Design Patterns

1. Creational Design Patterns

- Creational Design Patterns are concerned with the way in which objects are created. They provide guidance on how to create objects (using keyword "new") when their creation requires making decisions.

1.1 Singleton

- **Intent:**
- Ensures that only one instance of a class is created.
- Provides a single global point of access to the object.

- This design pattern is used in our program in the following classes:
  - **Game Action:** which represents the game itself (Fruit Ninja) with all the actions needed to control the game. Since the game is created only once, Singleton is used to create only one instance of it.

  - **Factory Provider:** which represents the factory of factories that is responsible for dealing with Fruit Factory and Bomb Factory, and creating objects through these factories. Therefore, instead of dealing with different factories, client can deal only with the Factory Provider. Hence, only one instance is needed.

1.2 Factory

- **Intent:**
- Creates objects without exposing the instantiation logic to the client.
- Refers to the newly created object through a common interface. In other words, defines an interface for creating an object, but let subclasses decide which class to instantiate.

- This design pattern is used in our program in the following classes:
  - **Factory Provider:** which represents the factory of factories that is responsible for creating objects of type Fruits through Fruit Factory or of type Bomb through Bomb Factory.

- **Fruit Factory:** which represents the factory used for creating objects of type Fruit: Apple, Orange, Watermelon, Double Score Banana or Magic Bean (according to the type been given as a parameter on calling the function). Therefore, the client is not exposed to objects' instantiation.

- **Bomb Factory:** which represents the factory used for creating objects of type Bomb: Dangerous Bomb or Fatal Bomb (according to the type been given as a parameter on calling the function). Therefore, the client is not exposed to objects' instantiation.

2. Behavioral Design Patterns

- Identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

2.1 Memento

- **Intent:**
- Capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

- **Originator:** Game Action **Memento:** Game Memento

- This design pattern is used in our program to keep the state of the game. Since the player can save and load, so we have to keep the state of the game concerning: Difficulty Level, Current Score, High Score, Remaining Lives, Game Objects with the details of each object (Type, X-Location, Y-Location, Maximum Height, Initial Velocity and Final Velocity).

2.2 Command

- **Intent:**
- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- **Invoker:** Game Engine **Receiver:** Game Action
- This design pattern is used in our program to execute the **commands** that should be done upon slicing a game object:
    - **Increment Command:** used to increase the score by 1 on slicing a Fruit (Apple, Orange or Watermelon).
    - **Decrement Command:** used to decrease the lives by 1 in case of slicing a Dangerous Bomb or in case that a Fruit (Apple, Orange or Watermelon) has moved off the screen without being sliced.
    - **End Life Command:** which is executed upon slicing a Fatal Bomb that makes the player lose all his lives and shows him "Gam Over".
    - **Bonus Command:** which is executed upon slicing a Bonus Fruit (Magic Bean) that increases the score by 25 in case that the player has 3 lives, otherwise (if the player has less than 3 lives), this command increases the player's lives by 1.

2.3 Observer

- **Intent:**
- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- This design pattern is used in our program in the following classes:
    - **Game Engine:** which represents the **Observable** that can add observers that will notify its observers upon any updates concerning (score and remaining lives).
    - **Game Play Window:** which represents one of the **observers** because it shows the score and the remaining lives, so it has to be notified upon any updates.
    - **Score Window:** which represents one of the **observers** because it shows the final score and the high score, so it has to be notified upon any updates.