Accounting System

Edward Harding

[1 Analysis 3](#_Toc511761115)

[1.1 The problem back ground 3](#_Toc511761116)

[1.1.1 Research into Accounting 3](#_Toc511761117)

[1.1.2 The current system 4](#_Toc511761118)

[1.1.3 Clarification of The End User 4](#_Toc511761119)

[1.1.4 User requirements 4](#_Toc511761120)

[2 Design 6](#_Toc511761121)

[2.1 Why I chose to do an Object-Oriented Project 6](#_Toc511761122)

[2.2 System Design 6](#_Toc511761123)

[2.3 Class Diagram and Objects Used 8](#_Toc511761124)

[2.4 Additional Class Diagrams 10](#_Toc511761125)

[3 Technical Solution 12](#_Toc511761126)

[3.1 Why I chose to use Naked Objects Framework 12](#_Toc511761127)

[3.2 Solution Layout 12](#_Toc511761128)

[3.2.1 Client 13](#_Toc511761129)

[3.2.2 Model 14](#_Toc511761130)

[3.2.2.1 Objects 14](#_Toc511761131)

[3.2.2.1.1 The Advantages of LINQ 14](#_Toc511761132)

[3.2.2.1.2 Object Example – Account 15](#_Toc511761133)

[3.2.2.1.3 Repository Example – Accounting Service 18](#_Toc511761134)

[3.2.2.2 Auditing 20](#_Toc511761135)

[*3.2.2.2.1* Audit Service 20](#_Toc511761136)

[*3.2.2.2.2* Audit Record 22](#_Toc511761137)

[*3.2.2.2.3* Audit Menu 23](#_Toc511761138)

[3.2.2.3 Use of an IViewModel 24](#_Toc511761139)

[3.2.2.4 Usability and robustness 27](#_Toc511761140)

[3.2.3 Server 29](#_Toc511761141)

[3.2.4 Database 32](#_Toc511761142)

[3.2.4.1 Entity Framework and SQL Servers 32](#_Toc511761143)

[3.2.4.2 DB context 34](#_Toc511761144)

[3.2.4.3 DB Initializer 35](#_Toc511761145)

[4 Testing 39](#_Toc511761146)

[4.1 Test 1 39](#_Toc511761147)

[4.2 Test 2 42](#_Toc511761148)

[4.3 Test 3 46](#_Toc511761149)

[4.4 Test 4 56](#_Toc511761150)

[4.5 Test 5 60](#_Toc511761151)

[5 Evaluation 67](#_Toc511761152)

# Analysis

## The problem back ground

Utopia Handmade and Vintage is a retail and clothing company. It is run by my brother Mr. T. Harding and his partner. The business is quite small and they tend to make most of their sales through online third-party websites such as etsy.com and also on stalls at multiple different music festivals. As the business is starting to grow and the company is transitioning from something more of a hobby over to an actual business, my brother has come to the realisation that taxes and bookkeeping are legal responsibilities which will become very prevalent in his future. A lot of small business owners tend to use Microsoft Excel or bookkeeping software which can be bought online such as ‘Quickbooks’ and ‘Sage’ however I intend to make a program which can be used (most likely by him) to help assist and record the bookkeeping for the company.

As I personally had a very limited understanding of bookkeeping and accounting I decided that step one would be to do some basic preliminary research into this field. I came to the understanding that double entry bookkeeping is the standard method used in the modern world. Some other things I noted is the need and usefulness of balance sheets and profit and loss statements.

The second step I decided to take was to arrange a meeting with my brother as he would be the end user of the product I intend to build. The two things I was hoping to address with him were his current options for bookkeeping software which he could purchase and use, and the requirements he would need in a bookkeeping software for his business. After some quick research we concluded that although excel was the most cost-efficient option not only did my brother not know anything about using it but it also only allowed cashbook accounting where as he was looking for a double entry bookkeeping system. The other requirements he noted will be mentioned further on.

To conclude I decided that I would start of by creating a very standard double entry bookkeeping system as this would act as a good foundation to add improvements to and tailor it towards the end users more specific requirements. This also helps as it allows for me to make sure the basics are all up and running before getting into a greater level of depth and detail.

### Research into Accounting

I came to the realization that I will need to learn a substantial amount more about accounting and bookkeeping to be able to fully take on and produce a suitable product for my client. To do this I have decided to use a very well acclaimed textbook which is used by most schools and colleges in the UK to teach accounting. It is called “Business Accounting 1” by Frank Woods and Alan Sangster. The book starts from the very basics of accounting and provides multiple practice questions and examples which I will be able to use in the testing and development of my project. I am hoping to progress my program as I progress through the textbook, so I can develop the program hand in hand with my learning of the subject. This is how I plan to create the foundation for my program as stated earlier.

### The current system

The company is currently quite small as I stated earlier and is therefore under the minimum tax bracket. This means that presently they do not have any system in place for bookkeeping as there has been no need for it so far.

### Clarification of The End User

Although my client is Utopia Handmade and Vintage I would say that Mr. T Harding will be my sole end user for the system as he is one of the owner’s and the most likely out of the two to be doing the bookkeeping. He will be my first point of contact for information on what he will require from me. Therefore, the system will be catered towards his requirements and he will also be the person I get to test the program once it is in its later stages and past the book level testing.

### User requirements

1. The program should be able to perform the basic functions of a double entry bookkeeping system, this includes but is not limited to:

* Allowing the user to create multiple different accounts.
* Allow the user to record transactions between the accounts.
* Use these transactions to form a balance sheet.
* Use transactions to make up a profit and loss statement for a requested time period.
* The program should also have the ability to audit user’s actions.

1. The system must be easy to use, simple to understand and navigate, and display data and information in a way which is tidy, efficient and easy to understand.
2. The system will NOT require any authentication process as It will be a locally run program therefore it will only be accessed from the clients own computing device.
3. The client would like the system to be as aesthetically pleasing as possible; as it will be used for long hours therefore it should not be dull.

I would like to expand on Auditing which was mentioned in point 1. Auditing is very much an important part of an accounting system, it does not add value to the arithmetical function of the program, but every accounting system should 100% have auditing functionality for legality reasons. Both Sage and Quickbooks do have an audit trail (excel has some ability to record user actions but it is not up to standard).

Of course, the example I mentioned earlier such as Quickbooks and Sage can do all of these things and more however I believe that as Mr. Harding the end user is new to bookkeeping it would in fact be more beneficial to keep the accounting side to the simplest level possible. One other important thing is that my code will all be free and open source (being shared via GitHub) this allows for other users to further extend its functionality and add complexity to it if needs be.

# Design

## Why I chose to do an Object-Oriented Project

The greatest reason I had for choosing object oriented programming was due to the ability to create user defined objects to be used in my program. After I decided on my project idea I sat down and thought about some of the basic functionality it would need to do and the easiest way to implement it. I had already had some experience with OOP and after looking at the basics of an accounting system I noticed that there were lots of clear objects which could be created. The ability to create your own object set properties and give functionality to the objects really stood out as a necessity to me in designing my project.

A more trivial reason for my decision to go with OOP was just for organisation of the code in visual studio. OOP allows for your code to be split up over multiple segments which allows for easy navigation and organisation of your program.

Another big selling point to use polymorphism is inheritance, having the ability to create sub/super classes of your use defined objects is incredibly useful to reduce the repetition of code and functionality, which in turn reduces the likelihood of errors and usually makes them easier to fix if they effect one part of a class hierarchy rather than finding an error in a part of code in an object which is in other objects and then having to individually fix them all.

## System Design

The base user interface will be created by Naked Objects Framework, this includes Action bars, menu buttons etc. However I will make a few changes to the base design such as changing object colours to help provide standardisation between objects, this will help the user distinguish between types when multiple objects are being displayed on the same page. Below I will explain the main components of my project and what I want them to achieve in terms of functionality, usability, display and restrictions. Actions on instances of objects can be accessed via the action tab at the top of the UI which is on all instances of objects. The parameters for specific objects are explained in the following part of the design section, here I will be discussing more about there layout and integration into the user interface and methods they may have.

**Login / Authentication**

The login system will be done using Auth0. Auth0 is a third-party login / identity authenticating software which can be integrated into a program which has access to the internet. Auth0 will allow for people to login into my project using a google account, this will then provide me with the data I need for auditing. I feel that using Auth0 is the right direction as it will reduce the time I need to spend on the login side of my system which isn’t fundamental to the overall functionality which I wish to achieve from this project, however it is needed for Auditing capabilities.

**Home Page**

The Home Page will be the first page displayed to the user after they have Logged in. It won’t have any specific functionality, it will just be used to separate the two sides of the project, that being the double entry bookkeeping side and the Auditing / Administration side.

**Accounts**

Accounts will be accessible from the Main Menu via two options, via the “All accounts” option or by using the “Find Account by Name” option. Along with transactions; Accounts are the base of a bookkeeping system as they are effectively the conduits for which money is spent and earned through. There are no methods / actions I need on accounts as they are primarily used by other aspects of the project as they are mainly used for holding data. The Main Menu will also need an action for creating an Account. Accounts should contain a name, a type of account (capital, assets, liabilities), balance (based on transactions) and a list of credit and debit transactions involving that account.

**Transactions**

The main menu should contain an “All Transactions” option which will display all transactions in date order, displaying the newest at the top. A “create new transaction” option should also be available, this should implement both a drag and drop and auto fill features for filling out the Accounts field. The Transactions themselves should contain an action called “Show History” which is part of the auditing system. This action should show a list of previous versions of that transaction (if it has any previous versions). This is useful because if a transaction has been updated and you are looking at it in the Audit Record menu you will be able to click that transaction to go to and then perform the “show history” action to see the previous versions. Transactions should show the date of transaction, the debit and credit accounts, the amount and the name given to it by the user.

**Balance Sheet**

The Main menu will contain a method to view all balance sheets and a method to create new balance sheets. The balance sheet will evaluate the transactions and accounts to assert that all movements of money have been accounted for, this is done by checking that the sum of assets and capital minus liabilities equals zero. Instances of balance sheets should display all accounts of type asset followed by their balance total and then the same for Liability accounts and Capital accounts. Right at the end there should be the total for Assets Minus Liabilities and Capital and then the difference for the whole sheet which should always equal 0.

**Profit and Loss**

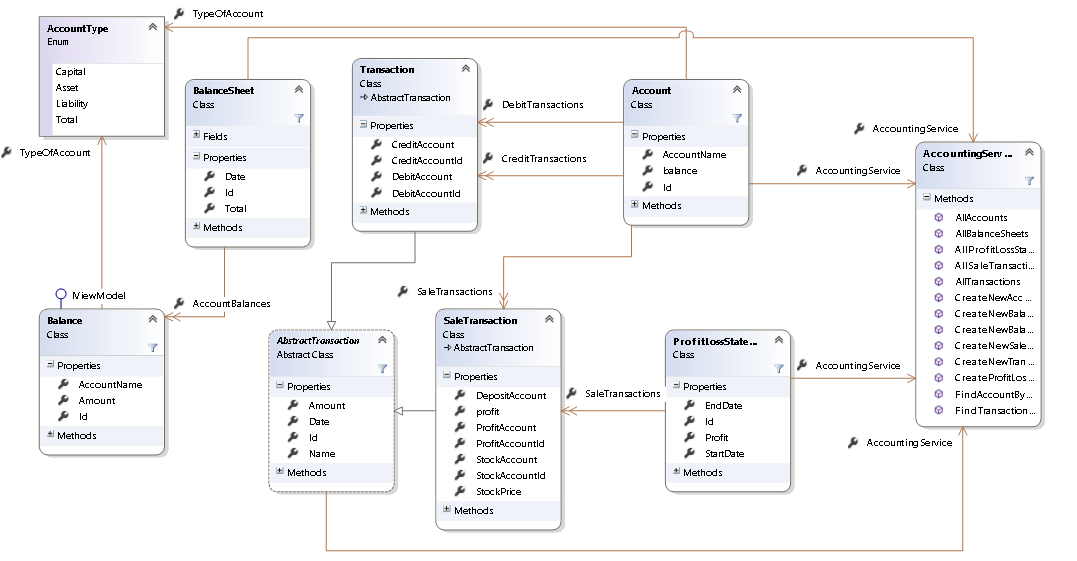
The Main Menu will contain a method to view all profit and loss statements and a method for creating a new profit and loss statement. The profit and loss statement will need calculate the profit made from sale transactions between two dates (entered by the user when created). It will do this by adding up the profit made on each sale transaction. The information for these values on individual items will be held by a sale transaction object which will act similarly to transactions (explained further on in this section). No actions should be needed on Profit and Loss statements as they are basically reports. Profit and Loss statements should show the start and end date for the statement and the profit over that time.

**Auditing**

As mentioned before the Auditing system will need to have user login details to work as the point in auditing is that you know what action was performed, who did it and when it happened (the action, the user and the date time). Unlike the other parts of the project I want Auditing to have its own menu accessible from the Home Page and not be a part of the Main Menu. This is because Auditing is more of an administration section of my project rather than actual functionality to do with double entry book keeping. Inside the Audit Menu I will create Audit ‘reports’ such as “List all Audit Records” and then some specified ones such as “List all Audit Records for Persisted Objects”, and more. This is just for usability purposes so that the user can access specific Audit Records easier than having to search through all of them.

## Class Diagram and Objects Used

Overall Class Diagram.



Account

The account class is – as it is named – meant to represent an account in a running business. Every business has a list of different accounts which are used to record the movement of money and value of objects owned by the company. For example, a small retail company may have an account for the current cash held by them in store.

The account contains a collection of transactions and a calculated balance from these transactions. As this balance is used in the balance sheets and balance sheets can be made for past dates, a function to calculate the balance at a specific date is needed. For both of the above a function to calculate a total balance is needed therefore a function can be made to do this to reduce repetition.

Balance

The Balance class doesn’t really have a business application behind it, it was just necessary to have for functionality purposes to be used to retrieve and format the account data into the balance sheet.

Balance Sheet

As it is named the balance sheet object is made to create a balance sheet and calculate the balances of all the accounts and make sure that the capital equals the company assets minus liabilities.

A title builder method is implemented, this is so that an apt title can be created automatically for the user using the date for which the balance sheet is for. This allows for them to be ordered in the user interface by date for user convenience.

Abstract transaction

An Abstract super class for Transactions and Sale Transactions.

Contains elements used in both types of transaction i.e. a Title builder which builds a title consisting of the inputted transaction name and the date of which the transaction occurred, again this is just for user convenience and for data recording purposes. It should also have a date, an amount, and validation on the date (to make sure it’s not in the future) and amount (to make sure it’s not negative) etc.

Transaction

The most common type of transaction made in a double entry bookkeeping system. It debits and credits individual accounts as it should. As you can see from the class diagram it is a sub class of Abstract Transaction.

Transaction should have auto complete methods to help fill out the debit and credit accounts as there could potentially be loads of different accounts to sift through.

Account Type

The accounts in a double entry bookkeeping system fall into three categories capital, assets and liabilities. I need a way to represent them, so that total balances of each category can be calculated so that the balance can be checked as already mentioned in the Balance Sheet. The easiest way to do this was to create an enum.

Accounting Services

Accounting services doesn’t really represent anything specific in business terms, it just adds functions which are useful to multiple different objects and to the user in the user interface, such as being able to add new instances of objects like Account and transactions. It also contains methods for the user such as listing all the accounts, transactions, balance sheets etc.

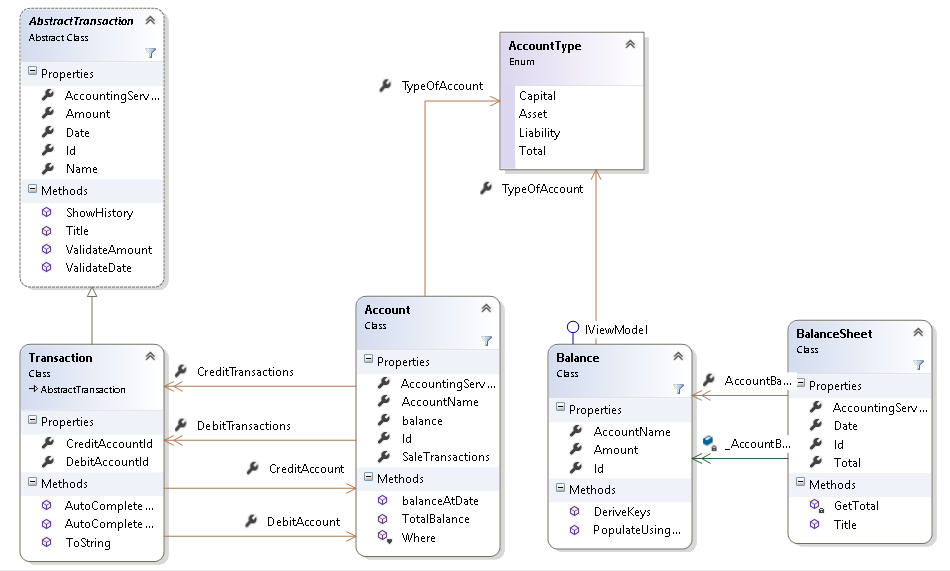
Sale Transaction

A sub class of abstract Transaction. Sale Transaction needs to be able to take in the sale price and stock price of an item being sold and from this calculate the profit. It will need to hold information about the accounts its affecting as three accounts will be affected the stock and profit accounts, and then whichever account the money is being deposited into (i.e. whether the payment came in cash or bank transfer form).

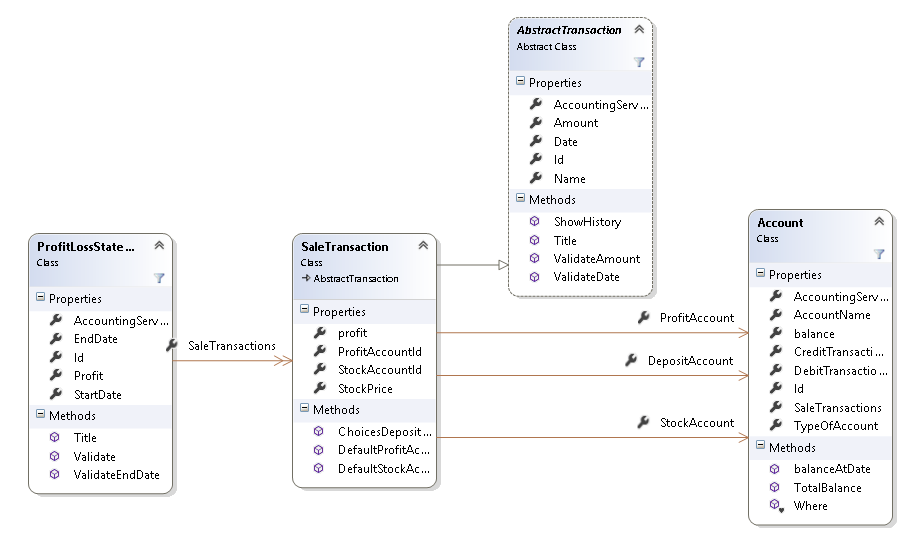
Profit Loss Statement

As the name suggests this class creates a profit and loss statement for a given time frame. It does this by getting the Sale Transactions made in that time frame and adding up the profit value for each transaction together to get a total.

## Additional Class Diagrams



Above is a slightly more in depth look at how the Balance sheet system will work. As you can see a transaction has both a credit and debit account and intern an account contains a collection of these transaction in which they are used. The Balance will inherit its keys from Account, and both balance and Account uses the TypeOfAccount enum. You can see balance sheet gets the balances and account information from Balnce. As you can see Transaction is a sub-class of abstract Transaction and therefore inherits its properties and methods.



Above is a slightly more in depth look at the Profit/Loss System. As you can see Sale Transaction is a sub-class of abstract Transaction and therefore inherits its properties and methods. The Profit Loss Statement Contains a collection of Sale Transactions. Sale Transactions also have three account fields which need to be filled as show above.

# Technical Solution

## Why I chose to use Naked Objects Framework

I chose to use Naked Objects Framework for multiple reasons. The biggest advantage for using it is its UI. The client side of NOF is brilliant for database orientated projects as the UI will automatically display objects in a tabulated way similar to how it would look in a database. It also separates object methods and adds them to the actions button on the UI. This is all customisable from inside the objects. The UI’s colour and layout can all be customized, this includes individual objects and everything.

NOF also contains the ability to set up user profiles using Auth0, this can be done via a developer-built profile system or by using pre-existing identification such as a google or Facebook login, this became useful in the auditing section.

Another advantage I found when researching NOF was that it provides ‘hooks’ which can be used for auditing. A hook will pick up certain user actions and they allow for me to write code to determine what happens when they are picked up. This allowed for me to implement an auditing system. Using the hooks and the NOF Auth0 user profile system I was able to create an Audit service which would use the ‘hooks’ to audit the user by saving information about the user’s profile and the action they performed into an Audit Record object class.

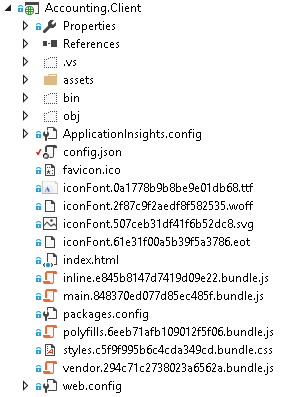
As you can see the advantages of using Naked Objects Framework allowed me to direct my focus towards building a more complex object model and better overall solution as I didn’t have to worry as much about these other ‘tit-bits’.

## Solution Layout

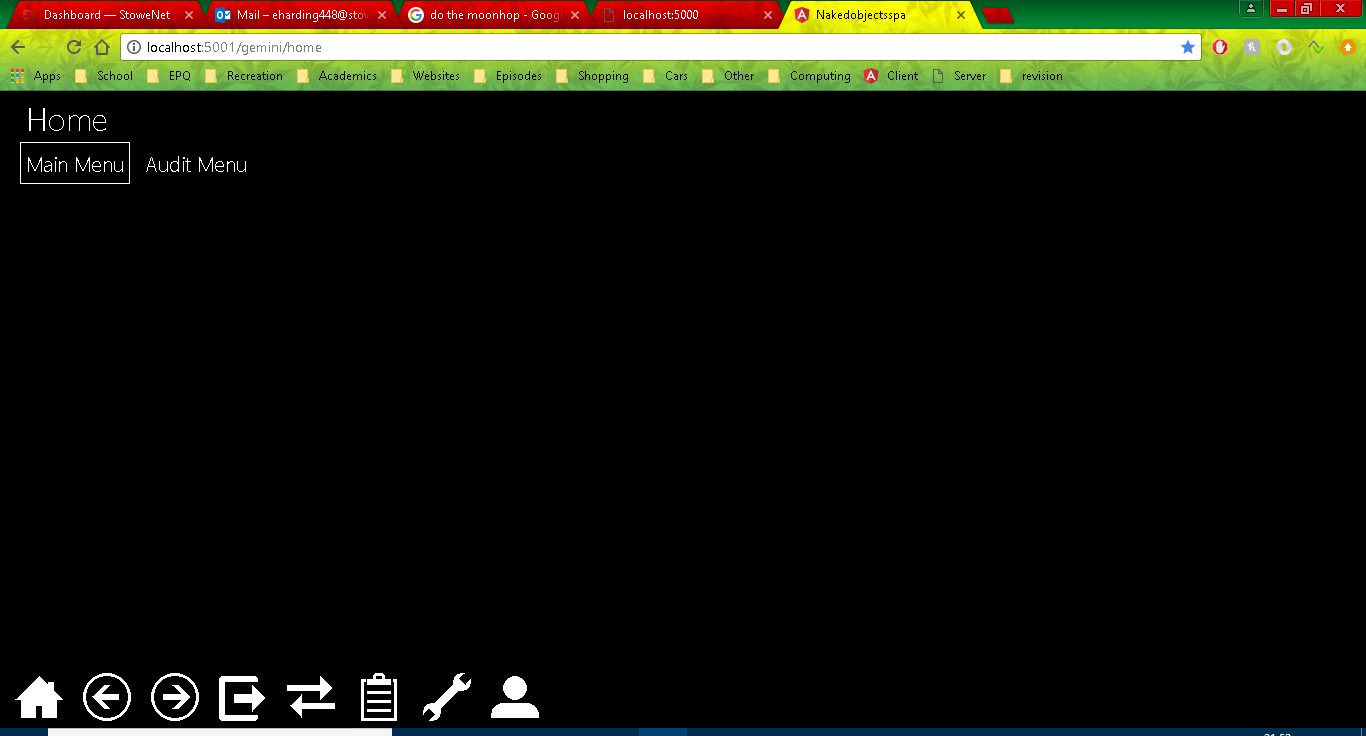
The Solution consists of four separate projects. The Client, Model, Server and Database projects. This format is a way of using the Naked Objects Framework, it works using the client-server mode. By this I mean that everything the user does via the UI (provided by the client project) goes through the server via requests and responses. I started my project by downloading the NOF Template Solution which comes with the four projects included. The NOF Template solution can be downloaded by anyone and is referenced in the NOF user manual. Below is a breakdown of the projects into more detail.

I will show a screenshot of the project explorer for each section.

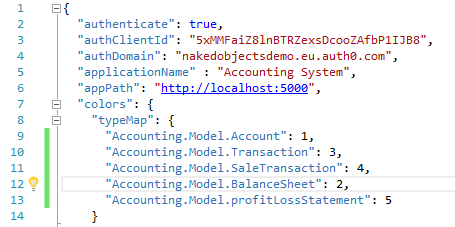
### Client



The client project provides the User Interface for the solution (pictured below).

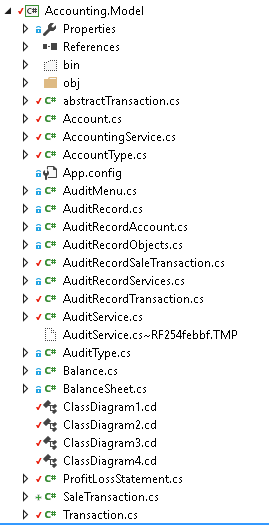


The client uses the Objects and repositories created in the Model Project to create the interface. The actual interface itself can be customized by the programmer in additional ways, I personally decided only to change some of the colours of the objects this is done in the config.json file, the code for this is shown below. The colours themselves will be visible in screenshots of the program.



The UI also supports simple usability functionality like drag and drop and validation, I will go into further details on these later on in the technical solution.

### Model



The Model project is where almost all of the actual program is written, it contains all the classes and services used in the program.

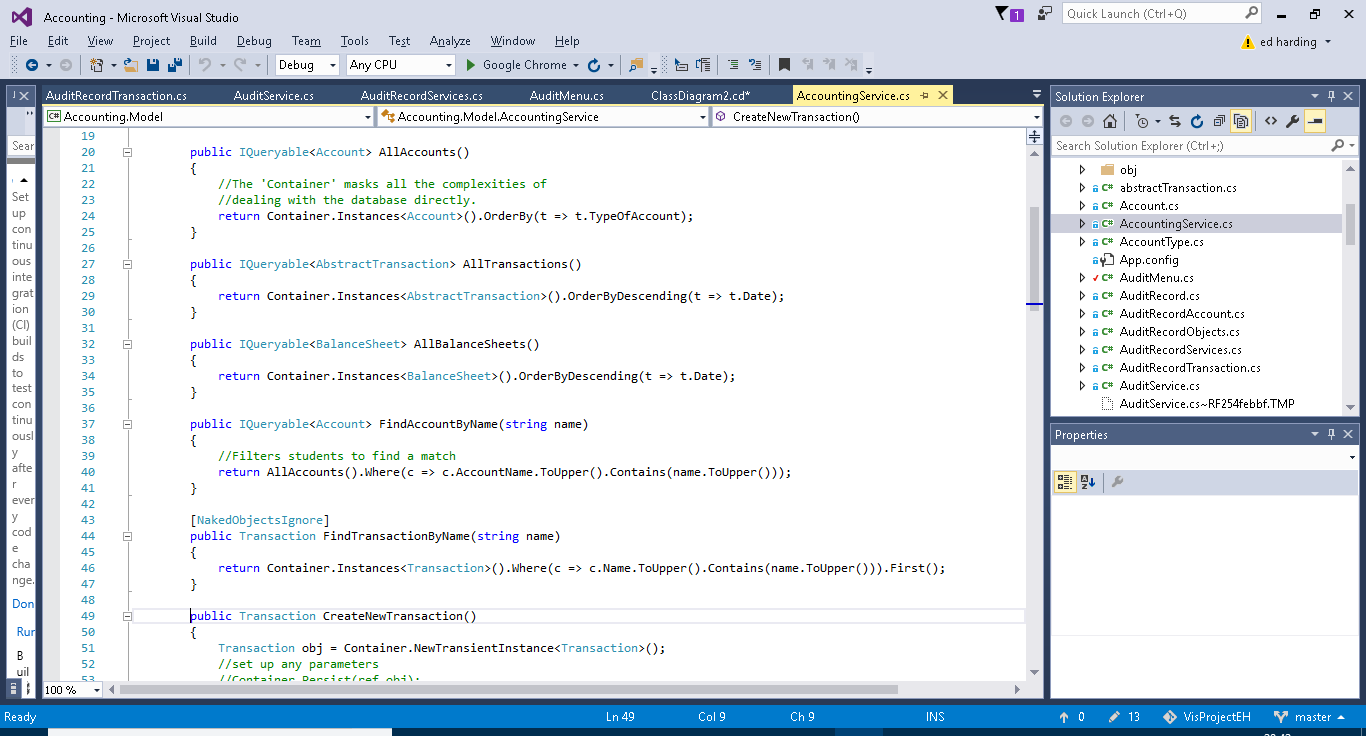
#### Objects

In this section I will analyse some example classes from my code to demonstrate what is going on under the covers.

##### The Advantages of LINQ

LINQ is a uniform query syntax which is integrated into C#. It is very similar to SQL in the way that it is used to save and retrieve data from a database. The significance of LINQ is that because it is integrated into C# it can easily interact with databases created from within my project.

There are many advantages to using LINQ over SQL. Structured Query Language (SQL) has been around since 1974 and in this whole time it has never truly been revamped it has just been extended. As you can probably guess this has caused it to become very clunky in comparison to today’s languages. LINQ is a lot more compact compared to SQL, below is an example of a LINQ statement from the Account Services class.

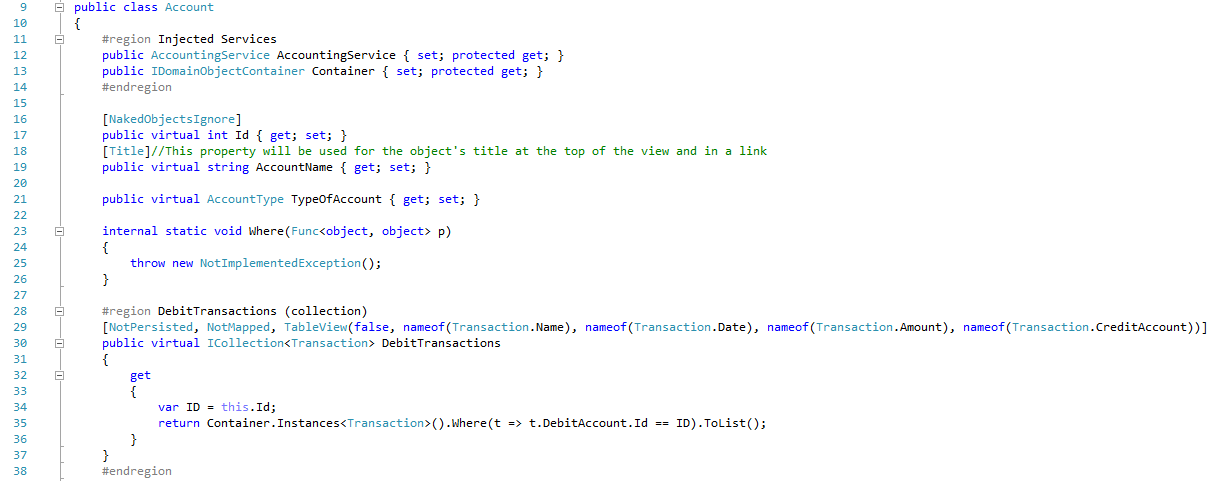


As you can see this is a lot more compact compared to the traditional SELECT FROM WHERE SQL statement which can get very bulky the more complicated the query gets. Another advantage of LINQ is that it is written in a way which is very similar to C# it uses the “.” syntax and C# and user made methods can be incorporated into the query. For general coding it is easier to stick to one form of coding rather than switching between two.

A final point I would like to make about LINQ is that it is technically a form of functional programming. Functional programming is a relatively new programming paradigm which does not allow for any sequential command execution and instead uses functions made up of one single expression. LINQ achieves via the use of a lambda expression. This can be seen in the previous example of code; the lambda expression is the “c => c…….” part.

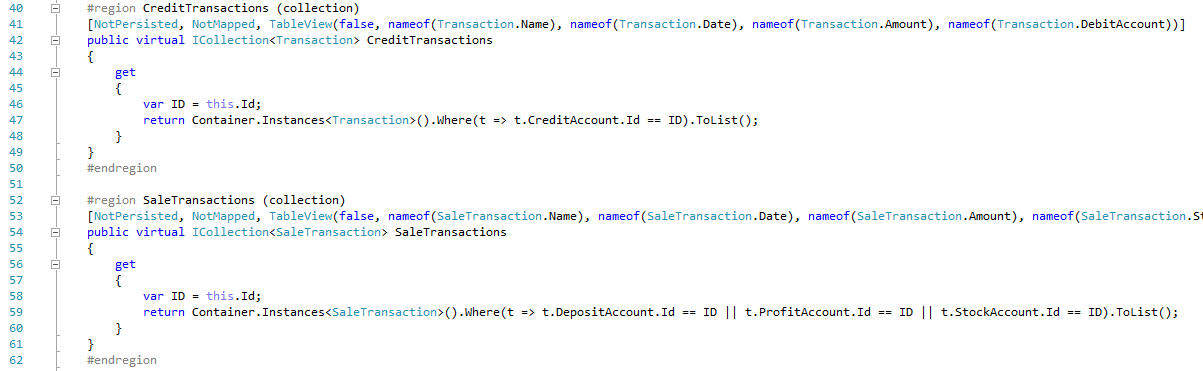
##### Object Example – Account

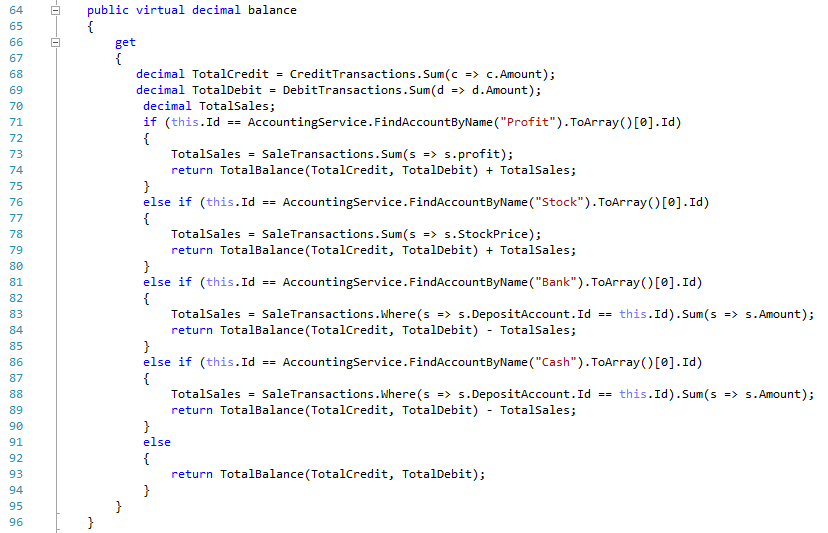
Below is an example of one of my objects, I have chosen to use my Account model as it is one of the most used objects in my project.

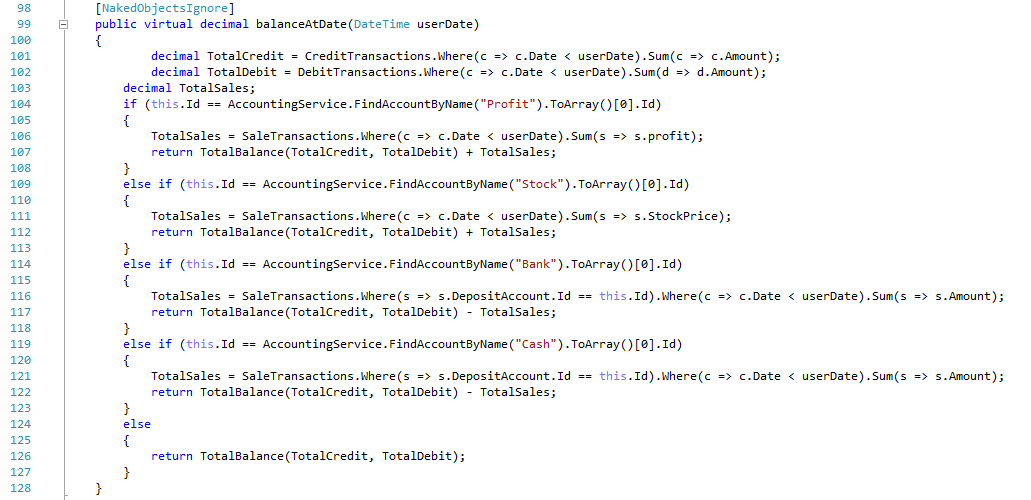


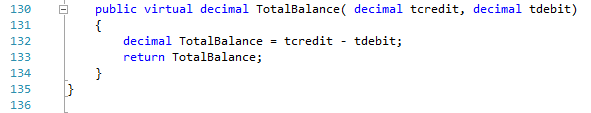
Line 11 is a region which contains any injected services used in this class, a lot of my classes have these regions. The Container service (a NOF service) is used in account and therefore must be injected.

Above (line 28) and below (line 40 and line 52) are all collections used to hold the list of debit, credit and sale transactions applied to the specific account. As you can see the get clause uses a LINQ statement to get the data for the collection. Further down is a screenshot of the UI showing this.







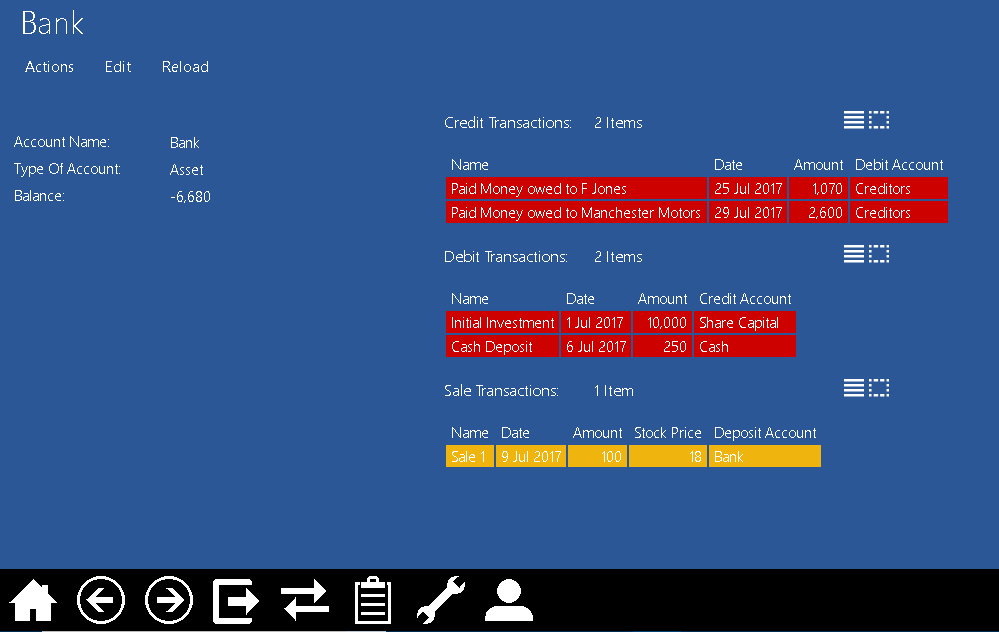


Above (line 64) is a property which gets the balance from the collections of transactions mentioned earlier, it uses the if statement to check if the sale transactions need to be taken into account as sales transactions affect three accounts (but only certain accounts).

Lines 98 does the same as the above method except it gets it at a certain date not at its current stage.

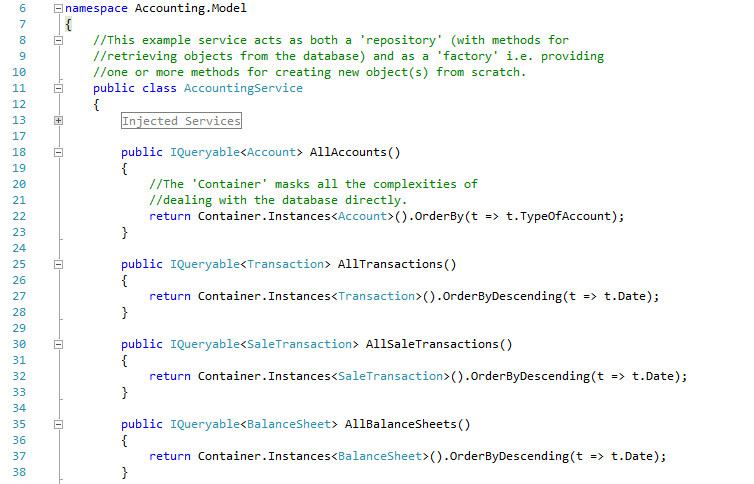
Line 230 show a method which is used in the above two methods for getting a balance between the debit and credit transactions.

Below is a screenshot of an account, on the left you can see the collections of transactions, and on the right you can see the accounts name, type and balance.

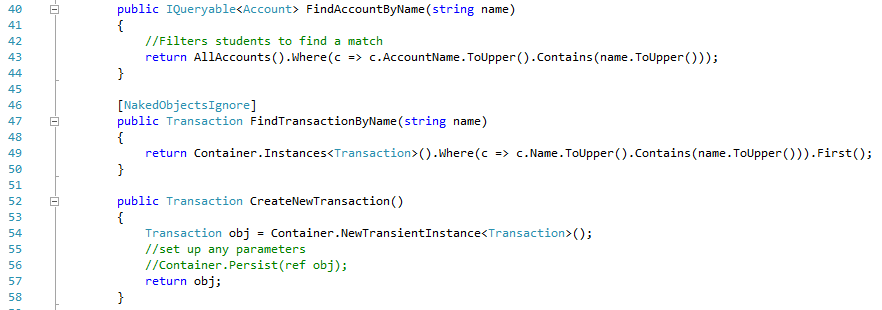


##### Repository Example – Accounting Service

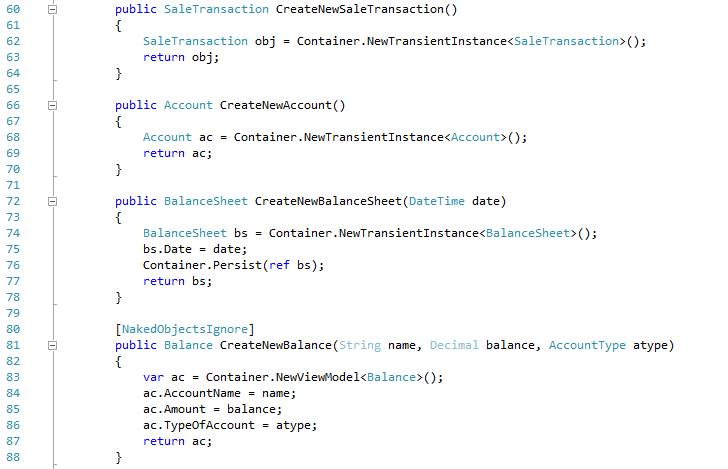
Most methods I talk about will be demonstrated one way or another in the testing section.

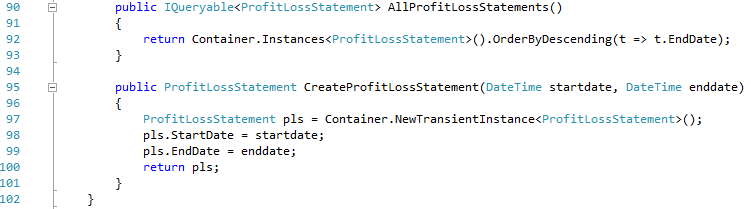


Line 18, 25, 30, 35, 90: All of these lines contain different methods which all follow the same pattern. These methods use LINQ statements to get all the instances of a specific object, e.g. Account, these are then returned to the user as a list. The list is ordered in different ways depending on the object, as you can see The AllTransactions Method lists them by date in descending order.



Line 46 is an attribute for the method on line 47, the Naked Objects ignore attribute means the method below it is not displayed to the user in the menu, it can however be used in other objects. The method on line 47 is another query LINQ statement.





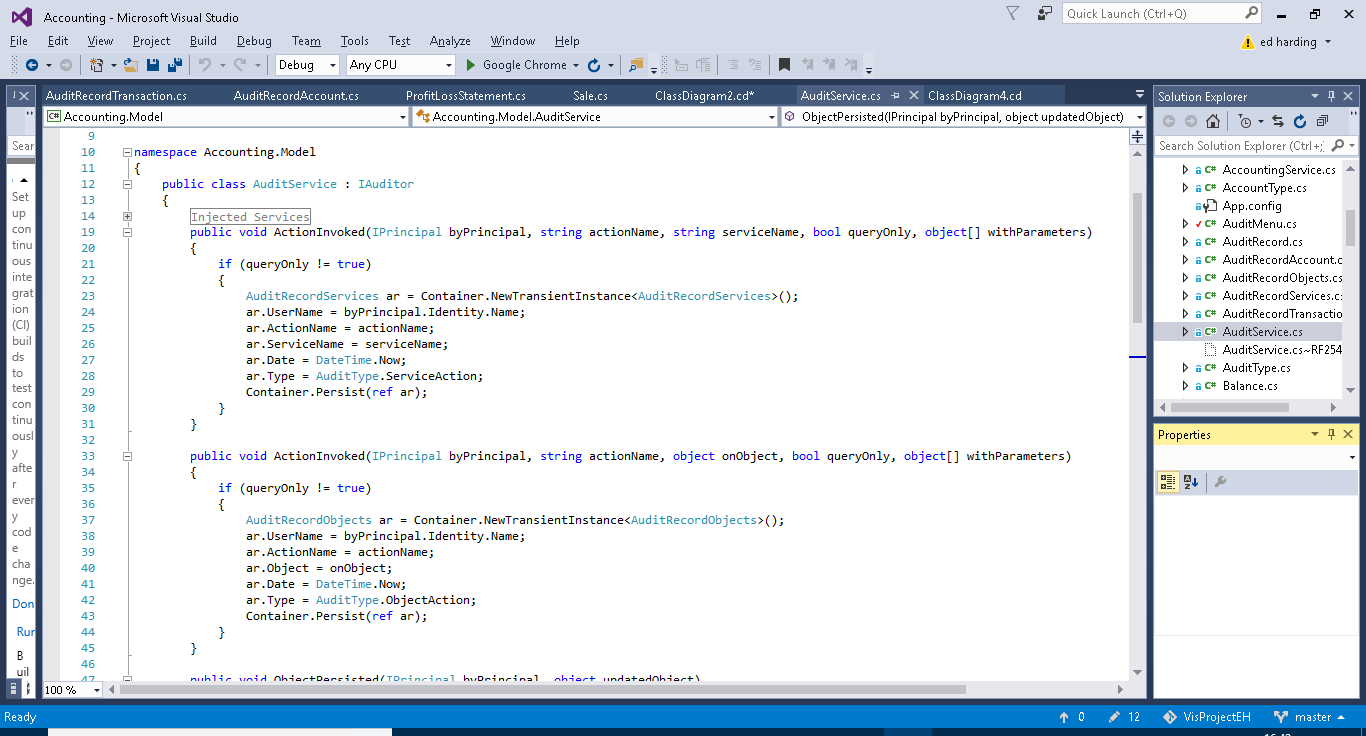
Lines 52, 60, 66, 72, 81 and 95 are all methods used for creating other objects, again they all follow a similar pattern. They use the container service (injected at the start of the code) to create a new instance of the desired object. Each method has different parameters depending on the different properties which need to be defined by the user. The user entered properties are then assigned to the corresponding properties on the newly created instance of the object.

#### Auditing

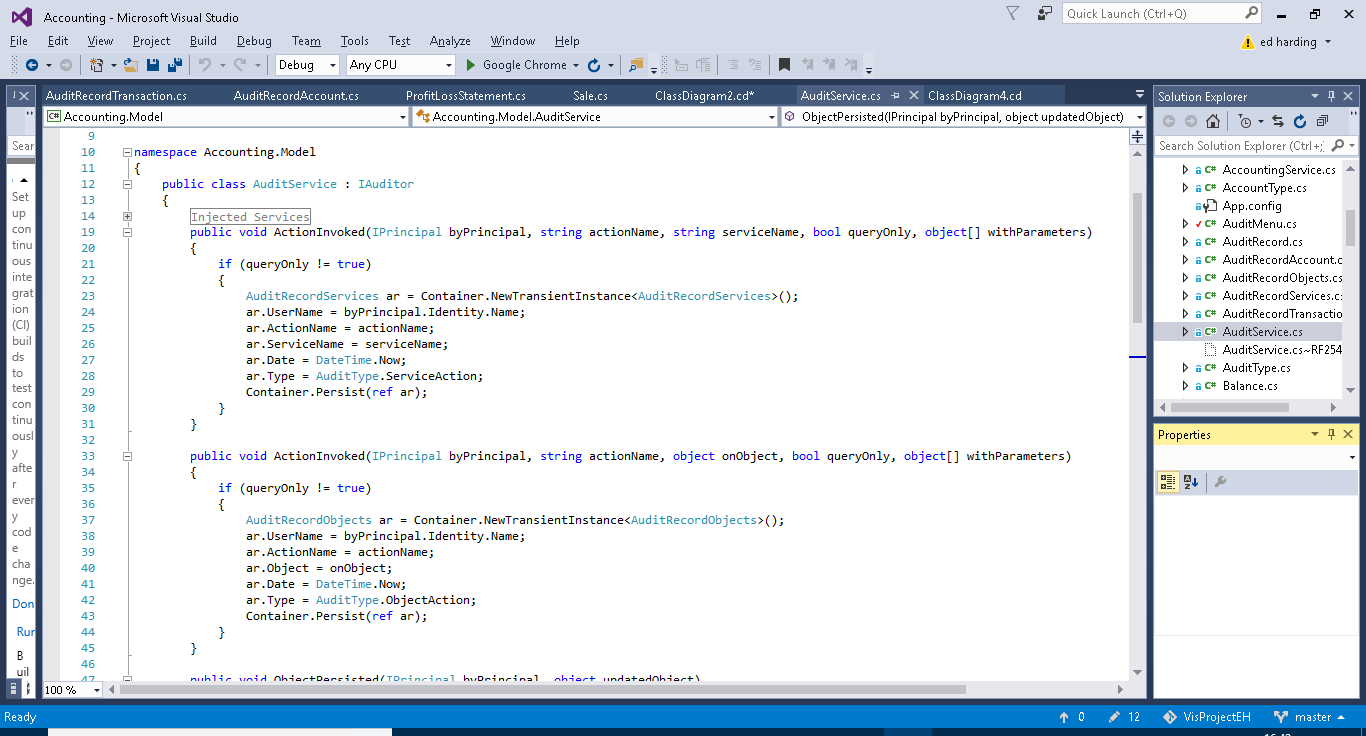
Auditing is one of the more complex capabilities of my program. My auditing code can be split into 3 sections (I will go into more detail of each below). The AuditService which detects when actions have been perfomed by a user, the AuditRecord which is used to save information about the action and user and finally the AuditMenu which is the UI end of the audit capability, it is used to display the audit records to a user.

##### Audit Service

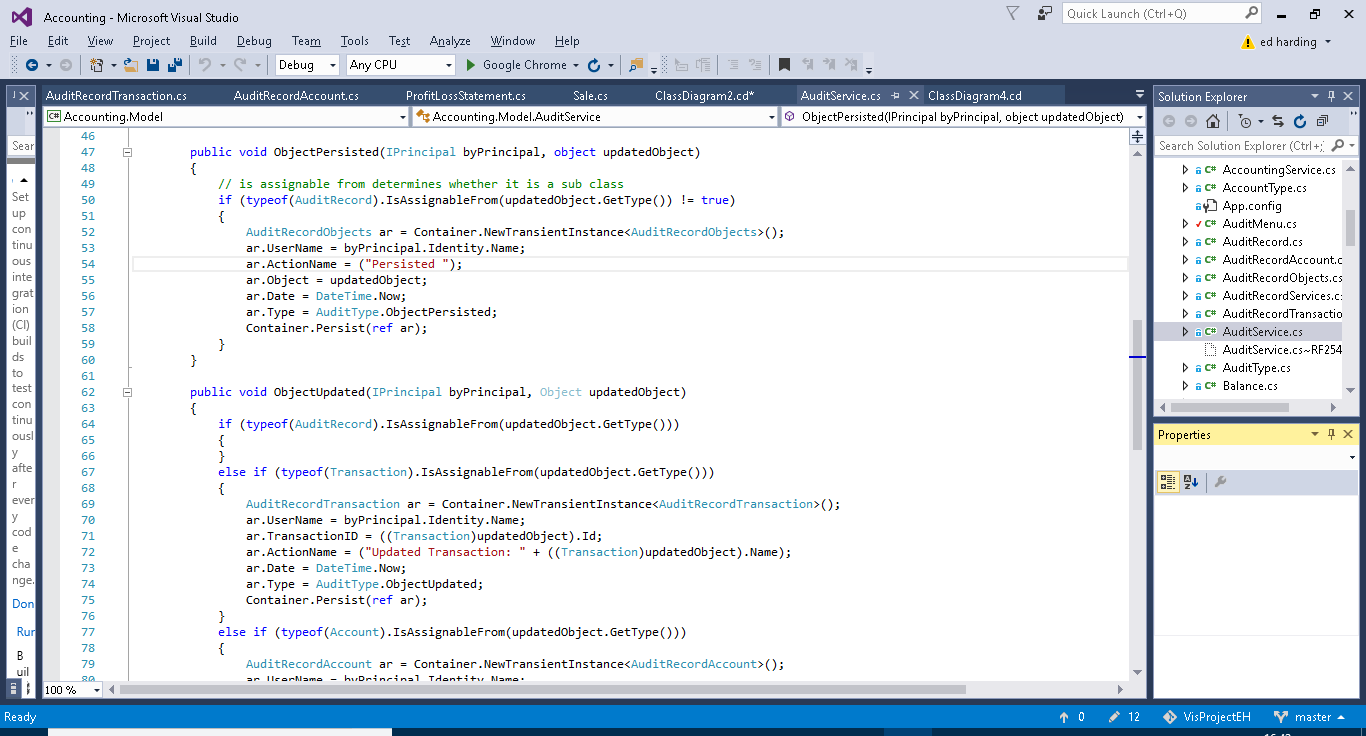
The Audit service constists of 4 methods each will be called under different circumstances.



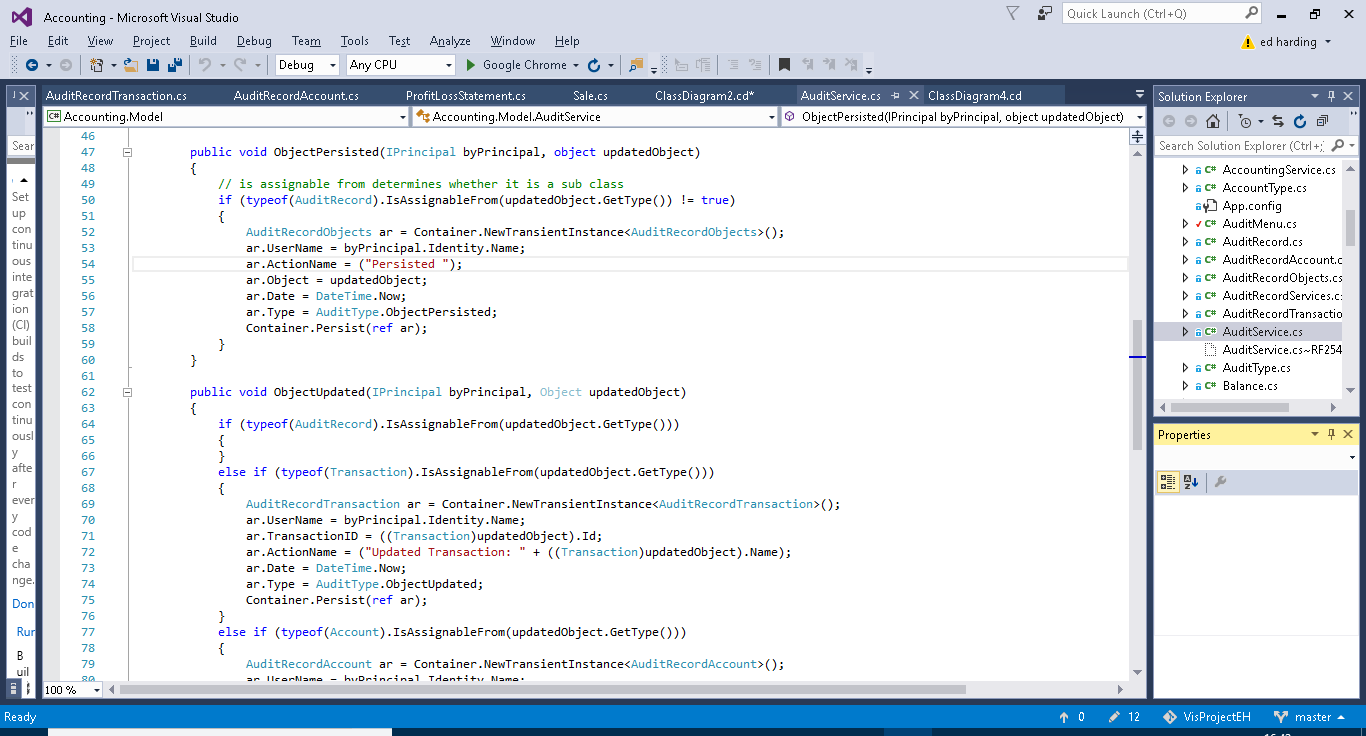
The first method (above) is called when a service action (e.g.) is executed by the user. The if statement checks to see if the action was a query only action, if it is nothing will happen as queries don’t necessarily need to be recorded as nothing critical is happening to the data. If the action isn’t a query action the method will create a new instance of AuditRecordService (AuditRecords are explained further on in this section). It then fills out the fields from AuditRecordService and persists the new instance. The username is obtained from the byPrincipal parameter. The username comes from the Auth0 log in system. AuditType is an enum I created for querying purposes you will be able to see this in use later on.

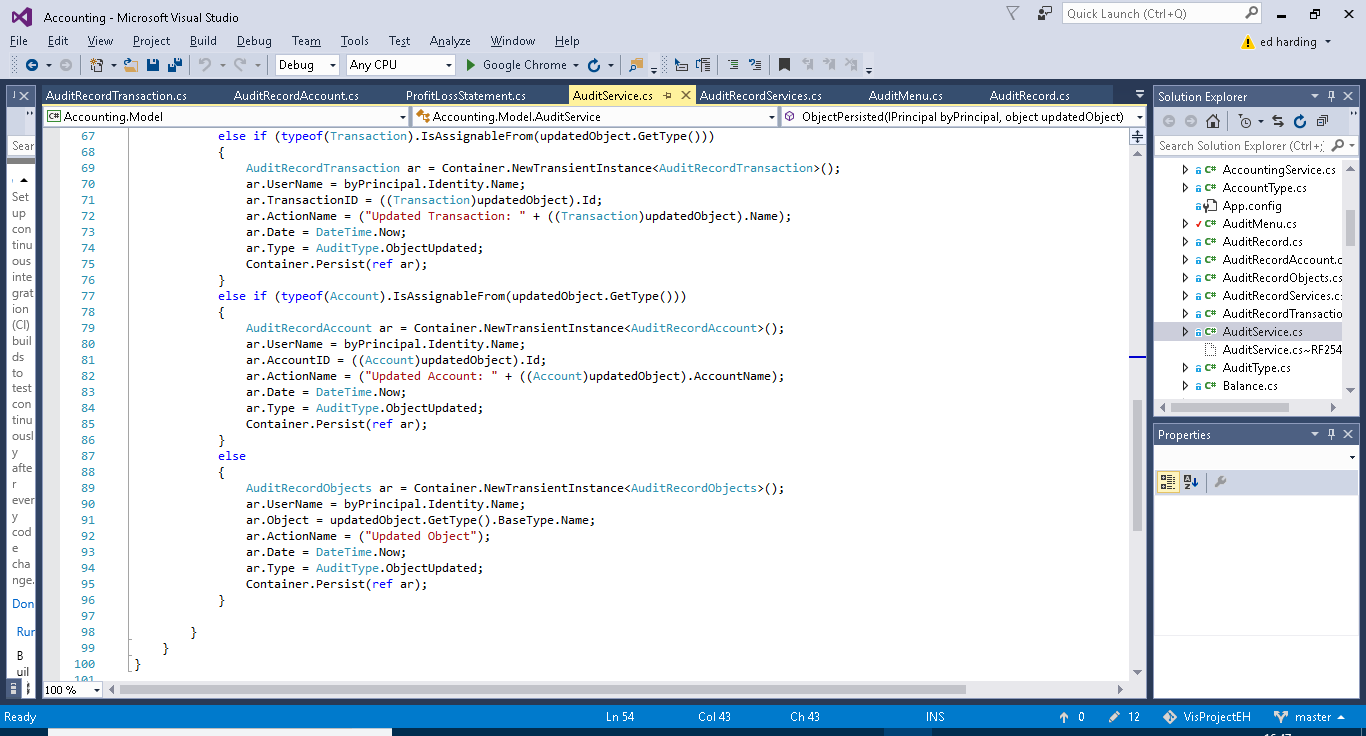


The second method is very similar to the first however it acts upon actions performed on objects (e.g.). Like the first method it checks whether the action is query only or not, it then creates a new instance of AuditRecordObjects, fills out the fields and persists the new instance.



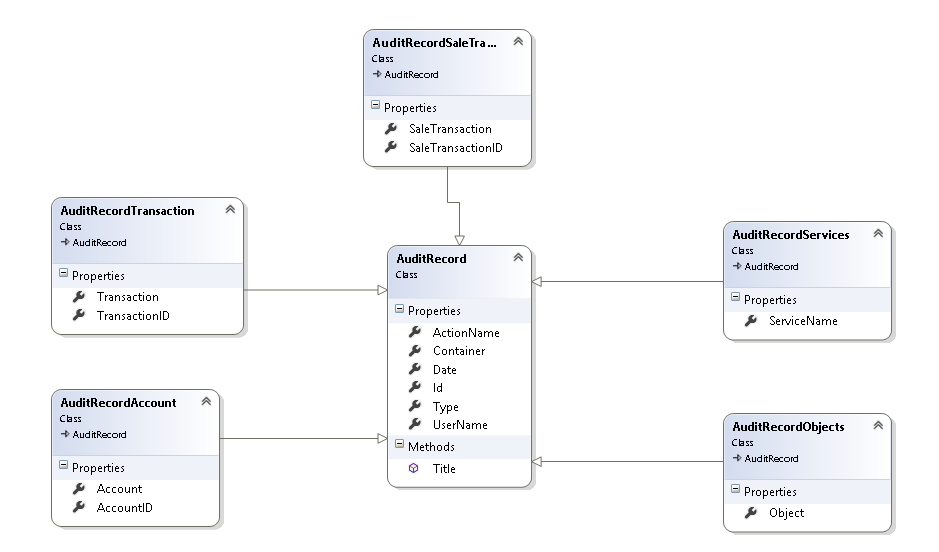
The third method is for auditing persisted objects. The first if statement is very important, its purpose is to assert whether or not the object which has been persisted is a sub class of AuditRecord (e.g. AuditRecordService), if it is, then the method does nothing. This is important as without this if statement this method will be called whenever any AuditRecord sub class is peristed such as in the previous methods. As this method itself persists a subclass of AuditRecord it then will get stuck into a loop and a stack overflow error would occur, this is in fact what happened the first time I tested my audit methods, as I did not have the if statements to catch the problem I ended up with my program halting everytime I did anything to an object. If the persisted object is not a sub class of Audit Record then the method will create, fill out and persist a new instance of AuditRecordObjects. The action name is not passed in through a parameter as in previous methods and is therefore declared as “persisted” by the method.



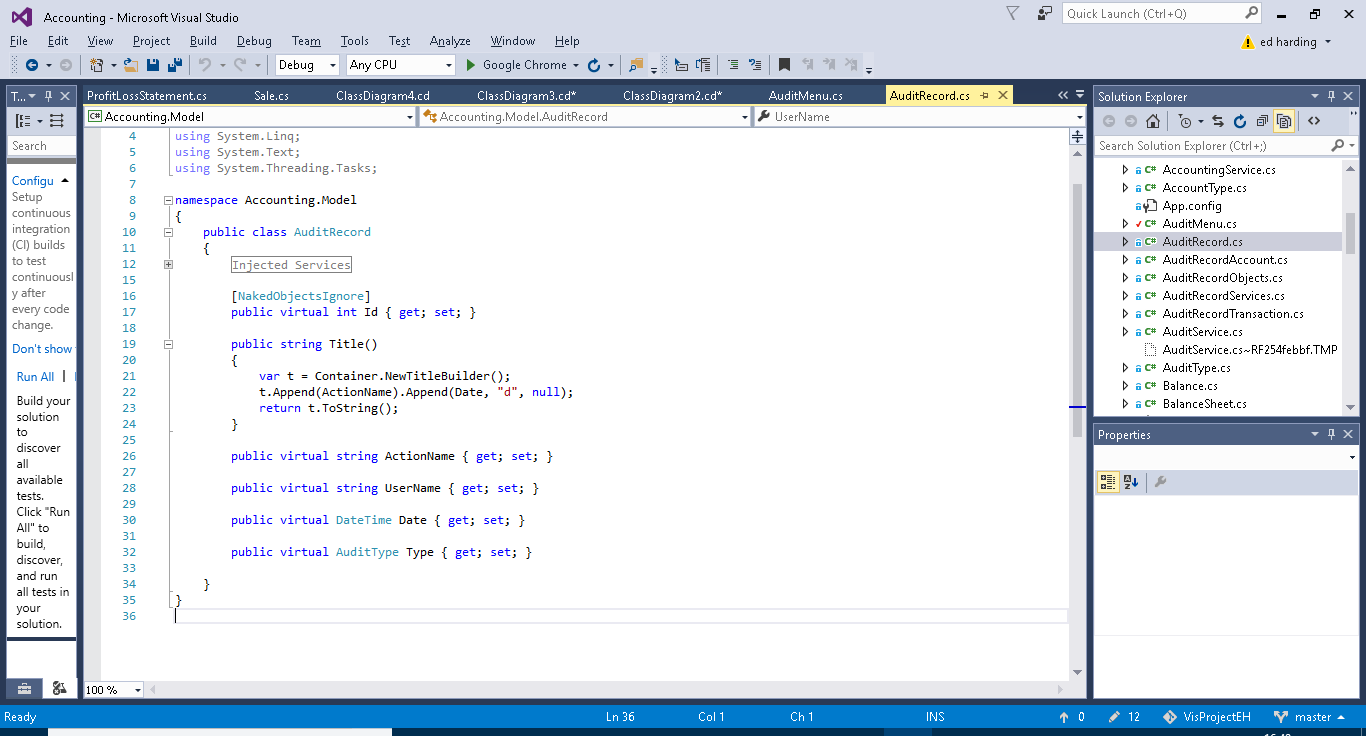


Similiarly to the previous method the first if case checks that the updated object isn’t a sub class of Audit Record, again this is to avoid stack overflow errors as described before. The next three if cases are to establish if the updated object is a transaction, account or other object. This is because transaction and account have their own subclasses of AuditRecord. As before a new instance of the required AuditRecord subclass will be created the fields initialised and then the instance will be persisted. The advantage of having Account and transaction records separate from one another and other objects is so that the name of the account or transaction can be added to the action name for userbility purposes and for querying advantages.

##### Audit Record

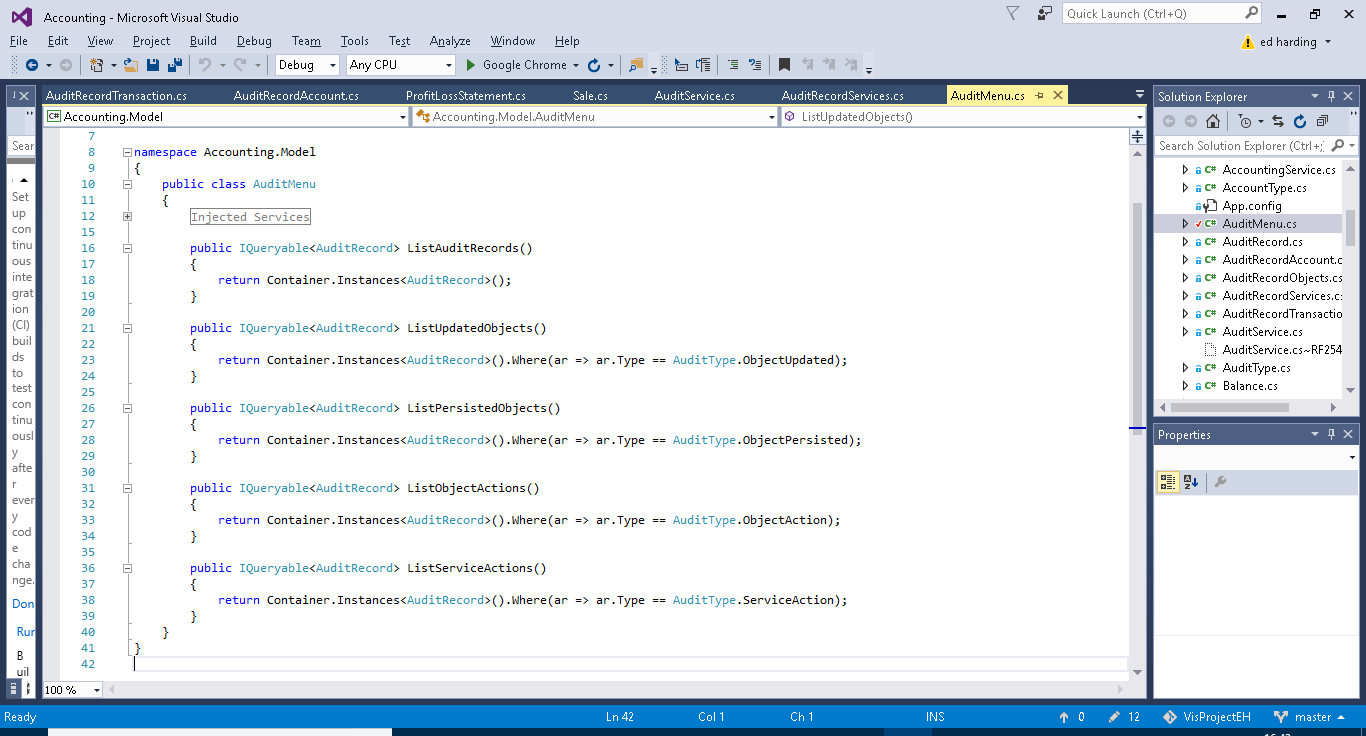


Above is a class diagram showing the four subclasses of account record. These were used in the methods in Audit Record (previous section).



This is the super class Audit Record. It contains the Username ActionName and Date fields, very typical and useful information for someone auditing the system. The AuditType field is for querying purposes. And the Title Builder is for usability purposes i.e. making the name of each instance of AuditRecord more presentable and convenient for the user to find what they need as it combines the date and the action name. The subclasses each have one or two additional fields specific to themselves.

##### Audit Menu



The audit menus sole purpose is to provide pre-built query methods on the entirety of saved AuditRecord. These methods just make it easier for the user to find what they want. The first method just outputs an entire list of all AuditRecord (this being all the instances of the subclasses of AuditRecord). The second method Lists all the instances of AuditRecord which contain information on Updated Objects. The third method Lists all the instances of AuditRecord which contain information on Persisted Objects. The fourth method lists all the instances of AuditRecord which contain information on executed object actions. The fourth method lists all the instances of AuditRecord which contain information on executed service actions.

#### Use of an IViewModel

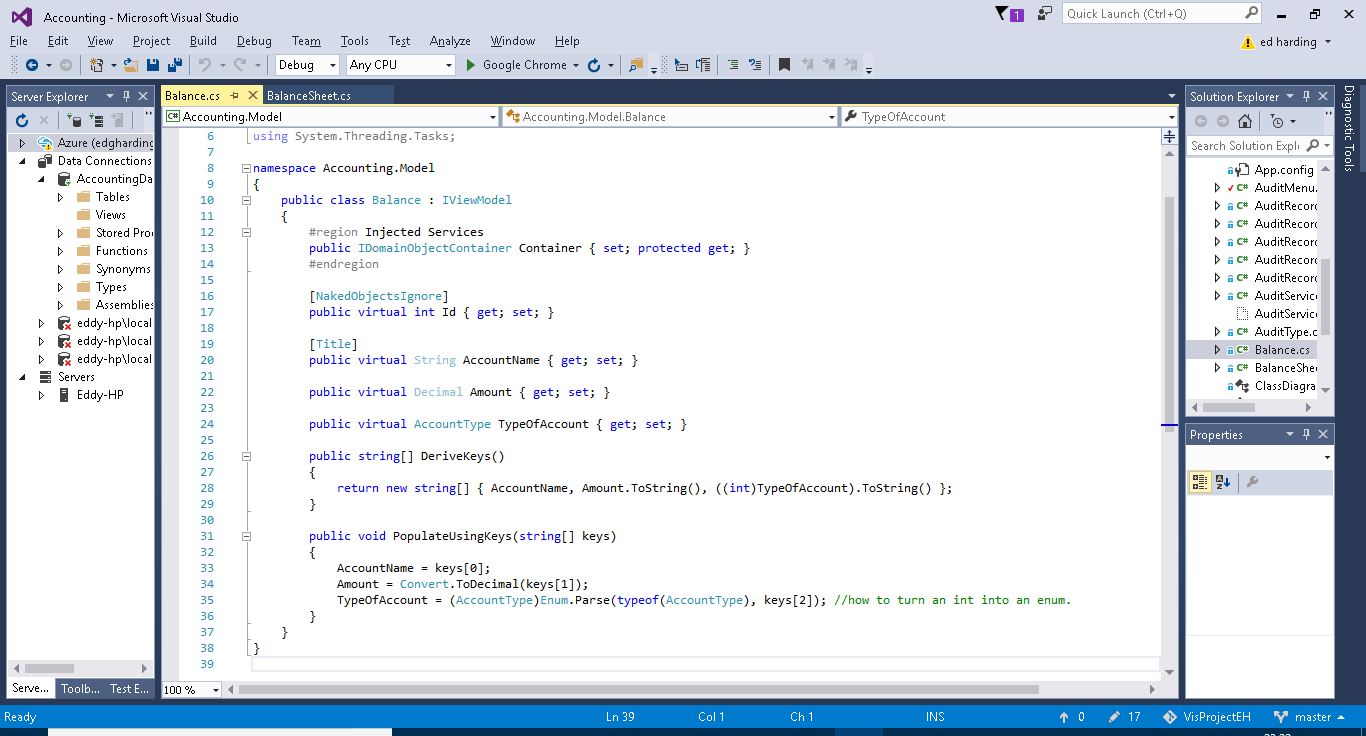
For creating the balance sheet, I needed to get the balance for each account as of a specified date. The balance of an account depends on the different credit and debit transactions applied to it. I could simply have accessed the accounts balance directly from the balance sheet class, however I saw 2 main advantages to using a view model class instead.

First, I will explain what a view model is and then I will move on to the 2 reasons I decided to use one. A view model is a class which appears to the user through the GUI as if it was a persisted object but is not.

1. The fact that a view model is not persisted means that it must instead get the information from somewhere else, in my case this is from the Accounts class, the advantage of this is that the balance sheet is created from all of the previous transactions up to a specific date, therefore the balances need to be based on the balance of the accounts at the specified date. However, it is of course possible to add transactions (which effect an accounts balance) with past dates. Therefore if a balance sheet is created and then a transaction is added which has a date before the date of the balance sheet, this would cause a problem if the balances are persisted objects as the balance sheet will not update and will therefore be inaccurate. Therefore, using a view model causes the balances to be created live every time the balance sheet is opened (how this works will be explained later on).
2. The other advantage of using a view model is that it allows for an increased level of customisation of how the balance sheet is displayed to the user.

These advantages are demonstrated in the code break down which follows.

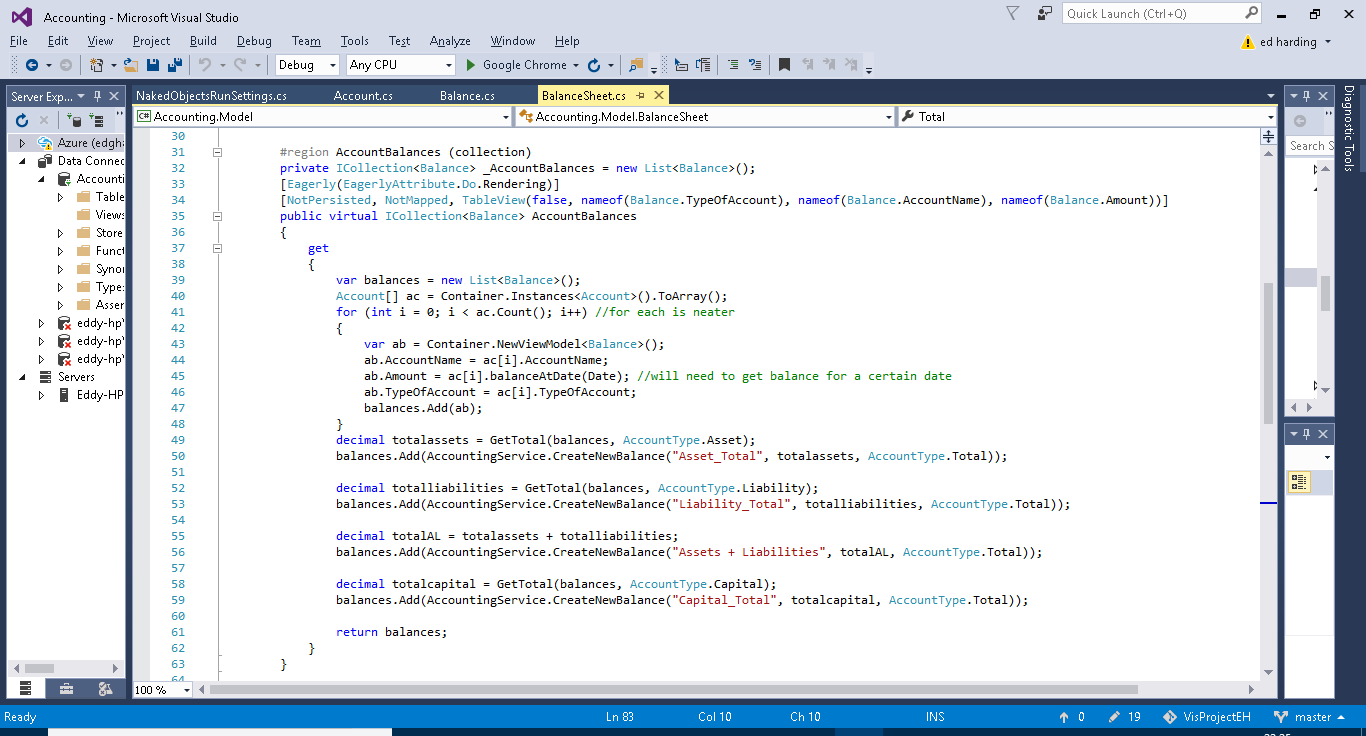
Below is the code for the Balance View model, as you can see it ‘copies’ some of the data held in the accounts class. It achieves this by using two method DeriveKeys and PopulateKeys, I will go into more detail of what they do below.



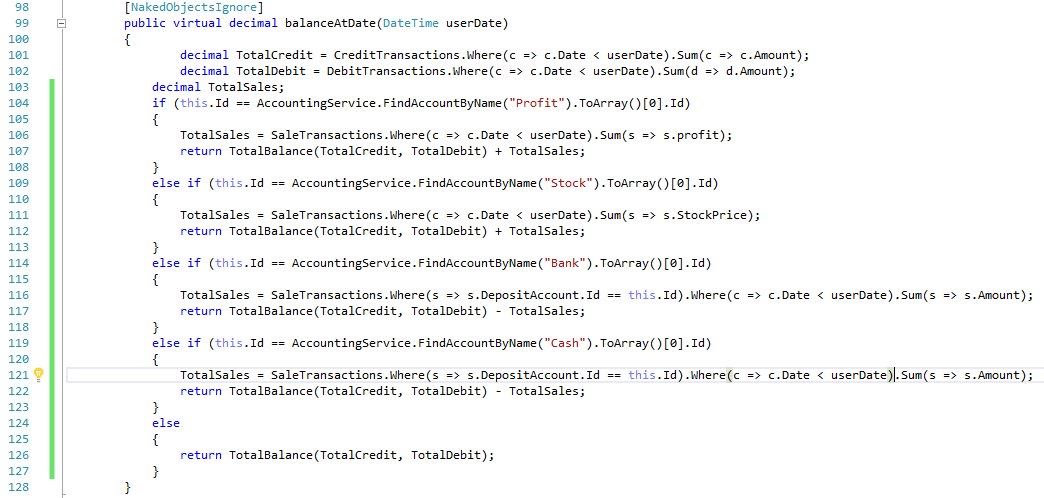
DeriveKeys: This method returns a string array containing the keys derived from the object which the view model represents, so in this case it contains three of the properties from the Account class; AccountName, Amount and TypeOfAccount.

Populate Keys: This method populates the properties of the View Model with the keys which are retrieved by the DeriveKeys method. As you can see the Balance AccountName is being set to keys[0] which in the string of keys returned by DeriveKeys is the AccountName gotten from the Account Object. This is how the View Model is able to be created live every time it is called upon.

The below screenshots are of the balance sheet class and show the use of the balances view model inside the balance sheet class.

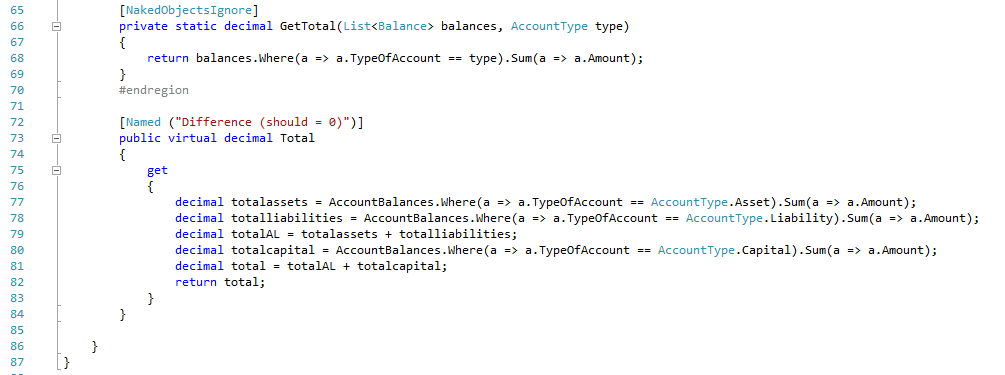


Here you can see a collection of balnces being created, I chose to use a collection as it would be able to create a nice ordered table of the accounts (as the balance class represents the accounts class), this provides and easily readable and follows the typical format of a balance sheet. As you can see above (line 39 to 48) a new list of balances is being created and the balance of each individual account is being assigned to a new instance of balance every time until this has been done for all of the accounts. The balnceAtDate method comes from Accounts and allows for the balance of the account to be calculated at a specific date, the method is shown below, this like a lot of my methods is done using LINQ queries to find the data required.(back to the above screenshot) I am then creating 4 custom balances and adding them to the list of balances. These custom balances are each totals which are used for calculating the overall balance of the balance sheet (Assets +liabilites – Capital), this is done using the GetTotal method which is show and explained below. Again thanks to the View model, and that the overall whole thing is a collection, these will be included in the table of accounts (balances view model).



Below (lines 98 to 128) is a method for getting the total balances of all of the accounts of a certain type, it is pretty straight forward and uses a LINQ query to do this. The If statement section is just to make sure that the SaleTransactions are accounted for when the balance for Profit, stock, bank and cash accounts is being calculated.

Below (lines 72 to 83) is the final part of the balance sheet, the total. If a double entry bookkeeping system has been programmed and used correctly the total of a balance sheet (which uses the equation total assets + total liabalities + total capital) should always = 0. Below is simply the method I created which calculates this from the totals balances which I mentioned earlier. Line 72 is just renaming the property so when it is displayed it reads “Difference (Should =0)” rather than total, as total is a slightly misleading name.

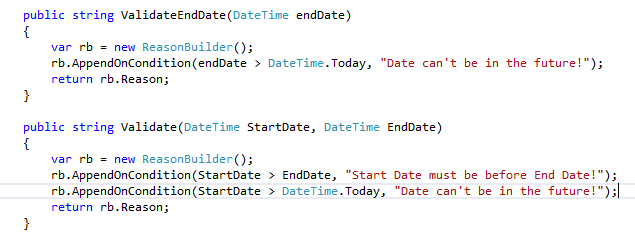


#### Usability and robustness

In this section I will show off some of the code which adds to the user friendliness of the program and increases its robustness, demonstrations of these features will be shown in the testing section.

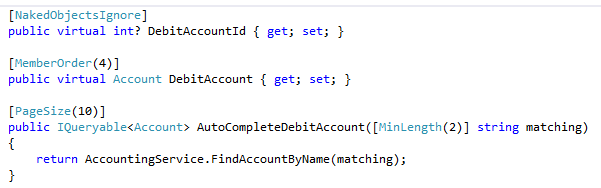
The following are not the only places in which validation or usability is added in the code they are merely examples as the other features follow very similar formats to the following.

The below is taken from the ProfitLossStatement class.



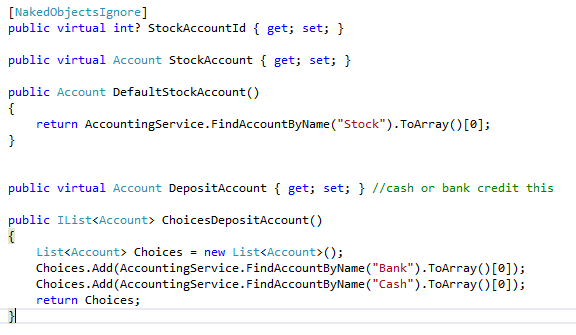
The above code shows two validation methods, one acts only on the EndDate property where as the other acts on both the EndDate and StartDate as it is comparing them to make sure the EndDate comes after the StartDate. These validation methods use a ReasonBuilder which is part of NOF, if the condition is met then the reason is added and all the reasons for any mistakes are returned to the user so they can fix any incorrect data entries.

The below is taken from the Transaction class.



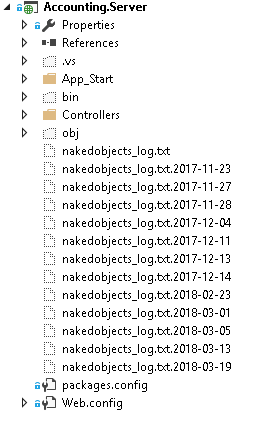
Above is an example of the auto complete feature primarily used in creating new Transactions. As you can see the auto complete method takes in a string of minimum length 2, it then enters the string into the Find Account By Name method from the Accounting Service and returns the option in a drop down list for the user to select, it will show a maximum of ten options in the drop down list.

The below screenshot is taken from the SaleTransaction class.



Above is an example of a default value and a restricted drop-down list. The default value simply works by using the Default method and returning whatever value wanted to be the default. In this case I wanted the Stock Account. The drop-down list is similar however you just return a list of choices, in this case the cash and bank accounts, the user will only be able to select one of those options.

### Server

(the nakedobject\_log.txt’s contains crash and warning information from previous errors)

As previously stated the solution follows the client-server relationship. Eventually the idea behind this architecture is that the server project would be deployed via a webserver such as azure so that it could be accessed anywhere. The project uses REST which helps communication between the server and client. It does this via a RESTful API, this allows for the client (the UI) to talk to the server (the object modelled database) to request and receive data. It does this using the http commands GET, PUT, POST and DELETE. REST also means that the database can be traversed through from one start point down to the lowest level, I have provided some screenshots below to show the visual representation of the RESTful Server.

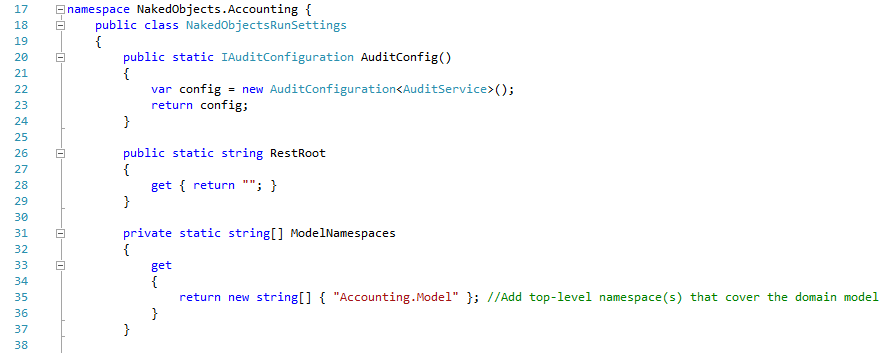


Above is the top layer and below is an account specifically the share capital account, as you can see I was able to navigate my way there fluently in the same way which the client server model does. (the below screenshot only shows part of the full account).



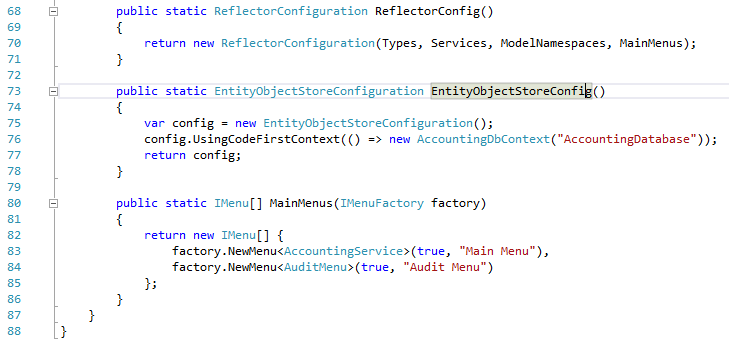
I will point out that during all my testing and use of this project the Server was completely run locally on my machine. There was not much in this project which I actually needed to edit other than configuring the naked object run settings.

There is some green annotation on the screenshots below.



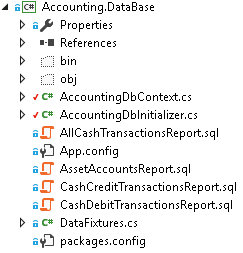


Line 56 Above is for defining which classes are services.



Line 80 above is for defining men’s from repositories/services, as you can see both my menus and the source data for them are listed.

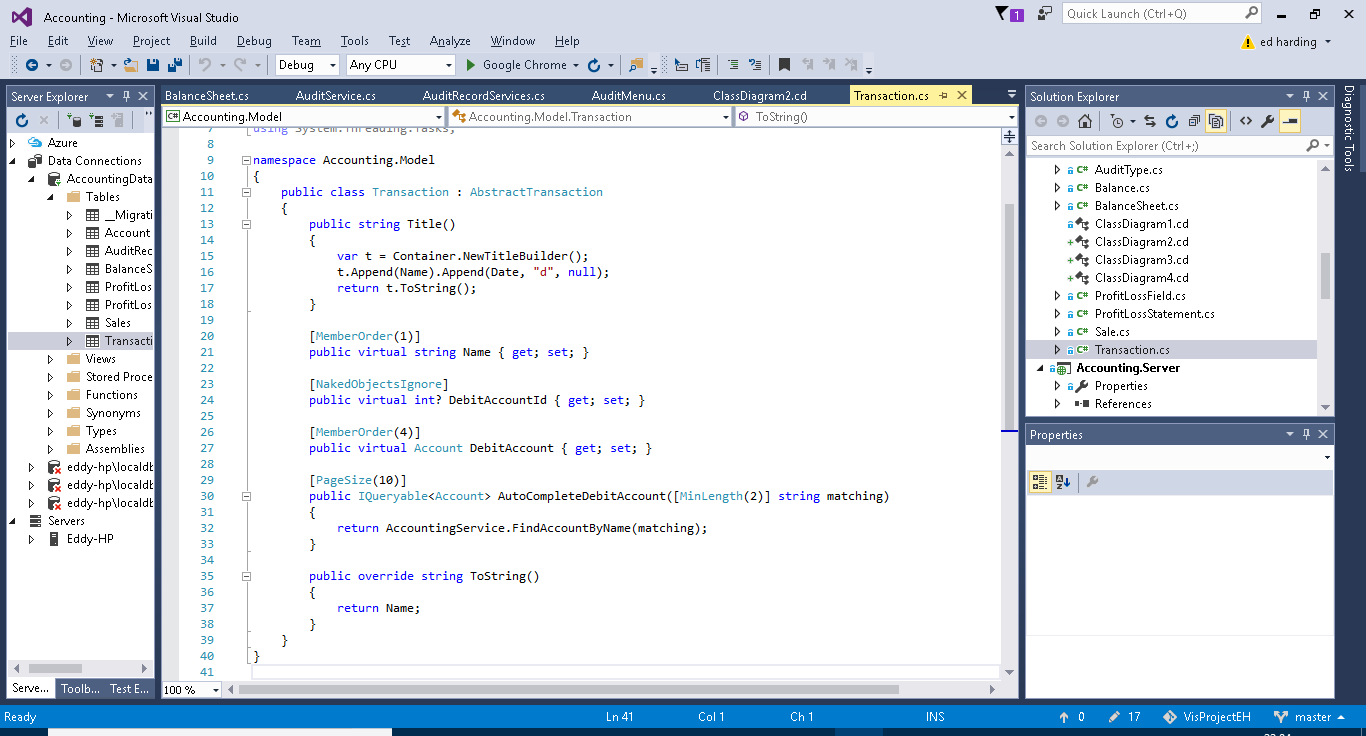
### Database



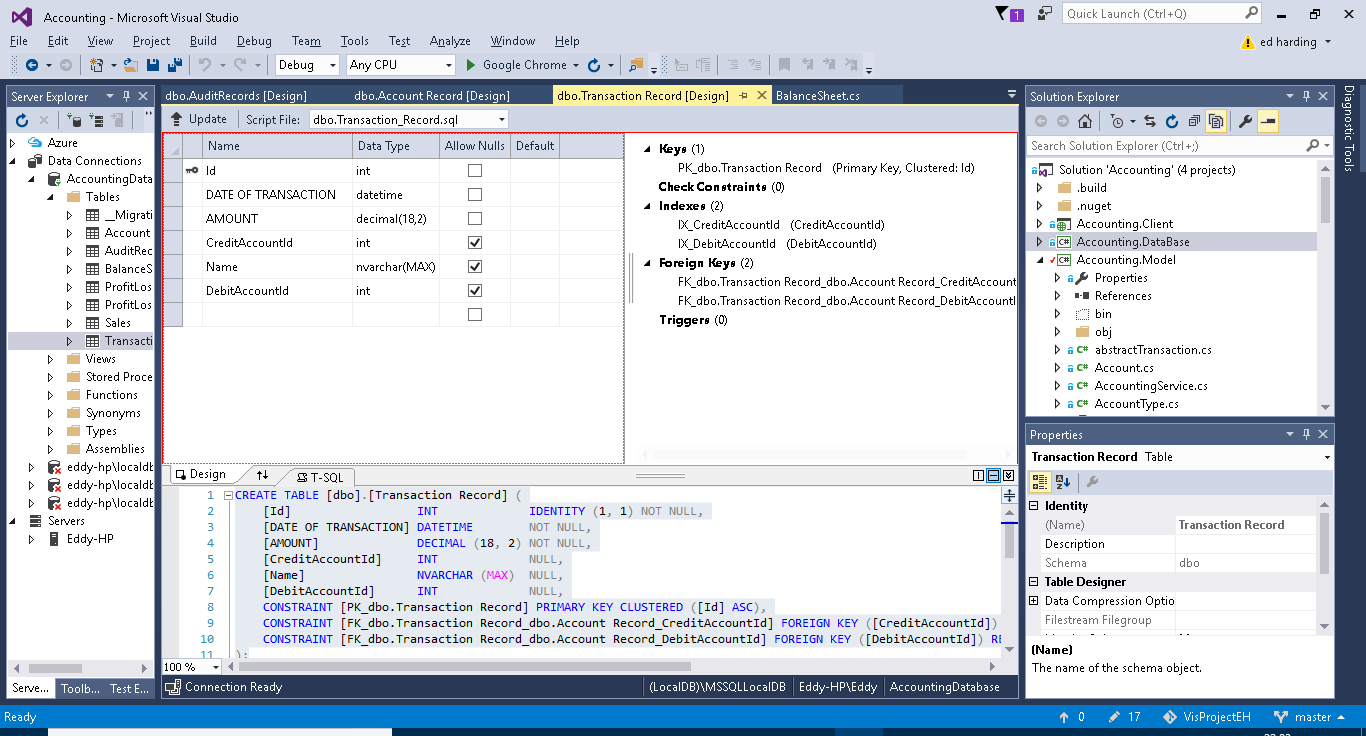
#### Entity Framework and SQL Servers

Entity Framework is an integral part of the project as it acts almost like a translator between the business-based objects I have coded and the SQL server database. Entity framework has the ability to format and map user made objects to a database by creating an Entity Data Model from the objects it does this by using a technique called object relational mapping. ORM is a method for converting data of incompatible types into a virtual object database. Entity Framework is a Microsoft product therefore combined with Microsoft Visual Studio there should be little bugs in this process. This therefore reduces a whole work load for the developer. It also allows the developer to perform LINQ queries in C# on the database. This is extremely useful when creating searching methods and such.

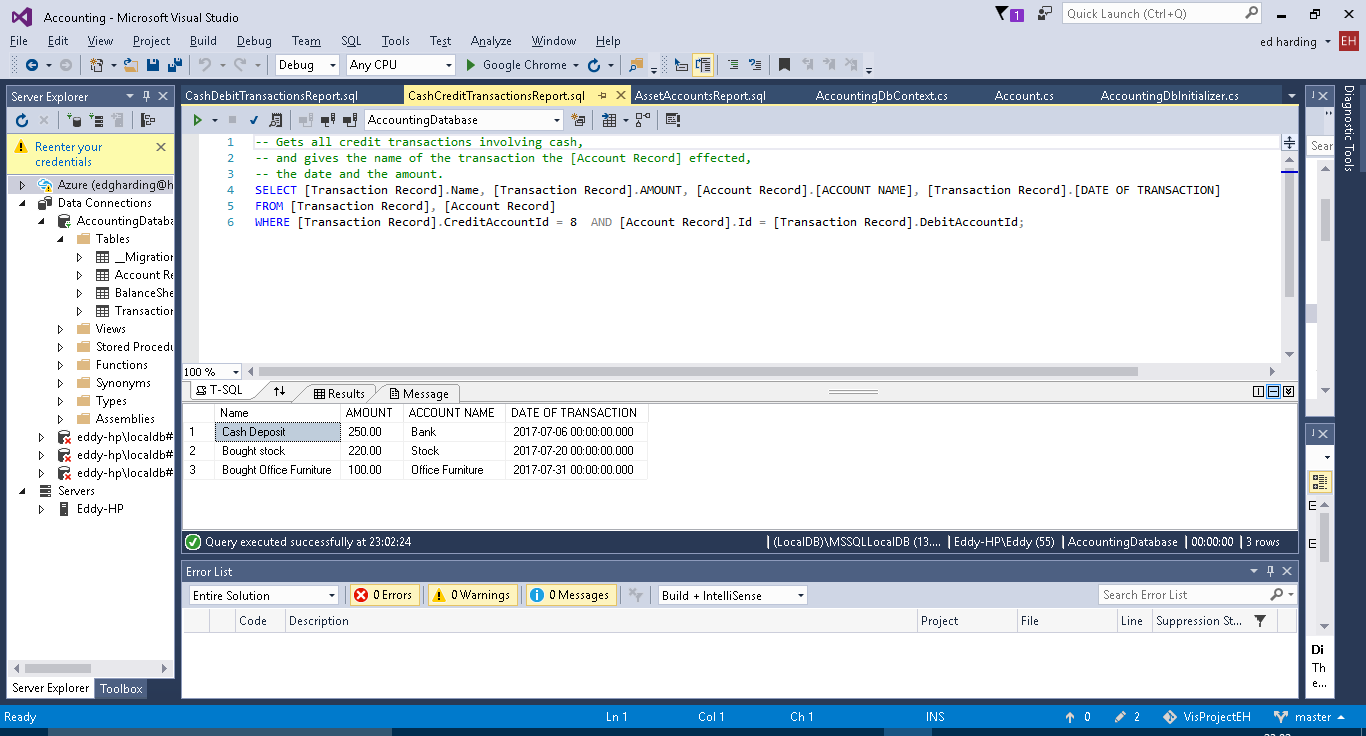
SQL Server is a relational database management system, which as mentioned earlier is used in tandem with Entity framework to model and query a database built from an object orientated project.



Above is the Transaction Object from my solution and below is the same object however in a tabulated form viewed via the database explorer.



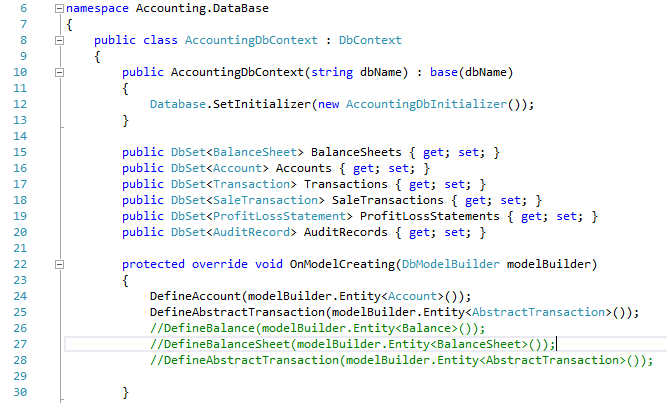
Below is a SQL query I wrote (what it does is explained in green in the screenshot). Below the SQL is the return message.



#### DB context

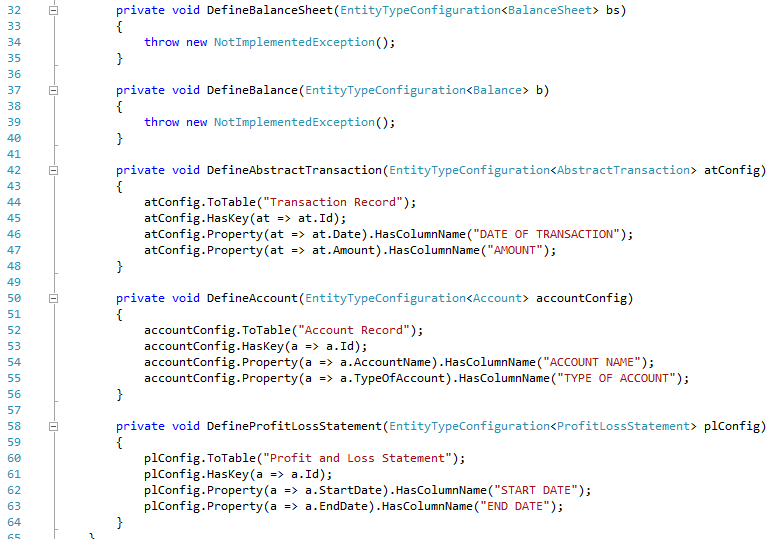
The DB Context class is used for specifying which classes need to be persisted and it is used for adding methods which allow for the customisation of the tables created by entity framework.

On lines 10 to 13 below there is a method which just tells the server which Db initializer (explained further on) is being used, this is because you could theoreitcally write lots of different ones and then call the one you want by changing these lines of code.



Above from line 15 to 20 are a list of the different classes which need to be persisted inside the database, this basically just points out the classes to entity framework so it knows which ones to model. It also allows for instances of these classes to be created from inside the DB initializer.

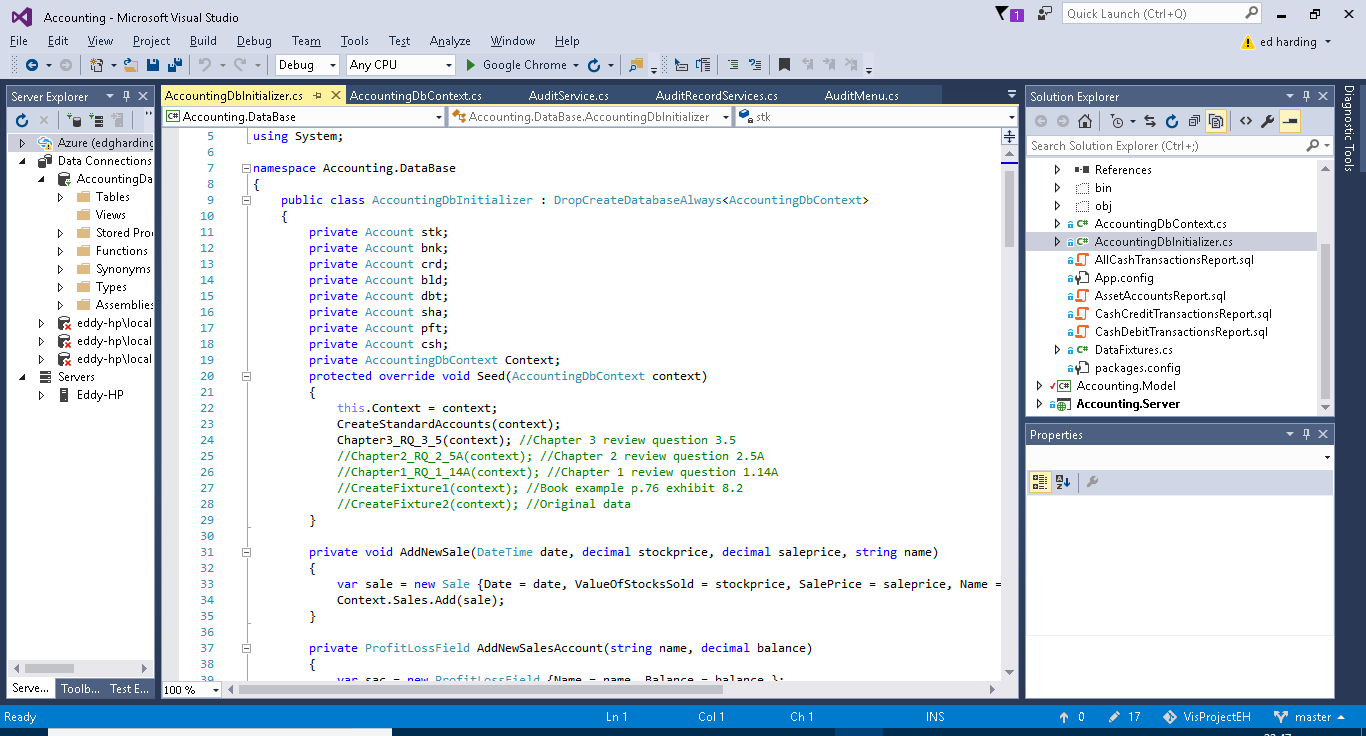
Above from line 22 to 30 is the method which overides the entity framework tables with the methods listed inside it, these methods are shown and explained below. The commented out lines are methods shown below on lines 32 to 40,as you can see I did not impliment anything here and therefore have commented them out in the overide method.



Lines 42 to 64 contains 3 methods, each do the same thing just for different objects. As mentioned previously these methods are for customization, the tables which entity framework creates. These methods simply help to format the tables created, as you can see the key for the tables is being set to the ID of the class, and then the statements following that are just giving the columns more appropriate names. I mainly did this for the use of the SQL statements, it means that when a table is returned from a SQL report it will have more user-friendly column names etc.

#### DB Initializer

The DB Initializers purpose is for seeding data for the project, this can be for testing or providing a default start up for the user. Below are some screenshots of the DB initializer which I will explain the aspects of.



Above:

The first thing to talk about is the DropCreateDatabaseAlways on line 9. This line simply means that each time the solution is run the old data from the previous session will be discarded and the seeded data held in the DB initializer will be added again. This means that the state of the solution is not saved. This is currently set this way for testing purposes. It is also possible to change this to DropCreateDatabaseIfModelChanges, this means that the data will only be reseeded if the Model project has been altered.

From lines 11 to 18 I have simply declared some accounts which are used regularly throughout my examples such as sha (shares) and bnk (Bank). Below this is where the data is being seeded. The Data in CreateStandardAccounts is added followed by one of the few options of data I have provided below that. The ones I am not using are commented out in green and the comments to the right of them just explain where the data used in them has come from. This system allows me to switch between different test and example data very easily.

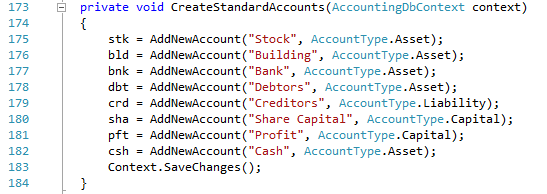
Below:

The following screenshot just shows some methods I needed to define for creating instances of the objects.



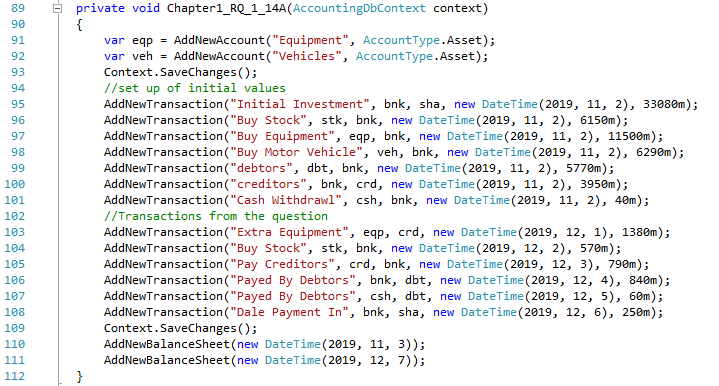
Below:

Here is the data used for the CreateStandardAccounts which I mentioned earlier, this uses the pre-declared variables from the top of the class file, it was necessary to declare the accounts outside of creating these data banks as the standard accounts are used in the test data, and therefore it had to be declared before use.



Below:

This is one of the multiple different examples of test data I added Here you can see the addition of new accounts, the addition of mutliple transactions and finally the creation of a couple of profit and loss statements dated before and after the transactions, to show the effect of the transactions.

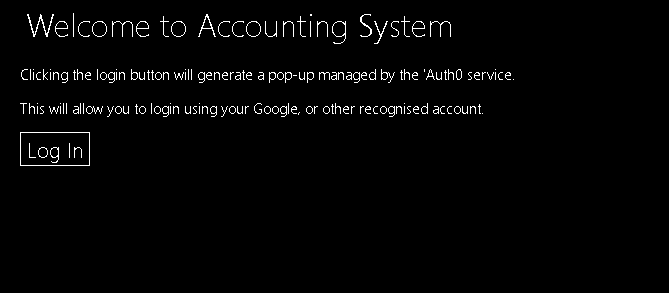


# Testing

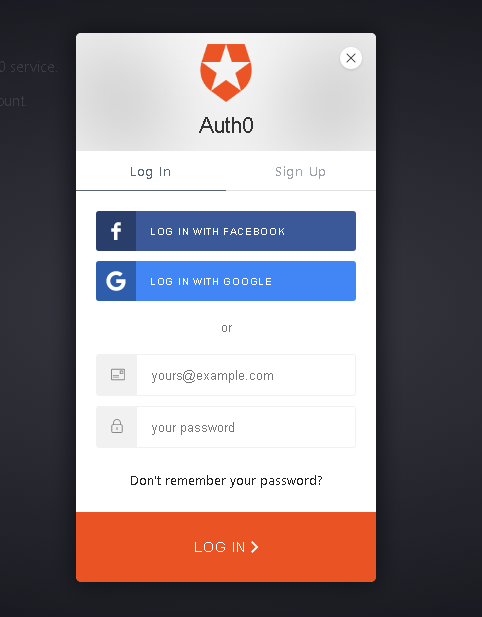
## Test 1

Story: A user logs onto the system and creates a balance sheet for today’s date (the date of which this test was run).

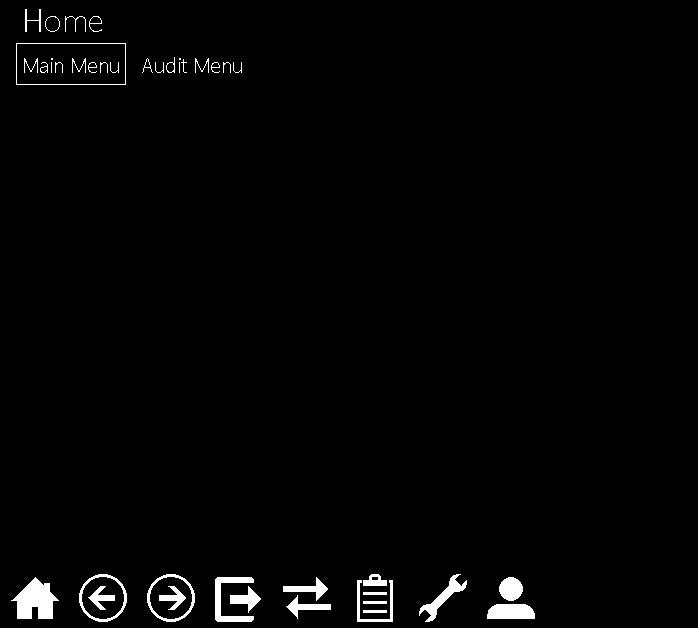
This is the startup screen:



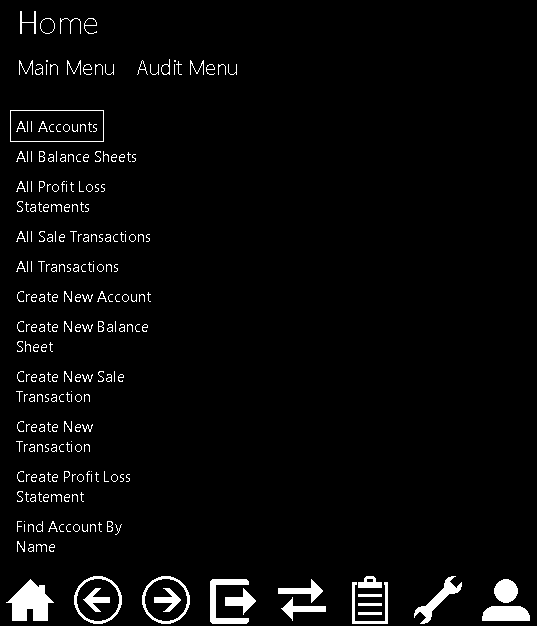
When Log in is clicked the window below pop’s up.



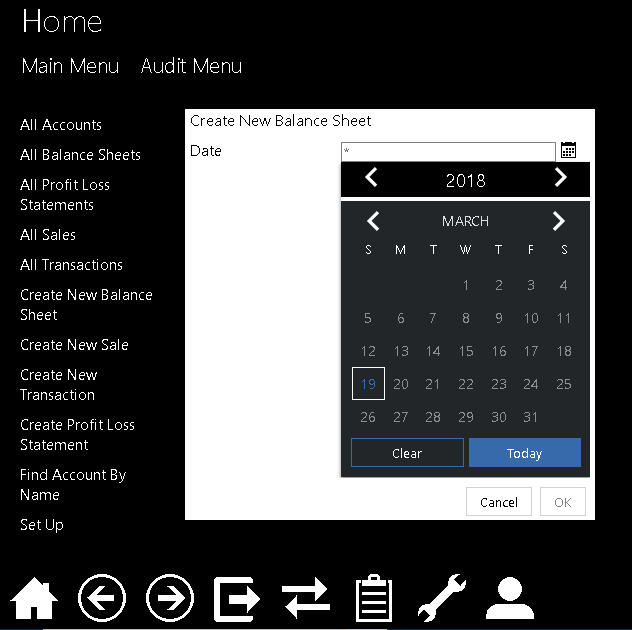
After logging in with valid credentials the Home screen (below) is displayed.

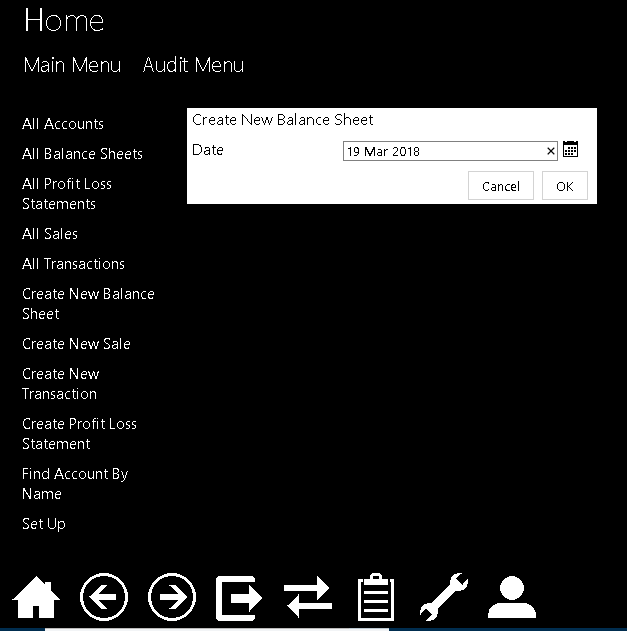


The user can select between two different menus, after selecting Main menu the below options are shown.

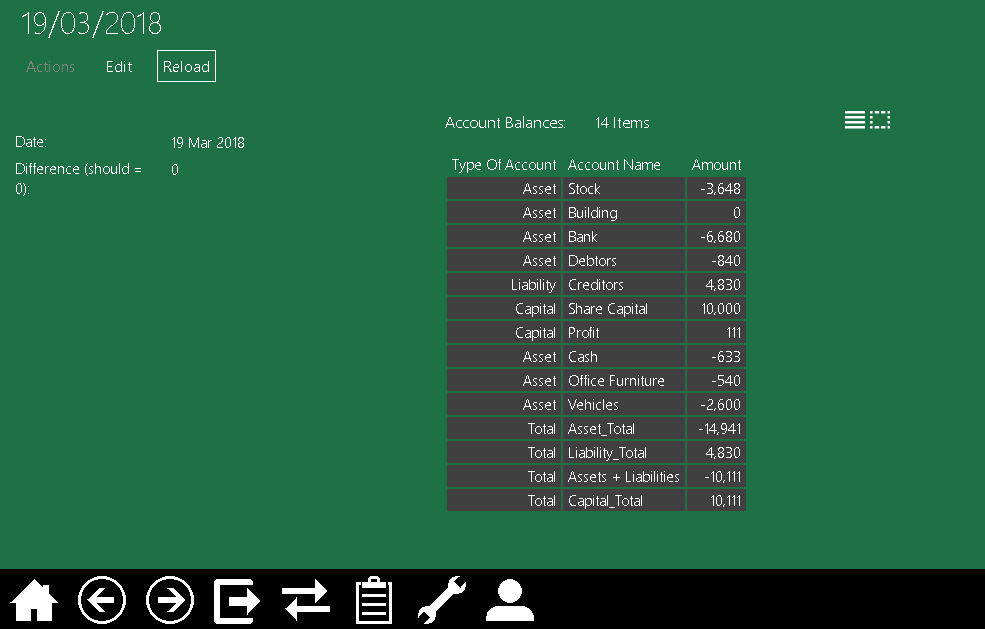


When Create New balance sheet is clicked the below form pop’s up and requires filling.





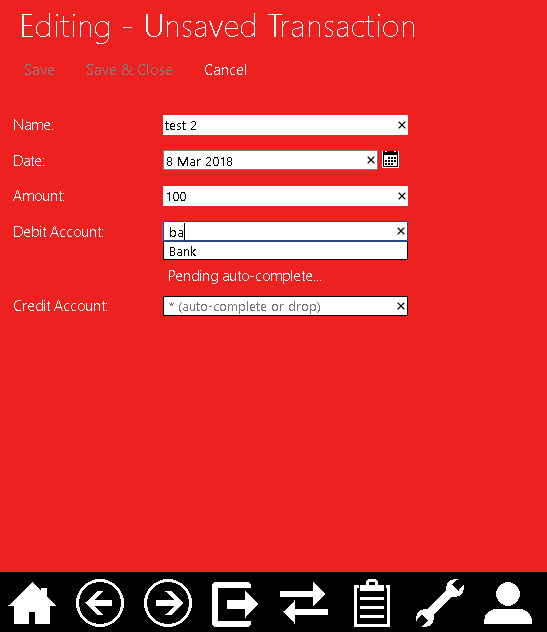
The form is filled and ‘ok’ is clicked to create the balance sheet, it is then displayed as shown below:



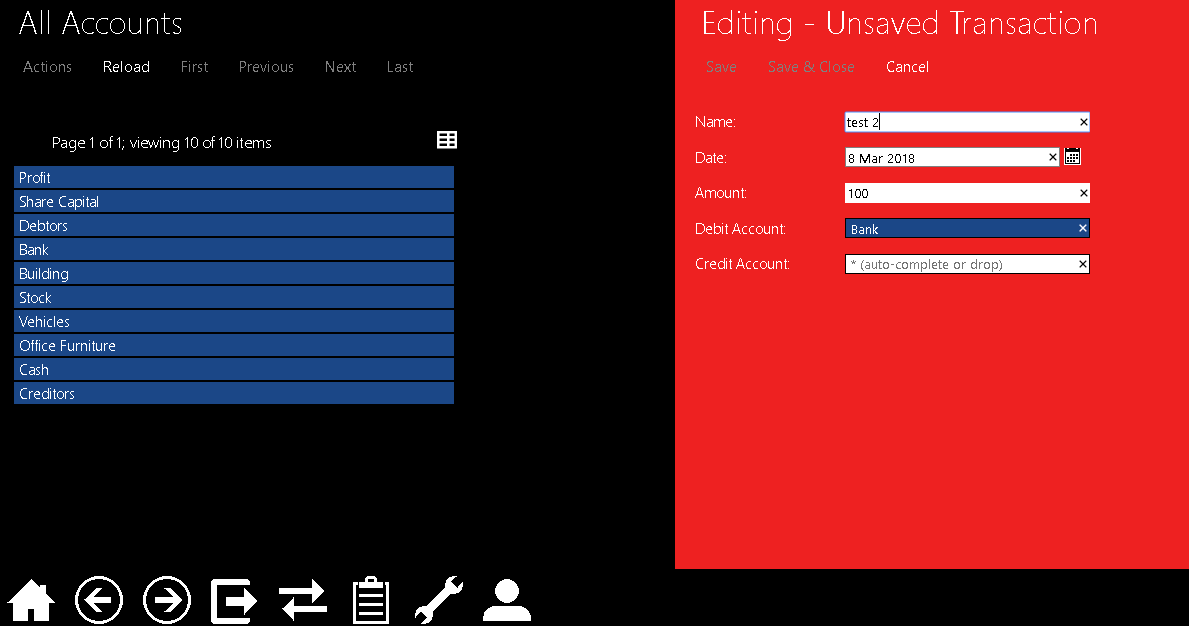
## Test 2

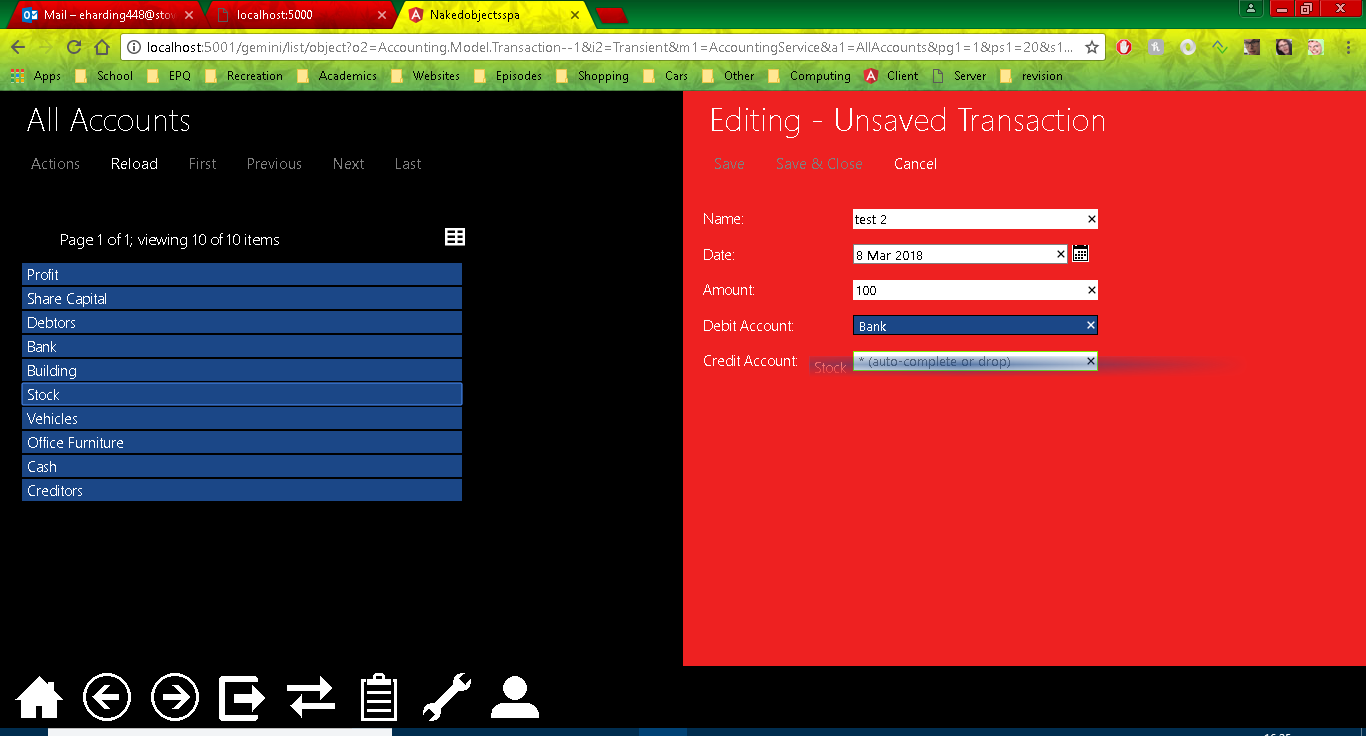
A transaction is created with a date before the date of the balance sheet which was created in Test 1, then the balance sheet is viewed showing that the balance sheet has updated to take in to account this new transaction (this is thanks to it being a view model).

Below you can see I have filled out most of the fields, but here you can see the auto-complete function in action when selecting the Debit Account.

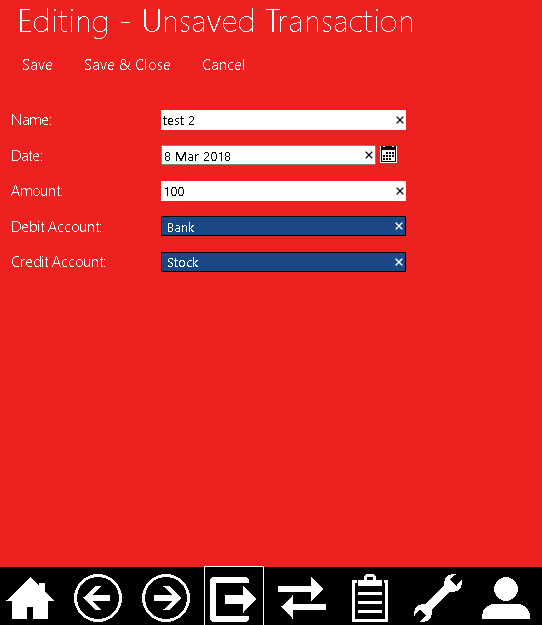


The below two screen shots show using the split screen NOF option to drag and drop an account (listed in the all Accounts menu option) into the Credit Account box.

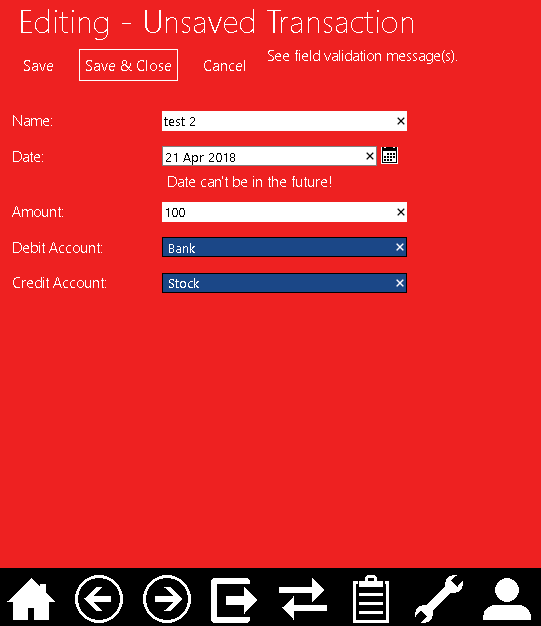




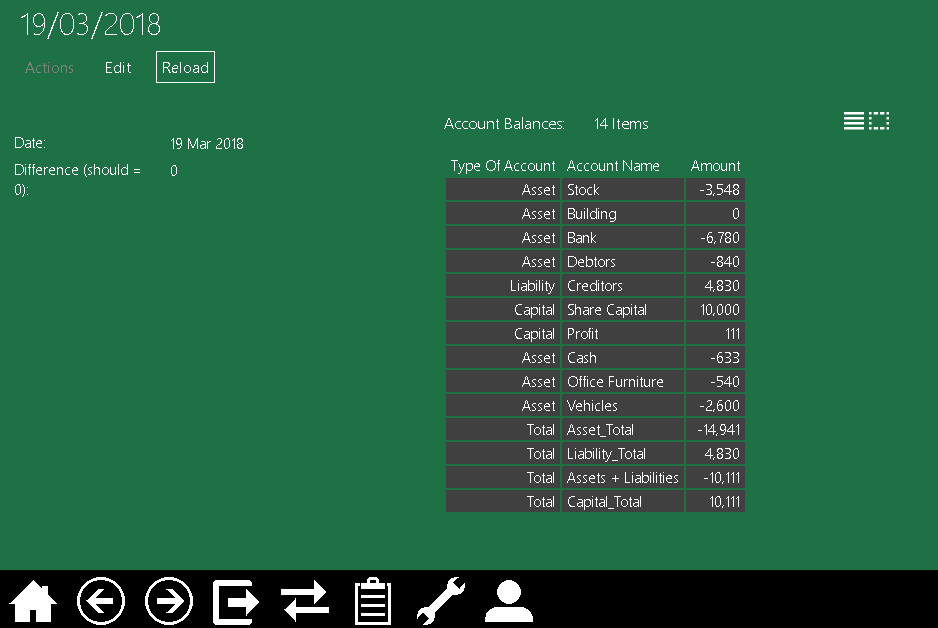
Below is the final filled out transaction before it is saved.



Here is an example of validation, as you can see I have tried to create a transaction with a date in the future, it has blocked me from doing so and pointed this error out.



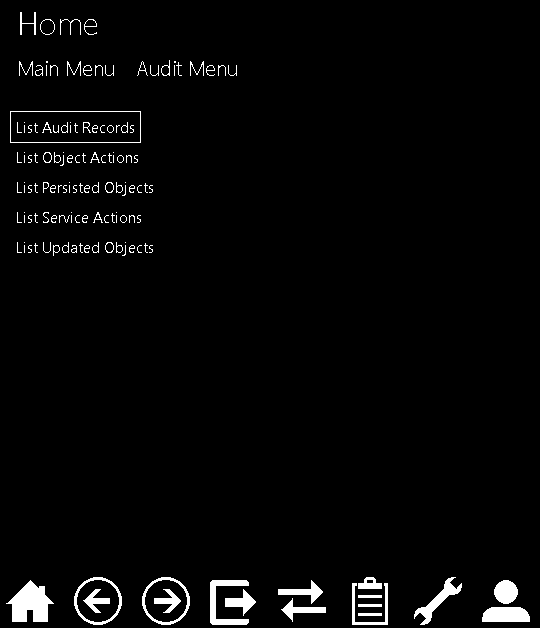
Below is the balance sheet from test 1 however as you can see the amount of the values and stock accounts have changed due to the added transaction.



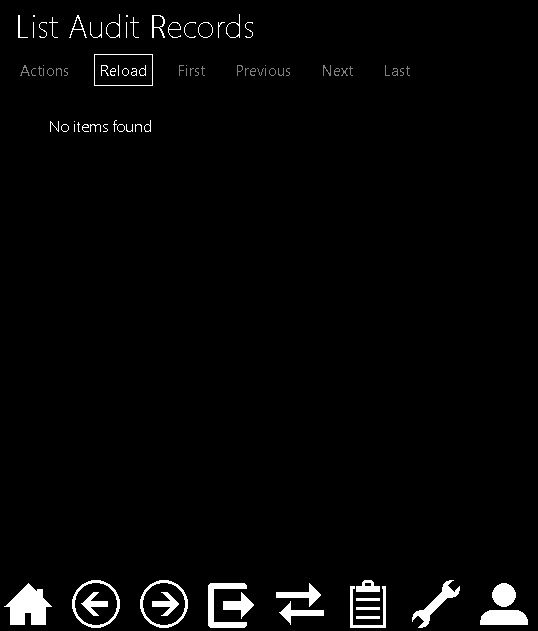
## Test 3

This test shows off the Audit system and how it can track everything the user does which is relevant (i.e. anything which isn’t just querying/viewing the program).

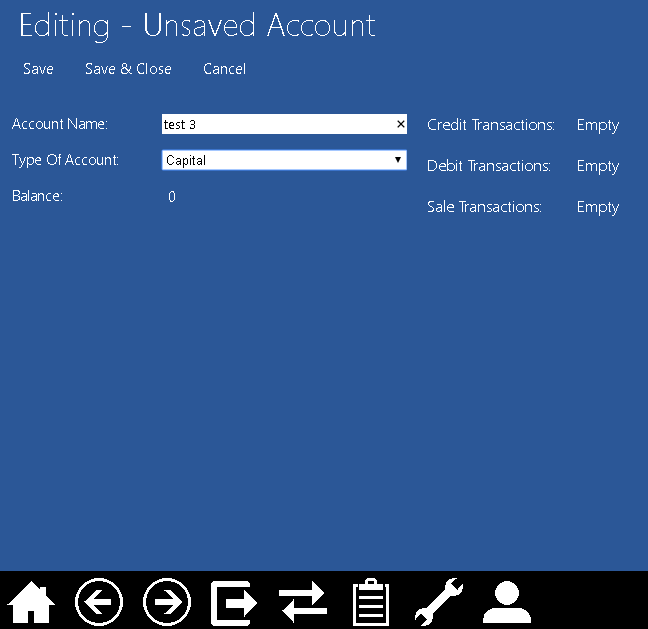
From the Home screen the Audit Menu is selected, this is shown below. The first option shows all of the Audit records, the four options after that just show specific types of Audit Records like only Audit Records which are concerned with objects which have been persisted.



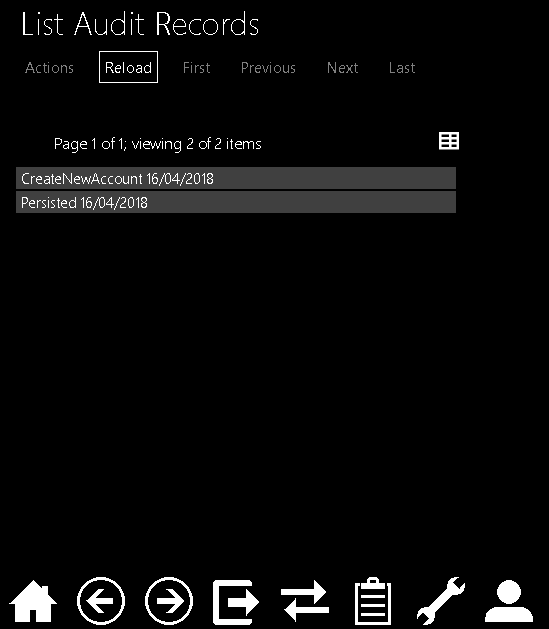
Below is the empty Audit Records, although there is data in the program as this was added inside the code and not by a user it has not been audited.

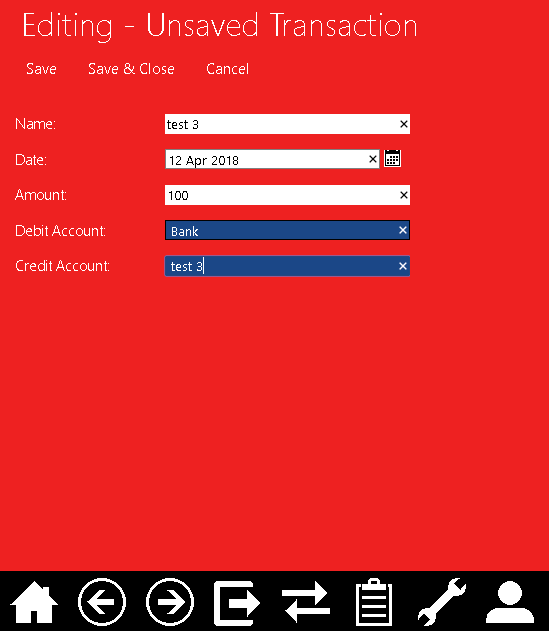


Here is the creation of an Account.

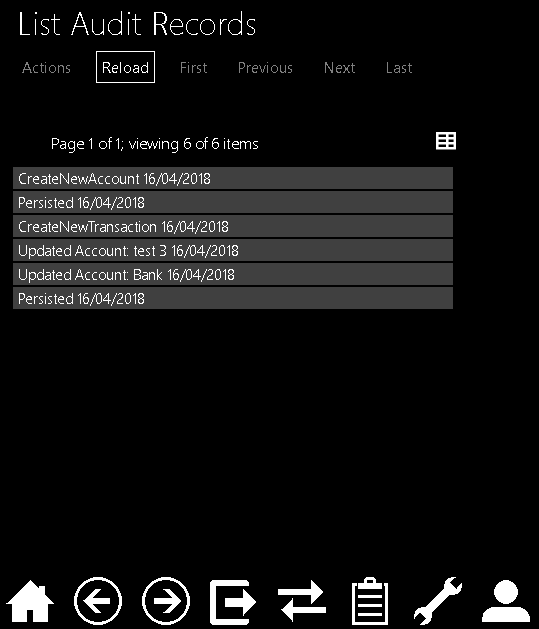


After the account is created, the Audit Records has updated to show the saved records about what happened, as you can see it has picked up that the CreateNewAccount action was invoked and that the object was then persisted.

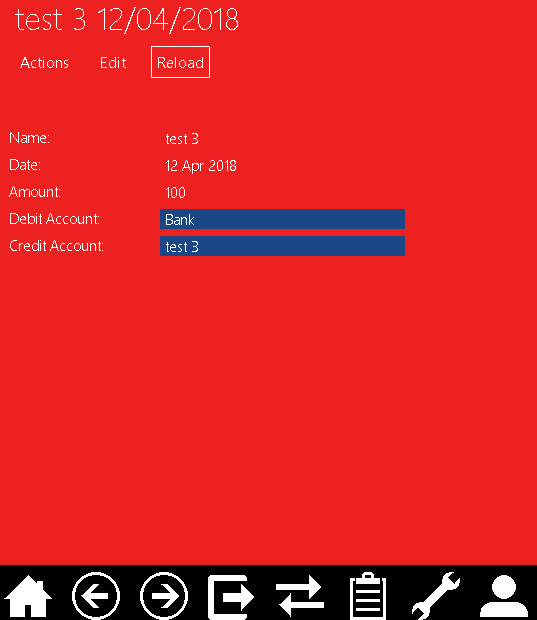


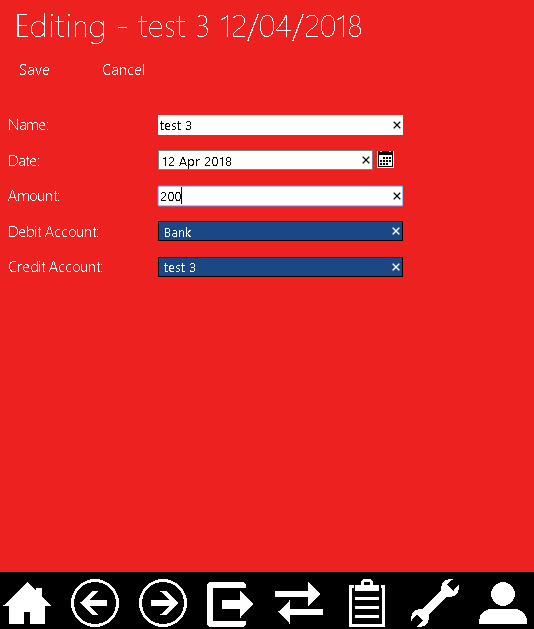
Below is the creation of a new transaction, using the new account as the credit account.

As you can see below there are now more Audit records created from the creation of the transaction.

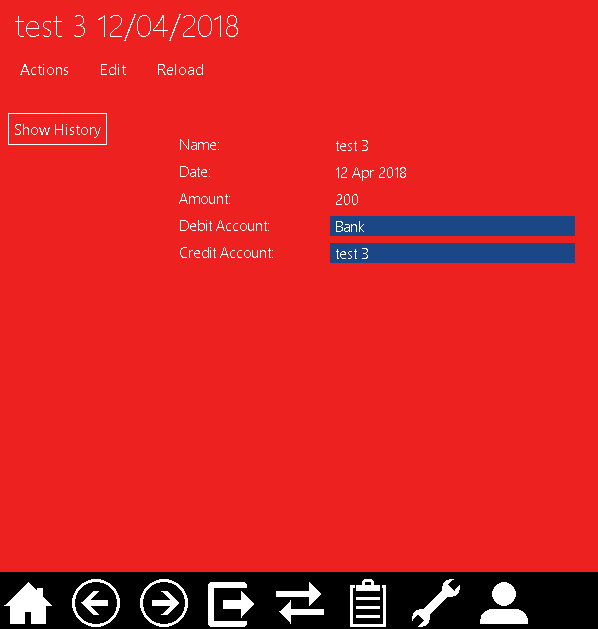


Let’s say a transaction was created and some data was entered incorrectly, for example the amount of money in the transaction, the transaction can be edited as shown below.

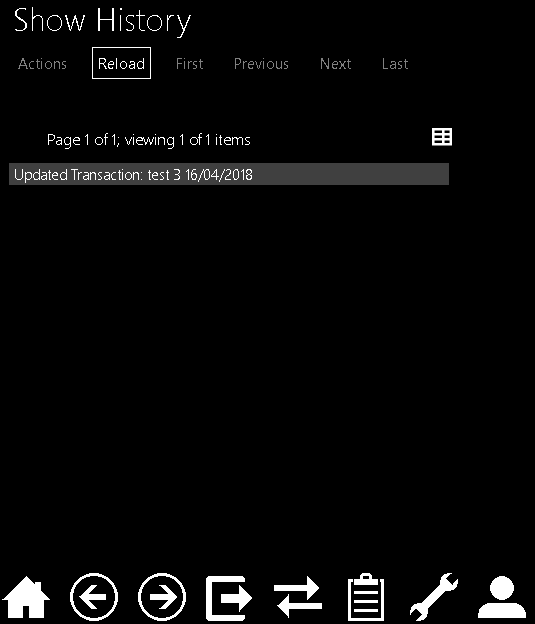




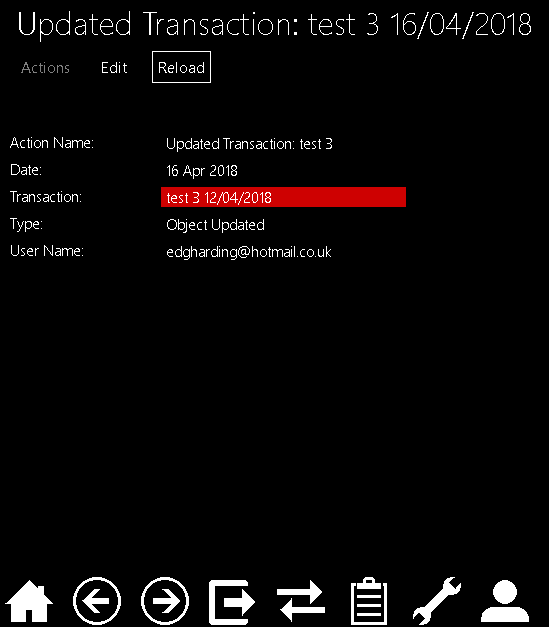
Editing a transaction is something which could cause problems for a business therefore the edit is recorded, and each transaction contains the show history action which can be shown in use below.



As you can see below the Audit record showing that the transaction was updated is shown below.



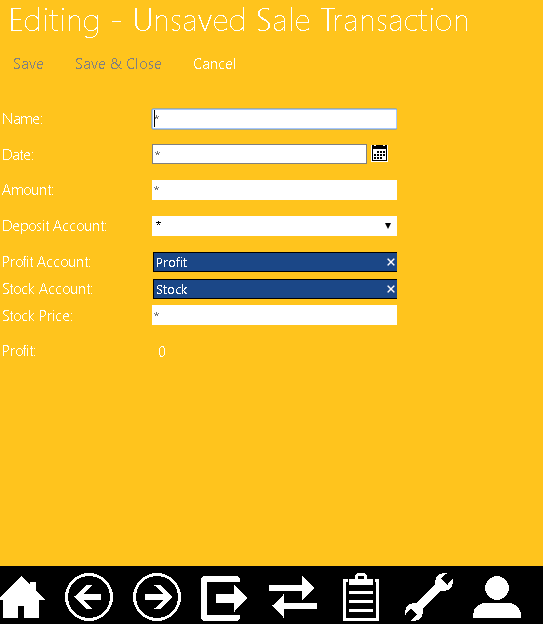
Below is a look into what an audit record might look like specifically the one for the transaction update, as you can see it contains the date of when the event occurred, the action, a link to what was affected (the transaction in this case) and who performed this action.



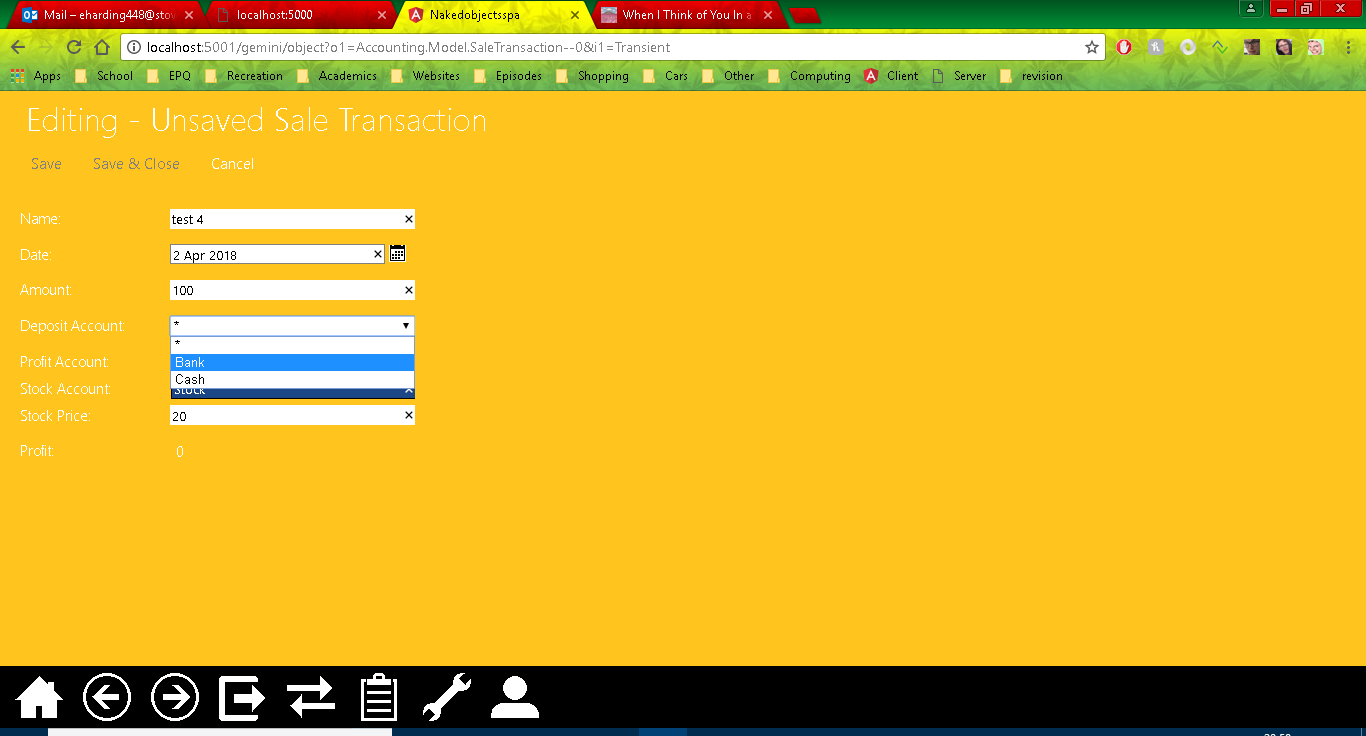
## Test 4

In this test I will simply be creating a sale transaction, this is something the user will probably be doing a lot of as it is how they should record all their actual sales.

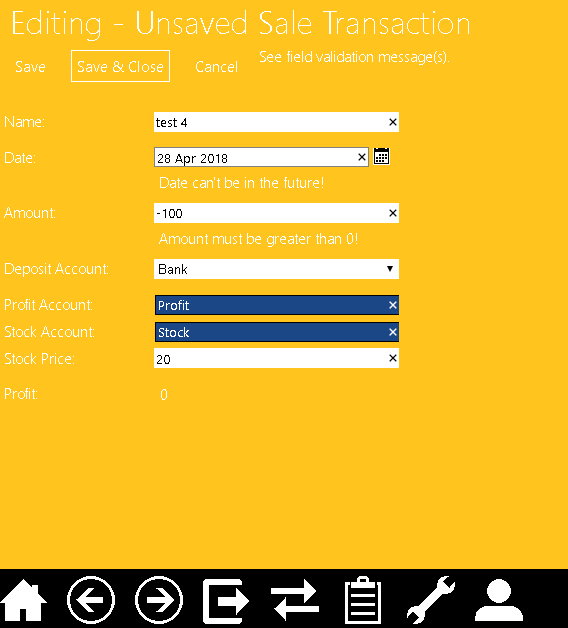
Below is what the user will see when they select the create Sale Transaction option from the main menu, as you can see the Profit and Stock Accounts are prefilled to the accounts they should be. The profit is currently 0 as it is calculated after the Transaction is saved.



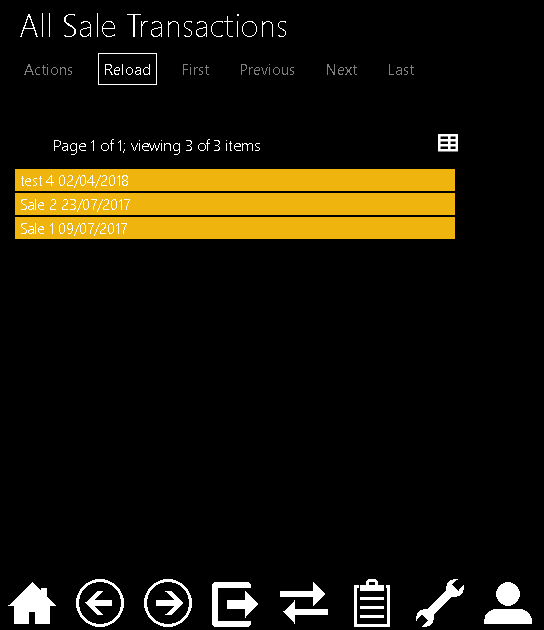
Below you can see the drop-down list for the deposit account, the user can only select bank or cash accounts for the deposit. The other fields have also been filled out.



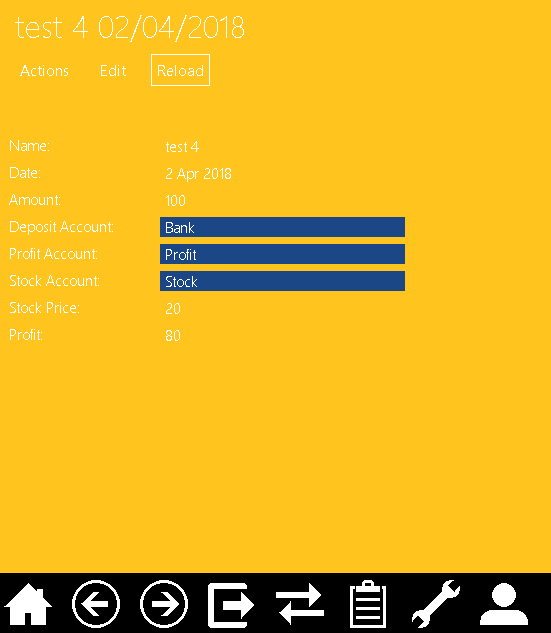
Here are some more examples of validation, again the future date has been blocked as you can’t record a sale you haven’t mad yet. It also blocks the use of negative numbers, as you aren’t going to pay someone to take your stock.



Below is the list of all the Sale Transactions, Sale 1 and 2 are from seed data.



Below is the sale transaction itself and as you can see the profit has now been calculated.



## Test 5

This test will show off the profit and loss system.

Below is the creation of a profit and loss system, however the dates entered are future dates and the start date is after the end date.



As you can see below the program picks up on the future date first getting the user to change that.



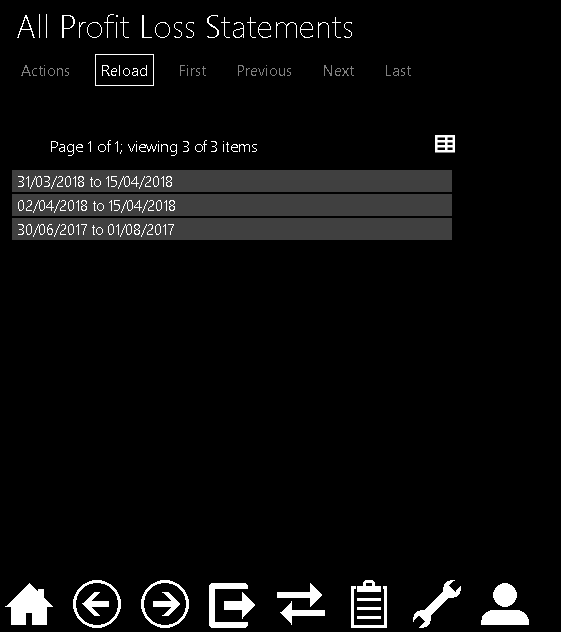
Once that is changed it picks up on the start date errors and tells the user what is wrong with it.



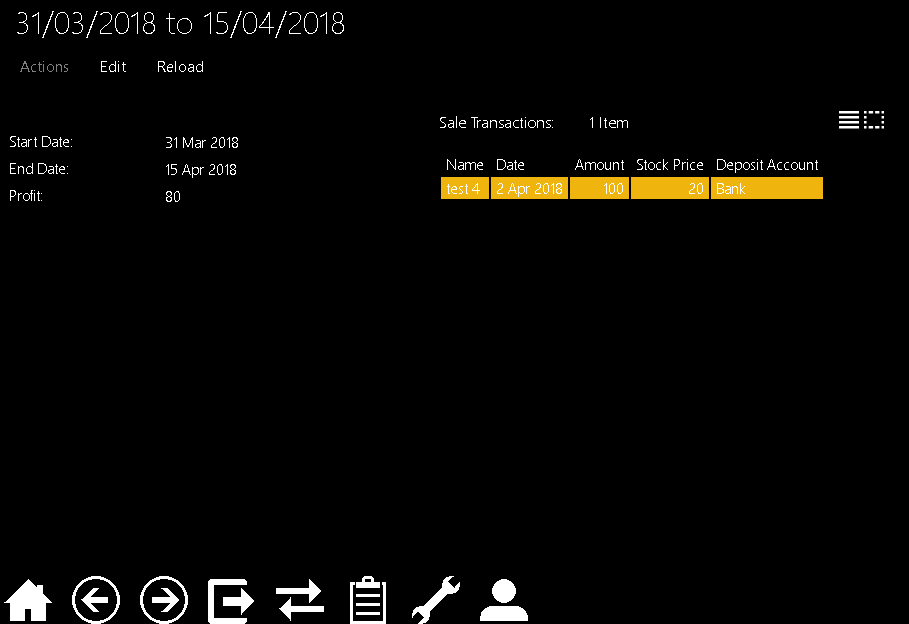
Below is the valid profit and loss statement.



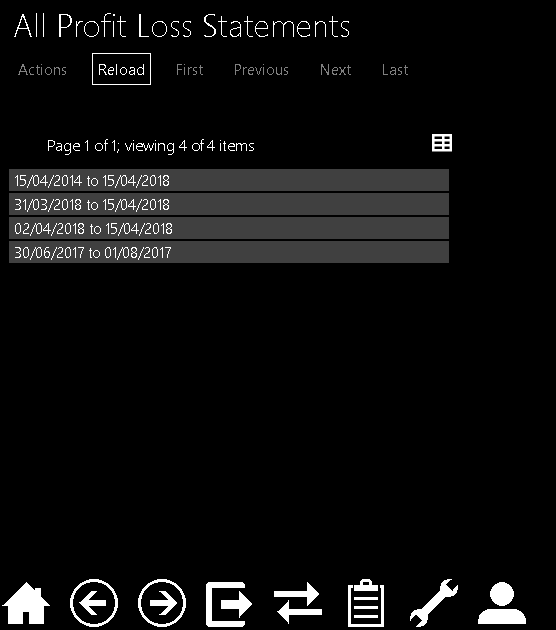
Once created the profit and loss statements can be viewed from the main menu option as shown below.



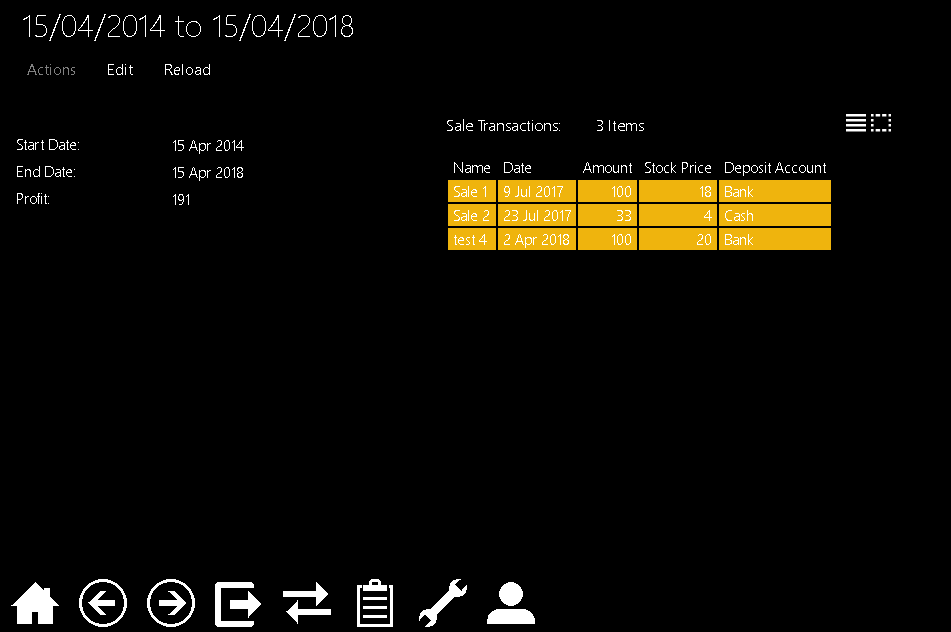
Below you can see the profit and loss statement created earlier which contains the sale transaction created in test 4. As you can see the Profit and loss statement has calculated the total profit.



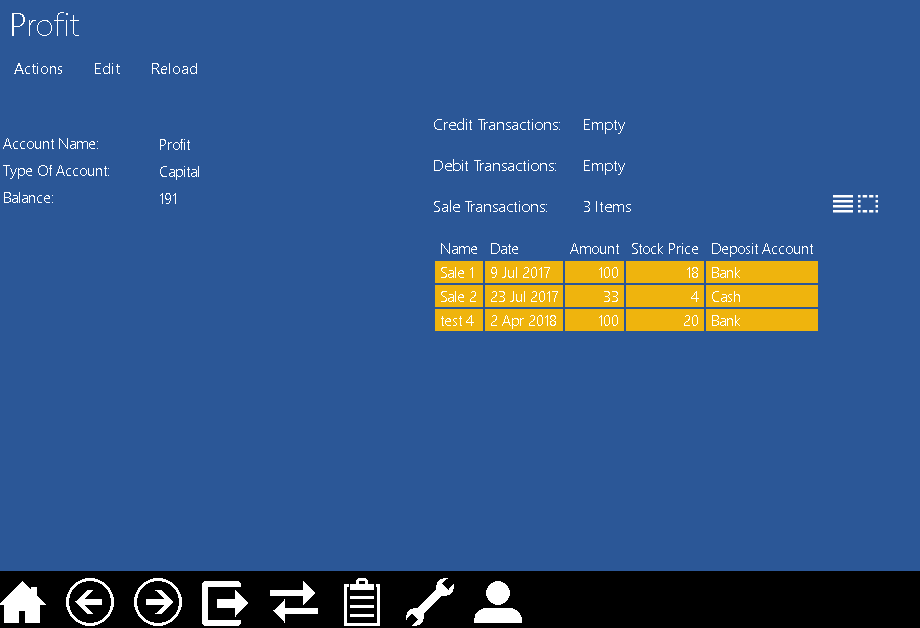
Below you can see I have created a new P/L statement which spans enough time (an entire year) to contain all the sales Transactions,



Below I am looking at this statement in more detail and you can see it has a collection of all the sale transactions and the calculated profit.



Here I have looked at the profit account and again you can see that the balance is essentially equal to the total profit, the difference is the Profit account will always take every sale transaction into account rather than those between specific dates.



# Evaluation

The Criteria I set out in the Analysis were as follows:

1. The program should be able to perform the basic functions of a double entry bookkeeping system, this includes but is not limited to:

* Allowing the user to create multiple different accounts.
* Allow the user to record transactions between the accounts.
* Use these transactions to form a balance sheet.
* Use transactions to make up a profit and loss statement for a requested time period.
* The program should also have the ability to audit user’s actions.

1. The system must be easy to use, simple to understand and navigate, and display data and information in a way which is tidy, efficient and easy to understand.
2. The system will NOT require any authentication process as It will be a locally run program therefore it will only be accessed from the clients own computing device.
3. The client would like the system to be as aesthetically pleasing as possible; as it will be used for long hours therefore it should not be dull.

As demonstrated in the testing section My program completes all of the requirements indicated in point 1. These are the basic requirements of a double entry bookkeeping system and I believe my program is capable of performing its function in that sense.

Concerning point 2 I believe my program is very straight forward to use, the two menus for auditing and the main program help to keep the two sides separate. Everything can be accessed from the main menu and important objects are coloured to allow them to be easily distinguishable from one another. The program also sports some helpful features like drag and drop, auto complete, drop down lists, default inputs and validation messages for when the user has tried to persist something invalid. However, I think I could definitely improve on this point, validation can always be extended further and further making it easier for users to realise what they might be doing wrong and potentially protecting the company from bookkeeping errors. I also think I could have perhaps separated the menus into groups underneath main menu such as Accounts, Transactions, Balance sheets etc. and then have the relating menu options under those menus, this would have allowed for easier navigation and a tidier menu. Another improvement I could have made would be with the History on transaction in the Auditing, I feel like this could have been extended to work for all objects (accounts etc.) and have it save a version of the instance ‘pre-edit’ to allow the roll-back of the edit.

Regarding Point 3 even though in the analysis I said authentication would not be needed I soon realised I needed at least a user log on so that the auditing could record who did what.

Finally point 4, as mentioned previously and as shown the coloured objects helped with this a lot. The main part of the display is quite dark, however with the mix of the brightly coloured objects and the non-clashing contrasts between them keeps it relatively aesthetically pleasing. This could be improved by perhaps changing the back ground to the company logo and using some softer pastel colours as the current colours are bright and could be quite harsh on the eyes for extended periods of time.

If I were to revisit the problem I would try and include the improvements mentioned above however, I don’t think I would change my actual approach to the program as the final product works well with no bugs which I have found.

After showing the system to Mr Harding (the end user) and discussing what he would like improved if this project were to be continued (although the system serves its purpose as a bookkeeping system which is what he was looking for) he suggested after noticing the advantage of the profit and loss system, that this could be extended further to include products, allowing them to add products to a stock list, then when a sale transaction is made you could select the item rather than inputting the cost it was bought at, the item could then be removed from the stock list and the stock account amount value would be decreased. This would allow for a good on the go system for a market trader which does more than just bookkeeping. Leading on from this he said that if the program was deployed onto a web hosting service like azure it would be useful as it could then be accessed on multiple devices from anywhere. I had already mentioned that with NOF this is very easy to do thanks to the way it is designed, this could be done easily however I felt it unnecessary, if this project was continued I would do this.

The final thing I would like to touch on for improvements is, if this was a fully implemented business system then the data it is managing is very important to the user, therefore I would suggest implementing a data back up system integrated into the program so that it can be done by the click of a button.