Design patterns

In our project, we expect to use several design patterns, in order to produce cleaner and better code:

- **MVC (Model-view-controller) design pattern:**

This design pattern allows us to separate concerns, improving the code readability and making it easier to implement new features in the future. To implement this pattern, we separated our entities' models from our entities' views.

The model will contain all the data related to the entity, such as: position, physics' body, type, animation's state time, etc.; and will be responsible for managing this data.

The view will contain all sprites/animations related to the entity and will be responsible for choosing the correct frame to display, based on the state of the entity.

The controller (*GameController)* will act as a "bridge" between the model and the view. It will be responsible for updating the data of each model as the game progresses.

By following the MVC pattern, we can easily change a model's view as many times as needed, without having to refactor the code over and over.

- **Singleton design pattern:**

This design pattern restricts the instantiation of a class to only one object. It's extremely useful, especially when it comes to controllers (since we only want one instance and we need them to be accessible from any part of the code). For now, we expect to have three singleton classes: *GameController*, *InputController* and *GameLevel*.

The *GameController* will control the physics of the game, updating the models accordingly; the *InputController* will catch and handle any input from the user and, finally, the *GameLevel* will represent the game's map (only one per gameplay) where all the action is going to happen.

- **Object pool design pattern:**

In order to improve efficiency and reduce the instantiation of unnecessary view objects, we created the class *ViewFactory*. This class contains a cache (using an *HashMap*) and implements object pooling. Whenever a certain model requires a view, the cache is searched for an existent view. If found, the cache returns it; if not, then it instantiates a new view object, adds it to the pool (for later use) and, finally, returns it.

This way, we can reutilize existent views for different models of the same type.

- **Decorator and strategy design patterns:**

We also expect to implement these two design patterns in the entities' logic, in order to easily alter and define different behaviors for different entities.