Isaac Basque-Rice

12basque-rice.i@thestudio-liverpool.net

EVaporate Documentation

Documentation for a WPF Application Based on the Concept of Video Game Distribution

[1. Introduction 2](#_Toc5301670)

[2. Designing Software Solutions Using the Object-Oriented Approach 3](#_Toc5301671)

[2.1. P - Exploring a User-Defined Problem and Identifying the Solution 3](#_Toc5301672)

[2.2. P - Designing an OOP solution using Diagramming Techniques 4](#_Toc5301673)

[2.3. M - Describing the Techniques Used 11](#_Toc5301674)

[2.4. D - Justification of Techniques 12](#_Toc5301675)

[3. Implementing Object-Oriented Applications 14](#_Toc5301676)

[3.2. P - Planning and Implementing the Design 14](#_Toc5301677)

[3.3. P - Using an Automatic Source Code Generation Tool 18](#_Toc5301678)

[3.4. P - Using Two Library Functions 21](#_Toc5301679)

[3.5. P - Implementing Database Connectivity to the Solution 27](#_Toc5301680)

[3.6. P - Checking if the Problem has been Solved and Reviewing My Own Approach 35](#_Toc5301681)

[3.7. M - Evaluation in Terms of Usability 37](#_Toc5301682)

[3.8. D - Justification of Two Ways in which the Program could be improved 38](#_Toc5301683)

[4. Understanding How to Maintain and Test Programs 41](#_Toc5301684)

[4.2. P - Production of a Plan to Test the Program 41](#_Toc5301685)

[4.3. P - Designing a Comprehensive Review Strategy for Updating and Maintenance 45](#_Toc5301686)

[4.4. M - Analysing the Results of Testing and Document Actions Taken 46](#_Toc5301687)

[4.5. M - Explanation of the Reasons for Reviewing and Maintaining the Program 53](#_Toc5301688)

[4.6. D - Evaluating the Effectiveness of the Test Plan 54](#_Toc5301689)

[5. Understanding How to Produce Documentation 55](#_Toc5301690)

[5.2. P - Creating User Documentation for the Program 55](#_Toc5301691)

[5.2. P - Creating Technical Documentation for the Program 64](#_Toc5301692)

[References 76](#_Toc5301693)

# Introduction

This document will describe the process and document the development of a desktop application for the distribution of Video Games in a mass market context. To achieve this, Use will be made of an Object-Oriented Programming methodology, an approach wherein the whole programming experience is based on the concept of objects, data structures and so on, as opposed to standard calculation and procedural actions and logic, this will allow me to manage the application much better.

This distribution platform will be known as ‘EVaporate’ and requires the following:

* Authentication at the point of login
* The data entry of sales into a live database
* The ability to search said database based on the following:
  + Genre, Publisher, Developer, Game title, Cost, Release date, Delivery type, Quantity/availability
* Branding
* Display of existing products for sale
* The display of existing and fulfilled customer sales order data.
* Existing sales records to be updated
* Archiving of complete orders

The application will be written in C# using the WPF[[1]](#footnote-1) format, the database will be based locally for this version in Microsoft SQL Server and written in SQL, a database management programming language. Additionally, Use will be made of two aspects of at least one .net programming library, code generation will be achieved using Entity Framework, which should allow for automated class development and recording of data from an instantiated object. Finally, Documentation must be made of an extensive testing process, which is shown towards the end of this document[[2]](#footnote-2).

# Designing Software Solutions Using the Object-Oriented Approach

## P - Exploring a User-Defined Problem and Identifying the Solution

My task, as previously outlined, is to design a solution for a user defined problem, this problem is the display and distribution of video games within a computer application framework. This app, as previously mentioned, will be built using WPF, an implementation of the C# programming language, which makes use of the object-oriented approach.

With regards to the user-defined problem itself, liberal use will be made of a core principle of OOP, which is “objects”, these objects will be diagrammed out in the section below however the two central objects that have been decided on are “User” and “Game”, which will have different attributes depending on what they require. Once again this will be further explained in the diagramming section.

I will be following the software development life cycle as closely as possible, this is a series of steps that a software developer or development team must follow to result in a successful project. The way they follow it is up to them, for example a waterfall methodology would see a software developer go through each step iteratively and not return to any previous step.

The methodology shown in this document and in the process of development will be incremental, repeating three steps of the development process several times over the course of development, specifically design, development, and testing (in that order), this will allow me to improve my application over time as opposed to having to rely on pre-set design specifics

The SDLC is as follows:

* Analysis

This stage has already been carried out in the current context, it has been determined by a third-party that there is a market requirement for a video games distribution platform and as such it must be delivered.

* Planning

The requirements as set out in the specification must be analysed thoroughly, in the current context, upon reflecting on the specification the current document was created and headed appropriately to correspond with the different requirements of the application.

* Design

The design section of the SDLC hinges primarily on the production of front-end designs, such as mock-ups, and back end design techniques, such as the use of Unified Modelling Language (UML) and CRC cards, which allow us to look at the attributes of classes that the developer intends on creating and how each class interacts with one another.

In the current context, the following sections (2.2, 2.3, and 2.4) will be dedicated to building the previously mentioned diagramming techniques, and describing and justifying the said techniques.

* Development

This section is fairly self-explanatory, this will be the time that the developer will actually produce the application. For the most part documentation of this will take place in section 3 (subsections 3.1-7) with some possibly bleeding over into section 4, specifically subsection 4.3, which asks to analyse the effectiveness of the test plan on the developed (or currently in development) solution.

* Testing, Deployment, and Maintenance

Finally, this is the section regarding the testing and maintenance portion of the SDLC, this will be fleshed out more in section 4 however in the current context a plan will be produced to test the application for various bugs and errors and carrying it out as well as is possible. This will be the case for deployment and maintenance, also.

## P - Designing an OOP solution using Diagramming Techniques

### Class Generation using Noun-Verb method

A problem one may be very quick to stumble into is that of actually creating classes and subclasses, it can be difficult to think like a machine in this respect. As such use will be made of a very simple tool, this tool is to walk through each of the steps in words and highlight the nouns and verbs separately, the nouns being classes and verbs being methods

Classes

Methods

Login: The user can login or register

Purchasing games: the user can view and buy games

Viewing purchased games: users can view games already purchased in the store within their UserSpecificLibrary

Adding funds: the user can add and check funds using their CardDetails

From this we can gather that the classes and methods we will have are as follows:

* User
  + Login
  + Register
  + View Games
  + Buy Games
  + Add funds
  + Check funds
* Games
  + View Games
  + Buy Games
* Store
  + View games
  + Buy games
* UserSpecificLibrary
  + View purchased games
* CardDetails
  + add funds
  + check funds

As you may have noticed there is a significant amount of overlap in these classes, this means that both the classes with 100% overlap such as Games and Store can be rolled into one if they have the same items stored within their databases (which they do). If there is overlap but it is not 100% then in the CRC cards (see below) they will be considered linked classes, as denoted in the side of the card.

### UML

The following is a UML class diagram of the five core classes of the application as it is currently intended, as well as the relations between them.

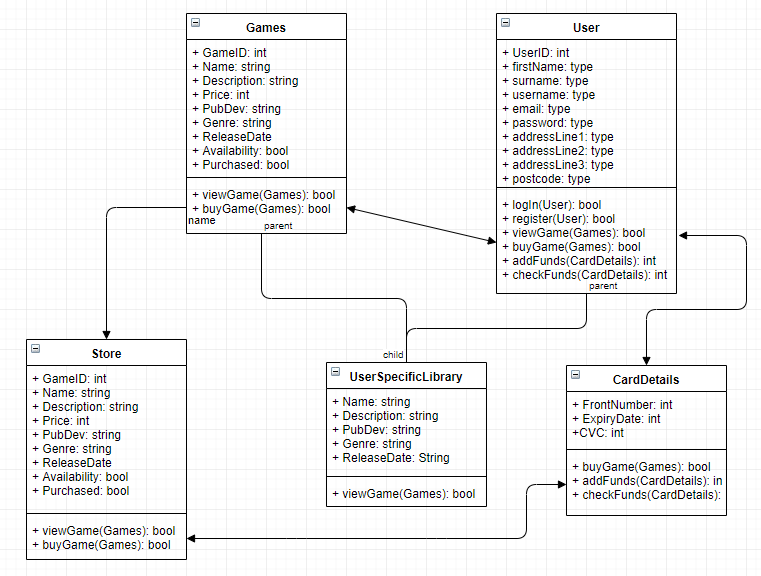


Figure 1, A UML class diagram of the five core classes of the app

### CRC cards

The following are the main five examples of CRC cards that will be used in the development process, these cards have been constructed with, for the most part, the use of the Noun-Verb technique used above.

|  |  |
| --- | --- |
| **Games** | |
| **Responsibilities** | **Collaborators** |
| * Stores:   + Name[[3]](#footnote-3)   + Description   + Price -int   + Publisher/Developer   + Genre   + Release date   + Availability -bool   + Purchased -bool * Methods:   + viewGame   + buyGame | * User   + viewGame   + buyGame * UserSpecificLibrary   + viewGame   + Purchased bool * Store   + All Storage   + All Methods * CardDetails   + buyGame |

Figure 2, CRC card for the Game superclass

|  |  |
| --- | --- |
| **User** | |
| **Responsibilities** | **Collaborators** |
| * Stores   + Username   + Email   + Password   + FirstName   + Surname   + Address line 1   + Address line 2   + Address line 3   + Postcode   + Funds -int * Methods:   + Login   + Register   + viewGame   + buyGame   + addFunds   + checkFunds | * Games   + viewGame   + buyGame * UserSpecificLibrary   + viewGame * Store   + viewGame   + buyGame * CardDetails   + buyGame |

Figure 3, CRC card for the User superclass

|  |  |
| --- | --- |
| **UserSpecificLibrary** | |
| **Responsibilities** | **Collaborators** |
| * Stores   + Name   + Description   + Publisher/Developer   + Release Date   + Genre * Methods:   + viewGame     - If purchased (bool) = true then it is viewable | * Games   + viewGame   + Purchased bool * User   + viewGame * Store   + viewGame   + Purchased bool   + description |

Figure 4, CRC card for the UserSpecificLibrary subclass

|  |  |
| --- | --- |
| **Store** | |
| **Responsibilities** | **Collaborators** |
| * Stores:   + Name   + Description   + Price -int   + Publisher/Developer   + Genre   + Release date   + Availability -bool   + Purchased -bool * Methods:   + viewGame   + buyGame | * Games   + All Storage   + All Methods * User   + viewGame   + buyGame * UserSpecificLibrary   + Name   + Description   + Publisher/Developer   + Release Date   + Genre * CardDetails   + buyGame |

Figure 5, CRC card for the Store subclass

|  |  |
| --- | --- |
| **CardDetails** | |
| **Responsibilities** | **Collaborators** |
| * Stores:   + Long number on front   + Expiry date   + CVC code * Methods:   + buyGame   + addFunds   + checkFunds | * Games   + buyGame * User   + buyGame   + addFunds   + checkFunds * Store   + buyGame |

Figure 6, CRC card for the CardDetails subclass

## M - Describing the Techniques Used

In this section of the brief the techniques used were threefold, firstly being a fairly unorthodox method branded the noun verb method, secondly being a UML class diagram, and third being a series of CRC cards.

Firstly, and as described above, we have the noun-verb technique, which sees an individual writing sentences that describe the different sections of the program and generating class names from the nouns in the sentence and methods from the verbs within said sentence, an example of this would be the following:

A car: the car must be able to drive forwards, reverse, break, and honk its horn

From this relatively simple sentence we have derived two classes, car and horn, and four methods, driveForwards, reverse, break, and honk. Additionally, we can determine from this sentence that horn is a subclass of car, or car is a superclass for horn if you wish, due to the fact that the horn is intrinsically linked to the car and could not normally act outside of that context.

Next, we have CRC cards, there are 5 Classes, Games, User, UserSpecificLibrary, Store, and CardDetails respectively. The main body of the card is separated into two distinct sections: what the card stores (e.g. what it relates to in the RDBMS), and the methods it makes use of, as described in the verb section of the noun-verb technique. The smaller section describes the other classes which each of the class cards relate to, for example Games would relate to all of the other cards as they stand through its methods viewGame and/or buyGame. These relations are best shown through the UML class diagrams above.

With regards to the UML class diagram, Games and User are placed at the same level of importance, they occupy the ‘superclass’ role, from both of these we inherit UserSpecificLibrary, which displays to User X ONLY the games that they have purchased, and only displays its name, description, genre, publisher/developer, and release date.

CardDetails is a separate class however it relates directly to both user and store for obvious reasons, it is kept as a separate class purely in order to separate the user from their card details for the sake of possible security reasons, however there is a possibility that this will be changed in the near future.

Finally, we have store, which relates to Games directly as it is merely a means for displaying the list of games pulled from the Games class, however it is designed to be a middle man for purchasing said games. This class is in a similar position to CardDetails wherein its position as a separate class at the moment is undecided, however for the sakes of the early design phase the developer will be operating under this diagram.

## D - Justification of Techniques

All of the techniques used herein are specialised for the object-oriented model, they are organised around the concept of performing any number of actions (methods) on a given data structure, what the object stores, models such as the UML class diagram show how these data structures and methods act on one another.

The first technique provides a simple and easy to understand method of creating classes and methods, due to its use of plain and simple language that the programmer writes themselves, the ease of understanding for anyone from a programming novice to an expert is far beyond any other technique used, with special consideration to merely coming up with them from scratch. As mentioned in the description for the technique, the creation of classes is not an especially easy thing to do for a human, this method provides an unparalleled level of understanding and ease of creation.

Next, we consider UML. UML, or Unified Modelling Language, is a modelling paradigm or language that has a standardised set of iconographies for visualising and constructing the components of large software systems[[4]](#footnote-4). UML offers a number of diagrams that serve different purposes depending on what one wishes to depict. For example, the UML diagram that has been made use of in this document is what is known as a class diagram, which is an illustration of the relations and contents of the different classes. This has been used in order to give the developer a clear and simple idea of how the application should function, and also be able to convey this same message to any non-development-oriented individuals in an organisation, such as business managers etc.

CRC cards serve as a method of graphically displaying classes and their relationships with other classes, the examples above demonstrate this nicely. Within the cards above there are two distinct columns, Responsibilities and Collaborators (when one understands CRC stands for Class Responsibility Collaborators this makes more sense), which are responsible for what is stored, the variables, and the methods, or how the data is manipulated. The justification for the use of CRC cards would, firstly, be their ease of understanding, which allows for those who are not programming-oriented to have a basic grasp of the inner workings of the program/product and as such be able to contribute and brainstorm ideas within the context of a meeting where they are used.

Their primary status as a physical medium additionally mean they aren’t reliant on any necessary hardware (other than paper, obviously), and as such can be ‘cheap, portable, readily available, and familiar”[[5]](#footnote-5).

# Implementing Object-Oriented Applications

## P - Planning and Implementing the Design

The front-facing design of my application requires, obviously, several design choices to be made. The first choice is with regards to the physical size of the app within the window. The size of the app, or at least the size that the app will be developed in, is irrelevant due to the fact that WPF scales applications when the app is resized. As such the design and development of the app will take place in the standard 450\*800px resolution.

I have decided, in the interest of good design, to go for a mainly dark-themed approach with regards to colour scheme, the following is a list of hex values, the representative colour, and the use that the colour will have in my application:

|  |  |  |
| --- | --- | --- |
| Colour | Hex Value | Use |
|  | #1A1A1D | Primary grey, Background, some text on navigation (such as in logo and headings) |
|  | #4E4E50 | Secondary grey, Text on the background that doesn’t need to be emphasised, “normal” text |
|  | #C3073F | Primary red colour, used for navigation bars and the like |
|  | #950740 | Secondary red, for emphasis on primary grey, used for headings within the context of content |

I will be making use of GIMP2[[6]](#footnote-6) as my primary frontend design tool due to its availability to me as a free and open source product.

What follows is the front facing design template based off of these colours, this will be the “master page” if you will and will cover roughly everything that is required. The fonts used are a standard sans-serif 25px for the majority of the contents with the logo’s font being “Blanka”, a font by Emmeran Richard[[7]](#footnote-7), at 20px. The part that says [title] represents a dynamic section of the app that is intended to be implemented, depending on the page a user is on the [title] will change to reflect that. The search function is denoted with the magnifying glass in the top right-hand corner. Finally, the large black rectangle in the centre represents where the contents will be placed.

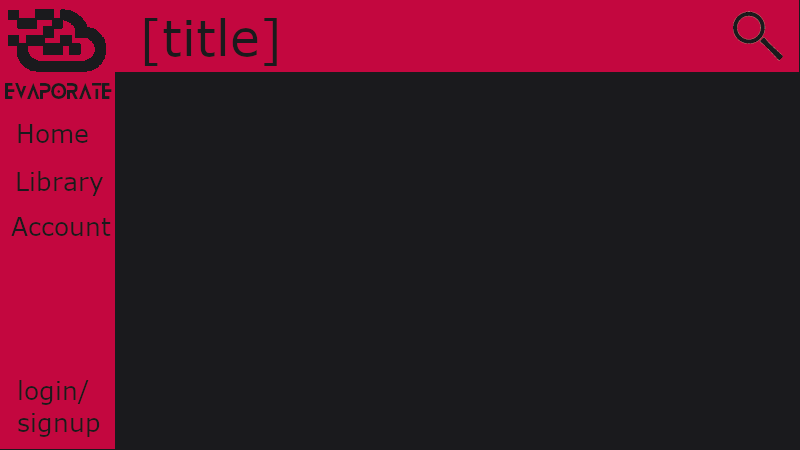


Figure 7, the template for front facing design

Next, we have the game display/purchase screen, the white square with “Game” in it represents the image for the game in a 250\*100px image ratio. Genre, Release date, and developer and publisher information is displayed next to the image in 10px, below the image is the description, and below that is the price and ‘buy’ button, both in 20px for extra emphasis.



Figure 8, the rough template for a game's display screen

Here we see the solution explorer for the application, this is by no means the final version however it gives one a good idea of what will be included in the app.

The assets folder, firstly, is where I will store all of my assets such as a logo, the images for the games, and so on. Underneath we have classes, which contains the five classes spoken of previously (CardDetails, Games, Store, User, and UserSpecificLibrary), below this is the models folder, which is presently empty, however during development will be used to store classes that are used to model data, for example, the Games class would be stored in here and when a game is retrieved from the database a new Game object would be created from the model in this folder.

Finally, we come to the pages and windows folders which serve similar functions but have been kept distinct because of their separate properties. A window, firstly, serves as an entirely separate dialog box, if you were to take Steam, an already popular games distribution platform, as an example, the ‘friends’ tab would serve as an example of a window as within this app a separate window opens up when you click this tab. Pages, on the other hand, are different in that they occupy the same space as other pages within a window and one must navigate between them. To continue the link to Steam, within this app examples of pages would be Store, Library, and Community.

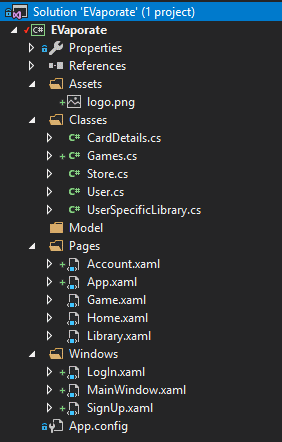


Figure 9, the solution explorer

The following is the appearance of the application with near no content but with functionality to open two other windows, LogIn and SignUp

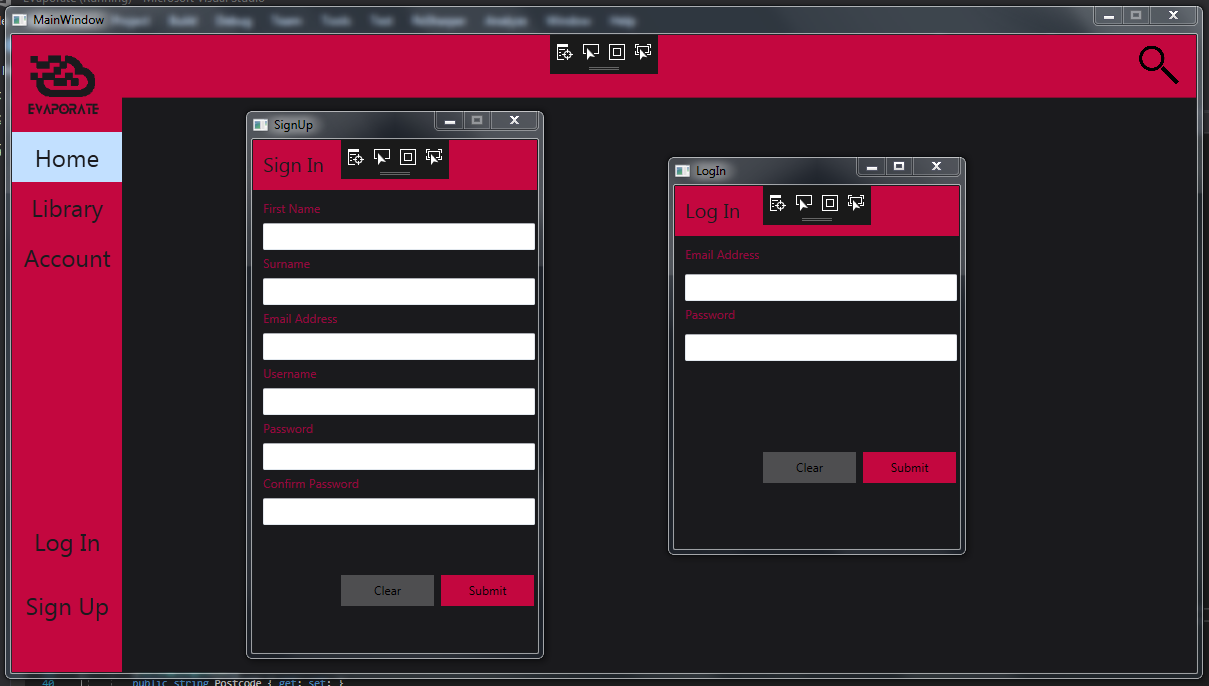


Figure 10, the main window, log in and sign up windows

## P - Using an Automatic Source Code Generation Tool

Within this solution I made use of an ORM, or Object Relational Mapping tool, by the name of Entity Framework. EF serves as a framework for ADO.NET (ActiveX Data Object .net, a branch of the .net framework), and fulfils my code generation tool requirements.

As the screenshots below show, I made use of EF in its capacity as a database-first tool, meaning I built a database first in MS SQL Server and then made use of EF to generate classes in the ‘Model’ folder (shown above) from objects in the database, so I could tread DB objects as objects in code. This serves to make the development process fairly simpler as I can manipulate the data with significantly more ease.

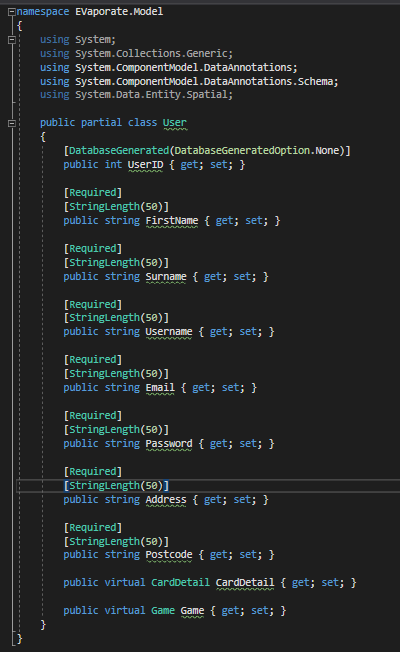


Figure 11, the code generated for the 'User' database using Entity Framework

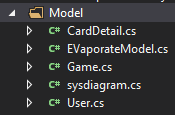


Figure 12, the 'Model' section of my solution explorer after implementing EF

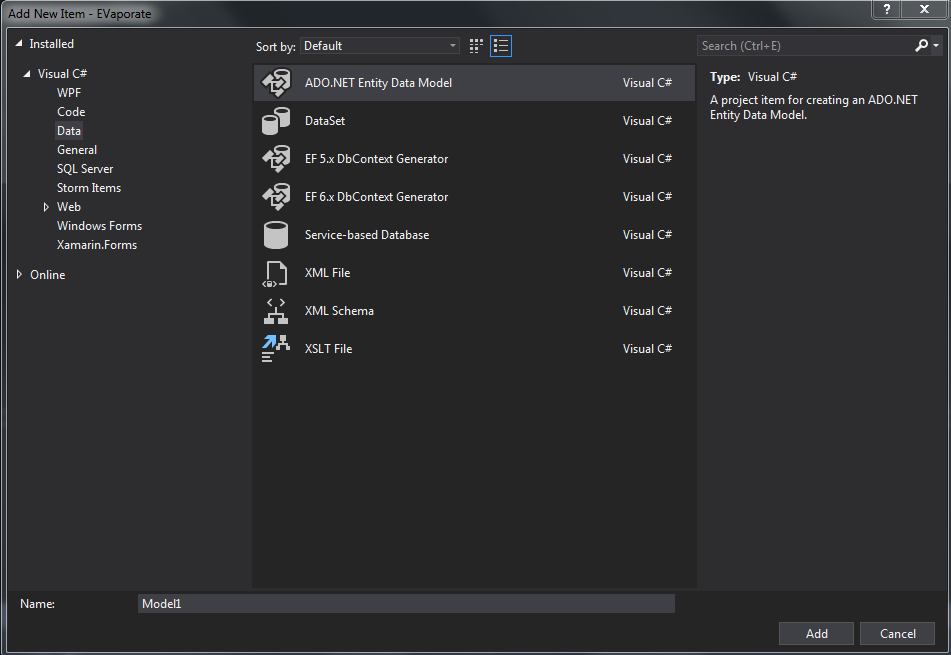


Figure 13, the 'add new item' page

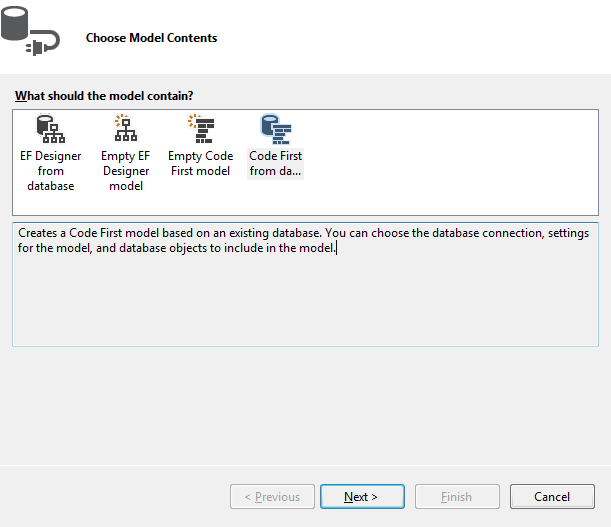


Figure 14, the option I chose in the Entity Data Model Wizard, it produces code from a pre-existing DB

## P - Using Two Library Functions

### Entity Framework

As mentioned previously, an example of one library I have made use of in my solution is EF, or Entity Framework, which has allowed for the manipulation of data in a much simpler fashion. As an ORM, it deals with the creation of database connections and command execution, abstracting this away so the developer can deal with logic in other parts of the solution.

The way EF works is by the formation of different classes within a solution that represents databases. For example, the User database will have a UserID, a First name, Surname, Username, Email address, a password, an address, and the corresponding postcode. The way EF represents this is by creating a C# class in a designated location in the solution and generating code that represents this class as a sort of middleman between the programmer/input and the database.

The code for the User class is below, note how the specifications I have laid for the different database fields in SQL Server carry over in the class in the form of [Required] And [StringLength(50)] functions.

namespace EVaporate

{

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

public partial class User

{

[DatabaseGenerated(DatabaseGeneratedOption.None)]

public int UserID { get; set; }

[Required]

[StringLength(50)]

public string FirstName { get; set; }

[Required]

[StringLength(50)]

public string Surname { get; set; }

[Required]

[StringLength(50)]

public string Username { get; set; }

[Required]

[StringLength(50)]

public string Email { get; set; }

[Required]

[StringLength(50)]

public string Password { get; set; }

[Required]

[StringLength(50)]

public string Address { get; set; }

[Required]

[StringLength(50)]

public string Postcode { get; set; }

public virtual CardDetail CardDetail { get; set; }

}

}

Below is an example of valid code using the EF framework, this code creates a new user in the database based off of the above class.

//instantiates the 'User' object in Model and sets the value of each variable to be equal to input in its respective text box

User user = new User

{

FirstName = tb\_FirstName.Text,

Surname = tb\_Surname.Text,

Username = tb\_Username.Text,

Email = tb\_Email.Text,

Password = tb\_Password.Password,

Address = tb\_Address.Text,

Postcode = tb\_PostCode.Text,

};

### MaterialDesignInXAML

As one can imagine, MaterialDesignInXAML (By Sebastian Dymel, available at <http://materialdesigninxaml.net/home> and github repo at <https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit>) is fairly simple, it adds a Material Design User Interface to the front end of the application. In practice and in the larger scope of the app this changes fairly little, however from an aesthetic perspective this is a noticeable improvement, take the following screenshots as examples, note the distinct lack of white space where the text boxes are, this decreases eye strain on the user:

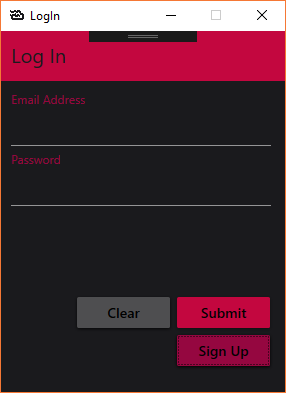


Figure 15, the LogIn window using material design

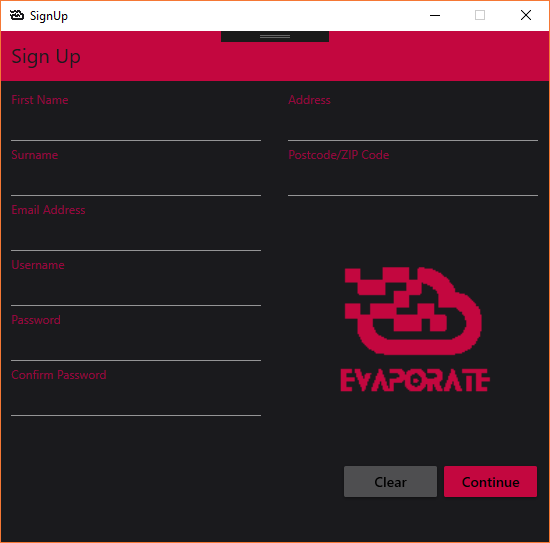


Figure 16, the SignUp page using material design

Additionally, from the screenshot below, note the shaded circular shape radiating from the ‘Home’ button, this is a distinct feature of Material that lets a user know a button has been pressed and serves as a benefit to aesthetics:

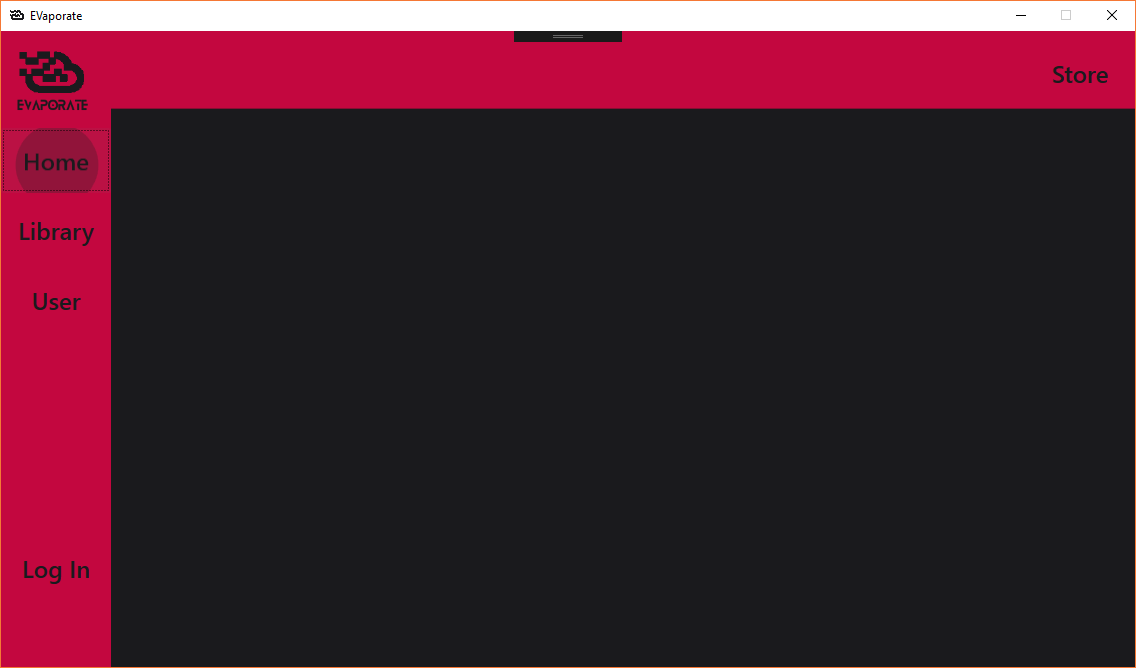


Figure 17, the main screen with the home button pressed.

Note must of course be given to the extremely helpful MaterialDesignDemo app which allows a developer to reference and copy snippets of code for stylings, which has come in very handy for button stylings as there was an issue with buttons all but disappearing. Below is the demo application and below this is an example of code for a correctly styled button in the Material format.

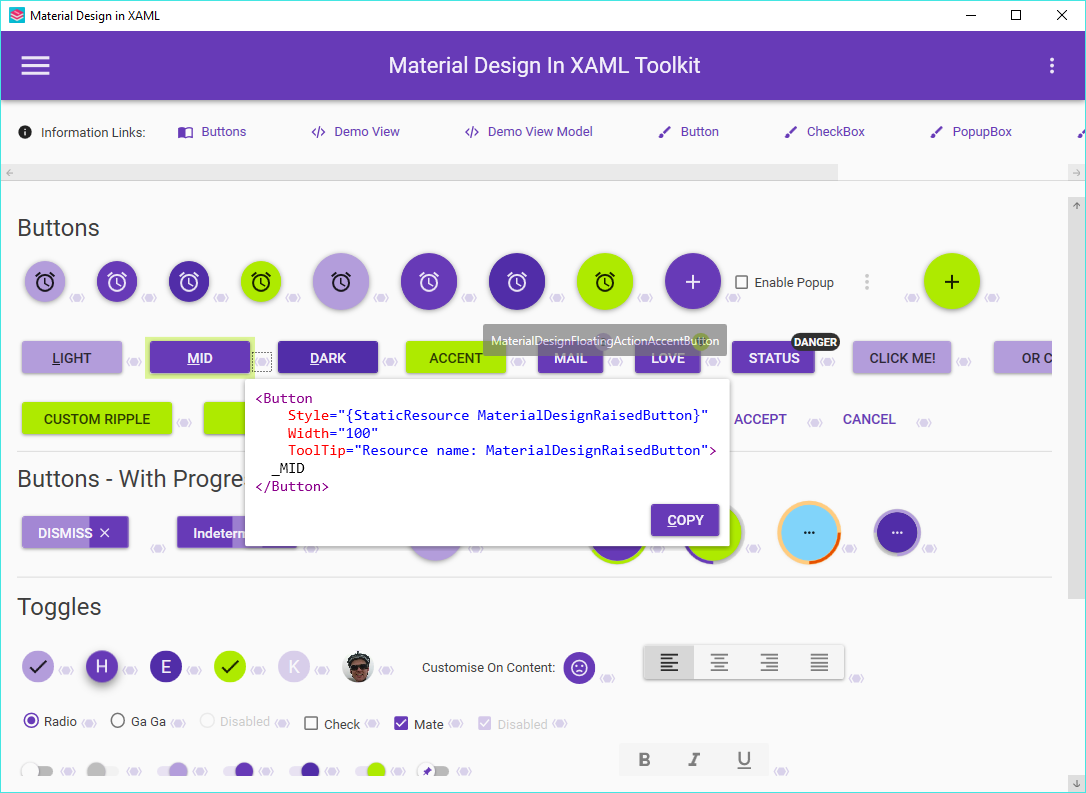


Figure 18, the Buttons page on the MaterialDesignInXAML Toolkit demo app, which is very helpful indeed

<Button

Style="{StaticResource MaterialDesignRaisedButton}"

Width="100"

ToolTip="Resource name: MaterialDesignRaisedButton">

\_MID

</Button>

## P - Implementing Database Connectivity to the Solution

At the present moment the SQL database I have created is based in Microsoft SQL Server (Through the MSSQLS Management Studio serving as an RDBMS) and contains three tables of note, these tables are[[8]](#footnote-8):

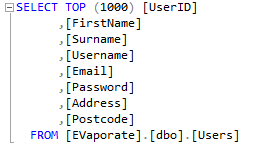


Figure 19, the Users table

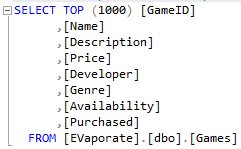


Figure 20, the Games table

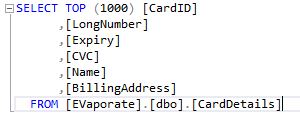


Figure 21, the CardDetails table

From these tables I have generated classes, as seen above, that serves as a method of manipulating data as if it were a code object as opposed to sending information directly from the input to the server, making the process much simpler.

Below is code from the SignUp.xml.cs page, this code is prompted to run when the button ‘submit’ is clicked on the main window and is set so that whatever is inputted into the form is converted to ‘readable’ text in the context of a database and moved to the Model.Users.cs class, and committed into the database

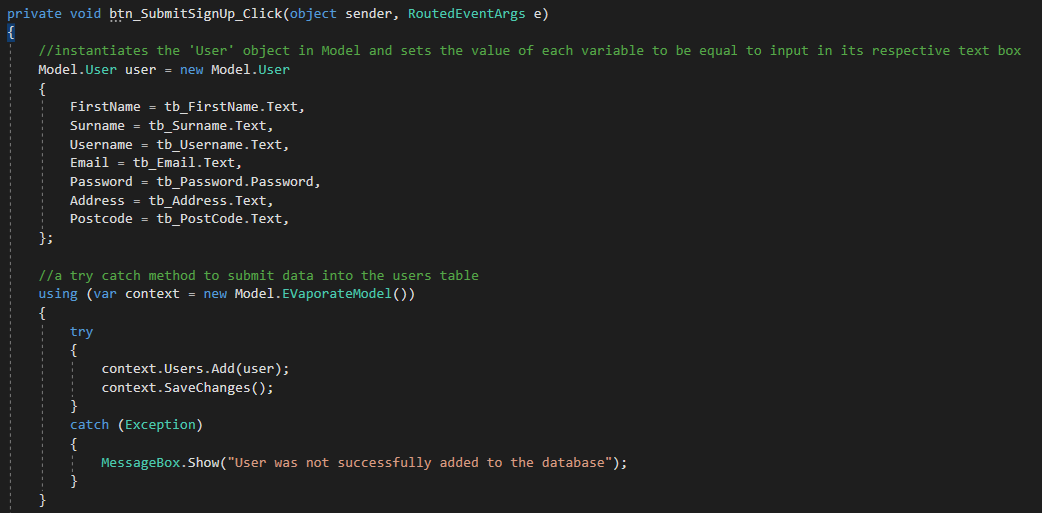


Figure 22, Code from the SignUp.xml.cs page

What follows is information about the connection itself, beginning with the data connections menu in the Server Explorer of VS, with ‘Tables’ expanded, then moving to information on the connection string itself sourced from the App.config file, then finally the properties window of the connection itself.

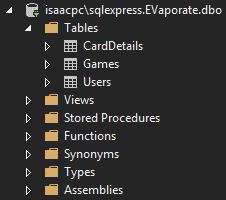


Figure 23, information on the connection from the server explorer

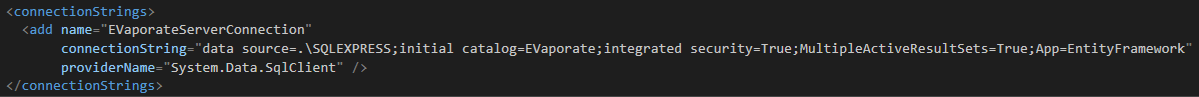


Figure 24, XML for the connection string, note how 'name=”xyz”' allows me to abstract the full connection string into something more manageable if necessary

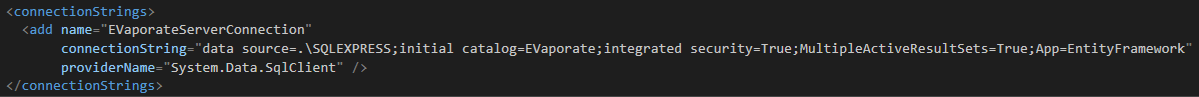


Figure 25, the second half of the above image

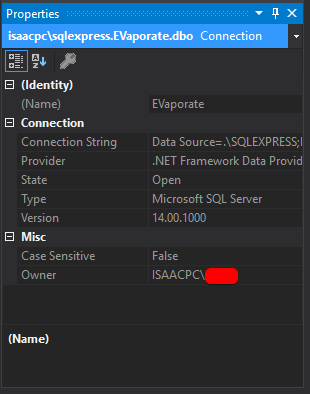


Figure 26, the properties window for the connection

The implementation within my solution is a particularly simple one, and the main form of implementation for databases in most solutions across the board. This is a series of functions that allows for logging in and signing u to the application. Additionally, I have functions in place to check and validate cards, including the Long number, CVC, and expiry date numbers.

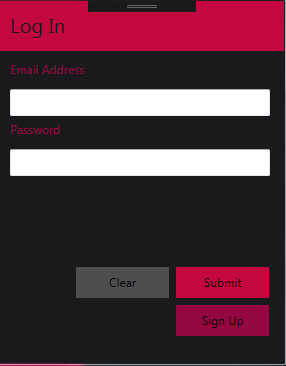


Figure 27, the Log In page

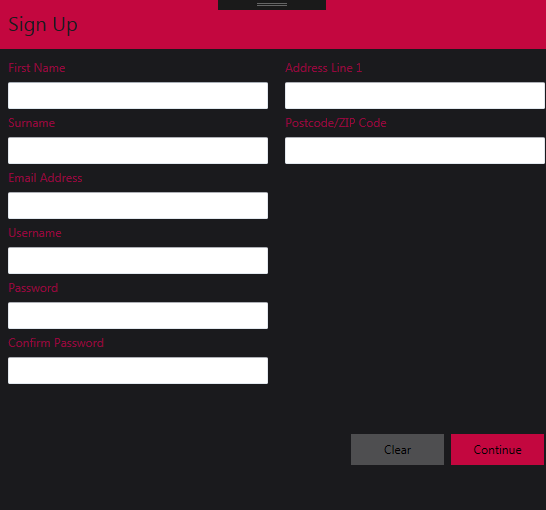


Figure 28, The Sign-Up page

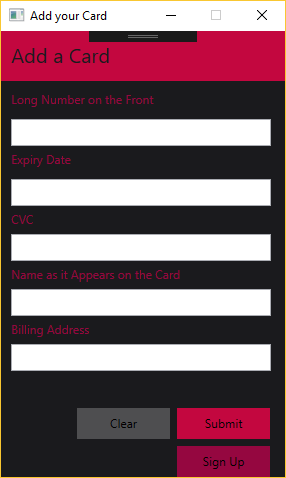


Figure 29, Card Input window

As one can clearly see from the below screenshot of the database, input from the form window to the database succeeds flawlessly

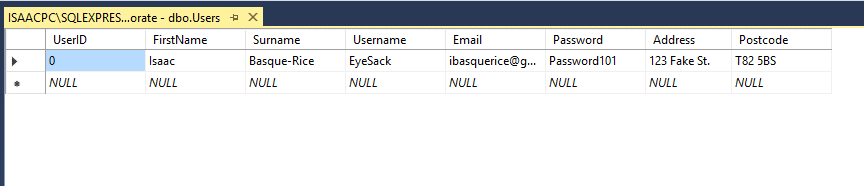


Figure 30, the SQL database with the first user's details in

## P - Checking if the Problem has been Solved and Reviewing My Own Approach

Before continuing with my description of the problem at hand and my solution to it, it must first be mentioned that the existing EVaporate system has become slow and contains various bugs, which results in slower display of its content, which is leading to the loss of customers, as such, the company has contracted me to develop a replacement system.

The solution, as it has been laid out in the brief document, has had the following requirements:

* User must authenticate at the point of login
* Data entry of new sales data into a live sales database, which requires the following:
  + Genre
  + Company/developer
  + Game title
  + Image
  + Cost
  + Release date
  + Availability
* Branding
* The display of existing products for sale
* The display of existing and fulfilled customer sales order data.
* Allow existing sales records to be updated.
* Archiving of completed orders.

As such, what follows is a point by point review of the software I have developed to fulfil this task, and a review of my own work. The problem as laid out above has been resolved due to the creation of a new system which is significantly faster in its running and therefore customer retention is expected to be higher and that the system now is relatively bug-free providing a better user experience.

Firstly, with regards to authentication at the point of login, the method that has been used in this instance during the development process sees the user opening the app and then electing to log in or sign up of their own volition, this process may not be necessarily what was asked of on the spec, however this process was put in place for the sake of ease of development and ease of use, as the user may wish to look at products before logging in or signing up, like in the case of Amazon who allows their users to do this.

Next, we must consider the Data storage and entry methods of the application, as it stands there is no way of entering the information for a game into the database through the app itself, it must be done through raw data input into the SQL Database.

Otherwise, however, the majority of the required data fields are present in the database table, as can be seen below:

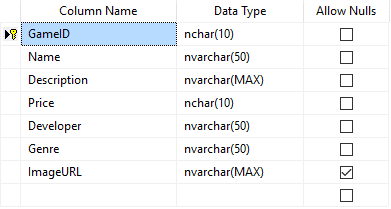


Figure 31, the game database table design

In the case of the other data fields, namely release date and availability, these are currently not present in the application however they will almost certainly be added at a later date, with availability as a bool expression and release date to only be displayed in the event that a game has not yet been released on the platform.

With regards to branding, I have created and stuck to a set colour scheme of reds/pinks/purples and dark grey, and have also developed a logo that you can see below:



Figure 32, the EVaporate Logo

## M - Evaluation in Terms of Usability

This section concerns, as the title suggests, to what extent is the program actually usable in the normal end user context? Research I have conducted into Usability for this project (sources of which can be found here[[9]](#footnote-9), here[[10]](#footnote-10), and here[[11]](#footnote-11)) has found that Usability is not an especially easy concept to measure from the perspective of the developer, as they will have intimate insider knowledge with the program and will have built it so they themselves can use it, in addition to this, an application is almost never intrinsically usable to everyone, as all users may require different things from their experience, and it’s near impossible to cater to every possible user. However, with this in mind the following will be an attempt to evaluate my app from the perspective of an unknowing end user.

The log in and sign up process as documented extensively elsewhere is fairly simple and should hold no challenge to the average user, where one my stumble up initially however is the lack of dynamic content on the MainWindow page with regards to an individual user, in short, there is no “Hello [username]” function within the app. This may serve to hinder some users who may assume they are not logged in when they initially see no visible changes in the application.

From an aesthetic standpoint within text boxes most cursors seem to remain black, due to the lack of visibility of the cursor a user may not be able to see where they are about to type, which could cause some significant issues.

On the positive side however for the most part the content is extremely simple to navigate through, if a user wants to go to the store they click ‘store’, if they want to go to their user profile they click ‘user’, ‘home’ for the home page and so on. An issue with the current application however is a noticeable lack of content which may affect user retention, this should most likely be fixed if development is continued on the project.

Central to the philosophy of most if not all successful software companies are placing the usability and user experience of their product at the very top of their priorities. It is tempting to think of “adding” usability as if it were some code or an ingredient, this is not the case, usability is more a philosophy, something one must always keep in mind in the development process. To this end, focussing on the needs of the users, particularly the edge-cases, is vital for usability’s sake. Examples of how this can be done is through accessibility considerations for those who are disabled (e.g. allowing screen readers, allowing navigation through multiple means, such as keyboard only, and so on)

To conclude, the usability factor within the EVaporate app comes from its simplicity and well-established design philosophy, further research into user experience must be conducted, ideally through black box testing, in order for a true sense of the end user’s experience to be established and worked upon. In addition to this, usability and accessibility studies must be conducted around the application in order to gather what considerations the developers will need to have going forward in the development process.

## D - Justification of Two Ways in which the Program could be improved

### Addition of method of adding games in-app

As of the current version of the application there is no method of adding a video game into the app without making the addition in the SQL Database behind, this presents a challenge to prospective developers and independents insofar as they must come to me, as the developer, in order to add their game to the marketplace. Of course, the benefits of this do exist, namely the fact that video game curation would, in theory, be much more stringent that just allowing any old person to upload any old (possibly buggy) program into the store. However, the inability to do this could, in theory, badly affect possible developers in that they would have to wait many weeks to see if the game they possibly spent months’ worth of time and money on makes it into the store, without the guarantee that it would be admitted.

Ultimately, however, having a usable graphical user interface for uploading games to the store is convenient and useful for all involved, the way I would implement this would be as follows:

1. Allow developers to create a “developer account”
2. Encourage them to send game files to a curator through the application
3. If the game has been accepted, generate a submission key, a string of 3 sets of 9 random characters separated by dashes (e.g. HD7dgR7G5-qbey7Hoq-BbwyYr15F)
4. Allow them to publish the game files with the included key as validation

### Addition of a recommended game algorithm

A recommended game algorithm, or a recommendation engine as it will be referred to from now on, will be extremely beneficial to the application, as it will predict what games a user may like as opposed to just making a user search a specific term, this will almost certainly result in an increased rate of purchases on the platform and serve, most likely, to increased rate of enjoyment as users will be able to play games they’re more likely to actually enjoy as opposed to having to take a stab in the dark, if you will.

There are two methods by which one can implement a recommendation engine, the first being based off what other users enjoy, this is not ideal as it will offer already popular games locking independents out of the market and only recommending games that users are already likely to know about. The second, more workable method is by implementing tags and recommending games to users based off popular tags on that account.[[12]](#footnote-12) This is because users are, overall, more likely to enjoy specific genres (MOBAS, RPGs, FPSs, Sci-Fi, Fantasy, and so on) than what others enjoy, necessarily. The framework for this system is already in place but admittedly requires updating, within the SQL database I have a ‘Tags’ field, however I feel as if it would be easier to have several tags fields with one tag in each, this restricts tags to a set number and as such allows for an easier development process.

# Understanding How to Maintain and Test Programs

## P - Production of a Plan to Test the Program

### Introduction

This test plan encapsulates the EVaporate video games distribution platform and serves to document and track the information necessary to fix issues with the application, in this particular instance. Not counting this section this plan will be put forward with nine distinct sections, these are as follows.

Scope, a description of the extent of the area of testing, essentially a description of the Evaporate platform. Tests, which, as the name implies, is a description of most if not all necessary tests as they occur during development. Tools, which encapsulates environment requirements and other tools I have and will use to complete development. What is and is not to be tested, as well as a strategy for updating the product, a review of the final product as it relates to the test plan, an explanation of the reasons that this has been undertaken, and finally an evaluation of how effective the test plan was/is.

### Scope

The scope of the testing within this project falls under three distinct categories.

Firstly, the front-end code, this is measured in terms of basic end-user usability and graphical integrity, e.g. if it looks good or not. Next, we have the code behind, this will most likely give the most trouble and be the section on which most of the work will be done. This section concerns the logic of the application and encapsulates the logging in/signing up process, logic for the buttons, and other such things. Finally, we will have the back-end, the RDBMS that is being used, how well it functions, etc. This section very rarely gives issue due to its simplistic nature so chances are features herein will not need much in the way of testing.

Due to the fact that the development of this project is being undertaken by a single full stack developer not much, if anything at all, is out of scope, that is, beyond anything that isn’t actually included in the plan. If a new feature is added beyond the requirements for the product this will be out of scope, however at this point no such thing has arisen.

### Tests

All tests that will be carried out within the context of the app will be making use of the debugging feature in Visual Studio, and indeed all IDEs This feature compiles and runs the program in question exactly as the developer has instructed in order for them to discover and fix any bugs or issues that they have inadvertently created in the development process.

For issues in front end code (XML) this will be all that is necessary, however for the backend code (C#) I will need to make use of breakpoints, points in code where the program pauses in order for the developer to be able to understand what is occurring in that program at that specific point, this will be excellent for issues with things such as SQL statements and other defective backend code.

A strategy I have devised is as follows:

Step 1. Begin debugging, if it works, move to step 2, if not, begin searching for a solution

Step 2. Create a new account, if account creation works move to step 4, if not, look at the SQL code for the sign-up form

Step 3. Add a new card, this process will happen anyway regardless if step 2 functions or not, if it works move to step 4, if not, check the SQL

Step 4. Browse the app, looking for any anomalies, if any are found, look in the relevant code (back or frontend), if none are found, move to step 5

Step 5. Purchase a game, if this functions as normal, move to step 6, if not, check code

Step 6. Begin development on the next section, move to step 7, if this does not apply, move to step 8

Step 7. Focus on debugging this new section until it is seemingly ready to go, then move back to step 1, if all planned features are finished, move to step 8

Step 8. If any bugs were found in the previous steps move back to step 1, if none were found and all features are completed, move to step 9

Step 9. Declare the work finished and hand in.

This process will help me develop and test as efficiently as humanly possible whilst also being as thorough as possible with regards to problematic features of the app.

This step through method is, however, not strictly the only method of testing, if, in the course of my development, I find a bug in a specific part of the application, the testing will switch from a more general-purpose testing plan to a more specific one. Simply put, I will focus on what part of the app needs my attention most.

Below here is a testing grid with the columns “Test No. (number), Description, and Expected Outcome. This grid is a basic outline of some of the tests I will be carrying out during the development process, later on I will be filling out a similar grid to this with the addition of two new columns, “Actual Outcomes” and “Actions Taken”.

|  |  |  |
| --- | --- | --- |
| **Test No.** | **Description** | **Expected Outcome** |
| 1 | Initial launch | Application launches successfully, no build errors, able to use it in whatever state it’s currently in |
| 2 | Basic control addition and functionality (opening new pages and so on) | On clicking a button for a new page in MainWindow to open OR for a clear button to clear all fields in a form |
| 3 | Opening a new window using a control | On clicking specific buttons for windows to open (for example a login/signup button) |
| 4 | Ensuring styling is present and correct in all windows/pages (material design may affect this) | Perhaps a few misplaced controls or strange changes, this is due to the fact that I did not initially design in material and the changes to material can have unforeseen consequences |
| 5 | Making sure game windows display properly | On clicking a game button in the store page, a window for that specific game opens. |
| 6 | Making sure connectivity to the database works through the data model | Database connection to be successful |
| 7 | Adding a user to the database | User addition to be successful and progression onto card addition being smooth |
| 8 | Adding a card to the database | Card addition to be successful and the window to close without error message or exception |
| 9 | Logging in successfully | Using an already known account to sign in to the application without exception or error |
| 10 | Purchasing a game | The card confirmation screen to come up and, upon validation, the game to be added to the user’s account |

### Tools

In order to effectively test and develop the application I have needed to make use of a number of crucial development tools. Most notable are the IDE Visual Studio, the RDBMS MS SQL Server, the languages the program is written in, XAML and C#, as well as the other miscellaneous tools such as hardware and so on.

Beginning at the hardware level, I am running development on two separate machines, my Studio-based computer, a Fujitsu ESPRIMO E410 E85+ loaded with Windows 7 64-bit Enterprise version, an Intel i5-3330 CPU and 8GB of RAM, and my home PC, a self-built machine running Windows 10 64-bit, an i5-6500 CPU and similarly 8GB of RAM.

With regards solely to WPF development the two machines are fairly similar and while the difference in operating system is present the effect it has had up to this point is entirely cosmetic and has presented no challenges whatsoever.

The IDE that’s running on the school machine is Visual Studio Community 2017 Version 15.8.9, Microsoft’s proprietary IDE with seamless integration with other Microsoft development tools such as C#, the .NET Framework, WPF, and MS SQLServer. The version on the home machine is 15.9.2, whilst the versions are in fact different, these differences are not significant enough to cause any unsavoury effects.

MS SQL Server 2014 serves as the relational database management system for the solution, which is a GUI for the database where all data submitted to my application is stored.

I have made use of XAML, an XML-based mark-up language, for the front facing design aspect, and C# for the backend code and logic.

### To Be Tested/Not to Be Tested

In the context of app testing it is important to clearly outline precisely what you intend to test and, consequently, what you do not, this section is dedicated to outlining this.

For the most part due to the fact this app is developed by a single person the testing strategy is very much along the lines of ‘test everything, always’, however there are specific areas of interest that a developer should pay close attention to. Namely, the connection between the program and the database, this has proven to have some difficulties in the past and as such it is important to focus on it, perhaps by creating a new account and logging in every time.

Aside from this not much seems to require immediate or constant attention, this is due to the fact functionality for the app is fairly simple as of now.

## P - Designing a Comprehensive Review Strategy for Updating and Maintenance

The intent with the application going forward will be to place it in the common three-stage soft release structure, a private alpha version, a more public beta version, and then a full release (v1.0). During Alpha and Beta testing feedback from the users of the application will be beyond encouraged, these stages will be the periods of greatest change in the app.

After releasing v1.0 it is more than likely that we, as developers, will release a separate, more unstable ongoing beta version of the app to try out publicly, this method will be similar to that of the popular instant messaging and VOIP service Discord, who have the official stable release alongside another, separate application they dub ‘canary’ named as such after the proverbial canary in the coal mine, which allows users to test out new features.

Version numbering will follow a fairly strict process, We will start with version 1.0.00 and each update will be categorised by substance, for example a small bugfix will increment the 3rd 0 by 1 (making it version 1.0.01), slightly more substantial bugfixes and security improvements will increment the 2nd 0 (making it v1.0.11), small content additions will increment the 1st 0 (v1.1.11) and large content updates will increment the 1st digit (v2.1.11).

Updates will go live every second Tuesday, allowing for a two-week development cycle and 1 day before release day in the work week for bugfixes and final touches to the new update.

The intent is to increment at the end of a six-week “sprint”, a period of time in Agile where a team sets out completing a set amount of work, most of these will constitute bugfixes and such, and as such increment the smaller two digits, larger content updates will take place after a number of sprints, most likely after 3-6 month periods, and then a change in major operating version will occur once every year, or approximately 52 weeks. The project will be “sunset”, e.g. ended, at the end of three or four changes in major operating versions (156-208 weeks, or 26-35 sprints). This will allow for developers to move on to a new project whilst also assuring this specific product is at the highest level of quality it can be.

Bug reporting takes priority, naturally, over the addition of most features, referred to normally as “nice-to-haves”, and as such upon the reporting of a bug in the system priority would shift away from the addition of a new feature to the fixing of the bug, most likely, however, the more notable bugs will be fixed and prioritised in the first cycle or two (6-12 weeks) after a new operating version comes out, such as 1.1.

## M - Analysing the Results of Testing and Document Actions Taken

In order to truly analyse and demonstrate my use of the test plan I have decided to document a test, making use of the test plan as laid out above, on my application once I believe it has been completed and fully developed. This allow me to go through a full walk through of the program and as such take a more head on approach to working out bugs and such.

Launch successful after approximately 1 minute, may need to look at efficiency and symbol loading

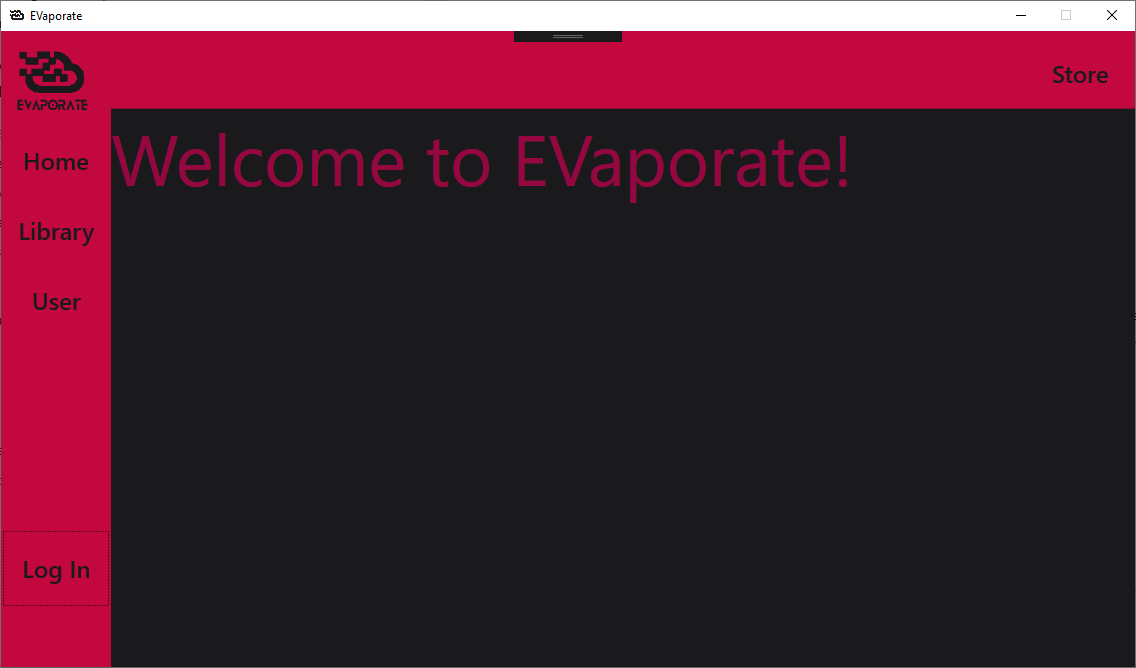


Figure 33, Successful launch

User was not successfully added to the database, as one can see by the image below. At this point I decided to stop debugging and begin searching for errors in my code:

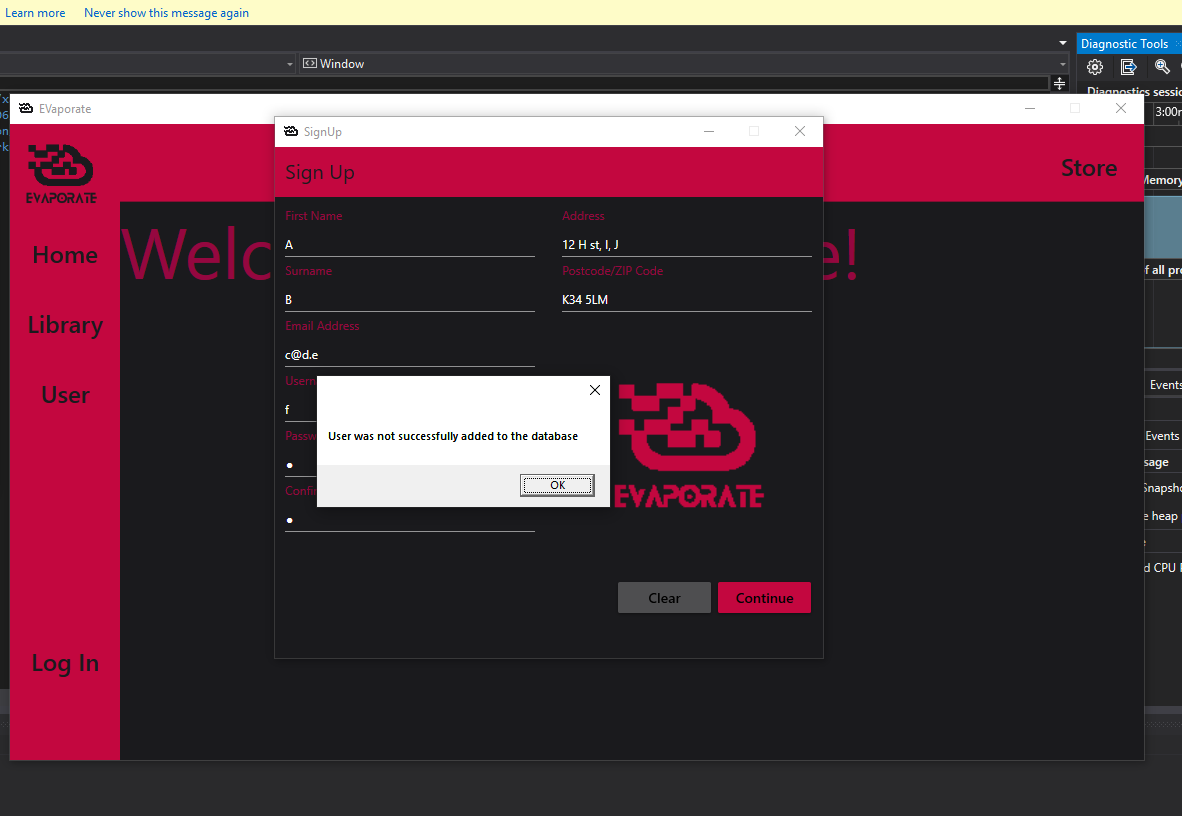


Figure 34, User was not successfully added to the database

In order to actually view the exception, I had been advised by a fellow developer to add a local variable after (Exception) in my Try/Catch method for the message box and then add code which showed the message attached to the variable, this would be in order to see what the generated exception was. This generated the below message box:

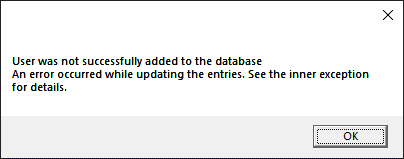


Figure 35, the error message box

This then prompted me to set a breakpoint on the beginning of the statement and step through it until the exception was reached. After doing this I noticed that in the locals menu, UserID was set to 0, which was already occupied by another previously created account, now it was a matter of incrementing UserIDs to fix the error.

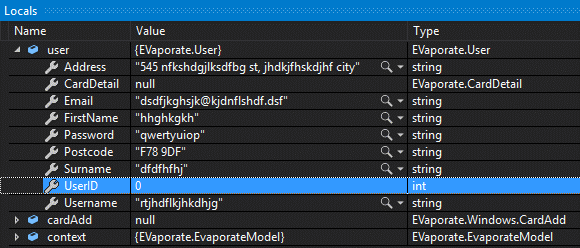


Figure 36, the Locals box, showing UserID to be set to 0

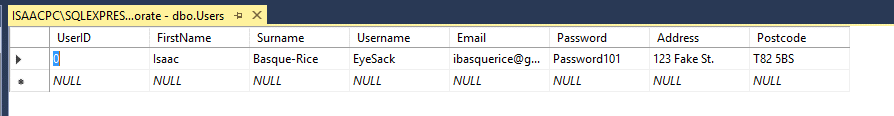


Figure 37, the Users database table showing the UserID column to already have 0 occupied

This was additionally confirmed when I removed the message box and viewed the exception it was throwing in particular, which was as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | InnerException | {"Violation of PRIMARY KEY constraint 'PK\_Users'. Cannot insert duplicate key in object 'dbo.Users'. The duplicate key value is (0).\r\nThe statement has been terminated."} | System.Exception {System.Data.SqlClient.SqlException} |

Figure 38, the grid showing the exception as copy-pasted directly from the exception manager window, the highlighted section shows my suspicions to be correct

After some searching, I discovered in my SQL Server RDBMS that I had the “Identity Increment” property set to false on my UserID Primary Key, this meant that it would not try to increment the ID every time, meaning additionally that it would remain at 0 and fail every time following the first addition to the database table.

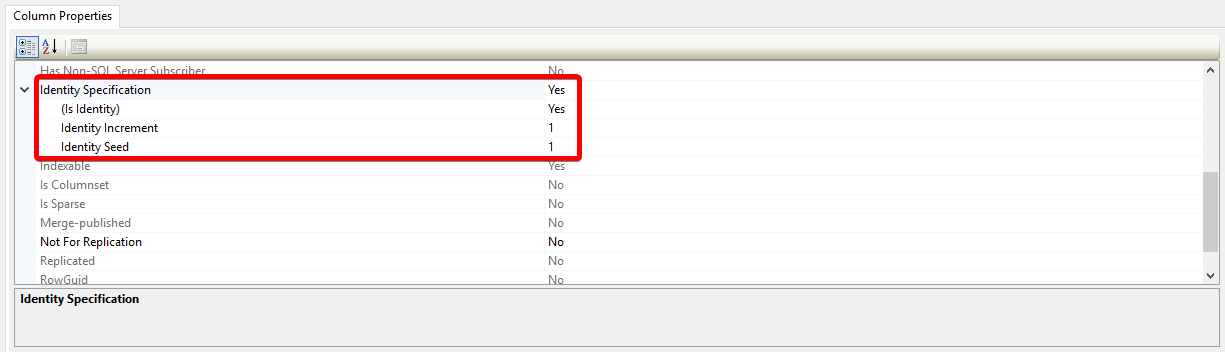


Figure 39, the UserID Properties window with Identity Specification (the relevant field) highlighted

Believing I had fixed the issue I then returned to the strategy at the point it had been left at, step 2. Fortunately, this had indeed resolved the issue, and I was able to create a new account (albeit with some stuttering).

The next step, adding a card, also produced an error, being that it was in approximately the same circumstances, before I even began to debug again either settings changed, I checked the relevant table in the RDBMS for the same issue as before. This, of course, was the case, and as a result of this I performed the same procedure as before and the result here was the same as the result with the user additions.

Beyond these issues I have also taken the liberty of creating a testing table, which I will display in two grids herein. These grids will demonstrate the results of tests carried out on my application using the different aspects of the plan. The first grid is concerned with the description of what I am testing, and what the expected outcome of the test is, and the second grid is concerned with the actual outcome and what steps I have taken, if required, to remedy the issues that may have arisen. I have divided the plan into two grids for two reasons: Firstly, to allow for some room to write in each of the cells and not have it cramped, but also in order to differentiate between theory and what actually happens. Both grids are numbered with corresponding test numbers, Test 1 on Grid 1 is the same as test 1 on grid 2, and so on.

|  |  |  |
| --- | --- | --- |
| **Test No.** | **Description** | **Expected Outcome** |
| 1 | Initial launch | Application launches successfully, no build errors, able to use it in whatever state it’s currently in |
| 2 | Basic control addition and functionality (opening new pages and so on) | On clicking a button for a new page in MainWindow to open OR for a clear button to clear all fields in a form |
| 3 | Opening a new window using a control | On clicking specific buttons for windows to open (for example a login/signup button) |
| 4 | Ensuring styling is present and correct in all windows/pages (material design may affect this) | Perhaps a few misplaced controls or strange changes, this is due to the fact that I did not initially design in material and the changes to material can have unforeseen consequences |
| 5 | Making sure game windows display properly | On clicking a game button in the store page, a window for that specific game opens. |
| 6 | Making sure connectivity to the database works through the data model | Database connection to be successful |
| 7 | Adding a user to the database | User addition to be successful and progression onto card addition being smooth |
| 8 | Adding a card to the database | Card addition to be successful and the window to close without error message or exception |
| 9 | Logging in successfully | Using an already known account to sign in to the application without exception or error |
| 10 | Purchasing a game | The card confirmation screen to come up and, upon validation, the game to be added to the user’s account |

|  |  |  |  |
| --- | --- | --- | --- |
| **Test No.** | **Actual Outcome** | **Actions taken (if necessary)** | **Retest required Y/N'** |
| 1 | The application launched with no issues | n/a | N |
| 2 | Buttons were added in the positions I stated and they functioned well, opening pages and clearing and such, however they looked a bit off | Height, width, and positioning of the buttons were changed to appear more aesthetically pleasing. Additionally, there is a problem with the buttons appearing white when clicked | Y |
| 3 | When pressing a button, the desired window was opened | n/a | N |
| 4 | After implementing MaterialDesignInXAML, whilst all previously implemented styles were still present, they looked a tad strange, example: buttons moved around and placed themselves in strange positions I did not dictate | Moving the buttons and other styles back to where they were using the visual designer fixed the issues. | N |
| 5 | On pressing each individual button, a blank window would appear and nothing else, this was due to the fact I could not, at the time, figure out how to put that functionality in. | I connected the game window to the data model and then assigned each button a tag corresponding to the ID of the game in question, e.g. Overwatch was given tag number 1 on the button and because it was connected to the data model on launch the window displayed the relevant information which I had pre allocated through the use of the “binding” feature in XAML. | Y |
| 6 | As evidenced by the GameWindow issue being resolved, this was not an issue, additionally, as will be seen later, any issues with creating accounts/adding cards etc. are not to do with the connection | n/a | N |
| 7 | As described above, there were issues regarding database incrementation and such | Fixed by enabling incrementation on the database side. | N |
| 8 | Same issue as above, database incrementation was not present due to an oversight in early stages of development. | See above | N |
| 9 | Logged in successfully | n/a | N |
| 10 | Purchased game successfully | n/a | N |
| 11 | Retest of test 2 | No issues found apart from an aesthetic one where buttons appeared white when clicked, this issue has gone since the addition of MaterialDesignInXAML, however | N |
| 12 | Retest of test 5 | No test necessary, the function worked perfectly as designed | N |

## M - Explanation of the Reasons for Reviewing and Maintaining the Program

Program maintenance is crucial for several reasons, what follows is a list of the primary reasons as to why a developer may wish to maintain their code and an explanation of each of these.

Human Error is the first of these reasons, in the process of developing any given application it is more than likely that, given that a human is writing the code and that humans are intrinsically imperfect when it comes to writing code and/or expressing logic (as compared to machines whose only purpose is to do this), It is only natural that issues will appear in the program from time to time. Most of the time, luckily, the issues are caught by the compiler and expressed to the programmer through the use of an error message.

However, in some instances there may be issues, or ‘bugs’ in a developer’s code that are not caught by anything or anyone in the initial phase of development, it would undoubtedly be bad practice for a developer to leave these bugs in the program, which means that maintaining the program is the only logical and right course of action. These bugs can pertain to a number of things, security issues such as having a sign-up form open to SQL injection, memory management problems, graphical issues, and so on.

Next, of course, we have technological advancement, it is a matter of fact that almost all programs will be updated, this includes tools one may use for development, as such it is good practice to continue to develop an application in order to keep up with these other updates. Additionally, it may be useful to a developer to update their app in order to take advantage of hardware advancements, perhaps in order to accommodate a superior backend structure or, in the case of many video games, adapting their game to be played in Virtual Reality.

Finally, we come to performance improvement, this factor is essentially an umbrella for all of the previous factors in addition to many more, the main reason one would wish to update an application is in order to improve performance, not just necessarily from a software standpoint, although optimising apps to run with as little memory requirements as possible is always a good thing, however improving performance from a usability standpoint is crucial, also.

## D - Evaluating the Effectiveness of the Test Plan

The test plan that was devised for this project had a slew of pros and cons that I did not necessarily foresee at the beginning of the development process. Ultimately, however, I can say with confidence that this plan did help me in developing the application, specifically where bugs are concerned, however, as one can imagine, this came at the expense of a few things.

One such thing was time, due to the fact that the testing method was fairly thorough, stepping through the process took a significant amount of time that could have been spent somewhere else. Ideally upon retrospection, there may have been some areas of the test plan I could have shaved off in order to save time, for example if I noticed there was an issue in the card details section it may have been easier and simpler for me to skip directly there instead of having to go through the creation of an account and so on and so forth. However, on the other side of the coin so to speak, the benefit of this was peace of mind that I had a functioning app, as least as far as some features were concerned, and that a change in one part of the app did not start a sort of butterfly effect that caused unforeseen consequences on another part of the app that may seem unrelated, as is so often the case in software development.

Another example of somewhere where the test plan failed was in efficiency, due to the fact that the test was, by their nature, conducted on the front end of the program, it made improving my code fairly difficult in that because the program simply worked, I believed, during development at least, that no improvements to the code were required, however this was not necessarily the case. Improvements with regards of efficiency and memory management could absolutely have been made, but for the most part were not due to the fact the test plan was not concerned with this in the slightest, in future improvements to the plan I intend to add this feature.

# Understanding How to Produce Documentation

## P - Creating User Documentation for the Program

### User Instructions

This section of the document concerns teaching you, the user, normal usage of the Evaporate application. It’s only natural for a series of instructions be set out for frequently used aspects of the app, as such, below are these instructions for signing up for an account, logging in, and purchasing a game, the three main functions I the app itself.

First, we start with signing up for an account. The way to do this is simple and is as follows:

1. Press the “Log In” button in the bottom left hand corner of the main window button, this will open up a new window prompting you to log in
2. When the window opens, press the “sign up” button in the bottom right corner, this button is a dark pinkish colour and stands out from the other two
3. Fill out all the details it prompts you to (name, email address, and so on), and then hit “continue”, a new window will come up prompting you to input your card information
4. Input the required information and click “submit”, clicking “sign up” will revert you back to the sign-up window, assuming there have been no errors along the way, your account has been created successfully

Another thing you may need to be instructed on is the method by which you can buy a game, this, similarly, is an extremely simple process and nothing to worry about:

1. If you are not already logged in, do so.
2. Navigate to the “store” page in the top right-hand corner of the main window
3. Now you will be presented with a list of games, click on the one you wish to view
4. A new window will then appear giving details of the game you wish to buy
5. If you wish to but the game, press the “buy” button in the bottom right hand corner
6. A new window will pop up prompting you to put your card details in, do so and then hit the “submit” button in the bottom right-hand corner of the window
7. Assuming there are no errors, the game has now been bought

### Install

The installation process is, once again, extremely simple. Once you have received the installation files, click on the “setup” icon, it should look something like the highlighted file below:

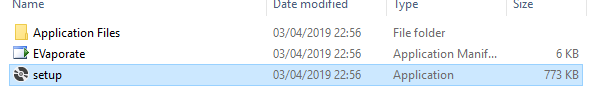


Figure 40, click on Setup

Once you have clicked on setup a pop-up window should appear asking if you’re sure you want to install the app, this pop up should look something like this:

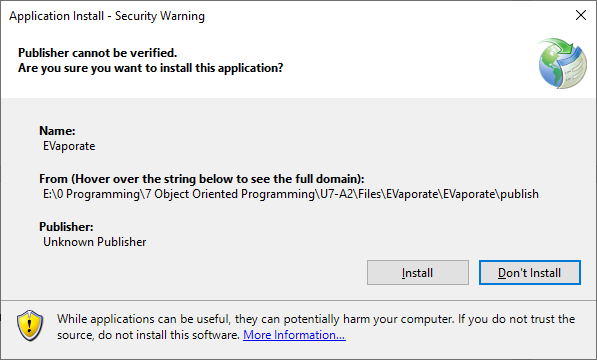


Figure 41, the application installs warning pop up

Click “install”, then this window should appear showing you that it is, in fact, installing. Once this is done, the application should open and that’s the installation process over

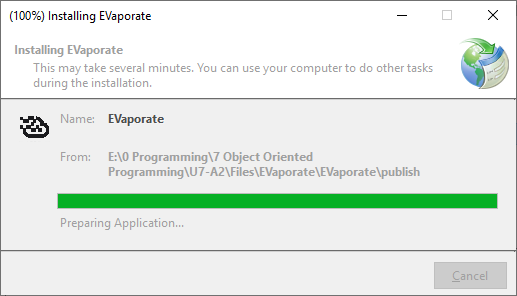


Figure 42, the installation confirmation window

### Uninstall

In order to uninstall the application, one must navigate to the control panel. You can do this in a number of ways, searching “control panel” in the windows search bar is the easiest of the options, however, and it’s recommended you do that. After you are in control panel, navigate to “Control Panel\Programs\Programs and Features”, you should be greeted with a window that looks somewhat like this:

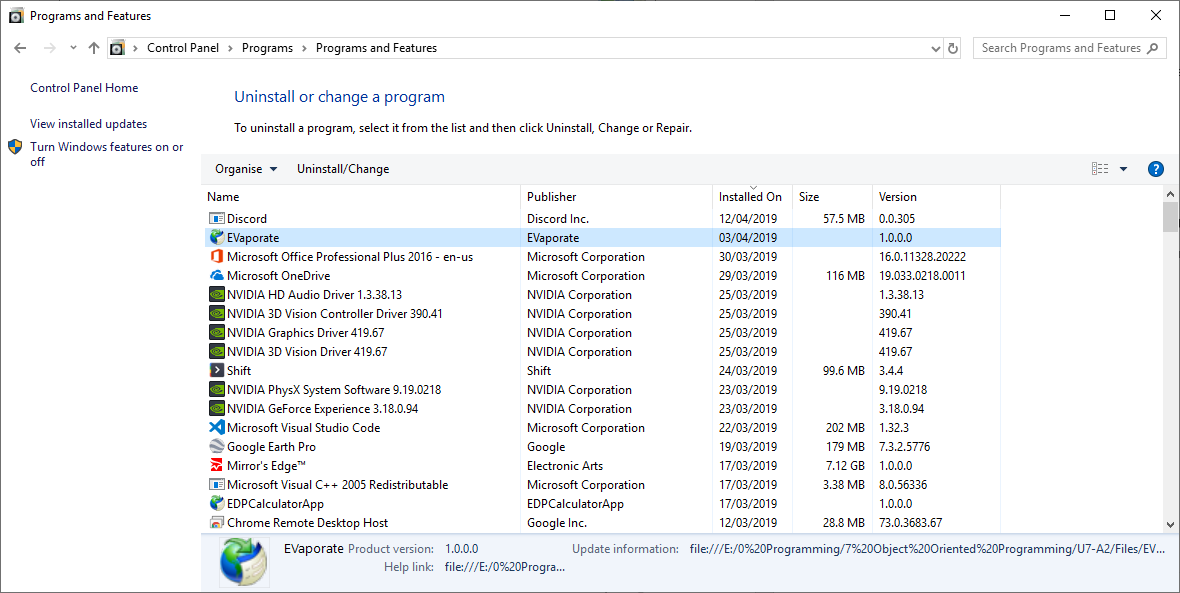


Figure 43, the uninstall window in control panel

In order to uninstall Evaporate, all one needs to do is click the “Uninstall/Change” button at the top. After you do this, you are greeted with this window:

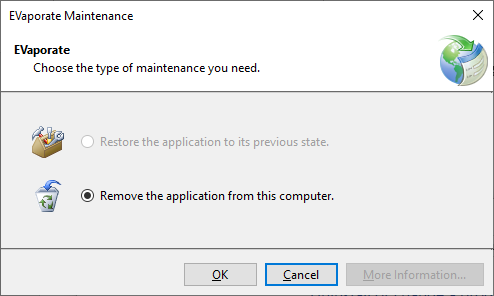


Figure 44, the uninstall window for Evaporate

Hit “OK” and then the application will be entirely uninstalled.

### Screenshots

Screenshots do exist above, however in this section I will be going through (almost) all the pages it is possible to visit in the app in order for you, as users, to familiarize yourself with the app.

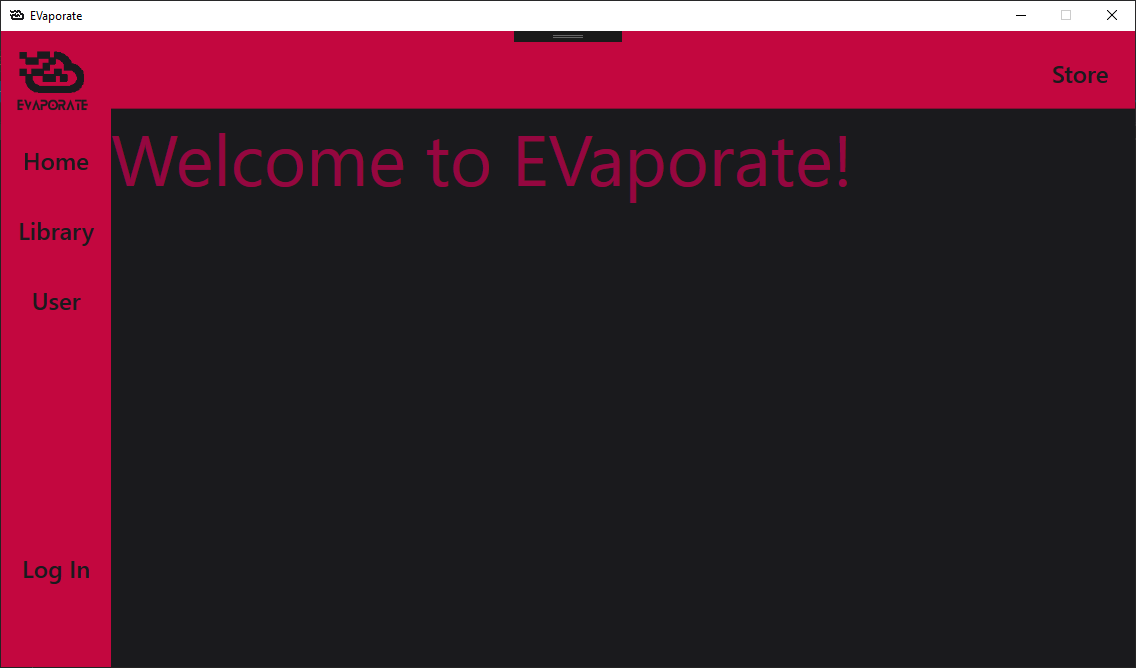


Figure 45, the welcome page, what greets you when you first load in

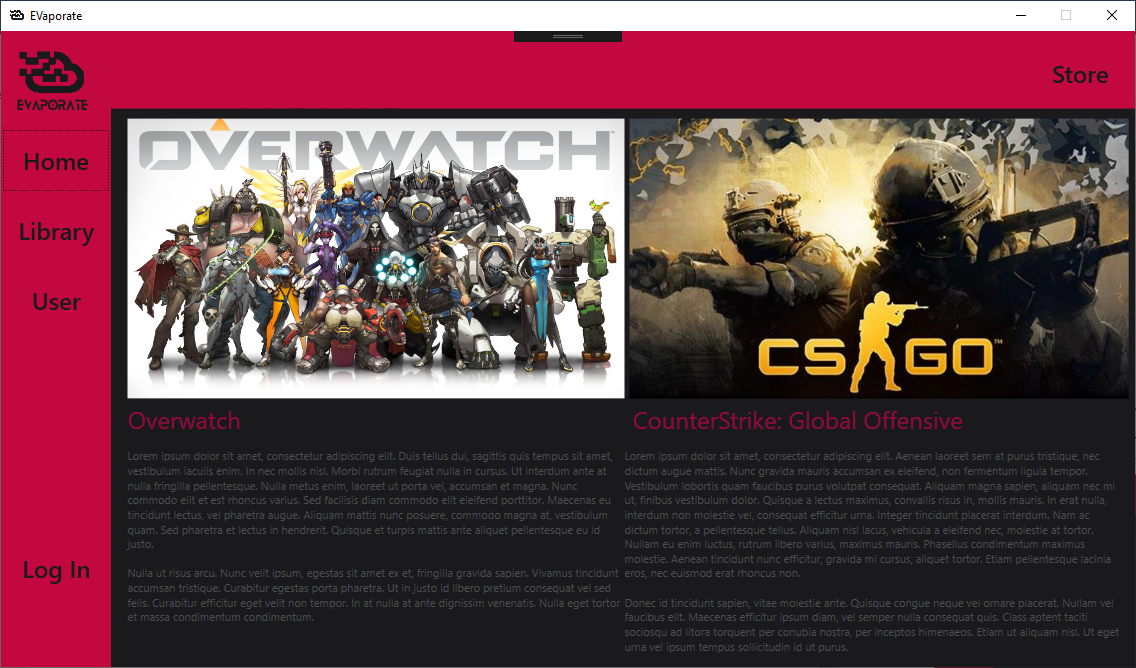


Figure 46, the home page, will be filled with news and other information

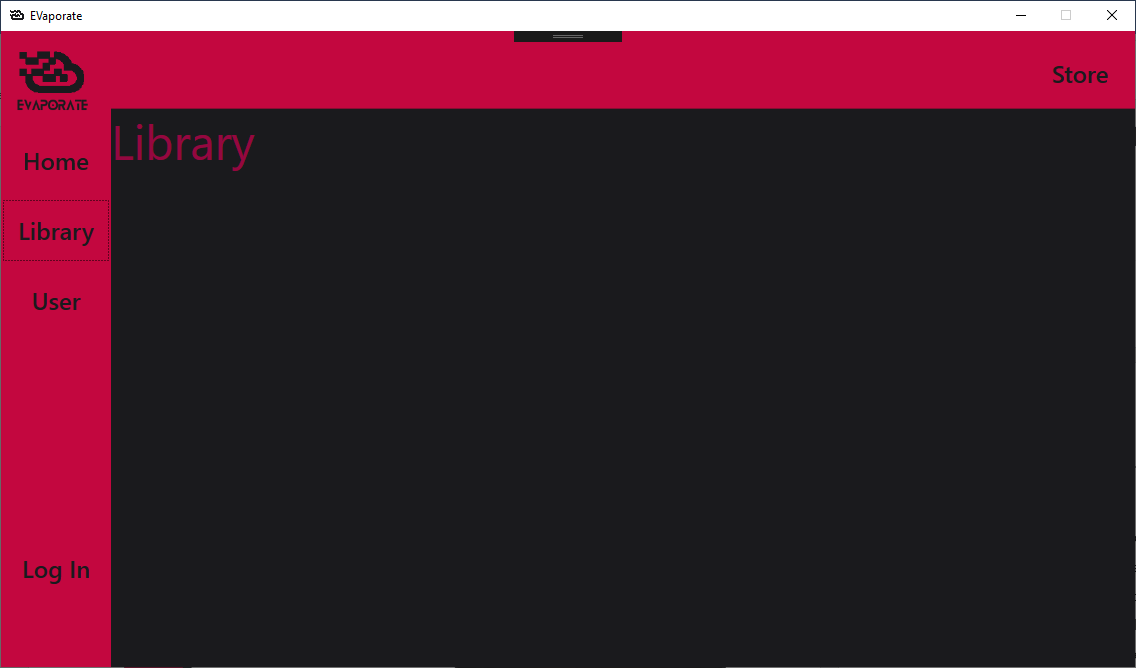


Figure 47, the Library, currently empty

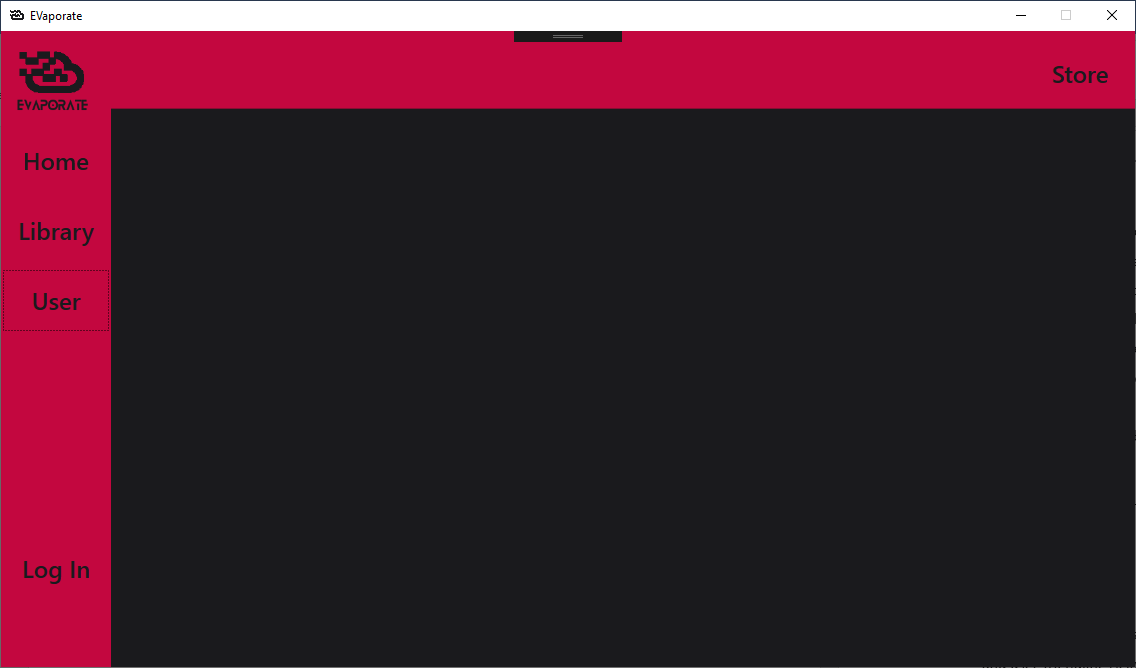


Figure 48, the User page, also currently empty as I have not logged in

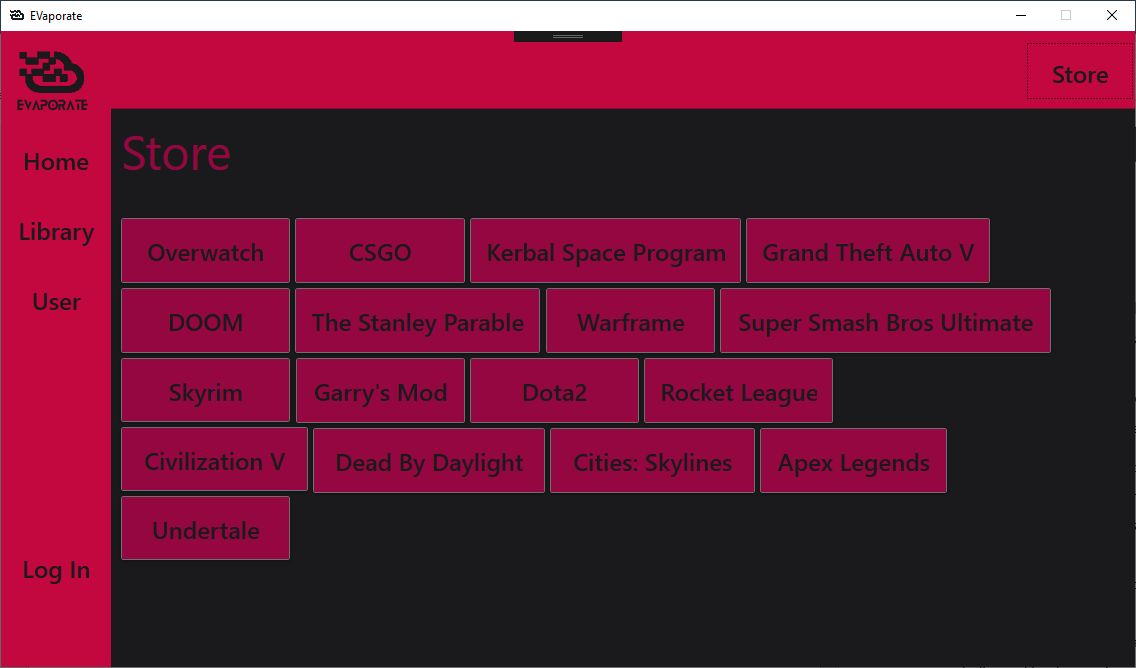


Figure 49, the Store page, showing off our wide collection of games

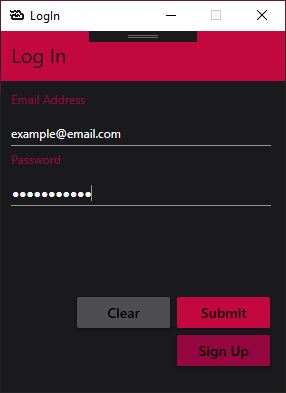


Figure 50, the Log In page with fields filled out, press submit of you wish to submit that information as correct

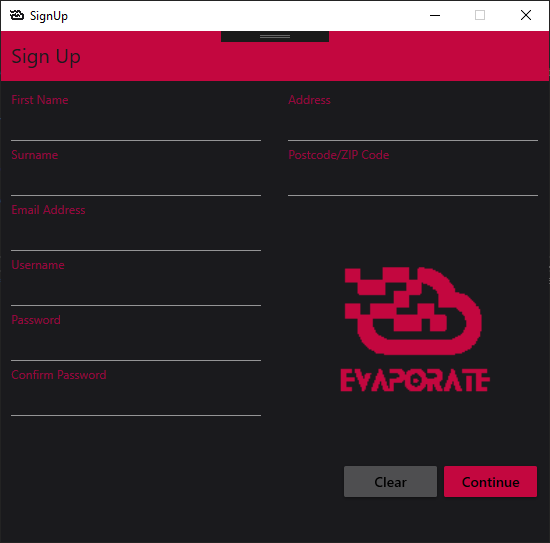


Figure 51, the sign-up window

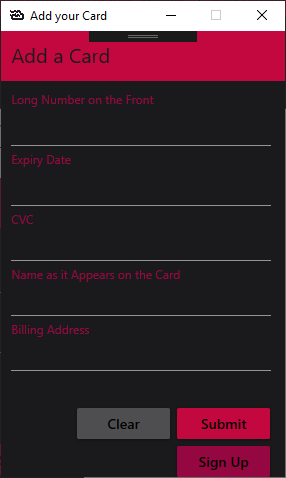


Figure 52, card addition window

### FAQ/Troubleshooting

* How do I receive the games?

Through the postal system, this is the reason you need to input your address and post/zip code. You will receive your game as fast as possible permitting location (The US and UK will most likely be fastest)

* What if my account is compromised?

Request a password change at once if you think your account may be compromised, you can do this by contacting us, there will be a feature in future to change password in the app which we are working on hard.

* How do I close my account?

Once again, simply contact us with a request to take down the account, we will need full user details to complete this to confirm that you are the account holder.

* Will I need to update my version of the app?

We are currently in the process of adding a whole bunch of new features to the app, when a new update is live (which we hope is fairly frequently!) you will be notified through the app itself and be able to decide whether you wish to update.

* Any further questions, problems, or just need some help?

Email [12basque-rice.i@thestudio-liverpool.net](mailto:12basque-rice.i@thestudio-liverpool.net) for any further enquiries

## P - Creating Technical Documentation for the Program

This section of the document is concerned primarily with the documentation required to continue development of the application into the future, as such, a number of areas in which the above documentation has been lacking have been identified and will be explained here for the benefit of future developers.

### Requirements Specification

Firstly, we must fill out a requirements specification, a specification for what the application should include, what its purpose is, requirements, and so on. So, to that end, let us begin.

The purpose of this application is to distribute, through the most efficient of preferred means possible, video games to customers wanting to purchase them. It must be noted of course that the application is a replacement system for a previously buggy version, and is by no means an original creation and as such the requirements of the application are very much dictated by the previous version. In addition, there is (at present) no means of downloading games, as such, the company (Evaporate) deals mostly in delivery of physical copies of video games to people’s locations.

The system, at its core, is built on a relational database that stores three key things: User data, Game data, and Card data. User data stores information such as a User’s name, username, password, email address, physical address (for delivery), and so on, card data is stored separately but is related to individual users through the relational database management system, and game data stores the game’s name, developer, description, price, and so on, for display and backend use. When a user wishes to purchase a game, their information and card information is validated against each other to insure the correct person is using the correct card, and then the game is related (registered) to the user’s account.

On top of this we have a user interface layer, built in WPF using Visual Studio, XAML, C#, and so on, which allows the user to browse games, set up and log in to an account, and so on and so forth. Due to the fact that this is what I made use of, a developer requires the following in order to ensure no issues will come up prior to the development process:

* The Microsoft Windows operating system, chosen for its wide availability, support, and user-friendliness, in addition to the fact it supports WPF
* A computer capable of running Visual Studio 2017 to an acceptable standard
* A copy of Visual Studio with the “.NET desktop Development” and “Data Storage and Processing” workloads (if one were to attempt to be as safe as possible I, at present, am using Visual Studio Community 2017 v15.9.2, if any issues present themselves and you are not on this version, please revert if possible as a first course of action)
* Microsoft SQL Server Management Studio 17 with a copy of the database

Further requirements, such as coding conventions, have been detailed elsewhere.

### Data Flow Diagrams

Below you will see a series of Data Flow Diagrams the process of signing up for a user account within my application. One can think of DFDs as a simplified form of UML diagram, in that it represents the interaction between tangible elements within my solution, but also one can think of it as a flow chart, as it represents the movement of data in a fairly linear fashion.

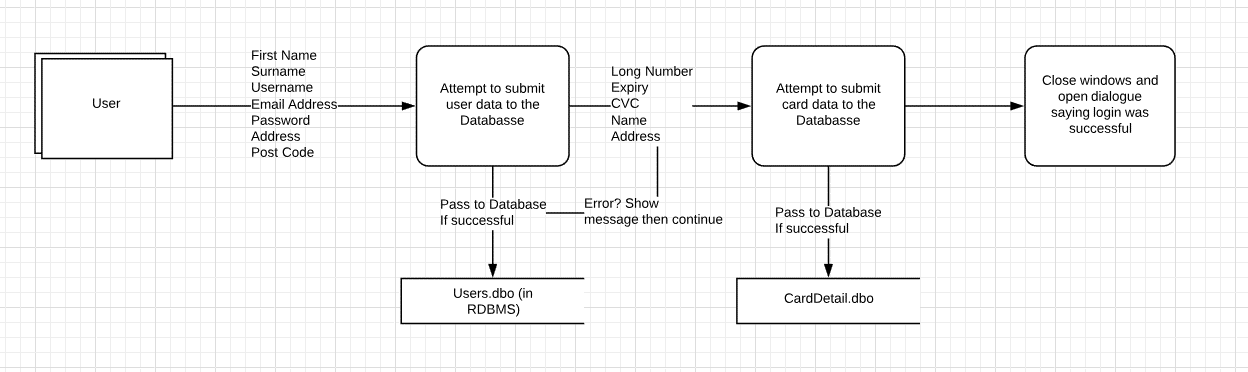


Figure 53, A DFD for the sign up method in the solution

### Entity Relationship Models

Below you will see the entity relationship model between the tables in my database. An Entity Relationship Model is a model of a series of entities, in this case a database and the tables within it, and the relationships between the elements and tables within it. AN “entity”, in this case, is a thing or object that is in some way distinguishable from the background, for example, an employee in an office is distinguishable from their surroundings. Continuing with this analogy, let’s say there is an employee, Natalie, and let’s consider her to be an Entity. Natalie has a name, a height, weight, salary, role/position, and so on, these are the Natalie entity’s attributes. In this way, you see, an entity is very similar to a class, but I digress.

This model serves to show the relationship between the different tables in the context of their use. You may, of course, notice that the Game table is not connected to the Users and CardDetails tables, this is, of course, purposeful, as all users require as a matter of utmost importance their card details, this is (in future) going to serve as an extra method of validation. However, whilst all users do require a card attached to their account, they do not require a game in their account in order to have it, as such, I have decided to separate them at this level.

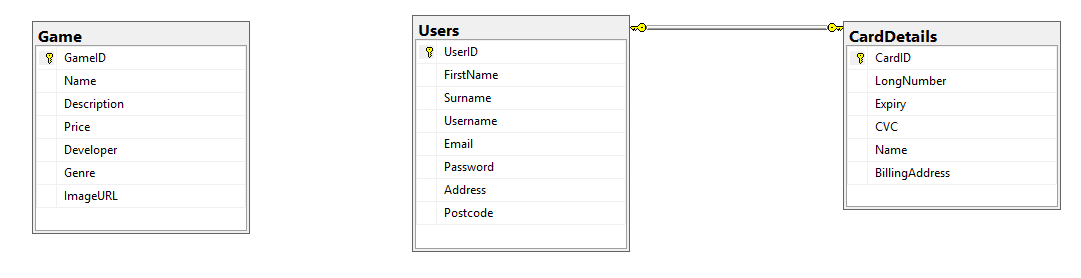


Figure 54, the entity relationship model between tables in my database

### Data Dictionary

A data dictionary is, in essence, a table filled with metadata about a data table in a database, the Data Dictionary has a name, data type, and a detailed description about what it’s used for, where it fits in, and so on and so forth. Without further ado, below is my data dictionary for the three tables I have used in my solution, a note, Int means Integer, or whole number, Nchar10 means a string with a maximum of 10 characters, and Nvarchar50 means a string of a maximum of 50 characters, whether they allow nulls or not (blank fields) is placed in brackets next to the data type.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** | **Column** | **Data Type (null?)** | **Description** |
| dbo.Users | UserID | Int (no) | The Primary Key for the Users table, this abstracts a user into an individual ID for reference later, relates to the CardDetail table |
| dbo.Users | FirstName | Nvarchar50 (no) | First name element, the user inputs this to a form on the front end of the application |
| dbo.Users | Surname | Nvarchar50 (no) | Surname element, inputted onto a form |
| dbo.Users | Username | Nvarchar50 (no) | Username element, individually chosen |
| dbo.Users | Email | Nvarchar50 (no) | Email element, allows for verification and contact |
| dbo.Users | Password | Nvarchar50 (no) | Password element, MUST be salted and hashed in a future update, allows for verification |
| dbo.Users | Address | Nvarchar50 (no) | Address element, allows for the sending of games to the client/user |
| dbo.Users | Postcode | Nvarchar50 (no) | Post/Zip code, does the same as above but is also required for the shipping of packages |
| dbo.Game | GameID | Nchar10 (no) | The Primary Key for the Game table, this abstracts a Game into an individual ID for reference later |
| dbo.Game | Name | Nvarchar50 (no) | Name of the game |
| dbo.Game | Description | NvarcharMAX (no) | Description of the game, requires as many characters as possible because many descriptions can be lengthy |
| dbo.Game | Price | Nchar10 (no) | Pricing of the game, Nchar10 because very rarely will games go above £100, however they may do in other currencies, further research required. An Nchar as opposed to an Int because Ints do not allow for currency symbols to be displayed |
| dbo.Game | Developer | Nvarchar50 (no) | Developer of the game |
| dbo.Game | Genre | Nvarchar50 (no) | Genre of the game, may be in multiple tag forms, consider having a tag database in future that games can inherit from |
| dbo.Game | ImageURL | NvarcharMAX (yes) | URL for an image, does not work yet however will do soon, MAX because URLs can get lengthy |
| dbo.CardDetails | CardID | Int (no) | The Primary Key for the CardDetail table, this abstracts a Card into an individual ID for reference later, relates to the Users table |
| dbo.CardDetails | LongNumber | Nvarchar50 (no) | Long number on the front of the card, required in many purchases, should really be an int |
| dbo.CardDetails | Expiry | Nvarchar50 (no) | Expiry date, Nvarchar as opposed to int for the same reason as Price in Game table, requires non-numerical characters (the slash) to denote specific information |
| dbo.CardDetails | CVC | Nvarchar50 (no) | Three-digit number on the back of card, should be int |
| dbo.CardDetails | Name | Nvarchar50 (no) | Name as it appears on the card, may in future be cross-referenced with the user’s name, however this may not be ideal as sometimes people may wish to use others’ cards to purchase games |
| dbo.CardDetails | BillingAddress | Nvarchar50 (no) | The address to which the game is billed, usually the same as the users’ actual addresses, but not always. |

### Class Schemas

Below you will be able to see a simple class schema for my program, this diagram demonstrates how the classes relate to each other tangibly within the solution. As one will be able to see, all three major classes inherit directly from a superclass, Entity, this class is as yet empty, however my intent moving forward is to integrate this into the program, perhaps through some kind of shared attribute(s) that will be developed at a later date.

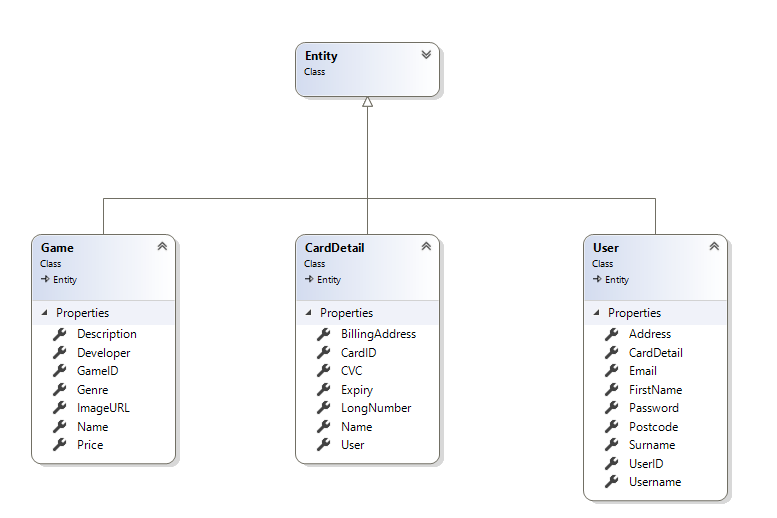


Figure 55, the main class schema for the program

### Commented Program Code

In the context of development, very frequently I have found, a developer is prompted through the warning system as follows: “CS1591: Missing XML Comment for publicly visible type or member ‘example’”, this warning appears when the compiler detects a publicly visible code snippet within a class that does not have a summary of why it exists around it. In order to remedy this, it is recommended that a summary section is placed above the snippet in question, these, by default, look like the following:

///<summary>

///class for card details, connects to db

///</summary>

public partial class CardDetail: Model.Entity

In my solution, I have replaced the word “<summary>” with just “<s>”, as it is my belief that this causes a significant degree of decluttering, making reading the comments more bearable to an extent. To that end, below is a full-page example of where I have made use of this to the fullest extent possible, in order to clear the CS1591 warnings:

namespace EVaporate

{

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

//note: <s> stands for summary

///<summary>

///class for card details, connects to db

///</summary>

public partial class CardDetail: Model.Entity

{

///<s>

///gets and sets CardID from the app input to db

///</s>

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int CardID { get; set; }

///<s>

///LongNumber to db

///</s>

[Required]

[StringLength(50)]

public string LongNumber { get; set; }

///<s>

///Expiry to db

///</s>

[Required]

[StringLength(50)]

public string Expiry { get; set; }

///<s>

///CVC to db

///</s>

[Required]

[StringLength(50)]

public string CVC { get; set; }

///<s>

///Name to db (name o front of card)

///</s>

[Required]

[StringLength(50)]

public string Name { get; set; }

///<s>

///BillingAddress, ideally would be cross-checked with actual address

///</s>

[Required]

[StringLength(50)]

public string BillingAddress { get; set; }

///<s>

///References and instantiates the user class

///</s>

public virtual User User { get; set; }

}

}

In addition to this, I have also made use of comments outside the CS1591 context, specifically in places I believe requires further explanation, a simple example of this is as follows, on the clear field button event handler:

//event handler that clears all the relevant fields

private void Btn\_Clear\_Click(object sender, RoutedEventArgs e)

{

tb\_Email.Clear();

tb\_Password.Clear();

}

A more substantial version of this is within the sign up button logic, where I first set the values of the text boxes to be the values of the data fields, and then go on to create a try/catch method that tries to submit user data into the data field, but failing that returns an error message, the code for this is as follows:

public void btn\_ContinueSignUp\_Click(object sender, RoutedEventArgs e)

{

//instantiates the 'User' object in Model and sets the value of each variable to be equal to input in its respective text box

User user = new User

{

FirstName = tb\_FirstName.Text,

Surname = tb\_Surname.Text,

Username = tb\_Username.Text,

Email = tb\_Email.Text,

Password = tb\_Password.Password,

Address = tb\_Address.Text,

Postcode = tb\_PostCode.Text,

};

//a try catch method to submit data into the users table, nested within an if statement to check whether the two passwords match

using (var context = new EvaporateModel())

{

//checking if the passwords match to ensure the user has inputted the correct one

if (tb\_Password.Password != tb\_ConfirmPassword.Password)

{

MessageBox.Show("The passwords do not match");

}

else

{

//tries to ass a user to the user model which then masses it to the db, if not possible returns an error message with some details on a new line

try

{

context.Users.Add(user);

context.SaveChanges();

}

catch (Exception f)

{

MessageBox.Show("User was not successfully added to the database" + Environment.NewLine + f.Message);

}

}

//then closes the window

Close();

}

//opens an instance of the CardAdd window to continue the login process

CardAdd cardAdd = new CardAdd();

cardAdd.Show();

}

### Documenting Identifiers

Identifiers are an extremely important aspect of design and implementation of that design into a full development context. Identifiers, in short, are the small prefixes that go in front of a control or a specific item’s name in order to classify it in a group and differentiate it from another group of controls/items. For example, the identifier “btn\_” identifies buttons, “tb\_” identifies text boxes, and so on and so forth.

Examples of identifiers used in the solution so far are (taken exclusively from the log in window), “txt\_Email”, “tb\_Email”, and “btn\_Submit”, These represent text fields (un-editable text), text boxes (editable text), and buttons, respectively.

To this end, in continuing development, it is crucial that this naming convention is continued, for the sake of continuity and global understanding of the program going forward. In the case of the addition of new controls to the solution, the addition of generally accepted identifiers is a must, there is no official standard for the naming of identifiers, however the suggested guideline for this instance would be a three-letter abbreviation in lower case if possible (btn, txt, etc.) followed by an underscore (“\_”), and then the name of the individual control beginning with an upper case (for example “btn\_LogIn”), everything after the underscore will be PascalCased, as is good practice.

### Maintaining and Identifying System Decay

System decay, sometimes shortened to software rot, is the process by which a system becomes slower, less responsive, buggier, and generally decreases in performance over time. This is not usually caused by anything wrong with the program, per se, but rather within the changing context around it (such as updates to the language, the framework, WPF, the IDE, and so on). A piece of software that has undergone a significant amount of system decay is usually referred to as “legacy” and as such will require an upgrade. This section of the documentation concerns the detection of when a piece of software should be relegated due to system decay and is not just in dire need of an update.

Almost always, software rot occurs in instances where the program has been neglected by developers in some form, when it has not been updated in a while and compatibility starts to deteriorate, having a program in “dependency hell”, where it is dependent on specific versions of other programs and requires those versions in order to function properly, or even aspects of the same program which are changed without thought for another part of it, is a real issue in this instance.

Next, let’s assume you are a developer a few years down the line, say three or four years into a development task, and the IDE you’re using, for example, is 6 years out of date, but it can’t be upgraded because doing so would cause all kinds of dependency issues and bugs in your software, this means a developer cannot move on. On the other side of this, upgrading to a new version would, in this instance, also cause software rot in that the app you, as a developer, have been developing is now woefully out of date, bugged, and unusable, perhaps. How is this problem navigated? Well, the old adage of “if it’s not broke, don’t fix it” holds true in this case, if there are no issues with the old program then, for a while anyway, there should be no issues. However, as the rest of the world moves on from an older version, support for it begins to lack, there’s no longer a wide array of additions and changes you, as a developer, can make, at this point it may be a sensible idea to “sunset” it, that is, to finish development and start on a new task.

Finally, if one wishes to maintain a program out of system decay this will, no doubt, be a difficult thing to accomplish. Multiple considerations will need to be taken into account, for example, if one wishes to chase compatibility and upgrade all their tools to meet modern development standards, then deep research into how the program would be affected by these upgrades must be taken into account, reworks must be undertaken, bugs must be dealt with in a timely manner, and, above all, the program must be vigorously tested in order to be pushed to a live state. Another cause of system decay is unused or rarely used code, in the normal process of updating a program, it is very easy to miss a spot, as it were, with regards to aspects of the program that are not commonly used, as such, a test plan MUST be put in place in order to test the program to the fullest extent possible and avoid software rot.

# References

Beck, K. & Cunningham, W., 1989. *A Laboratory For Teaching Object-Oriented Thinking.* New Orleans, Louisiana, OOPSLA'89.

Dymel, S., n.d. *Material Design In XAML.* [Online]   
Available at: http://materialdesigninxaml.net/home

Gerber, S., 2015. *10 usability considerations for your mobile app.* [Online]   
Available at: https://thenextweb.com/apps/2015/08/28/10-usability-considerations-mobile-app/

Kennedy, J. & Satran, M., 2018. *Microsoft Docs: Usability in Software Design.* [Online]   
Available at: https://docs.microsoft.com/en-us/windows/desktop/appuistart/usability-in-software-design

Kimbal, S., Mattis, P. & Team, G. D., 2018. *GNU IMAGE MANIPULATION PROGRAM.* [Online]   
Available at: https://www.gimp.org/

Richard, E., 2014. *BLANKA — Free font.* [Online]   
Available at: https://www.behance.net/gallery/15451401/BLANKA-Free-font  
[Accessed 3 November 2018].

Ridwan, M., n.d. *Predicting Likes: Inside A Simple Recommendation Engine's Algorithms.* [Online]   
Available at: https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine

Soegaard, M., 2019. *Usability: A part of the User Experience.* [Online]   
Available at: https://www.interaction-design.org/literature/article/usability-a-part-of-the-user-experience

Visual Paradigm, n.d. *What is Unified Modeling Language (UML)?.* [Online]   
Available at: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

1. Windows Presentation Foundation [↑](#footnote-ref-1)
2. A Note: Distinction criteria (Marked with a “D” at the start of the title) will always be comprised of a (minimum of one full page) write-up, Merit criteria will usually be a combination of development and write up, and Pass will almost always be solely a development task. [↑](#footnote-ref-2)
3. Unless stated otherwise, assume is a string [↑](#footnote-ref-3)
4. Visual Paradigm, n.d. What is Unified Modelling Language (UML)? [Online]

   Available at: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/> [↑](#footnote-ref-4)
5. Beck, K. & Cunningham, W., 1989. A Laboratory for Teaching Object-Oriented Thinking. New Orleans, Louisiana, OOPSLA'89. [↑](#footnote-ref-5)
6. Kimbal, S., Mattis, P. & Team, G. D., 2018. GNU IMAGE MANIPULATION PROGRAM. [Online] Available at: <https://www.gimp.org/> [↑](#footnote-ref-6)
7. Richard, E., 2014. BLANKA — Free font. [Online]

   Available at: <https://www.behance.net/gallery/15451401/BLANKA-Free-font> [↑](#footnote-ref-7)
8. A note: everything that looks like [this] in the next three screenshots are the fields within the database [↑](#footnote-ref-8)
9. Kennedy, J. & Satran, M., 2018. Microsoft Docs: Usability in Software Design. [Online]

   Available at: <https://docs.microsoft.com/en-us/windows/desktop/appuistart/usability-in-software-design> [↑](#footnote-ref-9)
10. Soegaard, M., 2019. Usability: A part of the User Experience. [Online]

    Available at: <https://www.interaction-design.org/literature/article/usability-a-part-of-the-user-experience> [↑](#footnote-ref-10)
11. Gerber, S., 2015. 10 usability considerations for your mobile app. [Online]

    Available at: <https://thenextweb.com/apps/2015/08/28/10-usability-considerations-mobile-app/> [↑](#footnote-ref-11)
12. Ridwan, M., n.d. Predicting Likes: Inside A Simple Recommendation Engine's Algorithms. [Online]

    Available at: <https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine> [↑](#footnote-ref-12)