Project implementation report
Project GameStop

Olivér Izsák

Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

xizsak@stuba.sk

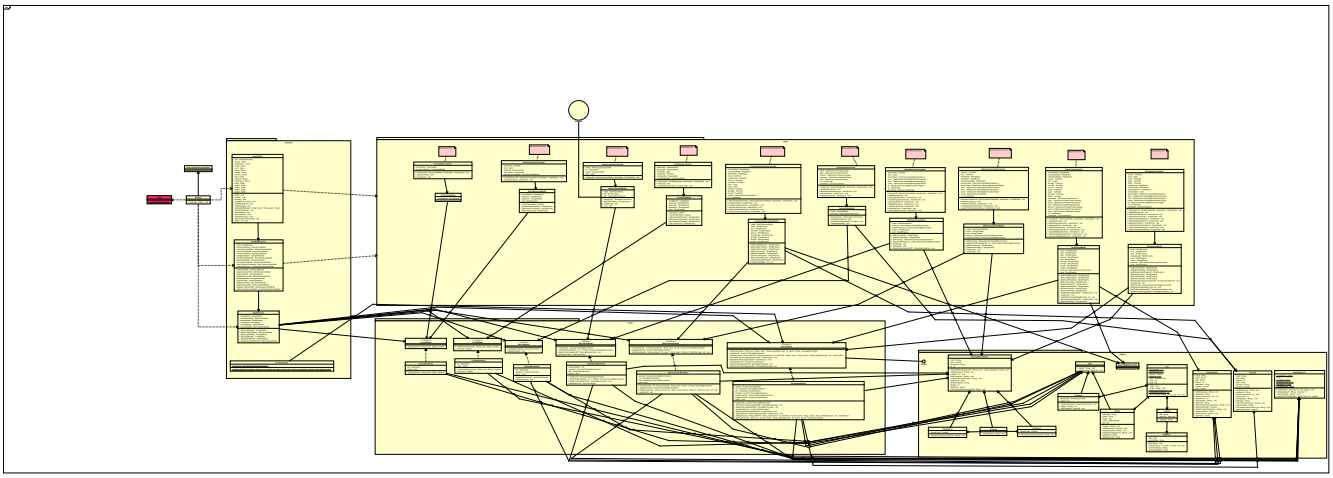
16. máj 2021

Project Intention:

Project GameStop was designed for keeping track of data, demand and orders of different video games and gaming merchandise.

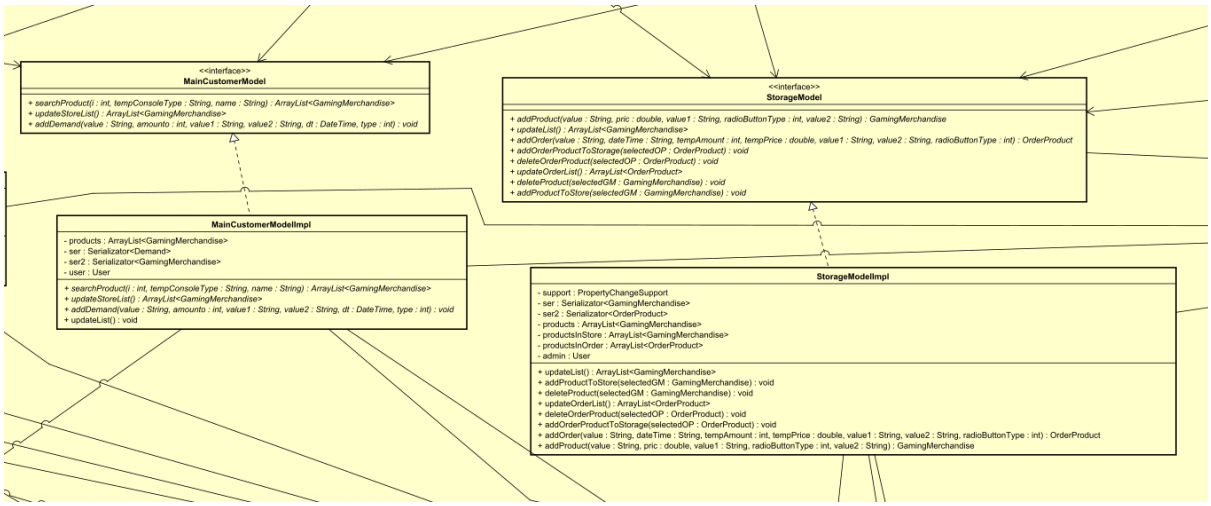
The goal of this project was to create a system that can make the work of the employees, managing and buying the products easier more cost and time efficient. The system can be used by the employees and also by the customers. The customers can only use 2 functionalities of the system. One is checking whether a store has a specific product. Second one is to make a demand, which means the customer can write in the information about a product that he would like to have in the store. This way the management will know which kind of products are in high demand and that they need to order more of that product. On the other hand, the employees have access to a lot more functionalities, such as checking different information about every product. They also have access to storage, where they can keep track of the amount of products they have. They can also see if there is an order in process. However to use these functionalities the employees have to know the password and name. The system will have not have a traditional account system in place, which means the employee section of the system can be used by all the employees that have access to the password and name. Meanwhile anyone can access the system as a customer.

Class diagram:



In my project I used the model view view model architectural design pattern.
Therefore it consists of 3 different parts:

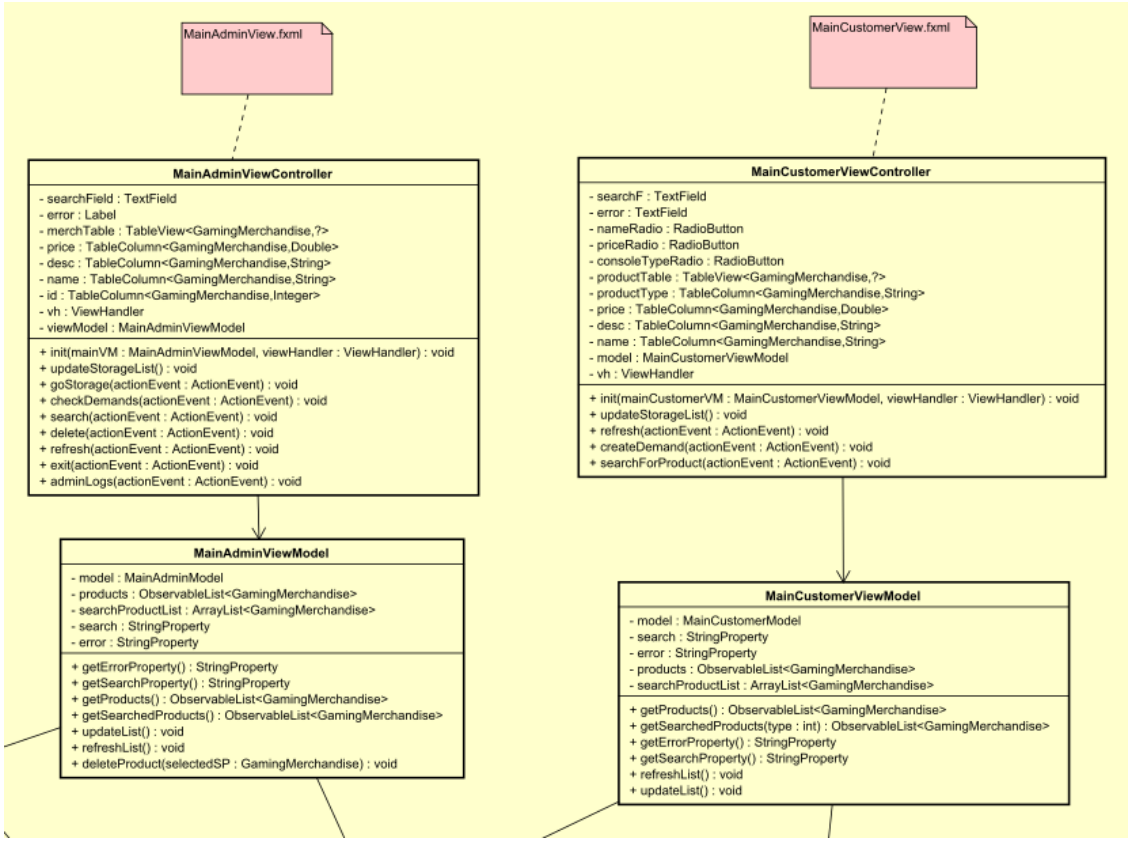
Model: containing the model implementation classes, which can be found in the model package.These classes containg the main logic of the application. All of these models have a corresponding interface to create loose couplings between the View models and the models.



View model : containing the view model classes which are responsible for checking the inputs from the GUI and passing on the values to the model classes. This class is also repsonsible for editing the GUI.

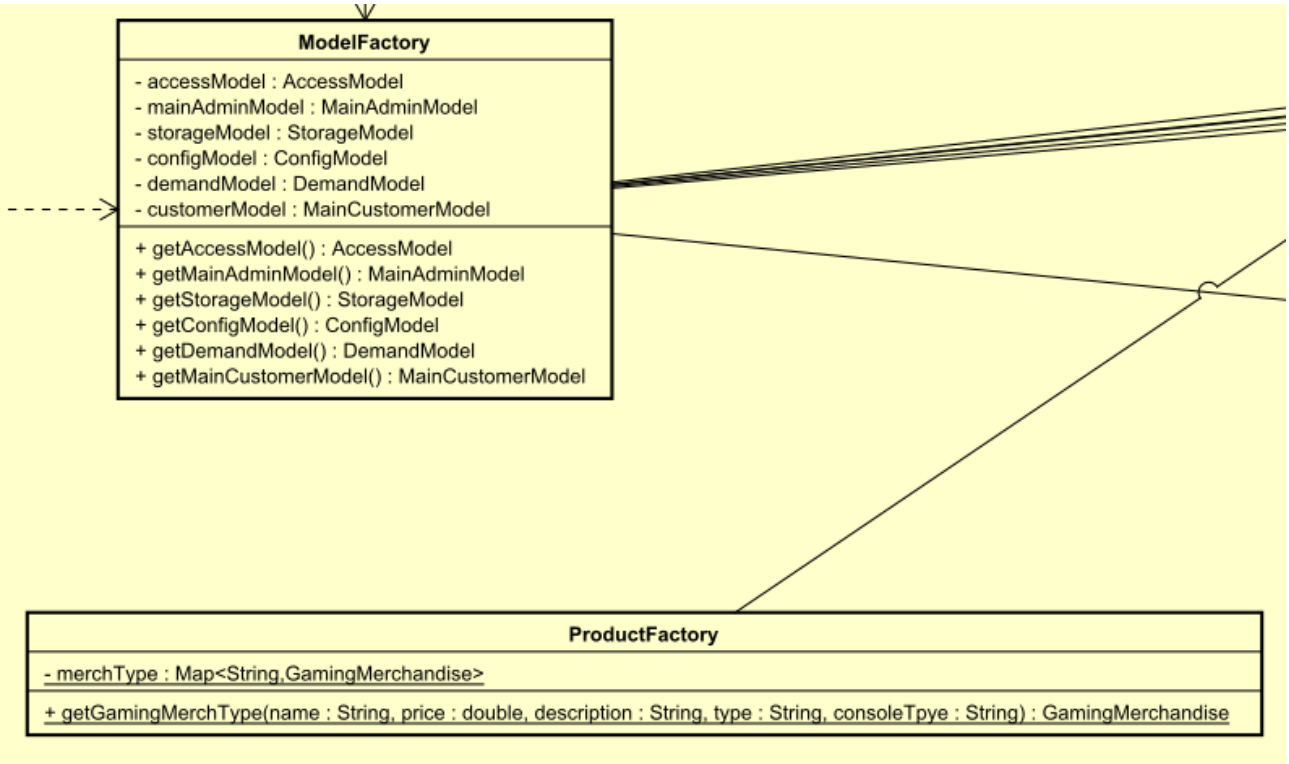
View Controller: These classes' editable and changable parts are bound to the View model thus delegating the editing of the GUI to the view model class.

View models,View controller classes, and fxml files are all classes associated with the GUI, so they can be found in the UserInterface package.



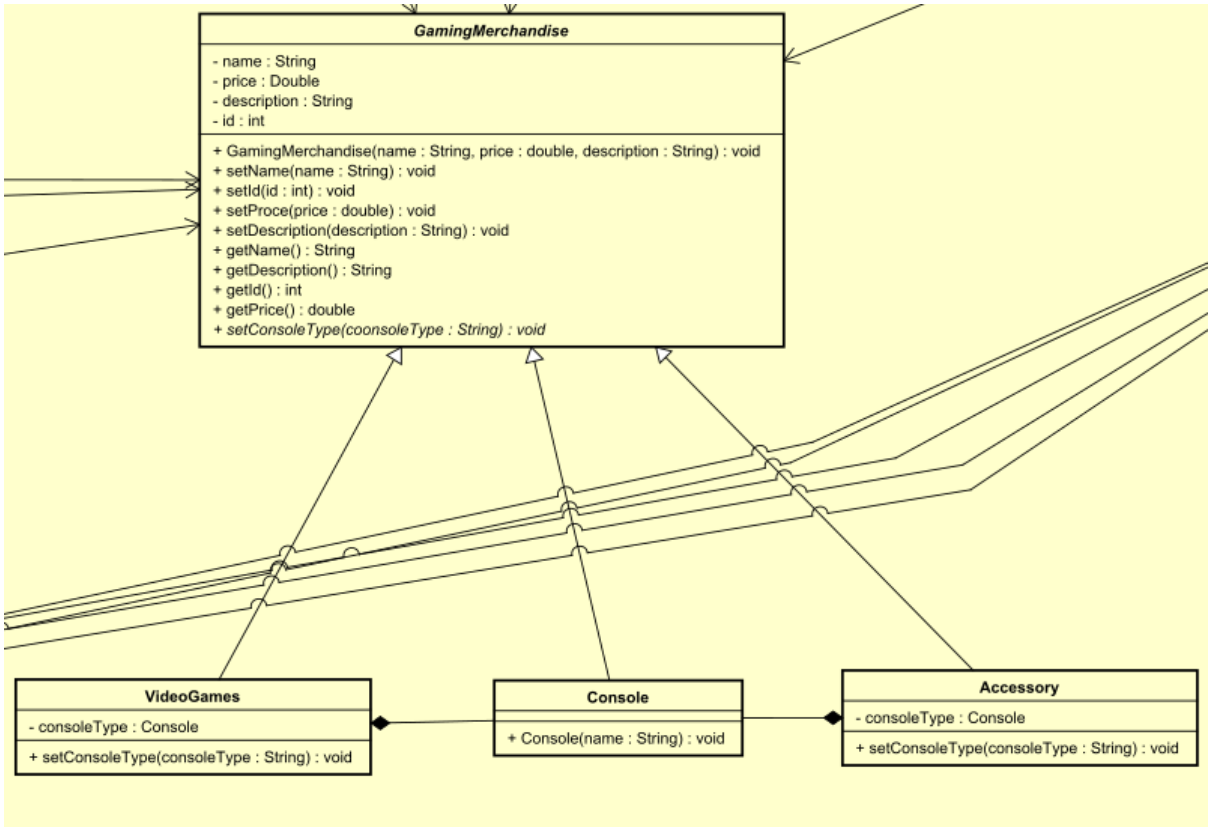
Other design pattern I use is the Factory method design pattern:
Which uses factory classes for the creation of Models,ViewModels and View Controllers.
There are classes that are tied together : ModelFactory, ViewModelFactory and ViewHandler.
The ModelFactory is responsible for creating the model classes through lazy instantiation.
The ViewModelFactory is responsible for creating the view model through lazy instantiation.
These two classes make sure that each model class is only created once.
The third class ViewHandler, is not exactly a factory class, but since it is tied together with ViewModelFactory and ModelFactory I included it in the same package as them.
The main responsibility of ViewHandler is to create different scenes for the GUI (with lazy instantiation again, so there is only one of each Scene). While creating the scene's is also ties together the Models and ViewModels to the Controller using the 2 other factories.

There is a 4th class in this package, called the ProductFactory which is not tied to the other 3, but it is a class with public static variable and method, so it can be called from anywhere and the reason behind this because this class uses Flyweight to reduce the amount of memory used for creating objects. Since this program is able to keep track of hundreds or thousands of different Gaming Merchandise objects and Products it can be quite memory intensive. So to solve this problem, I implemented a Flyweight pattern through the ProductFactory, which keep track of each invidividual product, and if an identical product is created it only returns the instance of the one that can be already found in the hah map of the ProductFactory. If we wish to create product that cannot be found in the hash map, it creates the object gives him a unique ID and adds it to the hash map.



The objects package contains all the different objects used throghout the program such as GamingMerchandise, OrderProducts, Logger, Serializator, Demand etc..

The start class is responsible for running the program, it also launches the Run.class which extends Application and has a start method. It also initializes the ModelFactory, ViewModelFactory and the ViewHandler.



Fulfillment of the main criteria:

Inheritance – Inheritance can be found in the Gaming Merchandise class, which acts a parent (superclass) for its children (subclasses) which are Accessory, Video Games, Console.
The children inherit the all set and get methods from the parent class, except the setConsoleType abstract method which is only used for Video Games and Accessory type children. An example of inheritance in use can be found in ProductFactory class, where I initialize GamingMerchandise object with one of its children.

Polymorphism – Polymorphishm can be found in the User class, which is an abstract class containing 2 abstract methods, which can be used on an User object, but it will be called on the Customer or Admin (as subclasses they have the overriden method) depending on how the User object was initialized. The method does different things depending on which children it is being called on. An example of polymorphism in use can be found in the MainCustomerModelImpl where there is a User class initialized as a customer. And then the log method is being called on the user object for example in the searchProduct method.

Encapsulation – This can be found in all classes, since it is the most basic and important concept of Object Oriented Programming. Eg.: Demand,OrderProducts etc..

Aggregation – Example of aggregation : the LogLine class is used and created only whithin the Log class.

Fulfillment of other criteria:

1.The user of design patterns:

1.1 **Model View ViewModel (MVVM)** – This architectural design pattern is responsible for separating the graphical user interface(viewmodel, view) from the business logic (model). The viewmodel is responsible for handling the view's display logic, meanwhile the model is handling the main business logic of the program. Since it is used throughout the whole program the example is the way the classes in the Model, UserInterface and Factories package are connected together.

1.2 Modified Factory Method Pattern – This design pattern uses the combination of singleton and factory method pattern. It is in combination with the MVVM pattern. It can be found in the factories package (ModelFactory,ViewModelFactory,ViewHandler). It is responsible for tying together the models, viewmodels and viewhandler. ViewHandler is responsible for creating each scene by initializing the specific models and viewmodels through their get methods.All of them use lazy instantiation, so each object is only created once, if it has already been created it just returns the same instance.

1.3 Flyweight pattern – This design pattern can be found in the factories package as the ProductFactory class. It is responsible for checking whether the Gaming Merchandise we want to create already exists, if it does it just returns the same instance, so there are no unnecessary duplicated created. On the other hand, if it has not been created before, then it creates it and adds it to its hash map, to keep track of this object.

1.4 Observer pattern – A simple observer pattern is used in the StorageModelImpl(addOrderProductToStorage - firePropertyChange) and in the StorageViewModel (constructor -addListener method called on the model, Refreshlist method updates the GUI). It works with the use of PropertyChangeSupport object and the Subject interface that can be implemented by any class that wants to fire an event. The listener has to have an instance of the class that implemented the Subject interface, then it has to call the addListener on that class' instance. This means now the listener class will be listening to the class implementing the subject, which can fire a propertyChangeEvent from a method and the listener will be able to catch it and react to it through a method.

2.Handling a created exception - Example can be found in the OrdersViewModel and in the Object package the IncorrectDateException class. In the OrdersViewModel class it is the addOrderProduct method (line 160- 174). An exception is thrown if the date of the order given is before the current date. The exception is also being caught and throws out an error message in the GUI.

3.GUI separation from the Application logic – This is handled by the MVVM pattern since the GUI is handled throught the Controller/View model classes.

4.Multithreading – This can be found in the AdminLogsViewModel/ AdminLogsViewController it is implemented as a clock in the GUI. Which creates a new thread for the counting the second of the clock and constantly refreshing it in the GUI, and there is also the main thread where the other parts of the GUI are running. In the AdminLogsViewController's constructor there is a new thread initialized and started and the AdminLogsViewModel is added as parameter to the thread. This class also implements runnable thus having a run method which is where the clock is running in a while loop.

5.Use of Generic Types – Use of generic types can be found in the Serializator class , which is responsible for serializing arraylists containing different type of object.

6.Serialization – The Serializator class is responsible for serializing different type of objects when storing them into a bin file.

7.Use of RTII – RTTI is used in the ViewHandler when using the getRootByPath method, which uses getClass() method for setting the location of the FXML loader.

8.Lambda expression – Lambda expression can be found in the AdminLogsViewModel in the run method as Platform.runLater method on updating the GUI.

List of the important version of the project development on Github :

Version 1. :

21c62086093080d5cdf717d34c9d6b0d71e555e9

Before this version there was another version which was actually the first version, where I only had the MVVM pattern and some of the main classes done. But I have got into a disagreement with github about some conflicts and I didn't manage to find out how to solve it , so I just deleted that version and reuploaded a completely new first version on Github. Where I had the most parts of Admin part of my program done.(Except the orders)

Version 2.:

6e1d57b1b85c2498e55b4b7839fb66e43993c980

In this version I managed to finish all the functionalities that was mentioned in the Project GameStop – project intention pdf. Even some extra functionalities, such as logs.
This version was almost done, only things missing was some design patterns and the inclusion of some of the main and other criteria.

Version 3.:

47b292e6c4be9bed8d56eb25d5681eabe3a4fa4c

I added the the remaining criteria I was missing from my code. Such as some of the design patterns, exceptions etc...

Version 4.:

2de864c050cb4c6924b6645a75bf9c15e8c3696f

Last version where I worked on the program. Hear I fixed some minor bugs or inconsistencies in the code. And also commented every method and class, except the get and set methods since those don't need clarification.