TURUN YLIOPISTO

CUSTOMTOOLS as general integration platform for SOLIDWORKS – ERP data exchange

Software Engineering
Information and Communications Technology
Department of Computing

Master's Thesis in Technology

Author:
Simo Erkinheimo

Supervisors:
Professor Ville Leppänen (University of Turku)
Assistant Professor Tuomas Mäkilä (University of Turku)

June 2021

Master's Thesis in Technology
Department of Computing, Faculty of Technology
University of Turku

Major subject: Software Engineering
Degree Programme: Information and Communications Technology
Author: Simo Erkinheimo
Title: CUSTOMTOOLS as general integration platform for SOLIDWORKS - ERP data exchange
Pages: 95 pages, 0 appendix
Date: June 2021

This thesis introduces the extreme complexity behind CAD – ERP integrations in real world scenarios as a reasoning why a general-purpose integration for all system variations simply does not exist. Delivered integrations are mostly fully customized projects, solving the same issues repeatedly with varying success and promise of usability.

To attempt at least some level of generalization, the CAD system is chosen to be SOLIDWORKS, and CUSTOMTOOLS for SOLIDWORKS is chosen to provide partial solution by itself as well as an integration platform for any target ERP system.

Subject matter expertise is first heavily relied on determining the most common requirements of SOLIDWORKS – ERP integrations and design science methodology is used to generalize a solution. CUSTOMTOOLS Profile is configured to meet some of the requirements and CUSTOMTOOLS API is thoroughly examined for meeting the rest, eventually resulting in generalized solution in the form of data model configuration rules and integration base implementation, embedded in the CUSTOMTOOLS core product. The true complexity of the common requirements can be easily seen in the generalized solution architecture, and benefits of the provided solution are recognized by consulted SOLIDWORKS - EPR integration project experts. The provided solution will be taken in use for future integration projects.

Keywords: ERP, CAD, integrations, SOLIDWORKS, CUSTOMTOOLS

Table of contents

1 Introduction

1.1 Role of an ERP in manufacturing industry

Generally, in all industries, it is required to be efficient to reduce operational and manufacturing costs. Many Enterprise Resource Planning (ERP) solutions aim to help with this goal by collecting business data under the same system, thus making it possible for different departments or sites to interact seamlessly for greater value. A simple example would be having warehousing, manufacturing and sales all under the same system. For salespeople to sell, there must be items to sell in the warehouse or at least in the manufacturing pipeline for later delivery. Having it all available in the same system should bring obvious advantages over separate systems. (Singh & Khamba, 2017, pp. 42-44) (Hwang & Min, 2013)

In the manufacturing industry a single design company may easily create thousands of CAD models in a month. So large amounts of design data and documents are produced that, for design time efficiency, must all be accessible and available for reuse by other designers of the company. (Eustache, et al., 2002) This problem is handled with Product Data Management (PDM) solutions that are usually tightly integrated with single or small selection of CAD software to provide real efficiency for continuous design processes. However, while essential and usually the first system to invest in the CAD design business, a PDM alone is generally not meeting the requirements that arise from a resource planning point of view. (Hou, et al., 2008)

As much as there is product data for a PDM system to handle, there can be the same amount or even more corresponding items- and bills of materials data that are required to be managed in the ERP system. While a single CAD document in PDM may describe a piston of an engine and how to manufacture it, in the ERP the piston is just one of the items that are required to fulfill a purchase order of an engine. Items that are manufactured usually require many types of resources and time, and as such it might sometimes make better sense to order that piston from elsewhere than to manufacture it from the resource planning point of view. This is exactly what an ERP system is designed for and thus also justifies its place in the manufacturing industry. (Muni Prasad, et al., 2013)

1.2 Problem statement

As the design and manufacturing industry dwells on large amounts of relational design data (Figure 1.2.1), it is basically unreasonable to expect that anyone would manually handle the data exchange from the design environment to the ERP. Manual work has sometimes seen to be done on a very abstract level of design itemization but resource planning efficiency with that can only go so far, not even to mention the human resource cost for the manual work itself.

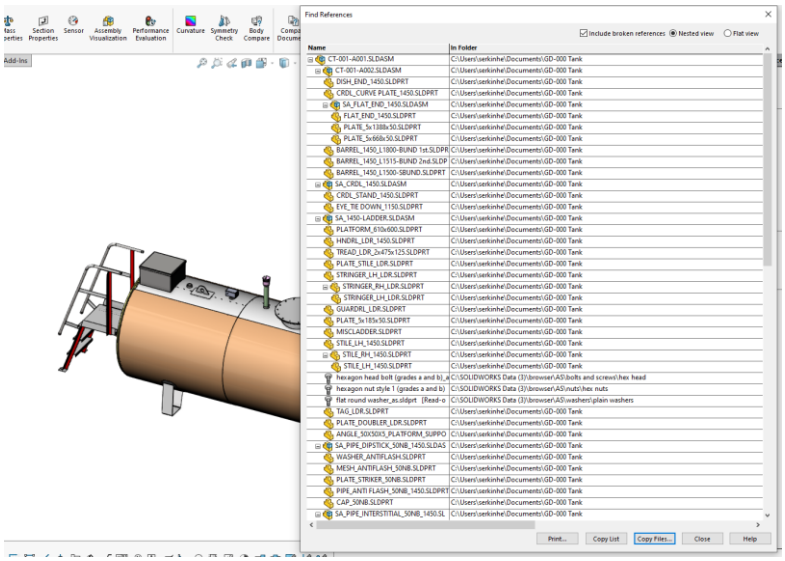


Figure 1.2.1: Even a simple tank can have hundreds of components.

Thus, to get real benefits out of an ERP system in the design and manufacturing industry, the data exchange between the design and ERP environments must be automated. Because many different and even fully customizable ERP systems as well as PDM systems exist that can sometimes support single or a selection of CAD software, in practice the reality of an integration is always a CAD specific customized solution at least to some extent. Some ERP specific integration products of course do exist, but if they require a very specific design data model to work, they just will not with existing designs without migration; unless they are configurable by the full data complexity that the CAD system allows. *Impossibility to migrate data from a legacy system to a new system* (Muni Prasad, et al., 2013, p. 47) is one of the key issues what it comes to integration projects. (Fawzy Soliman, 2001) (Hwang & Grant, 2011)

Also, as batch exporting design data to ERP is the top requirement of an integration as will be discussed in Section 2.6, this type of one-way integration alone is not enough even for a task as simple as creating a new design for an existing ERP item. Already from these requirements it follows that an integration must be bi-directional and configurable integral part of the CAD designer's familiar

system, without forgetting the importance of design cost efficiency to successfully serve its purpose. As the existing design data must be considered as well, configurability of the itemization has a large role in successful integration (Figure 1.2.2).

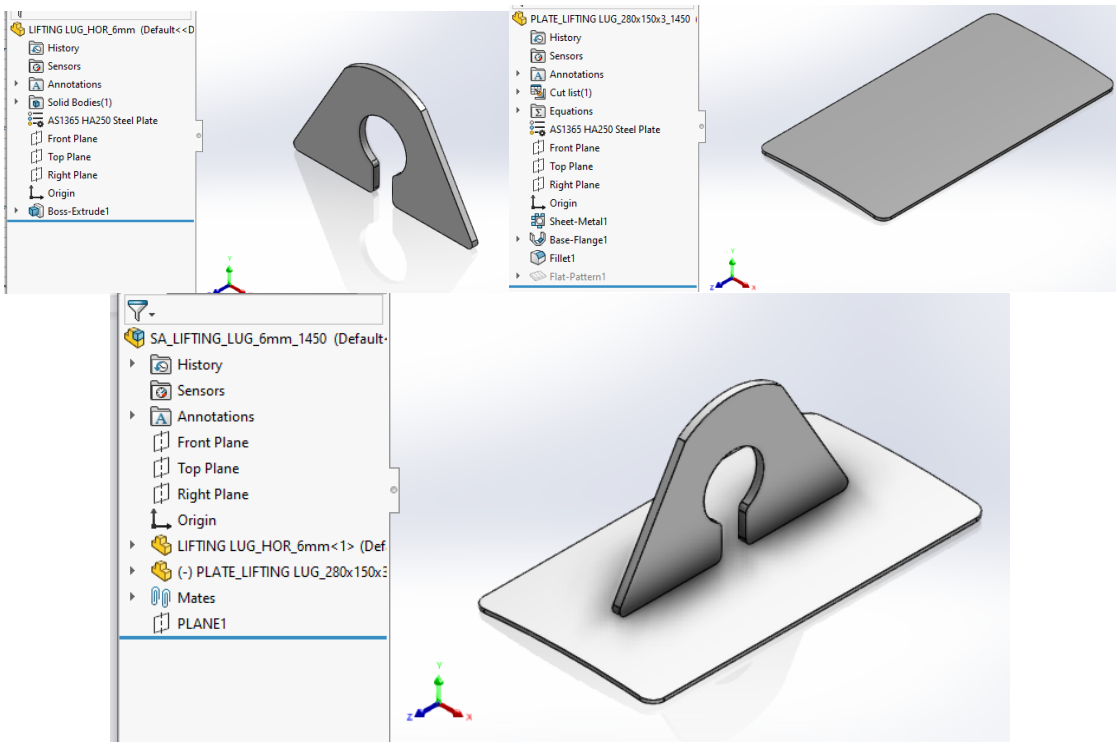


Figure 1.2.2: It is non-trivial to conclude directly from a CAD design which assemblies or parts should be considered as ERP items.

The generalized problem is the overall required complexity of an CAD – ERP integration when considering the most common requirements that such integrations must meet. This extreme complexity is likely the reason why such generic solution simply does not exist.

To narrow down the problem, this thesis focuses on a specific CAD system, SOLIDWORKS, that has a well-documented API sufficient for required data exchange. Its model specific arbitrary design data is a list of key-value -like Custom Properties (Figure 1.2.3) for the document, and also another set of them per each configuration of the document. While this data model is quite simple by itself, the data access in design time is done by other tools that create their own view and access to the design data (Figure 1.2.4). It is this view that has determined not only the data keys and values for all of the existing designs but also the expectancy of data inheritance and distribution between configurations when an entry is modified from that view. Data consistency over the whole company is usually ensured with some sort of shared profiling or a PDM system. (Hou, et al., 2008)

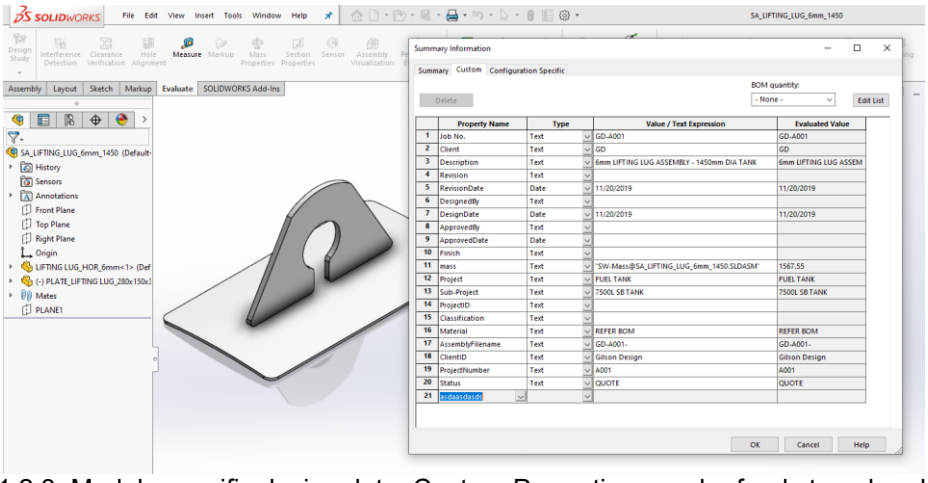


Figure 1.2.3: Models specific design data, Custom Properties, can be freely typed and edited.

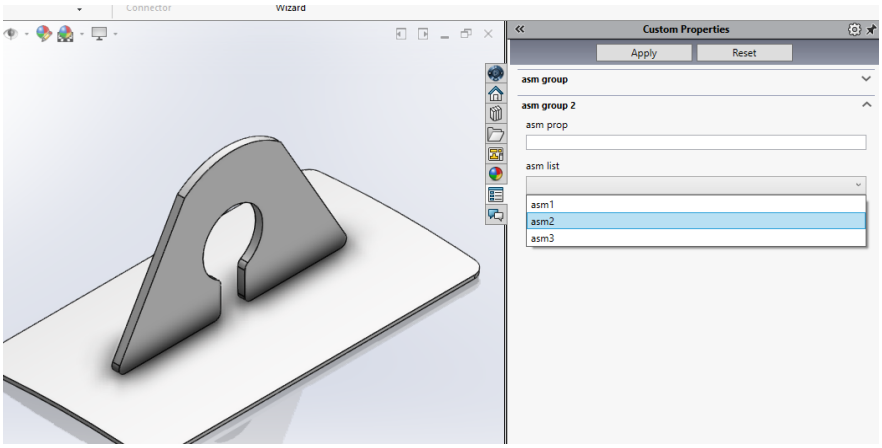


Figure 1.2.4: SOLIDWORKS Tab Builder (right) is one way of accessing Custom Properties in a preprofiled manner to guide designers for consistent data filling.

SOLIDWORKS is also capable of providing a dynamic Bill of Materials (BOM) (Figure 1.2.5) for models that, when correctly configured, one could expect to work as a perfect data source for item and BOM export to ERP. (Zhu & Yan, 2018, p. 59) This option will be thoroughly examined and shown in Section 2.5 why it is not a feasible data source in real world applications.

ITEM NO.	PART NUMBER	DESCRIPTION	QTY.
1	SA_PIPE_DIPSTICK_50NB_1450	BSP THREADED PIPE C/W ANTIFLASHING GAUZE & CAP - CALIBRATED AL RHS	1
1.1	PIPE_ANTI FLASH_50NB_1450		1
1.2	WASHER_ANTIFLASH		1
1.3	MESH_ANTIFLASH_50NB		1
1.4	PLATE_STRIKER_50NB		1
1.5	CAP_50NB	CAP 50NB	1
2	SA_PIPE_INTERSTITIAL_50NB_1450	BSP SOCKET C/W DROPPER & PLUG	1
2.1	PIPE_INTERSTITIAL_50NB_1450		1
2.2	SOCKET_50NB	SOCKET 50NB	1
2.3	PLUG_50NB	SQUARE PLUG 50NB	1
3	SA_PIPE_VENT_SBUND_50NB_1450	BSP SOCKET C/W RISER AND WEATHERPROOF CAP	1
3.1	PIPE_VENT_50NB_SBUND_1450		1
3.2	SOCKET_50NB	SOCKET 50NB	1
3.3	WEATHERPROOF CAP_50NB	WEATHERPROOF CAP 50NB	1
4	SA_PIPE_BUND_VENT_50NB	ASSEMBLY 50NB PIPE ELBOW FOR BUNDVENT	1
4.1	PIPE_ELBOW_BUNDVENT_50NB	50NB PIPE ELBOW FOR BUNDVENT	1
4.2	SOCKET_50NB	SOCKET 50NB	1
4.3	WEATHERPROOF CAP_50NB	WEATHERPROOF CAP 50NB	1
5	500MM MANWAY	ASSEMBLY - 500MM MANWAY	1

• • •

Figure 1.2.5: Part of SOLIDWORKS generated bill of materials

To wrap up the targeted problem of this thesis, it is an extremely complex task to achieve a SOLIDWORKS - ERP integration of true value and therefore such integrations tend to be either expensive or poor quality, or simply ignore some of the common requirements and fail to deliver the advertised value for the integrated environment.

1.3 Motivation of this thesis

The author of this thesis is currently employed in a software development team whose main product is CUSTOMTOOLS for SOLIDWORKS, a SOLIDWORKS add-in that among other things has extremely configurable Custom Property access. It can respect most cases of logical data inheritance models that have been used in legacy designs to provide as familiar a user experience and data model as possible. Its view and data access for the Custom Properties is simply called Properties (Figure 1.3.1) and it can support a variety of per-property data sources as well as value automations and configuration targeting, and they are extendable for ERP item mapping too as will be shown.

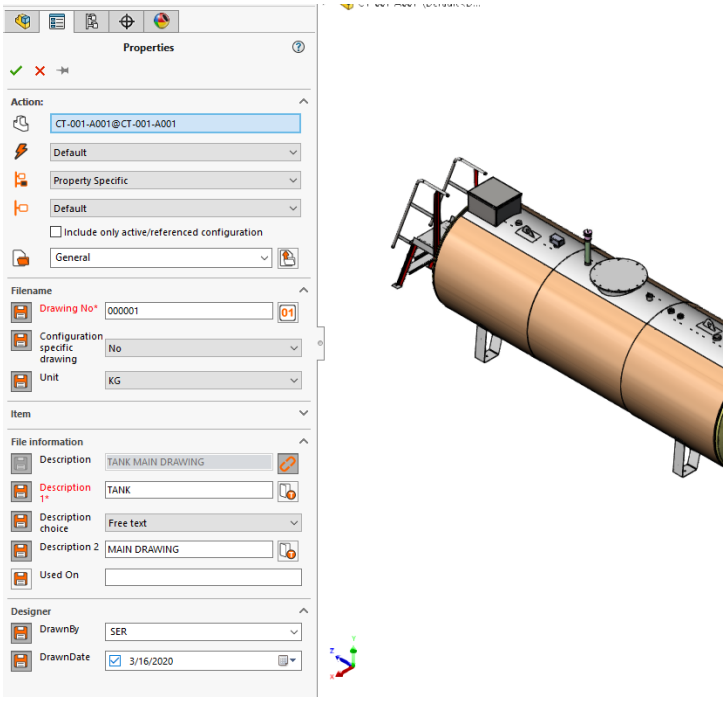


Figure 1.3.1. CUSTOMTOOLS Properties

CUSTOMTOOLS also provides its own BOM -like view which can be configured to collect any data from the models into the same view, Export (Figure 1.3.2). This is originally designed especially for customized report generations and simple integrations, but the API has since extended way beyond the original intentions due to different kinds of requests and requirements. The author recently had an opportunity to solve more specific extensibility requirements which led to the birth of a new extensibility framework, CUSTOMTOOLS Extensions API. It complements the original event based CTInterface API with class abstractions that can be extended and are treated as they were specific core features (CUSTOMTOOLS API Help, 2021). It is already known that the common ERP integration requirements can be met with what is now available. However, due to overall complexity, the same problems have been solved and resolved already countless times by different developers with varying success. As the common requirements are always similar and the platform underneath is the same,

there is a clear motivation to provide a generalized solution, which will hopefully lead to more robust implementations and faster integration development cycle as well as growing user base.

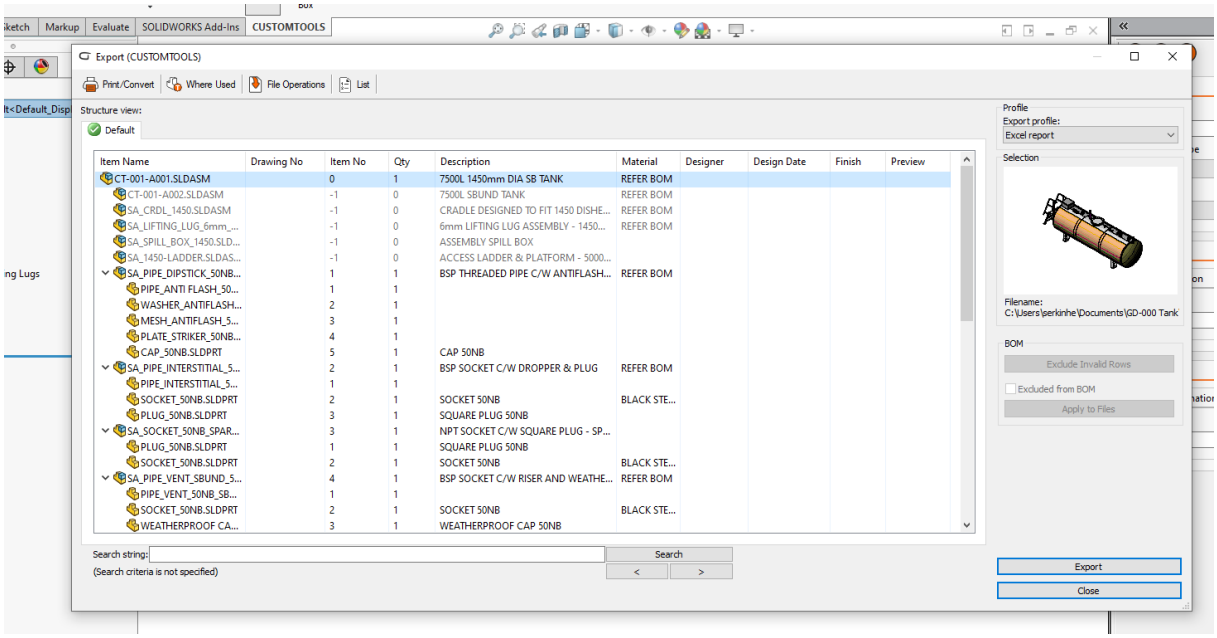


Figure 1.3.2. CUSTOMTOOLS Export

1.4 Content

Chapter 2 describes very basic usage of SOLIDWORKS covering its terminology, design methodology and how a multicomponent design with its property data results in a bill of material (BOM). Bills of materials are the starting point of structural itemization of SOLIDWORKS models, and it is important to understand how they are built and what is the designer's role in their formation to understand why the BOM is not sufficient data source as it is for ERP integrations. Finally, the most common requirements of SOLIDWORKS - ERP integrations are introduced, sourced by subject matter expert consultation, and for which this thesis will aim to provide a generalized solution in Chapter 4.

Chapter 3 introduces CUSTOMTOOLS for SOLIDWORKS, a SOLIDWORKS add-in that can handle profiling of properties and their data content as an extremely configurable integrated solution. Also, it provides its own SOLIDWORKS BOM like structural data representation, CUSTOMTOOLS Export, that together with the open CT API, can tackle the common real-world issues that the SOLIDWORKS BOM has as a data source (Section 2.5.4). Other ERP integration related CUSTOMTOOLS features are briefly overviewed, like the Lookup Lists and Search Groups as well as CUSTOMTOOLS' own script deployment system that bring the same extended features for all users in the same environment. CUSTOMTOOLS will be used as an integration platform for the generalized SOLIDWORKS - ERP

integration. This chapter lacks citations since the author of this thesis is CUSTOMTOOLS Product Manager (2021) and therefore subject matter expert of its content and claims.

Chapter 4 uses design science methodology in generalized solution architecturalization. It goes more in depth with CUSTOMTOOLS' profiled behavior and its API by utilizing subject matter expert knowledge to introduce rules for data model configuration and generalized base implementation for all integrations targeting the same common requirements. C# generics is heavily utilized to provide strong typing and compile time type safety, having the aim at implementation ease-of-use and solution quality when utilizing the provided base.

The given solution is put to test at Chapter 5 with an example implementation and then evaluating the difference in required complexity when implementing the same requirements from scratch versus using the provided base implementation. In-house feedback from subject matter experts that have many years of experience implementing said common requirements for integrations is also presented, highlighting the significance of this work for them.

Chapter 6 discusses this thesis not only in the presented scope but also in more generalized manner whether the conclusions could be extended beyond its context. Also, future improvements are suggested to generalize the provided architecture even further to extend the archived benefits.

2 SOLIDWORKS

SOLIDWORKS is a computer-aided design (CAD) software used mostly in mechanical designing. Its first release was in 1995 and in 2016 it had grown to 2.3 million active users globally. (Schmitz, 2016) It is a solid modeler that utilizes parametric feature-based approach for creating parts and assemblies that can be mated together to form more complex assemblies.

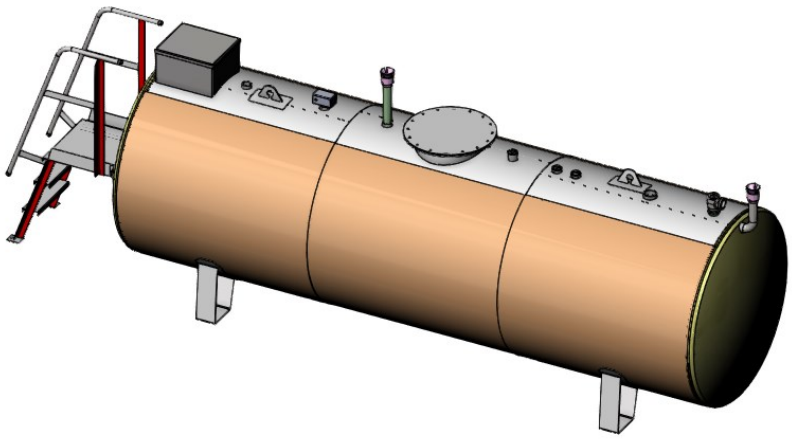


Figure 2.1: A tank design created with SOLIDWORKS.

2.1 Design methodology

In SOLIDWORKS design methodology, important concepts to fully understand are *parts, assemblies, configurations, components* and *drawings*. Parts and assemblies are 3D designs and referred to as *models*. *Drawings* are dynamic 2D projections of the models they represent. When a part or an assembly is used within another assembly, they are referred to as *components* of the assembly. (SOLIDWORKS, 2015, pp. 11-13) Due to the dynamic aspect of the drawings, a valid drawing cannot exist if the model it refers to is lost. The same non-validity applies for assemblies that depend on other assemblies or parts too. Thus, for instance a drawing is critically dependent on the existence of every single model that is included in the design.

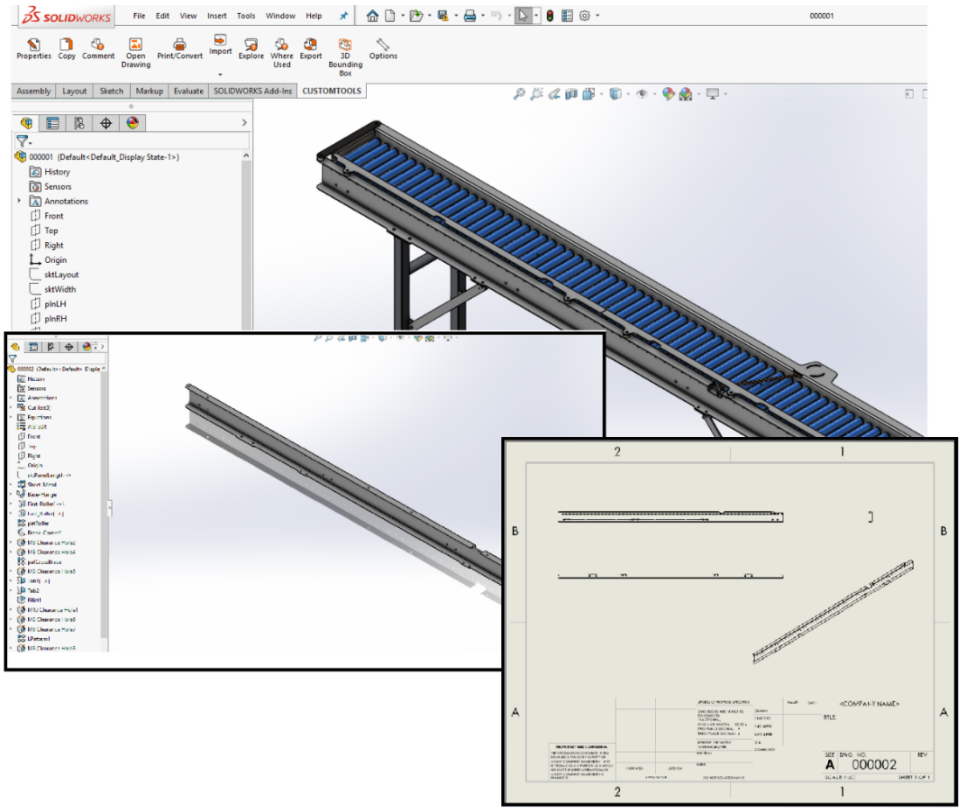


Figure 2.1.1: SOLIDWORKS Assembly, Part and Drawing

To be more accurate, a model or a drawing never just refers to a complete model, but always to a configuration of a model. A configuration defines the enabled features and their parameters in a model and there is always at least one configuration in each model. (Iancu, 2016) This means that it is possible to create a single design and refer to for instance different configured sizes of it within assemblies and drawings (Figure 2.1.2).

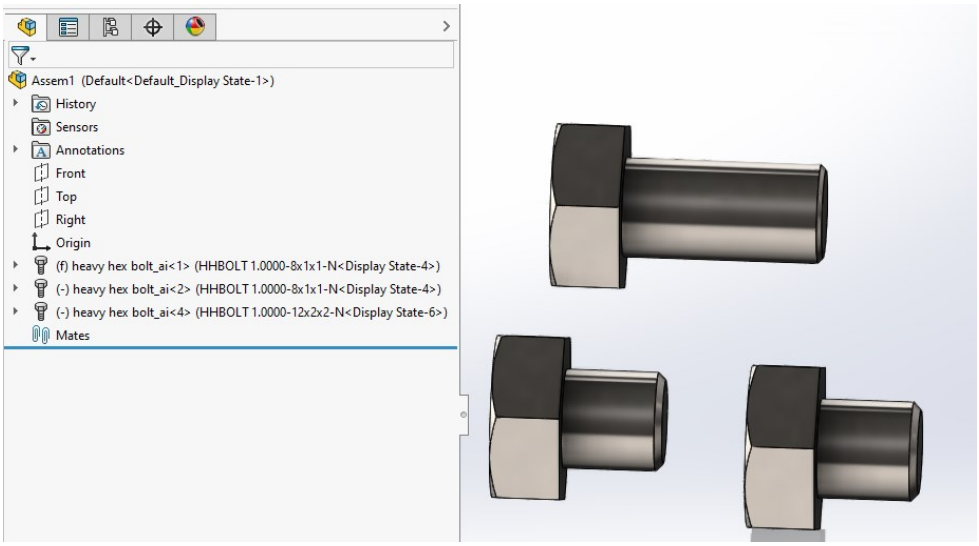


Figure 2.1.2: Assembly with 2 different configurations of the same heavy hex bolt_ai -part model

Target of the design process is usually to create something that can be manufactured. For manufacturing processes that involve humans, a 2D -drawing is usually required but lately also Model Base Definition (MBD) has been a growing trend. In MBD a capable device is used to display the whole 3D design so the required measurements can be taken directly from it. (Mäkinen, 2018)

For machine-based manufacturing like CNC machining, parts of the design are usually converted into more suitable formats like DXF or DWG, or for instance into STEP-files for 3D printing.

Drawings are not bringing much real value to the scope of this thesis, so they are only briefly mentioned.

2.2 Simplified design process

To create a design, one would start by creating a single solid object, a part, using parametric sketching and then applying features on it to form a volumetric object (Figure 2.2.1). (Jankowski & Doyle, 2011)

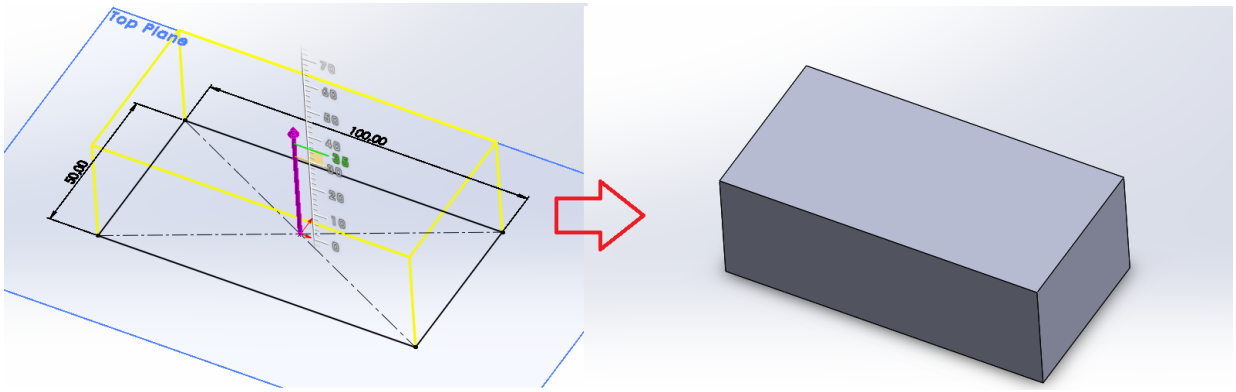
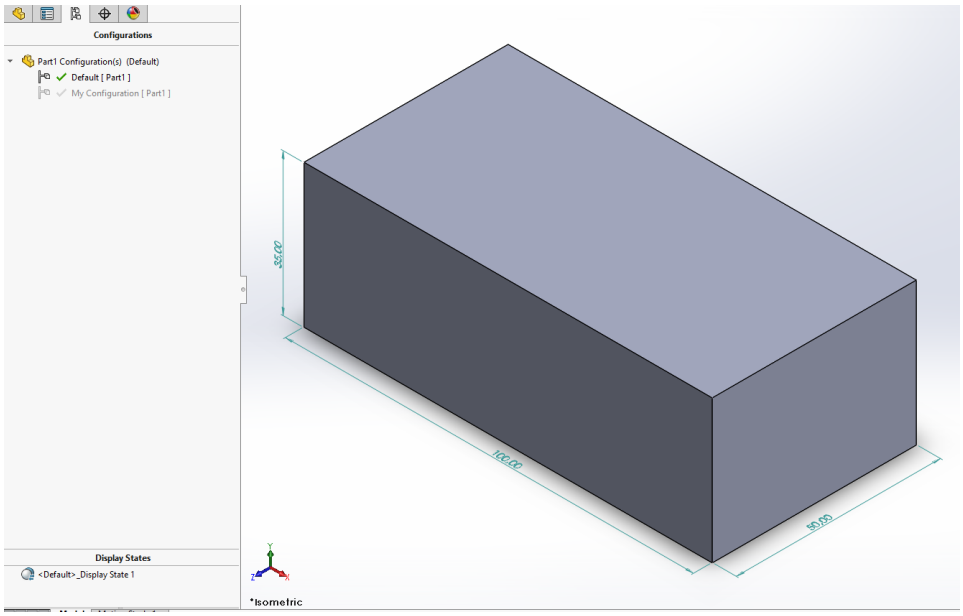


Figure 2.2.1: From parameters to volumetric object using sketch extrude

Already when a new empty part was created, it had a default configuration. It is possible to add a new configuration and then for instance configure dimensions (Figures 2.2.2 and 2.2.3). (Iancu, 2016)



Picture 2.2.2: Part1 having ‘Default’ -configuration active

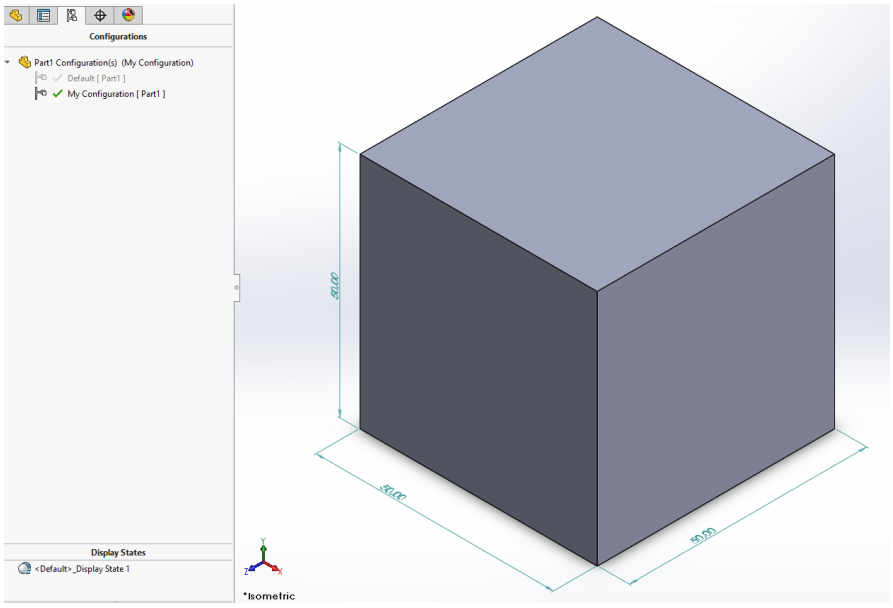


Figure 2.2.3: Part1 having ''My Configuration’ -configuration active

To join multiple parts together, an *assembly* must be created. It may contain any amount of configuration instances of any parts and even other assemblies, except that the model must not result in circular references. Also, all model files must have a unique name within the assembly. (SOLIDWORKS, 2015) An example of a circular reference would be a car -assembly that contains a car-body -assembly that again contains the car -assembly that contains the car-body -assembly ...it quickly becomes obvious why circular references are not possible even in real life.

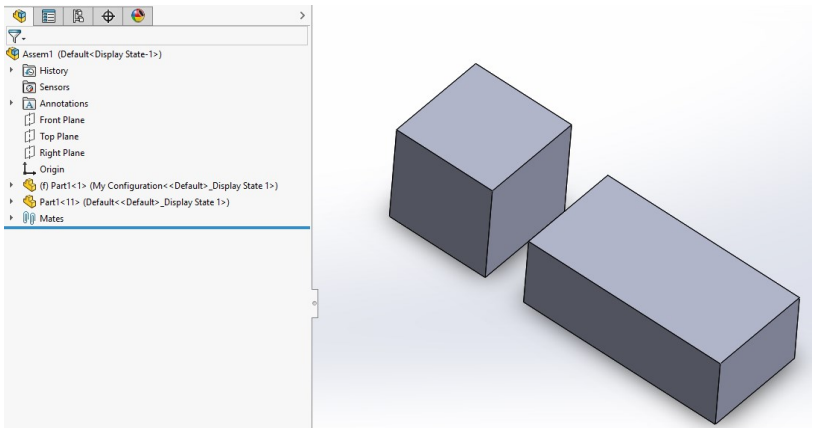


Figure 2.2.4: Assembly Assem1 with two instances of Part1; one in My Configuration and other in Default.

2.3 Model files and file reference handling

In the most basic case, every part and assembly would be their own file in the file system. However, it is possible to design models virtually inside assemblies which, from the file system point of view, would result in a single assembly file. This is particularly useful when the design is small, and its components do not have to be reusable. All references of a file, that are not virtual, are referred to as external references.

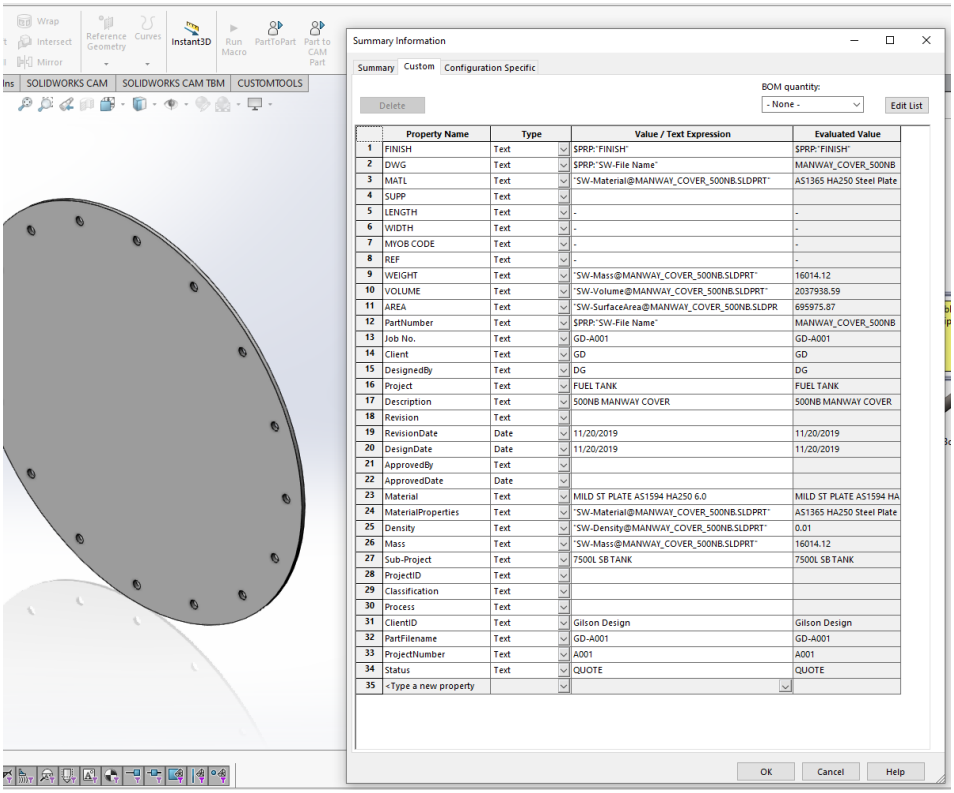
Every time an assembly is saved in SOLIDWORKS, all the paths to its first level external references are stored into it. Then when an assembly is opened, SOLIDWORKS tries to find those external references from the file system and include them to the model opening process. If the newly added components have external references, the same operation is done for them too and so on the whole transitive closure of components is traversed. (DASI Solutions, 2014)

As a single model may easily consist of hundreds of components, it becomes obvious that keeping track of and even creating design related files on the file system is something that cannot just be left as a wild west. File naming automation and design search are among the most basic needs even of a small sized design company. SOLIDWORKS PDM provides a great solution for this but can be out of the price range for small companies. Other SOLIDWORKS integrated systems trying to answer for the same needs are for instance CUSTOMTOOLS and SolidPDM.

2.4 Model specific arbitrary design data

To store arbitrary design information, SOLIDWORKS has Custom Properties that are key - value like attribute information for models and drawings. The Custom Property data is stored in the file in question.

For a single model, there are always at least two separate sets of Custom Properties, one for the document and one for every configuration of it. This is because a model always has at least one configuration. When a model is used as a component, it is referenced in the parent assembly with its configuration, so a configuration specific value for some Custom Property key is more significant than a possible document level Custom Property value with the same key. However, if there is no value found from referenced configuration, the search can fall back to the document level. This makes it possible to define document level properties (e.g., Designer, Design date, Customer, Project, …) that are common for all configurations as well as configuration specific properties that are only valid when referenced on that configuration (Length, Material, Weight, ...). This data can then be automatically mapped to various tables like Bills of Materials or to annotation at drawings. (CAD2M, 2018)



	Property Name	Type	Value / Text Expression	Evaluated Value
1	FINISH	Text	$PRP:"FINISH"	$PRP:"FINISH"
2	DWG	Text	$PRP:"SW-File Name"	MANWAY_COVER_500NB
3	MATL	Text	"SW-Material@MANWAY_COVER_500NB.SLDPRT"	AS1365 HA250 Steel Plate
4	SUPP	Text		
5	LENGTH	Text	-	-
6	WIDTH	Text	-	-
7	MYOB CODE	Text	-	-
8	REF	Text	-	-
9	WEIGHT	Text	"SW-Mass@MANWAY_COVER_500NB.SLDPRT"	16014.12
10	VOLUME	Text	"SW-Volume@MANWAY_COVER_500NB.SLDPRT"	2037938.59
11	AREA	Text	"SW-SurfaceArea@MANWAY_COVER_500NB.SLDPR	695975.87
12	PartNumber	Text	$PRP:"SW-File Name"	MANWAY_COVER_500NB
13	Job No.	Text	GD-A001	GD-A001
14	Client	Text	GD	GD
15	DesignedBy	Text	DG	DG
16	Project	Text	FUEL TANK	FUEL TANK
17	Description	Text	500NB MANWAY COVER	500NB MANWAY COVER
18	Revision	Text		
19	RevisionDate	Date	11/20/2019	11/20/2019
20	DesignDate	Date	11/20/2019	11/20/2019
21	ApprovedBy	Text		
22	ApprovedDate	Date		
23	Material	Text	MILD ST PLATE AS1594 HA250 6.0	MILD ST PLATE AS1594 HA
24	MaterialProperties	Text	"SW-Material@MANWAY_COVER_500NB.SLDPRT"	AS1365 HA250 Steel Plate
25	Density	Text	"SW-Density@MANWAY_COVER_500NB.SLDPRT"	0.01
26	Mass	Text	"SW-Mass@MANWAY_COVER_500NB.SLDPRT"	16014.12
27	Sub-Project	Text	7500L SB TANK	7500L SB TANK
28	ProjectID	Text		
29	Classification	Text		
30	Process	Text		
31	ClientID	Text	Gilson Design	Gilson Design
32	PartFilename	Text	GD-A001	GD-A001
33	ProjectNumber	Text	A001	A001
34	Status	Text	QUOTE	QUOTE
35	<Type a new property			

Figure 2.4.1: Document level Custom Properties of a model

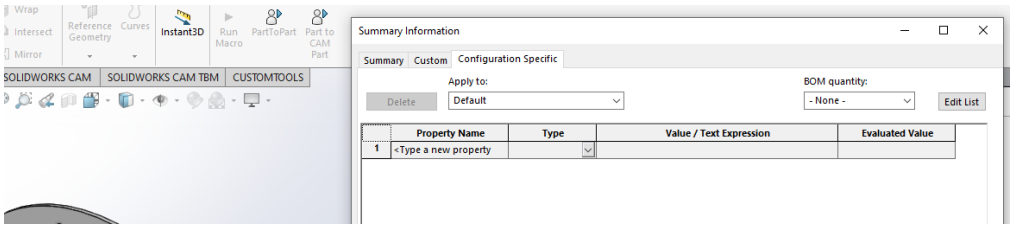


Figure 2.4.2: Configuration specific Custom Properties of a model. In this case the designer has decided to put all data on document level properties, so they automatically apply for all configurations.

Without any tools, SOLIDWORKS users would have to type in manually each required Custom Property key and value for the design documents, and this is very prone to human errors. There are a variety of SOLIDWORKS integrated tools that aid in consistent filing of the data and they are usually all based on some sort of profiling. For the very least the profile will pre-define the keys that should or can be filled but many solutions have taken this much further with limited selections, automated combinations or serials and even pulling predefined item data from 3rd party systems. Again, for instance SOLIDWORKS PDM, CUSTOMTOOLS and SolidPDM are designed to answer these needs as well as relatively very simple SOLIDWORKS Tab Builder. (Lombard, 2013, pp. 769, 387)

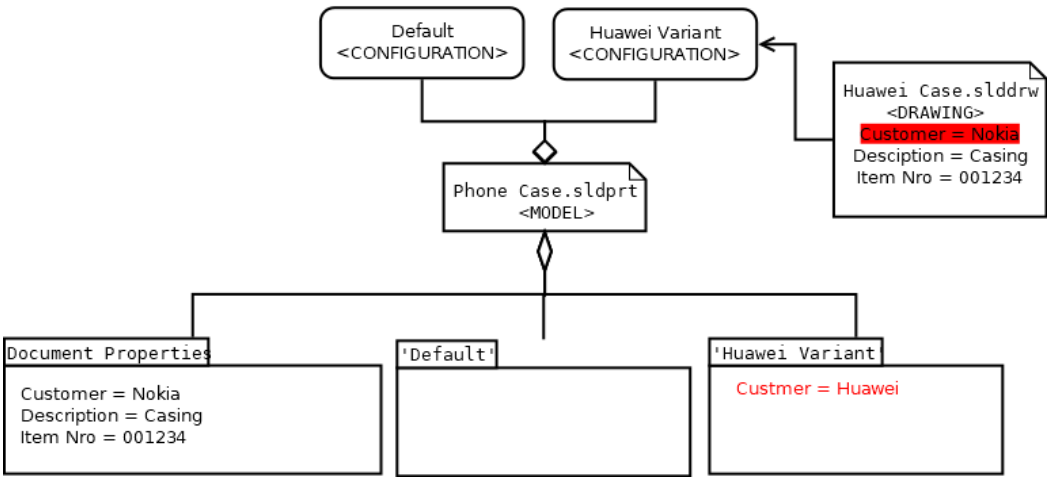


Figure 2.4.3: In this fictional case a phone case is initially designed for customer “Nokia”. The same design was later re-configured for customer “Huawei" but the designer typed *Custmer* instead of *Customer* to configuration specific Custom Properties. This caused Huawei to receive manufacturing documents with competitor’s name in them. An embarrassing mistake that would have been avoided using Custom Property profiling tools.

2.5 Bill of Materials

A dynamic bill of materials -table can be created for a design in SOLIDWORKS and at least selected BOM type, configuration grouping and some design time modifiers have direct effect on the outcome. The BOM generally tells *what* and *how much*, but for different purposes a different type of BOMs is usually used. For example, having only parts and their quantities would be a very usable BOM as a packing list while a structural BOM would usually be preferred at assembly lines.

It is important to understand how SOLIDWORKS BOMs are formed as they are the designers' tool for overviewing item-BOM relations on design time. It would be reasonable from the designer point of view to expect possible ERP -integrations to follow the same BOM forming principles in reflecting the design to the target system as what SOLIDWORKS displays.

A SOLIDWORKS BOM does not only collect the components, quantities and possible structures, but it's also possible to pull any model specific arbitrary design data (Section 2.4.) for each component to the table (CAD2M, 2018). From this it follows that the SOLIDWORKS BOM table could potentially be used directly as a data source for ERP integrations. However, this is not the case as will be shown in Section 2.5.4.

2.5.1 BOM Types

When inserting a BOM -table to drawing or model, the first selection to make is to select its type: *Top-level only, Parts only* or *Indented*. To better understand their difference, first consider the reference design structure in Figure 2.5.1.1 which introduces an assembly that references three different subassemblies and a part. The subassemblies then have their own component references, and the overall design also reuses some models in multiple different parent assembly contexts.

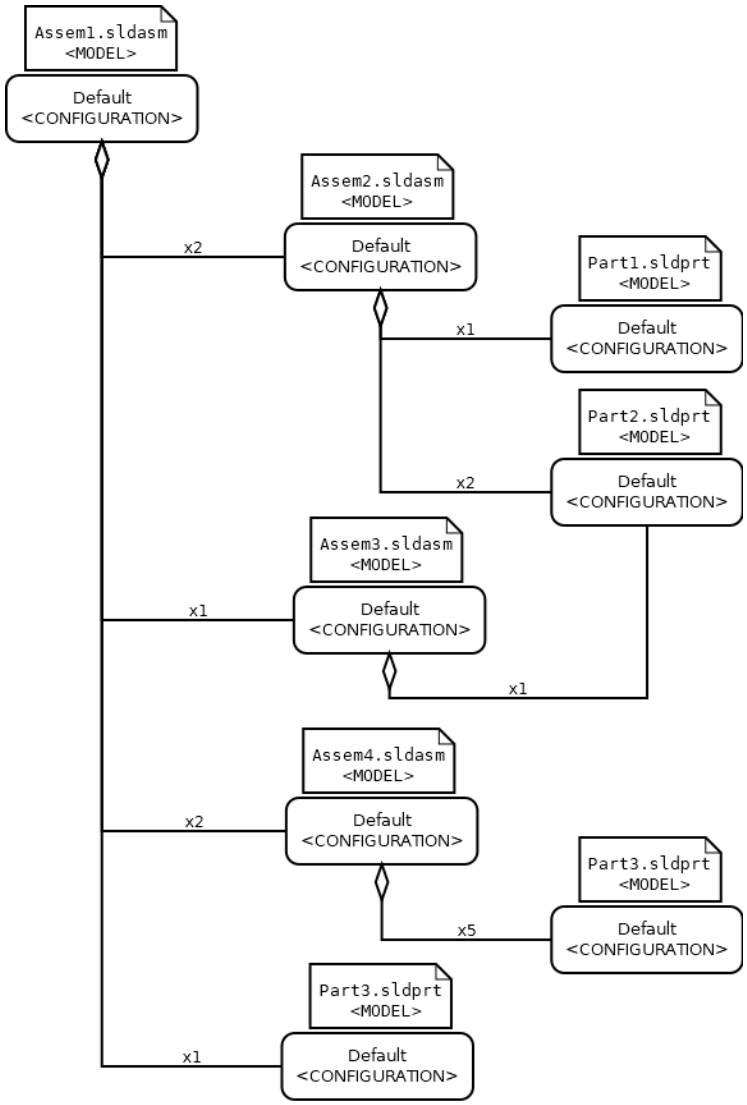


Figure 2.5.1.1: The reference design has components in multiple levels, and some are reused in different contexts.

An indented BOM -type would include the structural aspect of the design and report reference quantities as “quantity in parent” (Figure 2.5.1.2). This BOM -type simply follows the reference designs’ component structure and their quantities in the context of their immediate parent.

ITEM NO.	PART NUMBER	QTY.
1	Assem2	2
1.1	Part1	1
1.2	Part2	2
2	Assem3	1
2.1	Part2	1
3	Assem4	2
3.1	Part3	5
4	Part3	1

Figure 2.5.1.2: Indented BOM of the reference design

A top-level only BOM includes only the first level components and their reference quantities to the table (Figure 2.5.1.3) and parts only BOM has only part -type models in the table with the total quantity of the part in that overall design (Figure 2.5.1.4).

ITEM NO.	PART NUMBER	QTY.
1	Assem2	2
2	Assem3	1
3	Assem4	2
4	Part3	1

Figure 2.5.1.3: Top-level only BOM of the reference design

ITEM NO.	PART NUMBER	QTY.
1	Part1	2
2	Part2	5
3	Part3	11

Figure 2.5.1.4: Parts only BOM of the reference design

2.5.2 Configuration grouping

Another important BOM -table setting is the *part configuration grouping*. Despite its name, it also affects assemblies. The setting is a choice between three possibilities:

- (1) Display configurations of the same part as separate items
- (2) Display all configurations of the same part as one item
- (3) Display configurations with the same name as one item

Choice (1) is the default and should make good sense without further explanations. Choice (2) groups components of the same model but different configuration to the same row under common parent. This is useful when configurations are not used to define the component, like having minimum and maximum angle positions of a joint as different configurations in an assembly. But since the option also groups different assembly configurations together and it is possible to have varying amounts of components in assemblies using configurations, it is trivial to make a design for which SOLIDWORKS fails to calculate any meaningful quantities for its BOM using this setting. The previous reference design is revised with following changes to prove a point (Figure 2.5.2.1):

- Part3 has two configurations, Default and C2
- Assem4 has two configurations, Default and Stripped
- Assem4 in Default references to 2 instances of Part3 in Default
- Assem4 in Stripped references to 3 instances of Part3 in Default plus 2 instances in C2
- The main assembly references once to Assem4 in Default and once in Stripped, and once directly to Part3 in C2.

The indented BOM with configuration option (1) follows the reference structure and the result is as one would expect (Figure 2.5.2.2). However, when using the configuration option (2) to group different configurations together, the resulting BOM has quantity and configuration notes that do not make any sense (Figure 2.5.2.3). While the grouped amount of Assem2 is 2 (1 Default + 1 Stripped), one could expect the quantity of Part3 in grouped Assem2 to be sum of Part3 instances in both of those parent assemblies (1*2 + 1*3 + 1*2 = 7) instead of the given 2. It seems that the structural quantities are actually retrieved from the first occurring instance in the BOM and as in this case the first occurring instance is in Stripped -configuration, the given quantity is 2. To pour some more salt on this, the configuration column also incorrectly shows the configuration to be Default.

According to SOLIDWORKS Helps (SOLIDWORKS Online Help, 2020, p. Bill of Materials PropertyManager), the third configuration option should group together parts with the same configuration name under the same parent. However, in SOLIDWORKS 2018 SP3 this option does not seem to have any effect on BOM with the revised reference design which seems to be a software bug. One could expect at least Part1 and Part2 to be grouped together under Assem2 as they are both parts under the same parent referenced in configuration named Default. If the option would work and only for parts as advertised, it should not have similar structural issues as with the previous option as only assembly grouping may have structural differences. However, the previous option was also described to have effect on parts only but proved to work differently. So, it is reasonable to assume that even if this option would work, it could still result in invalid BOMs in trivial design cases.

Intention of this demonstration was to prove that while many cases seem trivial to create BOM based solely on the SOLIDWORKS component structure, it can still be non-trivial when all supported BOM types are considered even with a simple design. Using other than the default configuration grouping option is something that cannot be done without fully understanding its effects. The author, despite about a decade of experience how different companies use SOLIDWORKS, cannot come up with a single case to which it would be recommended. Still, it is not uncommon to see designers using them.

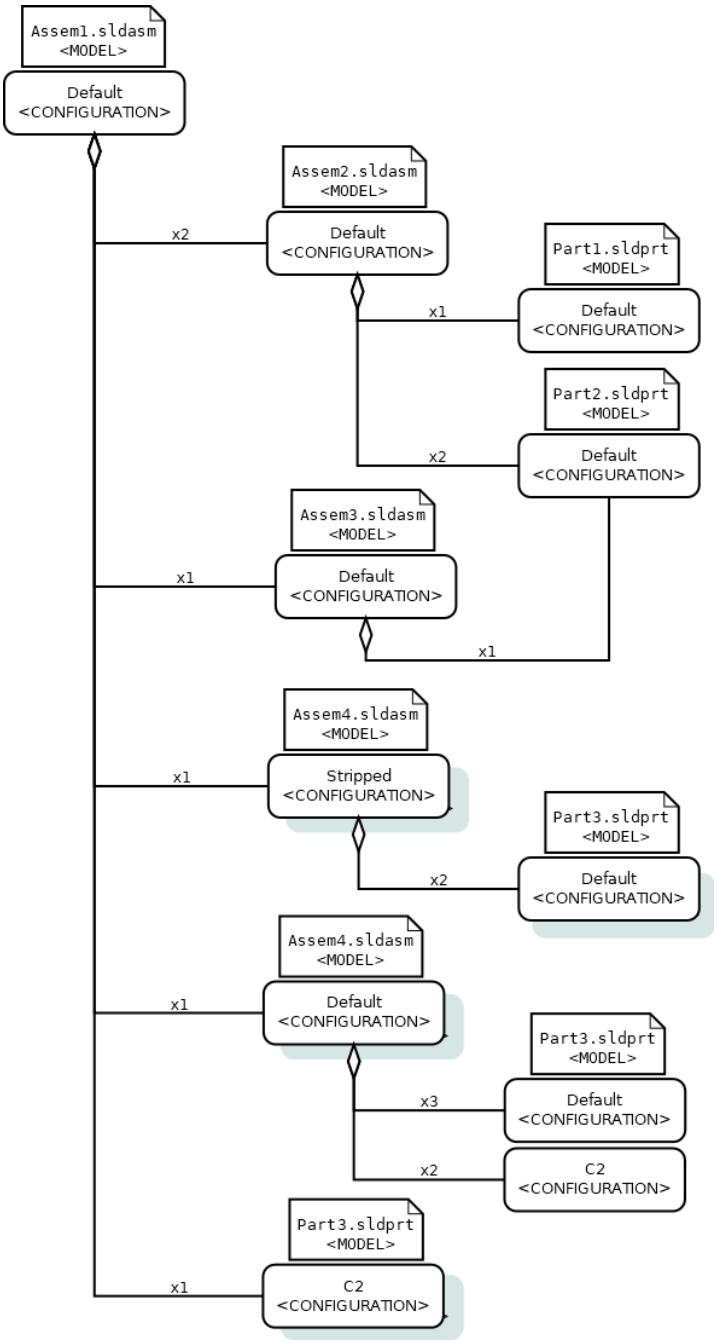


Figure 2.5.2.1: Revised reference design with configured component references

ITEM NO.	PART NUMBER	SW-Configuration Name(Configuration Name)	QTY.
1	Assem2	Default	2
1.1	Part1	Default	1
1.2	Part2	Default	2
2	Assem3	Default	1
2.1	Part2	Default	1
3	Assem4	Stripped	1
3.1	Part3	Default	2
4	Assem4	Default	1
4.1	Part3	Default	3
4.2	Part3	C2	2
5	Part3	C2	1

Figure 2.5.2.2: Indented BOM with separated configurations of the revised reference design produces expected results.

ITEM NO.	PART NUMBER	SW-Configuration Name(Configuration Name)	QTY.
1	Assem2	Default	2
1.1	Part1	Default	1
1.2	Part2	Default	2
2	Assem3	Default	1
2.1	Part2	Default	1
3	Assem4	Default	2
3.1	Part3	Default	2
4	Part3	C2	1

Figure 2.5.2.3: Indented BOM with grouped configurations of the revised reference design. Part3 in Assem4 doesn't seem to reflect any kind of reality from any perspective.

2.5.3 Design time BOM modifiers

Assuming that some assembly is designed as assembly just because of some technical reasons or maybe it is a purchased component which child components are simply irrelevant in BOM. For this case, the designer may set "*Child component display when used as subassembly*" -option from the assembly's *Configuration Properties* to **Hide**. This will have an effect in BOM that when the assembly is referenced in that configuration in some other assembly, it will be treated as a part in the other assembly's BOM. (SOLIDWORKS Online Help, 2020, p. Configuration Properties PropertyManager)

Another case is when some parts are designed inside an assembly just to have them nicely grouped under the same parent. In this case the assembly that groups the parts has no meaning from the BOM point of view and should be omitted. The same setting as in the previous case has an option to **Promote** child components and in the BOM it will then appear as the components of that assembly would be directly referenced by the parent assembly. (SOLIDWORKS Online Help, 2020, p. Configuration Properties PropertyManager)

A trivial design time BOM modifier is flagging components to be excluded from BOM. This is useful for example when having a reference model in the current design just to provide some design context, but it obviously does not belong to the BOM of the current model. (SOLIDWORKS Online Help, 2020, p. Excluding Assembly Components from a Bill of Materials)

2.5.4 SOLIDWORKS BOM as automatic data source for ERP

As demonstrated in Sections 2.5.1. and 2.5.2, only an *Intented* type SOLIDWORKS BOM with configuration grouping set to *Display configurations* of the same part as separate items can be used as a starting point for dynamic itemization of the design as only that structure type is always correctly representing the model. With component specific *Exclude from BOM* -setting it is easily possible to leave out specific components that were required during the design process but really do not belong to the actual designed product at all and therefore also to its bill of materials. Ignoring an assembly as an item while still correctly having its child components listed in BOM and ignoring child components of an assembly in BOM are both also very achievable using the *Hide* or *Promote* option in assembly's configuration setting *Child component display when used as subassembly*.

While all the pieces should be there to create a fully automated ERP itemization from the design using the SOLIDWORKS BOM, this does not come without issues. First, the settings are ridiculously complicated to find and to understand what they represent when used for this purpose. So, it is not reasonable to expect they would be used as we would now expect or at all during the design process for existing models unless there had been another reason to need this type of BOM already from the start. Secondly, as the settings are component specific, they will not just affect that one BOM we would care about but all BOMs of that product design and its sub designs. For instance, all assemblies would likely be preferred in BOM at manufacturing and assembly line but if a setting promotes one's child components because it is not an item, the assembly line will have a difficult time figuring out the design with their incomplete BOM. Because of these reasons the SOLIDWORKS BOM cannot be used as a data source for generic ERP integration. In other words, SOLIDWORKS' *Engineering BOM* is not configurable enough to be able to produce *Manufacturing BOM* on its own. (Xu, et al., 2007)

2.6 Common requirements for SOLIDWORKS - ERP integrations

The list of common requirements is based on subject matter experts' experience in SOLIDWORKS – ERP integration sales and projects. The thesis author has composed the list by consulting the experts in related sales cases as well as project implementation and delivery. All interviewed parties have 15 to 20 years of professional experience on this exact area; having been involved directly or indirectly in up to hundred cases.

Subject matter experts (Salonen, et al., 2020):

- Tero Salonen, Product Director, ATR Soft Oy
- Francois Simon, Sales Manager, ATR Soft Oy
- Eric Franc, Project Manager, ATR Soft Oy
- Tim Rosendahl-Halvrosen, Certified SOLIDWORKS Expert, CADWorks.dk

The first requirement is to be able to export all or specific components of the current model to the target system in batch. This includes that the target system usually has expectations and requirements regarding the model specific data and how it is represented during the operation:
R1 - Must be able to create items to target system in batch by analyzing the current design and model specific Custom Properties.
→ "Export the design as items to ERP respecting property requirements of the ERP"

Because individual models can be included in multiple structures that should be able to be exported to ERP as in Requirement R1, therefore:
R2 - Must be able to export the design or its sub-designs multiple times to target system without issues.
→ Must be able to map model and corresponding ERP item to correctly handle multiple exports.
→ "Unique identification of models and target system's items" is required.

A secondary use case is to create a new model (or map legacy model) for an already existing EPR item. The model can still be part of multiple designs and get exported with them multiple times, so:
R3 - Must be possible to create a new design for an already existing item in ERP so that the item will be mapped as in Requirement R2.
→ "Design time ERP item mapping"

In many cases only certain fields at the ERP should be updated based on the values filled by the model's designer and sometimes some values can be conditionally updated, therefore:

R4 - Must be possible to define data update/ownership model per Custom Property, e.g., Update "Description" to ERP item only if item is new or description at ERP is empty.
→ "Item data update rules and ownership"

It is almost never a requirement that only items should be exported, but usually the models' structures should also be interpreted for the target system as some sorts of Bills of Materials consisting of those items. In many systems, BOMs are separated from items as individual identifiable entities so the same identification rules apply for them as for items. Target system BOMs can also vary and if it does not support BOMs within BOMs, then usually a flat representation of model structure is needed. However, a structured one is a much more common case.
R5 - Solution for Requirement R1 must also create BOM(s) based on design structure and map/identify them similarly as items in Requirement R2.

a. Flat BOM. Only created for the ERP Item corresponding to the topmost model.
b. Structured BOM. All (or specified) items will also have their own BOM at ERP.

Expectations for structured representation may vary. Some have practiced strict discipline with BOM modifiers and expect their item/BOM representation to follow SOLIDWORKS' BOM as an exact match. Others may expect the BOM structure to be configurable based on property value rules and be more forgiving regarding those BOM modifiers. Everyone expects the BOM to be at least based on current design structure. Therefore:
R6 - Structured BOM(s) (Requirement R5b) must be

- dynamically based on the design structure.
- configurable to consider if assembly BOM modifiers Hide and Promote, and component BOM modifier Exclude from BOM are applied to structure or not.
- overridable based on predefined Custom Property value (e.g., "Ignore BOM if Purchased = Yes").

After the first export, or even before it in legacy data cases, the BOM exists at the target system. As it might get supplemented in ERP with items that are not part of the design or even get completely modified, there is a very common non-trivial problem with BOM ownership. The model might change between exports so from that point-of-view it has the correct BOM. On the other hand, if the BOM is supplemented at the ERP, there is no generic way to differentiate that case from a model item that was dropped out of the design in later export; and that was only one trivial but problematic example of many. This requirement generally is entirely custom handled as all possible solutions have a drawback. But it is also almost always required to be handled based on some set of rules, so:

R7 - Must consider that BOM(s) to which mapping should be done might already exist in ERP before first export. Also, BOM(s) can be changed and supplemented in ERP which is usually not allowed to be overridden.
→ "BOM update rules and ownership"

Design companies are usually looking to invest in ERP integrations after they have noticed they cannot work efficiently without one anymore. At this point, the amount of existing data can be huge, and it cannot be thrown away just because a new system does not like to work with it. For these types of systems, a data migration step is usually required (Muni Prasad, et al., 2013). However, being able to use existing models with a new system has obvious benefits; including being able to test the new system side-by-side with the old one before deploying it for the whole environment.
R8 - Must work for existing models with preferably no migration steps.

While not unheard of, it is not usually a preference to let the integration to execute on background based on just a set of given rules. Majority would not likely go with that type of solution at all even if everything else could be precisely handled. Therefore, as a common requirement:
R9 - Must be able to see a clear visual representation of what the integration is about to do.

Many ERP systems also like to have item preview pictures, link items to their actual up-to-date manufacturing documents or to even host them. It is a very common requirement to be able to provide these pictures, documents, or links to them while exporting to ERP.
R10 - Must be able to provide up-to-date preview, manufacturing documents and/or links for the ERP while creating/updating the items.

The model to item mapping discussed in Requirement R2 can many times be based on a value generated during the design process. However, many times it is the ERP system that provides the unique identification of its items. For this to be possible, the identifier must be stored to models during the batch export process, which can be a big problem if a PDM is used underneath. Therefore, it makes sense to separate this requirement from Requirement R2:
R11 - Must be able to let target system to provide unique identification of its items.

3 CUSTOMTOOLS for SOLIDWORKS

CUSTOMTOOLS is a set of applications including CUSTOMTOOLS Viewer, CUSTOMTOOLS Administration, CUSTOMTOOLS Task Add-In and CUSTOMTOOLS for SOLIDWORKS which all work in the same ecosystem. The last mentioned one is the actual SOLIDWORKS add-in that does most of the heavy lifting and so is often called CUSTOMTOOLS or CT for short. Its first version was released in 2000 and since has gained user space of somewhat under 10,000.

CUSTOMTOOLS environment consists of Microsoft SQL Server, single or multiple CUSTOMTOOLS clients and a common windows file share location. The database holds profiling settings, last known design file locations, last saved property values and reference data which are all available for all environment users.

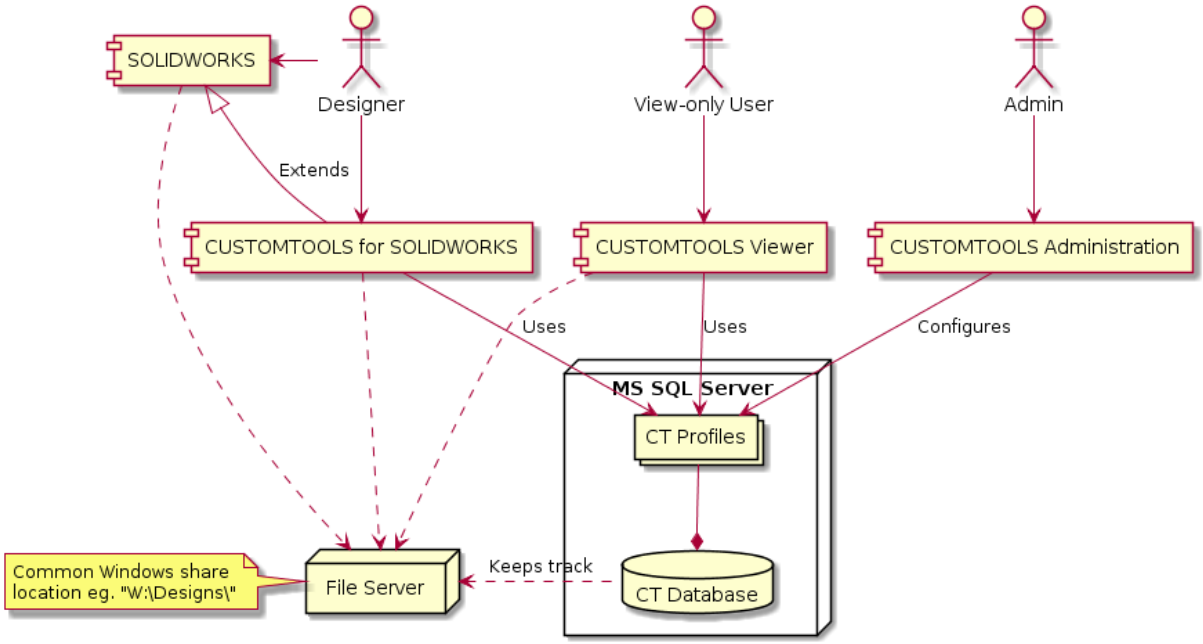


Figure 3.1: CUSTOMTOOLS Environment

Most important features of CUSTOMTOOLS in the scope of this thesis are its ability to provide consistency in design data via shared environment profiling, ability to build several different types of BOM structures following and overriding the design time modifiers, to include the consistent model data to the BOM and possibility deploy target system specific integration scripts to export the items and BOM(s) to various ERP -systems.

When CUSTOMTOOLS is taken into use for an already existing design environment, it is important to configure its Properties to match the existing environment as closely as possible. SOLIDWORKS' way of storing data to the models and drawings is document and configuration specific slots, Custom

Properties, that are addressed by string keys and can point to data of a few different base types like string, integer and boolean. CUSTOMTOOLS refers to these string keys as attributes and with its profiled Properties it provides a new access layer for the Custom Property data with much more specialized data typing and filtering possibilities.

3.1 Properties

CUSTOMTOOLS Properties is the profiled data access model for SOLIDWORKS model's configuration specific Custom Properties. Each CUSTOMTOOLS Property (*CT Property* from now on) is labelled and the label can differ from the actual target Custom Property. CUSTOMTOOLS calls SOLIDWORKS' Custom Properties *Attributes* and so each CT Property is bound to a single Attribute. The document level Custom Properties (Attributes) are generally referred to as *Document Properties*.

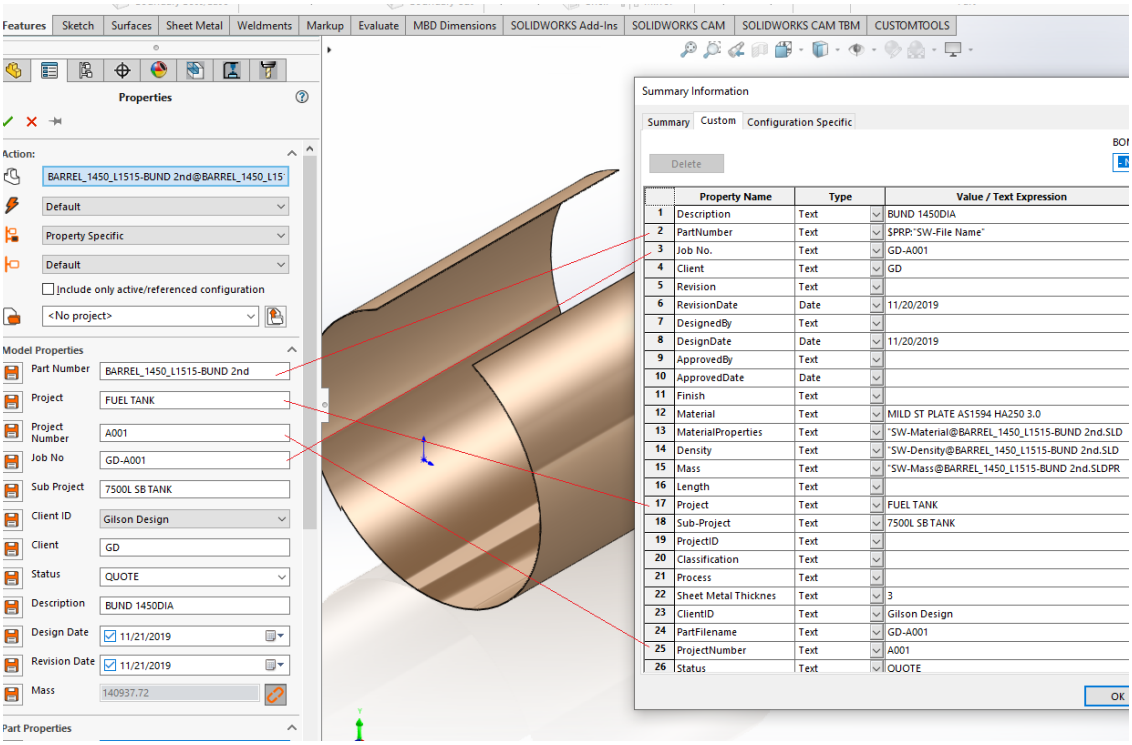


Figure 3.1.1: Illustration of CT Property mapping with SOLIDWORKS Custom Properties

3.1.1 Attribute inheritance model and Initial Configuration -setting

CT Properties can behave exceptionally well also from *a new user with existing models* -point-of-view due to its attribute inheritance model. In it, each CT Property first tries to map to the attribute of the active configuration. If not found and so defined in the profile, parent configurations are tried up to the top level and then the document level. This causes CT Property to map to whichever configuration the attribute was originally filled and allows modifying its value from the same view. Thus,

CUSTOMTOOLS Properties does not provide a view to just one set of Custom Properties but something that could be called a dynamic mixture of Custom Property views built to represent earlier usage. While it may sound complex, from the user point of view the inheritance model stays completely hidden.

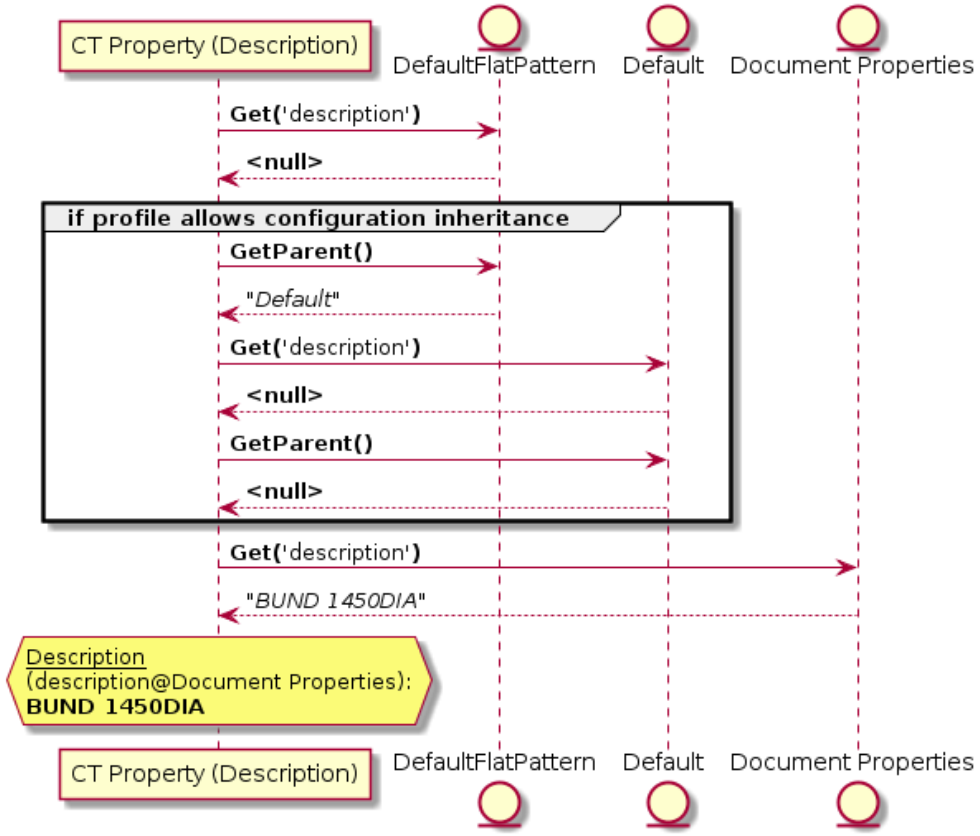


Figure 3.1.1.1: Assuming model with *Default* -configuration and its derived configuration (aka. child-configuration) *DefaultFlatPattern*, which is also the currently active configuration. CT Property tries to bind to the attribute from the most accurate configuration down to Document Properties until an existing attribute is found. This operation is invisible to the user.

If the data inheritance does not yield results, the CT Property chooses target configuration to bind based on its profiled *Initial Configuration* -setting. The setting may even target the property to multiple configurations at once but best practise is to use *Active Configuration* for all properties that may have different values between configurations (eg. Dimensions, Mass, Material) and *Document Properties* for values that are always document specific like Designer and thus can be dynamically inherited for all configurations.

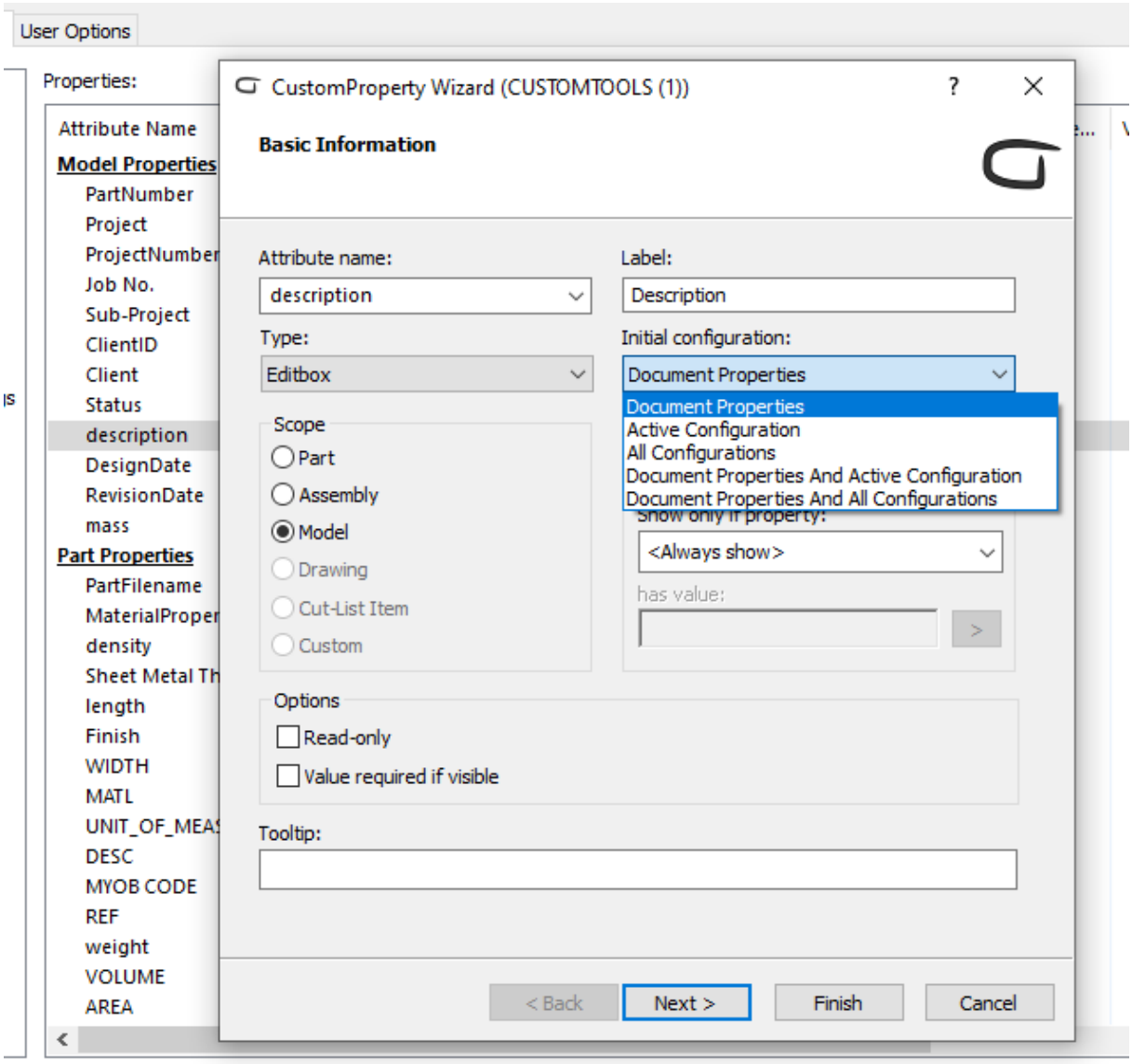


Figure 3.1.1.2: CUSTOMTOOLS Property Wizard, *Initial configuration* -setting of *Description* -CT Property.

3.1.2 Settings for Properties used to identify ERP Items

From ERP point of view the importance of the CUSTOMTOOLS Properties' inheritance model and the *Initial Configuration* -setting comes from a simple common integration requirement: *Unique identification of models and target system's items* (Section 2.6, R2). As the attribute data is used to identify an ERP item, having that identifying attribute data in *Document Properties* effectively causes every configuration of that design to be mapped to the same ERP item. However, *setting Initial Configuration* to *Active Configuration* is often not enough because existing models tend to already have some sort of ERP Item identifying data in their Document Properties which also should be used. In this case when using CT Properties, the inheritance model accesses the attribute from Document Properties and just keeps editing it causing gray hair for the user (Figure 3.1.2.1).

However, using *Document Properties and Active Configuration* saves the value for both configurations regardless of from which it was loaded. First time the properties are accessed and saved, the value comes from Document Properties and ends up as a more specific attribute for the active configuration and also to Document Properties. Then when the same is done for any other configuration, it will also get the value from Document Properties and store it to both. While this operation modifies the value in Document Properties, change in it has no effect as both configurations already have more specific attributes in their own contexts.

Therefore, as a general recommendation, CT Properties that link to ERP Items should be configured to use *Document Properties And Active Configuration* as *Initial Configuration*. This will cause identifying attribute data to be correctly retrieved for existing models but also to convert them to configuration specific items with just by normal usage of CUSTOMTOOLS Properties (Figure 3.1.2.2).

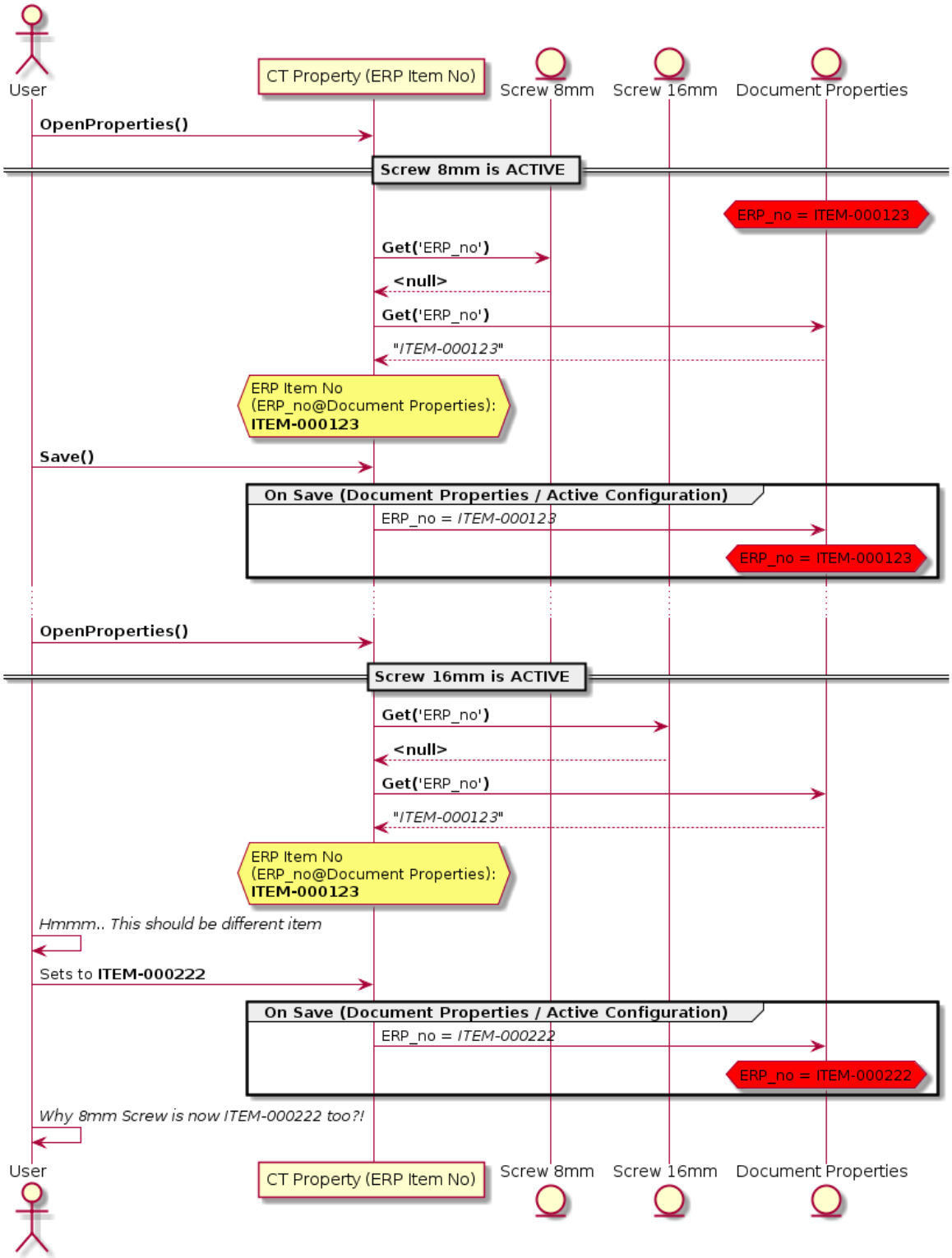


Figure 3.1.2.1: Using Document Properties or Active Configuration as Initial Configuration for existing models may result in undesired and unexpected situations.

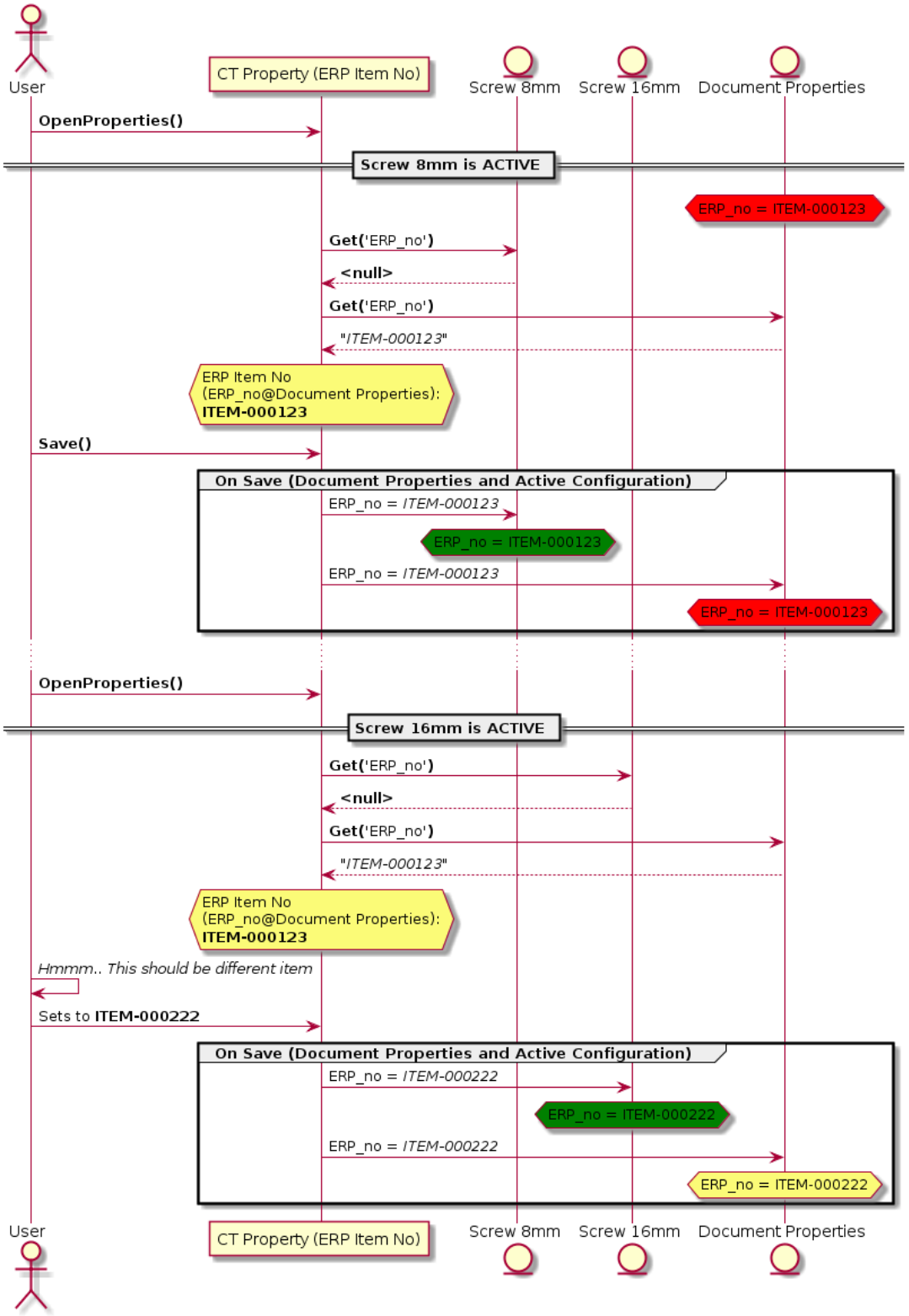


Figure 3.1.2.2:Using Document Properties And Active Configuration as Initial Configuration intuitively prepares the existing models as configuration specific items.

3.1.3 Property types

CUSTOMTOOLS Property types are Checkbox, Combobox, Date, Dimension, Editable Combobox, Editbox, Hierarchical Combo and Info. Type of a property is selected based on the characterics of the actual information that is going to be stored with the property. For example a combobox would be good for selecting a manufacturing method from a predefined list while a checkbox would be good to determine if the component is purchased or not. It is also possible to hide controls based on values of others; for example Manufacturing Method -combobox could be set to be hidden when Purchased -checkbox is checked.

The Custom Property string value of a checked/unchecked checkbox can be freely formatted, thus it can be easily configured to work for legacy designs too. A checkbox could also be replaced with a combobox, for example with two values like Yes and No. This is actually fairly common as it has the benefit that it is easy to see if a user has actually made a selection or not, which is not trivial with an unchecked checkbox.

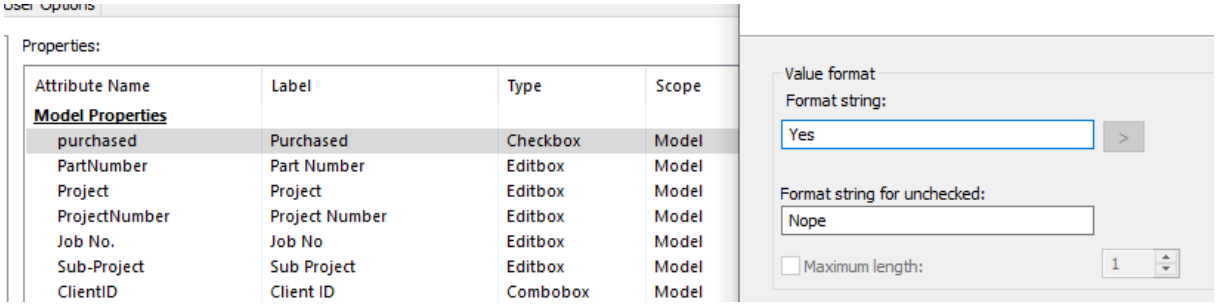


Figure 3.1.3.1: Settings for Checkbox values

Comboboxes get their content from various sources but the retrieval system is designed to temporarily append the current Custom Property value to the combo in case it doesn't exist there. This allows combobox -properties to preserve stored legacy data even if that data selection would not be valid for newly created models anymore. *Combobox* is a dropdown list that only allows selecting a value from the provided list and Editable Combobox allows also typing any arbitrary value as a value.
Date -properties allow fully defining the date format which again is perfect also for legacy Custom Property -data. It is also later possible to change the date representation format but in order to preserve the existing date data, it must have been saved for the model using CUSTOMTOOLS Properties at least once. The user interface for this property is a calendar.

The dimension -type is a CUSTOMTOOLS speciality that allows linking dimensions from the model to the properties. This type stores the actual SOLIDWORKS dimension's identification path which can be evaluated into a value in defined metrics. Values of these properties change dynamically as the model changes.

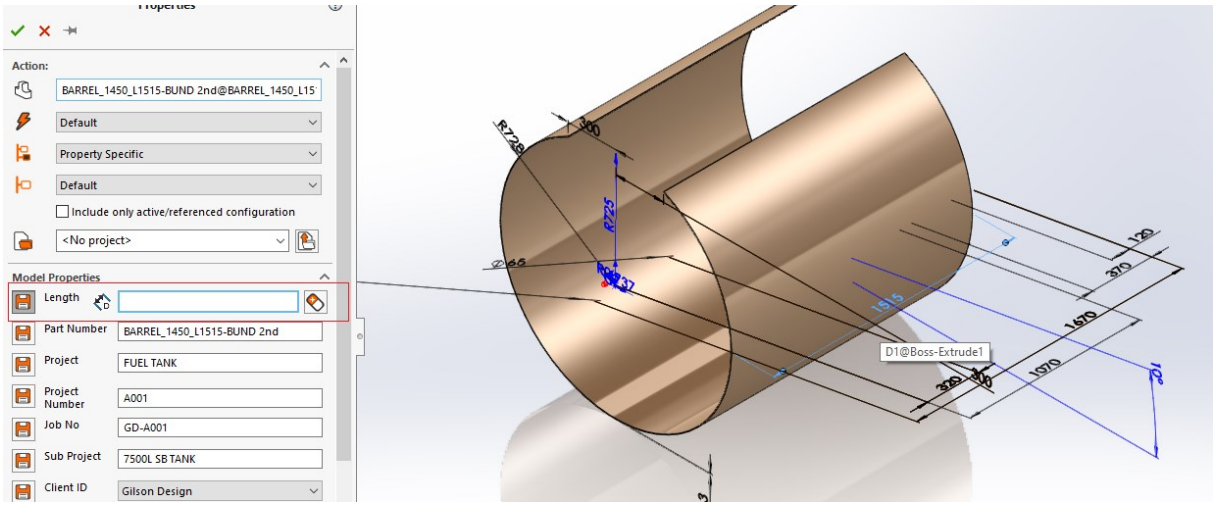


Figure 3.1.3.2: On click of a dimension -property, all models dimensions appear for picking. Value of the property is then linked to the picked dimension and thus also updates dynamically.

Hierarchical Combo represents predefined values that have dependencies to other values as a treeview dropdown. The user selected value is then stored to Custom Properties as a path from the root parent down to the selected item using pipe character (|) as an item separator. This is a very useful type when the list of possible values is large but categorizable.

Editbox is simply a text field and Info is a multiline field. Editbox is usually preferred as it supports more advanced scenarios.

All of the property types also have miscellaneous features like forcing value in uppercase and limiting its length; which both are surprisingly often needed when integrating with an ERP. Almost all types also support a variety of *Functions* that may for instance generate a specific value for the Custom Property or get some model specific dynamic data like mass.

3.1.4 Property functions

CUSTOMTOOLS Property Functions *are Before Function, After Function, Button Function* and *Data Function*. At the scope of this thesis it is not necessary to explain all of them in depth.

Before functions provide some content or value for the property and are executed when the Properties are visited. Valid before functions are *GetCurrentDate* (Date -property), *GetCombinations* (Combobox), *GetMaterials* (Combobox), *GetMass* (Editbox) and *GetUserInitials* (Combobox, Editable Combobox, Editbox).

After functions are executed after the property value has been changed. Valid functions are *SetColor* and *SetDensity*. For instance, *SetColor* applies the selected color in property value for the model.

Button functions create a small button next to the property in Properties and its valid functions are *GetCode, GetColor, GetDatabaseItem, GetRALColor, OpenDictionary, Revisions* and *SetEntityData. GetDatabaseItem* allows mapping a group of properties with an item in an external datasource which is one of the key features required in ERP integrations. *GetCode* is also important as it is often used in generating unique id-sequences for models. Its functionality can also be extended to retrieve a serial from a 3rd party system like an ERP.

Data functions retrieve values based on other properties. Valid functions are *GetCombinationValue, GetListKeyValue, GetParentItem* and *GetTranslation.*

3.2 Lookup Lists

Lookup Lists provide content for (Editable/Hierarchical) Combobox -properties and they support three different data sources: *User specified, Database* and *Custom*. *User specified* is simply a list that is manually filled in CUSTOMTOOLS Options.

Database -source option allows defining a query that is executed in the CUSTOMTOOLS Server and the returning resultset is interpreted as lookup list content (Figure 3.2.1). Since CUSTOMTOOLS is using Microsoft SQL Server, it is possible to define any third party system as linked server as long as a data provider exists for that system. This allows for example (but not limited to) all MS SQL Server, Oracle, Access, MySQL and even Excel based systems to be directly linked as combobox data sources in CUSTOMTOOLS Properties. This is very useful for eg. providing a list of allowed values for properties that would be used in sync with the ERP. One of the most common cases is retrieving allowed values for the property that is used as a *unit of measure*.

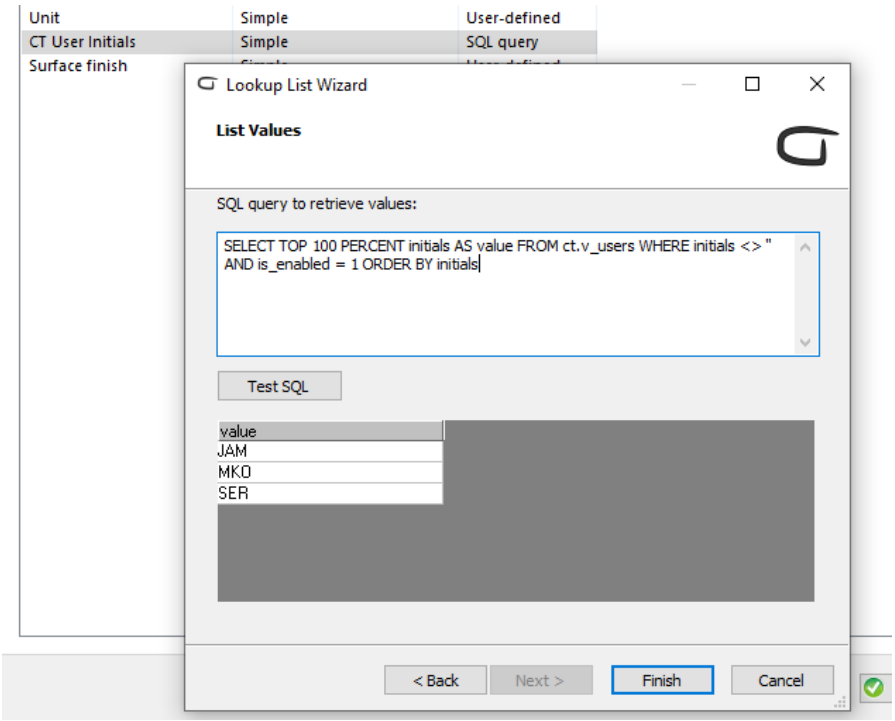


Figure 3.2.1: A lookup list may get its value also using an SQL query. This example pulls user initials of CT users into a list. It is fairly common to add the designer's initials to the document properties.

Custom -source option expects a CUSTOMTOOLS extension script to provide the content for the list. This is particularly useful if the list content must be queried some other means like using a web service. While this option can be bent to almost any possible use case, utilizing it requires C# or VB.NET programmer skills.

In addition to configurable data sources, three different lookup list types are supported: *Simple, Key-Value List* and *Hierarchical.* Simple -list is just a list of allowed values that can be attached to combobox's content without any further functionalities.

A Key-Value List allows defining a value list having a maximum of 15 arbitrary keys attached to each value. Those can be pulled up to 15 secondary properties with Property Data Function *GetListKeyValue* on a change of the combobox value selection. This is sometimes used to show "user friendly" values in visible combos (eg. customer names) while actually having all other logic and possible integrations to rely on the more identifier-like key value written to some hidden property. Also pre-defined decision making is a fairly common use case (Figure 3.2.2).

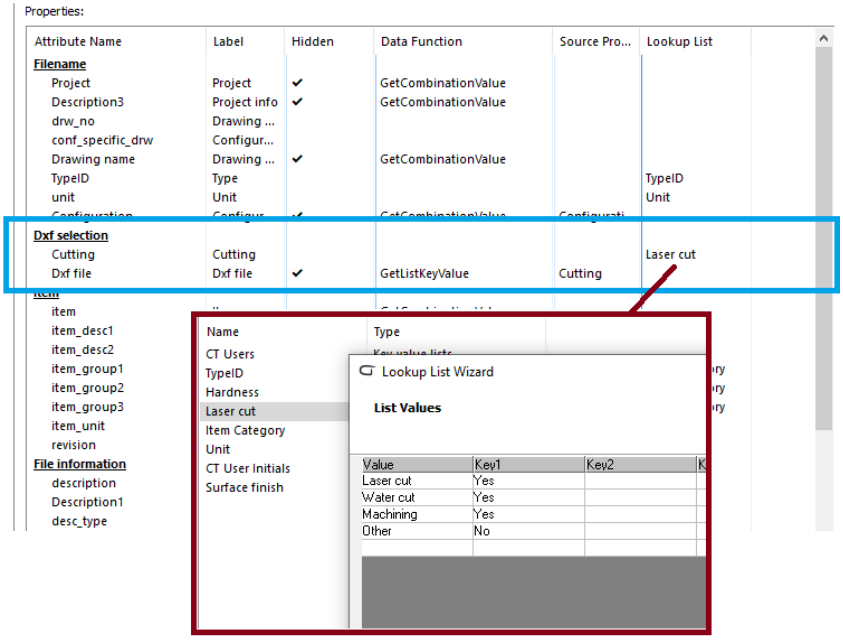


Figure 3.2.2: In these properties it is defined that the user has Laser cut, Water cut, Machining and Other as possible Cutting -property selections that come from a User Defined Key-Value list. Whether or not that selection requires a dxf -file to be generated at some later point, is retrieved to a hidden Dxf file -property from that list's key.

Hierarchical lists can be used as multilevel selections in which the content of a combobox depends on a selection of another (Figure 3.2.3). Some ERP systems may have similar, arbitrary deep selection models for specific types and categories.

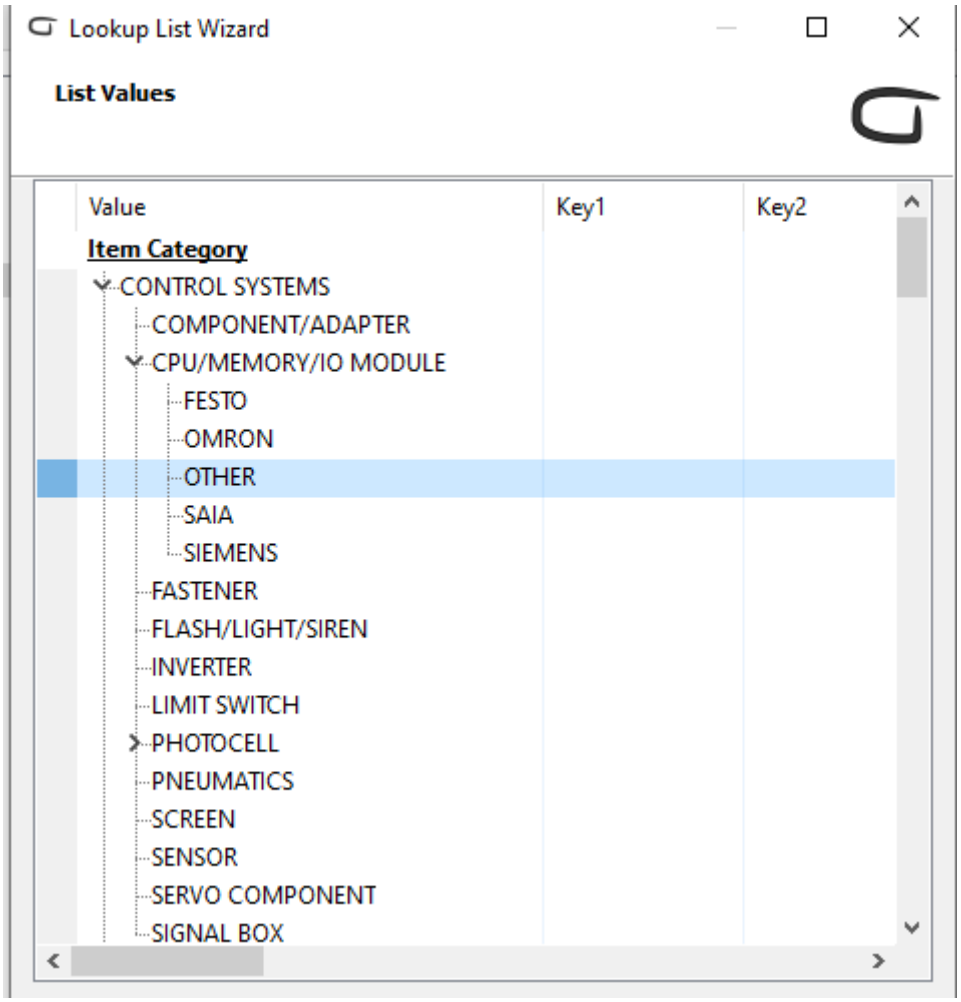


Figure 3.2.3: To select “OTHER”, the user must first select “CONTROL SYSTEMS” of a combobox. Then the user must select “CPU/MEMORY/IO MODULE” from a second combobox that uses the first combobox as a content source. Only then the user is able to select “OTHER” from yet a third combobox that uses the second one as a content source.

3.3 Search Group linking

CUSTOMTOOLS allows defining *Search Groups* that are essentially SQL table queries that return a resultset of rows containing queried sets of values. To successfully define a Search Group (Figure 3.3.1), the target server must first be added as a linked server using CUSTOMTOOLS Administrator. As in the Lookup List -case, also this can use all possible sources to which a data provider exists. In case the target system does not have a provider or possibility to be linked with same server (e.g., web service-based systems), it is also possible to create a Search Group that uses custom C#/VB.NET -handler script as result set provider.

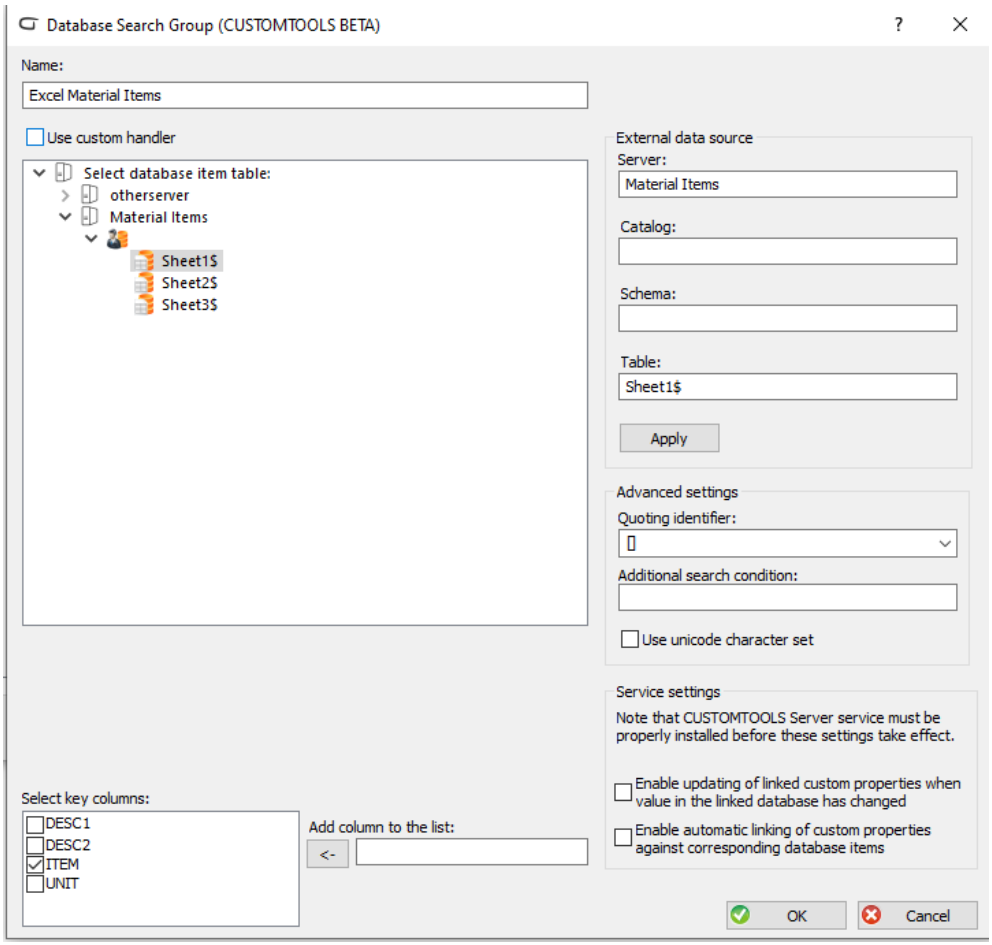


Figure 3.3.1: When the target server is linked with CT's SQL Server, a Search Group can be defined to link to its table. Here an Excel Material Item -search group is defined to pull possible material sets from an excel file that has been added as a linked server.

A Search Group always requires one or more source fields to be treated as keys. Keys combined should result in a unique identifier within the result set of the queried system. This is important to have well-defined mapping of values between systems, which is also the key insight for solving Requirements R2 and R3 (Section 2.6, R2-R3). Not surprisingly, the most common use case for Search Groups is to be able to map the model with an existing ERP item. Another use case is to get material information e.g., from which kind of square bar the model should be manufactured (Figure 3.3.2). Note that an ERP integration might require mapping with multiple Search Group datasets (e.g., item and material) for a single model to have a meaningful itemization dataset from the ERP point-of-view.

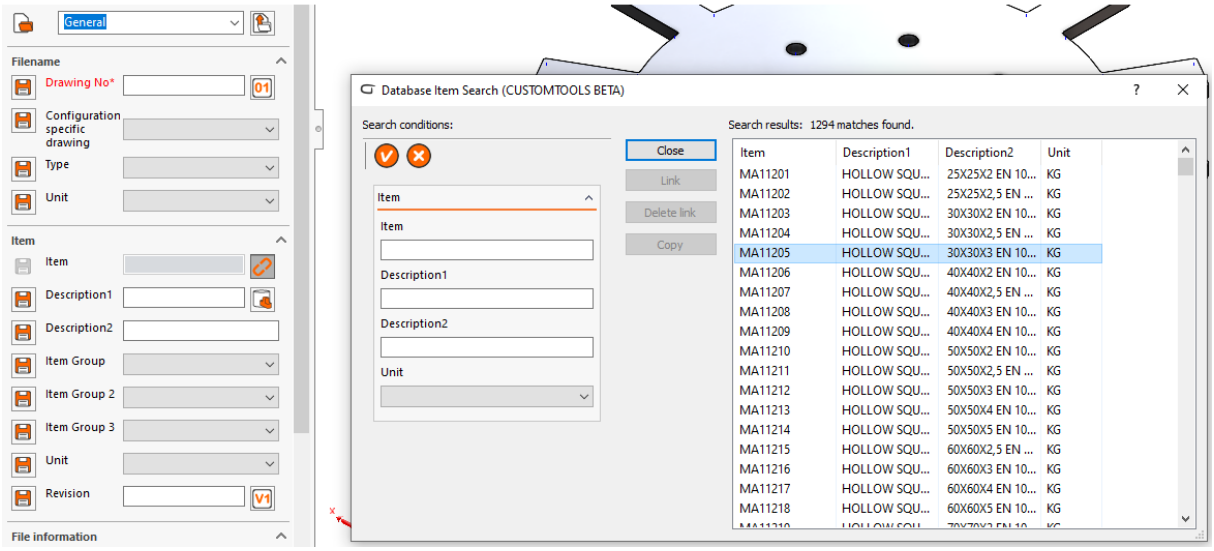


Figure 3.3.2: Database Item Search is invoked from a property that has *GetDatabaseItem* -button function. The Search Group bound to that property is queried for a result set covering values for all of its predefined properties and the user is able to select any row to be pulled as a value set into them.

3.4 Export

CUSTOMTOOLS' Export is the marketing attraction of CT integrations as it is used for exporting entire models at once to ERP which is the number one of all ERP integration requirements (Section 2.6, R1). It supports predefining multiple bills of material views, Export Profiles, that can be used to generate a well-defined BOM from the current model. It also supports the same BOM modifiers as SOLIDWORKS' native BOM (Section 2.5.3) but it also allows manipulating them.

Supported BOM types that come "out-of-the-box" are *Top level only, Parts only and Intented Assemblies*, which correspond to the exact same available SOLIDWORKS' native BOM types described in Section 2.5.1. Also, the configuration grouping (Section 2.5.2) is supported even though there are no known use cases when the grouping would make foolproof sense (Section 2.5.4) at least from itemization to ERP point-of-view.

Export supports all types of cut list items too, but they are intentionally left out from the scope of this thesis for clarity. In case cut list items are needed to be included in the integration, they are handled the exact same way as parts, and their parent parts are then considered to have a BOM just like an assembly with parts would have. CUSTOMTOOLS also supports properties of cut list items with minor restrictions.

3.4.1 Export Profile and its types

An export profile is part of the CUSTOMTOOLS profile and there can be multiple of them. To create one, the user has to define a name, BOM type and configuration grouping to use, and if additional BOM inclusions like referenced drawings and cut list items are needed (Figure 3.4.1.1).

The export also supports up to 4 user defined data matrices called CustomScopes. CustomScopes are table-like data (set of CT Properties) that can be bound to a single CT Property in ordered fashion and be used for example to determine manufacturing steps and step specific attributes of a model. In some cases the ERP might expect to get model specific manufacturing steps or work phase -information for which this feature is usually utilized.

Export profile has also a *Profile* type which by default is an XSL -transform using the predefined style sheet to create an XML BOM to the predefined output path. However, the main usage of Export is with Script Add-ins that may take control of the whole export process, utilizing the data in the defined BOM view and pushing it to whatever target system the script is built to work with. Legacy style scripts can just subscribe to CTInterface Events API and handle all or self-filtered export profiles in event-driven manner. However, to unlock more core features, execute in more strict context and to provide a configurable interface, it is recommended to implement CT Extension -style scripts that expose their dedicated export profile type.

CUSTOMTOOLS 2020 SP1 includes 5 built-in configurable Export Profile Extension types:

- **Cloud Connected** to export items, structures and manufacturing document types to ROIMA Product Information Cloud
- **Excel Report** to export the BOM view to a templated excel file including design previews
- **Odoo ERP Integration**, a separately licensed export type capable of creating items and BOMs to on-premise Odoo ERP -system, including export of manufacturing documents and design previews.
- **Dynamics NAV Integration**, another separately licenced export type, capable of creating items and BOMs to Microsoft Dynamics NAV, link items to manufacturing documents and to use its item numbering system to generate identifiers for the models on-the-fly (Section 2.6, R11).
- **Oscar ERP Integration**, also separately licensed export type which has similar capabilities as the Odoo integration
- **Vertex Flow Export**, which is sold as both separately licenced integration and as a special feature stripped productization of CUSTOMTOOLS called CUSTOMFLOW. Its capabilities are similar to Odoo and Oscar and target ERP is obviously Vertex FLOW.

Even though these Export Types are provided as built-in solutions in the the product, they are purely using the same CT Extensions and CTInterface Events API as is publicly available for third party developers, with only 2 exceptions: they can use ATR Licensing to verify their availability for user and they can invoke CUSTOMTOOLS Help system.

As can be seen from the built-in extension types, another common requirement when exporting items to ERP is also to be able to provide up-to-date manufacturing documents and/or document previews (Section 2.6, R10). CUSTOMTOOLS has its Batch Operation tool for executing predefined conversion rules (eg. “Convert drawing sheets that have DXF in name to DXF files to this path”) that can also be set to be executed during the export. This allows the export handler to be able to retrieve those freshly converted documents and to include them in the export process in any way they are needed.

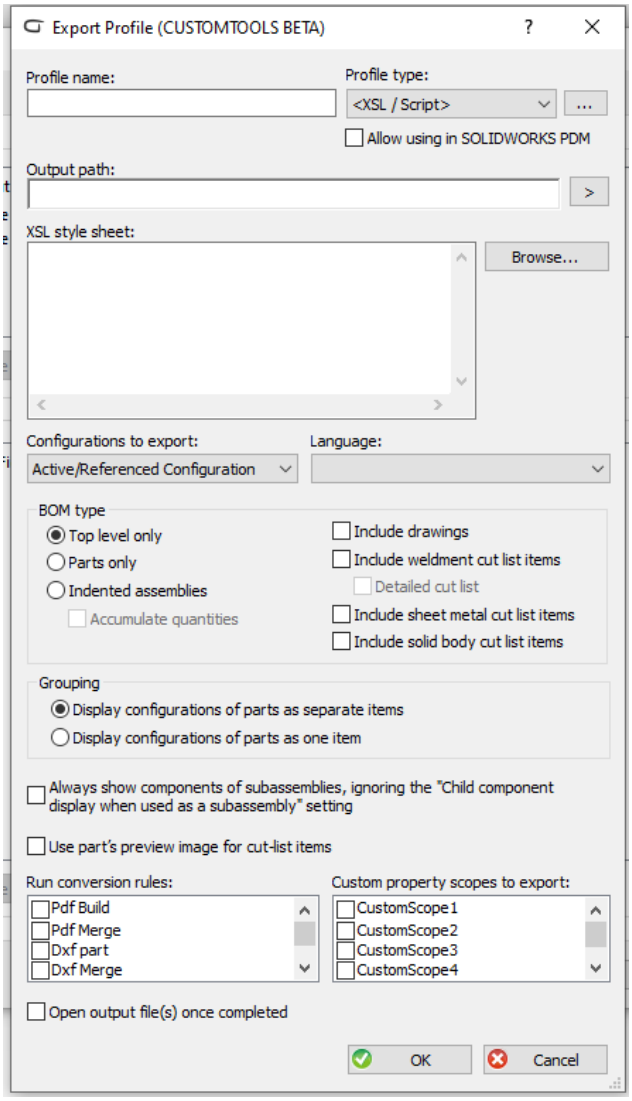


Figure 3.4.1.1: Setup page for an Export Profile.

3.4.2 Export Profile Fields

An export profile also includes a set of fields that collect the model specific data to the export view (Figure 3.4.2.1). The source for the data can be a special SOLIDWORKS property, like quantity or filename, or value of a Custom Property field profiled using a CUSTOMTOOLS attribute. Also just a value field without any source can be used for arbitrary purposes.

In addition, a field can have a maximum length to which the retrieved value is cut. This is useful in case the target system has restricted length for the field. However, it is recommended to have maximum length defined also for the source property so users will have feedback when filling the values. In the export the value is just cut without any user notifications.

A field can also be compulsory, which prevents users from executing the export in case the value of the field is empty.

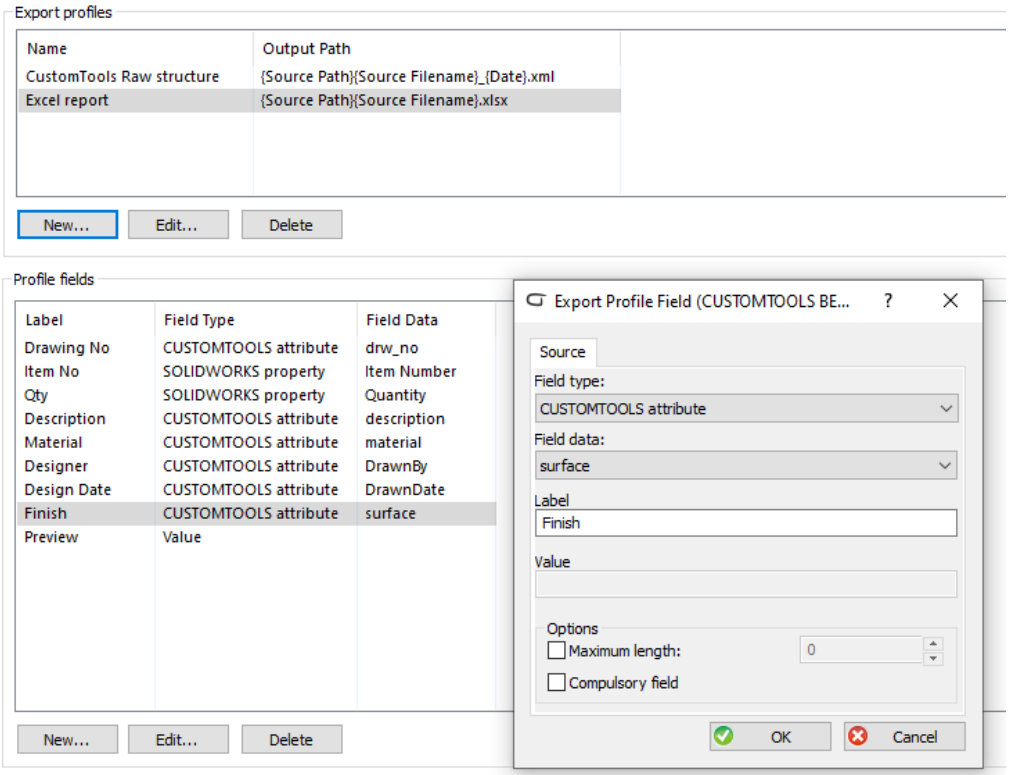


Figure 3.4.2.1: “Finish” -field of the Excel report -export profile has attribute ‘surface’ as its data source. Attribute is a CUSTOMTOOLS term for a SOLIDWORKS’ Custom Property.

3.5 Script Add-ins

CUSTOMTOOLS functionalities can be extended with script add-ins that utilize the public CTInterface Events API and CTExtension Interfaces API. A non-extension type script is considered legacy as they can potentially interfere with other scripts and extensions; however they are still supported to maintain backwards compatibility.

3.5.1 Architecture

An extension script has always at least 2 reference assemblies: *CTInterface.dll* and *interop.CTEngineLib.dll*. The latter one, CT Engine library, defines the native core object model of CUSTOMTOOLS to which CTInterface provides managed abstract base implementations for each of the supported extension types. A user extension is an implementation of the abstract CTExtension -class and it must implement **Hook** and **UnHook** -methods that subscribe and unsubscribe to events at CTInterface Events API. User extensions are not allowed to subscribe or unsubscribe events anywhere else and must also do so when the functions are called. The user extension is not allowed to call these functions itself. A user extension can return a specific class in response to **GetInterface** -call (overridable virtual function) and the returned class must be an implementation of corresponding *CTExtensions.Interfaces.XXX* -class. These classes are extensions of different CUSTOMTOOLS elements or core functionalities, like the Export, Options, some specific core objects etc. At the core level, everything is handled using the native CT Engine model but it is also all encapsulated and default implemented in managed code for user extension convenience and backwards compatibility of future additions. Architecture of an extension that can be used as an export type is described in Figure 3.5.1.1.

Compared to Extensions, the legacy style scripts are architecturally simpler but completely lack all discipline and sandboxing which are important when multiple scripts or extensions are introduced to the same system. Also, they are obviously not as capable as Extensions. A legacy script is as simple as a public class with a single public constructor having *ATR.CT.CTInteface.CTInterface* as an only argument. The script can then subscribe to CTInterface events in its constructor, but the obvious disadvantage is that the script must somehow be able to determine whether it should handle those events. It is basically never the case that an event handler should handle all possible invocations, so some sort of context testing is always required, and that is hard to make 100% correct.

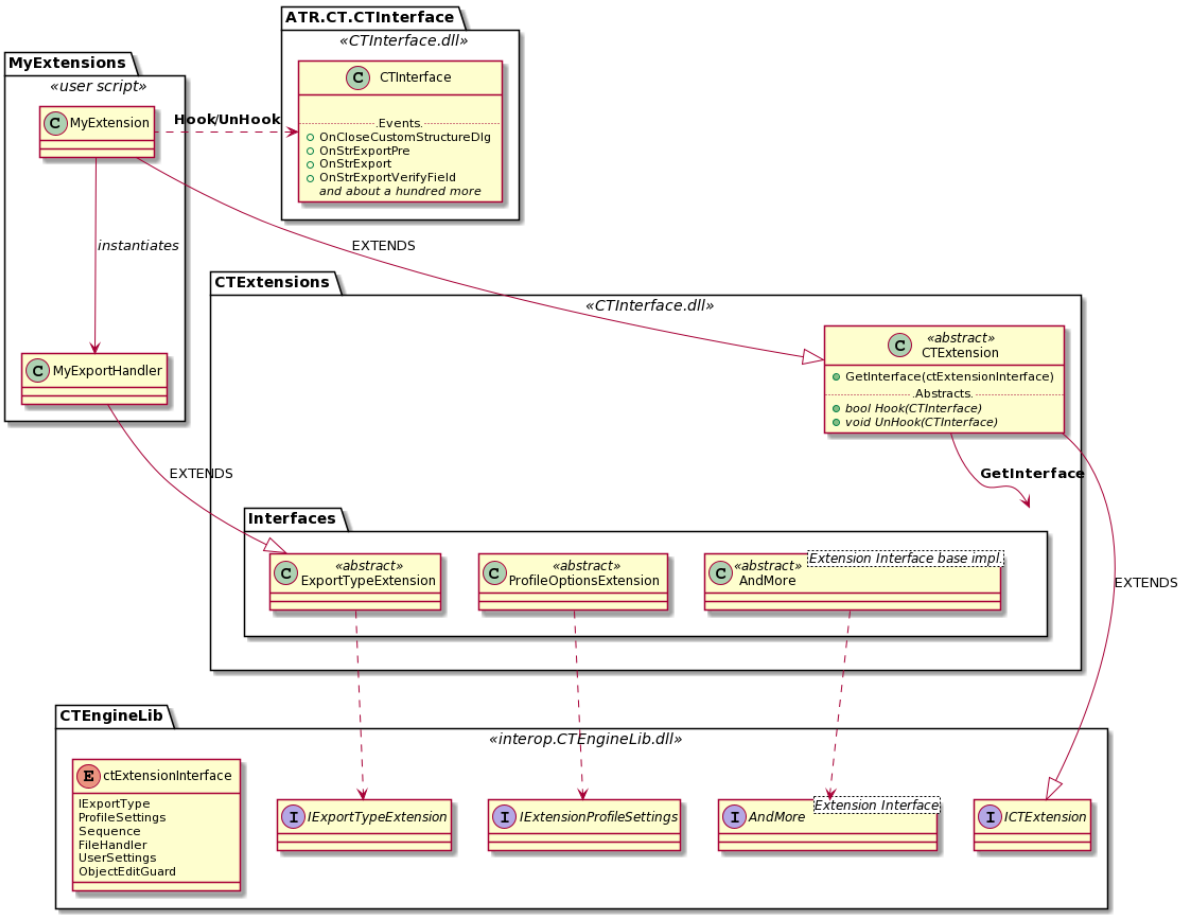


Figure 3.5.1.1: Overview of the class relations for Extension -type script add-in called *MyExtension. "AndMore"* -class and interface are just to indicate there are more similar interfaces and classes available in the system.

3.5.2 Deploying extensions to CUSTOMTOOLS Environment

As described in the beginning of this chapter, the CUSTOMTOOLS Environment is a shared environment for all of its users by the means of common database connection. Also the extensions are distributed for all clients of the environment. The extension scripts are managed code stored in plain text to the database and whenever a client application connects, it will check that the source code present in the local environment is up-to-date. If not, the latest source code is compiled on-the-fly against the user's local environment and the resulting assembly stored in the local application data folder. From there it works as an integral part of the locally installed CUSTOMTOOLS applications as it gets loaded to the application domain.

Extensions are added to the database using CUSTOMTOOLS Administration that has its own section for script management. When a new script is selected, a minimal example script is automatically added (Figure 3.5.2.1). Scripts are single text files but they may contain multiple classes and

namespaces. C# is usually preferred but it is also possible to use VB.NET. Language version of C# is 4.0 in CUSTOMTOOLS 2020 but it will be upgraded to C# 6.0 in CUSTOMTOOLS 2021.

As compiling managed code always requires some satellite assemblies, the following common ones are added automatically and does not have to be separately referenced: *mscorlib.dll, system.dll, system.core.dll* and *microsoft.csharp.dll* in case the selected language is C#. In addition to these, a script extension also requires at least references to *CTInterface.dll* and *Interop.CTEngineLib.dll* as can be seen from the Figure 3.5.1.1 in the previous section. These assemblies are located in the local environment at the CUSTOMTOOLS install location and can be dynamically addressed with **[CT_INSTALL_PATH]** -tag. It is also possible to deploy satellite assemblies with the script by adding them to additional files of the script and then reference them with **[ASM_PATH]** as path. In case of an integration to a third-party system, the referencing works exactly the same: in case the system to integrate is local, a common-for-all path must be known in order to reference its DLLs; and another option is to deploy the satellites with the script. CUSTOMTOOLS' install path provides a few commonly needed satellites like Microsoft Excel and M-Files public api.

Because the scripts are single file documents, they are easy to deploy as such from developer to customer environment and the only manual thing to do is to browse and add the references for the script. However, this is not needed if the script uses **@AUTO-REFERENCE** -syntax in comments in the beginning of the file. These are automatically parsed on compile and added as reference. Also, as the class name and language are automatically detected, the script deployment to the whole environment is basically a two-click operation.

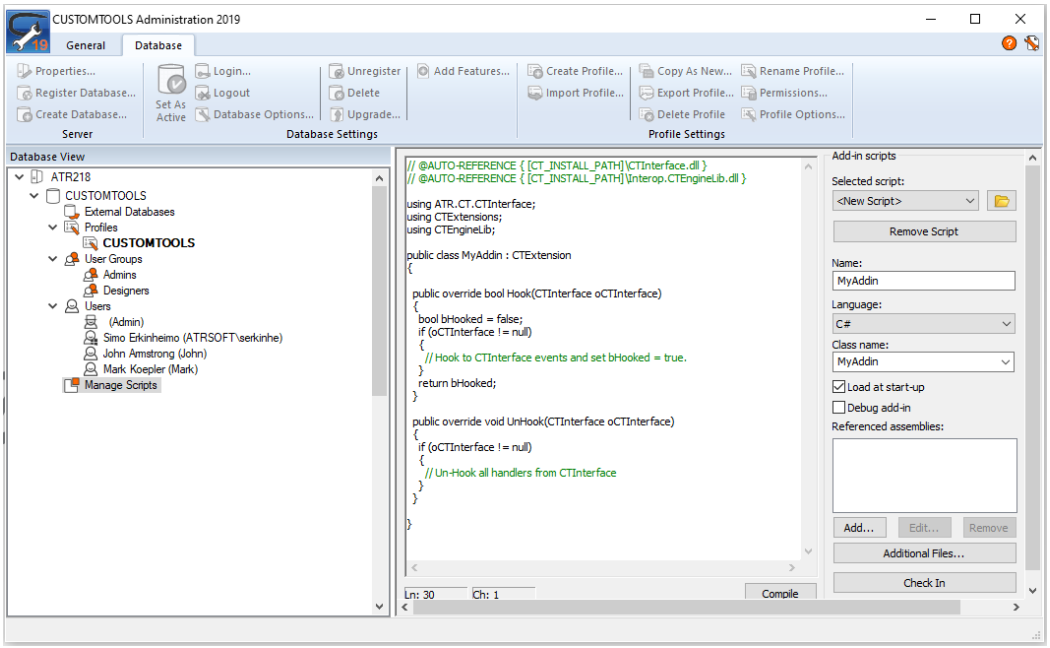


Figure 3.5.2.1: Script management and a minimal example script in CUSTOMTOOLS Administration.

4 ERP Integration with CUSTOMTOOLS

This chapter starts by examining the 11 common requirements (Section 2.6.) and how they affect the integration architecture. The integration base will then be generalized to cover as much of the common logic as possible as well as to make utilizing it for integrations very simple and efficient. The result of this chapter will be an abstract CUSTOMTOOLS Script Add-in (Section 3.5.) focusing on easy-of-implementation and compile time safety. The ultimate goal is to have that abstract base implementation included in source code of future CUSTOMTOOLS releases. The script -add-in is initialized in the beginning of this chapter, and its architecture gets extended while the requirements are examined as the chapter advances. Class name for the main extension is initially chosen to be **GenericSolution** in namespace **ERPIntegration** (Figure 4.1). The solution references classes and namespaces from *CTExtensions* -namespace as well as events from *ATR.CT.CTInterface.CTInterface* -class but full namespace declarations are dropped out to save space.

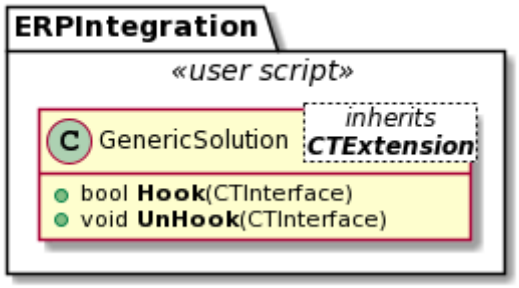


Figure 4.1:Initial extension. Hook and UnHook must be implemented as they are abstract in CTExtension base class.

4.1 Architecture from the requirements

Creating items (Requirement R1) and BOMs (Requirement R5) in batch to target system while modifying the dynamic BOM structure on-the-fly (Requirement R6), and showing a visualized representation (Requirement R9) of the task can be all done with CT Export (Section 3.4.) combined with an Export Type Extension -script. Also, CT Batch File Conversion Rules can be attached to CT Export -profiles which allows generating manufacturing documents to target systems at the same time (Requirement R10). Therefore, these five requirements (R1, R5, R6, R9, and R10) can be grouped into a single partial solution: CUSTOMTOOLS Export with Type Extension -script. For architectural clarity, it is recommended to separate the actual export script handler from the Type Extension as shown in Figure 4.1.1.

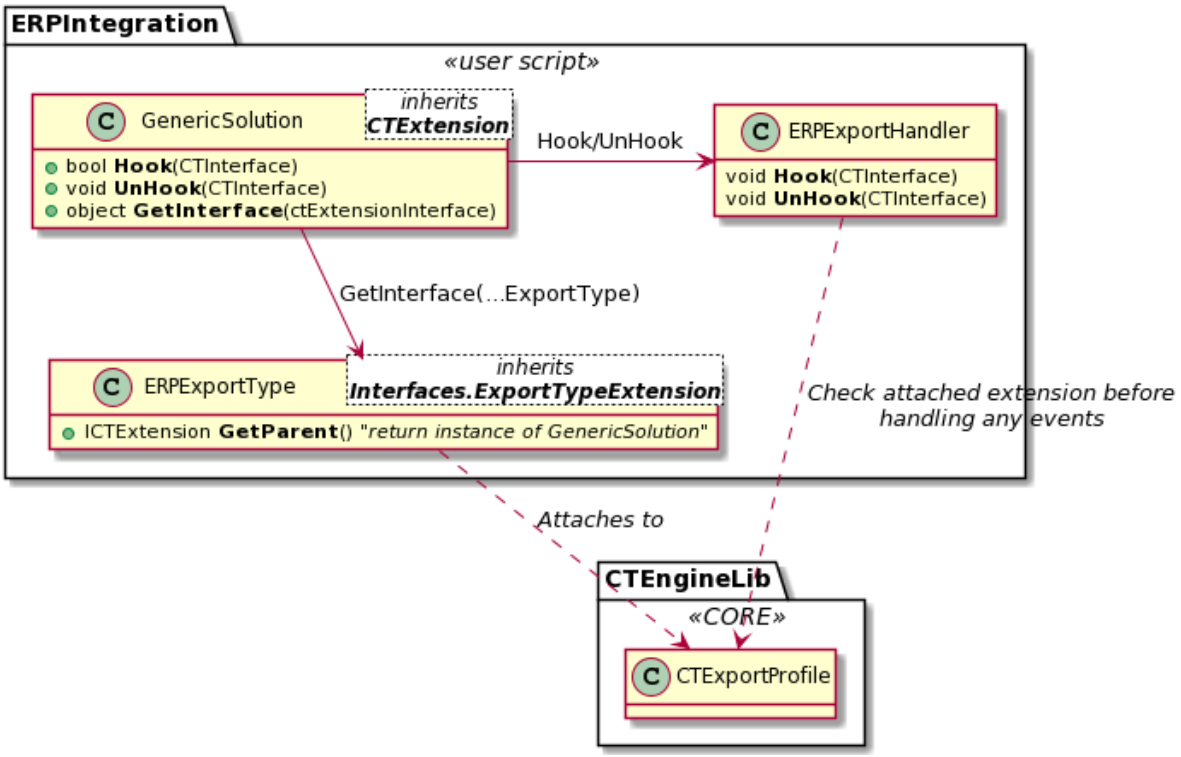


Figure 4.1.1: Architecture of a script add-in with Export Type Extension capability. All extension capabilities must implement GetParent() that returns the instance of the main extension.

A data exchange value field (Requirement R4), and BOM (Requirement R7) ownership and update rules could in theory be hardcoded into Export Type Extension -script. However, both of them should also be configurable as they are usually a matter of preference or highly dependent on the target system. Both requirements (R4 and R7) can be grouped under partial solution of *configurability* consisting of serializable settings objects as well as ObjectEditGuard type of extension capabilities. For the instances of these classes to behave in an expected way, their internal linkage must be well understood. Figure 4.1.2 describes four more classes needed for Export configurability as well as the core objects linking them together. Figure 4.1.3 adds possibility for CT Profile level settings page and Figure 4.1.4 shows added user specific login settings but with some other architectural entities removed for clarification.

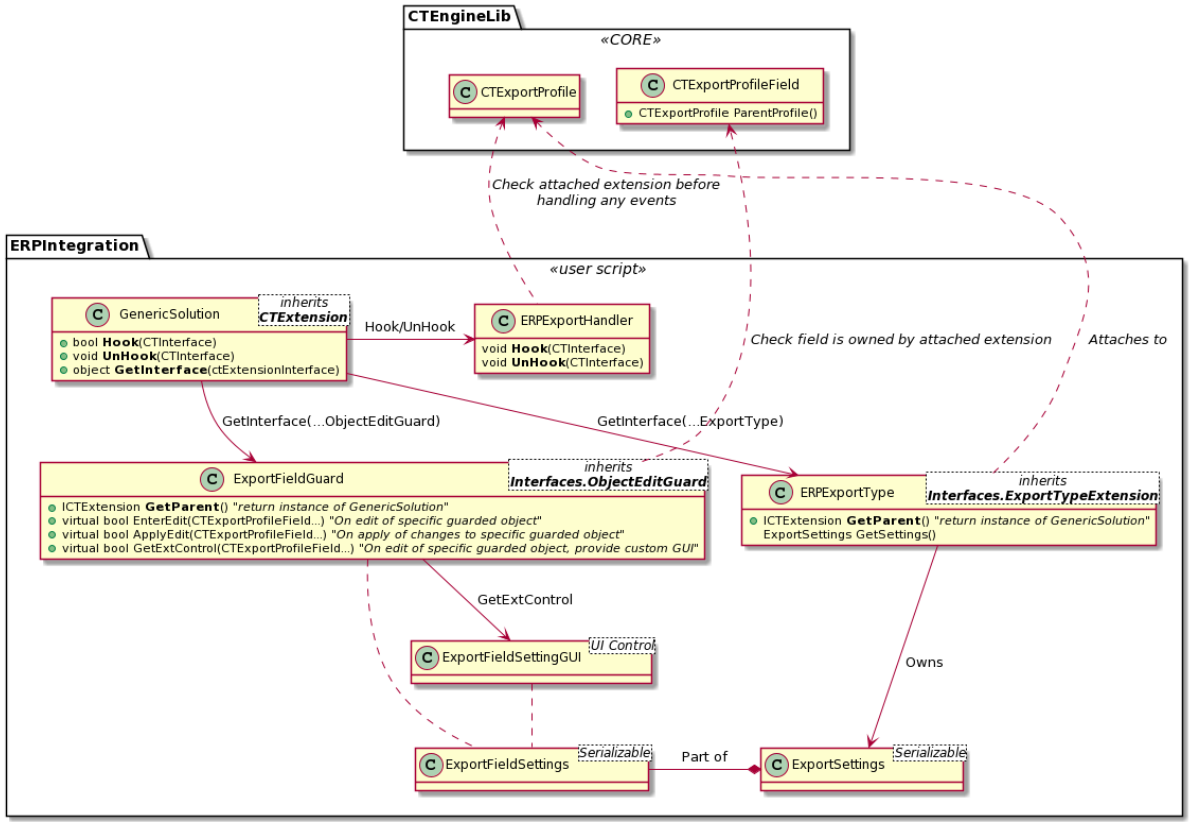


Figure 4.1.2: Export Settings, including field specific settings, are owned by the ERPExportType extension, but the same setting instances must be accessible also via ExportFieldGuard extension for editing as well as in various events at ERPExportHandler where the settings affect the export procedure.

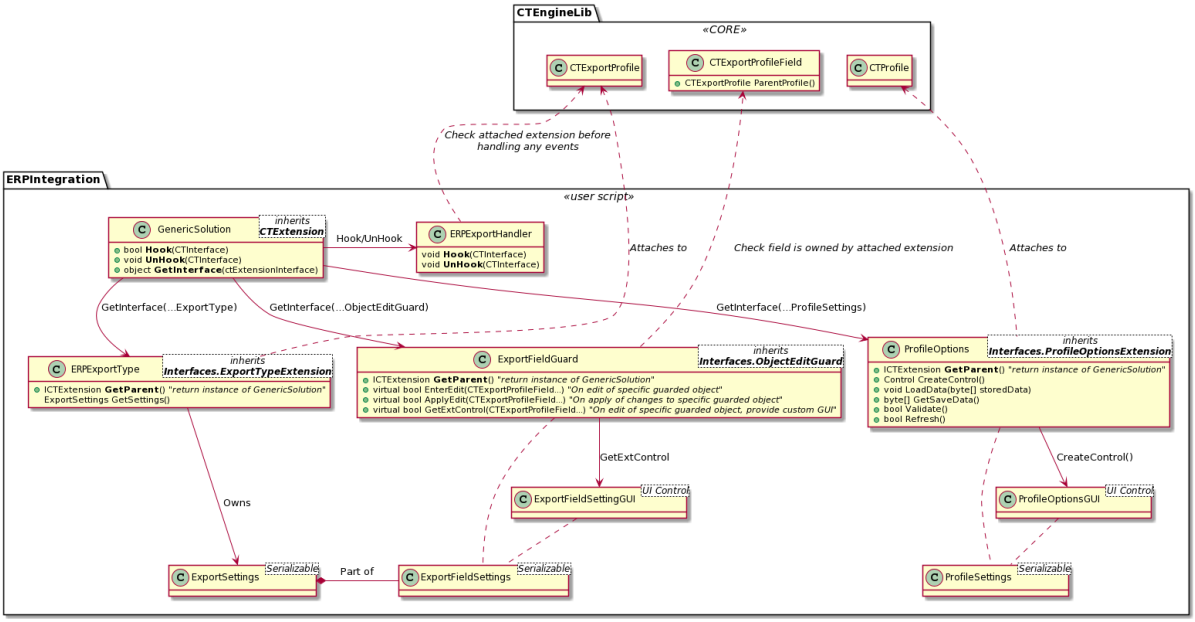


Figure 4.1.3: ProfileOptions extension is useful when the integration has some general settings that are not bound to a single export profile or a single user. Current CT Profile and its stored ProfileSettings data is accessible via CTInterface during the events.

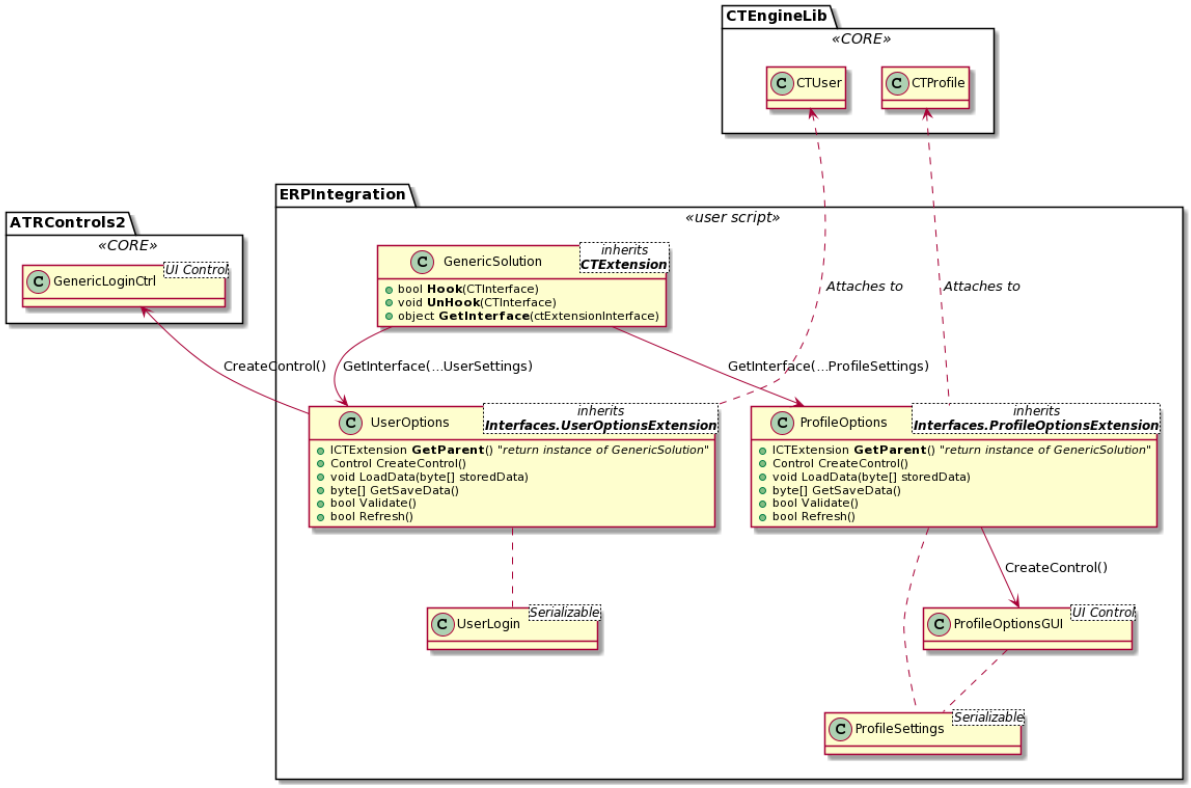


Figure 4.1.4: UserOptions are very often just user specific login credentials. For that, GenericLoginCtrl from the core assembly ATRControls2.dll can be configured to show various fields. Benefit for using this generic control is in its localizations and common look & feel.

Very often the target ERP system is not something directly usable as MS SQL Linked Server and therefore not possible to configure as CT Search Group (Section 3.3) to satisfy Requirement R3 *"Possibility to map an existing ERP item with a model"*. However, CT Search Groups support custom handlers i.e., an extension can provide a result set for given search terms by impersonating a linked server connection. This is done by handling *CTInteface.OnExecuteSearchGroupSearch* -event and by investigating that the search group is indeed "owned" by the integration. In addition, handling *CTInteface.OnCustomListGetColumns* to provide available target system fields will make CT Property mapping to target fields much more user friendly. From an architectural point-of-view, these events should be handled by a separate class of the extension, *SearchGroupHandler*. Figure 4.1.5 has the complete architecture.

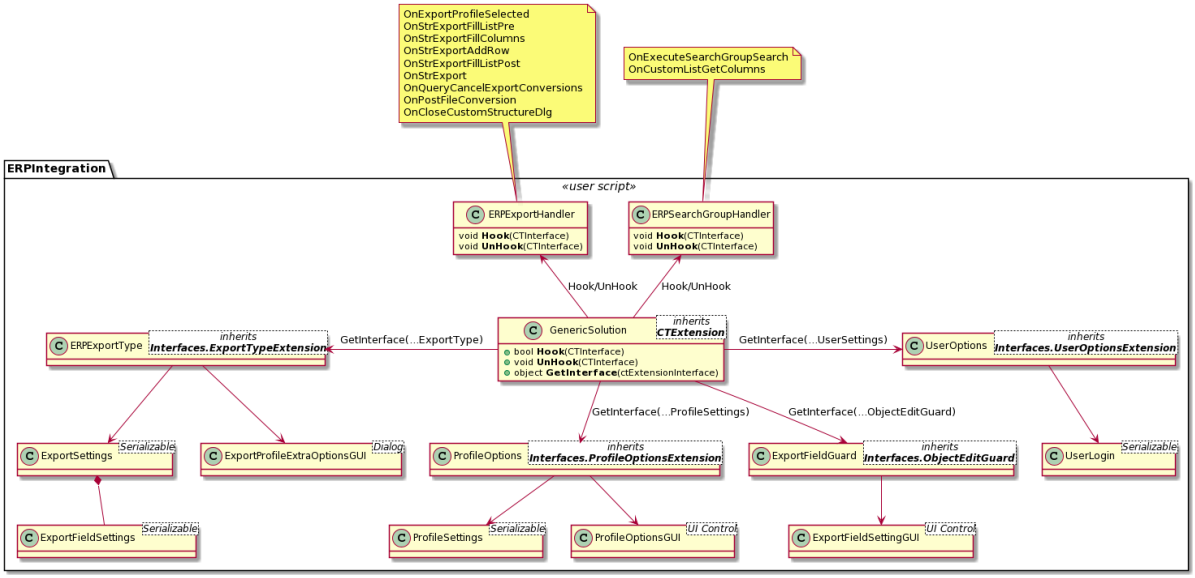


Figure 4.1.5: The complete high level ERP Integration architecture

4.2 Generalization

As the target is to provide a simple, reusable base implementation, the generalization must be simple yet highly configurable. All storable settings, whether they are Export Profile Field, Export Profile, Profile or User level settings, must be serializable and have a graphical control counterpart. Let us have abstract types *SettingsObject, ControlAdapter and ControlAdapter<T>* (Figure 4.2.1).

ControlAdapter is a derivation of *System.Windows.Forms.Control*, and the base class for the GUI counterpart of the user implemented settings object. It is constructed with *ICTExtension* which is the parent extension type capable of providing all possible data of the extension for the control. The constructor has protected level visibility just to make it slightly harder for users to accidentally derive from this type as that should not be done. This class utilizes generics in its *LoadFrom* and *SaveTo* procedures to provide *SettingsObject* type agnosticism for its direct callers on the core level.

ControlAdapter<T> derives from *ControlAdapter* and its main purpose is to provide strong typing between the user implemented setting object and its GUI counterpart using generics. It does it by routing the base class *LoadFrom* and *SaveTo* calls in its implementation to their strongly typed abstraction counterparts. Internally this means type casting the agnostic type to the templated one, which can be a major issue if the architecture is misused. Later the core architecture will evolve to a phase where a common base class will share the same strong types among all entities.

SettingsObject is the abstract base class for all user defined settings. It too holds an instance of the main extensions for possible broader data retrieval cases, and is *Init*ialized always immediately after

instantiation. As required by the extension data storage methods, it serializes to and from byte array, but it also forces its derived class to serialize/deserialize its content directly to/from *System.IO.Stream*. Coupled with *StreamExtensions* (Figure 4.2.2) to push/pop common types to/from stream, serialization in derived classes becomes a very developer friendly operation.

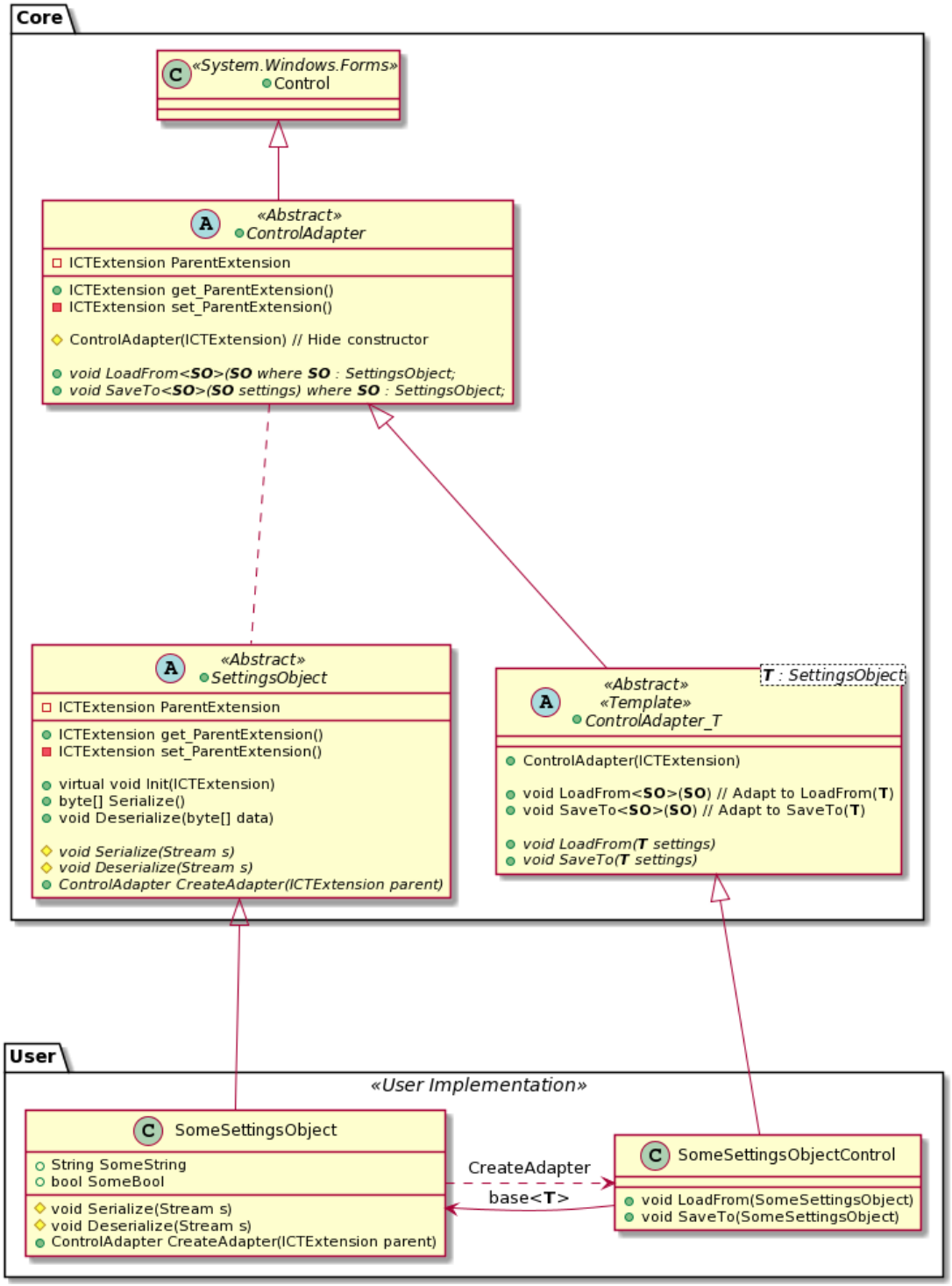


Figure 4.2.1: Architecture for user implemented settings and its GUI counterpart

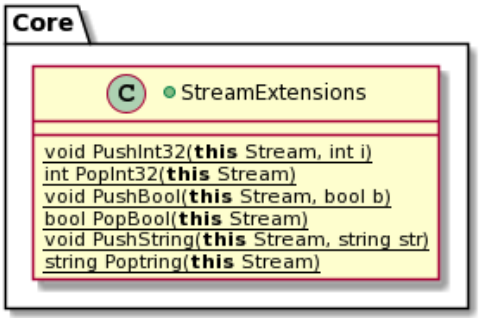


Figure 4.2.2: StreamExtensions provide syntactic sugar for serialization.

4.2.1 Export Settings

Minimal export settings for all ERP integrations are the field mappings, and a single export field setting object is bound to a field by its name. Export Field Settings are not separately serialized but they must be part of the Export Settings object and serialize with it. On the other hand, Export Profile Settings is not always required to provide GUI with extra options, so the minimal Export Settings object is simply a SettingsObject derivation with generic field type hosted in Dictionary from field name to instance of that templated field type. Let us call it *ExportSettingsBase<FT>* (Figure 4.2.1.1).

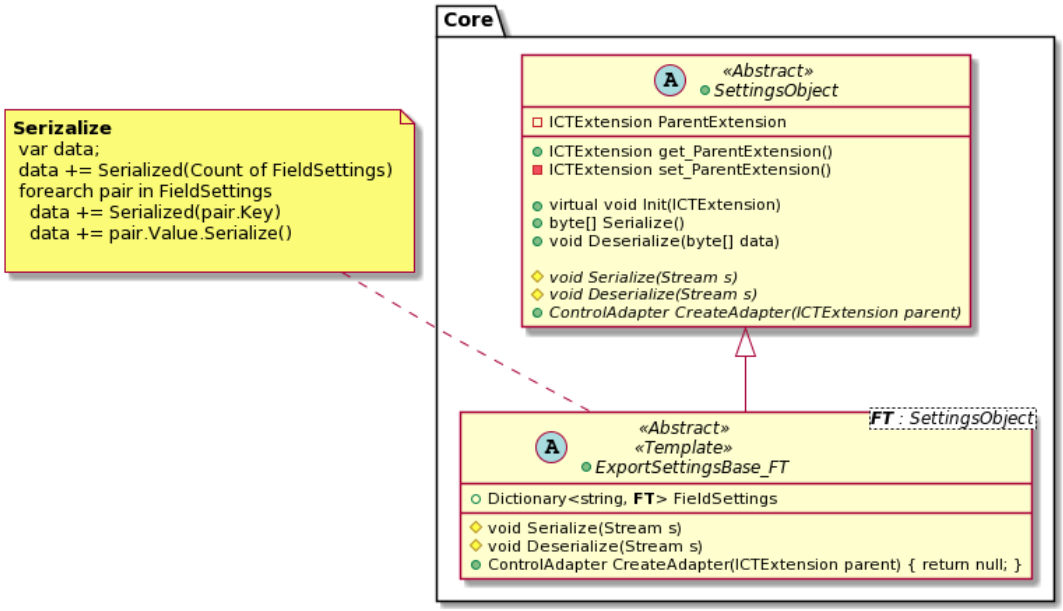


Figure 4.2.1.1: *ExportSettingsBase<FT>* return null for its own Control but hosts SettingsObject for all of its fields.

4.2.2 Base Extension with complete settings generalization

The base extension is an abstract class derived from CTBuiltInExtension, and has generic typing for all supported settings types: Export Field Settings, Export Profile Settings, Profile Settings and User Settings. To have all these capabilities interacting with CT core, implementations for *Interfaces.ExportTypeExtension, Interfaces.ObjectEditGuard, Interfaces.ProfileOptionsExtension* and *Interfaces.UserOptionsExtension* are also done and attached to the base extension. The corresponding implementations are *ExportTypeExt, ExportFieldGuard, ProfileOptions* and *UserOptions*.

Some refactoring is now applied and the whole new core addition is decided to be located under **CTExtension.ExportCore** -namespace and the base extension is renamed to **ExportBase**. CTExtensions is a core product namespace, meaning the introduced generalization will now be available in CUSTOMTOOLS' later releases. Bringing it all together we now have all possible integration specific Export Profile, Export Profile Field, Profile and User settings fully implemented yet all the object types and GUI controls fully configurable; i.e. the base implementation now fulfills all the configurability requirements to the furthest possible extent. Complete architecture so far is described in Figure 4.2.2.1.

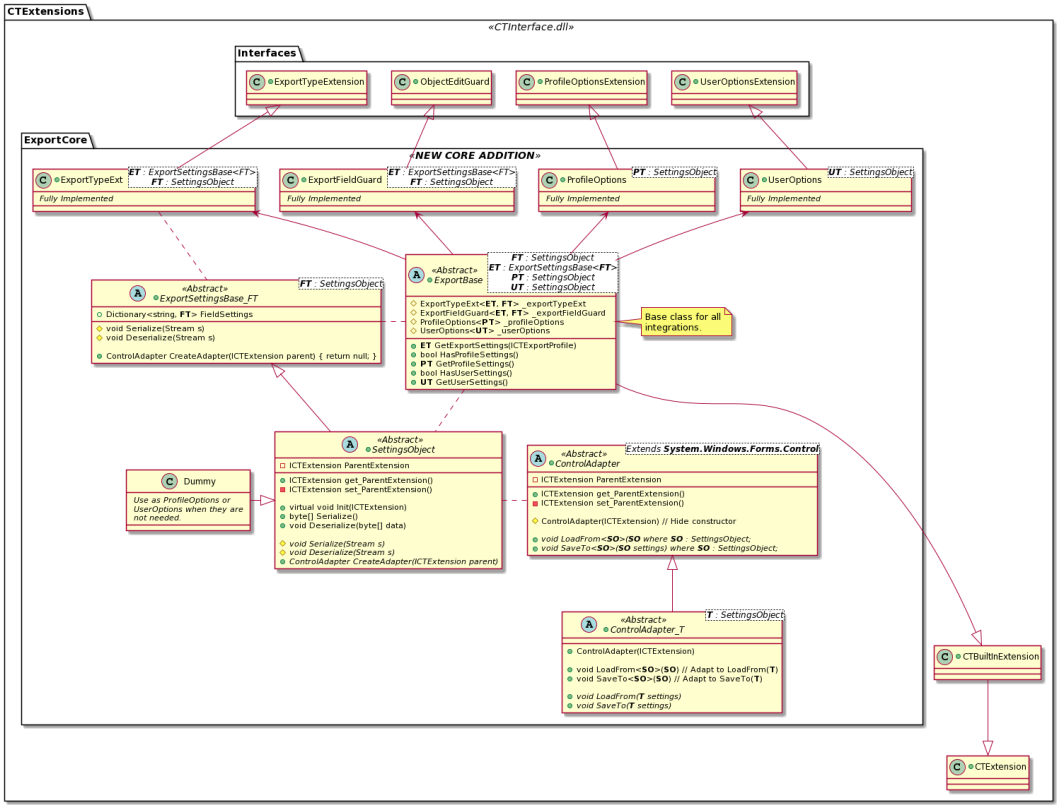


Figure 4.2.2.1: The complete architecture so far fulfils all the configurability requirements to the furthest possible extent

4.2.3 Event Extensions

The base implementation should also have some framehanding for custom sourced Lookup Lists (Section 3.2), custom sourced Search Groups (Section 3.3.) and most importantly, for item/bom Export (Section 3.4). While the more visual and core integrated *CTExtension Interfaces API* is used for configurability and user interaction, the *CTInterface Events API* provides simple event-based data manipulation and handling capabilities.

As **ExportBase** derives from CTExtension, it has to implement Hook(CTInterface) and UnHook(CTInterface) that are used for initializing the event based handling. Therefore, it makes sense to create a common base class for event extensions of ExportBase so that different, even more extended types can then be implemented and used as is with the base class. Generics and routing implementation can again be used to provide strong typing for the user implementation. ExportBase will then have a new abstract *GetEventExtensions()* method in which the user implementation should return all the event handlers it needs. EventExtension base class and connectivity with ExportBase is shown in Figure 4.2.3.1.

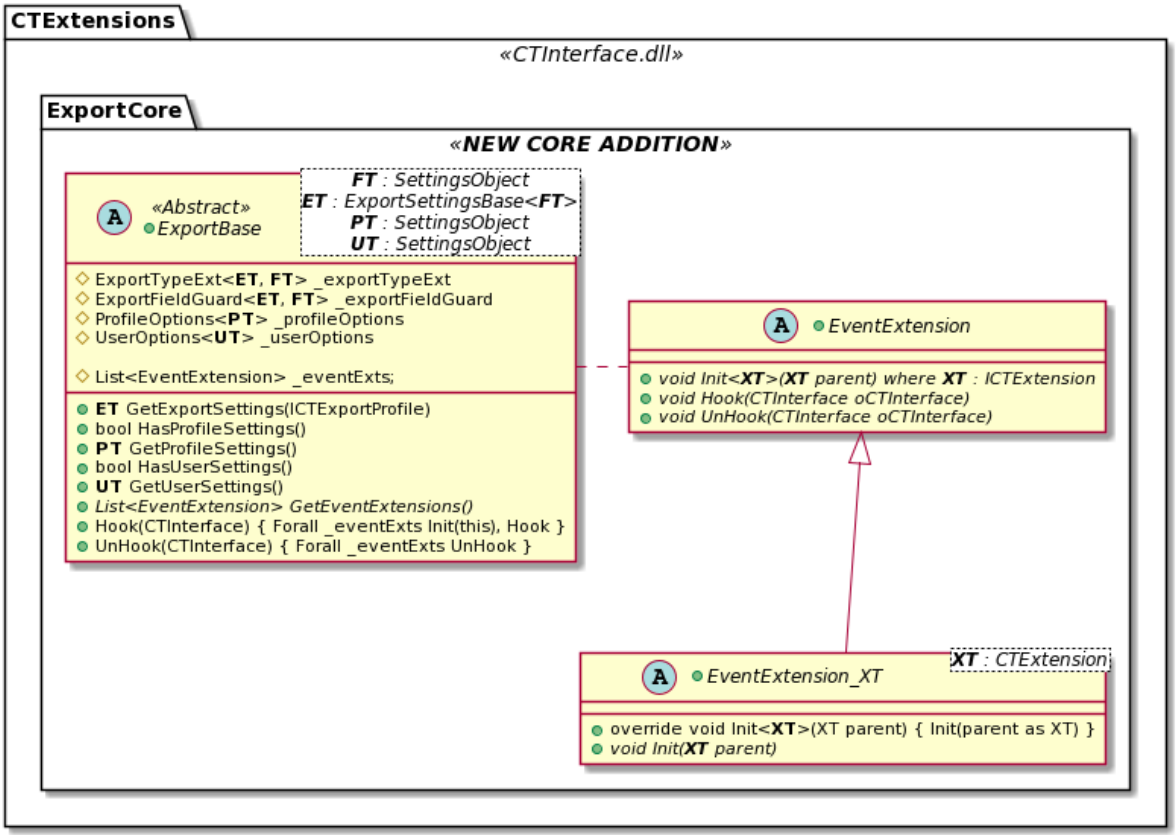


Figure 4.2.3.1: EventExtension architecture

4.3 Data model configuration

Being able to export the same items multiple times (Requirement R2), as well as getting it all to work with existing models (Requirement R8) are all about configuring the CT Properties (Section 3.1.) to support the existing Custom Property data model and the requirements of the target ERP system. Having the data available is the key requirement for everything else. This is also in direct relation with being able to map an existing ERP item with a model (Requirement 3) as Search Groups (Section 3.3.) can be configured to pull the data from a 3rd party system into the models' Custom Properties. Very closely related requirement is also the possibility to provide identification by the target ERP system (Requirement R11) but as this data model related requirement does not come without issues, it has to be discussed in its own sub section (Section 4.3.4.). Still, these four requirements (R2, R3, R8, R11) can be grouped under the same partial solution of *data model configuration.*

4.3.1 Target system requirements

Data model configuration should be started with the requirements of the target ERP system; mainly understanding what the valid values from ERP point-of-view in design-to-item field value mapping are. Let us consider for instance *Unit*. Since the target system likely has its own unit-based calculations on resource planning, it obviously expects the units provided to its items by the integration to be known ones. Therefore, the *Unit* in design components must be based on a list provided by the target system. In CUSTOMTOOLS this and all similar choose-from-list -type requirements would be a Property with a Lookup List (Section 3.2.) which content is either manually defined at CT Profile Options, dynamically retrieved with an SQL Query, or dynamically provided by a script add-in.

Another common ERP side data restriction is field value length. In many systems for instance *Description* is limited to a specific number of characters. Therefore, this should already be accounted for in design time Property filling. In CT Profile Options this is a trivial setting in Property Wizard's Additional Options -page (Figure 4.3.1.1).

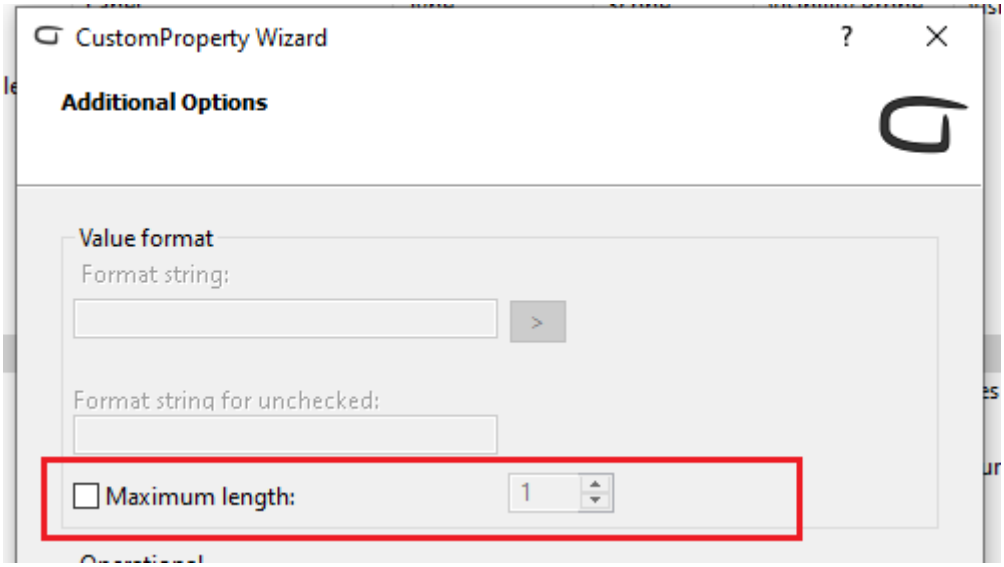


Figure 4.3.1.1: Maximum length -setting at Property Wizard

For some extra protection against values not written to models by CT Properties, the same maximum length can be defined for Export Profiles Fields as well, along with simple but efficient Compulsory field -selection. The latter one prevents the integration from executing if the field has no value at all, tackling the simple target system requirement to provide a non-empty value for a specific field (Figure 4.3.1.2).

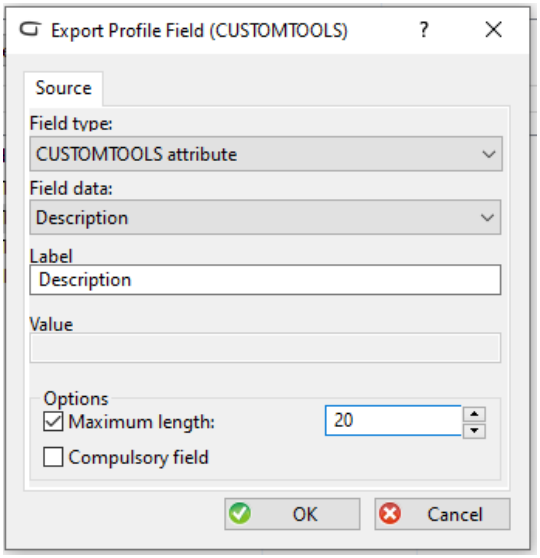


Figure 4.3.1.2: Export Profile Field settings

As a recurring example, let us define some set of fields in Table 4.3.1.3 that an imaginary ERP system could expect.

Table 4.3.1.3: Requirements of an imaginary ERP system

Field Name	**Value requirements / Explanation**
Item No	Freely definable, unique identifier of an Item
Description	String of maximum 20 characters describing the item
Unit	One of the following unit-values, user-friendly value in parenthesis: * KG (Kilograms) * M (Metres) * PCS (Pieces) * L (Litres)
Type	One of the following item type indicators defining its behavior in ERP system, user-friendly value in parenthesis: * 1 (Manufactured) * 2 (Purchased) * 3 (Assembled) * 4 (Product)
Weight	Decimal value, weight of the item
BOM	Array of strings containing valid *Item No*s already present in the ERP system or within the current transaction. Null or empty for Items that don't have BOMs.
BOMQtys	Array of integers containing Quantities of Items defined in BOM -field's array. Size of this array must match with BOM -field's array size.

4.3.2 User-friendly values in Lookup Lists

If the target ERP system expects non-user-friendly values for a field, expecting the designer to select those values in design time at Properties is not a good idea. Instead, a Hierarchical -type Lookup List can be used with a Hierarchical Combo -Property, and a script add-in that displays user-friendly values for each combo item while still writing the ERP-required data to the model. The event that needs to be handled to achieve this is *OnLookupListFill* and it is only supported for Hierarchical Combo -properties. It is however trivial to convert also *Simple* and *Key-Value* -lists to Hierarchical ones.

For example, let us examine the example target ERP system's requirements (Table 4.3.1.3) regarding the *Type* -field that determines the item behavior in the ERP system. ERP required values for that field are 1, 2, 3 and 4 and corresponding meanings are 1=Manufactured, 2=Purchased, 3=Assembled and 4=Product. The numerical values are the real data that needs to be stored to the model at design time so that it can then be exported in batch to the target ERP system. But the CAD designer is interested in choosing the meaning over some meaningless numerical data value. This type of list can be represented as in Table 4.3.2.1 with Hierarchical Lookup List -type, but no actual hierarchy is needed.

Table 4.3.2.1: Lookup List with values and descriptions

Value	**Key1**
1	Manufactured
2	Purchased
3	Assembled
4	Product

Then while handling *OnLookupListFill* -event for a Hierarchical Combo -Property, the *DisplayKey* -property in event arguments can be set from 0 to 1 to populate the drop list with Key1 -values. Keeping the *MainKey* as 0 uses the value field as the actual data to store. This method can also be used for user specific translations of LookupList values.

4.3.3 Design to Item -mapping

A valid 1-to-1 mapping must be determined between a Custom Property (or set of Custom Properties) and target ERP field (or set of fields). This is the basis for being able to export complex SOLIDWORKS structures as the same components easily are referenced in multiple different contexts. Therefore, in the SOLIDWORKS context, Requirement R2 to be able to export structures in batches multiple times is as important as just being able to create items to the target system once (Requirement R1). Being able to export something twice means that the corresponding item for the CAD design must be pre-mapped already before any export procedure is executed and some sort of value comparison logic applied. This also strengthens the Export -dialog solution for Requirement 9 that is about seeing a clear visualization of what kind of operation the integration is going to perform (Figure 4.3.3.1). This pre-export phase is also crucial for being able to have field/BOM specific update rules (Requirements R4 and R7).

The mapping is usually handled so that at the CAD -site some sort of unique identifier per itemized component is generated and this ID is then exported to some specific field to target the ERP system. This yields a requirement to the ERP system: the field used for mapping must stay unique. Unique number generation using CUSTOMTOOLS is however very simple and versatile using CT Sequences with CT Properties. It is also possible to form a complex combination of Properties to use as item identifier in design to item -mapping. Whenever possible, the item identification should be provided by CUSTOMTOOLS for simplicity. It is fairly common to use an item ID that does not resemble anything that would otherwise be created in ERP, just to have clear separation between integration -handled items and other items. However, some systems require that the item identification is generated by the target ERP system. This is possible but not trivial as will be discussed in Section 4.3.4.

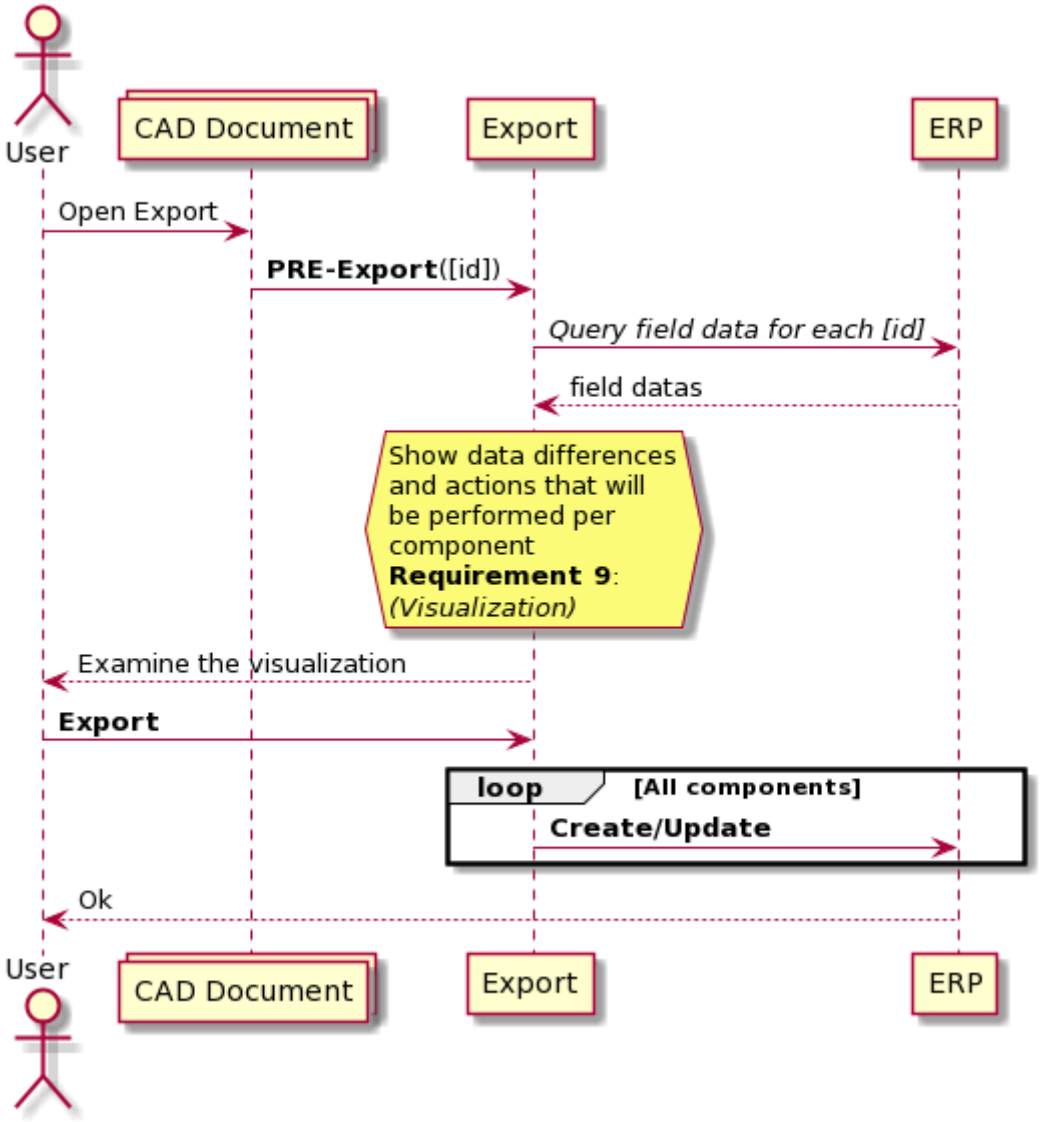


Figure 4.3.3.1: Pre-export -phase retrieves existing values from the ERP so that the value comparison can be performed and visualized.

4.3.4 Item identification by target system

Usually, EPR items have an ID that can be used for unique mapping but on some occasions the ID must be generated by the ERP system itself. This is problematic from CAD point-of-view as it would require write privileges to design files at ERP Export -phase, while this is usually the point that the design is already locked from all modifications by some sort of PDM system (Figure 4.3.4.1).

One solution for this is to instruct CAD designers to use the Export in a point that still allows write access to design documents, or to have a separate integration module that handles file write within the

used PDM system. As far as the author knows, these are the only possible ways to solve the case if it is expected that the design documents do not generally have the item identifier at Export.

On the other hand, the ID retrieval can be thought of as a design time task (Figure 4.3.4.2). This is the most natural use case for new designs but has an obvious downside that legacy designs will not have the value written. So, the overall export must anyway be equipped to handle the already described issues with them. For this, the Compulsory Field at Export Profile Field -settings (Figure 4.3.1.2) is a great addition for the ID -field. This will prevent execution of Export for as long as even one of the components is missing value from the field. This guides the user to go back to Properties to retrieve a new ID from the target system, and as this is an obvious change to the document, it is clear for the designer that the document must be reserved from the PDM system for this task (Figure 4.3.4.3).

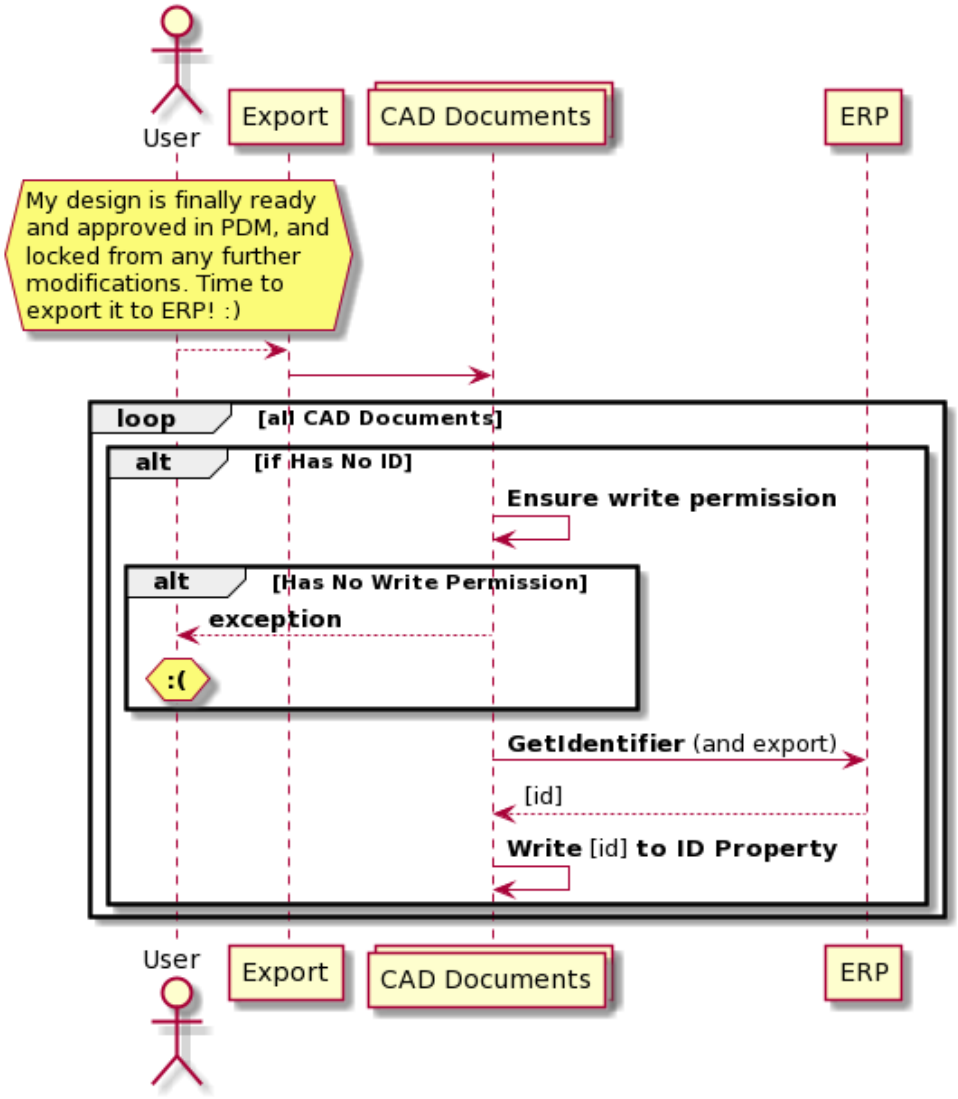


Figure 4.3.4.1: Ensuring write permissions is crucial but many times the process is executed on a phase where no modifications are expected anymore.

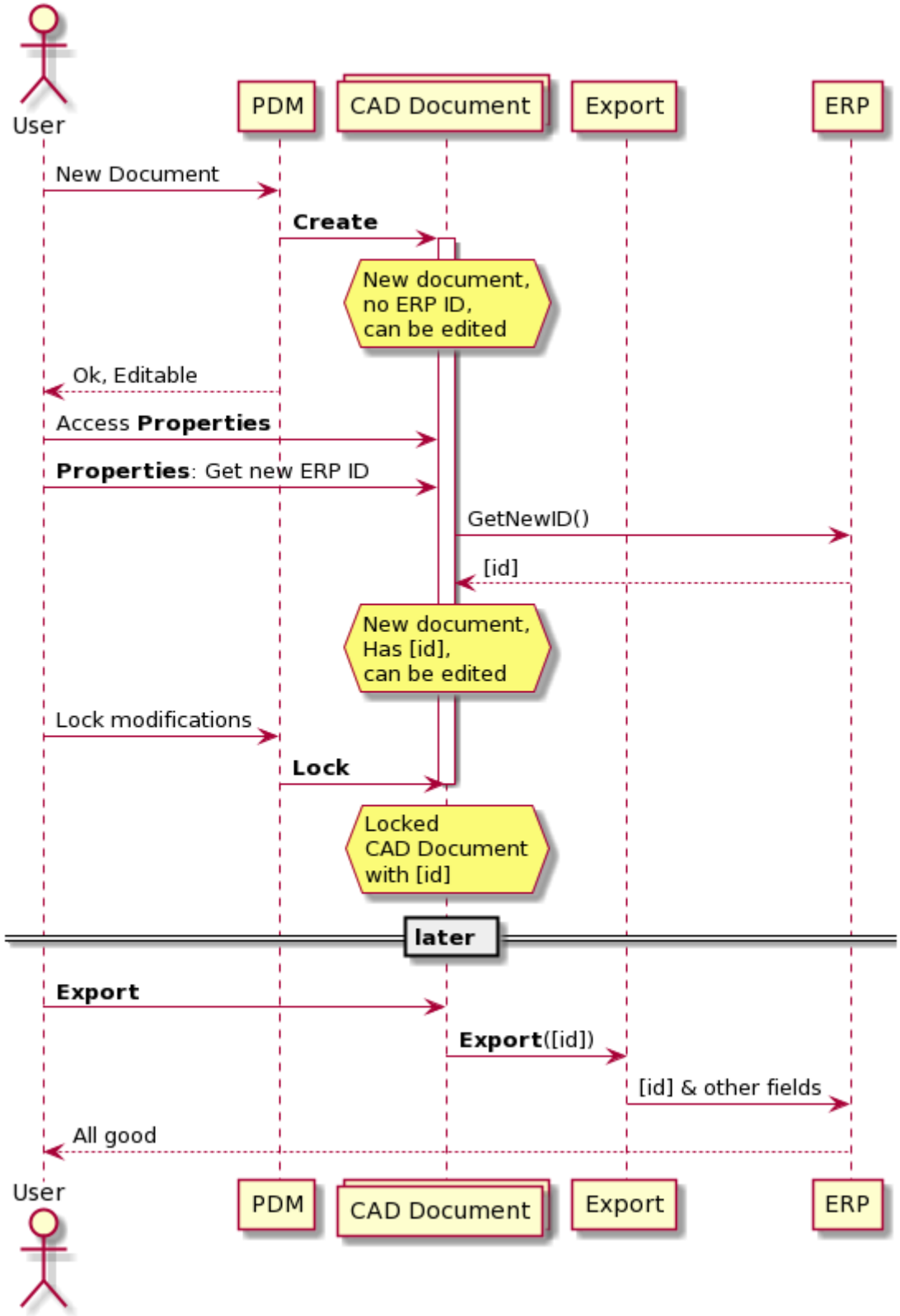


Figure 4.3.4.2: Design time ERP Item ID mapping with Properties

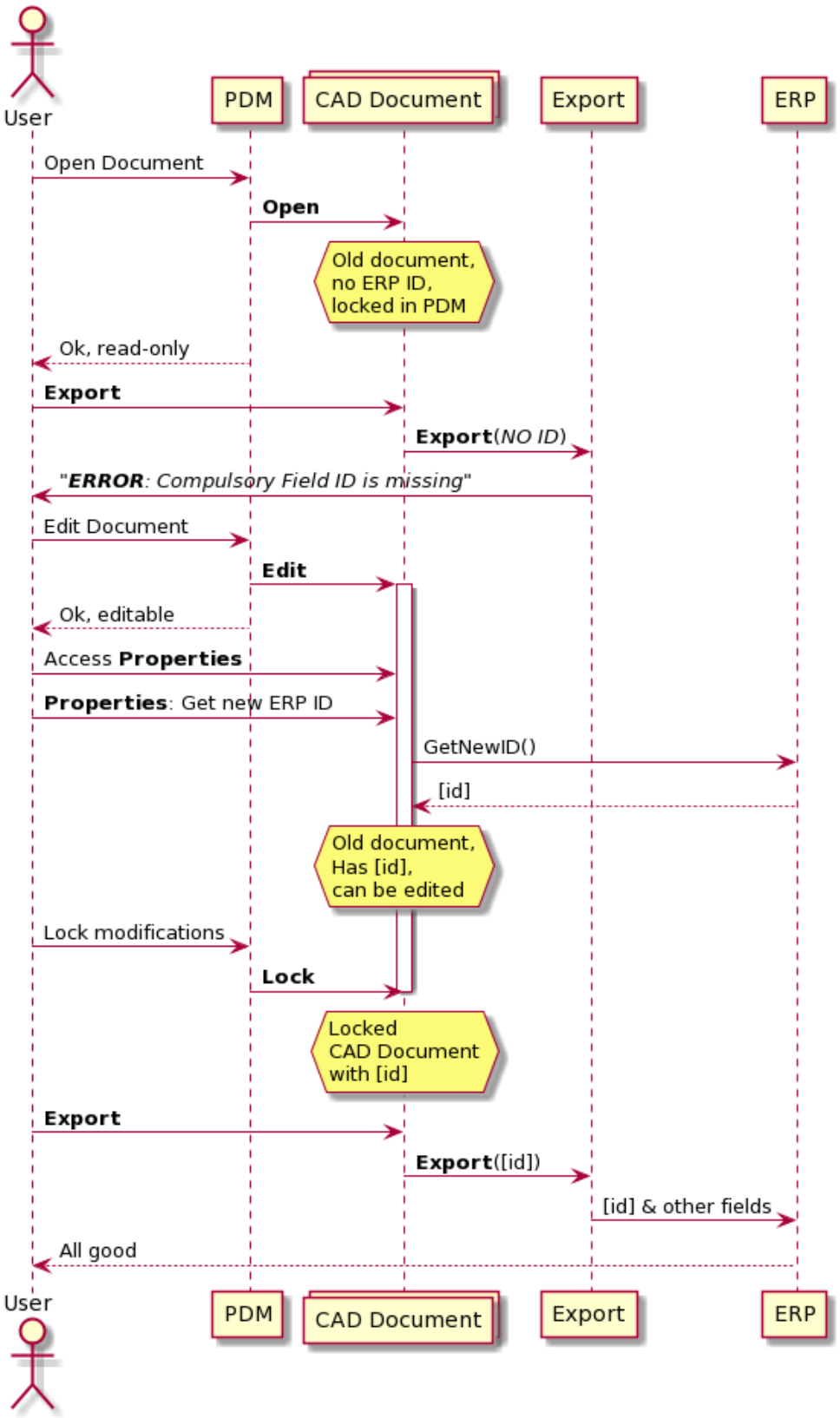


Figure 4.3.4.3: ID retrieval process for legacy CAD Documents

4.3.5 Design time ERP item mapping

One very normal use case that is surprisingly often forgotten from ERP integrations is the possibility to create a CAD design for an existing ERP item. For example, purchased components are very often already existing in the ERP system but if the integration is only able to create new items with new identifiers, then there is no possibility to take those already existing ERP Items into use for new or existing designs.

This is solved with CUSTOMTOOLS Search Groups (Section 3.3.) that are able to query anything that can be added as a linked server to MS SQL Server and map the result set directly to CT Properties already at design time. The Search Groups must then contain all the same properties that are also in the Export Profile, because otherwise the Export would lack field information of those existing ERP items and possibly even resulting in data loss at ERP site.

4.3.6 Configuring the CT Profile

Looking back at the example ERP requirements (Table 4.3.1.3), we determine that following Properties are needed:

- ***ERP Item Number***, a text field -Property that has CT Sequence attached to it to provide unique identifiers across the design environment. In the Export Profile this property needs to be linked via its Custom Property attribute to a Export Profile Field (Section 3.4.2) named **Item No**, which should also be set as a Compulsory Field. The Property itself needs to be linked to the defined ERP Search Groups' **Item No**. To support legacy models as much as possible and configuration specific items, the *Initial Configuration* -setting of this property must be set to *Document Properties and Active Configuration*.
    - This arrangement is also suitable to support Item identification by target system (Section 4.3.4) as the CT Sequence provider can be overridden with a *SequenceExtension* to provide identification from the ERP.
- ***ERP Description***, an Editbox -Property with length limitation of 20 characters. Liked via its Custom Property attribute to an Export Profile Field named **Description**, which should also have the length limitation of 20 characters. Search Group linking to **Description** -field as well. Initial Configuration -setting depends on if the designers are and/or have been using configurations to produce different kinds of items. Good rule of thumb is that if the description always contains measurement information or anything else configurable, then it

should be configuration specific i.e., *Active Configuration* or *Document Properties and Active Configuration*. The latter one is generally faster in design time property filling as the document property value is inherited for configurations that do not have their own value. But as such this may also result in retrieving non-examined descriptions of new configurations at item export. Some prefer these fields to be empty unless specifically defined by the designer, to which case the first option suits better. If the designs are never configured so that different configurations would have different descriptions, then Document Properties is the correct option to select. In addition, to query items of the Search Group, one of the properties must have a *Button Function* **GetDatabaseItem**. This property is a good choice for it as the *ERP Item Number* -Property already has a *Button Function* defined.

- ***ERP Unit***. Depending on if the unit values of the target system are user-friendly, this could simply be a ComboBox with a Simple -type Lookup List. If they are not, then this should be a Hierarchical Combo -property with a Hierarchical -type Lookup List to allow displaying user-friendly values in Properties. Its Custom Property attribute should be linked to an Export Profile Field named **Unit**. It is very common that the unit-like field is required by the target system so in most cases this field should also be set as a Compulsory Field. The Search Group should be linked to the corresponding **Unit** -field and from this case it becomes obvious why the "EPR required unit value" must be resolved dynamically at runtime (Section 4.3.2.) instead of for example at Export: It would not otherwise be possible to query the value from existing items and store them to properties as returned by the ERP system. Initial Configuration of Unit is generally *Document Properties* as configuring designs should not affect what the design actually is and therefore also in which unit its quantity should be represented.

    - Content of the list would be provided by the script extension, or by any other means discussed at Section 3.2.

- ***ERP Item Type***. This is technically exactly the same case as the ERP Unit with Hierarchical Combo. Export Profile Field is **Type** and it is also compulsory. The Search Group field is also **Type**. Initial Configuration is *Document Properties* in this example case. It is however possible in some cases that some configurations of a design are purchased while others are manufactured. In these cases, *Document Properties and Active Configuration* is the best choice.

- ***Mass***. This is a property that goes one way only, from design to ERP. Mass is calculated by SOLIDWORKS based on active/referenced configurations but the actual property holding the mass does not have to be configuration specific. It is a special value that is evaluated on request so when the document is open, the mass is usually retrieved correctly. However, to

also support cases when the documents are not open, it is recommended to use configuration specific property to have last known configuration specific mass stored to the design's properties. So as a middle ground to support also old documents *Document Properties and Active Configuration* is a good choice for Initial Configuration. Otherwise, the CT Property type is an Editbox with *Before Function* **GetMass** (Section 3.1.4) and its Export Profile Field is **Weight**. As the weight is always provided by the design, this property is not included to the Search Group.

- The mass value in Export is displayed per component using its document's defined unit system. This can be a problem if the units are not consistent, e.g., some component reports weight in kilos as others in grams or pounds. Usually design companies are using consistent units but if this becomes an issue, it can be solved with the *ExportExtension* using SW API to retrieve used document units and adjust the weight value accordingly. This is however a rare situation so it will not be included in the general solution.

The **BOM** -field required by the target system is handled by the *ExportExtension*. The **BOM type** -setting in the Export Profile however is either *Parts Only* for target systems that only allow single level BOMs or *Intended Assemblies* for multilevel. Configurations must be displayed as separate items as was discussed in Section 3.4.1. If the current design environment has been strictly using Design time BOM modifiers (Section 2.5.3.) then the options to *Always show components of subassemblies, ignoring the "Child component display when used as a subassembly" setting* can be left unchecked. However, it is the author's opinion that this is almost never the case and therefore the BOM/Item filtering should be completely handled by the *ExportExtension* to which the whole unmodified BOMs should be provided by checking this option.

To have Quantities for the **BOMQtys** -field, a new special Export Profile Field must be added with Type *SOLIDWORKS Property* and *Quantity* or *Material Quantity* as data selection. Material Quantity is a new option in CUSTOMTOOLS 2020 SP1 that allows defining the real need of some material instead of the instance count of it in design. The real need is usually the one that ERPs want but it is also usually paired with some raw material information. For example, if a design is a metal tube cut to length of 1m from a 2m bar, then the design's BOM Quantity should be set to 0.5 and raw material to 2m bar as every instance of it consumes only half of that bar. Then, if a design has 10 of these 1m bars, the actual need i.e., *Material Quantity* in the full design is 5 times 2m bar, which is usually also the purchased raw material. On the other hand, some ERPs have their own raw material calculations and in those cases the *Quantity* is the correct choice for data selection. Name of the field has no real role from the integration point-of-view, but *Quantity* is generally good for visualization.

Figures 4.3.6.1 - 4.3.6.4 show the related CT Profile settings defined so far, Figures 4.3.6.5 and 4.3.6.6 show the CT Properties and attached Search Group as they are available for the designer. Figure 4.3.6.7 shows how the data is stored to a model using current profile settings and Figure 4.3.6.8 how the data is collected to Export -dialog.

Properties:

Attribute Name	Label	Type	Required	Before Function	Button Function	Database Search Group	Database Column	Max Length	Lookup List
Properties									
erp_item_no	ERP Item Number	Editbox	✔		GetCode	Target ERP	Item No		
erp_descr	ERP Description	Editbox			GetDatabaseItem	Target ERP	Description	20	
erp_unit	ERP Unit	Combobox	✔			Target ERP	Unit		ERP Units
erp_type	ERP Item Type	Hierarchical Combo	✔			Target ERP	Type		ERP Item Types
mass	Mass	Editbox		GetMass					

Figure 4.3.6.1: Model Properties defined having the example target ERP -system in mind. Attribute names are set so that existing design data can be used as efficiently as possible.

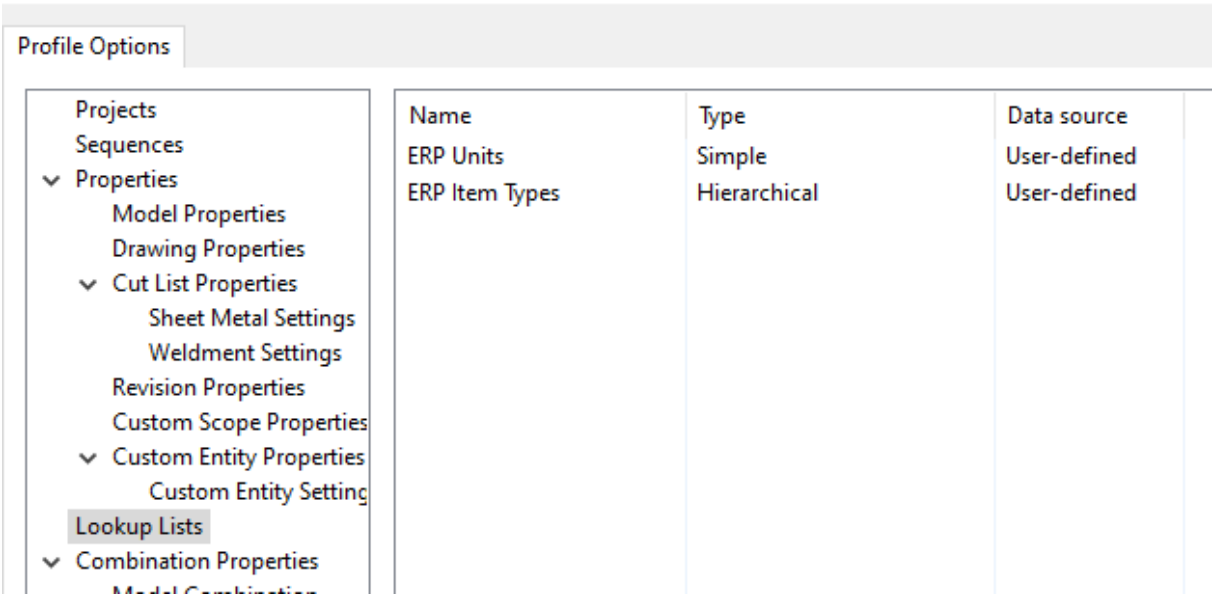


Figure 4.3.6.2: Lookup Lists defined having the example target ERP -system in mind

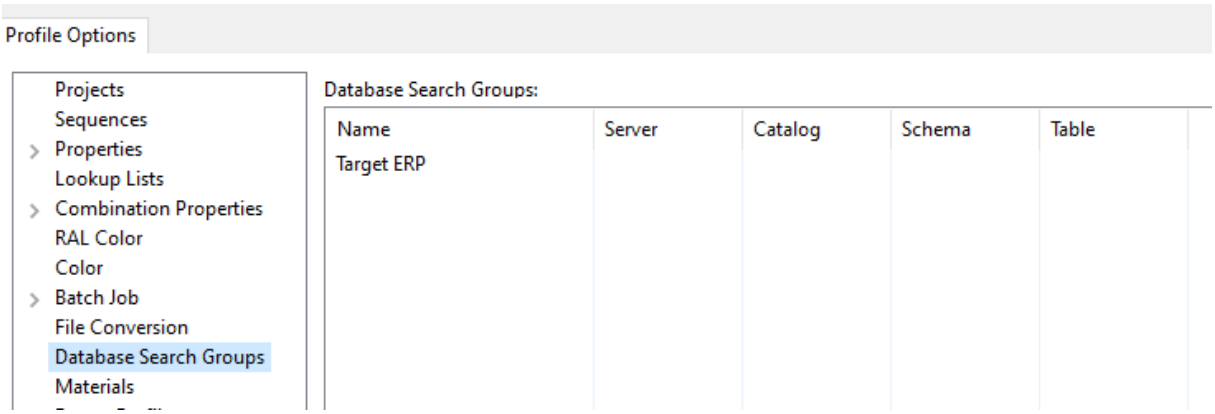


Figure 4.3.6.3: Target ERP defined as queryable Search Group.

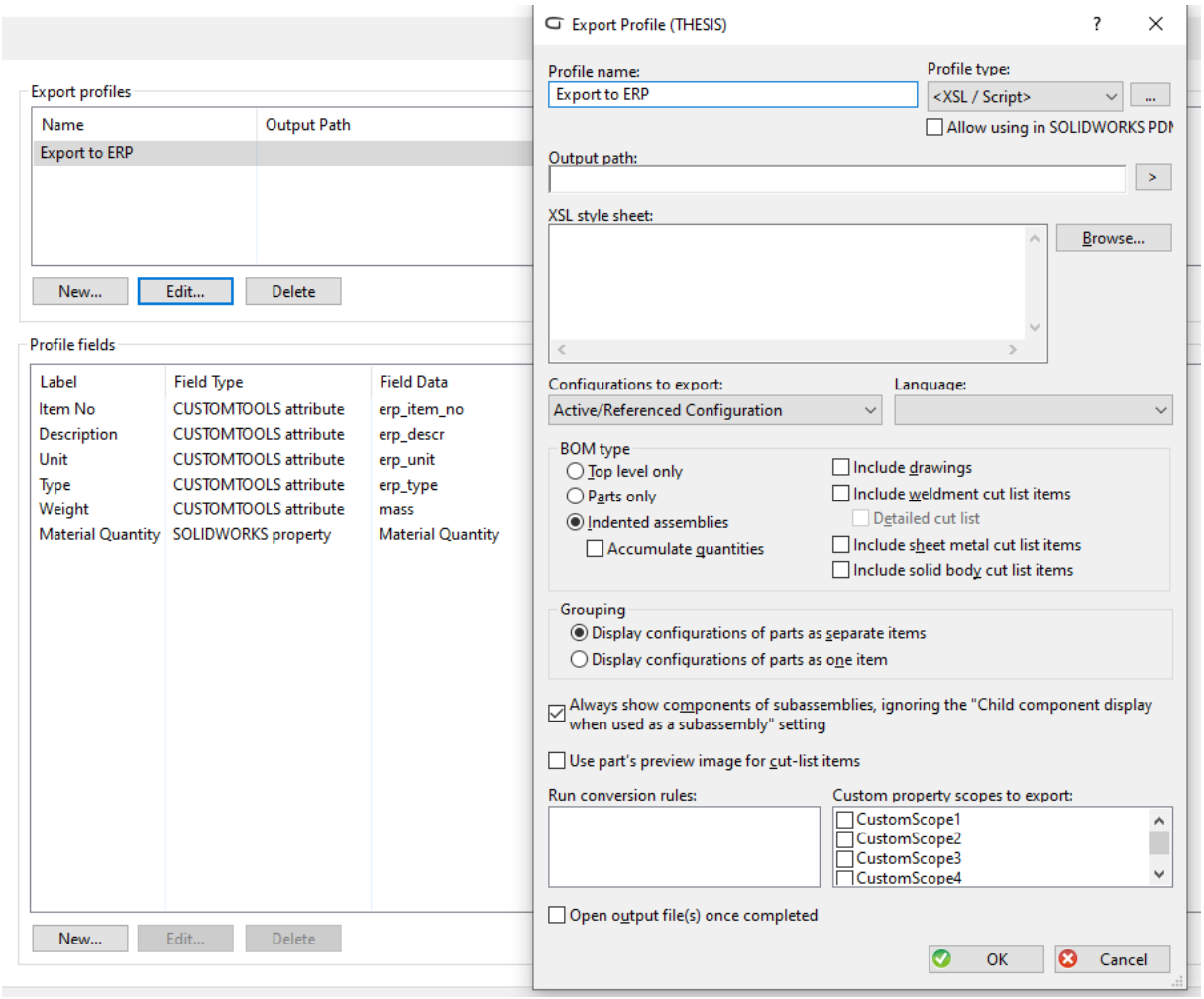


Figure 4.3.6.4: The Export Profile “Export to ERP” and its fields with recommended settings for target ERPs supporting multi level BOMs.

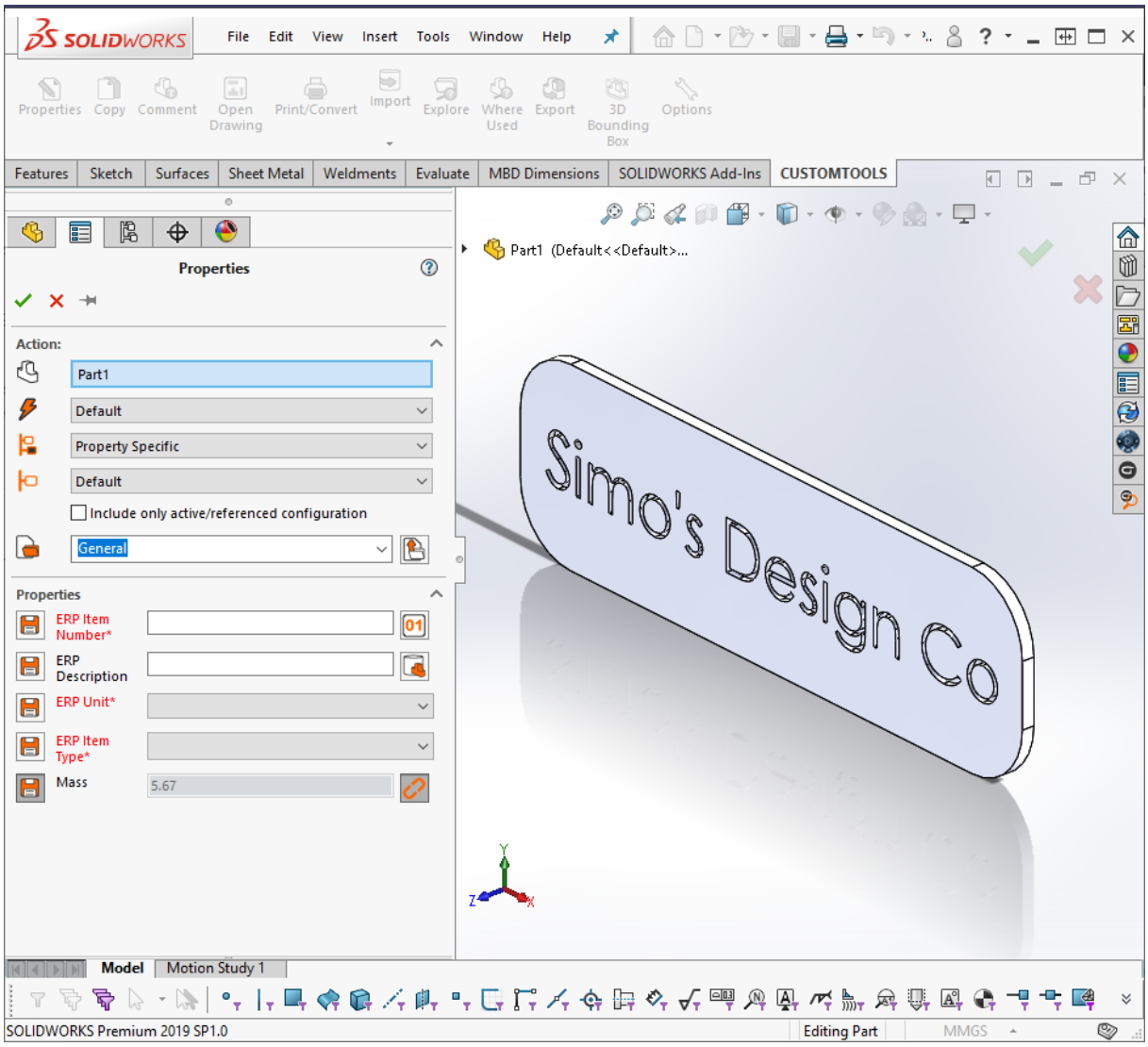


Figure 4.3.6.5: CT Properties as defined in the Profile Options.

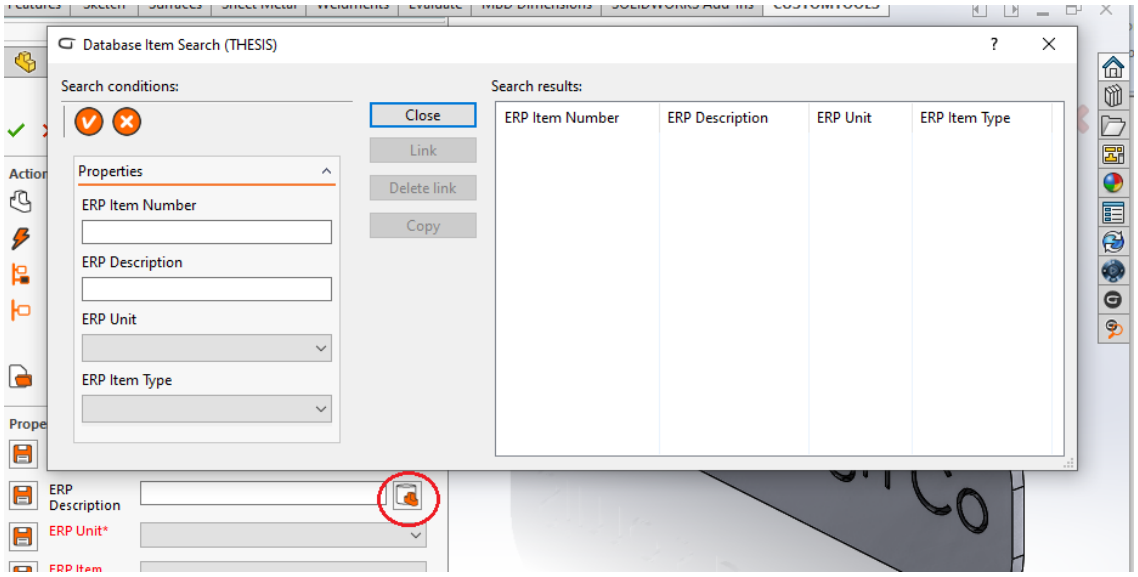


Figure 4.3.6.6: Access the Search Group that allows linking an existing item with this design.

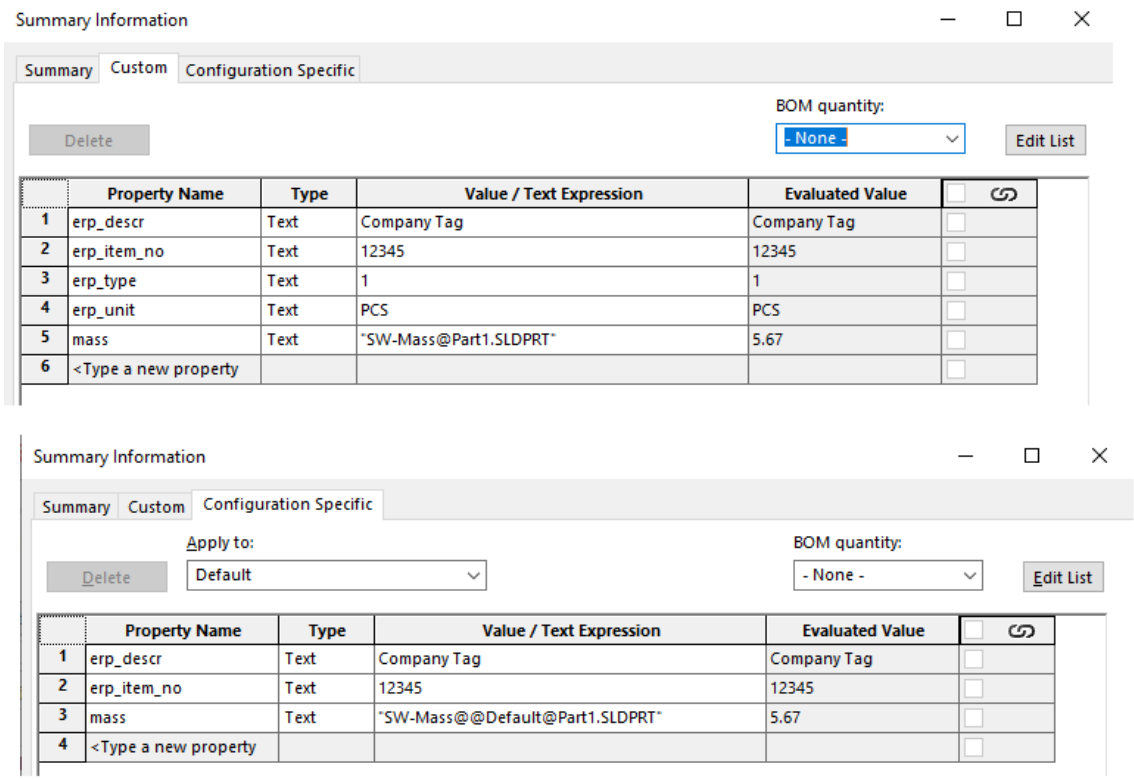


Figure 4.3.6.7: Attribute data stored to design document's Custom Properties with current settings of CT Properties. Item number, type and unit values are of course faked at this point as the integration is not yet implemented.

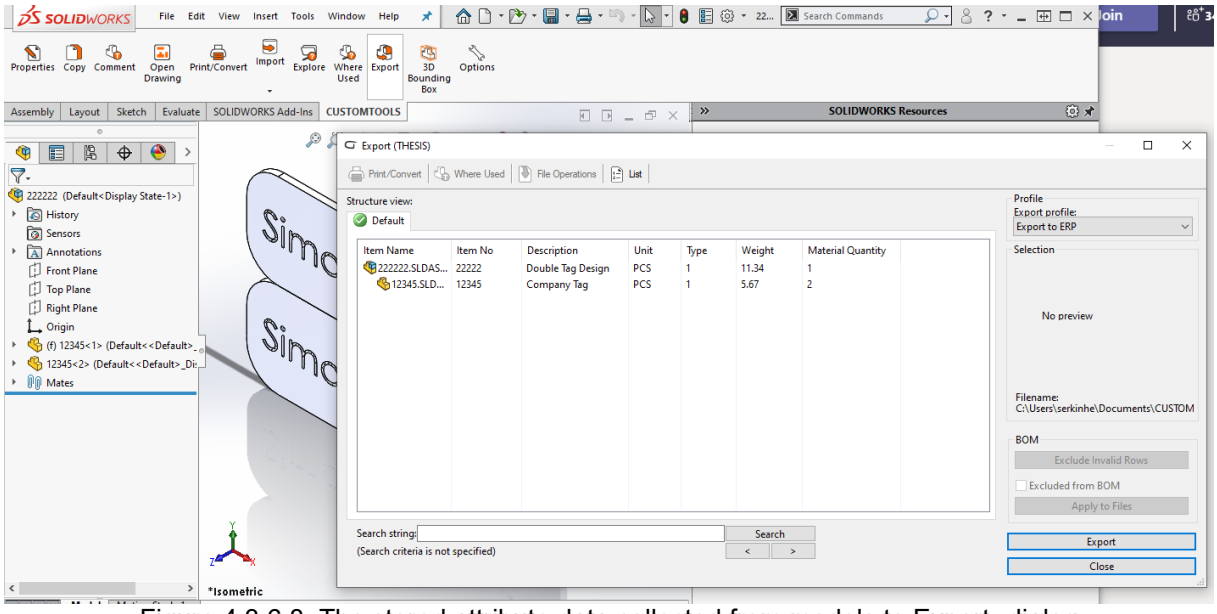


Figure 4.3.6.8: The stored attribute data collected from models to Export -dialog

5 Applying the provided solution

The 11 common SOLIDWORKS - EPR integration requirements (Section 2.6) were all addressed in Chapter 4: Requirements R1, R5, R6, R9, and R10 can be met with Export Type Extension script (Section 4.2.3), Requirements R4 and R7 with generalized architecture for configurability (Section 4.2) and Requirements R2, R3, R8 and R11 by configuring CUSTOMTOOLS Profile to support the existing data model (Sections 4.3).

The generalized steps for SOLIDWORKS – ERP integrations are:

1. Install CUSTOMTOOLS for SOLIDWORKS
2. Configure CUSTOMTOOLS Profile to match with existing design environment (Section 4.3)
3. Implement `CTExtensions.ExportCore.ExportBase` as required (Section 4.2)
4. Implement necessary Export Type Extensions for the ExportBase as required (Section 4.2.3)
5. Deploy the script (Section 3.5.2)

5.1 User implementation example

To demonstrate the power of the base architecture and implementation, the worst-case scenario implementation for data storing requirements is given. In this scenario, the ERP integration requires following abilities to store and configure information:

- Export Profile Field: Field-to-field mapping
- Export Profile: Export profile mapping to Company selection
- Profile: Web Service endpoint in Profile Options
- User: User specific login credentials

Assume `using CTExtensions.ExportCore;` for all the described scopes and classes.

5.1.1 Field-to-field mapping

The main requirement of being able to export items to the target system, even in its most basic form, requires at least mapping of source export field data to target system field. While the implementation example is broad in its overall data storing requirements, providing more complex data objects is unnecessary. Extending the provided object is very trivial.

Data object implementation, `TargetFieldSettings`, must only derive from `SettingsObject`, implement serialization (using conveniently provided `System.IO.Stream` extensions) and return the corresponding `ControlAdapter` for the GUI interaction.

```
public class TargetFieldSettings : SettingsObject {

  // The data to store/load per field
  public string TargetField {get; set;}


  protected override void Deserialize(Stream s) {
    TargetField = s.PopString();
  }


  protected override void Serialize(Stream s) {
    s.PushString(TargetField);
  }

  public override ControlAdapter CreateAdapter(ICTExtension parent)
{
    return new TargetFieldSettingsGUI(parent);
  }
}
```

`TargetFieldSettingsGUI` initializes a textbox to map with the actual `TargetFieldSettings` data object. Notice the simplicity and how strong typing provides compile time safety and ease of implementation regardless of using completely custom data objects with the base implementation.

```
public class TargetFieldSettingsGUI :
                ControlAdapter<TargetFieldSettings> {

  System.Windows.Forms.TextBox tb;
  System.Windows.Forms.Label lbl;

  public TargetFieldSettingsGUI(ICTExtension ext) : base(ext) {
    lbl = new System.Windows.Forms.Label()
        {
            Text = "Target Field:",
          Dock = System.Windows.Forms.DockStyle.Top
          };
    tb = new System.Windows.Forms.TextBox();
    tb.Dock = System.Windows.Forms.DockStyle.Top;
    Controls.Add(tb);
    Controls.Add(lbl);
  }
```

```
  public override void LoadFrom(TargetFieldSettings settings) {
    tb.Text = settings.TargetField;
  }


  public override void SaveTo(TargetFieldSettings settings) {
    settings.TargetField = tb.Text;
  }
}
```

5.1.2 Export profile mapping to Company selection

It's quite common that the target ERP system has tenants, company selections or similar for distinguishing different sub-areas to interact with. For example, testing and production environments may be at the same ERP instance but under different company names. In that case it makes sense that different Export Profiles are then mapped to different companies.

`ExportProfileSettings` holds the `TargetFieldSettings` but also stores Company information per Export Profile. Notice how similar this data object implementation is to the `SettingsObject` implementations even though it packs a complete second serialization level for the Export Field Settings by deriving from `ExportSettingsBase`.

```
public class ExportProfileSettings :
                      ExportSettingsBase<TargetFieldSettings> {

  // The data to store/load per Export Profile
  public string Company {get; set;}


  protected override void Deserialize(Stream s) {
    Company = s.PopString();
  }


  protected override void Serialize(Stream s) {
    s.PushString(Company);
  }


  public override ControlAdapter CreateAdapter(ICTExtension parent)
{
    return new ExportProfileSettingsGUI(parent);
  }
}
```

For the GUI interaction, `ExportProfileSettingsGUI` initializes a textbox to map with the actual `ExportProfileSettings` data object's Company information. Strong typing is the result of cleverly used generics.

```
public class ExportProfileSettingsGUI :
                ControlAdapter<ExportProfileSettings> {

  System.Windows.Forms.Label lbl;
  System.Windows.Forms.TextBox tb;

  public ExportProfileSettingsGUI(ICTExtension ext) : base(ext) {
    lbl = new System.Windows.Forms.Label()
        {
          Text = "Company:",
          Dock = System.Windows.Forms.DockStyle.Top
        };

    tb = new System.Windows.Forms.TextBox();
    tb.Dock = System.Windows.Forms.DockStyle.Top;
    Controls.Add(tb);
    Controls.Add(lbl);
  }


  public override void LoadFrom(ExportProfileSettings settings) {
    tb.Text = settings.Company;
  }


  public override void SaveTo(ExportProfileSettings settings) {
    settings.Company = tb.Text;
  }
}
```

5.1.3 Web Service endpoint in Profile Options

Even though the target ERP may have multiple sub-environments like Companies that are better to map on Export Profile level (Section 5.1.2), for instance the target system endpoint is usually the same for all Export Profiles and therefore better to store on the Profile level.

Again, the `SettingsObject` is derived into a very simple `ProfileSettings` data object, and otherwise at this point the implementation should already be very familiar.

```
public class ProfileSettings : SettingsObject {

  // The data to store/load per Profile
  public string Endpoint {get; set;}
```

```
  protected override void Deserialize(Stream s) {
    Endpoint = s.PopString();
  }


  protected override void Serialize(Stream s) {
    s.PushString(Endpoint);
  }

  public override ControlAdapter CreateAdapter(ICTExtension parent)
{
    return new ProfileSettingsGUI(parent);
  }
}
```

`ProfileSettingsGUI` initializes a textbox to map with the web service endpoint. While the ease of implementation and repetitiveness makes the code listing dull, it is given for the sake of completeness to later compare with implementation that does not use the provided base.

```
public class ProfileSettingsGUI : ControlAdapter<ProfileSettings> {

  System.Windows.Forms.Label lbl;
  System.Windows.Forms.TextBox tb;

  public ProfileSettingsGUI(ICTExtension ext) : base(ext) {
    lbl = new System.Windows.Forms.Label()
        {
          Text = "Endpoint:",
          Dock = System.Windows.Forms.DockStyle.Top
        };

    tb = new System.Windows.Forms.TextBox();
    tb.Dock = System.Windows.Forms.DockStyle.Top;
    Controls.Add(tb);
    Controls.Add(lbl);
  }


  public override void LoadFrom(ProfileSettings settings) {
    tb.Text = settings.Endpoint;
  }


  public override void SaveTo(ProfileSettings settings) {
    settings.Endpoint = tb.Text;
  }
}
```

5.1.4 User specific login credentials

For each user it should be possible to configure a username and password used with the web services.

```
public class UserSettings : SettingsObject {

  public string UserName{get; set;}
  public string Password {get; set;}

  protected override void Deserialize(Stream s) {
    UserName = s.PopString();
    Password = s.PopString();
  }


  protected override void Serialize(Stream s) {
    s.PushString(UserName);
    s.PushString(Password);
  }

  public override ControlAdapter CreateAdapter(ICTExtension parent)
{
    return new UserSettingsGUI(parent);
  }
}
```

Refreshingly differently, there exists a better option than creating a user specific credential control from scratch as was briefly mentioned in Section 4.1. and visualized in Figure 4.1.4. The assembly *ATRControls2.dll* packs `ATRControls2.WinFrom.GenericLoginCtrl` that derives from `UserControl` and can be set up to show different user credential fields, like Username and Password. Main benefit of using this control is that it is localized (which is especially good as user specific settings interact directly with end users of the environment) as well as it provides a common look and feel out-of-the box.

```
public class UserSettingsGUI : ControlAdapter<UserSettings> {

  ATRControls2.WinForm.GenericLoginCtrl loginCtrl;

  public UserSettingsGUI(ICTExtension ext) : base(ext) {
    loginCtrl = new ATRControls2.WinForm.GenericLoginCtrl();
    loginCtrl.ShowUsernameField(true);
    loginCtrl.ShowPasswordField(true);
    loginCtrl.ShowEnableCheck(false);
    loginCtrl.ShowCommunicationPointField(false);
    Controls.Add(loginCtrl);
  }
```

```
  public override void LoadFrom(UserSettings settings){
    loginCtrl.Username = settings.UserName;
    loginCtrl.Password = settings.Password;
  }

  public override void SaveTo(UserSettings settings) {
    settings.UserName = loginCtrl.Username;
    settings.Password = loginCtrl.Password;
  }
}
```

5.1.5 Simple Event Extension using the stored data

Of course, data is not only stored and configured, but also consumed. For that, let us create a very simple extension that subscribes to few commonly used export events and pulls all the stored data on demand in the scope of the current user. `EventExtension` must be derived using the main extension's type in its generics. The main extension will be introduced in the next section but let us now establish its name to **`MyIntegration`**.

```
// Handles export when the Export Profile is bound to MyIntegration
public class ExportEvents : EventExtension<MyIntegration> {

  // Hold reference to main integration for data access
  public MyIntegration ParentExtension { get; private set; }

  // Many events come in specific order so some
  // entry event is usually used to identify whether
  // or not this particular extension should or
  // would want to handle specific events after it.
  public bool HandleEvents { get; private set; }

  // Initialize this class
  public override void Init(MyIntegration parent) {
    ParentExtension = parent;
    HandleEvents = false;
  }

  // Subscribe to what is needed
  public override void Hook(CTInterface iface) {
    iface.OnExportProfileSelected += OnExportProfileSelected;
    iface.OnStrExport += OnStrExport;
  }


  // Unsubscribe what was previously subscribed.
  public override void UnHook(CTInterface iface) {
    iface.OnExportProfileSelected -= OnExportProfileSelected;
    iface.OnStrExport -= OnStrExport;
  }
```

```
  // When export profile is selected, it's binding information
  // will be at its Typename property corresponding to extension
  // identifying name. If it matches to our extension's, then we
  // know we want to handle the upcoming Export events too.
  private void OnExportProfileSelected(object sender,
                   CTInterface.ExportProfileSelectedArgs e) {
      HandleEvents = e.ExportProfile.Typename
                == ParentExtension.IdentifyingName();
  }


  // If the export profile was bound to us, handle the actual event.
  private void OnStrExport(object sender,
                   CTInterface.StrExportArgs e) {

    if (!HandleEvents) return;

    // Finally here we need all the stored data.


    // Export Profile specific settings
    ExportProfileSettings exportSettings =
            ParentExtension.GetExportSettings(e.ExpProfile);

    // Export Profile Field settings from the Export Profile
settings.
    Dictionary<string, TargetFieldSettings> fieldSettings =
            exportSettings.FieldSettings;

    // Profile Specific settings
    ProfileSettings profileSettings =
            ParentExtension.GetProfileSettings();

    // Logged in user's settings
    UserSettings userSettings =
            ParentExtension.GetUserSettings();


    // To wrap it all up
    string endPoint = profileSettings.Endpoint;
    string targetCompany = exportSettings.Company;
    string userName = userSettings.UserName;
    string password = userSettings.Password;
    foreach(var mapping in fieldSettings) {
      TargetFieldSettings targetSettings = mapping.Value;


      string sourceField = mapping.Key;
      string targetField = targetSettings.TargetField;
    }


    System.Windows.Forms.MessageBox.Show(
          "Exporting to Company `" + targetCompany + "`\n"
```

```
            + "at endpoint `" + endPoint + "`\n"
            + "using credentials: " + userName + ":" + password +
            ".");
  }
}
```

5.1.6 The Main Extension

The main extension glues all the previously listed abilities together. As was described at Section 5.1.4, we can equip all of them simply by providing them as generic types for our extension that derives from `ExportBase`.

```
public class MyIntegration : ExportBase<TargetFieldSettings,
                         ExportProfileSettings,
                         ProfileSettings, UserSettings> {

  // Friendly name is shown for the user in
  // various places
  public override string FriendlyName() {
    return "My Integration";
  }

  // IdentifyingName identifies this extension
  // for example object binding and data storing.
  public override string IdentifyingName() {
    return "MY-ERP-INTEGRATION";
  }

  // The event extension
  public override List<EventExtension> GetEventExtensions() {
    return new List<EventExtension>() { new ExportEvents() };
  }
}
```

5.1.7 “My Integration” showcase

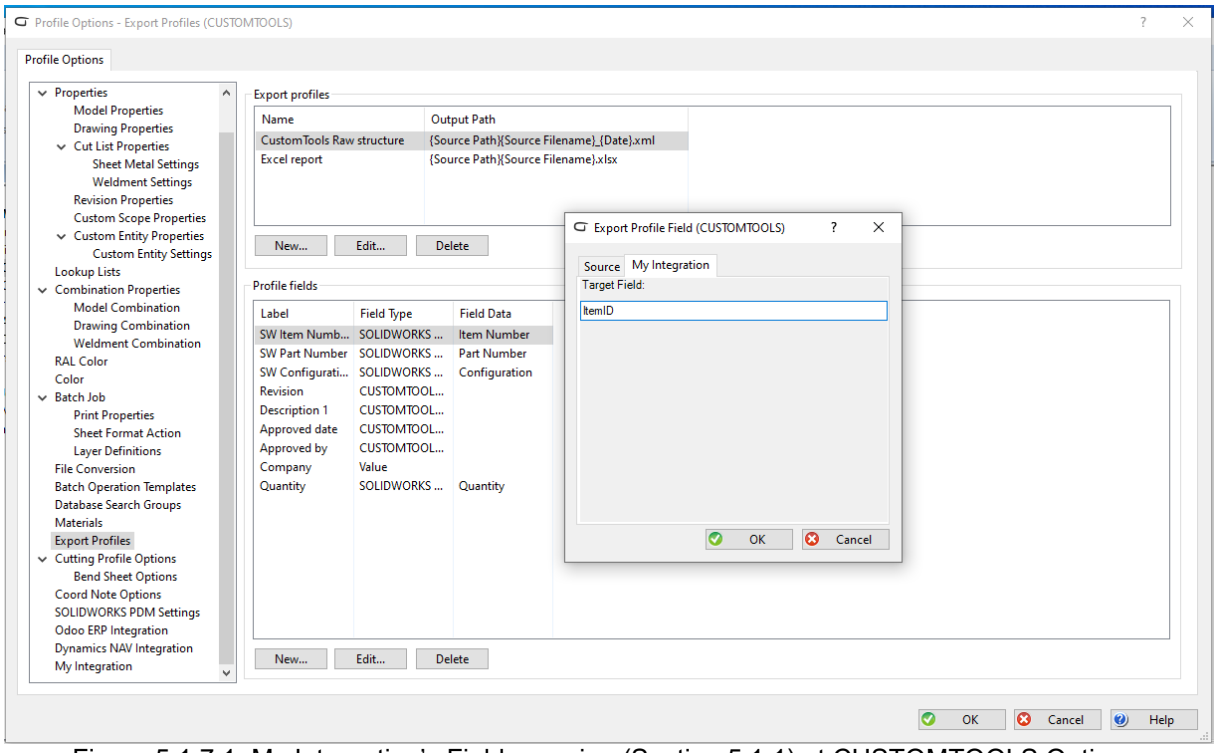


Figure 5.1.7.1: My Integration’s Field mapping (Section 5.1.1) at CUSTOMTOOLS Options.

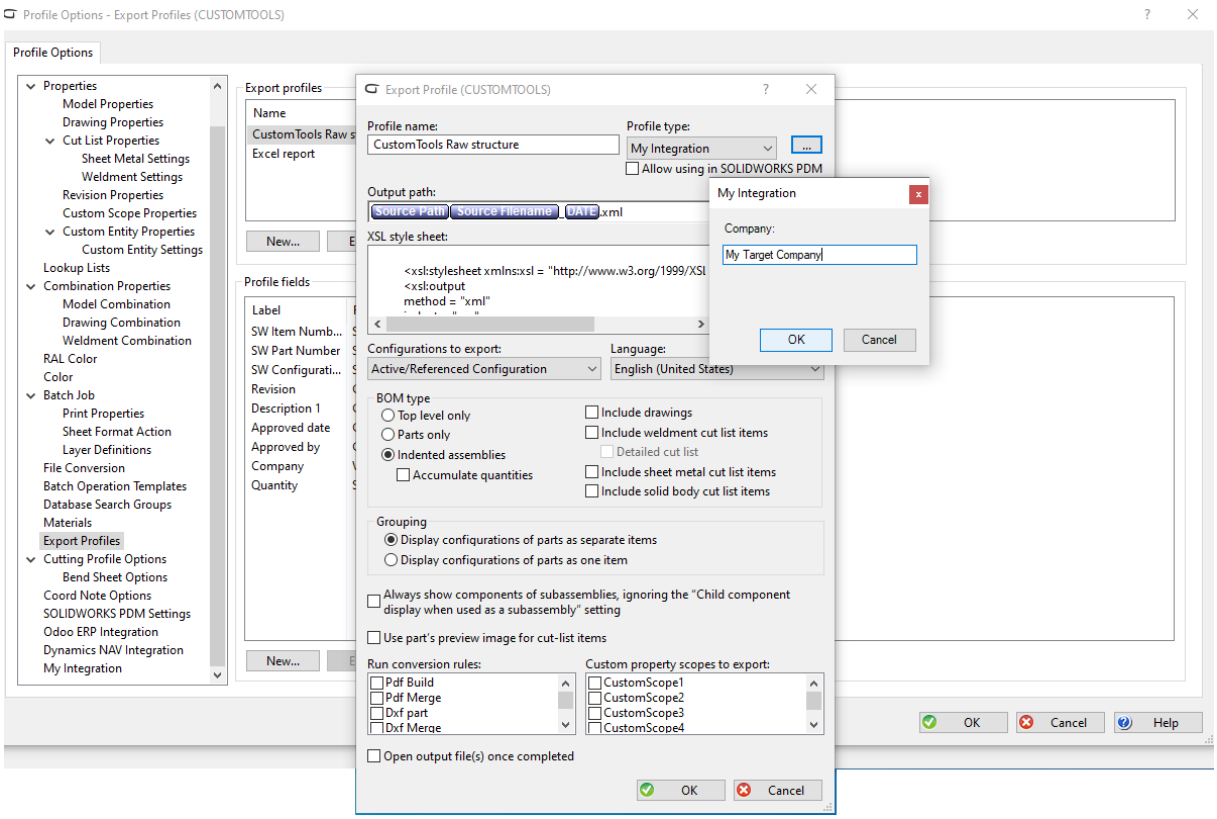


Figure 5.1.7.2: My Integration’s Export Profile binding (Profile type) and Company -mapping (Section 5.1.2) at CUSTOMTOOLS Options

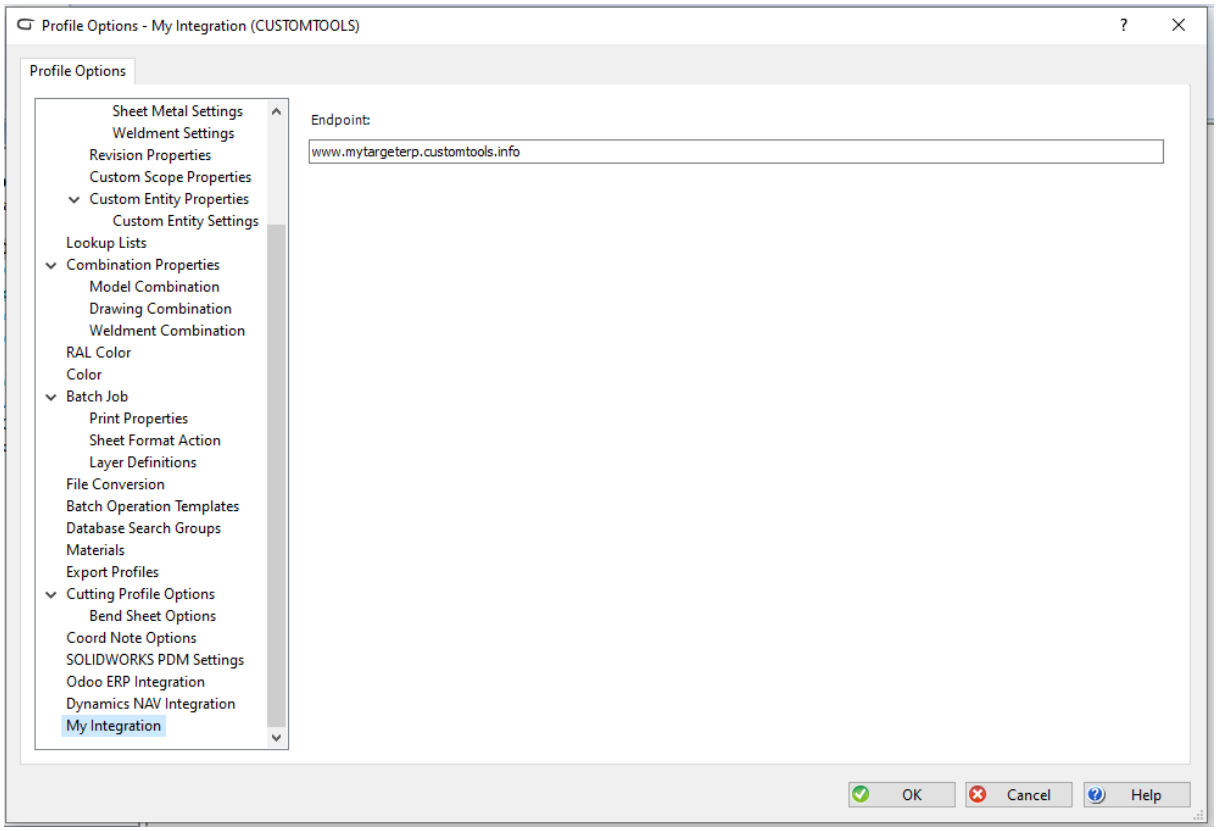


Figure 5.1.7.3: My Integration's Profile Options (Section 5.1.3)

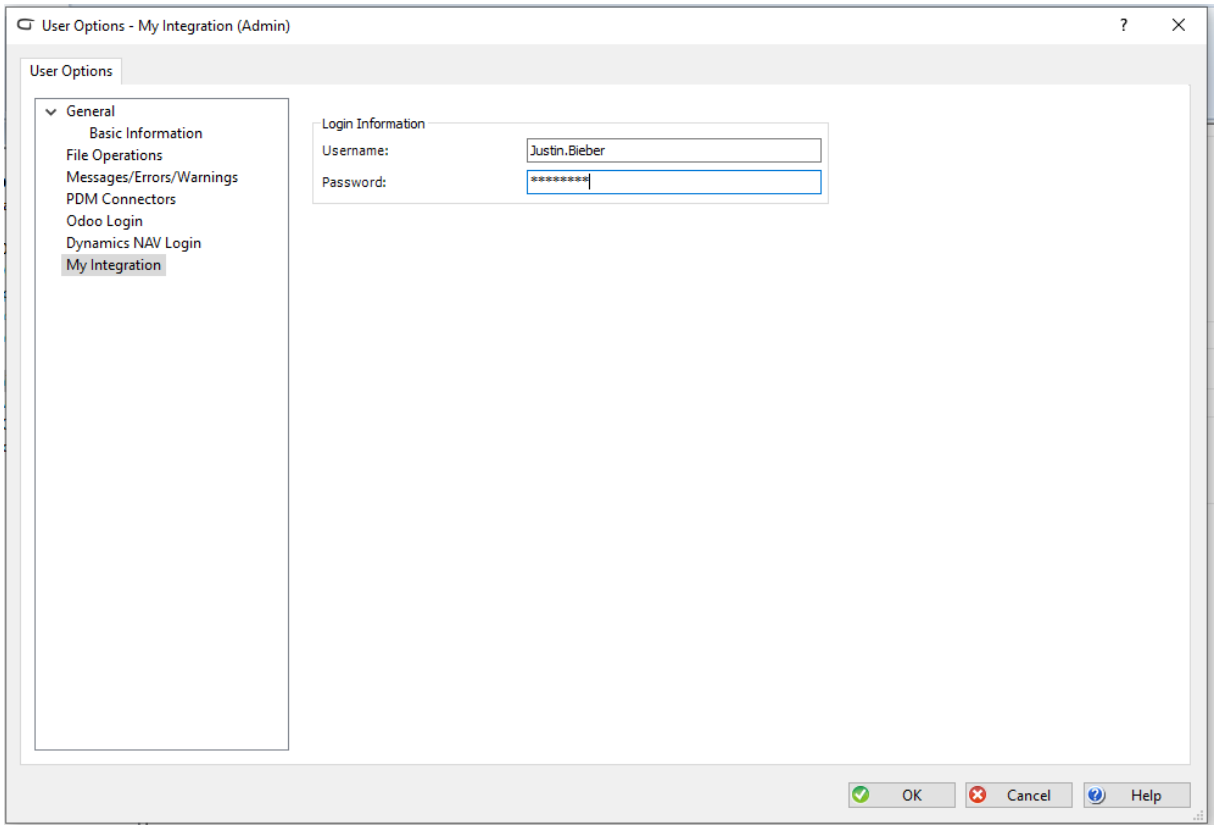


Figure 5.1.7.4: My Integration's User Options (Section 5.1.4)

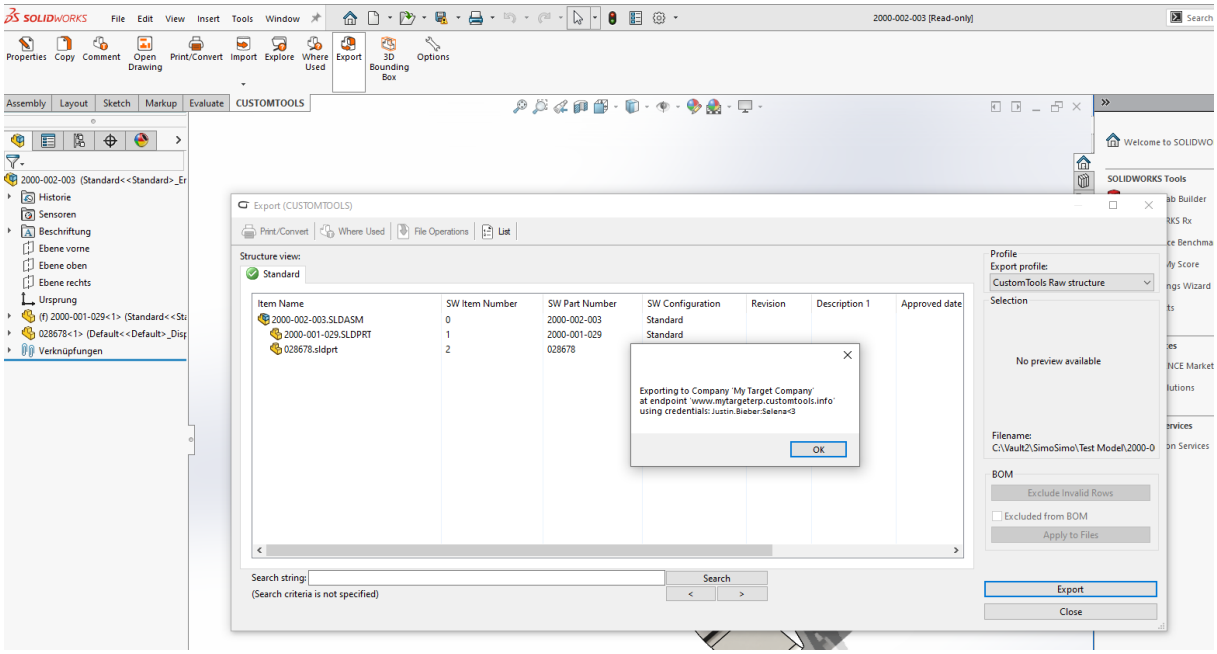


Figure 5.1.7.5: My Integration's bound Export Profile using the stored data (Section 5.1.5)

5.1.8 When the case is not the worst

Important point is that not every case is the worst i.e., it is not always required to implement data storage in any other level than Export Field Settings. For this, the `Dummy` object was introduced to satisfy the generics in `ExportBase`. `ExportEvents` is commented out as it uses both User and Profile settings objects. The add-in would compile if they were introduced in the code but since they are not provided for the `ExportBase` in the class signature, this integration returns null for them. Minimal ERP integration main extension class would then have the following class signature and additionally only require `FieldSettings` (as defined in Section 5.1.1):

```
class MyMinimalIntegration : ExportBase<FieldSettings,
                             ExportSettingsBase<FieldSettings>,
                             Dummy, Dummy> {

  // Friendly name is shown for the user in
  // various places
  public override string FriendlyName() {
    return "My Minimal Integration";
  }

  // IdentifyingName identifies this extension
  // for example object binding and data storing.
  public override string IdentifyingName() {
    return "MY-MINIMAL-ERP-INTEGRATION";
  }

  // The event extension
  public override List<EventExtension> GetEventExtensions() {
    return new List<EventExtension>() { /* new ExportEvents() */ };
  }
}
```

5.2 Comparison & Analysis

Every add-in has its main extension class. For *My Integration* (Section 5.1.6) the complexity is in class signature due to strong use of generics, however, the class implementation itself is extremely simple. In addition, it has hidden internals for extension interface handling and their lifetime/re-initialization control; all of which would have to be otherwise implemented by user script. As was shown in Figure 4.1.5 when implementing from scratch, the architecture of all of these capabilities requires at least the following implementations:

- Main Extension
- ExportTypeExtension
- ObjectEditGuard
- ProfileOptionsExtension
- UserOptionExtension
- Export Field Settings data object
- Export Settings data object
- Profile Options data object
- User Options data object
- Export Field Settings control
- Export Settings control
- Profile Options control
- User Options control
- Serialization of data objects

The above *worst-case requirements* are set to table (Table 5.2.1) and complexity comparison is performed between *Implement from scratch* and provided *ExportBase* implementation for each entry separately. The total and average complexity are calculated for both cases. The same comparison is then done again (Table 5.2.2) using only the *Minimal required capabilities* of an ERP integration (Section 5.1.8). The author himself has populated the complexity numbers for each entry but aimed to do it with emphasized objectivity. Since the ExportBase is completely new, there is no one in the team yet to give feedback about its complexity.

Table 5.2.1: Worst case requirements complexity comparison

Complexity 1-10 (higher more complex, - not needed)

Requirement	**Implement from scratch**	**ExportBase**	**Reasoning / Notes**
Main Extension	8	5	Main Extension has a lot of best practises that should be followed to have bug-free implementation. ExportBase is simple and implements all those best practises under the hood, but its class signature might be hard to understand due to generics. Also, those relatively complex extensions are all implemented for ExportBase.
ExportTypeExtension	6	-	
ObjectEditGuard	6	-	
ProfileOptionsExtension	5	-	
UserOptionsExtension	5	-	
Export Field Settings data object	1	4	Data objects are always very simple by nature but ExportBase forces deriving from SettingsObject that provides both linking with the GUI controls and serialization which raises complexity. From scratch Export Settings must implement some field handling which is very error-prone.
Export Settings data object	4	4	
Profile Options data object	1	4	
User Options data object	1	4	
Export Field Settings control	5	3	All ExportBase controls derive from ControlAdapter and provide strong typing of the Load/Save for the corresponding settings object. Otherwise, it's like WinForms Control, which is the case for from scratch implementations that have to define their own data load/save scenarios each.
Export Settings control	5	3	
Profile Options control	5	3	
User Options control	5	3	
Data Object Serialization	4	-	
TOTAL COMPLEXITY	**61**	**33**	46% less work with ExportBase
AVERAGE COMPLEXITY	61 / 14 = **4.36**	33 / 9 = **3.67**	16% less average work complexity with ExportBase

Table 5.2.2: Minimal required capabilities complexity comparison

Complexity 1-10 (higher more complex, - not needed)

Requirement	**Implement from scratch**	**ExportBase**	**Reasoning / Notes**
Main Extension	6	5	From scratch complexity drops from 8 to 6 due to dropping out of Profile- and UserOptionsExtension. No change for ExportBase.
ExportTypeExtension	6	-	
ObjectEditGuard	6	-	
ProfileOptionsExtension	-	-	
UserOptionsExtension	-	-	
Export Field Settings data object	1	4	Data Object complexity doesn't change per type. Some implementations are simply not needed.
Export Settings data object	4	4	
Profile Options data object	-	-	
User Options data object	-	-	
Export Field Settings control	5	3	GUI control complexity doesn't change per control. Some implementations are simply not needed.
Export Settings control	-	-	
Profile Options control	-	-	
User Options control	-	-	
Data Object Serialization	2	-	50% less to serialize, so complexity drops from 4 to 2.
TOTAL COMPLEXITY	**30**	**16**	47% less work with ExportBase
AVERAGE COMPLEXITY	30 / 7 = **4.29**	16 / 4 = **4.00**	7% less average work complexity with ExportBase

Results of the comparisons clearly state that using the provided ExportBase leads to significantly less implementation work, which at the same time seems to be slightly less complex in nature.

Additional benefits of using the ExportBase comes from the fact that it is an integral part of the core product. This means that script integrations provided for customers have extended maintenance and possibility to even have new features years after project deploy. Using it also unifies the codebase which can lead to better predictability and overall tolerance for future software environmental changes.

5.3 In-house feedback

The suggested ExportBase architecture was presented and demonstrated for the CUSTOMTOOLS Customer Project development team 1.5.2021. Participants were Simo Erkinheimo (Thesis author, Product Manager), **Tero Salonen** (Product Director), **Ilkka Kananen** (Software Engineer) and **Marko Laamanen** (Software Engineer).

The author requested input for the complexity tables (5.2.1, 5.2.2) after the presentation, but attendees were not comfortable doing so as they did not have any experience on the new architecture. The demonstration was not enough for complexity assessment, but as its benefits were still seen, verbal feedback was then requested.

Looks simple to use. Implementing the controls and the mapping has previously taken surprisingly long. The common architecture also reflects as a more unified look & feel generally. Common serialization is also a good idea. (Kananen, 2021)

Can't easily compare because I haven't ever implemented the basis before. It has been so complex that Simo has done it. However, the new architecture looks simple enough to actually use. Still, the reality can only be seen when actually trying to use it. (Laamanen, 2021)

It feels that this standard way of doing things is good. There will be less bugs and when something gets fixed, the fix applies for all. Now-a-days there's a lot of copy-paste and related errors. Common base makes project handovers and maintenance also easier. (Salonen, 2021)

6 Discussions & Closing Words

6.1 Conclusions

Though some individual cases of CAD - ERP integration projects may very well be simple in their nature, and even successful when the limitations are well understood by the user, it was also shown that simply trying to meet all the most common requirements can very easily result in significant complexity (Chapters 3-4). While every integration case is different, the high-level requirements are mostly the same (Section 2.6) and having a standardized solution seems to be welcomed by the integration project experts directly affected (Section 5.3).

The reduced work and complexity (Section 5.2) the introduced integration base offers as well as standardization should lead to better quality code, faster project delivery, significantly easier project handovers, support, and further maintenance. Sometime after the initial demonstration of the base implementation (Section 5.3), it was decided to be included as core component of CUSTOMTOOLS, from CUSTOMTOOLS 2022 SP0 release onwards, and all further ERP integration projects will start using it as an integration base. This can be considered a major success what it comes to offerings of this work.

Also, having the supported requirements listed, as well as knowing a standardized solution for them can be found, has a potential to significantly support sales procedure and even educate the potential customer about the overall solution they might need. Therefore, defining the 11 common requirements of SOLIDWORKS – ERP integrations (Section 2.6) is also one of the major offerings of this work.

Downsides are hard to find, but technically it is possible that faster project delivery has negative financial effect in short-term as the project sizes might reduce.

6.2 Limitations & future improvements

This work is strongly relying on the common requirements given by subject matter experts (Salonen, et al., 2020), which could obviously be argued against. There is no harm in supporting argued down requirements, however, the correctness of the overall provided solution could take a hit if a requirement of significant impact would have been left out.

Though the solution architecture (Chapter 4) considers all those common requirements (Section 2.6), it does not provide any further generalization for Requirements R1, R5, R6, R9, and R10 but simply states them doable as *Export Type EventExtension* (EE). There should be a lot of room for the actual

export procedure standardization now that the overall architecture is well-established for the general case, e.g., *ItemBomExportEEBase.*

Some other general purpose EventExtensions could consider e.g., *CustomSearchGroupEE* and *CustomLookupListEE*, both simplifying the case when target ERP system cannot be queried as linked server but with custom connector.

6.3 Extending the work for other CAD – ERP integrations

Standardization of complex issues has its obvious benefits. But it is the standardization itself that might not be easily achievable; being highly dependent on the CAD in question as well as how it is used in real life scenarios. To attempt similar generalized ERP integration solution with any other CAD system, it seems that at least the following is required:

- Mapping of a CAD component with ERP item (and BOM) must be possible.

- Expert knowledge on the CAD itself
    - Component itemization i.e., Item / BOM formation
    - Real world experience on user behavior for understanding the possible pitfalls and issues with legacy data.
    - Truly mastering the difference of Engineering BOM and Manufacturing BOM in current context

- Expert knowledge/data about past integration requirements with the CAD, so that the requirements could possibly be generalized. Generalizable requirements should rather cover too much than too little.

- Professional software architect to provide a solution following the given requirements: an integrated solution with configurable basis i.e., an integration framework.

- Comprehensive use-case based usability assessment of the solution is a must. General usability of the solution must also not be forgotten.

While above list might not be complete, it strongly suggests that a generalized integration cannot be done without true expert knowledge on the chosen CAD, its user behavior, and knowledge on the past integrations with it. If lacking any of it, it would be the author's recommendation to continue doing fully customized integration projects and attempt the generalization later when the level of experience meets the bullet points above.

References

CAD2M, 2018. *Organize Your Bill of Materials in SOLIDWORKS Like a Pro!.* [Online] Available at: https://blogs.solidworks.com/tech/2018/02/organize-bill-materials-solidworks-like-pro.html [Accessed 21 May 2021].

CUSTOMTOOLS API Help, 2021. *API Help.* [Online] Available at: https://taskpane.customtools.info/en/2021/APIHelp/html/68e69d43-b31b-409e-bc6f-c50f845eaf22.htm [Accessed 21 5 2021].

DASI Solutions, 2014. *Search path order for opening files in SOLIDWORKS.* [Online] Available at: https://blogs.solidworks.com/tech/2014/06/search-path-order-for-opening-files-in-solidworks.html [Accessed 21 May 2021].

Eustache, J., Maranzana, R., Lanuel, Y. & Gardan, Y., 2002. *Managing complexity in a CAD environment,* s.l.: s.n.

Fawzy Soliman, S. C. T. T., 2001. *Critical success factors for integration of CAD/CAM systems with ERP systems,* s.l.: s.n.

Hou, J., Su, C., Zhu, L. & Wang, W., 2008. *Integration of the CAD/PDM/ERP System Based on Collaborative Design,* s.l.: IEEE.

Hwang, W. & Min, H., 2013. *Assessing the impact of ERP on supplier performance,* s.l.: s.n.

Hwang, Y. & Grant, D., 2011. *Understanding the influence of integration on ERP performance,* s.l.: s.n.

Iancu, P. C., 2016. *About SolidWorks Modeling advanced features,* s.l.: Constantin Brâncuşi University of Târgu-Jiu.

Jankowski, G. & Doyle, R., 2011. SolidWorks For Dummies 2nd Edition. In: *SolidWorks For Dummies 2nd Edition.* s.l.:John Wiley & Sons, p. 384.

Kananen, I., 2021. *Sofware Engineer* [Interview] (1 May 2021).

Laamanen, M., 2021. *Software Engineer* [Interview] (1 May 2021).

Lombard, M., 2013. *Solidworks 2013 Bible.* s.l.:John Wiley & Sons.

Muni Prasad, G., James, G., Raj, B. & Satya, S., 2013. *Requirement analysis in the implementation of integrated PLM, ERP and CAD systems,* s.l.: Cranfield University.

Mäkinen, O., 2018. *SolidWorks-ohjelmiston MBD-sovelluksen käyttö teknisen tuotemäärittelyn kuvaamisessa,* Tampere: Tampere University of Technology.

Salonen, T., 2021. *Product Director* [Interview] (1 May 2021).

Salonen, T., Francois, S., Franc, E. & Rosendahl-Halvrosen, T., 2020. *Common requirements for SOLIDWORKS - ERP integrations* [Interview] (16 March 2020).

Schmitz, B., 2016. *The Growing SOLIDWORKS Nation.* [Online]
Available at: https://blogs.solidworks.com/solidworksblog/2016/10/growing-solidworks-nation.html
[Accessed 21 May 2021].

Singh, C. D. & Khamba, J. S., 2017. *Critical appraisal for implementation of ERP in manufacturing industry,* s.l.: LAP LAMBERT Academic Publishing.

SOLIDWORKS Online Help, 2020. *SOLIDWORKS Online Help.* [Online]
Available at: https://help.solidworks.com/2020/English/SolidWorks/sldworks/r_welcome_sw_online_help.htm
[Accessed 21 May 2021].

SOLIDWORKS, 2015. *SOLIDWORKS Introduction.* [Online]
Available at: https://my.solidworks.com/solidworks/guide/SOLIDWORKS_Introduction_EN.pdf
[Accessed 21 May 2021].

Xu, H., Xu, X. & Ting, H., 2007. *Research on Transformation Engineering BOM into Manufacturing BOM Based on BOP,* s.l.: s.n.

Zhu, D. & Yan, D., 2018. *Research on the integration of PDM and SOLIDWORKS,* s.l.: Shanghai University of Engineering Science.