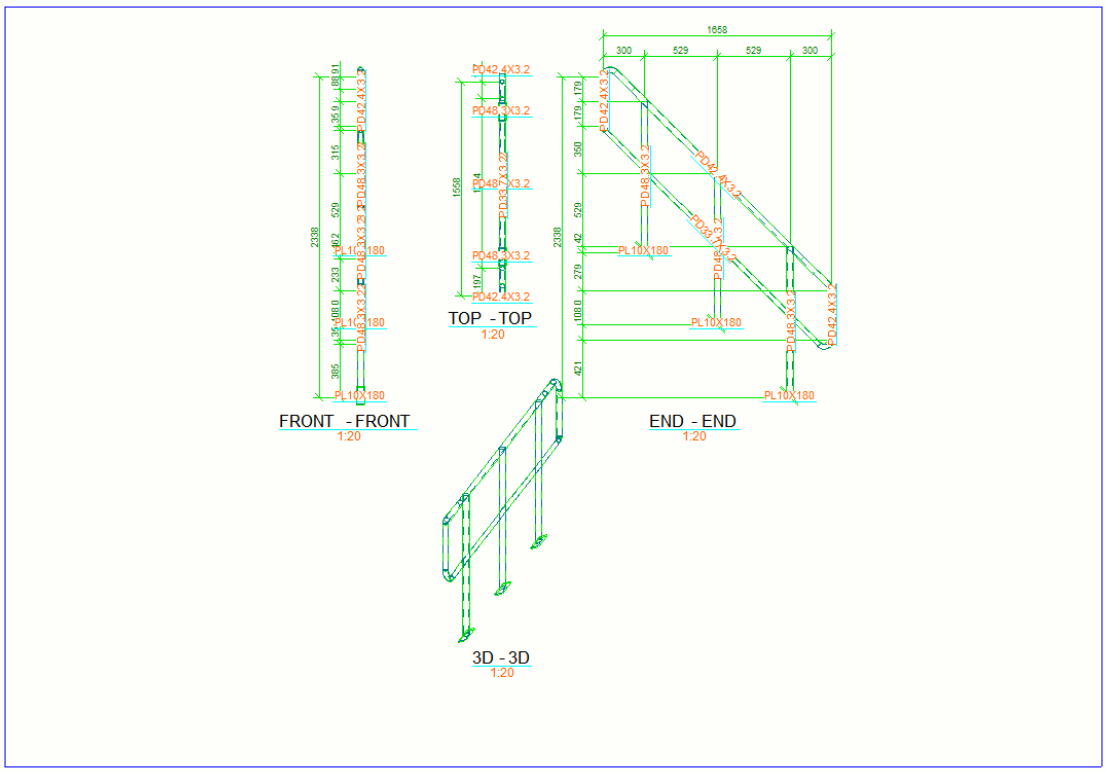

Figure 4.14: Cloned Marks

4.17 Validation

Sweco has been available for observations and questions concerning Tekla Structures. This has been very helpful when defining system architecture and requirements. Greater insight into the Tekla environment allowed for the requirements to be refined. This was done by asking more specific questions about the system.

With a more complete application requirements can be validated. A user at Sweco was given a demonstration of the application. He was then asked questions about the design and functionality of the application, a list of the questions can be found in the appendix. The user has experience using Tekla Structures and has cloned drawings many times.

This demonstration included the whole process of cloning a drawing using the application. Starting with opening Tekla Structures and preparing an assembly in the 3D-model. After running the program some intentional mistakes were made to check how well error handling is performed. Numbering was not done before trying to create a drawing and a message appear describing the problem. Any changes are reverted and the user performs numbering in Tekla Structures. With every checkbox and a template drawing selected in the ListView, a drawing is then created. The user can see the objects being inserted and modified in the drawing as the progress-bar fills up taking about ten seconds.

4.17.1 Evaluating GUI

For the application to achieve the non-functional requirement of being easy to use, certain design principles have been utilized. During the demonstration and interviews, questions based on these principles were conducted and can be found in the appendix. The interviewee is an engineer working with Tekla Structures daily and was conducted using the video communication software Zoom. The results are summarized and presented in this section. The evaluated design can be seen in Figure 4.15.

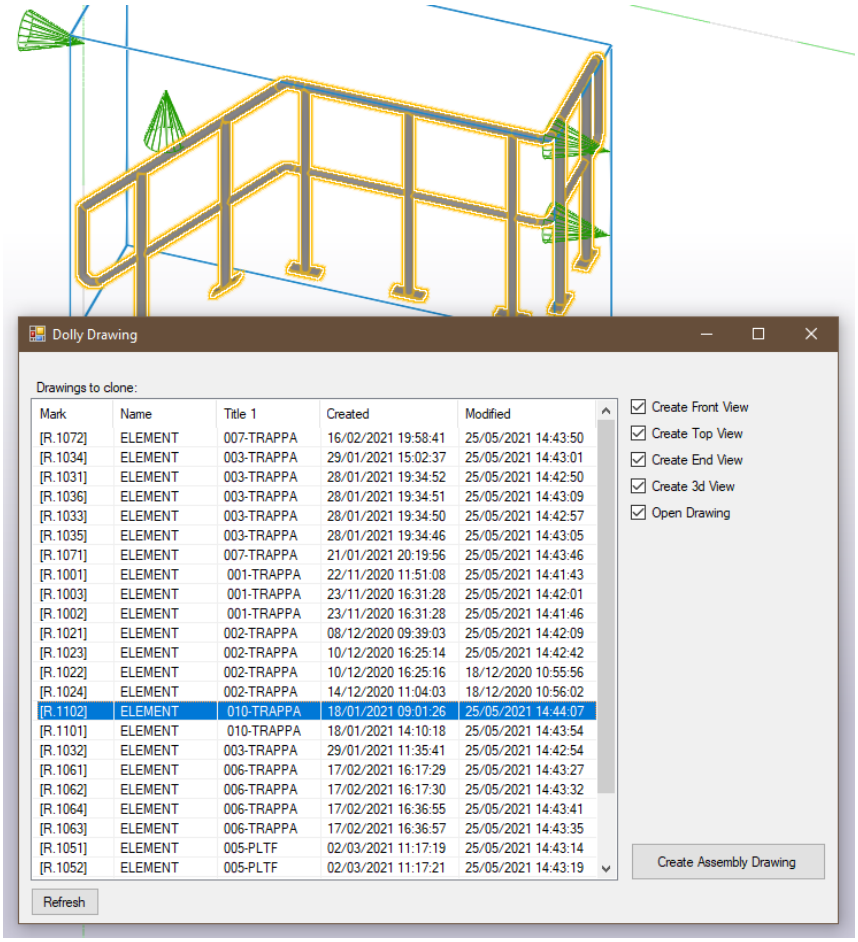

Figure 4.15: Integrated document manager with Tekla Structures 3D-model in the background

The first principles concern helping the user access, learn and remember. The application did abide by these guidelines according to the Tekla users except for the ListView element. It is easy to understand the symbols used in the application. Users at Sweco are used to working with the windows operating system. The symbols used in a Windows Form application are by default similar to that of Windows. The user explains the design is consistent and it is clear what purpose the elements in the window has. It is easier to be consistent with a relatively simple and small application. There was however some confusion about the ListView element, the purpose of it was not clear enough. The engineer says there is a lot of information to take in and it is not obvious it replaces the document manager.

Moving on to questions concerning giving the user a sense of control and knowing what to do. The application does fulfil the recommendations. The users found the application easy to navigate. Again this is not a common problem with simple applications with only one window. Except for potential bugs, the user cannot be disoriented. The logical placing of elements is enough to achieve an easily navigated application. As can be seen in Figure 4.15 the user first chooses a drawing in the list, then select options in the top right and lastly clicks the button to the bottom right. According to a Tekla user, the aspect of control was clearly stated when using the program. What the program does and what the user has to do is straightforward. Again this probably has to do with the simplicity of the application. However, users not familiar with the Tekla cloning process may not share this attitude.

One concern regarding the ListView element was raised in the interview. When working with large scale projects the list can be very long. According to the engineer at Sweco, it would be helpful for the element to provide feedback on what type of drawings are being displayed. The ListView element only displays drawing objects and not PDF-files for example. This is not clearly stated by the design. In Figure 4.15 there is no column showing the type of the files in the ListView element. Since the list shows every drawing, imagine if a project contains thousands of drawings the list would be very long and the scrollbar to the right of the list would be minimal and hard to use.

Concerning the principle of program safety, the application did perform well. The Tekla user liked how mistakes are treated by the program. A pop-up with an error message is reminiscent of working with both Tekla and the Windows operating system and can be seen in Figure 4.16. The style of the application is appealing. Again it does remind the user of working with Tekla Structures. Since all users will have some experience using Tekla, evaluating if the program is usable by different kinds of users seems unnecessary. The users did not suggest any wrongful actions during the evaluation, except for the actions done on purpose during the demonstration. Recovery from mistakes is done by a simple notification and explaining to the user what caused the message to pup-up. As said before the users liked this simple approach.

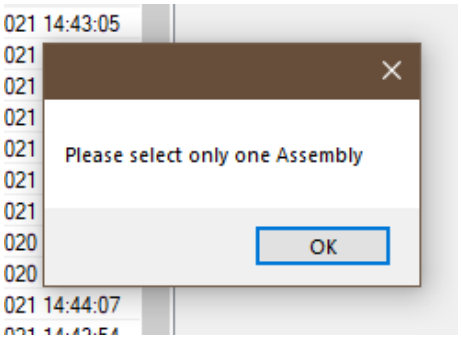

Figure 4.16: Example of application error message

The interviews ended with a more open discussion about what they would change. The user's main concern was to improve the ListView element. Clarifications of the content being presented and simplifying the design would result in a better design. For example, some columns are not necessary and are only making the design more cluttered. Clearly stating what type of objects is being displayed is another fix being discussed during this interview.

4.17.2 Evaluating functionality

While the first half of the interview touched on usability the second half is only directed at the functionality of the application. Questions asked are derived from the functional requirements of the system. The way the requirements are formulated is more directed at a developer deciding on different design decisions. Therefore the questions asked has to be formulated in a way to suit a user of the Tekla environment and not a developer. Since a Tekla user knows very well what a dimension and view is this terminology can be used. Phrases like classes and objects are not used since it is probable this would only be confusing to the participants. The questions concerning the functionality of the application can be found in the appendix.

According to the engineer who is a frequent user of Tekla the options provided are good, but some options are missing. Two views are missing, BottowView and DetailView, these are used frequently in drawings. It would also be good to be able to scale a view to enhance some details and discard others. It would also be beneficial to be able to decide on where views are placed before creating the drawing. The user had no objections to adding more options, this would not be confusing and it is better to create a more correct drawing at the cost of spending more time in the GUI.

One concern mentioned during the first half of the interview was the problem with the ListView element, this also has to do with the functionality of the application. When projects are large the document manager can hold an extensive amount of drawings. The engineer suggested a sorting mechanism to filter out unnecessary drawings. Assembly drawings can be filtered out and maybe another filter can be added to filter for only drawings with railing assemblies.

Moving on to the output of the program, the generated drawing. Starting with the views the 3D-view is too big according to the Tekla user. It has the same size as the other views which is not the case for the template drawing. Some manual editing has to be done to resize and reposition the views. Since the *PlaceViews* method automatically positions the views on the sheet the views are a little bit off. The most important view should be placed in the upper left section of the drawing and the 3D-view in the lower right corner. Moving

views in a drawing are relatively fast, however.

The parts represented in the views look good and the cloning of these parts are correct when examined closely by the user. The user 3D-view looks better than those I generated with the Tekla cloning algorithm. There is one object not in the drawing which is closely connected to the part representations in the views namely the centerlines. It is a dotted line inside the parts to display for example the centre of an object. When cloning a part objects the centerline is not included since it is a separate object.

Regarding the dimensions, the EndView looks the most correct. Since no dimensions are deleted in the process of inserting them the drawings ends up cluttered. It would be desirable to make the dimensions more separated and delete some unnecessary dimensions according to the engineer. Dimensions for other objects than beams are also missing, but the cloning seems to be correct. Since the method for creating dimensions uses the EndPoint and StartPoint of the Beam object some dimensions are not positioned correctly. We would like to have some dimensions be placed at the intersection of two beams. The distance property is set as a static value for every dimension in every view, the user expressed the need to be able to change this value. This could be achieved with another option before the creation of the drawing.

The user examined the inserted marks and confirm they look correct, but too many are displayed. The cloning of the marks is correct, but there are too many objects. Marks also has to be treated differently based on what view the mark is being inserted to. There is also the problem of rotating the marks to follow the object they are describing.

The Tekla user does not think the current version of the application will not be an alternative to the Tekla cloning algorithm. The task of manual editing is not reduced when using the application at this stage of development. Returning to the functional requirements of the application. The system has created a drawing but requirements 2-4 are not realized. If requirement five is satisfied can be argued. Compared to creating a drawing from scratch this application does reduce the need to modify properties, but compared to the Tekla cloning algorithm it is hard to say.

Discussion

This project has aimed to explore the feasibility of a software solution as an alternative to the Tekla cloning process. Utilizing well-known software developing techniques like prototyping has proven useful when exploring different solutions. Defining stakeholder requirements has made the problem description more coherent and decision-making during development easier. Understanding constraints imposed by other systems was crucial to the development process since the application is naturally dependent on Tekla Structures. The tools used also has a big impact on the way the application was developed. Program execution could be faster using other tools, but the solution is dependent on time-consuming methods like *Insert()*. Several enumerators were used to extract data from various objects which comes with constrains concerning how data is accessed. The requirements to use C# did not have any substitutional implication on development since all three prototypes are relatively minor applications.

Heavy focus on the early stages of the development process led to a reduced focus on activities like testing and validation. For a more complete product, testing has to be performed on more assembly drawings with a greater variety of sizes and shapes. It stands to reason that validation could have played a more vital role in development. It could have been more interwoven with the prototypes and could therefore have had a more central part in decision making early on. It was a conscious choice to leave out validation until later in the development process. Because the aim is to explore different solutions it was a priority to create the right kind of solutions using requirements engineering, which skewed the focus away from processes like validation and testing. The driving force of development was understanding the constraints of the API and Tekla Structures which also lead to the heavy focus on the early development stages to define the system.

Since the application is by no means a complete product, it would be interesting to keep developing the application and see what problems discussed during validation can be fixed. The main issue with the GUI is the ViewType element. Sorting of elements in the ListView element is also simple since it is a common function to include in lists. These issues can all be fixed since it is not any new functionality being implemented, only tweaking of how the elements are displayed or what objects are being displayed. The GUI did serve its purpose and the users were in general satisfied with the familiar look of the application.

The evaluation of the application's performance showed that one of five requirements where fulfilled. However, some of the requirements are possible to fulfil using a similar application in the future. Some object classes were missing, mainly because they were not prioritized during development. It is unknown how easy or hard these classes are to implement, for example, the leaderline object. Since the Tekla cloning algorithm performs well in creating these object classes, there is no reason to think for example the leaderline class would be harder to implement than the dimension class. Possibly the most important class to get right is the dimension class since it is the most time-consuming to edit. The placement of the dimension class can never be perfect. The beam property start- and the endpoint was helpful when drawing these objects, but if every object placement is a bit off then this method does not save any time thus not fulfilling requirement number four. During the validation it was made clear one important view was missing namely BottomView, which is easy to add with similar views already implemented. Adding the right amount of objects seem possible with some modifications to the application, but creating a reliable solution for different types of assemblies may prove difficult. More user options have to be used to fulfil this requirement. There is no object which cannot be inserted in the drawing and therefore the requirement of adding the right type of objects can be achieved in the future. The application was successful in cloning the object properties of every object, therefore this requirement can be fulfilled, but since the drawings are missing objects it cannot be argued that the need for modifying object properties is fulfilled as of now.

5.1 Recommendations for future research

Several different techniques can be utilized when trying to reduce the amount of manual editing. Possible solutions are reliant on the Tekla Open API. The API structure, class- and property names are heavily influenced by Tekla Structures logical composition and names. It is advisable for any developer tackling these kinds of problems in the future to have a background using Tekla Structures. This would not be a prerequisite if not for the limited documentation and lack of useful example code on the subject. It is suggested to employ a different approach to creating the dimension class. One way could be to aid the structural drafter or engineer to manually draw the dimensions with a drawing tool instead of replacing the Tekla cloning algorithm.

To conclude, a new prototype can help reduce time spent on manual editing and a new application can be useful to engineers working with Tekla Structures. It is recommended this application has a narrow focus, solving specific problems, no application can outperform the Tekla cloning algorithm in every aspect. One way of doing this is creating a program with a GUI and providing the user with more settings compared to the Tekla cloning algorithm. Keep in mind the constraints of the system, Tekla has to be opened and Tekla Open API has to be used. This means it is also suggested to use C# and Windows although there are faster programming languages.

Conclusion

The focus of this thesis has been on creating a product, but is it the right type of product? Creating a useful application using Tekla Open API is not low hanging fruit. The same can be said about the purpose of the application in itself, reliably cloning drawings is no easy task in itself. A few major constraints have to be taken into account when developing a new application as explained in the previous chapter. To be clear, time spent on editing Tekla drawings by employees at Sweco can be reduced using Tekla Open API in some way. This process can be improved, the next question is naturally to what extent. Can a cloned drawing ever be 100% correct without the need for manual editing? For more complex assemblies the answer is no. No algorithm improves the cloning process for every drawing. The most useful approach would be developing an ad hoc solution that can reduce the time spent editing. Shifting focus and trying to eliminate the most repetitive tasks of manual editing could be a good fit for these types of applications. The million-dollar question is how long does it take to develop a tool and how much time does this tool save? Hopefully, this thesis can prove useful when trying to answer these types of questions. To conclude and at the risk of being somewhat subjective it is advisable to develop new tools simply because of the tedious nature of editing Tekla drawings.

References

Printed

[2] David Benyon. *Designing Interactive Systems*. 3rd ed. United Kingdom: Pearson Education Limited, 2014.

[6] Chuck Eastman et al. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers, and Contractors*. New Jersey: John Wiley & Sons, 2008.

[13] Roger S. Pressman. *Software engineering: a practitioner's approach*. 5th ed. New York: McGraw-Hill, 2001.

[14] Ian Sommerville. *Software Engineering*. 9th ed. United States of America: Addison-Wesley, 2011.

[18] Karl Wiegers and Joy Beatty. *Software Requirements*. 3rd ed. Washington: Microsoft Press, 2013.

Electronic

[1] .NET. *.NET documentation*. Microsoft. 2021. URL: https://docs.microsoft.com/en-us/dotnet/.

[3] Bitesize. *Programming software and the IDE*. BBC. 2021. URL: https://www.bbc.co.uk/bitesize/guides/zgmpr82/revision/4.

[4] Coordinates. *Tekla User Assistance. Coordinate System*. Trimble Solutions Corporation. 2021. URL: https://support.tekla.com/doc/tekla-structures/2020/gen_coordinate_systems.

[5] Drawings. *Tekla Open API Reference. Drawings*. Trimble Solutions Corporation. 2021. URL: https://developer.tekla.com/tekla-structures/api/14/10012.

[7] Ecma. *ECMA-334. C# Language Specification*. Ecma International. 2017. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-334_5th_edition_december_2017.pdf.

[10] Model. *Tekla Open API Reference. Model*. Trimble Solutions Corporation. 2021. URL: https://developer.tekla.com/tekla-structures/api/14/13460.

[11] NuGet. *NuGet documentation*. Microsoft. 2021. URL: https://docs.microsoft.com/en-us/nuget/.

[12] Orientation. *Tekla User Assistance. Part Orientation*. Trimble Solutions Corporation. 2021. URL: https://support.tekla.com/article/understanding-part-orientation.

[15] Structures. *Tekla Open API documentation*. Trimble Solutions Corporation. 2021. URL: https://developer.tekla.com/tekla-structures/tekla-open-api-documentation.

[16] Studio. *Visual Studio IDE documentation. Welcome to the Visual Studio*. Microsoft. 2021. URL: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019.

[17] Trimble. *What is BIM?* Trimble Solutions Corporation. 2021. URL: https://www.tekla.com/resources/blogs/what-is-bim.

Interviews

[8] Simon Jonsson. "Manager at Sweco" (online meeting). 2021.

[9] Filip Kristiansson. "User at Sweco" (online meeting). 2021.

Appendix

A.1 Evaluation questions

A.1.1 GUI design

- Is it clear what the elements in the window are for? (1, 3)
- Is it clear what the symbols are for? (4)
- Is the design consistent? (2)
- Do you understand what actions you are supposed to perform and do you understand what the program does? (6)
- Is it clear what relationship elements have with their function? (7)
- How does the application handle wrongful actions from your part? Is it convenient to make a mistake? (8, 9)
- Is the application appealing to you? Do you like how it looks? (11)

A.1.2 Functionality

- Do you think the applications are useful? Has it the potential to aid you in your work in Tekla?
- Do you like the selection of options made available?
- Is the creation of a new drawing correctly performed?
- Is the creation of views correctly performed?
- Is the cloning correctly performed?
- Is the creation of dimensions made correctly?
- Is the creation of marks made correctly?
- What functions are you missing?
- What functions are most important to prioritize to make the application more useful?