## Rock Paper Scissors Technical Documents

I am choosing the Xamarin Forms framework with C# language to build the given assignment and the solution is run on android and iOS platforms. Used the **MVVM, Strategy,DI,Factory ,Publish Subscriber patterns,OCP principle** to design this mobile app.

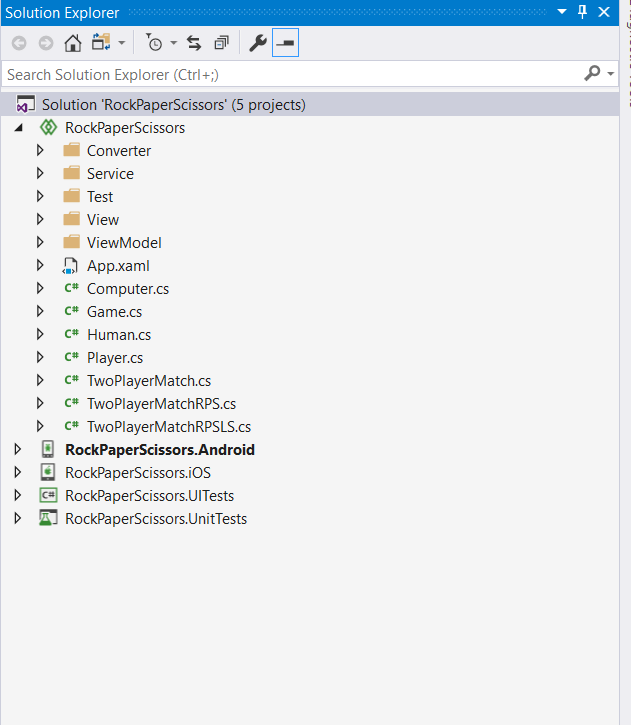
## Used Tools and SDKs

* Visual Studio 2017
* Xamarin Forms version 3.4.0.x
* Xamarin Android SDK 9.1.x
* Xamarin iOS SDK 12.4.x
* Xamarin UITest 3.0.5
* Autofac 4.9.4
* Autofac. Extras. FakeItEasy 5.0.1
* Android SDK 8.1 (Oreo)

## Application Structure

Given solution contains the following projects

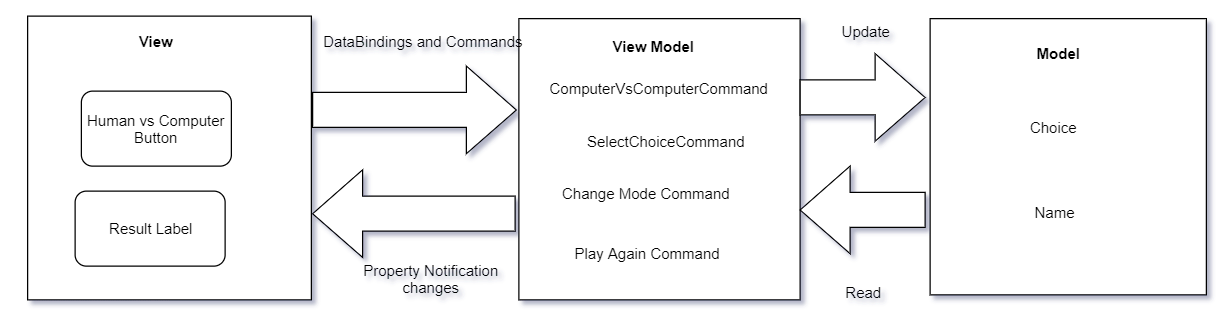
* RockPaperScissors (Contains common code used in both platforms)
* RockPaperScissors.iOS(Contains Specific code related to iOS platforms)
* RockPaperScissors.Android(Contains Specific code related to Android platforms)
* RockPaperScissors.UnitTets(Contains Unit Tests for View Models)
* RockPaperScissors.UITests (Contains UI Tests for View)



## MVVM

MVVM stands for **Model**, **View**, **View Model**.

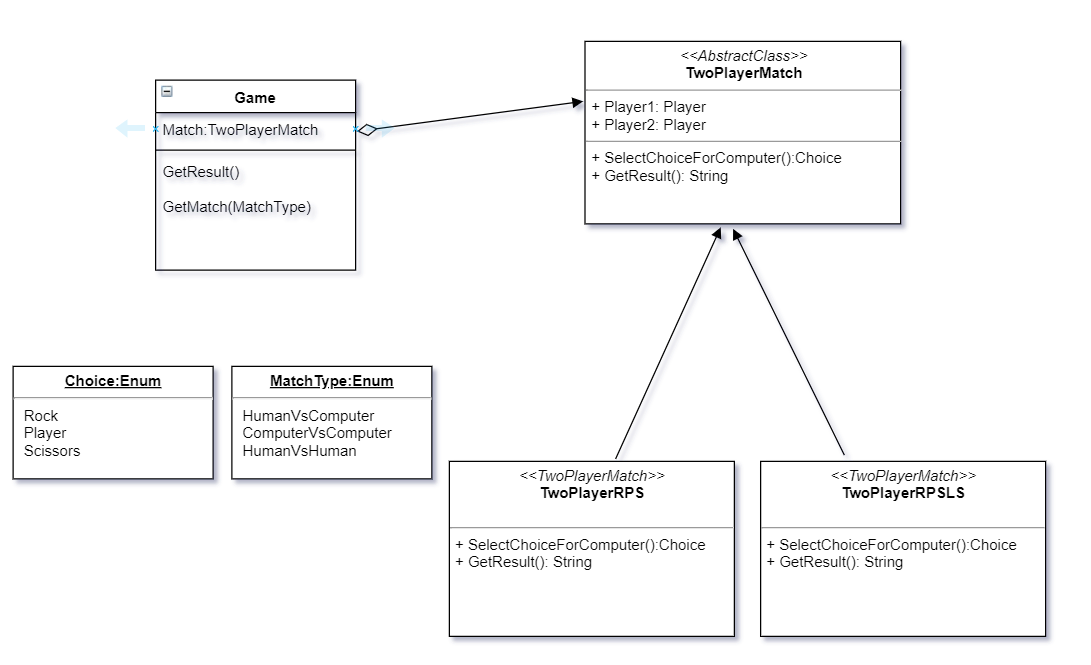
* **Model**: This holds the data of the application. It cannot directly talk to the View. Generally, it’s recommended to expose the data to the View Model through Observables.
* **View**: It represents the UI of the application. It observes the View Model.
* **View Model**: It acts as a link between the Model and the View. It’s responsible for transforming the data from the Model. It provides data streams to the View. It also uses hooks or callbacks to update the View. It’ll ask for the data from the Model.



## Strategy Pattern

Strategy lets the algorithm vary independently from clients that use it

Using the Strategy pattern to sperate the parts which frequently changed.



## Factory Pattern

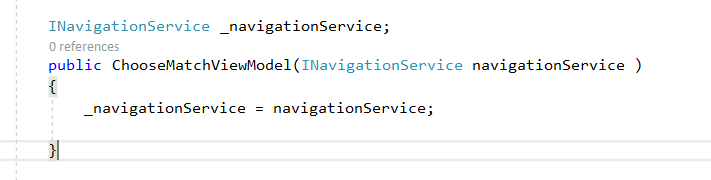
Used the factory pattern to get the concrete class based on the user selection .



## Dependency Injection using Autofac IoC container

Used dependency injection to decouple the code.

In our case passing the Navigation Service class to View Model class through the constructor. Thus I have implemented the DI.



### Publisher Subscriber Pattern.

This pattern is used to decouple code. It allows communication between two classes that know nothing about one another. In our case, we used this pattern to unit test the view model.

**Subscription**

MessagingCenter.Subscribe<PlayMatchViewModel>(this, MessageKeys.PLAYMATCHLANDEDKEY, (sender) =>

{

\_iSAppLandedOn = true;

});

**Publication**

MessagingCenter.Send(this, MessageKeys. PLAYMATCHLANDEDKEY);

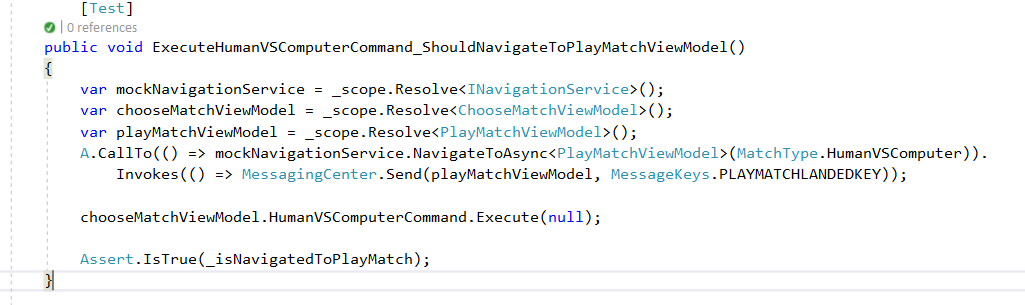
## Open close principle

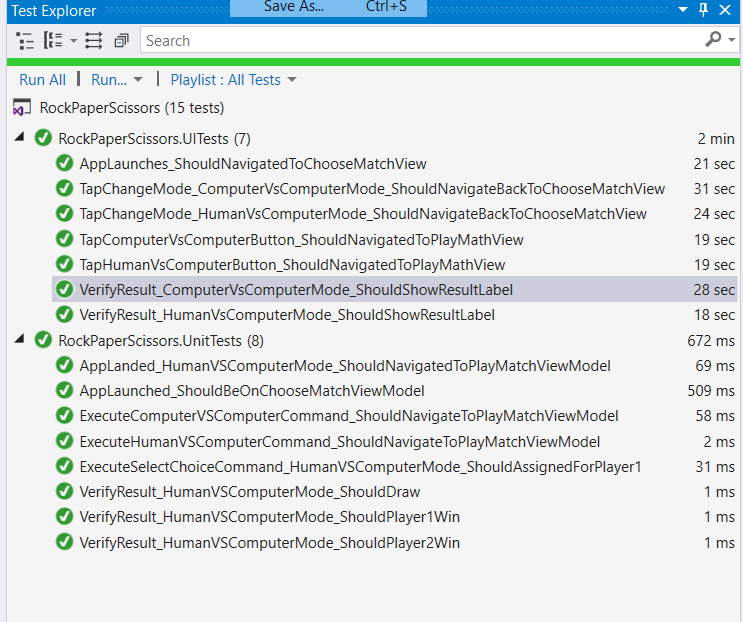
We can easily extend the given solution with out modify the existing code. From this solution we can play Rock,Paper,Scissors game. We can extend this solution with out modify the existing code if want to play Rock,Paper,Scissors,lizard,Spock game.



## Tests

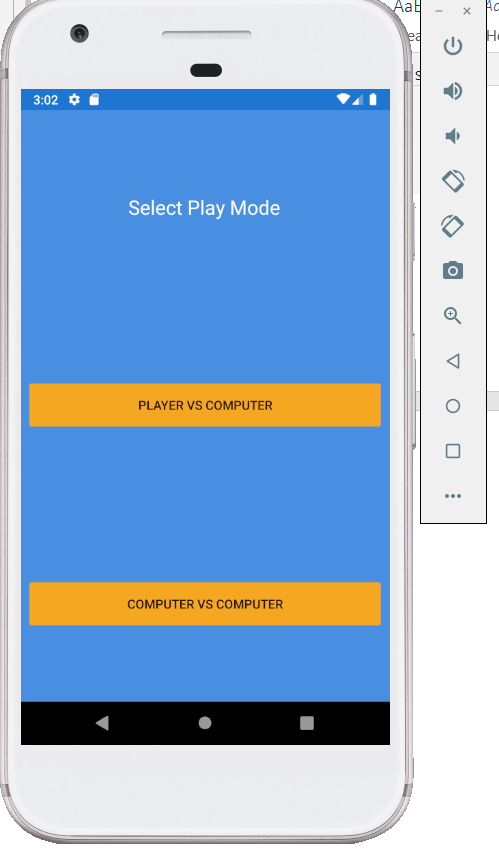
Used FakeItEasy nuget packages to mock the classes and its methods





Screenshots

Choose Match View



Play Match View

