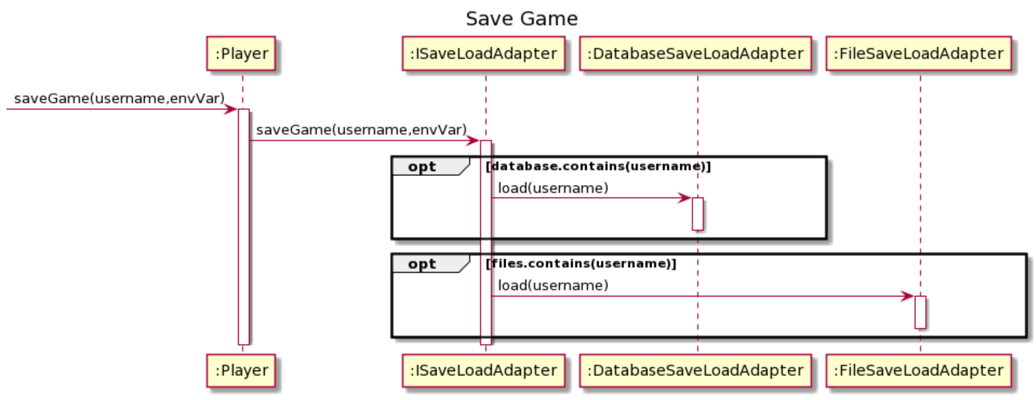
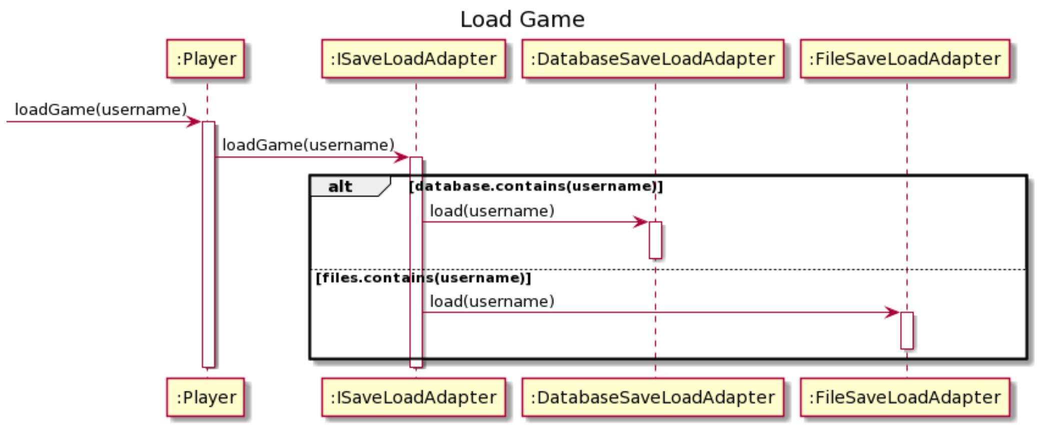
Table of Contents

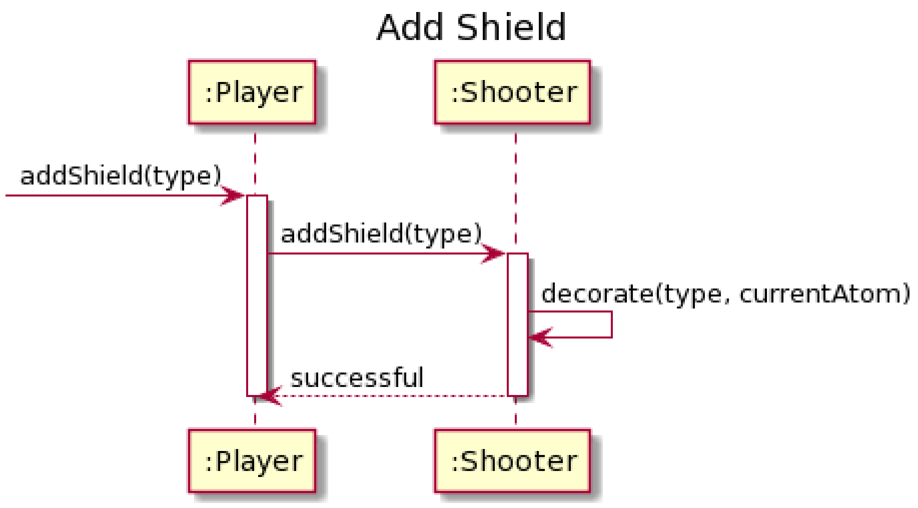
Interaction Diagrams

Group: Fark Etmez

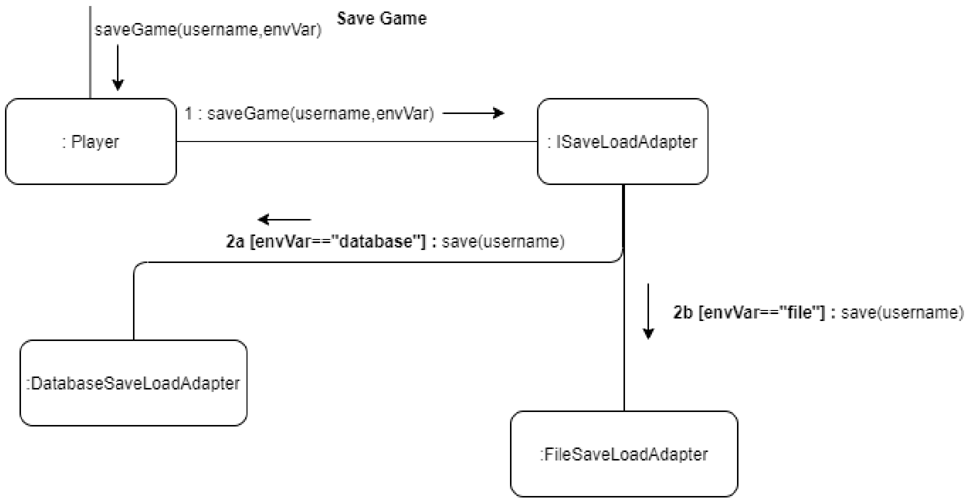
Sequence Diagrams:

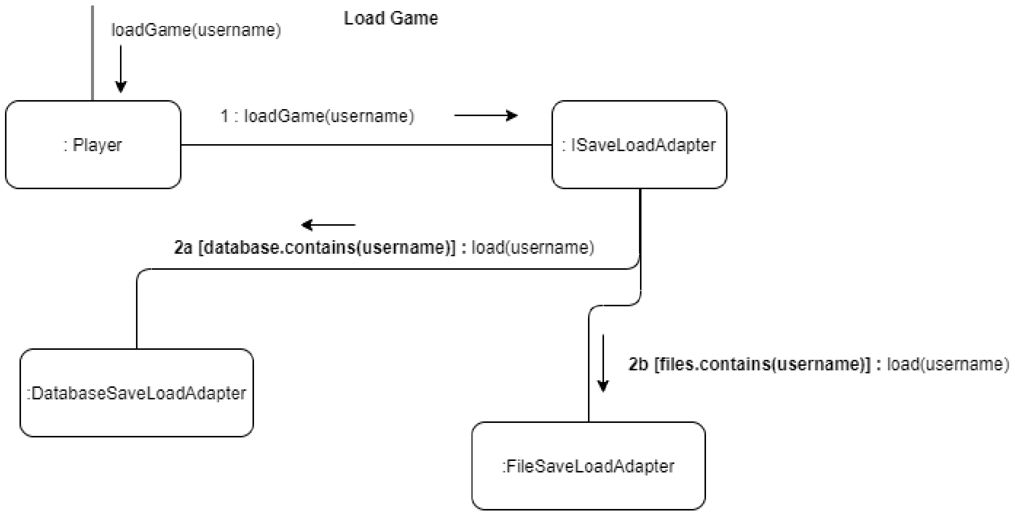


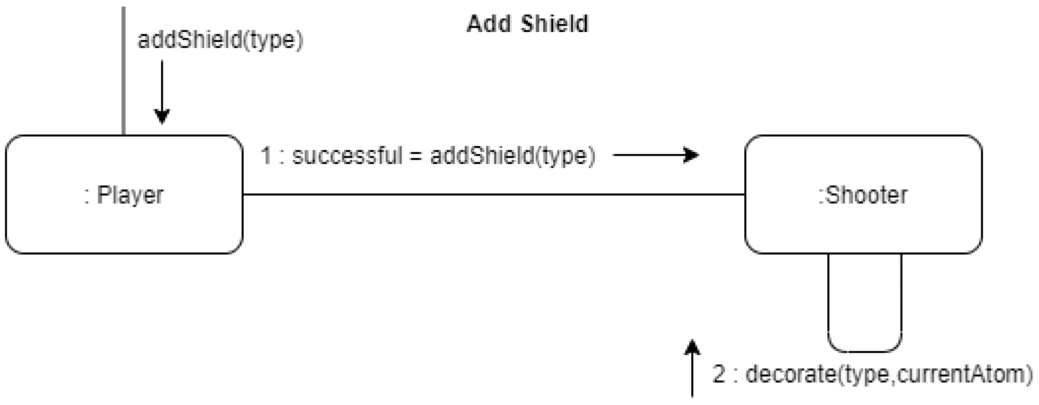




Communication Diagrams:

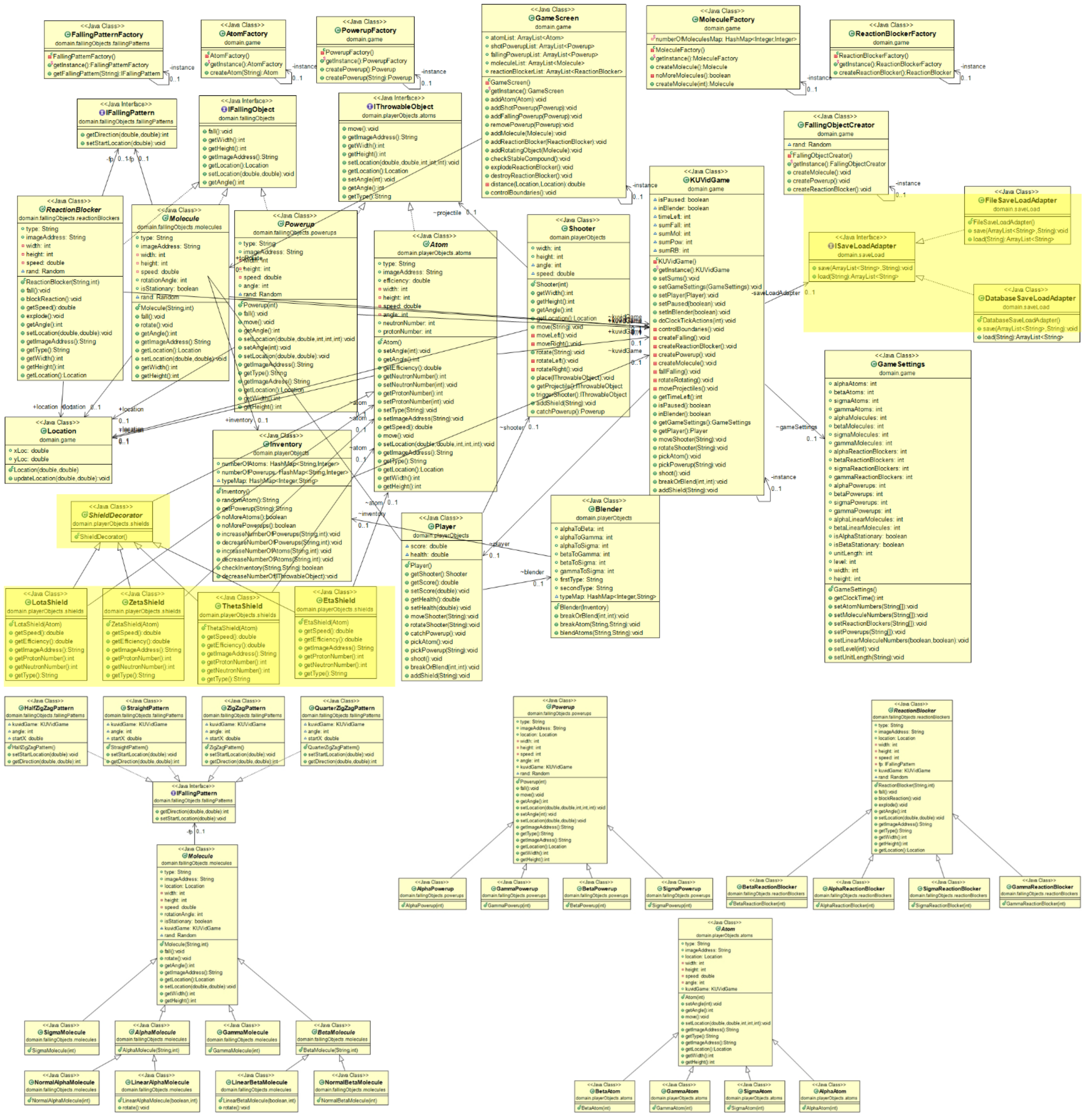

Save Game
saveGame(username,envVar)
: Player
1 : saveGame(username,envVar)
: ISaveLoadAdapter
2a [envVar=="database"] : save(username)
2b [envVar=="file"] : save(username)
:DatabaseSaveLoadAdapter
:FileSaveLoadAdapter




Add Shield
addShield(type)
: Player
1 : successful = addShield(type)
:Shooter
2 : decorate(type,currentAtom)

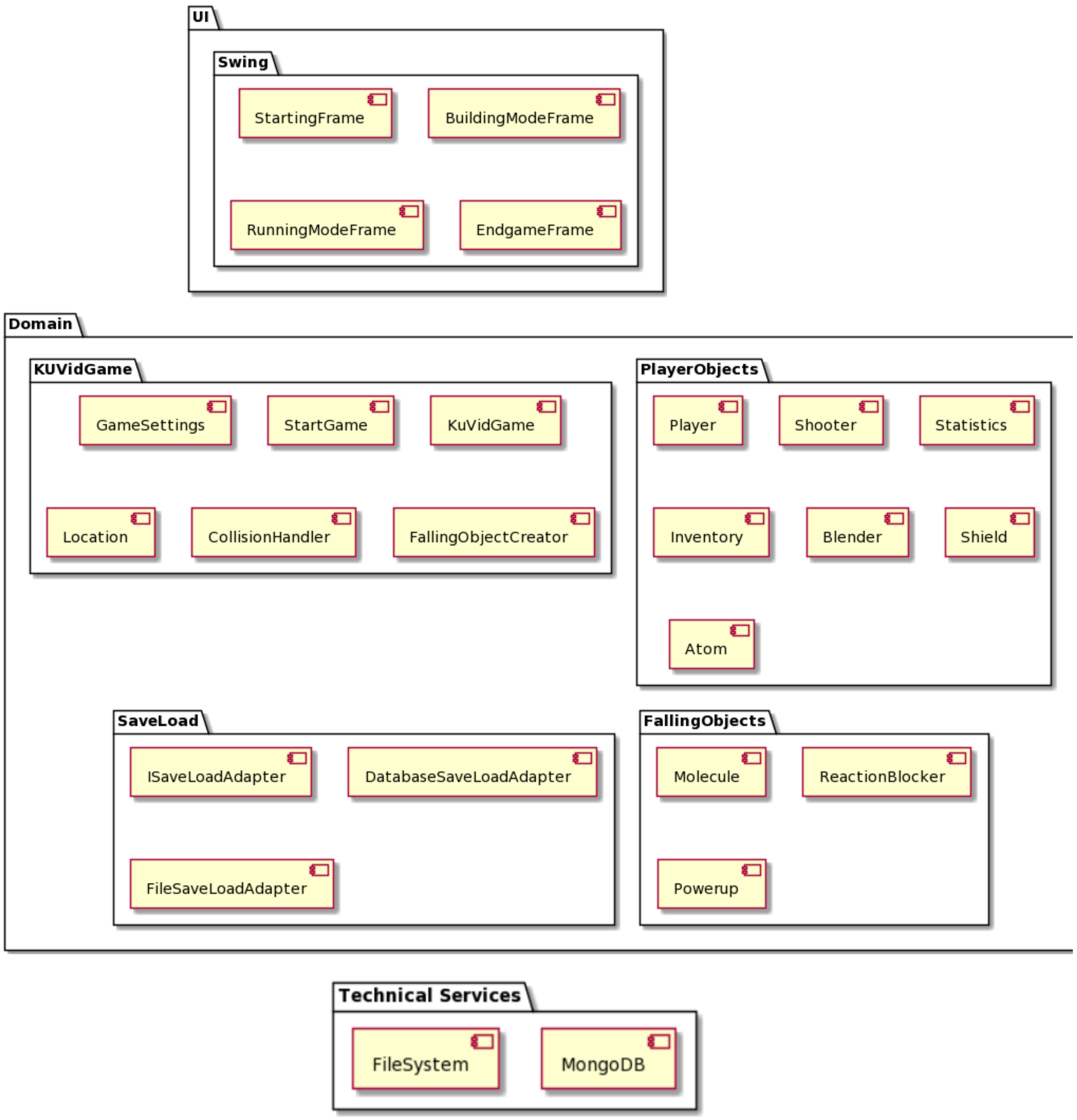
UML Class Diagram

Group: Fark Etmez



Logical Architecture

Group: Fark Etmez



Pattern Discussion

Group: Fark Etmez

Decorator: We will use the decorator pattern for shields, we decorate atoms with shields as suggested in the instructions. This pattern is our way of applying the Open-Closed Principle, since we can extend the atom class/instance, without any modification to the atom itself. While decorating the atoms, we decorate with the desired shield, which affects the speed and/or efficiency of an atom.

Adapter: Applying the Adapter pattern helps us implement the save game functionality according to the environment variable. The option to save either to a file or a database requires an implementation of environment variables. However, the method's signatures, postconditions, and preconditions can be the same for both thanks to the Adapter pattern.

Singleton: Singleton classes are really helpful for classes we need to access from a number of different classes, since without the use of this pattern, we would need to pass the instance of those classes as parameters each time we needed them. Therefore we have the KUVidGame class as a singleton. Also, the pattern is frequently used with the Factory pattern. All our Factory classes also use the Singleton pattern, so that we don't need to pass around the same instances as discussed before.

Factory: We use the Factory pattern to create atoms, molecules, powerups, and reaction blockers. It helps us to create game object instances in an efficient way.

Controller: In our design, we will be using the controller pattern to receive user inputs from the UI layer. The pattern allows us to delegate the inputs from the UI to the various classes in the domain layer without violating Model-View Separation. It represents the overall system. It is the main pattern that allows UI layer access to the domain layer, and therefore becomes the best option for controlling user inputs. So, we choose our Controller to be the KUVid Game class.

Creator: Our project requires instances of many types of objects to be constantly created, including Molecules, Powerups, Reaction Blockers. For this purpose, we will be implementing a FallingObjectCreator which will follow the GRASP pattern Creator. We decided to use a Creator pattern because FallingObjectCreator will have the initializing data for our falling objects when they are created. Thus making FallingObjectCreator an Expert with respect to creating falling objects. The use of a Creator pattern also supports Low Coupling.

Information Expert: In our design, to further reduce coupling, we will be using the Inventory class as an Information Expert. The Inventory class will be an expert in the sense that it will be the only class in our design that knows the amounts of the various objects that the player of the game possesses, e.g. number of atoms, number of powerups, number of atoms. For instance, the Player class will not have access to the number of atoms in their inventory directly. Another use of the Expert pattern will be the Blender class, which will have the information (rules) determining the relationship between various types of atoms while breaking and blending them. The distribution of responsibility between different classes also increases cohesion, while also making the code easier to follow. On the other hand, the pattern may result in designs that are problematic regarding cohesion, as can be seen in some of our sequence diagrams where the Player class calls the Shooter class and then calls the Inventory class with the return value.

High Cohesion: The best depiction of High Cohesion in our design is most likely the KUVidGame object, which instead of knowing what to do for all the possible actions for the game, delegates most of the commands, e.g. shoot, pick blender, etc., to different classes that are specifically designed to handle those methods; which enables us to maintain and reuse our code more frequently.
Low Coupling: The Player class does not shoot the atom directly, instead it triggers the Shooter to shoot what is currently loaded on it. If the Player class directly shoots an atom, it would be a high coupling. A similar example can also be given for **Indirection**, as rather than having the Player directly know/manipulate the number of items in the inventory, we have the inventory to do these tasks. Also, it enables reusing and understanding the code better without looking at other classes (just by looking at said class).

Polymorphism: Due to the nature of the project, where we have types (alpha, beta, sigma, gamma) of the same type of game objects (atom, molecule, powerup). The best way to handle this is by making use of the Polymorphism pattern. The main benefit of polymorphism is defining common elements and methods of objects in an interface or abstract class, therefore allowing us to write the bulk of the code once.

Strategy: In our design, the Strategy pattern is utilized in handling various falling patterns of some objects like Molecule, Powerup, ReactionBlocker. There are different strategies for falling objects. Also, the falling speed of some objects may change according to the difficulty of the game. While it may appear to make the design of the code more complicated, the Strategy pattern allows any change that may occur in any current or later stage of development to be carried out in a much easier fashion.

Observer: The Observer pattern is a must use pattern in most UI dependent projects, since it is the only relaxation of the most important principle, Model-View Separation. In our design, the use of the Observer pattern will be to update locations of the various game objects on the UI side, *according to the locations* which will be defined on the domain layer.