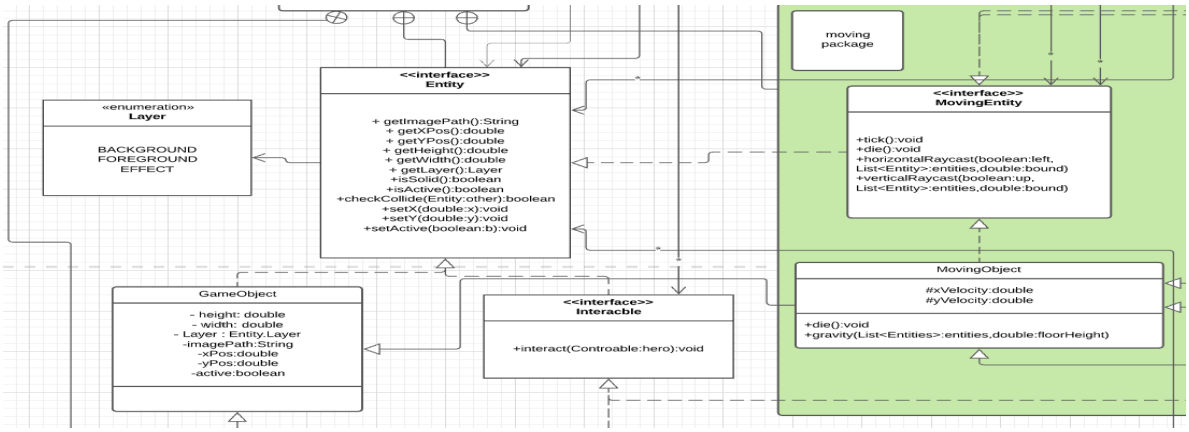
Code review

OOP Design

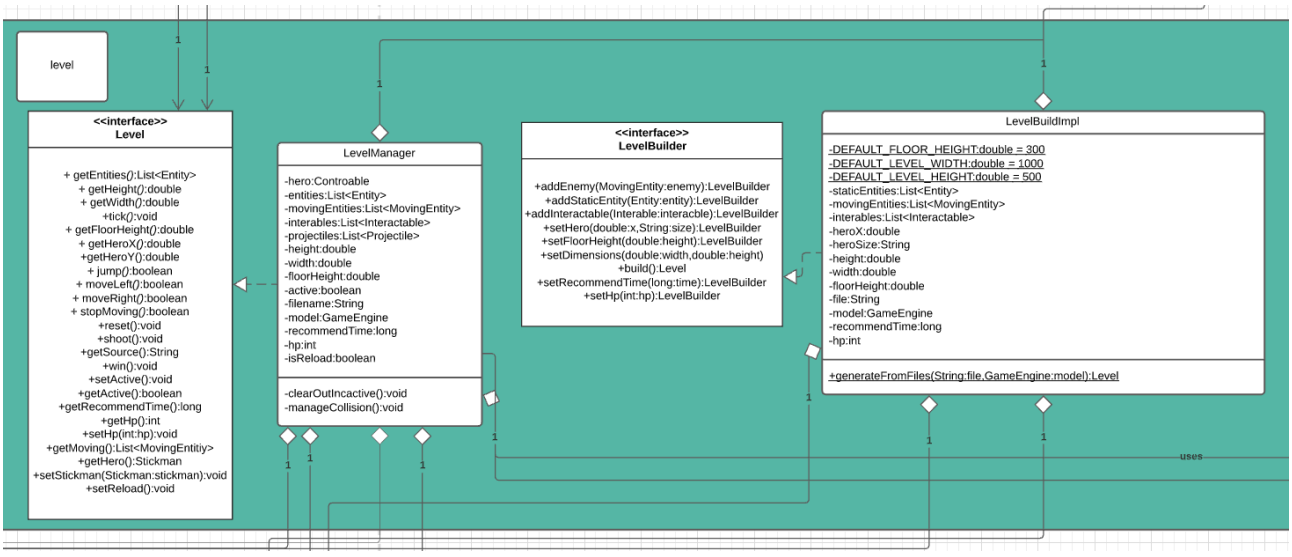
For the provided codebase, the programmer provide some really good oop design.



For the part of Entity, the author works hard to follow the single response principle, the best example is different entities which has different functions. Also for the different interface like Entity, MovingEntity, etc. these are all for the entity to take the single on respond.
For the usage of interface, the author also done good in open/closed principles. The interface are extends the first interface, for example, the MovingEntity interface extends the Entity, the Controllable extend the MovingEntity. Achieve the inheritance of the abstract, change the inherent behaviour by overriding its method, and implement the new extension method.
Also the parent Entity and children like StickMan, Slime also achieve the Liskov Substitution Principle. The Entity object can be casted to different specific Entity like mushroom to replace in particular situation.
Therefore, so much inheritance interface directly implement the Interface Segregation Principle. Which make the big interface become smaller and smaller, not put all methods which need implemented in single interface.
For Dependency Inversion Principle, the coding all around the interface and abstract class like GameObject which implements the interface, it's a good show of DIP.

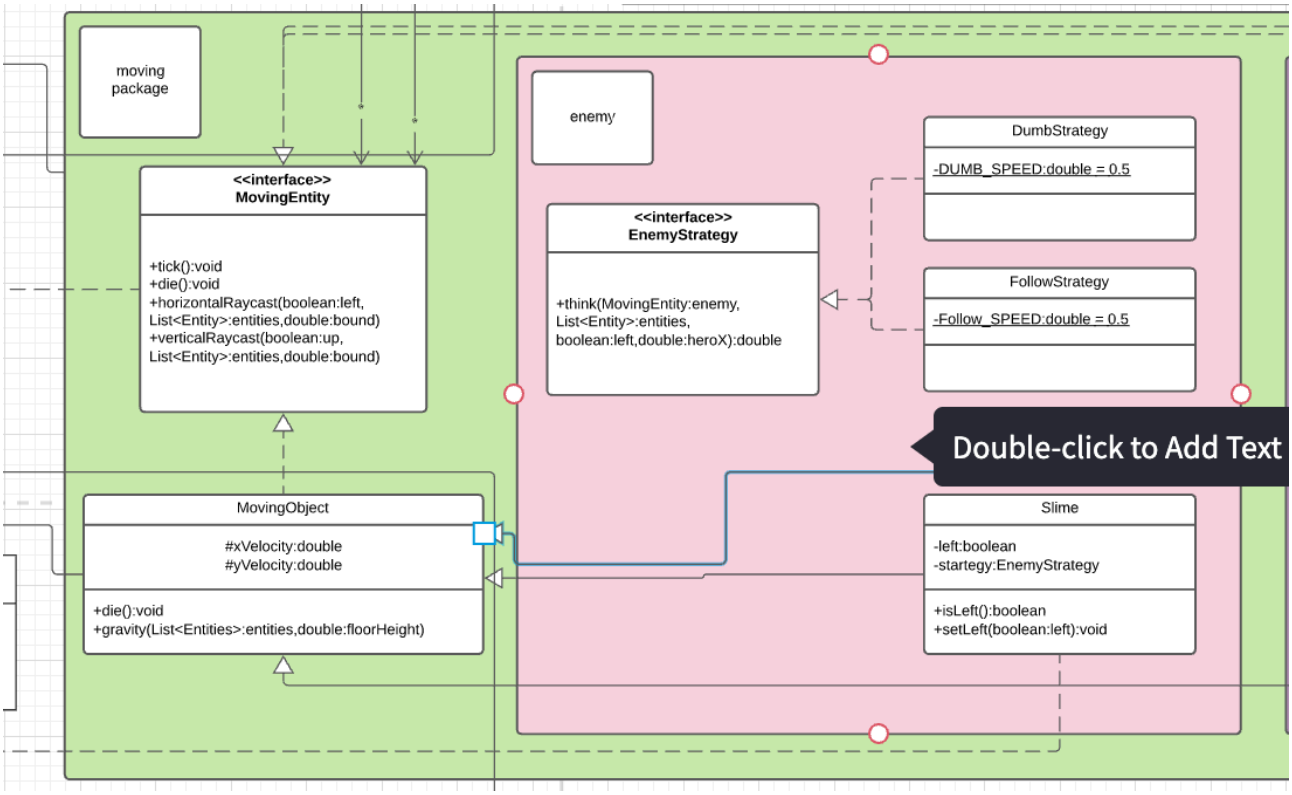
Design Patter

In the design pattern, I found two design patter which the author used.
First one is the Builder Pattern.



In the level part, the author uses the builder pattern as the LevelBuilder when build a new level. The build of a level is complex, so according to this pattern's purpose which is make complex build simple, the builder is suitable here. Also because the build target need many steps, so builder pattern here is better than factory method.

The LevelBuilder interface is the interface which contains the main methods which split the complex task of build a level. The return type is the same LevelBuilder, which set the matching attribute. The build method init will return the whole new Level.



The second pattern is the Strategy pattern. The reason why its suitable here is that: the slime should have different moving method, they cannot all moving in same logic. The purpose of the strategy is to package different calculate logic as one thing. Using the strategy prevent the recoding the same slime with just different moving logic. Makes the code more flexible and avoid waste of reuse code. But each strategy is hard to reuse if need many different strategies. So if the game grows larger there may be some problem.

Documentation

The README file is very useful and respect the copyrights, it shows the source of the picture on the internet. Also indicates the keys to run the game and control the human. The json format is great to let developer know what each parameter usage. But it didn't tells how to play the game like will killed by slime or eat mushroom to upgrade. That will make user bit hard to learn.

The comments are very useful, no lies that the comments really helped a lot when writing the extension. The format is great and let me try to learn that style to write comments too.

Difficulty on extension

The code is very good but still a bit harder to achieve extension. For the first requirement, just like the README file says, if want to change level, it needs to set the level manually in GameManager. Its kind like hard code and seems like no prepare for the future development.

For the second requirement, it's fine for me to coding because I use another way which will not influence the basic LevelManager.

For the last requirement, it's still fine for me to achieve the requirement, because the corporate with LevelBuilder and generateFromFile function very well.

Extension

What I done

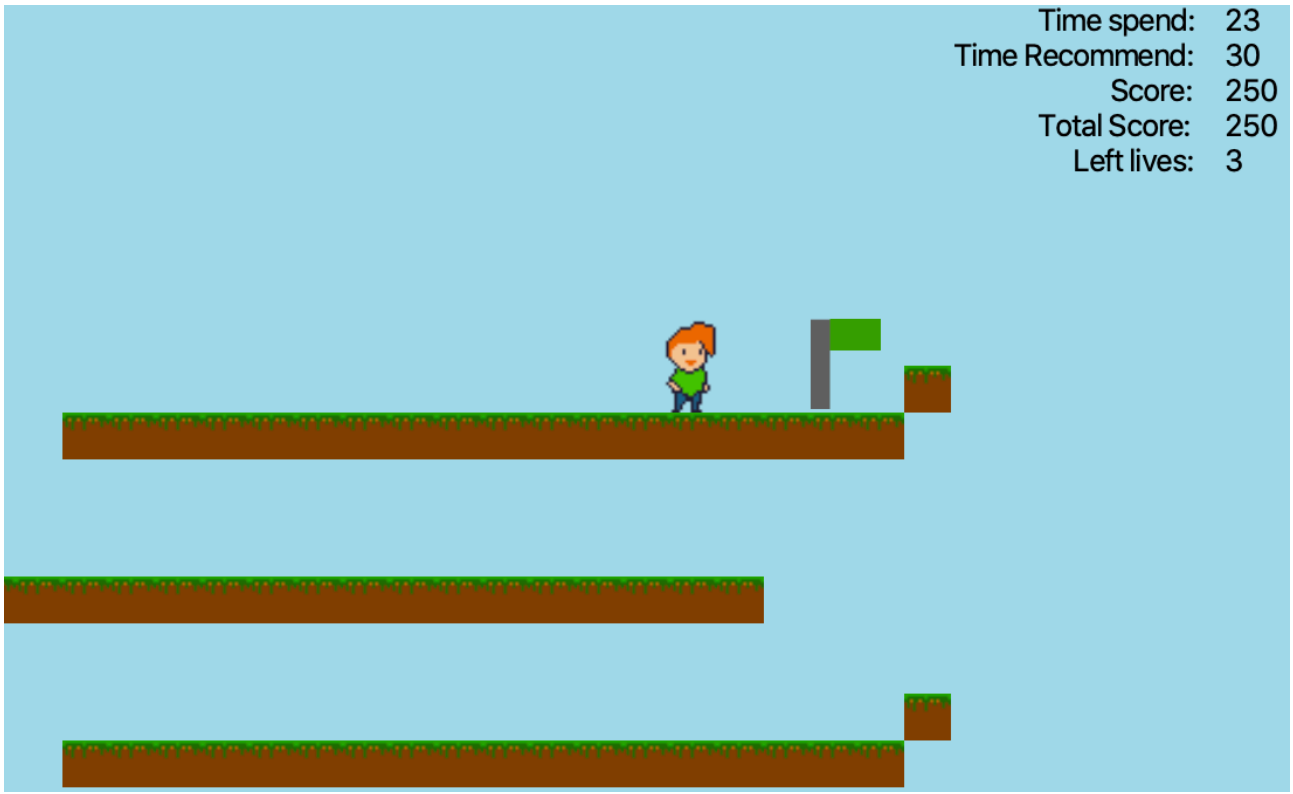
I've done the three requirements all of them. But include some of my opinions.

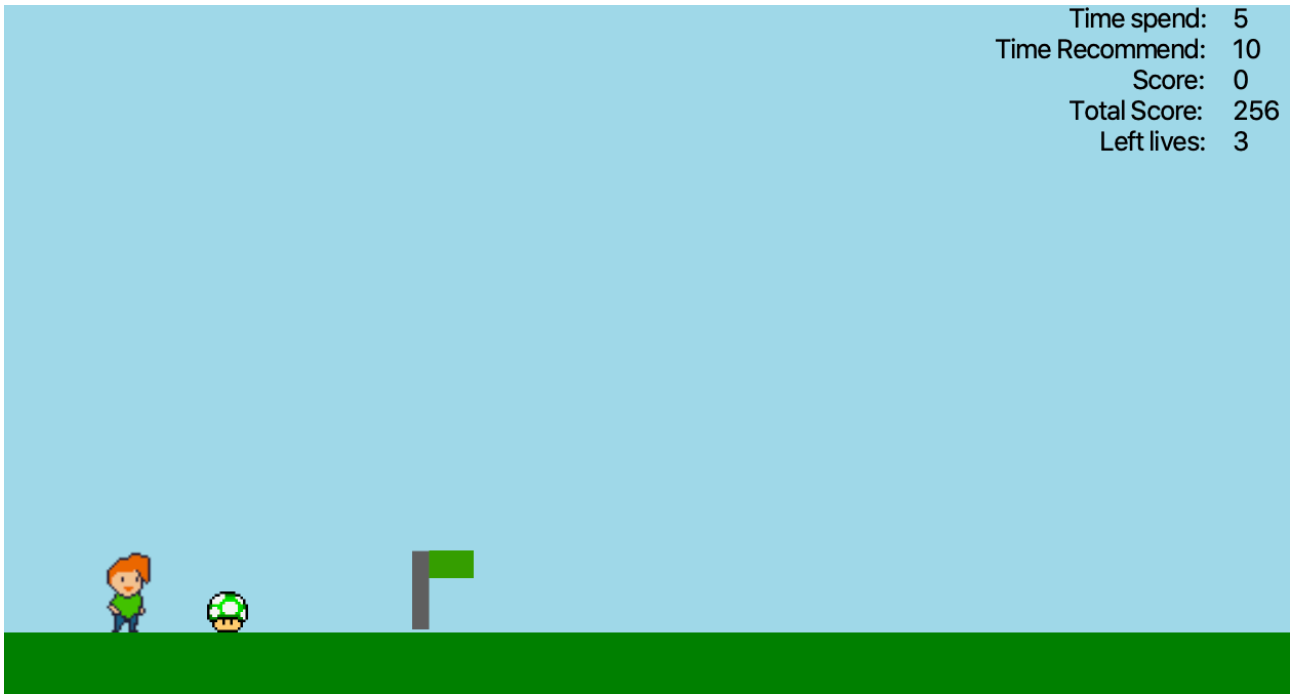
For the level transition, touch the flag will make you get into next level if this is not the final level. If it is, the win will show on the screen.

For the time and score:

The time part: the time which player pass the current level will be recorded. Just as the requirement, if the recommend time of the level is 20s, but the player just use 10s, the player will get extra 10 score when count total score. But if the user takes 30s, it will minus 10 score.

The score part: kill a slime will get 100, eat a mushroom will get 50, pass the level within or over time will get extra plus or minus score. In each level, the score is separate, which means if you get 250 in level 1, you will display 0 in level 2. But the 250 had been add to the total score(plus the pass level time score), the text under the current score.





For save and reload:

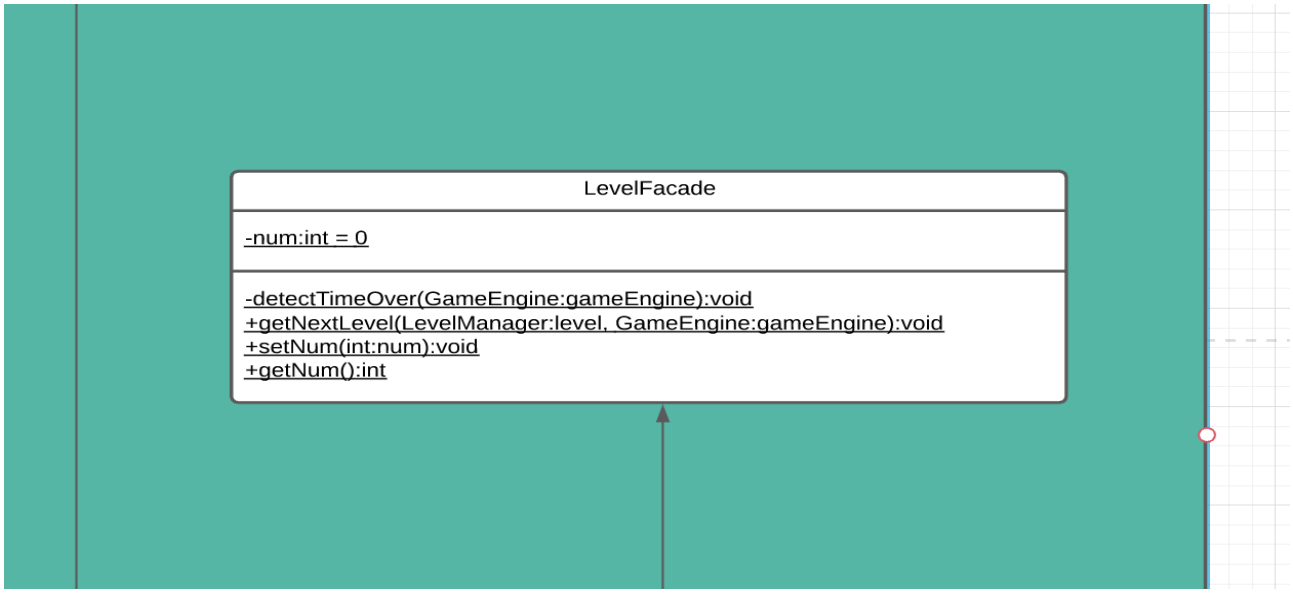
“S” for save, “L” for reload.

For the game equality, kill the same slime twice will not add the score. For example, when the level start, press save, then kill one slime, then reload, the slime you killed before will reload (because when you record the level it was there) but you will not get more marks from this slime, but you score which you kill this slime before will be recorded. Just the first time you kill that slime will be recorded.

Just like 'I Wanna', the reload is used for practice quicker pass the level and earn that part of score.

When the load pressed, the time, score, level, total score, recommend time, entities in the screen when you pressed save will be restored to the state which you press save.

OOP Design, Pattern, and Rationalise



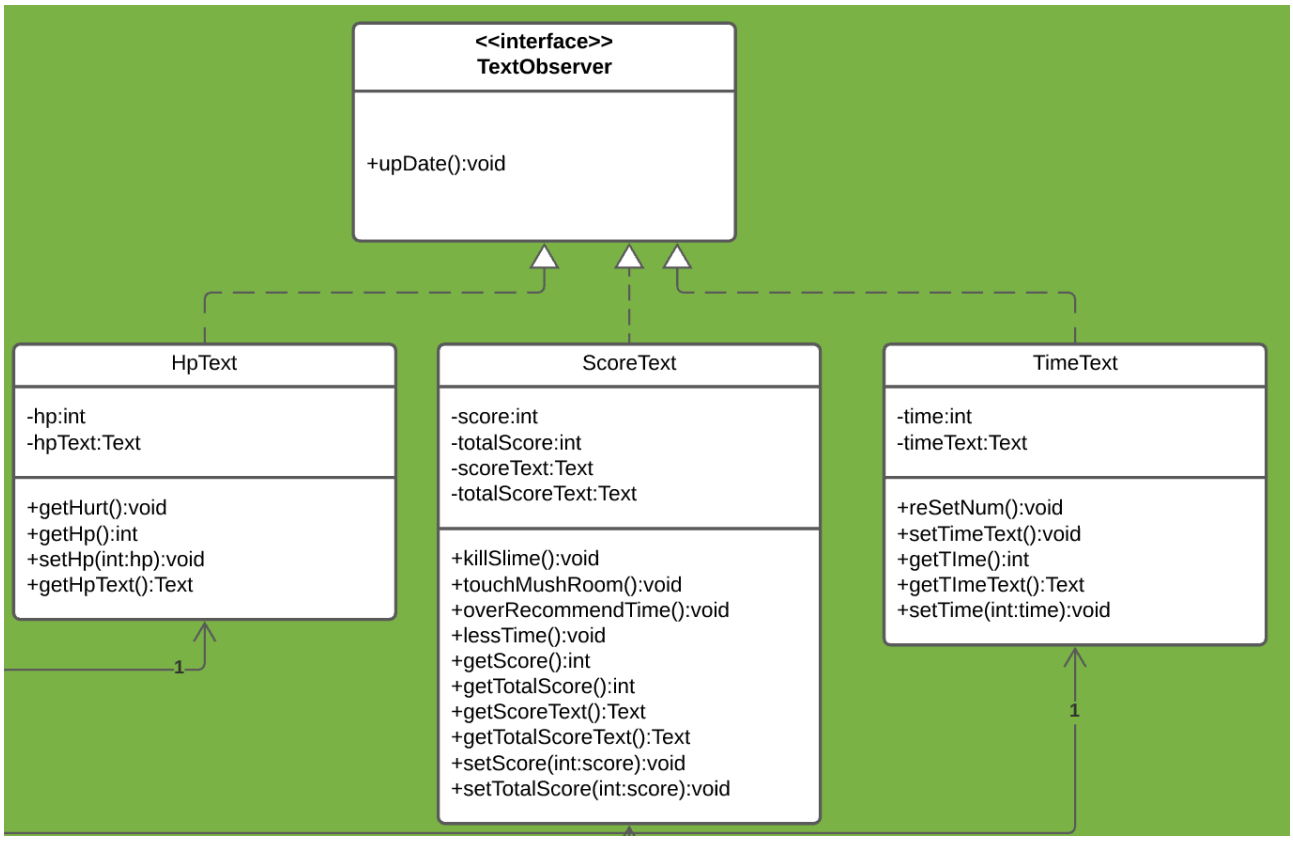
For the first requirement, I used the facade pattern to achieve the result.

Why I use this ?

First, I need a general controller to control the whole actions which need to use in level transition. Because the transition is a large part and need corporate with a lot things like time, score, etc. so to control the subsystems (which are time, score, text, lives, etc), a controller which follow the facade pattern is excellent. To satisfy the pattern, which is provide the only method for the client, the getNextLevel is used. When the hero touch the flag, this function will be called. And all decision logic, like mentioned above, if the final level, the time calculate, the score calculate, is all done here. These are all the subsystems need to corporate to get to the next level.

Second, it follow the Single Response Principles. The whole class is just for one thing, control the level change. Which will very suitable if I implement the first requirement in this way.

For the second requirement, the observer are used here.



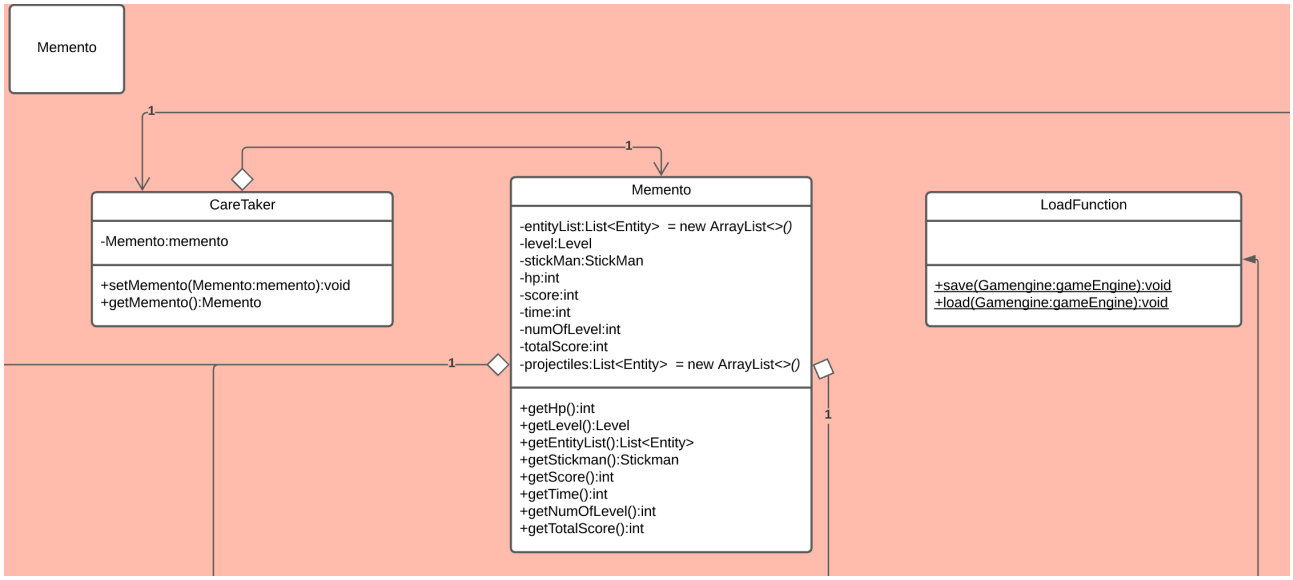
The three different object here are satisfying the Single Response Principle, each of them control the only thing, time, score or lives. They observe the change in time, score and lives, if they changed, the Text on the screen will changed rapidly too.

Why I use this?

Because of the codebase, also my way to inheritance the lives or score form last level, the perfect way is to use an extra object independent of the level. So to satisfy the requirement, three single response object were created.

These observers are recording, and control the display text when the things change.

For the third requirement, I used memento pattern.



Why I use this?

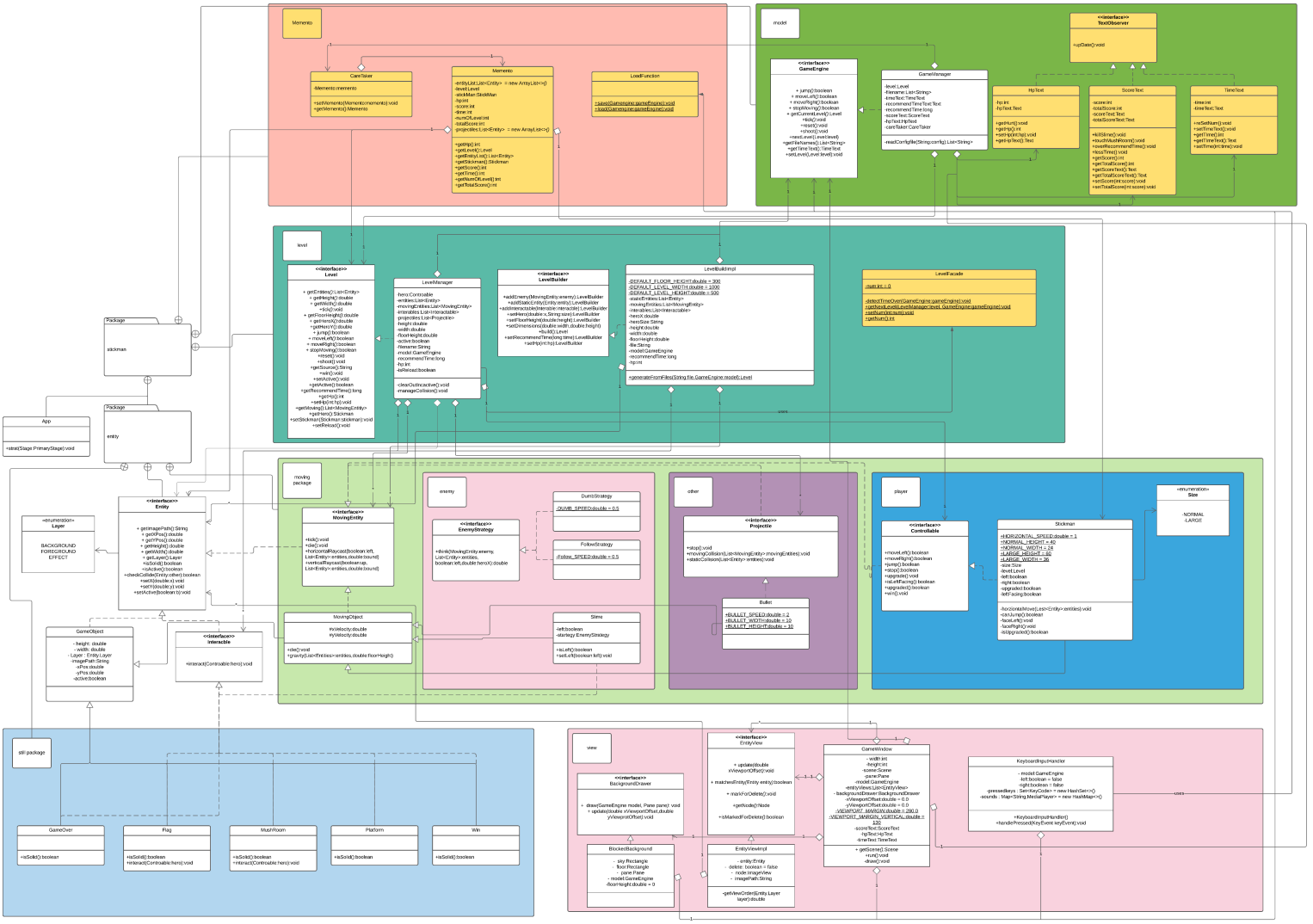
The best way to save the state is to use a memento, that's no doubt.

To not break the Single Response Principle, the LoadFunction class was introduced. The memento should just keep the state, the caretaker should just take care of memento, so to follow the SRP, I need a specific class to achieve to save and load the states.

The Load function is also a facade, which will provide static methods for key pressed, control all the subsystems to work and restore the game. It will read all the information of the memento and try to restore the same scene.

WHOLE UML

The yellow class blocks are highlight the pattern use



if you cannot see/read my UML diagram clearly, please refer to

https://lucid.app/lucidchart/e054b00b-b919-4874-9b4c-42c5ef7d6a6f/view?page=0_0#?folder_id=home&browser=icon