avaloq

CGTOnboarding

A Windows Desktop application for C# .NET 6.0 Core Windows Presentation Foundation (WPF) with MahApps Framework

Aidan Neil, Andrew Bell, Areeb Khan, Joachim Vanneste

2022

Contents

1 Overview

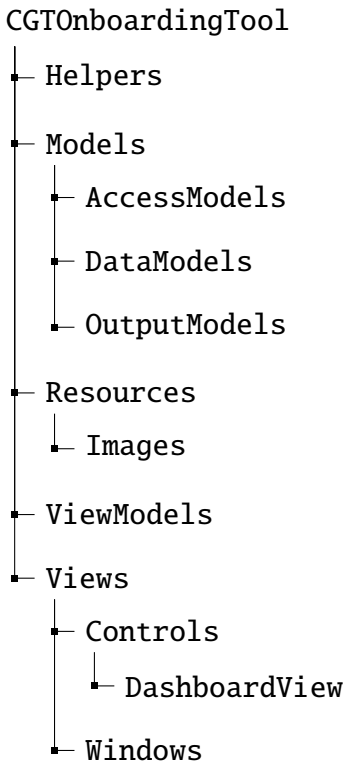
The application is written in C# .NET Core 6.0 and is designed to handle scalable, test-driven and loosely coupled code, adopting an MVVM pattern.

2 System Architecture

The system follows an **Model-View-ViewModel (MVVM)** design architecture. This design structure is very similar to the MVC Model View Controller design pattern, only the View is the entry point for the application, and ViewModels lend themselves to unit testing and event driving code. This pattern also facilitates the separation of the development of data representation (the models), User Interface Layer (views), and the business logic and data manipulation (view models). Messaging between ViewModels and Views is used to allow for validation to be performed on the ViewModel layer.

2.1 Folder Structure

The directories of the application are shown as a tree below. The application has purposely been designed to be extended easily and efficiently to allow for new functions to be build into the system.



3 Models

The models are split into three distinct sub-folder; data models, output models and access models.

3.1 Data Models

Contains the logic for the data-driven models (report, report header and report entries) and the logic for the drop-down items class. The *Report* class holds the tax effects to be displayed on the dashboard. Each report contains a header, a number of entries as well as some other private entities. Reports have a number of methods used to create new entries and update the system to reflect these changes. The *ReportHeader* class contains the name of the client and the years which the report functions span. Each report is constructed with a report header. The *ReportEntry* class describes the effect a particular tax effect (BUC) has to a position. The class has four constructors so entries can be build with a variety of data. These are displayed in succession on the dashboard.

3.2 Output Models

Contains the *ReportExporter* class which contains a method to export the current report. It writes the report header followed by each report entry to a file chosen by the user. In addition, there is the *SecurityExporter* class with an append method which can be called to add new securities to the text file of securities stored in Resources.

3.3 Access Models

Contains the *ReportLoader* class which is used to import a previously exported report. The import method creates a new report with the header given and sequentially creates entries from the chosen file. These are then displayed on the dashboard. There is also the *SecurityLoader* class used to create securities from names given in the text file stored in Resources.

4 Resources

Contains the file *StoredSecurities.txt* consisting of all the securities used in the report, the file is initially populated with five default securities. Securities are read and written to this file when needed. This folder also contains a *Images* sub-folder consisting of pictures used in the application.

5 View Models

Contains the UI-related data and logical states that the view can represent visually to the user. This consists of a *DashboardViewModel* which has methods to filter the report, a *ConstructReportViewModel* which can generate a report after validating the report header and an *AddSecurityViewModel* which can create and add new securities after a validation check. In addition, we have view models for all the CGT Functions (discussed below).

5.1 CGT Functions

Every function has a specific view model which subclasses the *CGTFunctionBaseViewModel* abstract class. Each of these view models overrides the *PerformCGTFunction* method used to perform the associated function. They also contains methods to validate the functions to ensure they can be properly calculated.

6 Views

Contains all of the UI views and elements that are required for the system. Most of the elements in these views make use of inline styling, with a select few labels, buttons and grids using style which are defined in App.xaml to give them a a predefined look that can be utilised system-wide. These views also have a C# file attached to them where all of the input is read from the view to pass information to and from the view models that relate them.

6.1 MahApps Framework

To help us achieve the look of the system that we had envisioned, we used this framework which gave us access to different custom controls and styles that are used in the system. These controls allow us to do things that are difficult to achieve in pure xaml. For example, the TextBoxHelper that is included in the framework enables us to give text-boxes a watermark making them more user-friendly. Additionally, custom messages can be passed into these watermarks to display different types of user errors, we use this when a user tries to input data in an incorrect format.

6.2 Controls

This sub-folder contains *DashboardView* which allows us to pass in custom controls, depending on User Interaction, into the views. The system in its current state, utilises this in relation to filtering a report on the dashboard. Depending on the option that a user chooses to filter by, new filtering menus get passed in and out of the view.

6.3 Windows

The different windows that are used in the system are housed in this folder separate from the rest of the views. Similarly to the main view, these UI elements make use of inline styling with some styles references from *App.xaml*. Under the MahApps Metro Framework, these windows are not regular *wpf* windows but MetroWindows, which, as explained earlier, gives us access to different custom controls and models.

7 App.xaml

Contains all of the references to the MahApps Metro Framework that is needed to achieve certain things utilised in the system. It is also where the styling is predefined for repeated UI elements. Having them defined here, instead of defined inline, makes the xaml code for the views easier to understand and more readable. Due to some of the MahApps Metro controls, it was not possible to have all of the styling defined in this file and is the reason why some elements make use of predefined styling and others use inline.