## Architecture & Design Patterns

### MVC Pattern

As was asked, I used the MVC Pattern to separate both logic and view in two different namespaces. This makes the separation very clear between the graphical representation and logic of the game.

### Observer Pattern

I used the Observer Pattern to properly notify and update the corresponding Views and Score of an update happening in the logic part of the code. For the Entity Models, I just linked them to a Subject, which updates the right view observers. Besides this, I also have a score observer that waits until it gets notified by the player that the score should update. I’ve decided to only make player notify the ScoreObserver by just having functions in the Player class that get called whenever an event happens; like picking up bonusses or reusing platforms (it doesn’t store anything in the player itself).

Two Observer types: ViewObserver / ScoreObserver

Subject:

* + notifyEntityUpdate() (For Entity Views)
  + notifyHeight() (For Score)
  + notifyBonusCollected() (For Score)
  + notifyPlatformReuse() (For Score)

By doing this, it makes it very easy to extend our views and entities.

### Abstract Factory

The Project makes use of an abstract factory called EntityFactory in our Logic part, which then gets implemented by the ConcreteFactory in the View part.

In this abstract factory we have pure virtual methods for creating every single entity: Player, Bonus, Platform and BGTile. This makes our Logic part be able to create entities without it needing to know how to create SFML views. In addition to those methods we also have a pure virtual cleanupViews method that we use to cleanup views from our factory.

## Singleton

Both Stopwatch and Random use the singleton implementation, so that only one instance of both will be available during execution.

## Code Organization

### Library Separation

As was required by the specifications of the project, I split up the Logic library to be independent from the View part. This makes it so we can properly link our Logic files to our View files without needing to be dependent on the View part.

### Class Hierarchy and Structure

The project follows the exact class hierarchy and structure provided by the specification.

A diagram of a game

Description automatically generated

### Header/Source File Organization

First of all, logic and view are two separate directories with both having an ‘include’ folder and a ‘src’ folder. The include folder holds all the header files (.h), while the src folder holds all the source files (.cpp). With both having their own CMakeLists.txt files, it made it very easy to link the Logic to the View code later on.

## Memory Management

The project makes use of smart pointers like the requirement. It also properly handles resources by creating new entities while our camera goes upwards and removing them when they’re out of our view. We use the erase/remove\_if pattern together with !camera->isVisible(y) to remove them out of our world and we use an isOffScreen() function as our check in our EntityView class, parameters for this function will be camera->getY() + camera->getBufferzone().

## Game Design Decisions

I’ve pretty much followed all the requirements surrounding how the game design should be. One thing that is pretty important, is that our platforms get generated by checking if they’re far enough away from a platform close to where it wants to generate. There’s also a heightFactor that keeps getting higher, the higher the player goes. This factor makes it less and less likely to spawn platforms the higher you go. Besides this, there are some random values that make it have a higher/lower vertical spacing, spawn left/right, etc. Also bonusses will check that they’re not too close to others.

Other decisions like the scoring system, are just done based on what felt good. For bonusses springs make you jump higher while jetpacks makes you fly for a bit.

## User Interface

I’ve added a small main menu screen that lets you decide on a framerate and also shows you your highscore. This main menu will only be open at the start of the game. It gets the highscore by reading it from the highscore.txt file if it exists. This file will get written to, whenever someone exits the game. This way it will keep the highscore saved even when you close the game.

There’s also a score and game over display that show you your score and score/highscore.

## Extensibility

It’s easy to add new platform/bonus types by just adding it to the enum and just adding cases in both switch statements in their respective update() functions. For other implementations, it should be pretty straightforward with how the logic/view is separated.

## Challenges

I’ve already created a small Tetris game in the past with ‘almost’ the same design decisions. But this was a long time ago and there were some different things that I hadn’t used yet.

Mainly the use of shared pointers, etc. was the first time I actually used them in a project, so it was a bit of trial and error. Also having good stopwatch/random implementations took a bit of time. Mostly I didn’t know which clock/generator functions to use.

Almost 3 pages, sorry!