Universität Zürich[UZH]

SYSTEMORPH

Extending a Domain Language for Life Insurance Reporting

University of Zürich in collaboration with Systemorph

Catharina Dekker – 18 723 718

Amos Neculau – 17 711 870

Han-Mi Nguyen – 13 760 657

Supervisor UZH: Doctor Sven Helmer
Supervisor Systemorph: Pedro Fonseca

Date of submission: 14th of July, 2020

Table of Contents

Acknowledgements

We are extremely grateful to Systemorph for allowing us to do our master project with them. The time and effort that has been put in by them, and especially by Pedro Fonseca, were essential to the successful completion of the project. Furthermore, we would also like to extend a big thank you to Sven Helmer who was open to the idea of pursuing a master project outside of the typical structure that the University of Zurich provides and willing to supervise us.

1. Introduction

This master project has been done in collaboration with Systemorph. A company specialized in building data driven software solutions for the financial services market, such as insurance companies. The goal of this project, while using some of the aspects that the Systemorph platform provides, is to create a web application where data from different users and in different formats can easily be viewed, edited and combined into one report using an extension of the domain specific language (DSL). The end-users will also have the possibility to add and edit business logic and rules themselves, without needing assistance from technical support. Thereby providing a solution to the hassle that many companies face, mainly sending and receiving endless files, in different formats, to and from colleagues. But also, eliminating the waiting time regarding deployment when a simple change needs to be made to the business logic, since the users can do it themselves in the web application. In practice we see that aggregating files in different share points and e-mails in order to construct a high-level report takes a lot of time. This browser application will provide a solution to optimize data management efficiently that can be translated to real industrial problems.

The application was built with an insurance company as customer in mind. Therefore, the input for our reports, were mortality rates and balance sheet data. Instead of having to perform the traditional extract, transform, load (ETL) process on the data to get it into the desired structure on the application we use the Systemorph Vertex platform to apply the ETL process on the fly using a DSL. Thus, the user is able to update their input files, combine reports with different formats, apply new business logic, and view data in graph format without having to wait for a new deployment of the application. During this process the code that is part of the DSL is compiled once a change is made, but the application as a whole is only deployed once. This brings great benefits to the end-users by saving time, avoiding human error and increased simplicity. However, once the data reaches a significant size, such as more than one million rows, several drawbacks are encountered. The user will experience waiting times of several minutes, which makes then application less intuitive and interactable. Thus, this tradeoff between flexibility and compilation time reduces with the quantity of data.

The report will begin by providing background information on the key elements needed to understand how the application was developed. Afterwards going into the requirements of the application. There the functional requirements have been laid out, along with the target use cases. We then continue by going into the design aspect, explaining how all the different components interact with each other. Having described all the necessary elements, the actual implementation is discussed by showing a user journey in the web application. Code snippets that highlight some of the challenging aspects of the project will be included in order to showcase how they were solved. The report will then continue into the evaluation of whether or not the requirements laid out in the beginning were achieved. This will be done by highlighting some use-cases with test driven data and discussing some of the usability. Then finally ending the report with the conclusion in which some hindsight and reflection will be provided along with a future outlook.

In figure 1 below, we see a high-level overview of the project. It is designed as a cloud web application using Microsoft Azure and a MSSQL database, as seen in in the lower right corner. The application can have files in different formats as an input and export data in various formats as well. These files make up parameters, reference data and actuarial data[1]. The web application performs data transformations and calculations. It can validate data, create reports and handle business processes such as changing report formulas. These actions can then be integrated with an actuarial application from the insurance domain. Furthermore, we see that two types of users can interact with the web application, mainly business users and administrators that will ensure security compliance and assign/ change roles to different users.

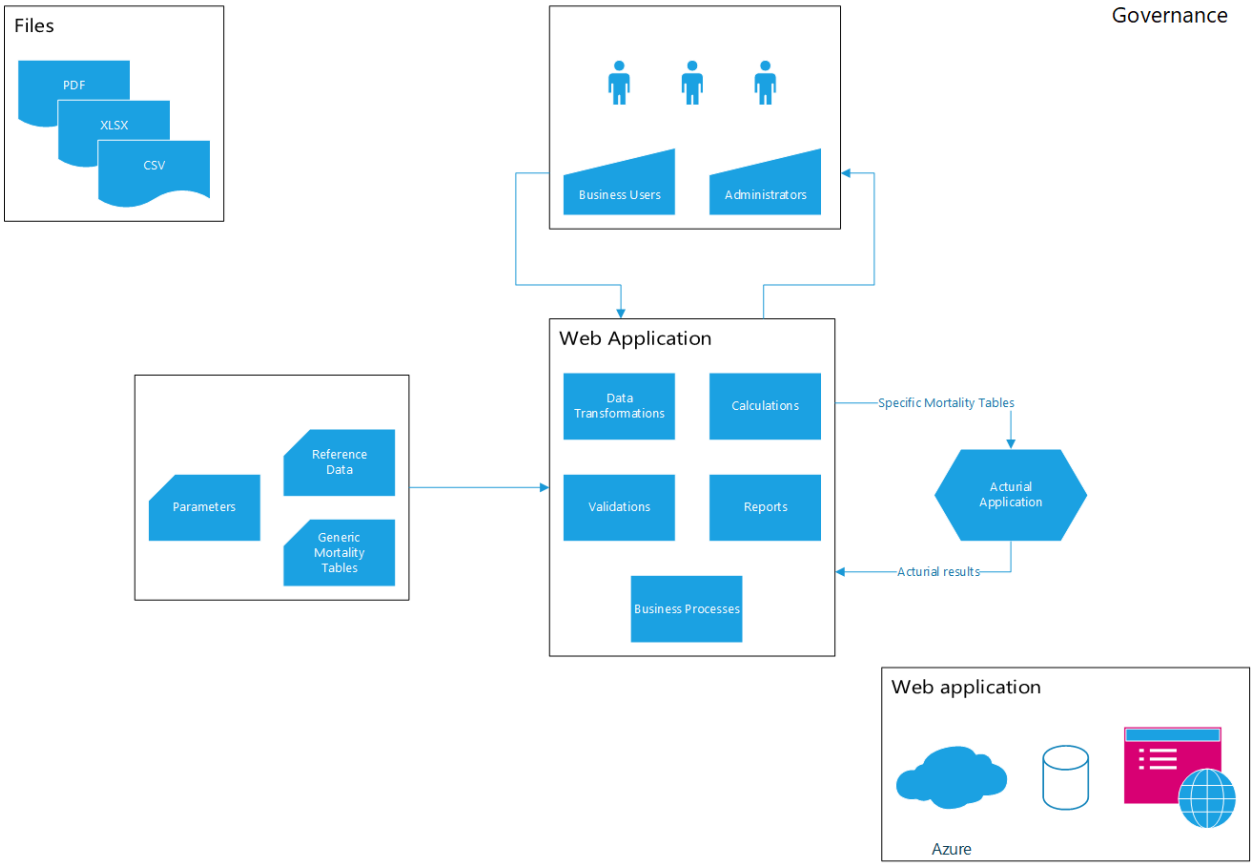


Figure 1: High-level overview regarding the governance of the application

[1] Actuarial data refers to facts and statistics relevant for insurance companies to make risk calculations for their business operations.

2. Background

2.1 Systemorph Vertex Framework

Vertex enables Systemorph products to centralize the capture, communications and control of financial services data. It powers the essential operations and features that are central for data management, regulatory reporting, risk management and other business processes. We built upon the Vertex framework because it allowed us to not have to build simple functional things from scratch, but rather focus on the implementation and extension of DSL.

The Systemorph Vertex Platform is designed to provide a cloud-based foundation of the Systemorph products. Mainly, rapid integration, simple deployment, and easy customization of company data. Furthermore, the Vertex platform allows for capturing control, and communication of data that automates key data management functions. Because of its modern, cloud-based architecture and features, Vertex provides unique capabilities that transform data management.

- Comprehensive data management functions including data-gathering workflows, a master data hub, a calculation engine, and analytics, reporting and visualization tools
- Versioning of data, where user induced changes are only visible in that user's version until it is pushed to production and is then globally visible
- Live tracking of all data dependencies for continuous data validation
- Simple drill downs to all underlying data
- Multiple concurrent workflows for data management and reporting
- Intuitive context-based navigation with full control over user roles, access, reporting & editing rights
- Communication and collaboration across multiple users, multiple tasks, multiple data sources
- Comprehensive documentation of context and reasons for data values

However, as every project is unique, we need to set up a unique data model to solve a specific reporting problem. Thus, Vertex is providing a ready-to-run graphical web interface together with a set of well-segregated interfaces that provide fast integration with the UI. Furthermore, it delivers an effective generic integration with Microsoft's **Object-Relational Mapper (ORM) Entity Framework (EF)**, thus reducing the developer's effort to the minimum (in some cases even automating) of handling the data in **Create/Read/Update/Delete** (**CRUD)** operations. Furthermore, another big advantage of using Systemorph's Vertex platform is the fact that the platform is taking care of migrations and versioning of variables.

In figure 2 below, a high-level overview can be seen of the interaction between the layers of our project. A database is the foundation of the project, where Object-Relational Mapper's Entity Framework (EF) interacts with the database through queries and create, update and delete statements. The Vertex Framework, provided by Systemorph, is the layer on top of the Object-Relation Mapper and allows for the creation of abstract classes, CRUD operations, data migrations and data versioning using EF. The web application built during this project interacts with the two lower layers through Domain Specific Language (DSL) in order to apply business logic and rules along with the concrete implementation of the domain model.

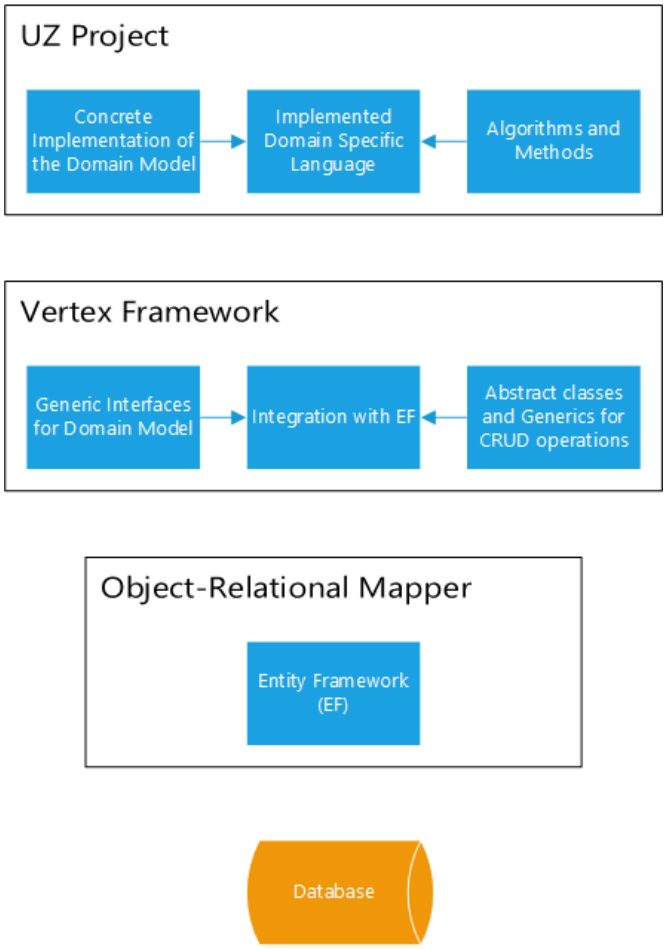


Figure 2: High-level overview of the interaction between the layers of the application

2.2 Software engineering refresher

In this subsection two of the most used design patterns in our project will be described: the **bridge pattern** and the **visitor pattern**.

The **bridge pattern** is generally used when one needs to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them, as seen in figure 3 below.

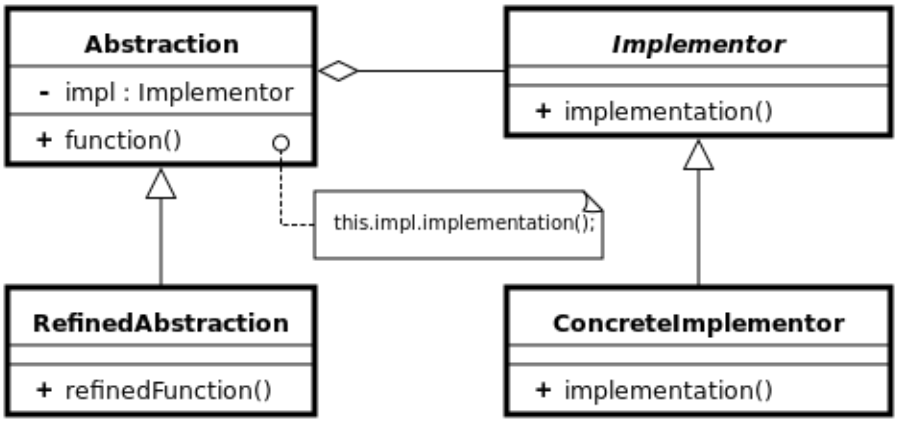


Figure 3: Bridge pattern

In this project, the bridge pattern is used for the implementation of various components of a **storage**, providing context to different resources (e.g. the calculator) to implement various functionalities for the different types of storages (import storage / report storage). Important is to note here, that a storage is not a simple data store, but a complex pool of objects that can be manipulated in various ways such as being filtered, aggregated, or even as new objects that are the result of a calculation between two or more complex data types. Storage will be further explained in section 4.1.

The **visitor pattern** is generally used when one uses a visitor class which changes the executing algorithm of an element class, as seen in figure 4 below. Through this, the execution algorithm of an element can vary as and when the visitor varies. This pattern comes under the behavior pattern category. As per the pattern, element object must accept the visitor object so that the latter handles the operation on the element object.

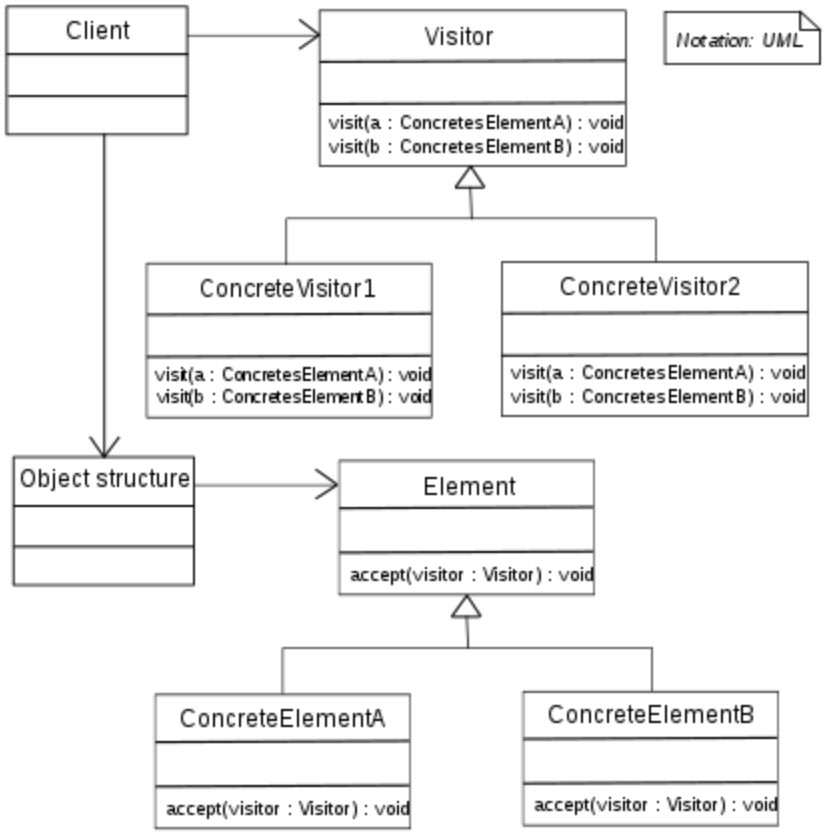


Figure 4: Visitor pattern

The visitor pattern is used in this project to perform both arithmetical and logical operations on the domain model objects (e.g. positions, entities) at runtime and thus making use of the Domain Specific Language (DSL) implemented in a generic way.

2.3 Domain Specific Language

The biggest competitor to the DSL is the Extract, Transform, Load (ETL) method. With ETL, data is extracted from the system, transformed with business logic and then loaded into the target system. Many businesses face this problem where it is needed to bring data in different formats from system A to system B, while adding business logic. The classic approach to this typical problem is to use ETL, the more modern solution is DSL. In our master project we implement DSL in order to validate data, transform data, and build reports. These goals will be described with more detail in section 3.2, functional requirements. In this section we would like to introduce what DSL is and how it can be used.

The classic example to explain DSL would be having data stored in system A and needing it in system B with some modified changes. It is not so simple as just performing a copy-paste because the files could be in different formats, have missing values, etc. If this problem is approached using ETL, the data will be extracted from system A, and transformed into the format needed for system B. During the transformation business logic can be used. For example, system A contains inflation rates of countries within the European Union and Switzerland, system B needs these inflation rates but displayed in a different way, perhaps with a column less. Furthermore, system B has a row for the country Bulgaria,

which is not in system A. At this point business logic can be used to fill in the inflation rate for Bulgaria by taking the average inflation of the other countries contained in system A. Since this problem's approach using ETL, it will be the case that the business logic needs to be applied by an IT professional and can be integrated in the application after the next deployment. Thus, the business user, such as a banker or insurance analyst can't apply the logic themselves. They must communicate it to the IT department and wait a considerate amount of time for an update of the program.

If this problem was approached by using DSL, the business user could write the logic themselves into the program to achieve the wanted transformation. Furthermore, in combination with the Vertex platform, the business user would not have to wait for the next deployment of the software and can immediately see the desired changes. Another benefit to DSL is that the syntax used to write the logic will be close to the industry domain. This is the case since the language is unique for the application with DSL. An example of DSL syntax that we are all familiar with is evident in mathematics. Where mathematics is the oriented domain. Operations such as summing, multiplication, division etc. are represented by symbols that we all know, i.e. +, -, ÷, ≠, ≥, and would be the logic entered by the business user.

Hopefully from the information above you will have started to see that the biggest advantage of DSL is the complete flexibility with registered functions and that it can be rewritten any time dependent on user needs. If you want to sum A and B in a situation with ETL, the code will ask you the values for A and B and will then display the answer. The operation, summing, is coded by an IT professional. Thus, the operation is contained in the code, meaning it has to be redeployed and then you have a running application with the implemented changes. However, with DSL, you have complete flexibility. The app will ask you on which values you want to perform an operation and which type of operation, and then display the result. With DSL you will only have to deploy once and then the compilation is done continuously for the code containing DSL on the fly during run time. Thus, functions can be added or deleted during processing, no need for further deploys or interactions with IT professionals. For example, it is extremely easy to change the formula of a position in the application, which means that changes in the business logic can directly be done by the actuarial user in the GUI.

An example can be seen in figure 5 below, where the business user can change the formula in the Formula box to the desired business logic for bonds. The user here performs a sum of all entities with entity type "BA" and entity type "AssetLiability". This can be seen in the formula box, where the Position("BA") selects all entities with entity type "BA", Position("AssetLiability") selects all entities with entity type "AssetLiability" and the plus sign represents a summation. Once the check mark is clicked, the desired changes from the formula will be evident in the application. This is only one way to change or implement

a formula using DSL. There are other methods to build formulas with DSL, for example with simple math (+, -, /, *, etc.), but also by building more complicated formulas using C#.

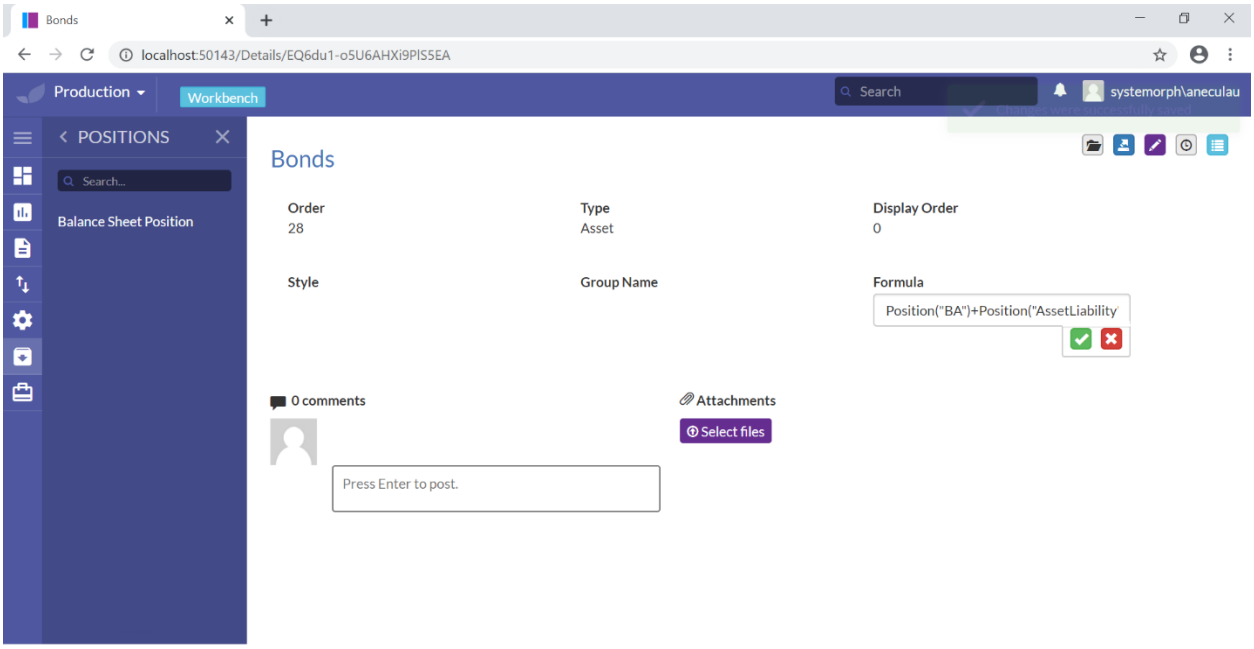


Figure 5: Insertion of a formula by the user with DSL

In our project we use DSL to extract the data from CSV files, such as mortality rates and balance sheet data. First, we create the dimensions needed to construct our entities. For example, with the mortality data we construct dimensions such as Gender and SmokerStatus where we then create a single class for them. Here we specify details, e.g. the dimension should be publicly available, that it should be an integer, etc. Using all the necessary created dimensions we then construct entities. Continuing with the example of mortality data, we construct an entity called “BaseMortalityTable”. This entity is where all the dimensions come together and is what we need to form an informative table containing mortality data, such as Gender, SmokerStatus, Country, Year, etc. An important note here is the fact that such dimensions can be manipulated directly by the user by means of adding new entries. After using DSL during the import, we continue to use DSL to validate data. For example, by ensuring that only recognized genders (male and female) are imported.

A further use of the DSL is the construction of reports in our application. We do not save the reports in the database, but instead save just the minimum amount of data needed to perform the desired operations in order to display the correct information. Usually the data is saved in different tables (the dimensions described earlier), therefore the DSL created must be powerful enough to perform tasks such as aggregation, filtering and basic mathematical operations. Every report needs to be calculated in a different way, hence, some specific functions can be implemented in the C# code and called by the DSL we created. As mentioned previously, we do not store the reports in the database using the ETL fashion. Thus, using the domain language, the business users are free to make the changes needed (i.e. through the business processes) and have the report compiled on the fly, being available in a matter of seconds,

for the entire company. Ultimately avoiding the waiting time associated with integrating changes in future deployments.

An example of a use-case where a business user might want to change the report could be during a pandemic. The user would like to increase all mortality rates for people older than 65. Using an extension of the DSL, the user can easily change the formula for the Mortality Table Report and implement the desired changes.

With the construction of reports we see a big difference between ETL and DSL. Usually with ETL there exist many instances of the application, such as a test instance and production instance. Changes are done by an IT professional on the test instance of the application. If the IT professional is satisfied with the changes on the test instance, the changes are also brought on to the production instance of the application. With DSL the business user makes the changes directly on the production instance. In this case, the production instance will have various versions of this instance. The business user will be able to view his/her respective version where the user can create and test desired changes using DSL. As soon as the user has made those changes, they can have their version be reviewed and then brought onto the production instance.

3. Requirements

In this chapter we will define and describe the use-cases, functional requirements, and non-functional requirements that we decided upon for the project. The data model we designed for this project should fulfil the functional as well as non-functional requirements for the application by having a suitable domain language, where the context of life insurance constitutes the chosen domain.

One of the functional requirements consists of the management and import of data in the form of mortality tables. Next, we are expected to specify relevant parameters for calculations. These parameters need to be updated quarterly and only the latest values should be used. Further on, the import and export of data between any other business application and the project application should ensue in a way such that respective deliverables, mortality tables and actuarial results, are being transformed and validated during the transfers. Moreover, our calculations performed during the import of data should be directly transferred into reports. Our application is to be expected to provide two visualizations of the actuarial results with the corresponding calculations. In addition, we need to design two business-processes that cover the above-mentioned requirements that follow soft and hard validation rules, meaning that a corresponding warning message should be triggered in certain events that either prompts the user to undertake some changes. Where a soft validation rule is in the context of a user not being allowed to proceed without implementing these changes. If the record is not permitted to be saved without implementing the correct changes it is referred to as a hard validation rule. Furthermore, several tests should be implemented to allow for validation of the work done.

In the case of non-functional requirements, we focus on the utility the web application delivers to business-case users and administrators specifically. Our task is to create a web application that can simultaneously fulfill the business needs of two different fields of expertise, namely from the accounting and actuarial department. Furthermore, the corresponding governance rules must be guaranteed to cater for transparency. Concerning the evaluation of the application's performance we distinguish between maintainability for the administrators and User Interface and User Experience for the business-case users.

3.1 Use-Cases

In the created web application, there can be different types of users with different access rights. For example, business users who create, update and delete data or administrative users who can additionally change access rights of other users. In this section three user flows will be described using figure 6 on the next page.

The first user flow will be of a business user who only has read rights to the data. They will be able to read reports and view the master data. There are several operations available at this point, since this user has read rights only, the only available operation will be visualizing the data and exporting it.

This is evident from the flow: Logged in user --> creation/ update/ deletion of data – **No** --> has appropriate rights (read) – **Yes** --> Retrieve data – the data is retrieved from the database – --> Available operations –visualize and exporting data.

The second user flow will be of a business user who has read and write rights to the data. They will be able to read reports and view master data, along with the ability to create, update and delete data. The ability to manipulate data comes along with the power to start business processes and perform operations. Meaning that they have actions available such as reviewing and submitting reports, more

detail on business processes is in section 6. For this type of user all types of operations would be available to manipulate the data, as seen in the top right corner.

This would be evident from the flow: Logged in user --> creation/ update/ deletion of data – **Yes**--> Open business process --> Perform data operations --> Store changes in the database --> Retrieve data --> Available operations – visualize, export, and filter, slice and dice data along with the use of the formula debugger.

The third user would be a business user who has read and write rights to the data long with administrative rights. Thus, having the ability to change access rights of other users. This can be in combination with the other two flows described above.

The administrative rights are evident from the flow: Logged in user --> Change access rights of other users – **Yes** --> Admin --> Perform operations on user rights --> Store changes in the database.

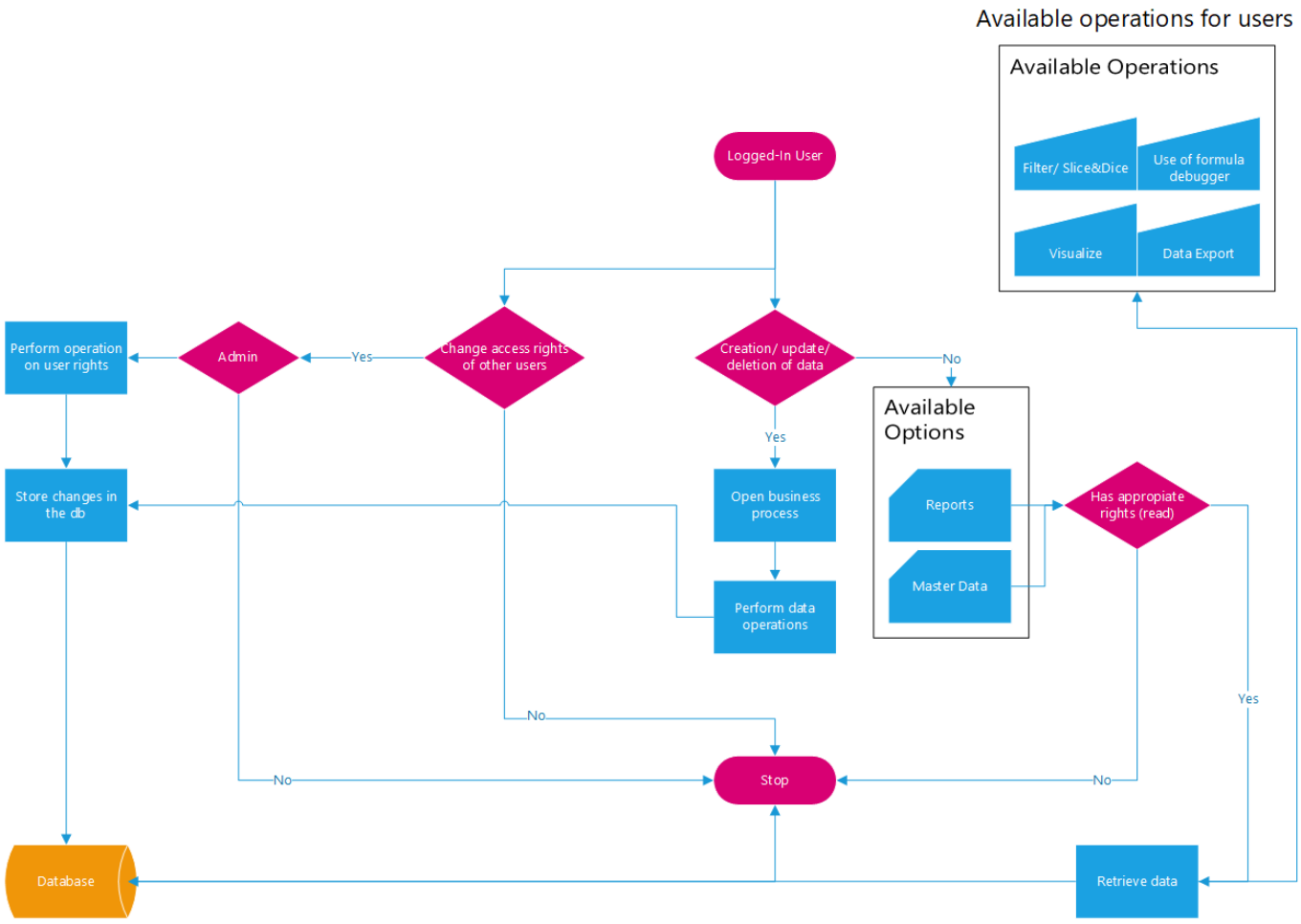


Figure 6: Available user actions depending on user rights

3.2 Functional Requirements

To display reports that are based on an aggregate of data collections of different formats our system must be able to integrate data from different sources. Thus, the first functional requirement relates to the import and export of data between the actuarial application and the project application.

This should be done in such a way that the imported mortality tables and actuarial results are transformed and validated during the transfer from the actuarial application to the project. The fulfilment of this second functional requirement assures the correctness of the transformations applied during the import process. This circumvents the traditional ETL procedure and delves into the core of the program, which is essentially performing ETL “on the fly”. Thus, much of the coding done to fulfill this requirement will also relate to the creation of our own DSL, as described in section 2.1.

It is important to transform the base mortality data to mimic actual practices that insurance companies undertake. Each insurance company has their own view on national mortality data. They take the national data as a base, then apply mortality factors on it. These consist out of their own predictions and calculations based on what they expect to see. Hence, by allowing the insertion of user-defined business rules in our application we provide users with the possibility to make the national mortality data more accurate and useful for their specific applications. This practice is what we aim to fulfill through our DSL during the import of the data and represents the third functional requirement.

The next functional requirement builds upon the import of actuarial and mortality data to visualize the results. Thus, the calculations performed on the mortality tables should directly be displayed as reports in our application. Meaning that the user would be able to see the mortality reports as a graphical overview in the reports section. The goal is to provide two visualizations. One displaying the mortality data as a life expectancy curve. The second displaying actuarial data as a snapshot overview. This would aid the user in their understanding of the data and allowing them to see a summary overview.

Upon completion of the reports, the fifth functional requirement that should be fulfilled are the business-process designs which additionally should cover the previously mentioned requirements. These business processes should be constructed in such a way that it especially guarantees the security of our import process. This should be implemented in three ways, mainly through specific business roles, versioning and testing. The business roles refer to the rights that users have, such as administrative rights where they can change anything in the application, such as data, formulas and user rights. There should also exist a data editor role, where the user is only allowed to change data and formulas within the application. The third and final role exists for a user who is only allowed to view the existing data, but is not allowed to make any changes to it.

The sixth functional requirement relates to testing, which in addition to the business roles allows for security during the import process. It is essential that tests are created for each functional requirement laid out in the beginning to ensure accuracy and precision throughout the project. We previously allowed for tailored business rules and with this functional requirement we will also check whether the resulting data complies to the business rules set by the users.

3.3 Non-Functional Requirements

As for Non-Functional Requirements we could for example measure the following two performance estimates, throughput per seconds and speed, seeing as our web application is interactive. Generally, the waiting time for a query can be decomposed into the following components: the run time to execute code language used in the application and time to query the database. However, by allowing the users to write their own business logic in DSL as function definitions, this supplementary piece of code equates a third operation and entails an additional time component which makes the run time of our application obviously slower. We are aware of this drawback as this constitutes the price to pay for a system that runs real-time.

Hence, we will only focus and work on our application's strengths that entail flexibility and versatility. The only non-functional requirements we define are mainly determined by the utility that our web application should deliver to two target groups. Namely, administrators who maintain and run the web application and business users who utilize, and if need be, fill the business logic for further analysis of the underlying dataset. The corresponding business needs are expected to be handled for each group separately while at the same time in such a manner that the procedures ensure transparency for their counterparts. Furthermore, the web application should provide the business users with simplified technical functionalities that might otherwise have been out of their field of expertise, meaning that the interaction with the database through the GUI should be comprehensible and comfortable.

3.4 Validations

Before diving into the definition and explanation of our data model we first introduce validation measures. These sets of validation functions should help our application to fulfill the above-mentioned functional as well as non-functional requirements, along with ensuring quality of data. In the current project we have two types of validations, i.e. **import validations** and **business processes validations** where import validations are executed when business process validation is not requested. The flow diagram illustrated in figure 7 shows how the fundamental Data Validation is guaranteed by putting the two validation types in context.

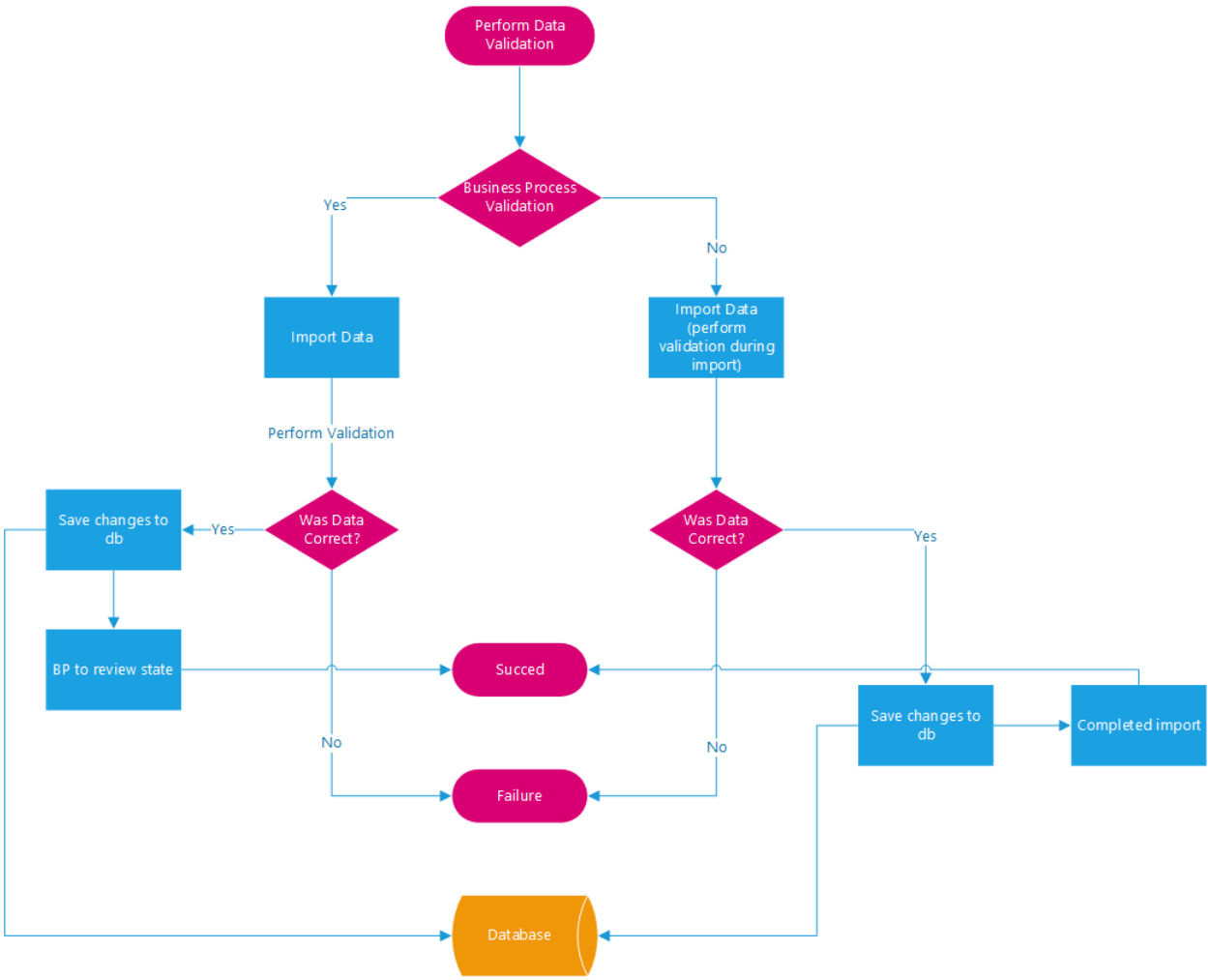


Figure 7: Data Validation Design

3.4.1 Import Validations

The import validations are put in place mainly to perform a check of the data file a business user is willing to import. This check is completed “on the fly” using some registered functions, i.e. the validation code is injected and used at the runtime via FormulaFramework. For common parts of the data files, e.g. the Main table, these functions can be shared (i.e. there is no need of function duplication for each data format) or not shared (i.e. the system performs the specified validations for a particular data format (e.g. “BaseMortalityTable data format” - which is used for the import of BaseMortalityTable entities). The

validations are very important from the point of view of error handling, i.e. it gives back a specific error or warning message in case something fails during import due to the file content. This will give the business user some relevant information, mainly about the reason and point of the import failure.

The following table gives an example of import validation functions and it describes their use.

Table 1: Import Validation Function Examples

Validation Type	Description	Scope	Shared
Validation of Main Args	Performs a validation of the Main Table that is usually used with every data format. It checks: 1) If the @@Main part of the data file exists, otherwise it triggers an error. 2) Makes sure if the data has the correct type, e.g. Year should be a numeral and Country should be a string. If any of this fails, an error is triggered.	All data formats	Yes
Validation of Smoker Status	Performs a validation on a specific data format. It checks if the literal contained in the data file is either: “A”, “S”, “NS” (which are defined in the data model). If this check is defect, then an error is triggered and the business user will see the specific row that made the import of the file to fail.	Base Mortality Table format	No

Most commonly, the validations are performed to check the most important parts of a data file, which are: check the existence of the tables expected, check if all expected columns are contained in the file, check if the data type of each cell is the one the system is expecting (e.g. numeral/ string).

3.4.2 Business Process Validations

The business process validations are a different type of validation compared to the ones used in the import. This type of validations is performed between the steps of a specific business process. A brief example of a business process validation is the one that performs a data check on the “Base Mortality Table” value contents. Generally, we expect mortality rates for females to be lower than males. For example, in the 20th value of the array with mortality rate values that is being imported, the female value should be smaller or equal to the male value. If this is not the case, it will trigger an exception, and the business user cannot change the status of the business process. The business user must correct the wrong data, before being able to continue with the submission of the data.

4. Design

4.1 Framework

4.1.1 Data model

The **data model** (DM) is created by using the essential concepts of Entities, Dimensions and Properties described below. What can be stated without any hesitation is the fact that a well-designed DM is the heart of the solution since the DSL is built to manipulate the data and lay it out in the desired shape. Thus, the Microsoft's ORM EF comes to real help by making use of the basic storage functions of creation, read, update and deletion. In the following part of this section, the most important concepts will be described, along with the context we each use.

An **Entity** is a collection of properties that represents a certain element of the Data Model. The data content of Entities and Dimensions is modeled with properties (fields). **Properties (Fields)** can be annotated with Attributes to extend or modify the default behavior and representation on the UI. We can classify the properties in two parts. The **primitive** properties which can have the type of any of the C# built-in primitive types like "string", or lists, arrays, while the other way is to have **complex** properties represented through another entity.

The **Simple Entity** is the most basic base-class to create Entities. Usually an entity represents a data entry. Some examples of simple entities of our project are the **Base Mortality Table** and **Mortality Factors**. The entities are defined as classes, ending up as tables in the database, while the defined fields (some of them being dimensions) are represented as the table columns in a database. In figure 8 below, the interaction between the dimensions and entities in the application can be seen.

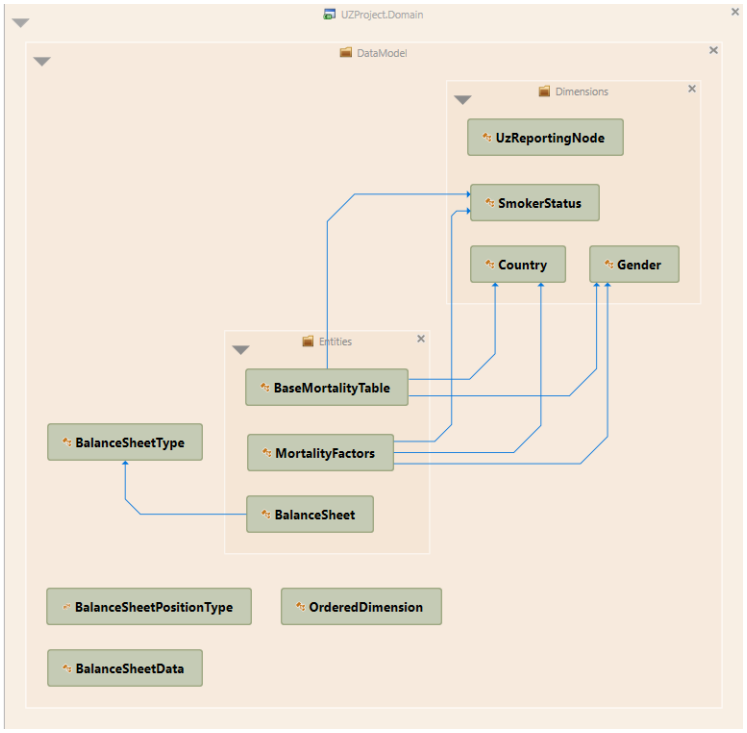


Figure 8: Interaction between dimensions and entities of the application

In figure 10 below an instance of the database is shown with the data model defined in the image above for the Base Mortality Table. In the column “Values”, an array of values is contained, thus the rows for this entity look a bit strange.

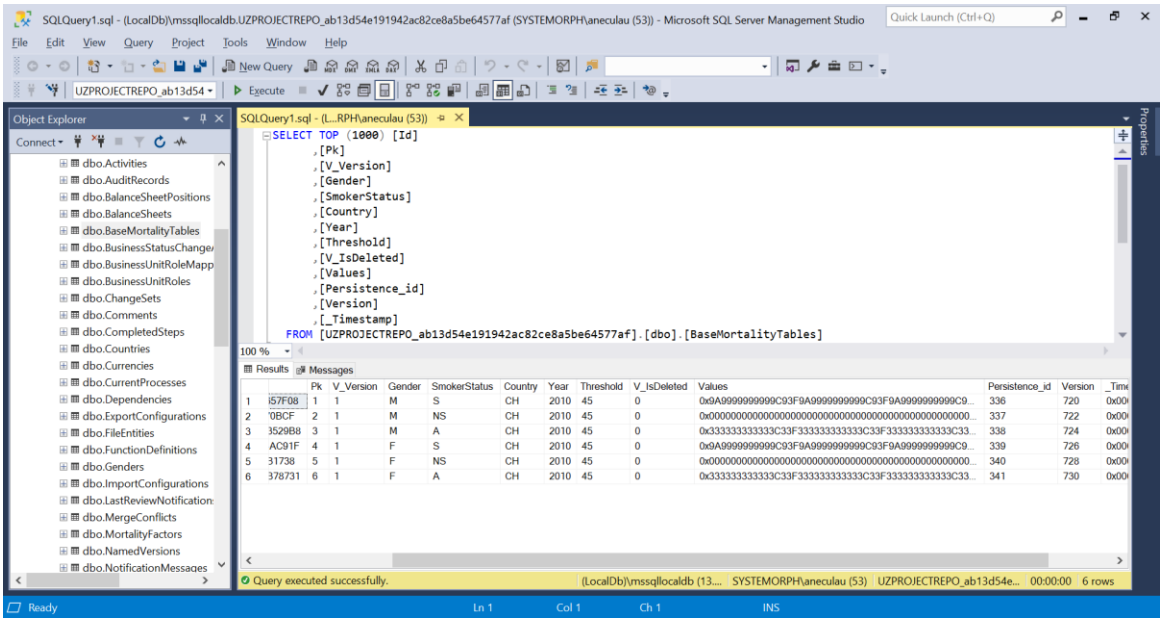


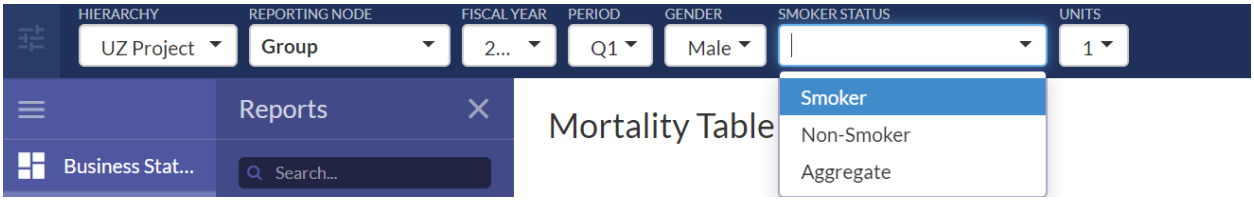
Figure 9: Database Instance for the Base Mortality Table using the data model defined in figure 9.

A **Dimension** is a special kind of Entity. It is not expected to change with time and thus was designed in this way to keep it separate from Entities for easy maintainability and consistency. Dimensions are unique in the system and can be identified by their SystemName as alternative identifier. Typical Dimensions would be “Country”, “Currency”, “Smoker-Status”, “Gender”, etc. The dimensions are not supposed to change frequently, defining the concept of order. Dimensions are used in simple entities as fields providing a certain control-mechanism of what is saved in the database. Furthermore, they are usually used for selecting the data at the GUI level. Below we see an example of the Dimension named Country, which only has Order as a property.

```
1  using Systemorph.Domain;
2  using Systemorph.Layout;
3
4  namespace UZProject.Domain.DataModel.Dimensions
5  {
6      [CatalogMenu(Constants.Dimensions)]
7      public class Country : Dimension
8      {
9          public virtual int Order { get; set; }
10     }
11 }
```

Figure 10: Example of the dimension "Country" with Order as its property

In this image below an example is shown of how dimensions are used for selecting data at the GUI level. For the dimension SmokerStatus we can filter by the entries from the respective table. In this case, Smoker, Non-Smoker or Aggregate (Smokers and Non-Smokers combined).



The **Position** concept is rather new and provides a powerful toolkit in the context of Master Data Management field. Positions are used both during the import and for report-displaying, basically being an entity as described above, but having the concept of the **"formula"** allowing us to perform **aggregations, filtering** and **calculations** at runtime. Furthermore, a position also acts as a reference. Thereby being different from a pointer, as they act more in the manner of a foreign key. To make this concept clear, we exemplify this concept through a toy example below consisting out of two separate tables.

Let's define some entities that will be imported and saved in the database:

Table 1: Raw Data

SystemName	DisplayName	EntityType	Value
A	A	Asset	1
B	B	Asset	2
A	A	Liability	3
B	B	Liability	4

We would like to have a more complex report displaying some calculations using table 1 above as reference. This report can be calculated during the import and displayed, while using a new Entity, in a separate table, table 2 below. Note that the calculations are not done on the raw data, but they are performed on a report level in table 2.

Table 2: New report that is calculated and displayed using positions

SystemName	DisplayName	Formula	Value
A	A	Position("A","Asset") + Position("A","Liability")	4
AB	A+B	Position("A") + Position("B")	10
ABL	A+B (Liability only)	Position("A", "Liability") + Position("B","Liability")	7
BL	B Liability only	Position("B","Liability")	4
AABL	Aggregated A + B Liability	Aggregate(Position("A")) + BL	8

In the table 2 it is evident how in the formula column DSL is used to incorporate simple business logic. By writing Position("A", "Asset") + Position("A", "Liability") we sum the assets and liabilities of all available positions A with entity type Asset and Liability. It selects the rows displayed in table 3 below, thereby the Position acts as foreign key towards entity A in table 1.

Table 3: Selection of table 1 done by the first formula in table 2 using positions

SystemName	DisplayName	EntityType	Value
A	A	Asset	1
A	A	Liability	3

In the second row of table 2 we sum every value of table 1 by writing Position("A") + Position("B"), resulting in 10. In the fourth row of table 2 it is quite interesting to see how the DSL can reference to table 1, but also use BL, which references to row 3 of table 2.

Given that is this is only a toy example, the calculations might not seem complex, exemplifying only trivial arithmetic examples. However, in a real-life actuarial application this is an extremely useful tool. Especially for business users that are not familiarized with a programming language, but share a domain jargon. These formulas can be changed via the graphical interface, thereby making the maintenance process as easy as possible.

4.1.2 Backend components

The concepts, amongst many, that are worth to be described in more detail in this section are the **storage**, **state** and **registries**.

Storage and its components

For both importing and reporting purposes, the application needs a **storage**. But this denotation does not describe its full functionality. The storage is a central concept of the backend of our solution, which can be illustrated as a bridge between the DSL, the application and the existing data. For the importing purposes, the storage is used to handle the raw data. Using the received raw data, the storage can manipulate this data through the importer by using the DSL. The storage will save the manipulated raw data to the database according to the data model provided. Thus, the name storage is a misleading name as it does more than just storing data.

For reporting purposes in our web application, the storage is the tool used to manipulate the sanitized data found in the database in the granularity desired, query it, and apply **transformations** if needed in order to display the reports as wanted. In this case, transformations represent business logic applied as a set of functions that manipulate the underlying data. Additionally, the storage also embodies an environment that exhibits data exploration features, such as aggregations, filtering, slicing or dicing.

For **import** purposes the storage is composed of several parts which will be described in this section, highlighting its "bridge" functionality. The interaction of the components can be seen in the figure 12 below.

- **Compiler**: Used to parse the "source code" of the DSL and compile it via the Microsoft's Roslyn compiler. The DSL code is injected on-the-fly on runtime.
- **Source**: DSL persisted source code, e.g. formulas, positions, calculations of the defined variables, which need to be evaluated by the compiler.
- **Calculator**: Provides logical, arithmetical and sophisticated filtering operations (via the visitor pattern) on existing objects hierarchies allowing dynamically creation/ update of existing objects (e.g. VariableContainer objects).
- **DataStore**: Delegated to cache the data into specific objects of the Domain Model that is imported.
- **Persister**: Used to interact with the database, e.g. perform CRUD operations on the delegated persisted entities.
- **VariableContainer**: An object pool, designed as a storage for variables and entities that are created or used for calculations during import.

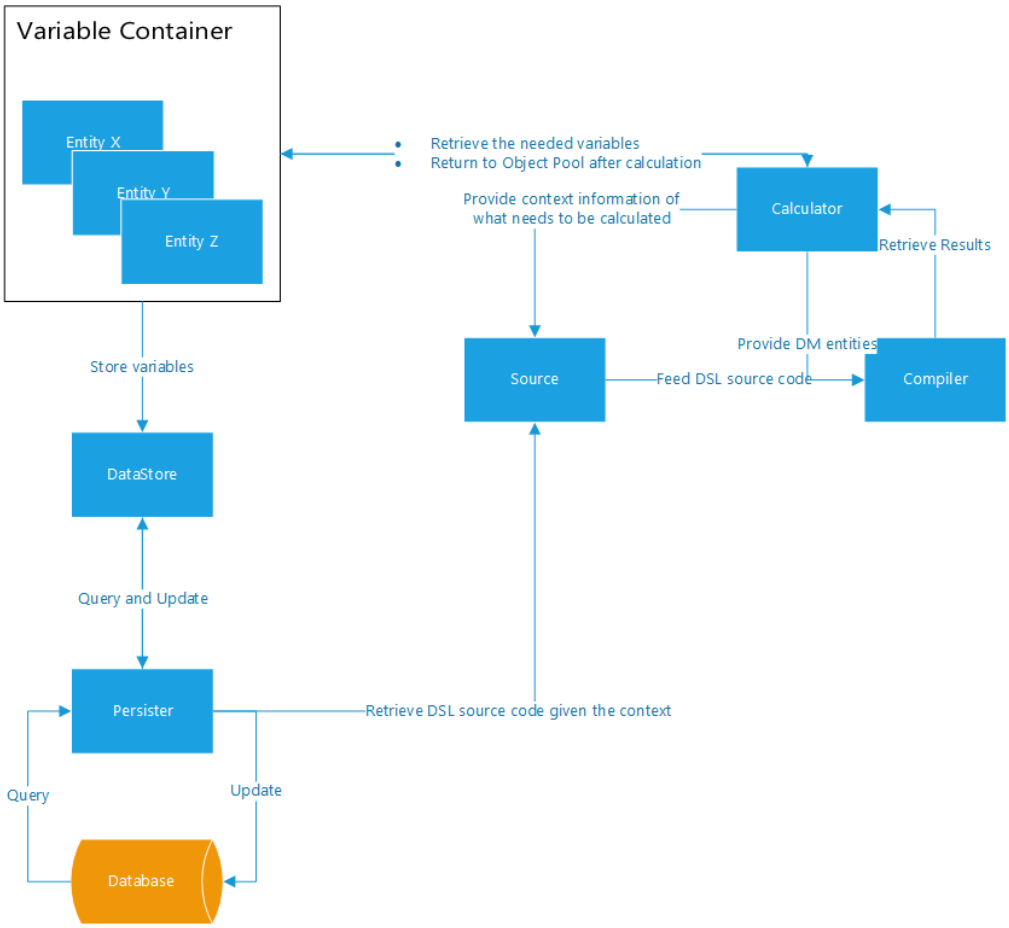


Figure 11: Interaction of components during the import process

Moreover, the storage contains an update method which is delegated to refresh the database with the variables created/ changed during the import in a generic manner.

As mentioned before, for **reporting** purposes, the storage manipulates the data retrieved by queries from the database. To make a clear image of what is going on, its components will be described in the following part.

- **ReportQueryProvider**: Used to interact with the database, e.g. query the data needed.
- **ReportArgs**: Used to filter the data while querying it.
- **Resultskeys**: Identifiers for particular data stored, e.g. reporting nodes, companies, etc.
- **VariableContainer**: Variables queried are stored in variable containers, then the data held by it can be easily transformed, or even used in the calculation of the report.

In addition, for displaying purposes, certain transformations must be performed on the data. These transformations are defined in form of registered methods, being applied on queried data during the report construction. To exemplify such a transformation, one can think of the basic concept of a transposition, e.g. transposing rows to columns.

```
25      static UzStorage()
26      {
27          Transformation<double[]>.Register(MortalityRows,nameof(MortalityRows));
28          TypeRegistry.RegisterType<MortalityFactors>(nameof(MortalityFactors));
29          TypeRegistry.RegisterType<MortalityFactorsData>(nameof(MortalityFactorsData));
30          FunctionRegistry<UzState,UzStorage>.RegisterFunction(nameof(GetMortalityFactors),vc=>vc.Calculator.Storage.GetMortalityFactors(vc.State));
31          FunctionRegistry<UzState,UzStorage>.RegisterFunction(nameof(GetBaseMortality),vc=>vc.Calculator.Storage.GetBaseMortality(vc.State));
32      }
33
34      private MortalityFactorsData GetMortalityFactors(UzState argState)
35      {
36          var ret = queryProvider.Query<MortalityFactors>().SingleOrDefault(x=> x.Gender == Args.Gender && x.SmokerStatus == Args.SmokerStatus && x.Year <= Args.FiscalYear);
37          return new MortalityFactorsData(){Gender = ret.Gender, SmokerStatus = ret.SmokerStatus, Values = ret.Values};
38      }
39      private MortalityFactorsData GetBaseMortality(UzState argState)
40      {
41          var ret = queryProvider.Query<BaseMortalityTable>().SingleOrDefault(x=> x.Gender == Args.Gender && x.SmokerStatus == Args.SmokerStatus && x.Year <= Args.FiscalYear);
42          return new MortalityFactorsData(){Gender = ret.Gender, SmokerStatus = ret.SmokerStatus, Values = ret.Values};
43      }
44      private static IEnumerable<ReportRow> MortalityRows(double[] mfd)
45      {
46          List<ReportRow> ret = new List<ReportRow>(mfd.Select((x,i) => new ReportRow<double>()
47                                                                {
48                                                                    RowDefinition = new ReportRowDefinition()
49                                                                    {
50                                                                        SystemName = "Age"+i,
51                                                                        DisplayName = "Age "+i
52                                                                    },
53                                                                    Row = x
54                                                                }).ToList());
55          return ret;
56      }
```

Figure 12: An example of a transformation method which transposes an array and adds an artificial column

The above example in figure 13 is a transformation method were an artificial column is added, i.e. **Age** and the double array “mfd” is transposed for display purposes. In the smaller red box, the transformation is brought into the storage component of the application. In the bigger red box, the transformation is defined. The transformation is then registered in the MortalityFactorsReport below, figure 14, and will then be linked to a report in the application.

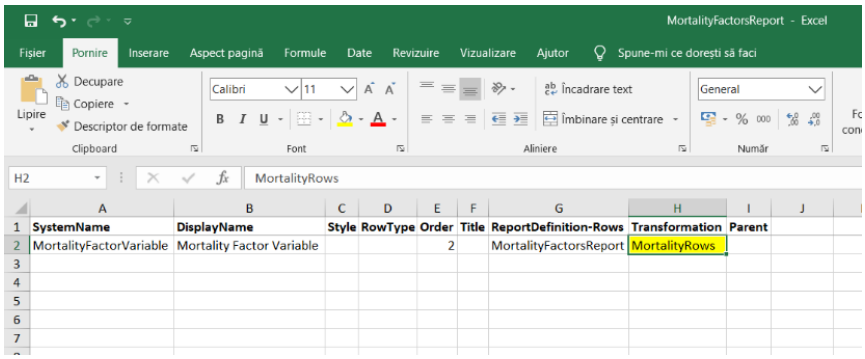


Figure 13: Registering the transformation into the MortalityFactorsReport

State

The **state** is a rather simple, but useful concept in our solution. The state is used mainly by **variable containers**, for both import and reporting purposes. The data held by the state is usually temporal or meta data, e.g. year, quarter, currency or reporting node (RN). As it may be expected, it is used for data partitioning while storing and it helps the filters to retrieve the relevant information to perform the calculations needed in order to correctly display it.

Registry

The **registry** is the integration between the DSL and the application backend logic (e.g. methods, classes, objects, global variables) that can, for example, be found in the state and storage (but not only). To keep it simple, the registry brings in context-compiled C# code to be used fully in our domain language.

4.2 Importers

The process of extracting, transforming and loading data out of multiple sources in our project is conducted with a rather unconventional and immediate solution. Instead of applying the traditional ETL process in separate steps we combined all necessary measurements in one single automatic process by defining **importer functions** written with the help of the DSL created. Thanks to the well-designed Domain Model, the multi-dimensional data can be validated during the import and appropriate transformations and intermediate calculations can be applied right at the import level. Please refer to the flow-chart in figure 15 below that exemplifies the chain of an import function.

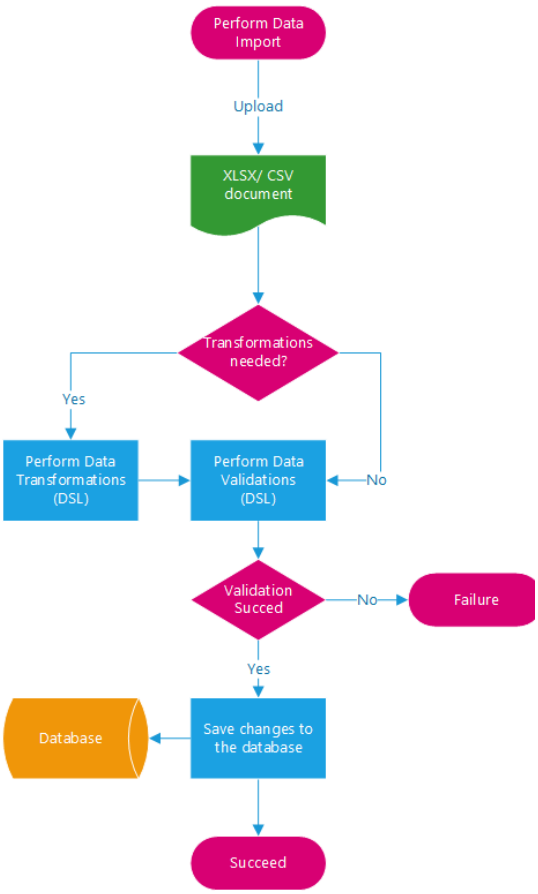


Figure 14: Chain of an import function

As described in the above sub-sections, an importer makes use of a storage and a state. However, these are not the only central components of an importer. An importer furthermore makes use of two other components that must be highlighted. Mainly, Importer Mappings, which contains other useful functions such as transformations based on certain dimensions or specific calculations that are triggered by some thresholds. These functions are registered and used by the DSL for performing transformations to the data, during data import and data validation in a dynamic manner at run-time. The interaction between these components can be seen in figure 16 below.

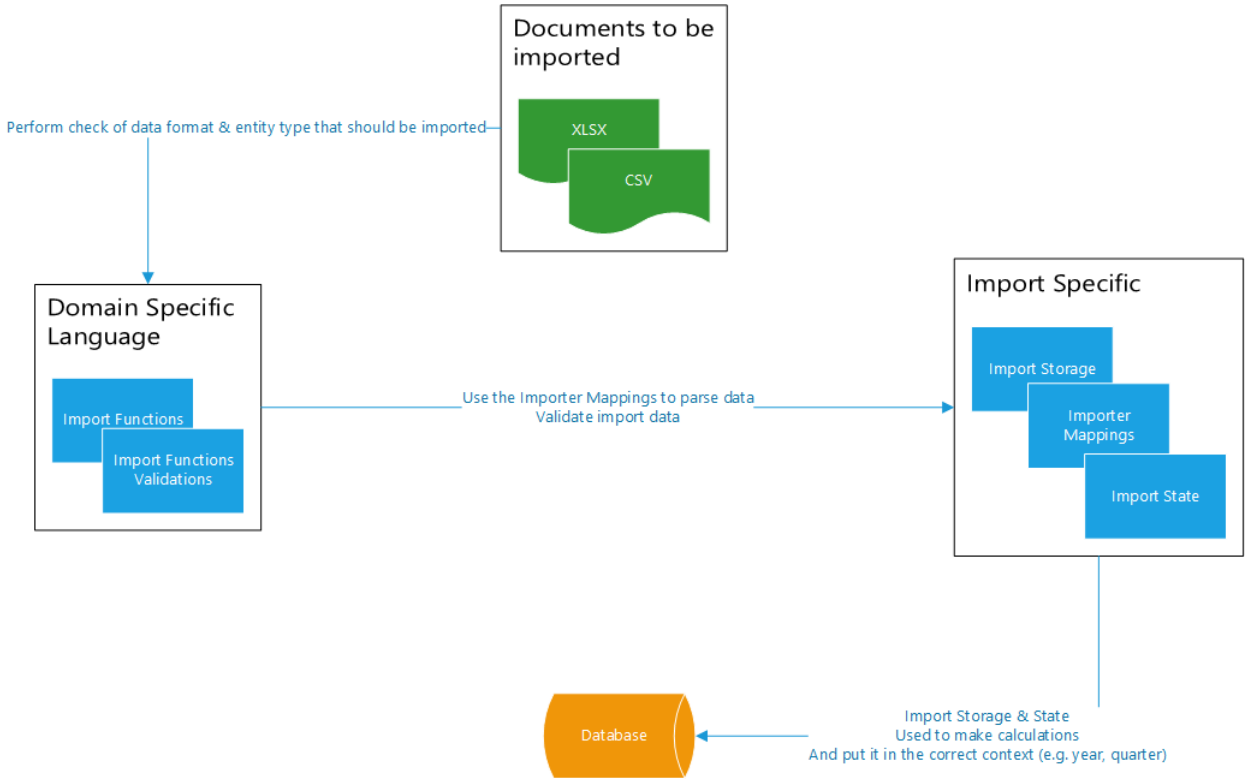


Figure 15: Use of importer mappings and DSL with the importer

Given the importance of importers in our project, these are explained and exemplified in a more detailed manner in the Implementation section.

4.3 Report Functionalities

From the business users' point of view, the reports are one of the most important features. They allow the business user to see the transformed data that was imported in a tabulate form. Furthermore, visualizations in the form of graphs are also included in the reports to allow for further insights related to the imported data.

As explained previously, the data can be manipulated in many ways through “transformations” such as calculations and mappings (directly parsing input when mapping value into the database or storage). For example, it is known that women seem to have a higher life expectancy in general compared to men. If you have the average life expectancy data of men and women combined, you would first apply a calculation where you give women higher life expectancies, and then map it to the female gender in your database. In the end the reports are being displayed in the described **branches**, which is a proof that they are indeed being calculated “on the fly". As part of business processes, there can exist different branches. Where in GIT the stable one is usually called master, in a business domain the stable branch is called production.

The application implements versioned data, meaning that each user gets their own version of the production branch. When a user introduces a transformation on his/her respective version, it is only available there. Whenever a change occurs and is desired to be implemented in production, a business process must be created. During this the branch will have been reviewed by someone else with the respective user rights to do so, this is described in more detail in section 4.4 below. Once this user's version is reviewed and accepted, it can be implemented into production, allowing every user to see the implemented transformations. Thus, the same report can contain different data in different versions. Thereby highlighting the fact that reports are calculated in real time with the existing data with the help of the created DSL.

Our reports are exhibiting the essential and useful features for business users such as **Filtering** and **“Slice and Dice”**. The data visualization problem is easily solved by appending filters for the different dimensions we have registered. When used, the UI filters automatically and sends a query to the database which triggers the calculation of the report in a different way. Hence, with the use of the DSL, this makes it very easy to recalculate or apply transformations to a report. Thus, with a good enough data model, we can build dynamic reports calculated in real time. This significantly improves time management of a division in a company by having the desired data at hand at every moment. We can say that our project uses state-of-art concepts in the data management area, giving us the desired flexibility as we can manipulate the data at the desired granularity.

The **Export** is the feature that allows the end-user to get the transformed data in a convenient format such as CSV. This is a useful feature when needed to share the data outside company or with other departments in no time.

For the sake of user friendliness, each report has a corresponding **interactive chart**, which is defined in the excel file described above. Even though it might not seem as a big feature, this allows the agent to look at the data in a very intuitive manner. Thus, the business user can visualize the data in the desired way.

The **formula debugger**, as part of the Systemorph’s platform, was designed as a tool to help developers debug certain parts of the data model. However, we realized that this is a very useful tool for business

users as well. This instrument allows the agents to investigate a certain position in the report, filter by the dimension desired and offers a step-by-step visualization of how a certain value is calculated. As a result, this powerful tool, provides intuition of how certain values end up in the report and how they are calculated, allowing the user to visualize the data they have in depth.

4.4 Use-case displaying Business Processes

In order to make this application useful for the insurance domain, different business processes must be designed. These business processes are required to create report that has been edited and reviewed at multiple points in a regional hierarchy. In figure 17 below, a high-level overview can be seen of a business process that will be described in more detail in the section below. These processes will be explained with a user story that consists out of creating and submitting a report, which requires several business processes. This demo is composed of 4 users, each with their respective business user roles. The aim of this section is to showcase how each part of the application previously described works together.

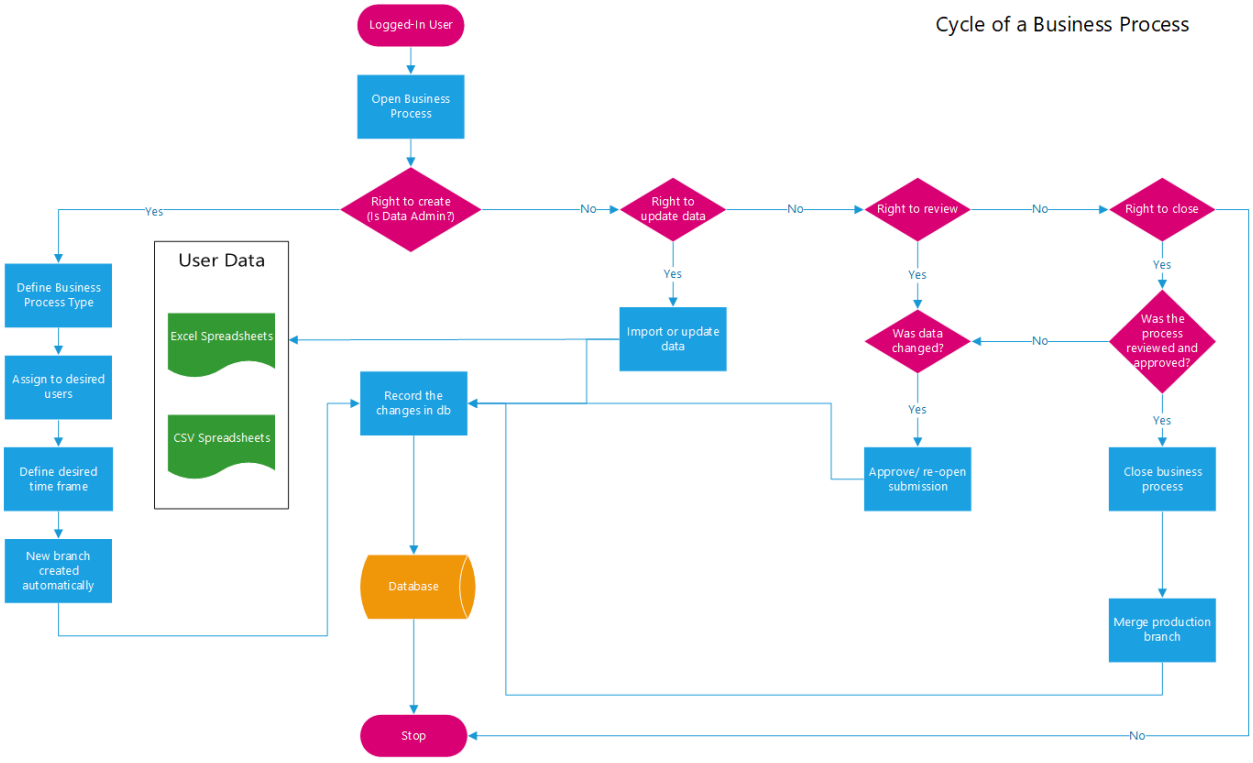


Figure 16: Business processes that exist for different types of users

The first business process is the **Hierarchy Update**, this occurs in local branches of the business. For example, each office of an insurance company located in the cantons of Switzerland have the ability to update their fiscal numbers. In this process the user will have to **submit (SUB), review (REV) and complete (CPL)** data for the report. With the possibility of needing to **reopen (ROP)** the submission if during the review process a mistake was found. The hierarchy update process happens continuously throughout the fiscal year.

- Submission consists out of submitting the data after all the data collection and entry has been done. After submission the user goes into revision.
- Reviewing consists out of reviewing the collected data that has been submitted and analyzing whether mistakes have been made. Upon completion of revision the user goes to the next step which is completion.
- Completion is the finalization of the report containing the collected data. Everything has been deemed to be correct and no errors have been found during the review process. Thus, the report will now go to the next user who works on **Reporting Preparation**.
- If errors have been found during the revision process, then the report will have to be reopened. During the reopening of a report, data can be changed, and errors can be corrected. After reopening the report, it will go back to revision. This cycle continues until the user can successfully review and send the report to completion.

After the completion step in the Hierarchy Update, the report continues to the next user who works in the Reporting Preparation section. During the Reporting Preparation, the local data from the branches get combined to form one big report. For example, all the data from the canton of Switzerland gets formed into a big report displaying the information of Switzerland as a unit. This is made up of the same steps as the Hierarchy Update, I.e. **submission, revision, completion and reopening**. However, since this is on a larger scale, all the steps are at a larger scale and the revision will be more cumbersome. The user will submit the final report of the combined local data streams, send it to submission, if an error has been found it will have to be reopened, if no errors have been found the report can go to completion. In the same fashion as described in the previous business user role, Hierarchy Update. The Reporting Preparation also happens continuously throughout the fiscal year, just as in the previous business user role.

After completion during the Reporting Preparation stage, the report moves on to the next user, who works in the Reference Data department. This user decides what data needs to be added. Since each company is unique and has a different view, they might want to alter the data. For example, different insurance companies have a different view on mortality rates, impact of smoking, etc. Due to these different views, they want to alter the reference data to slightly different predictions. For example, in Switzerland people generally eat healthier than people in Africa, however there is a higher smoking prevalence. This country wide behavior impacts the mortality rates, thus changes could be needed. In order to do these changes, the user would complete the same steps as the previous two users, namely **submission, revision, completion and reopening**.

At this point the report has been reviewed at the local level, such as cantons, during the Hierarchy Update, then at a higher, regional level during the Reporting Preparation. After this the report continues to the Reference Data step, where the company's views have been added. Now the report is ready to go to the **Data Admin** who on a quarterly basis views the report and performs the **Report Update.** This could be done by for example the intelligence department of the insurance company. If the report passes the inspection it will receive the local sign off, meaning that it's perfect and ready to be used. This step, as the others consists out of submission and revision. However, instead of completion, the report will go to local sign off. If mistakes have been found it will be reopened to correct for errors. After the **local sign off (LSO)**, the report will go to the process owner to be signed off, i.e. "**sign-off by process owner (SPO)**" and after that go to completion.

- Local sign off is the step in which the regional manager of the branch looks at the report and decided if it can be completed or needs to go to revision. Upon completion the report will go to the process owner who will perform the “Sign-off process owner”
- Sign-off by process owner is essentially the most important sign-off, meaning that the report is approved by the regional manager. After this the report will move to completion.

A summary of the different steps available for the different business users working with the report is summarized in table 4 below.

Table 4: An overview of the steps available for different business processes

DisplayName	Description	Period	OwnerRole	Steps0	Steps1	Steps2	Steps3	SignSteps0	SignSteps1
Hierarchy Update	Hierarchy Update	None		SUB	REV	CPL	ROP		
Reporting Preparation	Reporting Preparation	None		SUB	REV	CPL	ROP		
Reference Data	Reference Data	None		SUB	REV	CPL	ROP		
Report Update	Report Update	Quarterly	DataAdmin	SUB	REV	LSO	ROP	SPO	CPL

Table 5: Description of definitions

DisplayName	Description	Completion
Submission	Submission	Data complete
Reopened	Reopened for submission	Data complete after reopening
Review	Review	Review complete
Local Sign-off	Company signing off on submission	Local Sign-off complete
Approval data provider	Approval by data provider pending	Data provider approved
Approval reviewer	Approval reviewer pending	Reviewer approved
Sign-off Process Owner	Waiting for Process Owner Signoff	Signed off by Process Owner
Completed	Completed	Completed
Open	Open	Complete editing
Review	Review	Complete review
Closed	Closed	Closed

5. Implementation

5.1 Importer Functions

The role of the importer functions is to transform and validate raw data during the import. We not only extract relevant data into the application but also execute necessary transformations when loading the details by means of our import process. For this purpose, we wrote import functions that first check for validity and then applies necessary modification formulas to calculate measures that comply to the underlying business assumptions. By predefining the structure of our input files (csv or excel files) and incorporating them into our functions as arguments we can guarantee data quality. Meaning that extracted data can only be loaded into the report if it fulfills certain criteria (datatype, bit limit, redundancy, etc.) of the database. We map label names and facts in the sources to the corresponding attributes and data in the database. Additionally, data will be directly allocated to the established hierarchy group when going through the loading process. Especially in our **Base Mortality Importer** we can create multiple potential data out of one single input file of one single category (e.g. AS, "Aggregate Smokers") as data can be duplicated and modified depending on our restrictions resulting out of our assumptions of the population.

An example would be our assumption - that starting from the given values of average men who don't smoke (M "Male" && NS, "Non Smoker"). Figure 18 shows the csv file that acts as the input, where line 7 contains the mortality probabilities for non-smoking males.

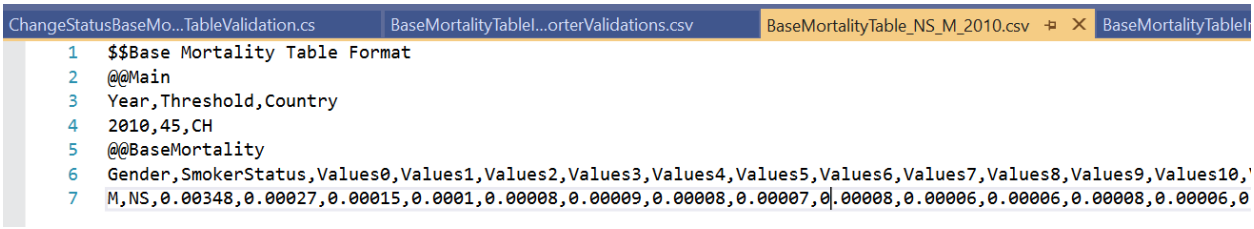


Figure 17: Mortality probabilities for non-smoking males

We propose in comparison to the male population that the mortality rate for women of same age higher than given threshold generally is lower. In our application, using DSL we set the mortality rate, arbitrarily, for women to be 10% lower than men, (0.35 - 0.25 = 0.1), as seen in figure 19 below.

```
",
4,"Base Mortality Table Format",GenderTransformation,ImportState,DefaultStorage,"
    double GenderTransformation(double value, string gender, int age, int threshold){

        if (gender == ""F"" && age >= threshold)
        {
            var aux = value + 0.25d; //increase probability by 25%
            return aux > 1.0d || aux < 0.0d ? value : aux;
        }

        if (gender == ""M"" && age >= threshold)
        {
            var aux = value + 0.35d; //increase probability by 35%
            return aux > 1.0d || aux < 0.0d ? value : aux;
        }

        return value;
    }
"
```

Figure 18: Adjusting the imported mortality rates for males and females

However, the mortality rate of all smokers (“S”) is expected to be higher by 0.2 (arbitrarily chosen). Additionally, the aggregate smoking group (“A”) has a higher mortality rate compared to the non-smoking (“NS”) counterpart, where the preceding values are further subtracted by 0.15, as seen in figure 20. All transformation functions are stored in CSV files that can be accessed and edited directly in the user-interface of our web application.

```
",
5,"Base Mortality Table Format",SmokerStatusTransformation,ImportState,DefaultStorage,"
    double SmokerStatusTransformation(double value, string smokerStatus, int age){

        if (smokerStatus == ""S"" && age > 13)
        {
            var aux = value + 0.5d; //increase probability by 50%
            return aux > 1.0d || aux < 0.0d ? value : aux;

        }

        if (smokerStatus == ""A"" && age > 13)
        {
            var aux = value + 0.3d; //increase probability by 30%
            return aux > 1.0d || aux < 0.0d ? value : aux;
        }


        if (smokerStatus == ""NS"" && age > 13)
        {
            var aux = value - 0.15d; //decrease probability by 15%
            return aux > 1.0d || aux < 0.0d ? value : aux;
        }


        return value;
    }
"
```

Figure 19: Adjusting the mortality rates for smokers based on the imported data from non-smoking males.

Hence, out of one initial single coding line of data for non-smoking men, figure 18, we can construct plausible values for various cases as explained above. This results in the following output, figure 21, listing in total six combinations of values for Gender and SmokerStatus. Six combinations have been created by having 3 scenarios for males and females: smokers, non-smokers and aggregate (smokers and non-smokers combined).

```
1  $$Base Mortality Table Format
2  @@Main
3  Year,Threshold,Country
4  2005,45,CH
5  @@BaseMortality
6  Gender,SmokerStatus,Values0,Valu
7  M,NS,0,0,0,0,0,0,0,0,0,0,0,0,0,(
8  M,S,0,0,0,0,0,0,0,0,0,0,0,0,0,0
9  M,A,0,0,0,0,0,0,0,0,0,0,0,0,0,0
10 F,NS,0,0,0,0,0,0,0,0,0,0,0,0,0,(
11 F,S,0,0,0,0,0,0,0,0,0,0,0,0,0,0
12 F,A,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Figure 20: The result in csv format of applying business logic contained in DSL to the input data from figure 18

The **Balance Sheet Importer** is built in the same fashion but is constructed differently in order to project the positions and corresponding facts of a company, or group of company, for a specific year and quarter. An example for a Balance Sheet might look as follows in figure 22 below.

BalanceSheet.cs | BalanceSheet_Group_2006_1.csv

```
1  @@Main
2  ReportingNode,Year,Quarter
3  Group,2006,1
4  @@BalanceSheetImporter
5  Position,Value
6  Assets,1000
7  Liabilities,500
8  Equity,2000
```

Figure 21: An example of the data used for the Balance Sheet Report

5.2 Implementing New Functions with DSL

We will now show an example of how DSL is used in the application. In figure 23 below we see how a user can create a new dimension, import it and apply business logic in the form of DSL on this new dimension[2]. In the image below the business user uses the web application's UI to create a new dimension with the display name "New Smoker Status" and the system name "NSS". Once the user clicks on OK, this dimension will be saved in the database and the data model is changed to include the new dimension. Note, that this change in the database and data model is only visible for this user's version of production. Thereby preventing potential mistakes in the data model or database to be present for all users.

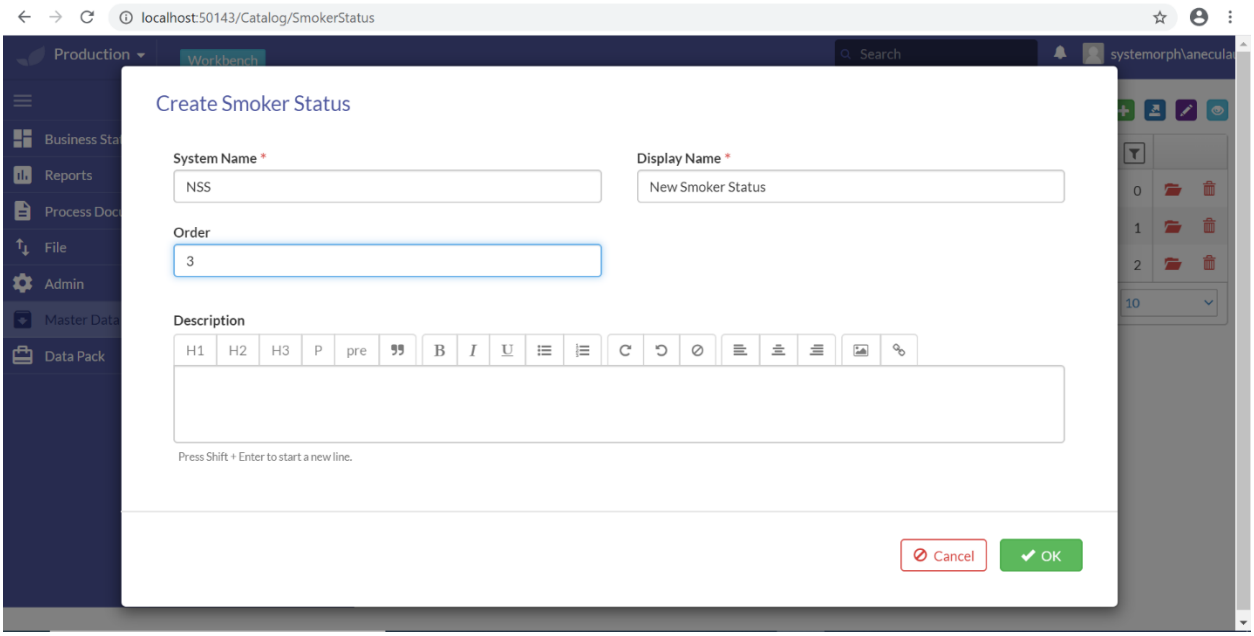


Figure 22: Example of a user creating a new smoker status

This will lead to the question of how the new dimension will be manipulated during the import. Fortunately, the DSL is not just a bunch of registered hard-coded functions that the external compiler takes care of. Given the power of Roslyn, the Microsoft's C# compiler, C# code will be injected and compiled at run-time. Hence, if we have the importer defined in C# outside the compiled-dll library, by means of the visitor pattern, we then can change the business logic after the change above for both importer and import validations. Moreover, a new function can be added any time and be used along with other functions. In figure 24 below you see how a user can create a new function. In the code box the user can enter code either in C# or in DSL to implement the desired business logic. Thus, the user has flexibility to implement changes to formulas that are immediately compiled without having to wait for the new deployment of the application.

[2] A dimension is a property of an existing entity which is modeled as a separate data element, a more detailed explanation will be discuss in section 4.1.1.

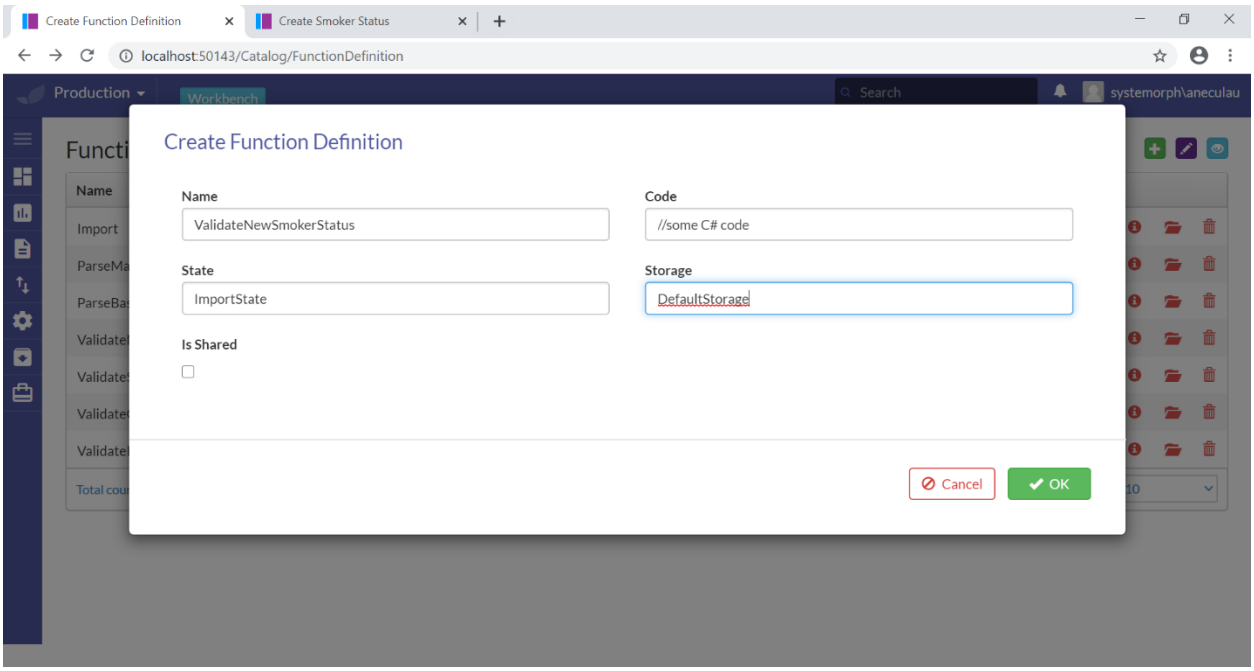


Figure 23: Example of how a user can create a new function using DSL or C# code

In this image we see an example of code that the user implemented to validate the newly introduced dimension “New Smoker Status”. The introduced code asserts whether imported data regarding SmokerStatus is valid and aligns with the data model. Note, that this function is integrated with the application without having to redeploy.

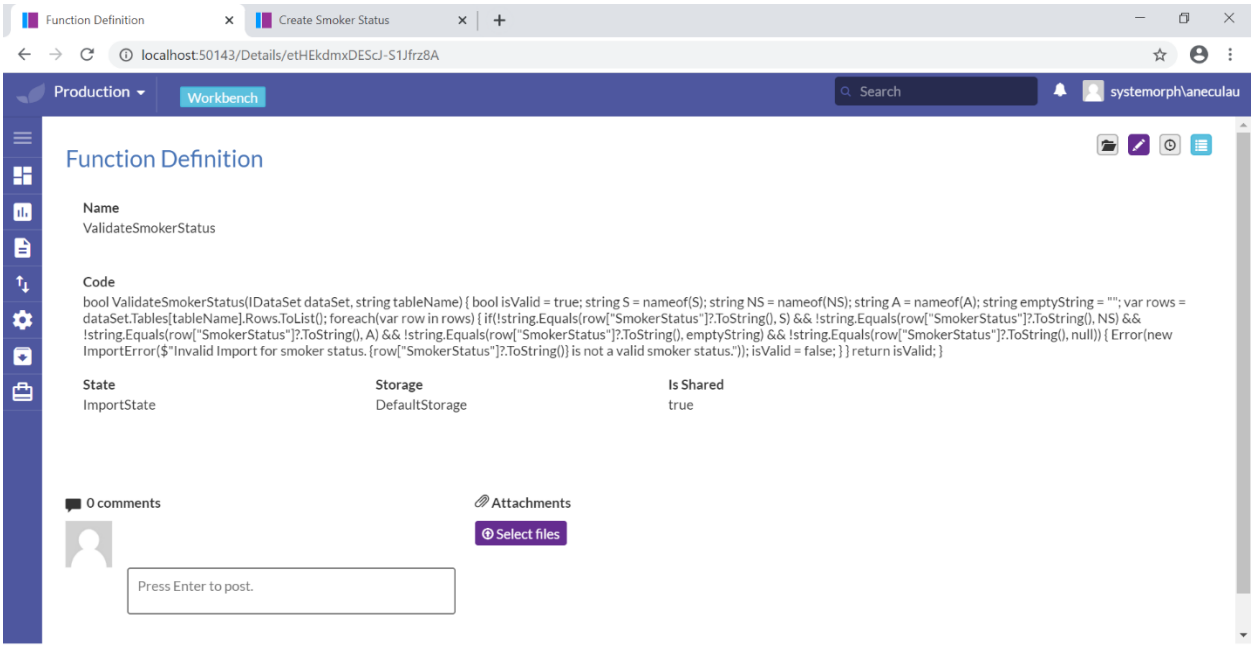


Figure 24: Integration of the user defined function with the application

5.3 Reports

Reports are needed in the application to allow for business users to have an overview of the data in the application. A report is defined in an excel file and added to the initialization convention - the platform which takes care of storing the relevant details in the database and then displays it in the desired way. Each report is defined in 4 tabs, i.e. **ReportDefinition**, **ReportVariableDefinition**, **ReportRowDefinition** and **ReportColumnDefinition**. In the end, this information will end up in four different tables in our database.

1. **ReportDefinition**
    - **SystemName:** Used as an identifier
    - **DisplayName:** Used for display purposes
    - **Grouping:** Used to group the existing variables
    - **LoadingBehaviour:**
    - **Layout:**
    - **Transformation:** Any transformations if needed
    - **ComparisonType:**
    - **Menu:**
    - **Process:** Assigns the report to a business process
    - **Chart:** Used to define the chart and its type for data visualization purposes

2. **ReportVariableDefinition**
    - **SystemName:** Used as an identifier
    - **Type:** Type, can be a primitive type (as double) or a predefined one (e.g. some registered type in our DSL)
    - **Formula:**
    - **ReportDefinition-Variables:** Assigns the variable to a specific report

3. **ReportRowDefinition**
    - **SystemName:** Used as an identifier
    - **DisplayName:** Used for display purposes
    - **Style:**
    - **RowType:**
    - **Order:** number, sets the order of the row
    - **Title:**
    - **ReportDefinition-Rows:** Assigns the row to a specific report
    - **Transformation:**
    - **Parent:**

4. **ReportColumnDefinition**
    - **SystemName:** Used as an identifier
    - **DisplayName:** Used for display purposes
    - **Width:** Used for display purposes, i.e. the width of the column
    - **Style:**
    - **Display:**
    - **IsPinned:**
    - **Value:**
    - **Order:** number, sets the order of the column
    - **ReportDefinition-Columns:** Assigns the column to a specific report
    - **IsGrouped:**

The reports are built dynamically, with the help of the data model and the DSL created. Therefore, this allows us to progressively create the reports, giving flexibility to business users. For this purpose, we can manipulate the data before we display it by applying **mappings and transformations**. Due to the multi-dimensional aspect, the structure allows us to modify data while querying it. Hence, some calculations can also be done after applying the queries. In figure 26 below an example of this can be seen with regards to the Balance Sheet Report. The highlighted rows sum the necessary positions to make up Total Assets and Total Liabilities. This is done through summing the positions while the report is being queried. Thus, the Balance Sheet Report in the application will display data that has already been manipulated during querying with DSL.

SystemName	DisplayName	DisplayOrd	Formula
BA	Bank Accounts	1	Position("BA")
Inv	Investments	2	Position("Inv")
Invtry	Inventories	3	Position("Invtry")
TrOthRec	Trade and Other Receivables	4	Position("TrOthRec")
PrepAndAccInc	Prepayments and Accrued Income	5	Position("PrepAndAccInc")
DerAssets	Derivative Assets	6	Position("DerAssets")
CurIncTaxAssets	Current Income Tax Assets	7	Position("CurIncTaxAssets")
AssetsHeldForSale	Assets Held for Sale	8	Position("AssetsHeldForSale")
TotCurAssets	Total Current Assets	9	BA+Inv+Invtry+TrOthRec+PrepAndAccInc+DerAssets+CurIncTaxAssets+AssetsHeldForSale
PropPlantAndEquip	Property, Plant and Equipment	10	Position("PropPlantAndEquip")
Goodwill	Goodwill	11	Position("Goodwill")
IntAssets	Intangible Assets	12	Position("IntAssets")
InvInAssocJointVent	Investments in Associates and Joint	13	Position("InvInAssocJointVent")
FinAssets	Financial Assets	14	Position("FinAssets")
EmpBenAssets	Employee Benefits Assets	15	Position("EmpBenAssets")
DefTaxAssets	Deferred Tax Assets	16	Position("DefTaxAssets")
TotNonCurAssets	Total Non-Current Assets	17	PropPlantAndEquip+Goodwill+IntAssets+InvInAssocJointVent+FinAssets+EmpBenAssets+DefTaxAssets
FinDebt	Financial Debt	18	Position("FinDebt")
TrOthPayables	Trade and Other Payables	19	Position("TrOthPayables")
AccAndDefIncome	Accruals and deferred income	20	Position("AccAndDefIncome")
Prov	Provisions	21	Position("Prov")
DerLiab	Derivative Liabilities	22	Position("DerLiab")
CurIncTaxLiab	Current Income Tax Liabilities	23	Position("CurIncTaxLiab")
LiabAssocAssForSale	Liabilities Directly Associated with As	24	Position("LiabAssocAssForSale")
TotCurLiab	Total Current Liabilities	25	FinDebt+TrOthPayables+AccAndDefIncome+Prov+DerLiab+CurIncTaxLiab+LiabAssocAssForSale
EmpBenLiab	Employee Benefits Liabilities	27	Position("EmpBenLiab")
DefTaxLiab	Deferred Tax Liabilities	28	Position("DefTaxLiab")
OthPay	Other Payables	29	Position("DefTaxLiab")
TotNonCurLiab	Total Non-Current Liabilities	30	EmpBenLiab+DefTaxLiab+OthPay

Figure 25: Applying mappings and transformations using DSL

Some reports display the so-called **positions**. A position is nothing but a special kind of entity that can have multiple dimensions. This being a very powerful concept, since using the DSL, a position can be manipulated in many ways during the calculations or mappings. Positions can be used to make different kinds of mathematical calculations, such as addition. The result of the calculations done by positions is being held by another position, as explained in section 4.1 earlier. It is also possible to filter by dimension, e.g. taking only the necessary data to perform the desired calculations.

6. Evaluation

In the requirements section, six functional and three non-functional requirements have been laid out that were decided upon in the beginning of this project. In this section it will be explained whether or not the project has met the pre-defined requirements.

6.1 Data Integration from Different Sources

The first requirement was that business users should be able to integrate data from different sources in order to make the application useful in the actuarial domain. Business users can import data during the initialization of the web application after the application has been deployed, as seen in the figure 27 below. Therefore, this fundamental functional requirement has successfully been met.

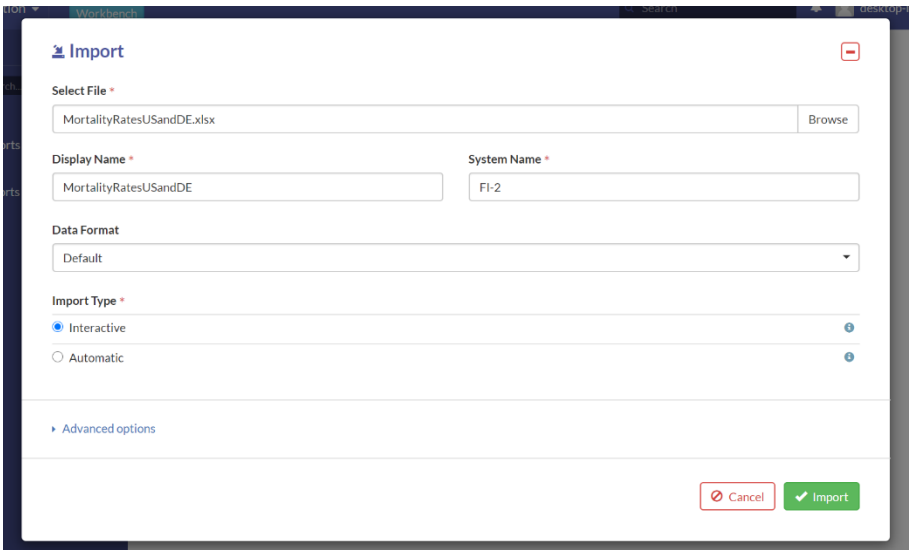


Figure 26: Example of importing data from an Excel file after the application has been launched

6.2 Data Validation during the Import of Data

The second requirement for the project was to ensure data quality during the importing process of data into the application. This is necessary such that the transformations during the import process lead to correct calculations. This requirement has been implemented and mostly coded with the DSL created for this project. Figure 28 below shows an example of a function written in DSL to validate the correctness of imported genders. In the domain of insurance companies, only two genders are accepted either male (M) or female (F). Thus, this example of such function checks whether the imported genders are empty, male or female.

```
2,True,ValidateGender,ImportState,DefaultStorage,"
    bool ValidateGender(IDataSet dataSet, string tableName)
    {
        string M = nameof(M);
        string F = nameof(F);
        string emptyString = """";

        bool isValid = true;
        var rows = dataSet.Tables[tableName].Rows.ToList();
        foreach(var row in rows)
        {
            if(!string.Equals(row[""Gender""]?.ToString(), F) && !string.Equals(row[""Gender""]?.ToString(), M) && !string.Equals(row[""Gender""]?.ToString(),
            emptyString) && !string.Equals(row[""Gender""]?.ToString(), null))
            {
                Error(new ImportError($""Invalid Import for gender. {row[""Gender""]?.ToString()} is not a valid gender.""));
                isValid = false;
            }
        }

        return isValid;
    }",
```

Figure 27: Example of data validation done using DSL

This functionality allows the web application to only accept imported files if they pass the user defined validation. We have written and successfully applied multiple import validation functions to fulfill this second functional requirement.

6.3 Inserting User Defined Business Rules

The third requirement specifies that business users should be able to insert their own business rules and logic. Our web application provides the users with the opportunity to create their own function definitions. Moreover, by leveraging the power of DSL in combination with Vertex platform, the DSL will continually be compiled, while the application only needs to be deployed once. Hence, this functional requirement has not only been successfully but also efficiently met in the project. Users can insert new functions in addition to making changes to existing ones.

Figure 29 below shows a simple example of a new function defined by the business user to restrict the possible ages to a positive range of numbers. Each text entry block must be filled in by the user, once they press on OK, the function will be saved and integrated into the application. Again, showcasing that the application can integrate new changes without having to be deployed.

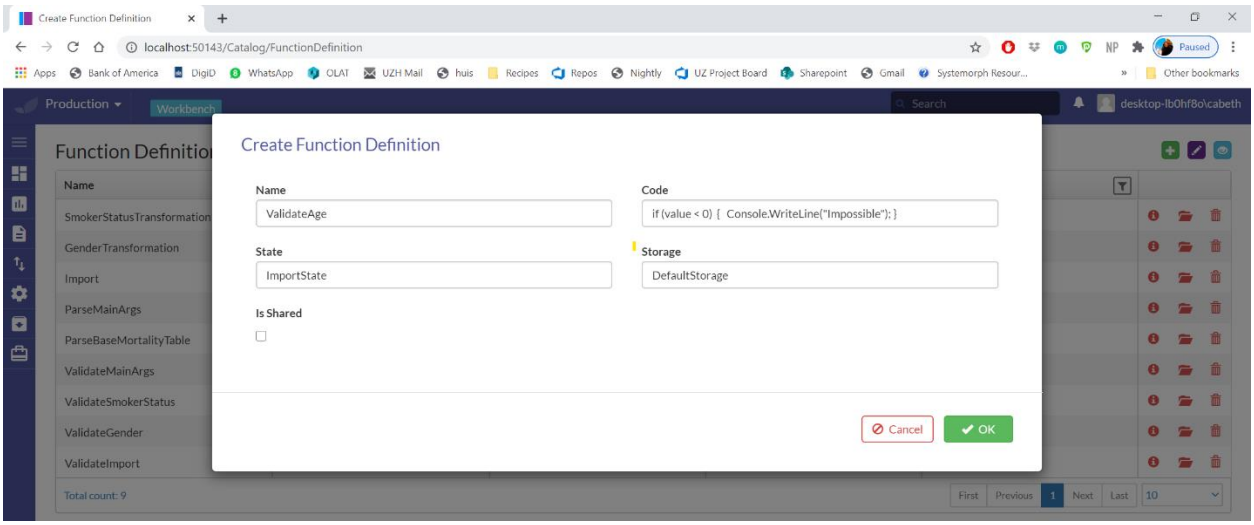


Figure 28: Example of the introduction of a simple user defined function into the application

Figure 30 below, shows how the business user is able to make changes to already existing functions. In this case, the function "ValidateGender" is shown. Once the user presses on the yellow highlighted edit button, they can make changes to the existing function.

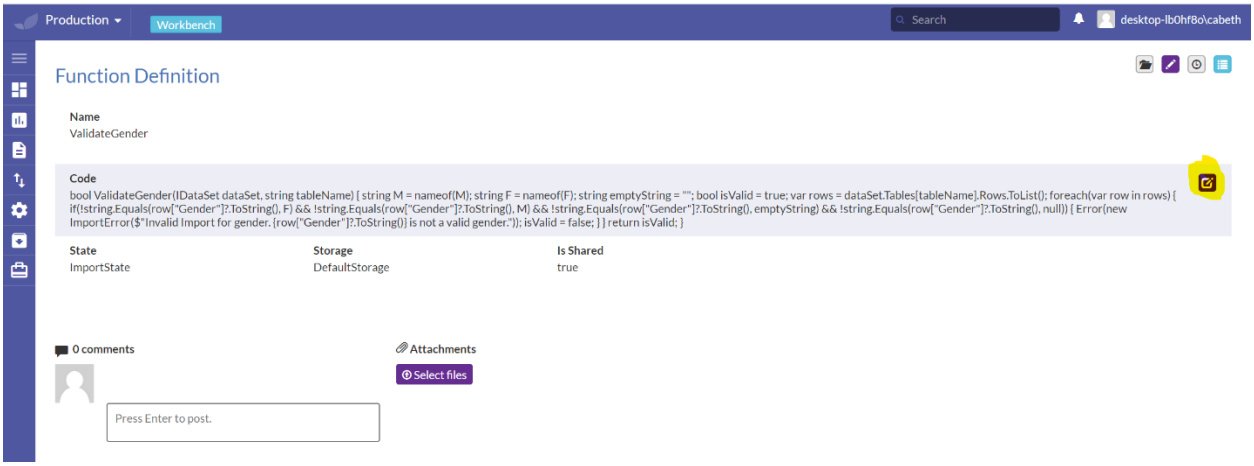


Figure 29: Example of how a user can make changes to already existing functions

Currently, one drawback of our application is that the viewing window is too small, as seen in figure 31 below. Resulting in the user having to copy paste the function into a text edit application such as Notepad to make the desired changes and then copy it back into the web application. This would've been something that we liked to change if more time was available.

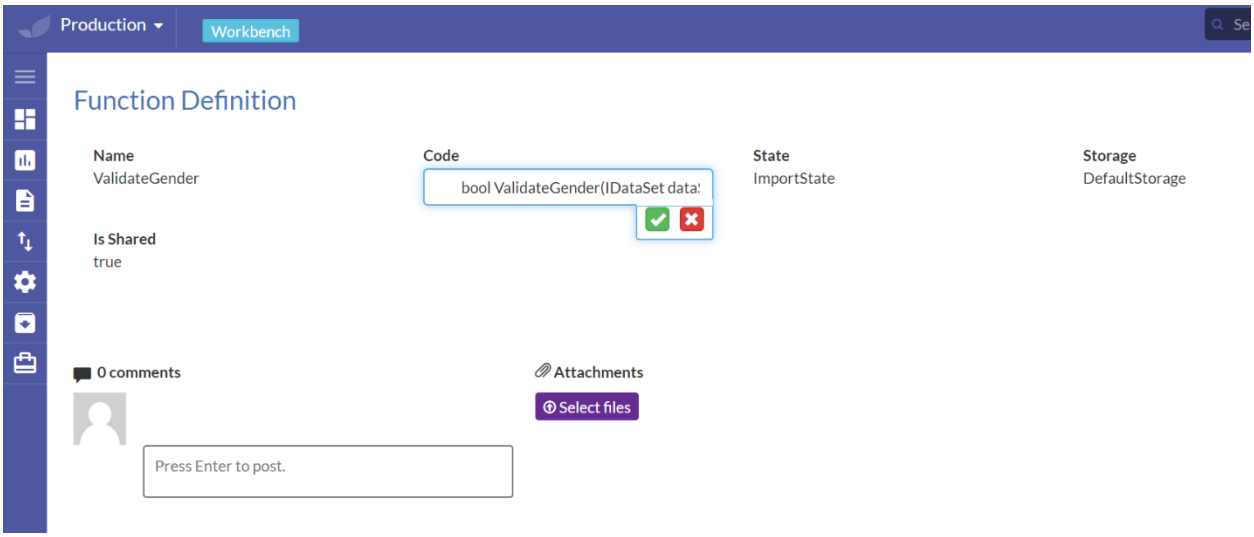


Figure 30: Example of how the Code window is too small to edit directly in the application.

Further below, in figure 32, it is shown how the registered variable values used in the business logic can be changed directly by the user. This leads to a manipulation of the calculation and will affect the reports. Here, the user changes a threshold of the Base Mortality Table that represents the age of women at which risk increases, which currently is set to 45.

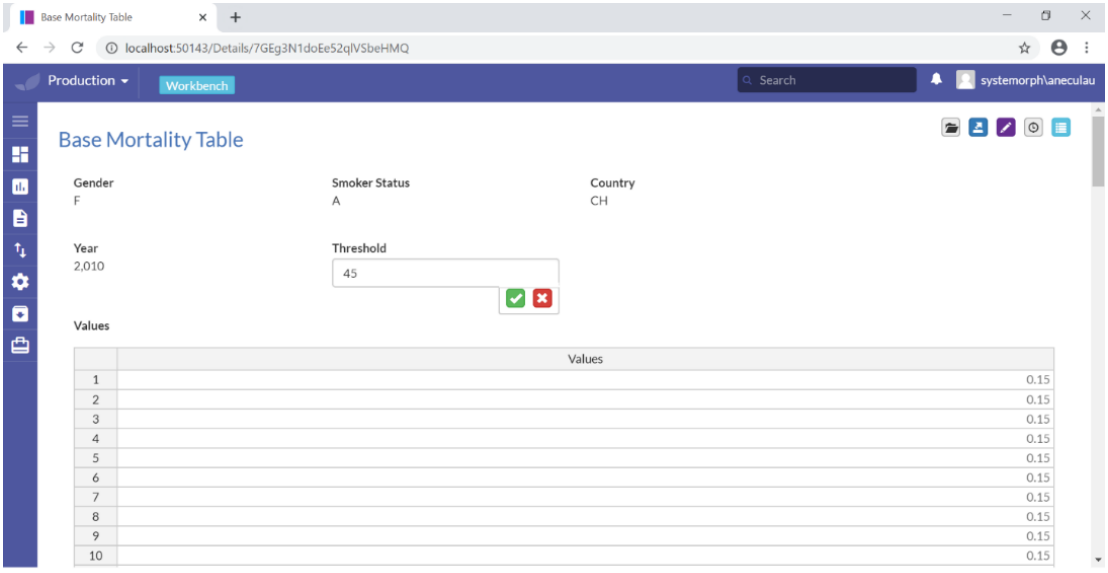


Figure 31: Example of how the user can change registered variables in the application, such as the threshold

6.4 Displaying Reports

The fourth requirement establishes that business users should be able to view their data in the form of numerical reports, but also in the form of graphs to have a visual aid in their understanding of the data. The web application provides reports for the data imported, such as the business sheet, mortality factors and the mortality table report. In figure 33 below, an example is provided for the mortality table report, along with a graph that gives a visual aid to the business user.

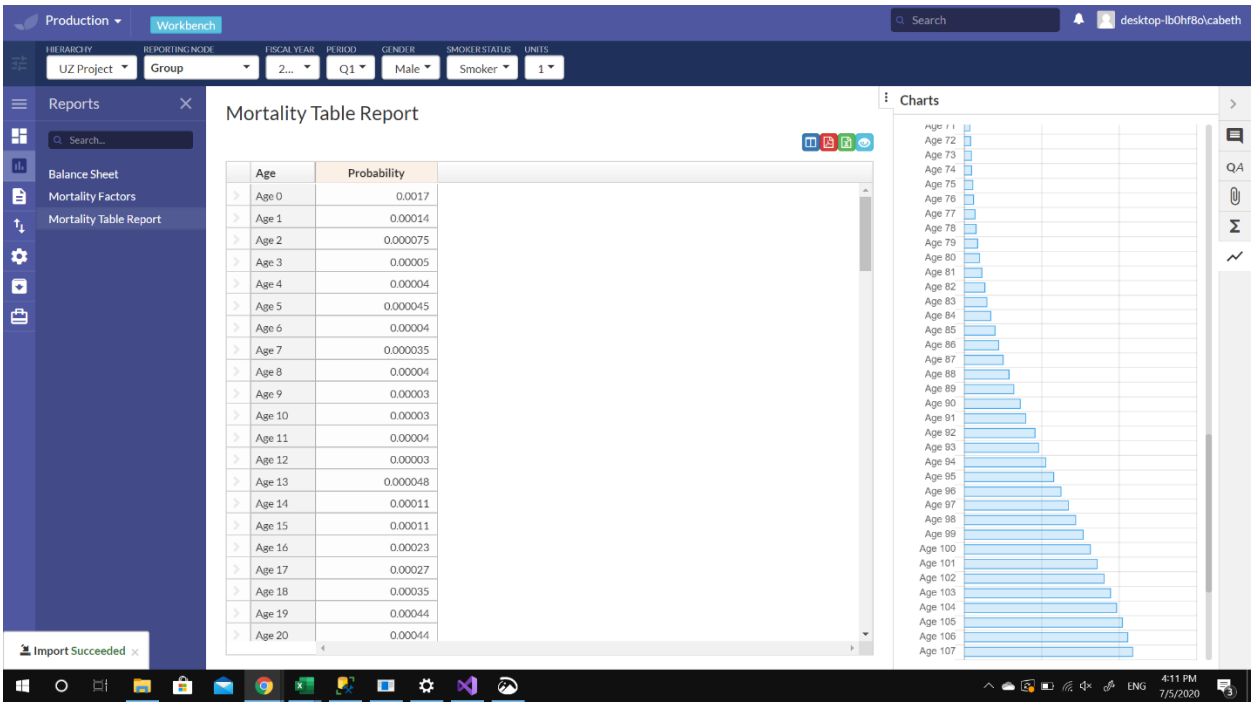


Figure 32: Example of the Mortality Table Report

We fully satisfy this functional goal as our applications provides two visualizations as initially planned, namely a life expectancy curve plus a balance sheet as snapshot overview of actuarial data. Moreover, we further provide a visualization of the relevant company-specific mortality factors which are based on distinct business rules. Thus, we can conclude that not only did we manage to suffice the prerequisite, but we also add valuable information to the application by introducing the notion of mortality factors and by allowing the user to adjust these mortality factors in accordance to their company ruling.

6.5 Business Process Designs to Ensure Security During Import

The fifth requirement relates to business process designs that allow for the security of the import process during the application. Different users have different rights with respect to data editing and administrative capabilities. In figure 34 below, three types of users can be seen, namely Admin, DataAdmin and DataViewer. The admin has the rights to change user rights for different users, being able to give them either the DataAdmin or DataViewer right. The DataAdmin is allowed to make changes to the data, formulas and introduce new formulas. The DataViewer is only allowed to view and export data, but not to make any changes to it.

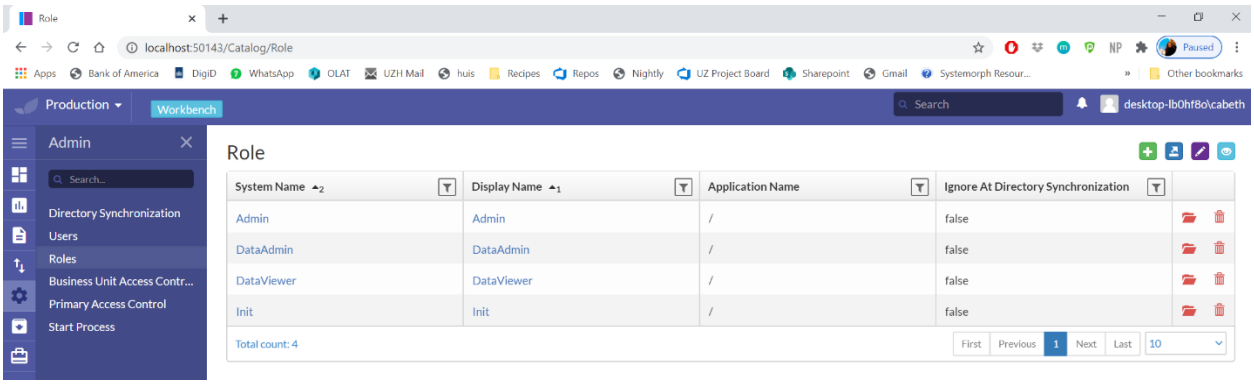


Figure 33: Different types of users with different editing rights

To further fulfill this functional requirement the application also has the capability of a versioning system as seen in figure 35 and 36 below. The user has the possibility to start a new process in order to introduce changes to his/her version of the production branch.

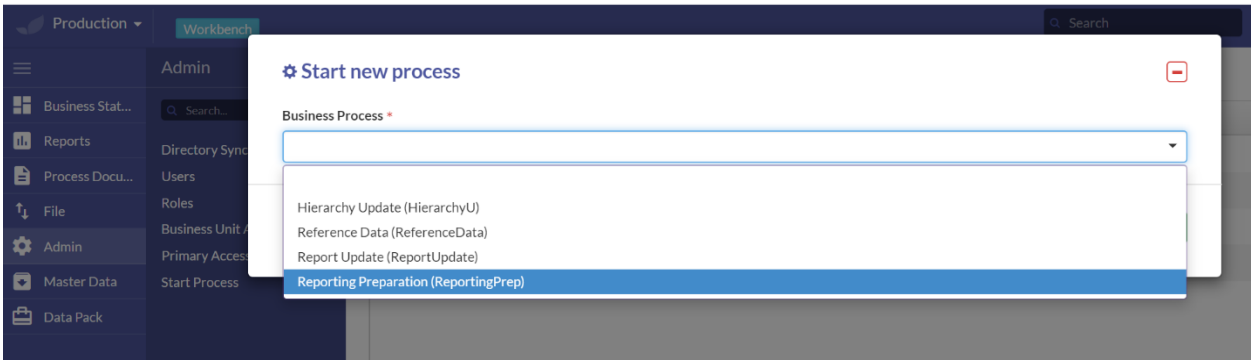


Figure 34: Starting new business processes to allow for changes in the user's version of production

Once this is done a separate branch is created. This version must be reviewed and completed before it can be merged into the production branch, named “Default”. This allows for a secure data editing process to minimize mistakes and keep a stable version of the data in the system.

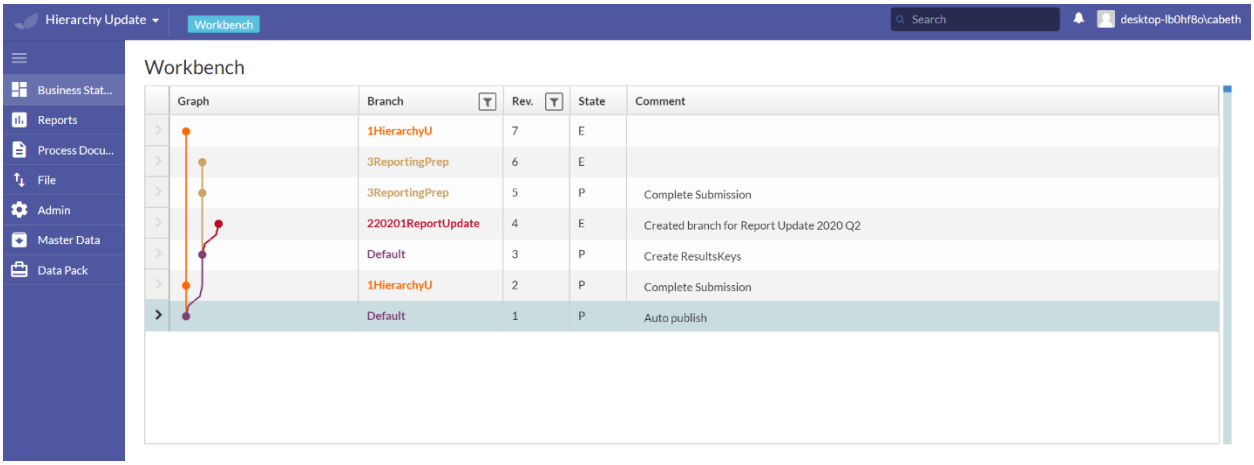


Figure 35: Example of the versioning and branching in the application

6.6 Unit Tests and Integration Testing

The sixth functional requirement relates to testing. Since business users have the capability to import data and add functions and edit existing functions in the application it is essential that tests ensure accuracy and precision throughout the project. Furthermore, test-oriented development is crucial with regards to maintaining the application efficiently and sustainably in the long term.

Two types of tests were created in this project to fulfill this requirement, mainly unit test and validation tests. The difference between the two is that unit tests are written to investigate the expected output of our application. Whereas the purpose of validation tests is to check if inputted data conforms to predefined properties of the database. On this account we defined four unit-tests that monitor whether predefined transformation rules during import are working independently on inserted data. A unit test takes an input as given, as well as the output transformed by the application using the functionality to be tested, and compares the latter against the expected result or benchmark.

In comparison to validation tests for the importing process - which test scenarios such as the validity of data types of inputted data - Unit tests represent powerful methods to detect issues. We specifically also performed unit tests concerning the functionality of importers for the entities BaseMortalityTable and BalanceSheet. In other words, we created unit tests for several independent instances of the input file. In each of them, we examine whether the process has succeeded by checking for the status of activity plus the presence of inserted rows into the database. In the case of **BaseMortalityTableImporterTest** the unit tests are conducted for the following possible occurrences: full import, imports of only one attribute (either gender or smoker status) and a combination of all available attributes (gender, smoker status and income).

For example, the unit test *TestBaseMortalityTableImporterWithSmokerStatusesOnly* checks whether transformations of gender are correctly performed if a file that only contains smoker statuses were to be imported. Please refer to the following code snippet to see that the mortality rate is reduced by 0.1 for all cases where ‘Gender == “F”’. If the imported file does not contain gender, then for this reason our importer needs to perform transformations that automatically create corresponding rows to extend the dataset.

```
201     leAValueAt45 = baseMortalityTable.SingleOrDefault(x => x.Gender == "M" && x.SmokerStatus == "A");
202     .That(maleAValueAt45?.Values[45], Is.EqualTo(0.0003));
203
204
205     maleNSValueAt45 = baseMortalityTable.SingleOrDefault(x => x.Gender == "F" && x.SmokerStatus == "NS");
206     .That(femaleNSValueAt45?.Values[45], Is.EqualTo(0.0003-0.1d));
207
208
209     maleSValueAt45 = baseMortalityTable.SingleOrDefault(x => x.Gender == "F" && x.SmokerStatus == "S");
210     .That(femaleSValueAt45?.Values[45], Is.EqualTo(0.0003-0.1d));
211
212
213     maleAValueAt45 = baseMortalityTable.SingleOrDefault(x => x.Gender == "F" && x.SmokerStatus == "A");
214     .That(femaleAValueAt45?.Values[45], Is.EqualTo(0.0003-0.1d));
```

Figure 36: Example of a unit test where gender transformations are checked

6.7 Utility gain for administrators and business users

In this section we will elaborate on the utility gain our application delivers to our two target groups, namely administrators and business users. For this, specifically the DSL will be put in comparison to its counterpart, the ETL. The ETL entails many instances and changes can only be deployed into production after going through the inspection of the IT department. The DSL however offers versions where the user can implement the test instance changes on the production machines. A versioned application allows several users to work on one solution simultaneously and independently by having their own versions merged to the production branch after individual modifications. Not only does this feature allow people to incorporate changes in form of code by themselves but this also ensures that possible mistakes are not affecting the production machine or versions of other users, i.e. ensuring correct and controllable modifications.

Thus, the non-functional requirement that requires the production environment to be transparent and handled separately for the users is entirely fulfilled as the web application offers versions of test instances. Moreover, the opportunity to specify own function definitions to make changes to the database effective by means of DSL represents a simple and versatile toolbox of pre-existing and modifiable registered functions for users without much technical knowledge. We would further like to mention an additional utility gain that though our web application with the DSL will take up a longer waiting time for single business users, compared to similar but conventional appliances with ETL, the total waiting time can be assumed to be shorter for business users will not have to wait for fixed changes of other users. Hence, there might be a performance loss on a single user level but at the same time this will easily be compensated by the positive externalities on a global level as well as in the long run. Furthermore, the flexibility of our web application is ensured by the versioning feature while its versatility is dependent on and thus directly reflected by the expertise of the business users.

6.8 Advantages and Disadvantages

In this application there are two main points that provide considerable advantages, but at the same time with a slight disadvantage that needs to be taken into account. Mainly the constant compilation of user implemented DSL in the background that affects performance and the empowerment of the business user through DSL by allowing for instant changes and introduction of new formulas.

Having the application run and compile on the fly affects the performance. Here, we see a tradeoff that is influenced by the size of the data that is being used. Once the data reaches an enormous size, such as one million rows, the application would experience performance issues. On one hand, we allow the user to directly implement new changes without having to redeploy the application, which gives great benefits. Having to wait for a new version of the program could take months. However, the constant compilation that is necessary to instantly implement these user changes comes with some waiting time. With a few thousand rows this waiting time is around ten seconds. Once the data reaches more than a million rows this waiting time can grow up to a few minutes. If the user implements one change a day, waiting a few minutes would not have a considerate effect. As in comparison, it is a few minutes per day versus a few months. But if the user is implementing many changes throughout the day, the waiting time would be a serious disadvantage for the application and go against the desired user flexibility.

A main advantage of the application is that the business user is empowered by being able to introduce changes into the application themselves. Thereby circumventing the typical approach where desired changes would have to be made by an IT professional. In our application the user is able to introduce these changes through the DSL. This however comes with two disadvantages; the user has to become familiar with DSL and the introduction of errors by the user. Since the DSL is written for the domain of the insurance company, the language should be close enough that the user is able to learn and implement it in a short timeframe. By the introduction of DSL and giving the user the ability to interact we shift the responsibility towards the user. We now face the slight disadvantage that each user is responsible for their own data and need to claim responsibility for their mistakes. Previously, this was all done by the IT department, and the responsibility could be placed away from the user. To address this disadvantage the application has a versioning system, where each user has their own version of the data.

Lastly, we would like to perform a little thought experiment to emphasize how our web application in combination with our six defined Functional requirements would perform compared to a standard data management and visualization application. Assume, we have 1 Mio formulas that are affected by the same change done by a user. The total compilation time would be calculated as the multiplication of number of rows by the time to compile one of such formula (i.e., 1 millisecond times 1 Mio formulas = 1000s = 16.67 min). The disadvantage would be that the business user who did the change has to wait 16.67 min such that change becomes effective. The solution to circumvent this problem of long waiting time or ways to improve the drawback is to recompile only if changes are made, i.e. keeping compilations of formula in memory. This refers to “lazy loading” which describes that only the minimal amount of formulas required will be computed. Once modifications have been made, user closes process. In the background the formula will be compiled. One business user indeed has to wait 16 minutes until the change has been executed on the production environment. But the first other user to see the changes only has to wait 15 milliseconds (normal waiting time) instead of the 16 minutes as modifications have already been compiled and stored in database, and can be used many times. Hence, waiting 16 minutes is better than waiting 1 month for the changes to be implemented by the IT department. Let’s extend this thought experiment.

Assume, we now have 100 Mio rows (instead of 1 Mio, meaning waiting time is approximately 100 times worse than before). The calculation would look as follows: 100’000’000 / (1000s *3660) = 27.78 h, which is more than a day and thus still better than waiting for a month. We could claim that the advantage would be that our solution is globally efficient and that the probability of unintended implementations decreases as the user that wants to make changes will effectively be able to do so without any third parties.

7. Conclusion & Outlook

Insurance companies are forced regulators to create reports to make their operational business transparent to the public. Our master project tackles this very task and delivers an interactive web application that imports and manages data with an unconventional and state of the art solution. Instead of using the prevailing Extract-Transform-Load (ETL) process we introduced the Domain Specific Language (DSL). This program code is purposely produced to generate an abstracted model of one specific area of interest, namely the domain, which in our case is exemplified by the insurance industry. The aim of using the DSL in our reporting application is to reduce the level of ambiguity to such a degree that a user with only the relevant business knowledge about the corresponding domain can operate the reporting tool and introduce new business logic without much technical know-how. Furthermore, the implementation of DSL allows for the users to circumvent the traditional waiting time associated with implementing new changes in the ETL process. Traditionally the users would have to wait for a deployment of the program containing the new business logic, however with DSL the new business logic is continuously compiled without having to be re-deployed.

Our web application encompasses the indispensable import functionality to store data, that can be factual company figures or external sources utilized for data enrichment. Besides that, we empower the business users with the right to make effective and immediate changes on the production machines when generating the reports. The assignment of business roles thereby guarantees data security on import and report level. For this, we also created business processes that ensure that modifications can be kept under control and do not impact the production environment until being approved on the testing instance. This multiversion concurrency control property has a supplementary advantage of allowing several business users to work independently on their individual data versions leading to a wider range of possible perspectives and modifications of the data.

Furthermore, the users are provided with the ability to configure the set of numerical facts according to their personal understanding, in particular by means of pre-defined but alterable calculation transformations as well as the option to create function definitions with the declarative and less technical DSL. However, the freedom granted by these powerful tools bring along the penalty of possible long waiting times that can occur if the data takes on a massive size. This is due to the fact that code needs to be continually compiled in the background. But for all that, it can be claimed that the overall development productivity of solutions is more efficient and sustainable.

In our Master project we have successfully developed a modern and adaptable alternative for a data management and visualization engine. Though our web application is only limited to serve a specific domain it potentially can manifest in boundless usage possibilities contingent on the applied business logics of the user. In combination to this, the versioned application reinforces but at the same time controls the influence of the user. Hence, the flexibility and manifoldness of implementation opportunities with our reporting tool gives rise to a multitude of other compelling prospect applications.

If more time were available, we would have loved to create more reports that contain more advanced visualizations. Furthermore, it would’ve been extremely interesting to see at which size of the data the tradeoff begins to weigh heavily between user empowerment and the application’s performance. Down the line we see that the solution provided by Systemorph allows business users to be more

technically empowered and claim more responsibility for their actions. Seeing as with the Systemorph platform, the desired data transformations are in the hand of the business users and not in the hands of the IT professionals. It will be interesting to see what outcomes this will provide for companies using the platform compared to companies who do not. We expect to see a more precise, and faster implementation of business logic since there will be less communication errors between the business users and the IT professionals.

8. References

[1] Systemorph Vertex 7.1 documentation (Pull request access contact Systemorph – www.systemorph.com)

[2] Systemorph Vertex tutorial (Pull request access contact Systemorph – www.systemorph.com)

[3] Educator: Dr. Pedro Fonseca

[4] Demo video: https://youtu.be/93ibnjQ5OUU

9. Appendix

In the appendix we have included the script for the demo performed.

Demo Summary / Script:

1. Interface
2. Hierarchy Update Business Process
3. Report Preparation Update Business Process (1/2) due to change in accounting logic
4. Transformations for data enrichment of Base Mortality Table
5. Report Preparation Update Business Process (1/2) due to change in actuarial logic
6. Import Validation
7. Business Process Validation
8. Report Update Business Process
9. Mortality Table Report

For the demo, we would like to go through our six functional requirements by showing them on concrete examples.

Six functional requirements

1) Data integration from Different Sources (Base Mortality Table, Balance Sheet, example xlsx)

2) Data Validation Using DSL (Base Mortality Table)

3) Data Transformation Using DSL (Base Mortality Table)

4) Generating reports using DSL (Mortality Table, Balance Sheet)

5) Business Process Designs to Ensure Governance

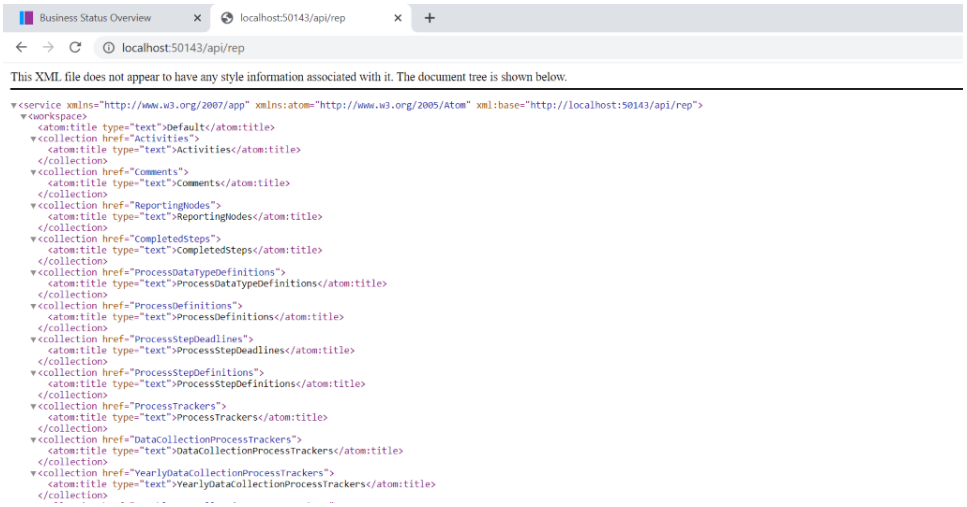
6) Unit Tests

The ultimate goal of our project was to import data and then produce reports using DSL. In the demo we create two reports, a mortality table report that the insurance industry can use, and a balance sheet report for accounting purposes.

What we are using is a web application and it's deployed locally (not on the cloud), thus for the demo only one person will be able to interact with the application. We interact with the application through the GUI. The browser itself is going to use ODATA to interact with the web application and the data. The data is available through the API OData interface.

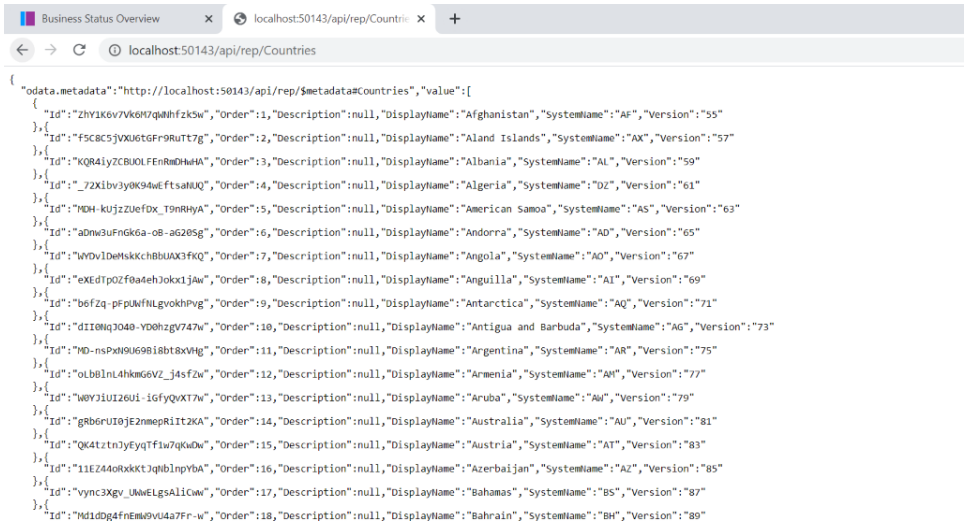
SHOW INTERFACE:

1. Data is available thru an API OData interface (http://localhost:50143/api/rep)

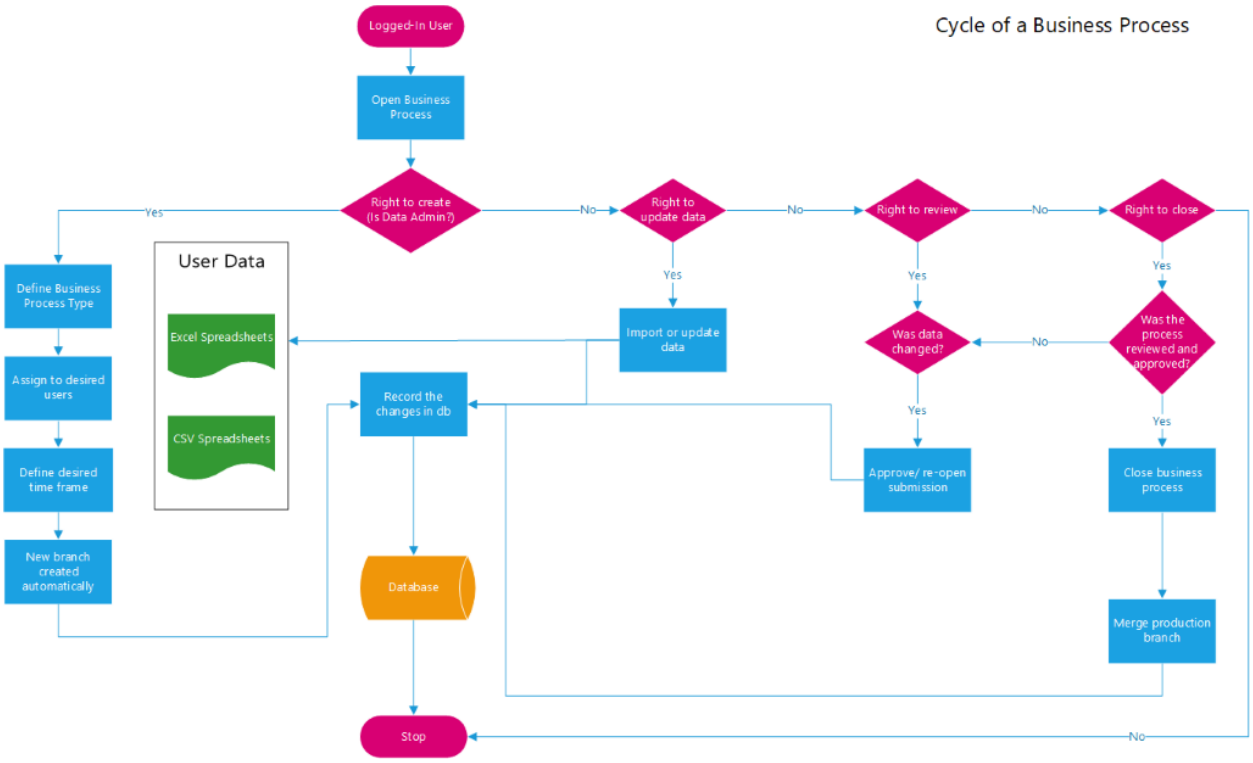


For instance, list of countries can be queried with http://localhost:50143/api/rep/Countries

The result is in JSON format.



We first start a business process to get access to our own version, and to make changes. Right now, we are still in the production version, where we can't make changes. Versioning is an essential part of the business processes to better manage and monitor changes of business users; we will show you more about this later in the demo. But essentially, we are creating a version of the application for the user, in which they can make changes that are only available to them, and not to all users.



*Thus, we have to **open a business process** in the beginning to be able to proceed with making changes.*

*There are **Simple Processes** – These are for all Reporting Nodes, all Years, all Quarters:*

- ***Hierarchy** (Reporting Node: Switzerland, Germany),*
- ***Reference Data** (Smoker Status),*
- ***Reporting Preparation** (Base Mortality Table)*

*And there are **Data Collection Process** – This is per Reporting Node (Switzerland, Germany, etc.), per Year, and per Quarter:*

- ***Report Update***

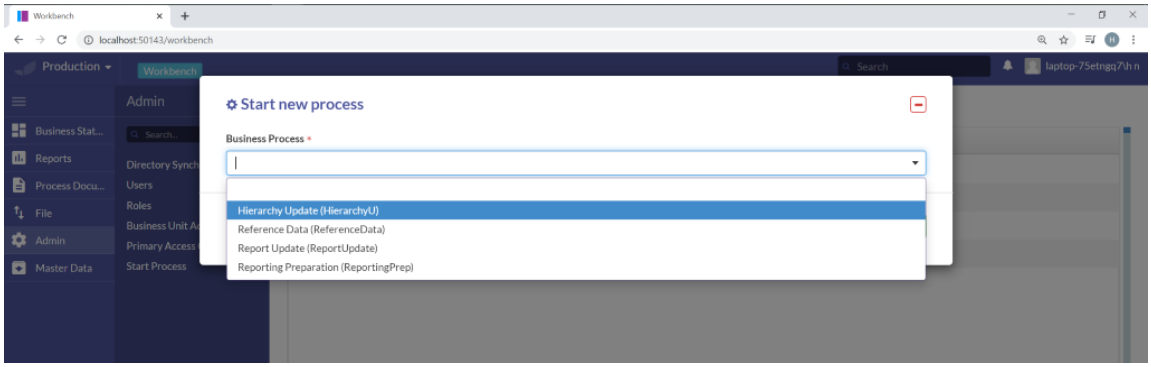
We will show the Hierarchy Update, Reporting Preparation and Report Update, where we first start with the former process.

We will start the demo by showing a hierarchy update process. With this we are able to update the different nodes of a company. For example, an insurance company based in Switzerland can have children companies in Germany and in France. The French company needs to be added as a new legal entity into the system, so here we will show how to do that.

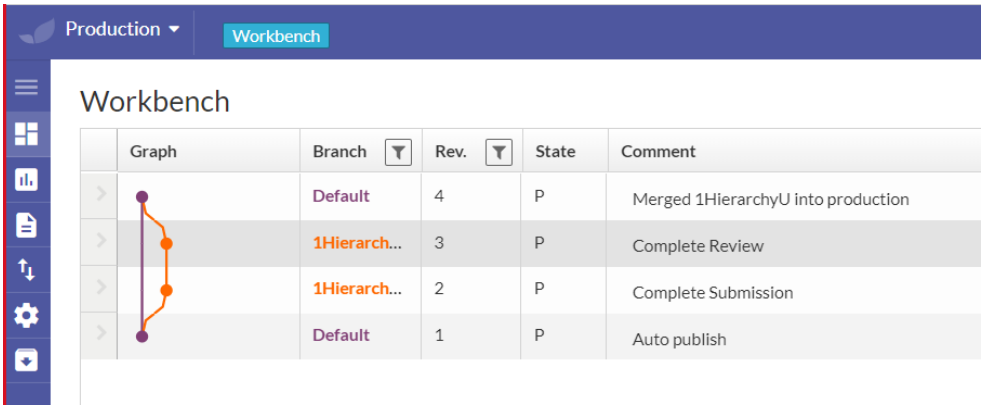
SHOW HIERARCHY UPDATE:

2. **Adding French company as new legal entity into system:**
   With DataAdmin role, start a Simple Process, e.g. the **Hierarchy**:

   (DataAdmin is an example of a Global Role. Another e.g. is Admin, who assigns roles)



a. Add a ReportingNode, e.g. France SA (FR) using the GUI – We just opened a branch in France.
b. Add FR to hierarchy (e.g. under Group)
c. Complete Submission => we are now in revision and Hierarchy Data cannot be edited. We can either Re-Open (and then it will be editable), or move on the process steps.



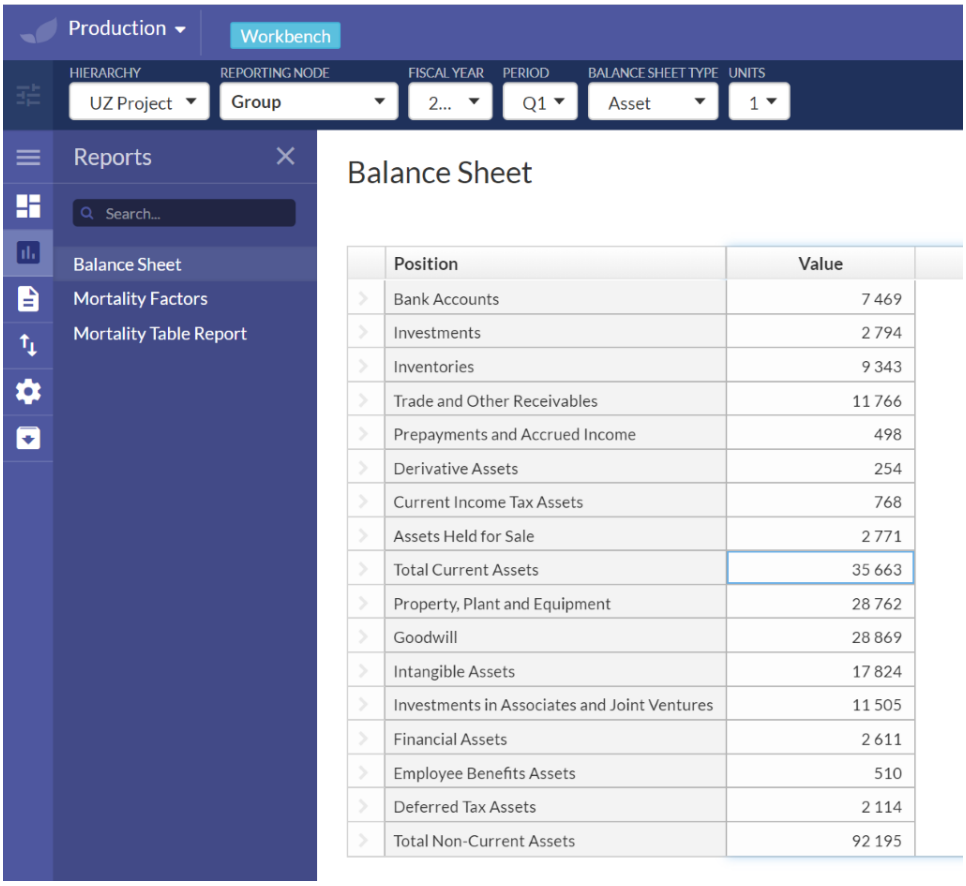
Now that the different business processes are clear using a simple example of adding a new child company to the parent, we will get into actually creating a report. During the initialization of the application we already imported some files, for the sake of the demo. But now some business rules changed, we want to showcase these changes in the reports to make them accurate. For this, we start a Reporting Preparation process, where we can adjust transformation and validation of the data using an extension of the DSL.

In this example, there is a change in the accounting logic, meaning that an accounting process might have changed and we need to rectify it.

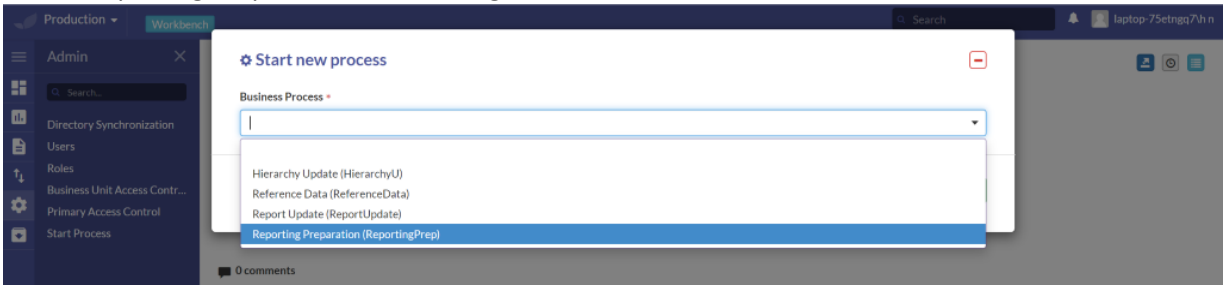
SHOW REPORTING PREPARATION UPDATE (1/2):

3. **Formula change in balance sheet report due to change in accounting logic:** accounting process has changed (either change operator or change sign before a number while operators remain)

a. Show BEFORE: Show number in Balance Sheet Report (in production engine)

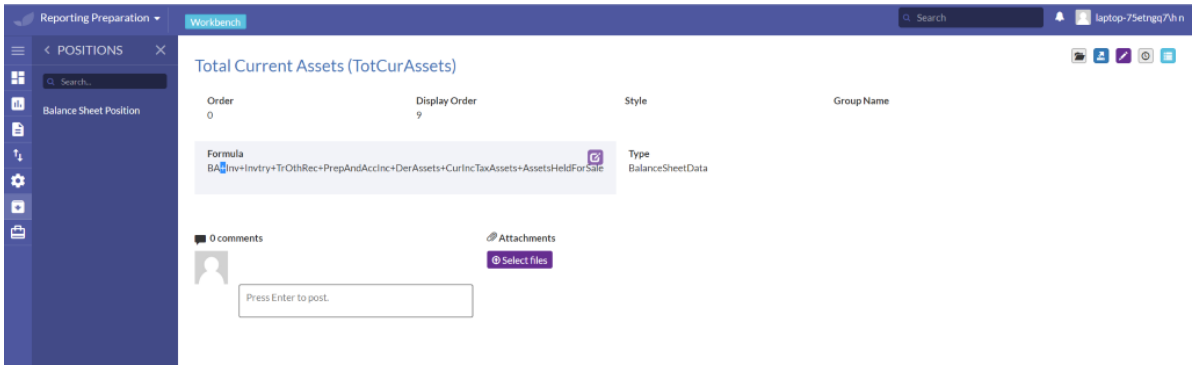


b. Start a Reporting Preparation Process to get own editable version.

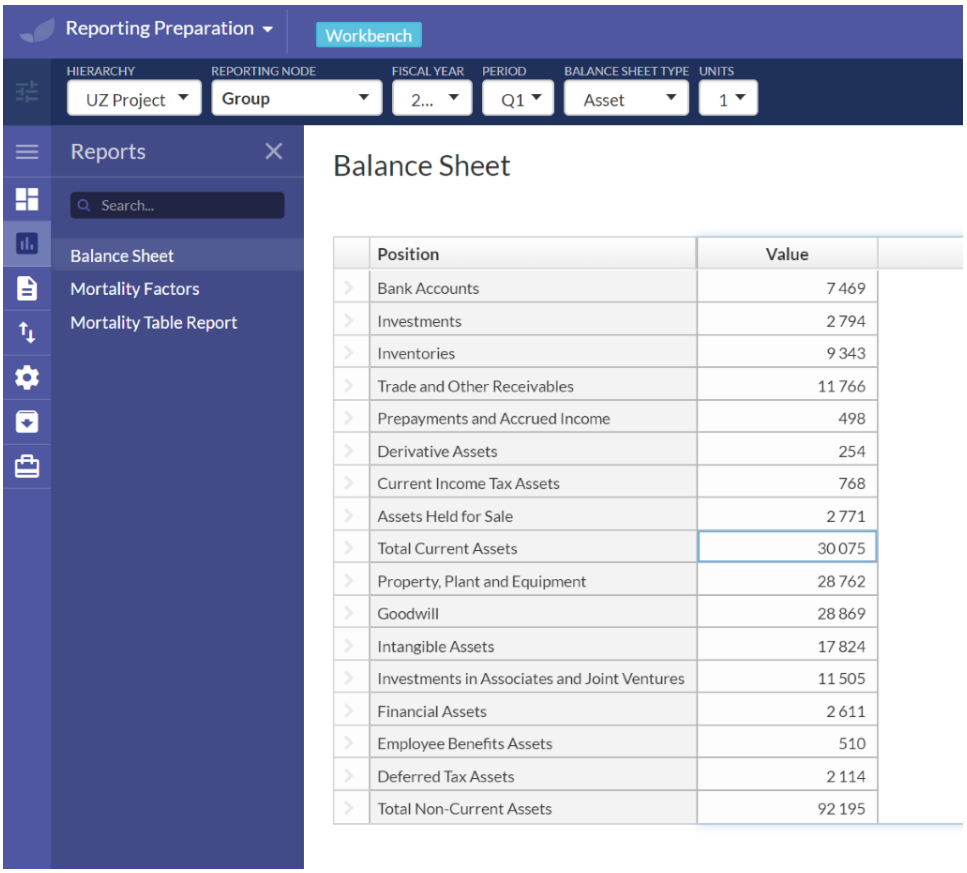


c. Master Data > Balance Sheet Positions > Total Current assets > click on correspdnding System Name

d. Edit the formula by changing the plus sign to a minus sign.

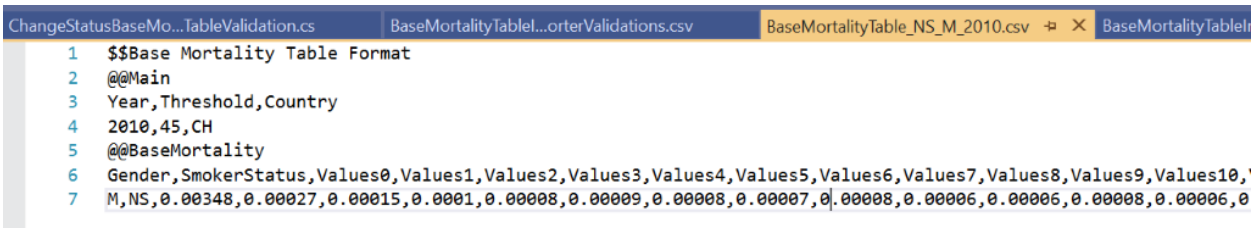


e. Compare updated changes in Report > Balance Sheet > Total Current Assets

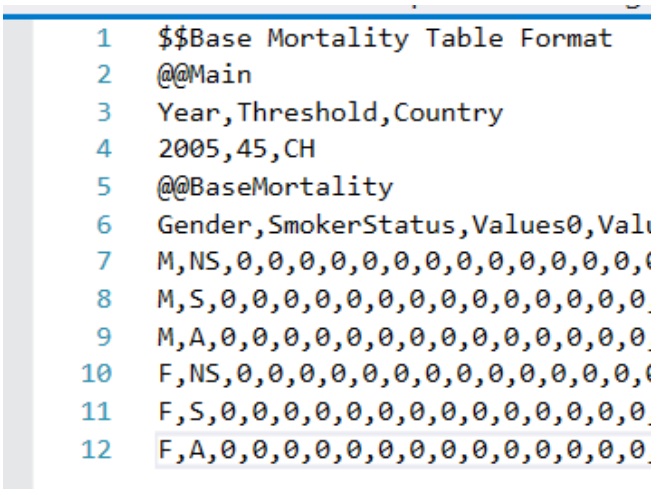


SHOW REPORTING PREPARATION UPDATE (2/2):

4. **Transformation implemented with DSL: Transformations** used to fill in missing data.



What we saw here is that we only had information for one type of person, a non-smoking male. Transformations are defined to fill in missing data. Here, we created 5 more rows based on the information of a non-smoking male. By using DSL, we were able to apply logic and transform the data from a non-smoking male into data for all categories. (smoking male, non-smoking female, etc.). If we had the data for all 6 types, we wouldn't need DSL to apply this transformation.



```
1   $$Base Mortality Table Format
2   @@Main
3   Year,Threshold,Country
4   2005,45,CH
5   @@BaseMortality
6   Gender,SmokerStatus,Values0,Valu
7   M,NS,0,0,0,0,0,0,0,0,0,0,0,0,0,(
8   M,S,0,0,0,0,0,0,0,0,0,0,0,0,0,0
9   M,A,0,0,0,0,0,0,0,0,0,0,0,0,0,0
10  F,NS,0,0,0,0,0,0,0,0,0,0,0,0,0,(
11  F,S,0,0,0,0,0,0,0,0,0,0,0,0,0,0
12  F,A,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

We have to re-import the file because the transformation and validation is already saved but becomes effective during the import. Note that, because we enriched our data set by means of transformation on one single row, all resulting rows will also be affected by the change in the rules. Hence, this automatic chain of transformation allows us to directly implement the change instead of waiting for the IT department to handle this matter. Transformation logic is also part of the data, i.e. change logic through GUI (takes 3 min, as opposed to a deployment which might take up to 3 month). Since rules change over time, the users, respectively, also receives the ability to use the extension of the DSL to implement new rules.

We will now show a change in the transformation during the import, which is an example of a change in the actuarial logic. We now want to increase the mortality rate of all non-smokers by for example 50% and show you how DSL written by us is used in action here.

SHOW REPORTING PREPARATION UPDATE (2/2):

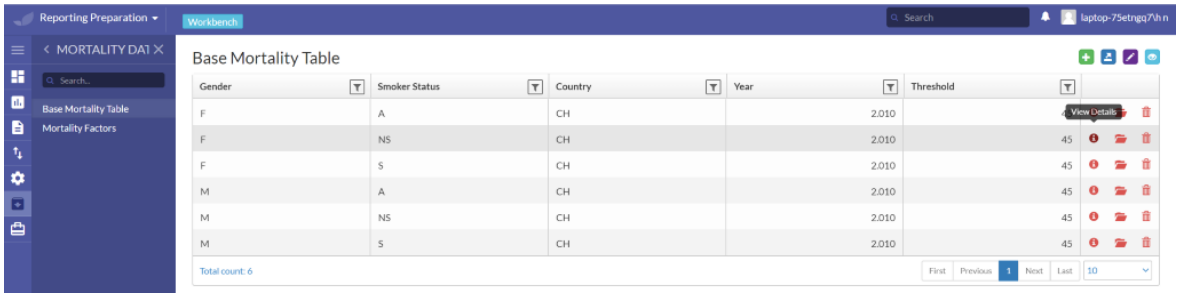
5. **Change in SmokerStatusTansformation formula in mortality table reports (BMT):** (e.g. add transformations for smokerstatus “NS” that increases the probability of dying)

   Change in actuarial logic: 1 row in BMT leads to 6 rows

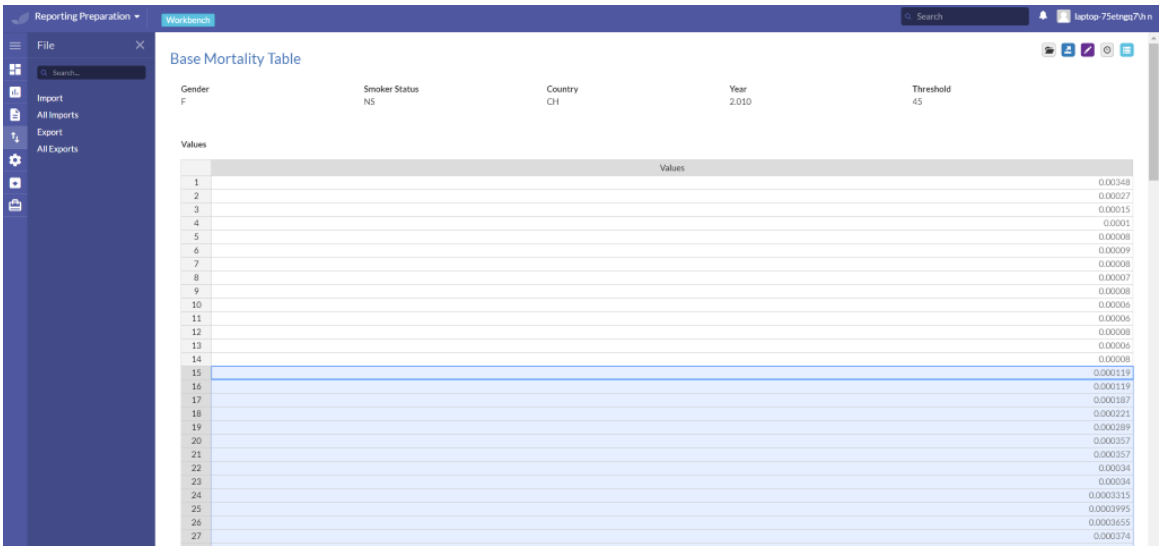
a. Again, start a Reporting Preparation Process (if completed submission before) or continue with the already opened Reporting Preparation Process from the example before. In the demo we closed submission to show the versioning property of our web application. In this example here, we continue with the Reporting Preparation process.

b. EXPORT Base Mortality table and show BEFORE by exporting the data for e.g. a non-smoking female.

   MasterData > Base Mortality Table > View Details

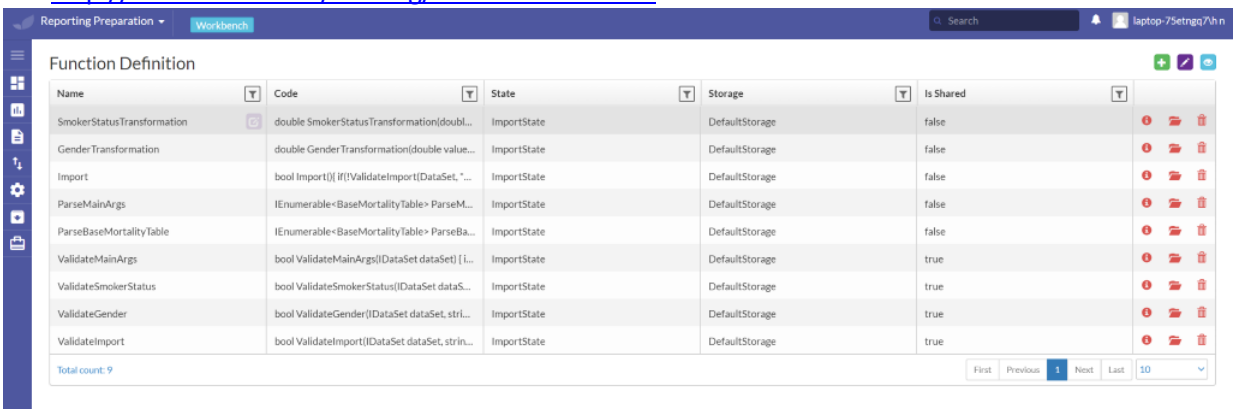


Values BEFORE change in actuarial logic:



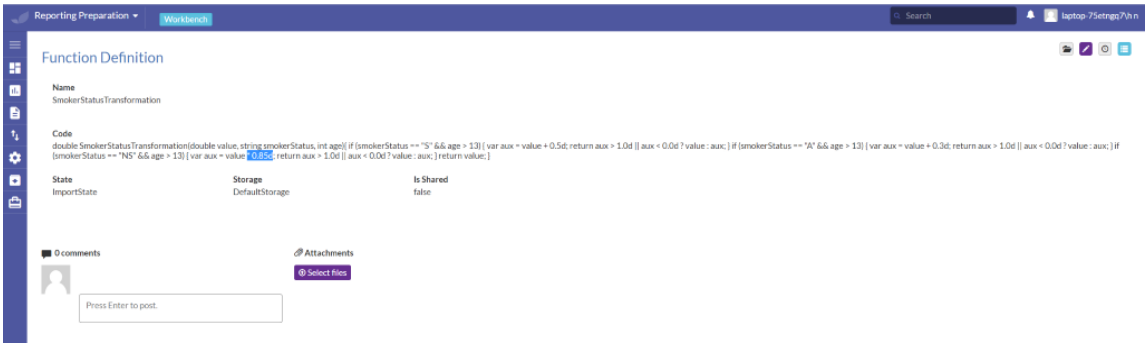
c. Catalog/FunctionDefinition > SmokerStatusTransformation > red info button

http://localhost:50143/Catalog/FunctionDefinition

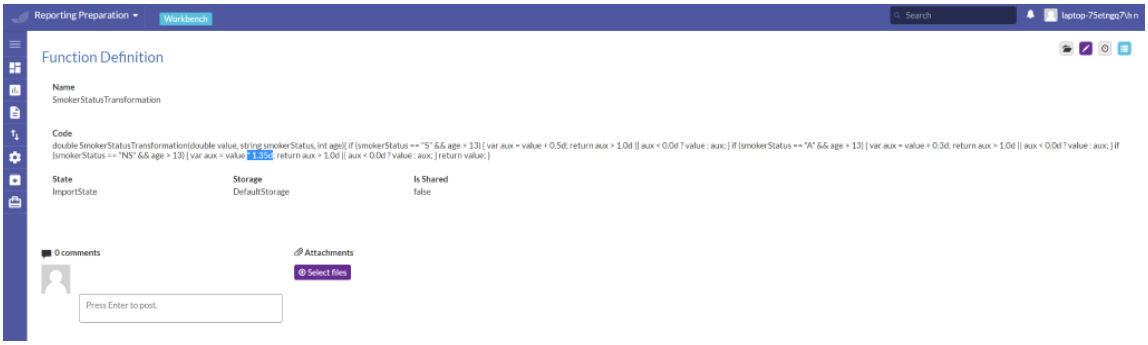


d. Edit formula: Increase mortality rate by 50% by changing factor from 0.85 to 1.35.

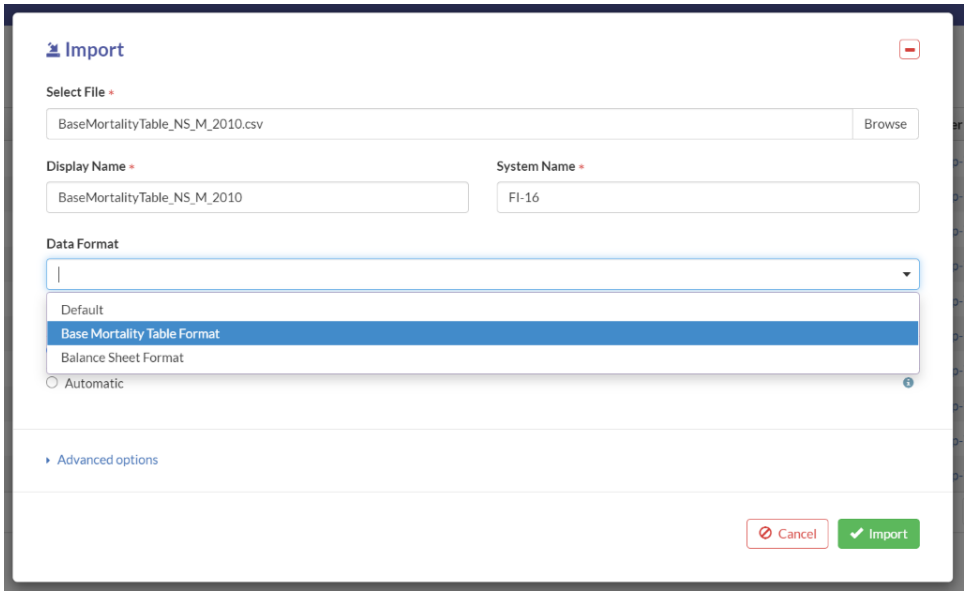
BEFORE editing the formula for SmokerStatusTransformation:



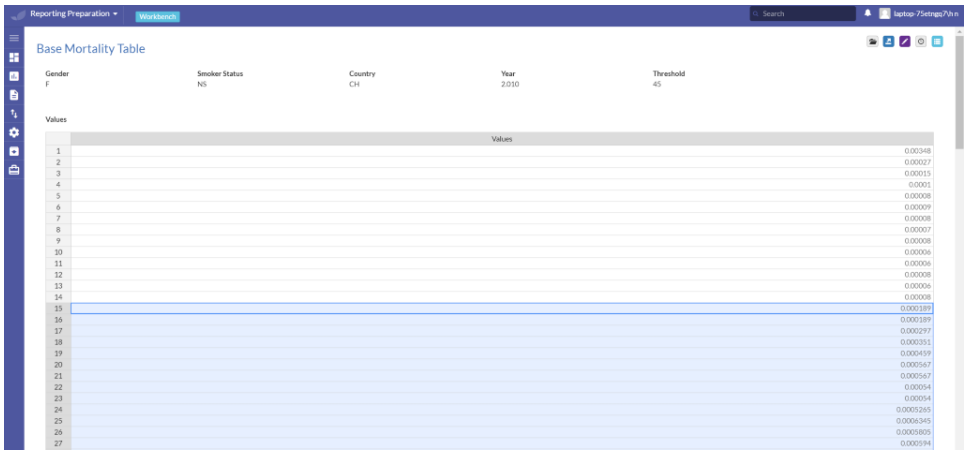
AFTER editing the formula for SmokerStatusTransformation:



e. Show EXPORT (AFTER version) (changes successfully displayed) (after being imported, as our transformation will only be effective during import)



Values AFTER change in actuarial logic:



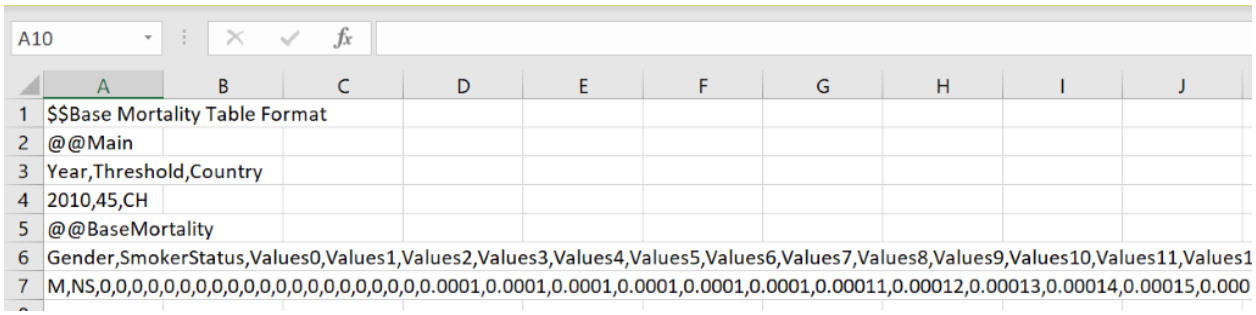
Indeed, mortality rates for all types of non-smokers have correctly been increased by 50% **after the age of 13 (13 excluded)**. **Additionally, note that** the values changed only after the 15th row (15th row included) because in our imported file, values are defined starting value 0 and not value 1 as in the GUI.

Now that we changed the logic, the base mortality report is updated with more accurate information and we can show you the changes. By the way, in theory, the base mortality and balance sheet report are actually connected, meaning that the mortality table will be used to generate numbers that show up in the balance sheet of the life-insurance company. Ultimately, the Balance Sheet will be the most important one.

The last but not least kind of change concerns the validation (change validation), meaning that changes, in the actuarial or accounting logic, are not permitted to be saved into the database due to circumstances that are prevented by our validation rules. Assume, we there is a change in the value of gender from "M" to for example "G".

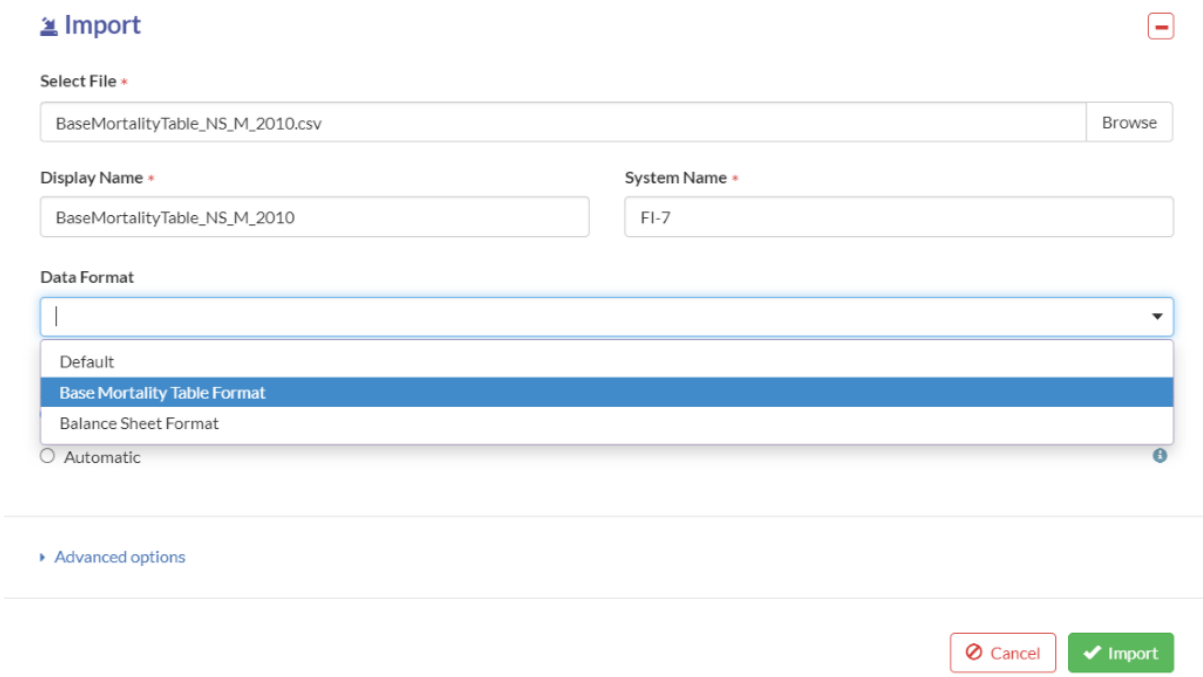
6. **Validation change in base mortality table** (e.g. change value of gender to a letter that is not "M" or "F"), s.t. user will receive an error message that states that file is not allowed to be imported.

a. BEFORE: in text editor show file to be imported

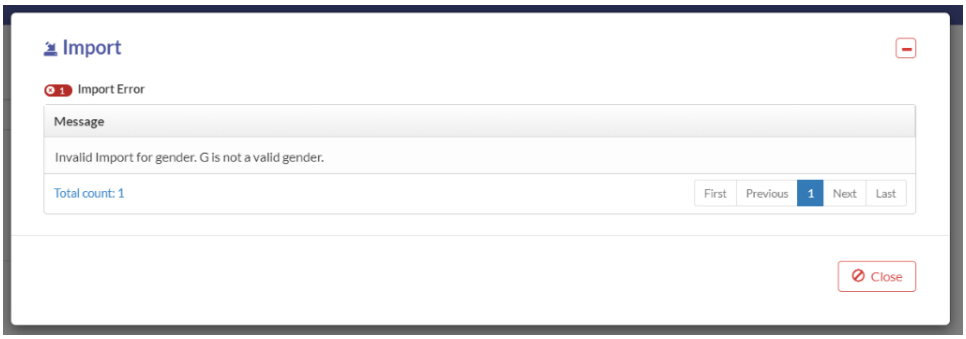


b. MAKE CHANGES: change gender “M” to e.g. “G” and SAVE.

c. IMPORT modified file



d. Error message:



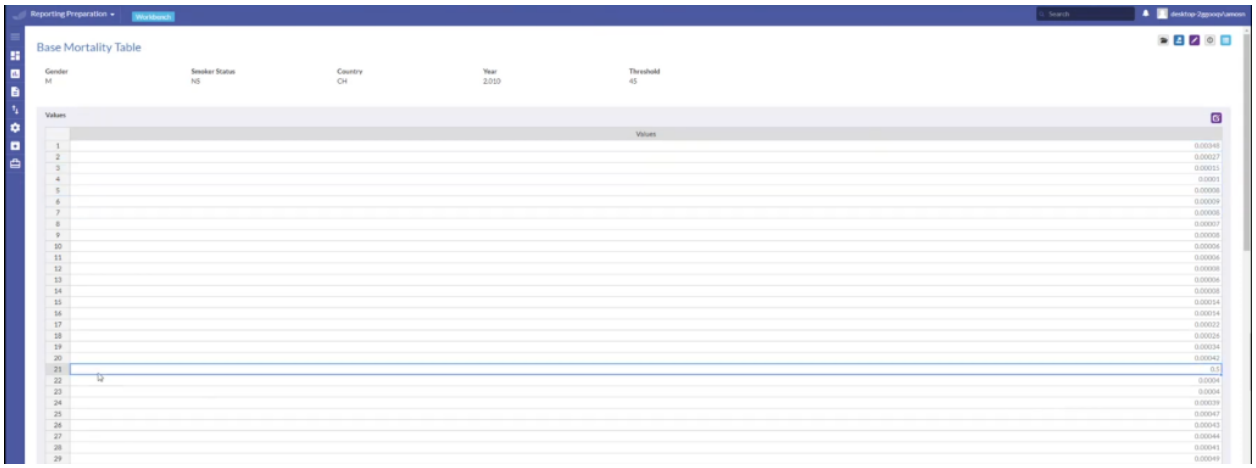
We have defined validation rules on two levels: on the import level (which we just showed you by the example of the GenderValidation rule) and the business process level.

In the example before we have seen that if the value of gender was valid, we successfully imported the base mortality table. Also, assume our mortality factors are also valid and thus can be imported correctly. Note that, the Mortality table is a multiplication of the base mortality table as well as the mortality factors. So, if the individual components have been validated during the import but the collection of these entities somehow does not satisfy some business logic, business users can business process validation rules that handle these errors.

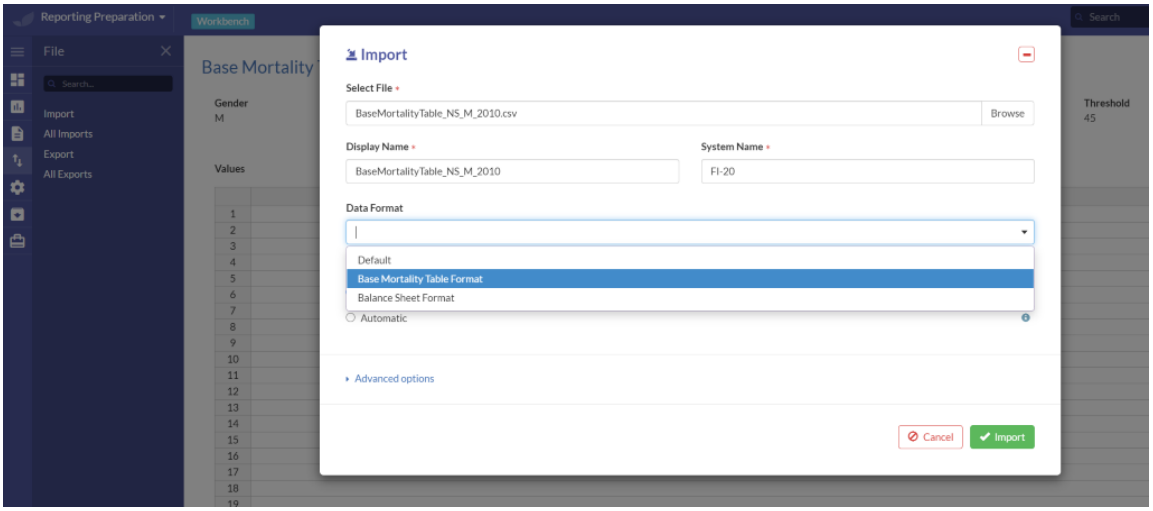
Show BUSINESS PROCESS VALIDATION:

7. **Business Process validation: In this concrete example, we do not allow probabilities that do not match our definitions, i.e. probability of dying must increase gradually when age increases. Or, the mortality rate at age 20 cannot be higher than at age 45.**

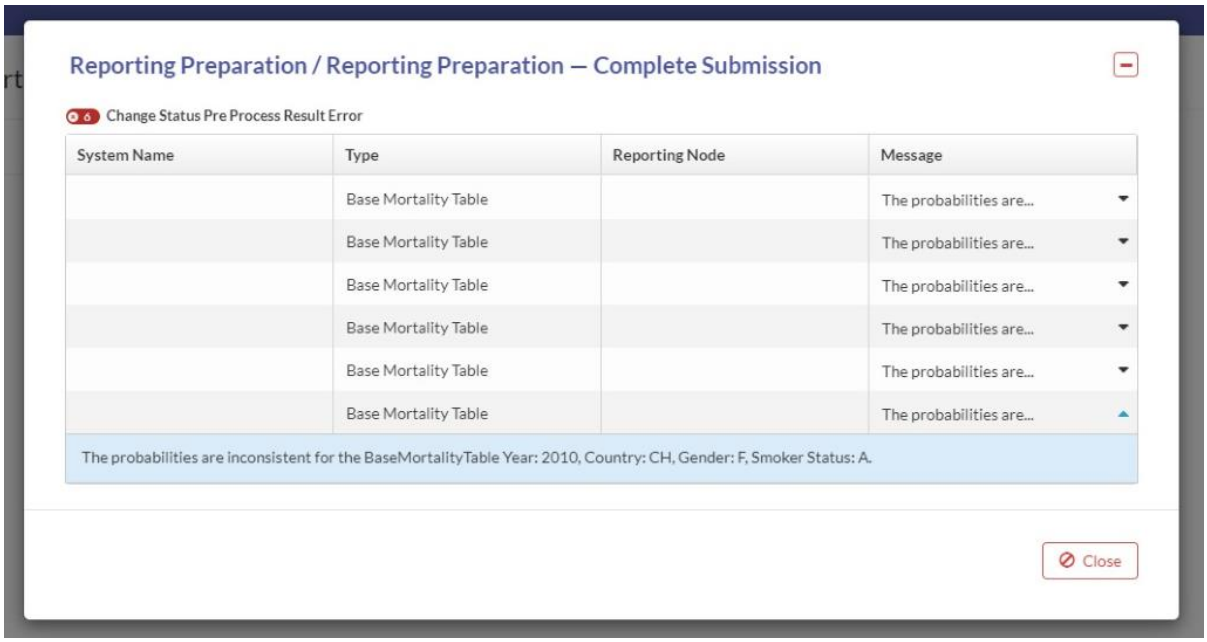
a. Identification that there is a spike in the probability.



b. Import file. Note that, there is no error message concerning a possible violation of import validation.



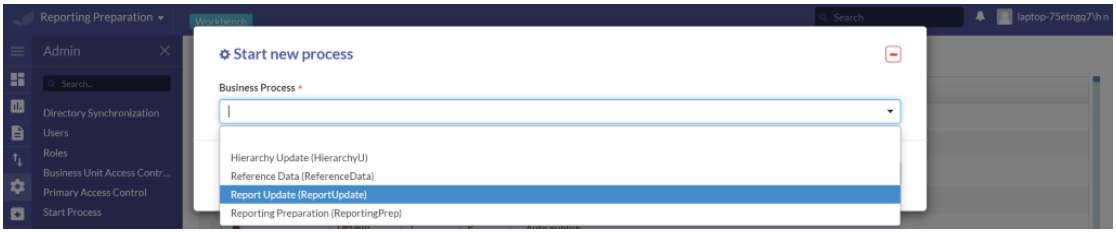
c. Completing submission of the Report Preparation Process triggers an error message and doesn't allow user to proceed before correcting the probabilities.



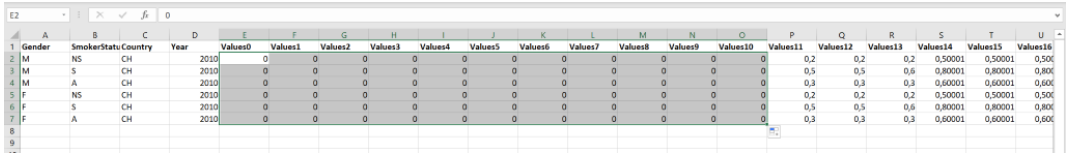
After all the necessary report preparations have been made, we will import the mortality factors to change its values that are used for the base mortality table. In fact, the base mortality table is calculated as the multiplication of the mortality table with some scaling factors, namely the mortality factors.

SHOW REPORT UPDATE:

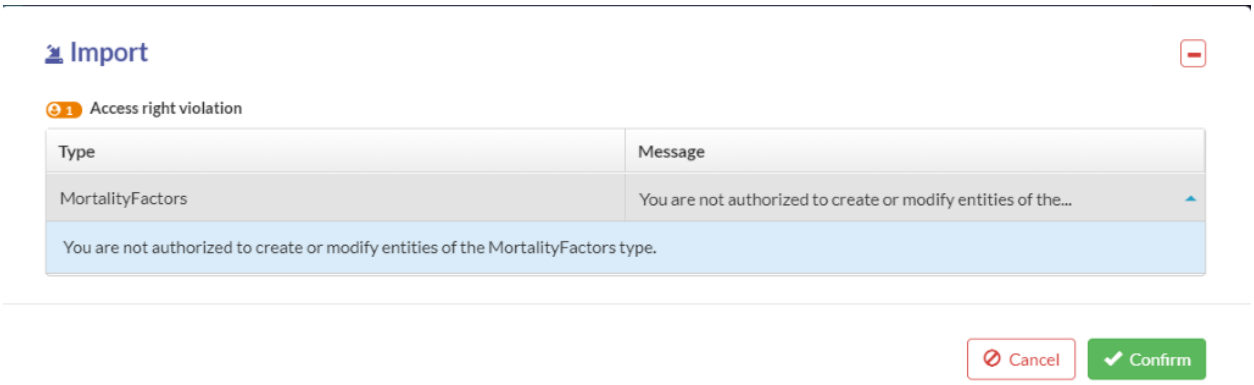
8. Start a **Report Update** process



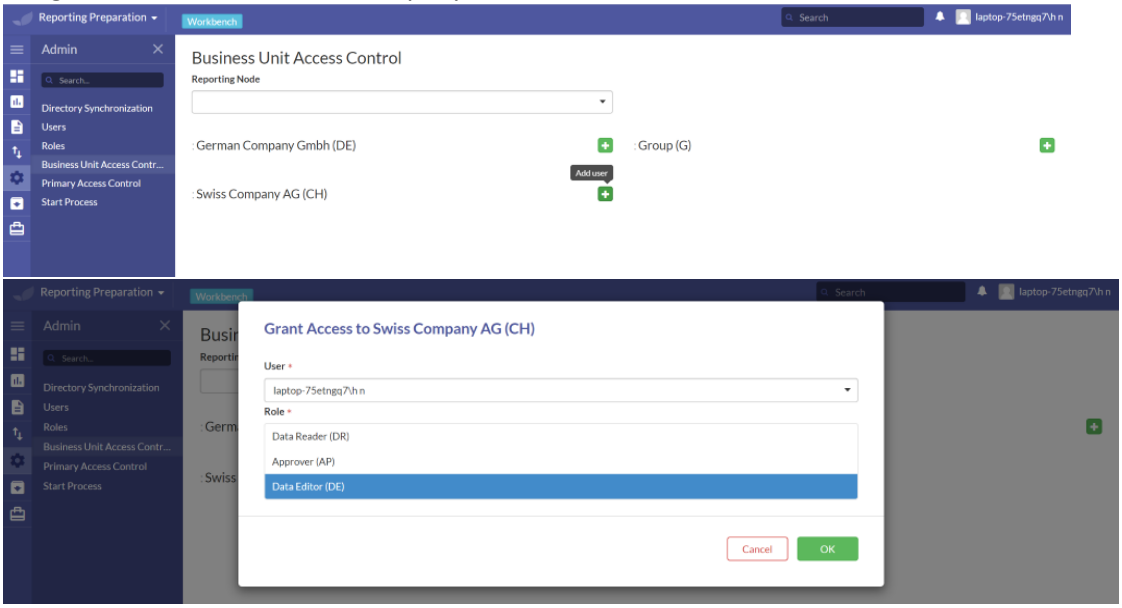
a. Change in mortality factors: setting all mortality rates to zero for all ages between 0 and 10.



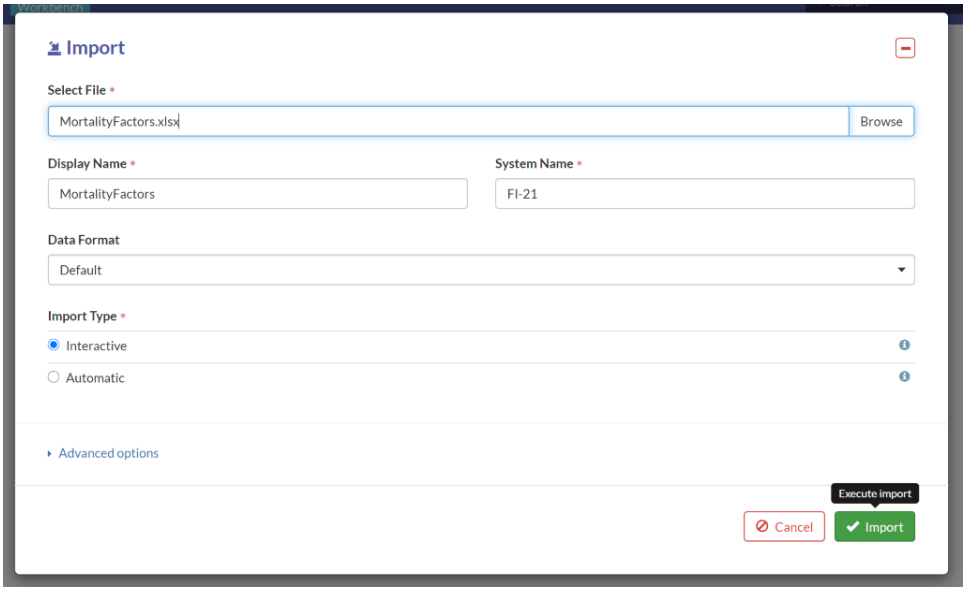
b. Import file of Mortality Factors as DataAdmin:



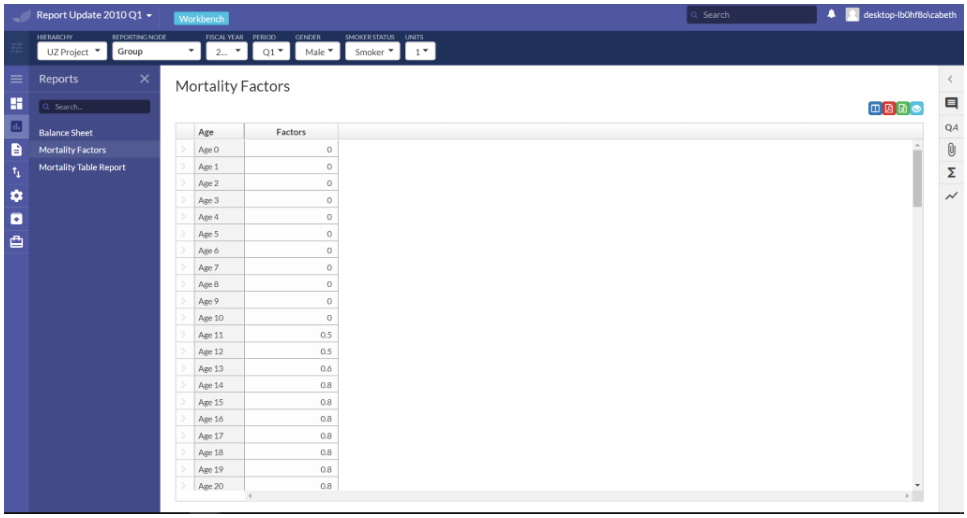
c. Assign DataEditor role for CH company.



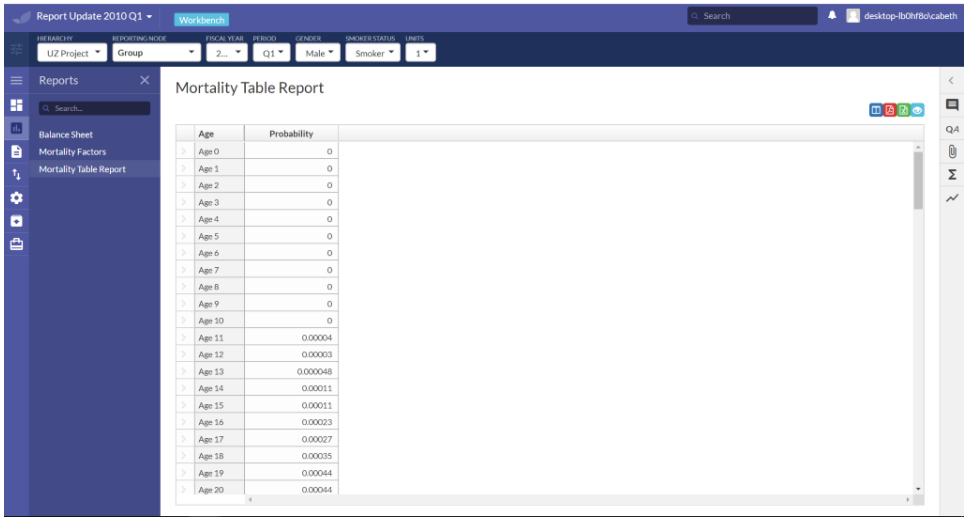
d. And import again.



e. Show the updated figure.



This change also translates to the mortality table report, since the mortality factors are an input for the mortality table.

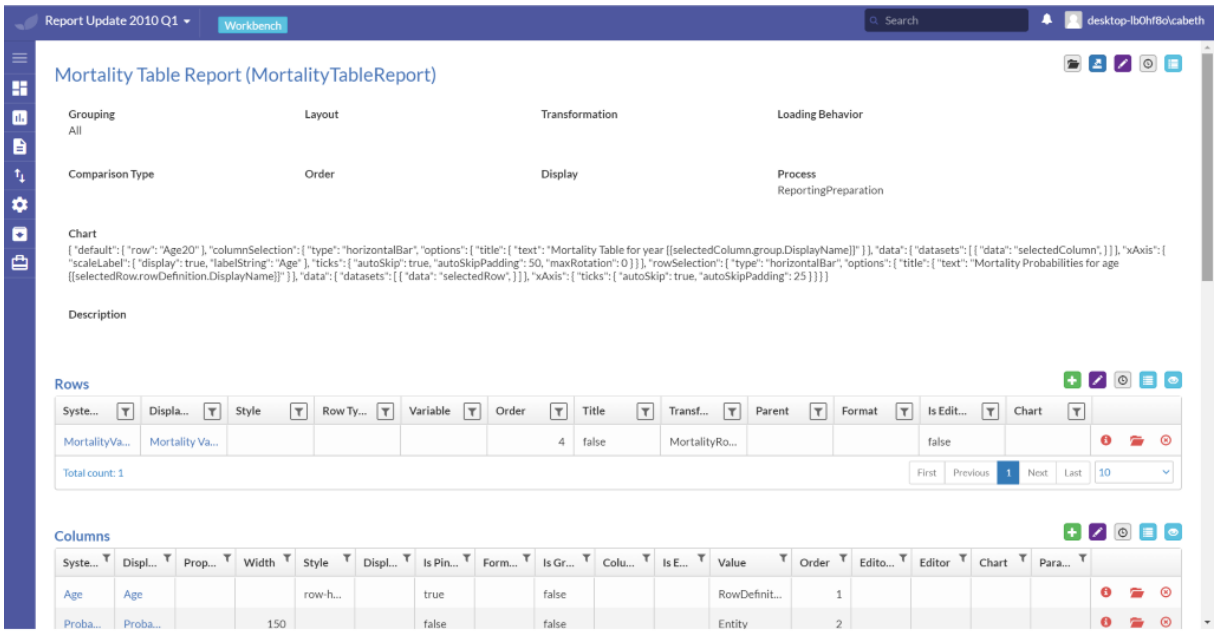


We can also show the multiplication formula which is accessible, visible and editable. For example, we multiply the status quo by 1.1 (e.g. pandemic, i.e. increasing all values by 10% as stress scenario and not put into production and for exploration purposes).

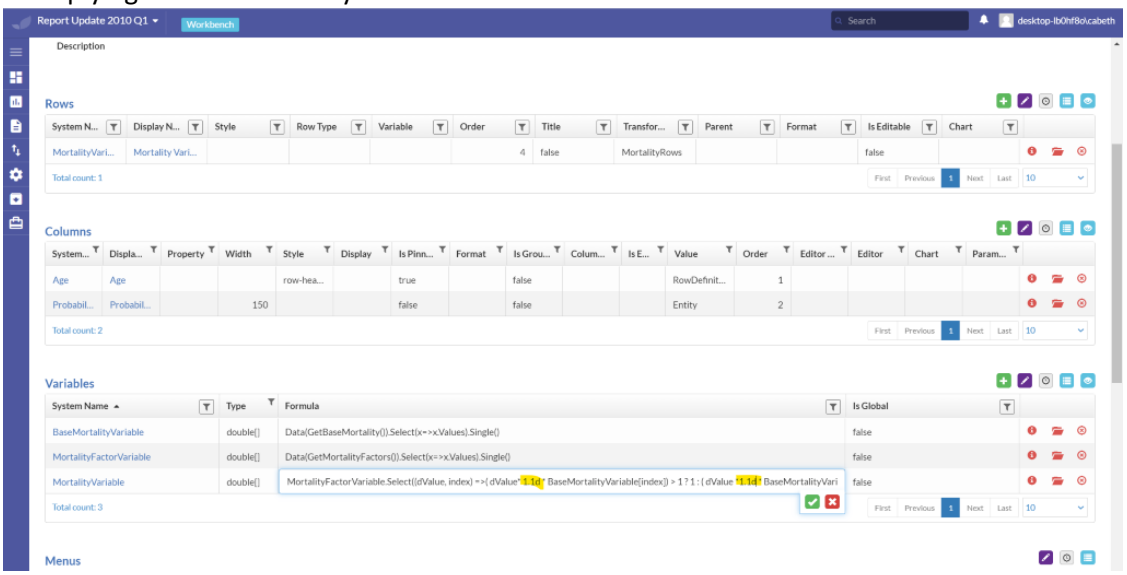
SHOW MORTALITY TABLE REPORT

9. **Show the logic behind the report Mortality Table report:** change formula and show effect.

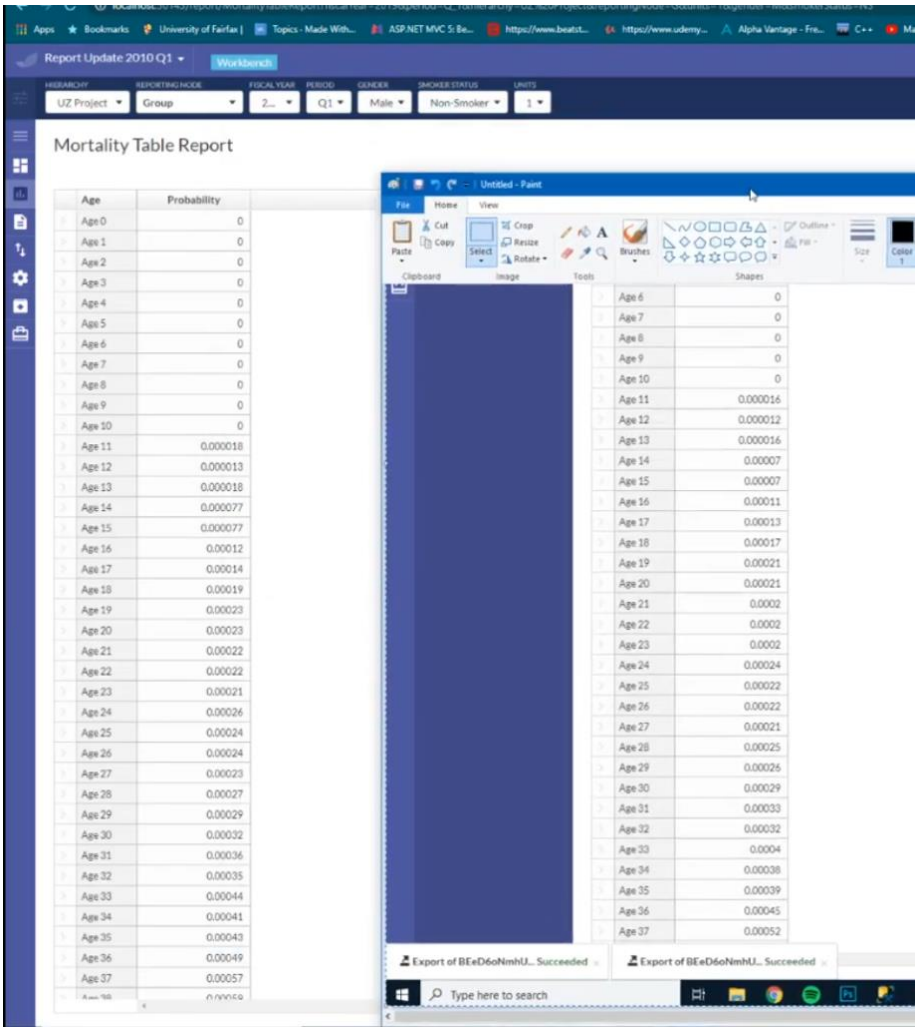
   http://localhost:50143/Catalog/ReportDefinition



a. Multiplying whole formula by factor 1.1.



b. By refreshing the web application, the modified values will directly be displayed (values before change on report-level captured in screenshot on the right hand-side).



The programming language of our DSL is actually C# which is selected by Systemorph as the base and gets enriched with extensions. Example of such extensions are (dothis, getthis functions). In some parts of our codes we only use DSL itself, meaning there is no other code in the background but in other sections we also use totally generic code from the background. Everything that contains some business logic will be brought into the DSL to allow business users to easily change it. Hence, we especially injected DSL in different places, namely in the importer functions as well as the first type of validation rules.

-end of demo-