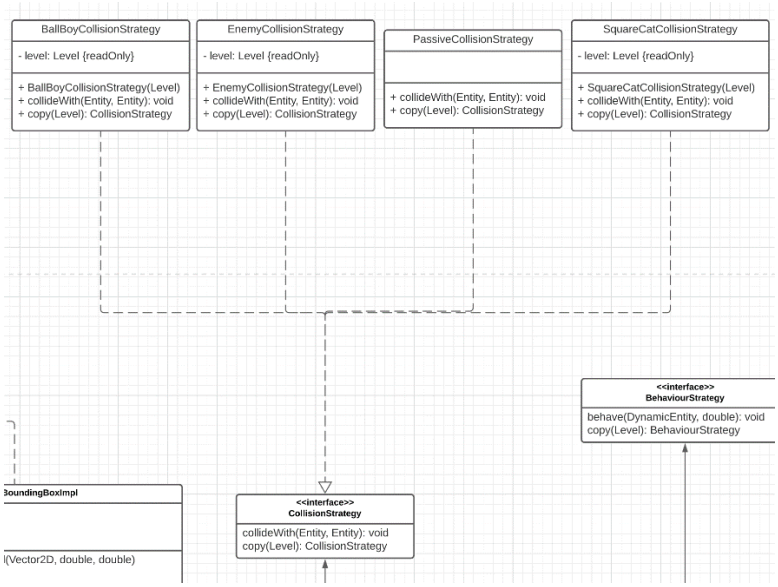
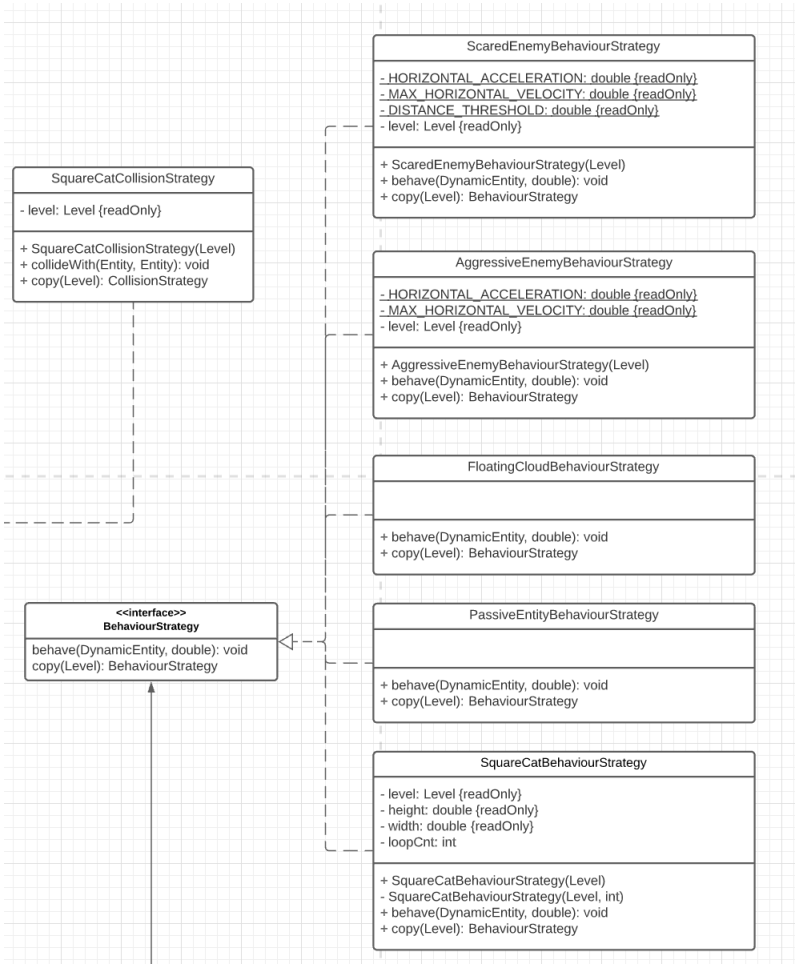
Report

1. Code review

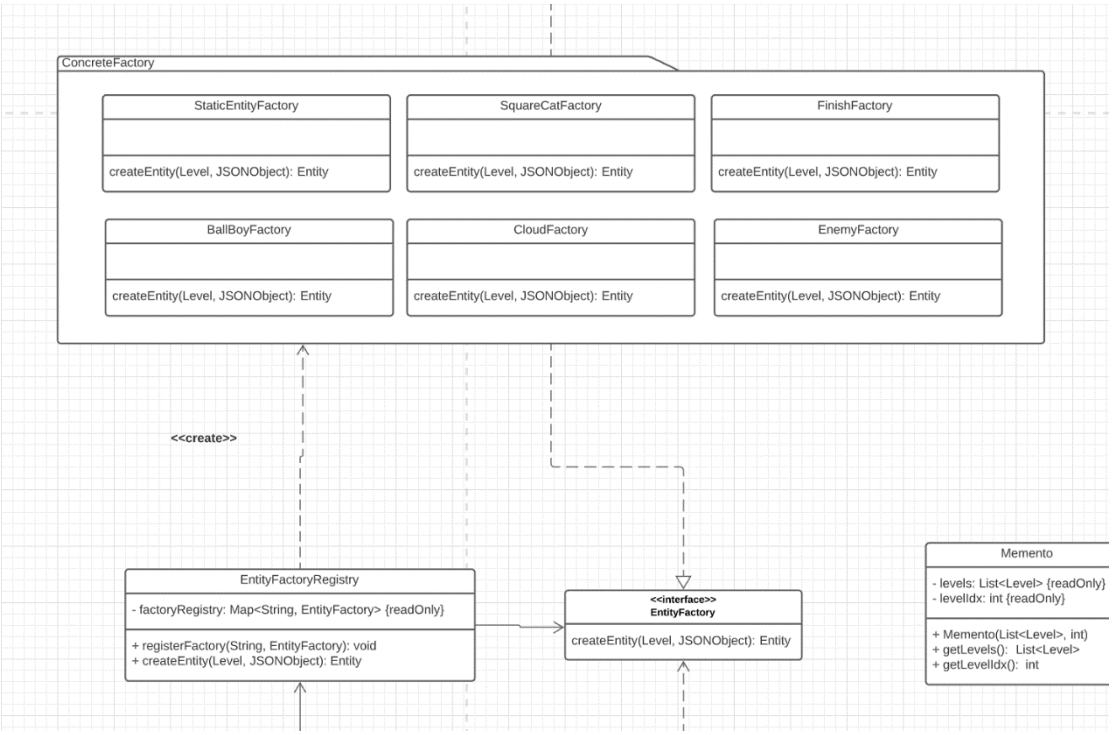
- Design patterns review
  a) Strategy patterns





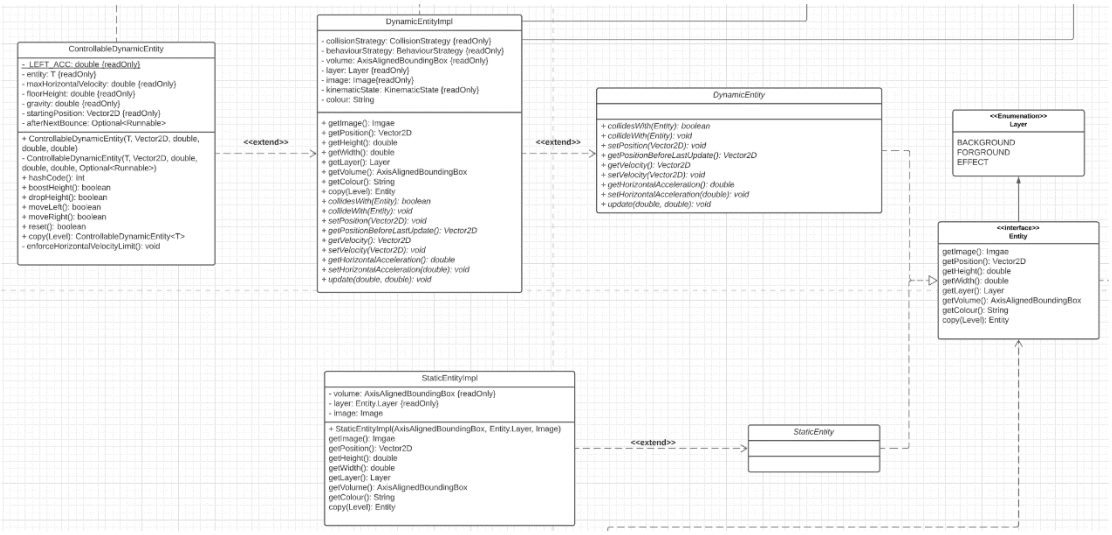
As can be seen from above two snippets, the original code base designs to use strategy pattern for the collision and behavior of entities. This design benefits making extensions by reducing coupling. This design obeys pure fabrication principles by assigning the abstract single responsibility (collied/behavior) to a concrete object which reduces the responsibility of entity class. This use of design pattern is also a good use of Polymorphism**.** However, there is a bad design which influences me adding save/load feature, which I will mention later.

b) Factory pattern



It designs to use factory pattern for creating different entities. This is also a good design because it makes adding new entity type easily by requiring only a new corresponding concrete factory. This splits the responsibility for creating entities from the app class, which reduces the coupling.

- Regarding to OO principle



This is a good design trying to satisfy OO principle. Firstly, it separates entity into two sub-classes, one is dynamic, and one is static, which is a good design. And using a abstract layer between concrete dynamic/static entity and entity interface is also a good design to reduce coupling.

- Documentation
  Documentation is splendid, there are comments for most of the methods, and the naming style and code style are perfectly understandable. However, it might be better to have more short comments between long codes.

- Difficulty encountered when adding new features
  a) When I was trying to apply memento pattern for the save&load feature by applying prototype feature for entity and level classes. The design of behavior&collision caused a problem. Because it can be seen some concrete behavior/collision have dependency on actual level object, which means the behavior/collision strategies should also be updated when the level is being updated. This dependency somehow increases the coupling and makes it hard to make extension relating to behavior/collision.
  b) When I was trying to extend the cat feature, I found the original code base does not support entity deletion. Therefore, I had to make some changes to the code. Specifically, add a remove function for level class to remove a specific entity from its entity list.
  c) Also is when extending the cat feature, I found this is some weird design for the collision strategy which could increase the difficulty making extension. Specifically, when an enemy collision strategy's collied function is called, how would it judge if it collied with a hero? It calls level's isHero method to check if the entity is hero. Then what should I do if I want to add some new entities that could do effects to enemies, like cat? Do I have to always add a method in level called isCat? It is weird because what if there is not only cat being added, but many other entity types that could interact with each other?
  So, I do believe it is better to design a way to distinguish different entities.

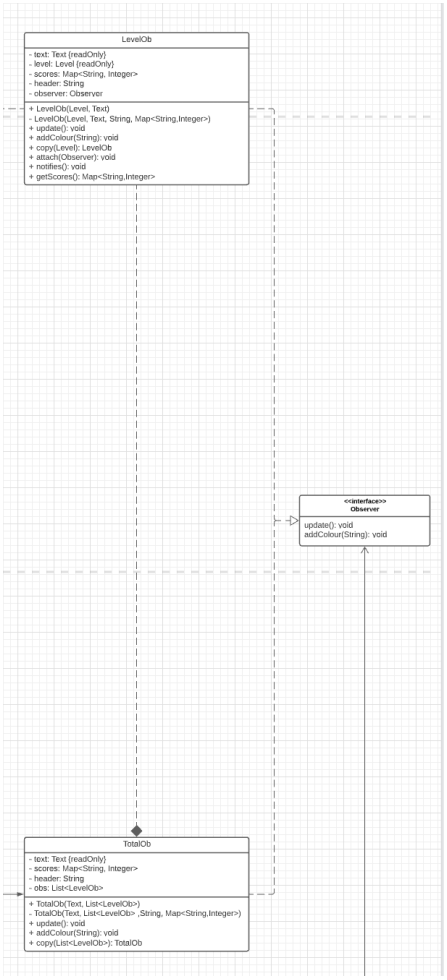
Currently what is in my mind is to create different sub-classes of dynamic entity.

2. Feature extension
   a) Actual extension made in the code

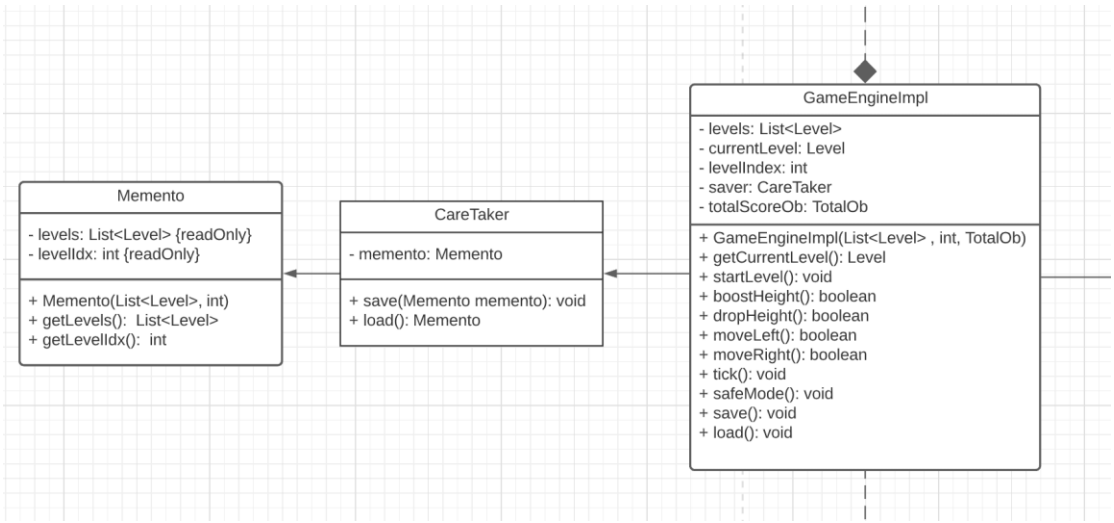
      To implement level transition, I made game engine to store a list of levels instead of only one level. And add a Boolean variable for level class named win which turns true when the level is finished. And the game engine would check if current level is won every tick, if so, level index would increase by one, if the index is over the numbers of levels, exit, otherwise it would process to next level.

      To implement the square cat, I create a new behavior and collision strategy for the cat, though it seems unnecessary at this moment. And I also add a new factory for the cat and add them into the factory registry inside the App's start method. And I add a cat variable to level class, I also changed enemy's collision strategy to make it dies when it collied with the cat.

      To implement the score feature, I made one observer interface and two observe classes implementing this interface. The total score is initialized in the App's start method and being used by the game engines. The level score is also initialized in the App's start method and will be observing one level each. Additionally, the total score observer observes all the level score observers.



To implement the save&load feature, I use memento and prototype pattern. A caretaker class and a memento class has been added and stored in the game engine. The memento class would save a list of a deep copy of the list of levels with the state when they are being saved. The deep copy is made with the prototype pattern where the levels copy themselves and coping all the included entities.



However, there is a design that I am not sure if it is good. Because many variables are private and the existing constructor cannot satisfy to make an entire same copy, therefore I create a private different constructor for many classes specifically in order to make their own clones.

b) Design pattern discussion

The observer pattern can reduce the coupling by assigning the observe responsibility to the observer classes. It is good for extension, for example if we want more features to be observed like hero's health, we only need to update the observers' saved state to make it update more components.

The memento pattern is also useful because if new features have added and should also be saved, only memento needs to be updated by storing the new features. It also satisfies the open-closed principle, because no changes should be made when extension are made. And for maintenance it is also beneficial because the coders only need to manage the memento without knowing how other classes work.

The prototype pattern is also good for extension and reducing coupling. Because all the extended entities only need to implement their copy function to supply save&load feature. And it is much easier when creating a new object because of the complicate data preparation and also the authority constraints.

c) Regarding to SOLID and GRASP principles

The observer pattern satisfies low coupling and high cohesion and polymorphism because the sub-classes of observer: LevelOB and TotalOB both implements the update method of observer interface though they observer different objects. It also prevents **protected variations** because nothing needs to be changed as long as the level's score storing is not changed. Using TotalOB to observer all the LevelOB rather than observing all the levels is using **indirection** to reduce coupling for better code reuse. As for SOLIP principles, it satisfies open-close principle and single-responsibility principle. The only responsibility is to observe its observed object. And it is open for extension and closed for modification.

The memento pattern also satisfies **open-close principle** and **single-responsibility principle**. The only responsibility is to store a list of levels. And it is open for extension and closed for modification. It enhances low coupling and high cohesion. And it allows controller to easier control the saved state by accessing the caretaker class. Caretaker class is also a good use of indirection by playing the middle role between memento and game engine to reduce coupling.

The prototype pattern is a good practice of **polymorphism** because all the

prototypes are created using the copy method required in the interface, however they are implemented differently. And to create a prototype, simply calling the copy method would work, it does not need to know how it is achieved.

d) Reflect on your extension design, highlighting any outstanding issues or improvements or discussing your impact on the extensibility of the code

For the level transition, I believe my design is perfect as the original features are incomplete because it would quit when the first level finish and the game engine only support one level. Now and numbers of level could be added, and the game engine could manage a list of levels and decide to load any one of them.

For the square cat implementation, the newly added behavior and collision strategy and factory have no problem because I only followed the old design. However, adding the cat into the level class and adding a isCat function into the level class is increasing the coupling and influencing the extensibility, though I believe this is due to the old design in this aspect.

For the observer design, it is not implemented well, because as they have dependency to the level objects, and they are initialized in the app class. I have to also make a copy of them to implement the save&load feature and I was choosing not to add the copy method into the observer interface. Therefore, I chose to use the class specifier instead of the interface, which definitely influences the extensibility and increases coupling. To improve this, I have two methods in mind, one is to move the initialization of observer from app to elsewhere, another is to include the copy method into the observer interface, however I am not sure if the latter method violates the observer pattern.

For the memento pattern, I believe it is independent enough from other classes, which means it does not influence extensibility. Because how the memento is saved or loaded could be easily updated in the caretaker class, and what is saved could be easily updated in the memento class, which means they are no influence unless a feature regarding to save&load feature is implemented. And even if the feature is implemented, it is not hard to update the caretaker and memento class.