Exercise 2: DESIGN PATTERS (30 pts)

This exercise will be split into 2 parts, according to the 2 chosen design patters.

PART 1: Startegy pattern (15 pts)

In our code we have a class called “GameObjects”, which stores the objects which control the objects in our view. For example a “Bubble” object can be added to the “GameObjects”. A “Bubble” will control the movement of the circle (which represents the bubble) in the view and also detect collision. The objects which are stored in “GameObjects” use the strategy pattern to update their position and check for collision.
Once every frame (60 times per second) the “GameLoop” will call the update method of “GameObjects”. When this method is called, “GameObjects” will notify all objects which it is currently storing to update their positions.

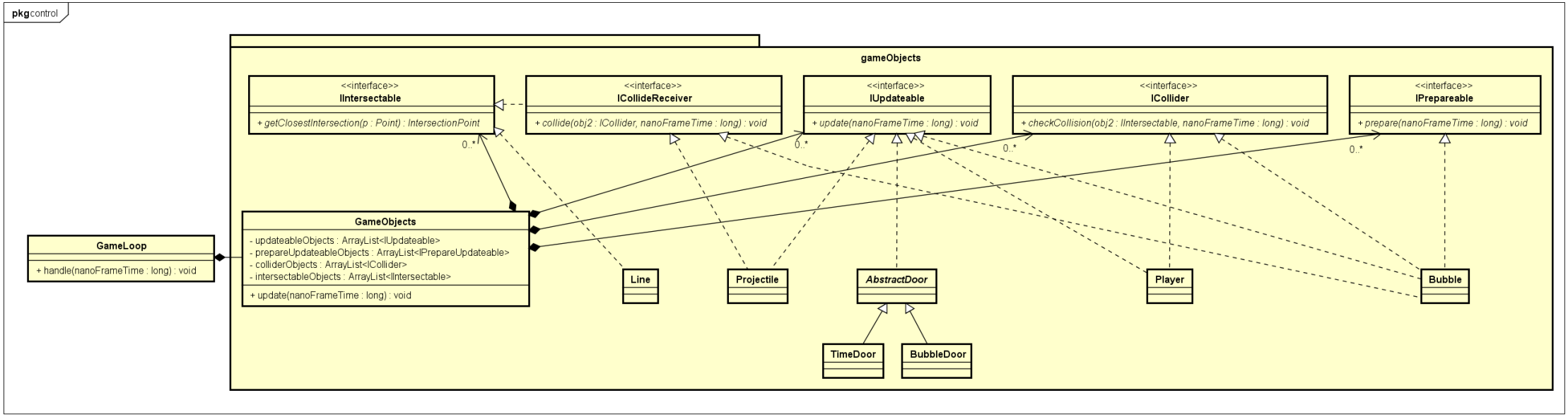
1. Write a natural language description of why and how the pattern is implemented in your code (5 pts)

As described above, the strategy pattern is used to update the position of objects in the view and to check for collision. The strategy pattern is very useful for this because:

- There are many different kinds of objects in the game which have to be updated.
- All objects got completely different update and collision checking algorithms. Collision with a line is for example calculated in a different way than collision with a circle.
- There are cases where not all functionality is needed; for example: static objects (walls) which don’t have to be updated, because they don’t move, but they do have physics: other objects can collide with them.
- Using a lot of interfaces makes the code very flexible. The kinds of objects in our game can vary greatly, to name a few: bubbles, walls, the player, pickups, etc. All of those objects need different behavior and it is unknown what kind of behavior will be needed in the future. To avoid the need to modify the already existing code to add new functionality, the strategy pattern is really useful.

We have implemented the strategy pattern by creating multiple interfaces (for updating and collision detection) and implementing only the necessary interfaces for each object.
In our “GameObjects” object we keep track of a separate list of objects for each interface. The “GameObjects” object will only have to call the methods of the interfaces in those lists and doesn’t have to know anything about the actual implementation of the object.
For all of the used interfaces and an explanation of those interfaces see exercise 2.

2. Make a class diagram of how the pattern is structured statically in your code (5pts)



Classes, methods, variables and relations between classes which don't contribute to the strategy pattern are omitted to avoid cluttering the diagram.

- **IIntersectable**

The IIntersectable interface is implemented when an object can be 'intersected', meaning it can be hit by another object. It contains a single method, which takes a point as input and returns the closest intersection point to that point. An intersection point contains the (x, y) coordinates of the intersection point and a vector which is a normal of the object at point (x, y).

- **ICollideReceiver**

The ICollideReceiver interface is implemented when an object has to be notified when another object has collided with it. For example: "Bubble A" calculated that it collides with "Bubble B", now "Bubble A" will call the method of the ICollideReceiver interface to notify "Bubble B" that it has been hit. This is mainly used to increase performance (if A collides with B, B won't have to check if it collides with A anymore).

- **ICollider**

The ICollider interface is implemented when an object has to check if it collides with other objects. For example a bubble moves, so it wants to check if it has to bounce off of another object, while a wall won't have to check if it hits another object, because it won't move anyway.

- **IPrepareable**

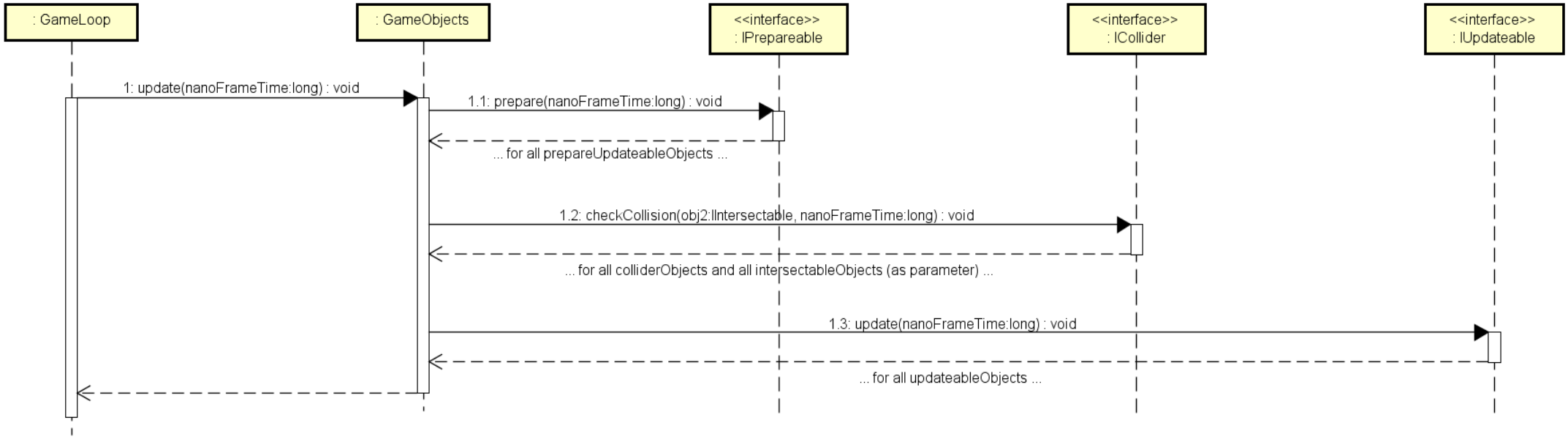
This interface is implemented when an object has to prepare before collision checks and updates are performed. Its method will be called before all collision checks/updates.

- **IUpdateable**

The IUpdateable interface is implemented by all objects which have to update their position every frame.

3. Make a sequence diagram of how the pattern works dynamically in your code (5 pts)

To show how the strategy pattern works, we have decided to make a sequence diagram of the update methods which are invoked at the start of each new frame.
The text: "… for all [ArrayList name] …" means that the above action will be performed for each object in the specified ArrayList before continuing with the next action.



GameLoop extends the AnimationTimer (abstract class), which is a javaFX class. This class will notify the GameLoop each time a new frame is loaded, so the GameLoop calls the update method of GameObjects once every frame. This is left out because it isn't important for the strategy pattern. It is however noteworthy for the sequence diagram.

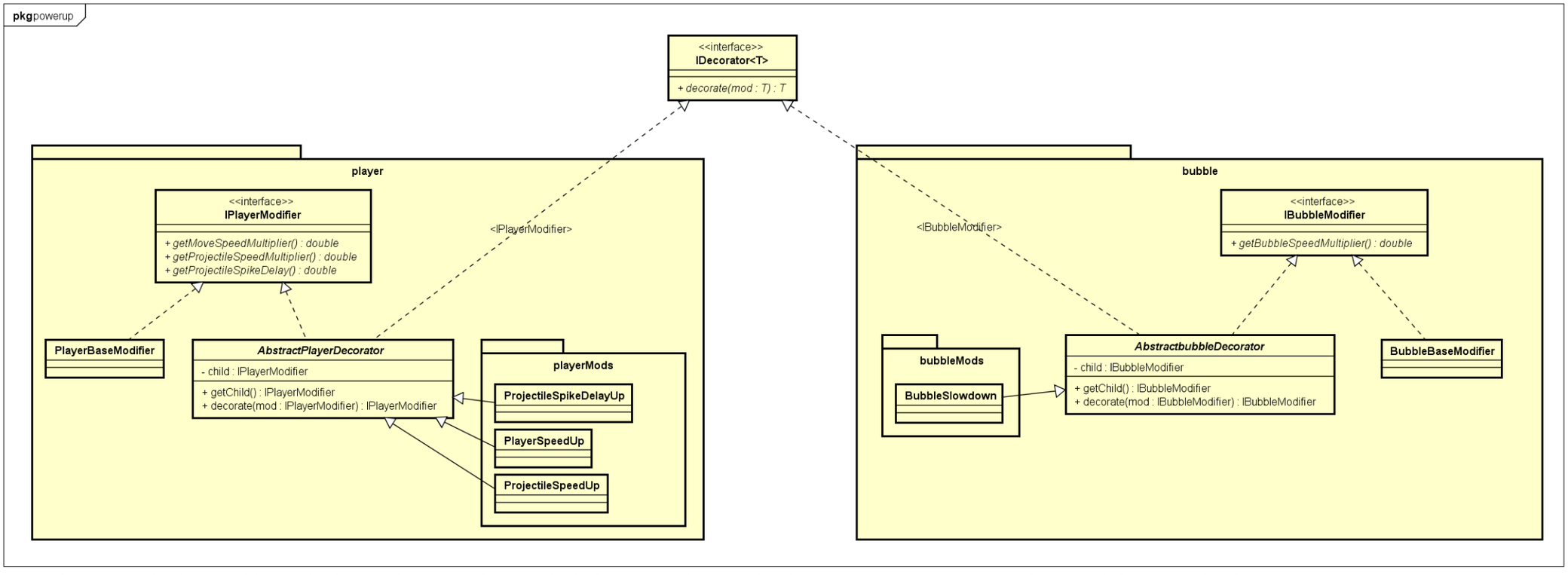
PART 2: Observer pattern (15 pts)

In our application we have 2 main objects in the view: bubbles and the player. When we implemented the powerups, we decided that powerups must be able to affect both the player and the bubbles. A powerup can for example slow down all bubbles, or it can speed up the player. Powerups must also be able to stack, i.e. if you pick up the slowdown powerup twice, the bubbles will be twice as slow and if you pick up the speed up powerup twice, the player will get two speed boosts.

1. Write a natural language description of why and how the pattern is implemented in your code (5 pts)

As described above, powerups should be stackable. For this to be possible there has to be some kind of history of which powerups have been picked up. This can be done with a list of values for each kind of powerup, but this way it would be very inconvenient to for example get the speed boost of the player if there are multiple different pickups which can influence the speed. Another drawback of this method is that you will have to add a lot of additional code if you develop a new powerup. Because of this we have decided to use the decorator pattern.
With the decorator pattern we only need a single object in each Player and each Bubble object to store the powerups. This object will have very simple methods, like “getSpeedMultiplier()”, so the Player and Bubble class won’t need much logic to use the pickups. The recursive aspect of the decorator pattern also makes it possible to keep track of the order in which the powerups have been picked up. This could for example be used to make more recently picked up powerups have a higher influence on the player than older pickups very easily (this is not currently implemented, but we wanted to make the code as easy to adapt to future changes as possible).
The way the decorator pattern is implemented in our code is very easy to understand, because it is implemented exactly as prescribed by the decorator pattern. We have actually implemented the decorator pattern twice: for the player and for the bubbles. To see the implementation and some explanation about the used classes, see the UML diagram of exercise 2 and description below it.

2. Make a class diagram of how the pattern is structured statically in your code (5pts)



Classes, methods, variables and relations between classes which don't contribute to the decorator pattern are omitted to avoid cluttering the diagram.
Notice that there are 2 separate decorator patterns implemented here. One for the player modifiers and one for the bubble modifiers.

- **IDecorator<T>**

The generic IDecorator interface is used to make it easier to generate a random pickup. It being a separate interface does not contribute to the decorator pattern.

- **IPlayerModifier and IBubbleModifier**

The IPlayerModifier and IBubbleModifier interfaces are the interfaces for objects that can have responsibilities added to them dynamically.

- **PlayerBaseModifier and BubbleBaseModifier**

The PlayerBaseModifier and BubbleBaseModifier classes define objects to which additional responsibilities can be added. They return default values for each implemented method (i.e. 1 for each "getMultiplier" method and 0 for all other methods).

- **AbstractPlayerDecorator and AbstractBubbleDecorator**

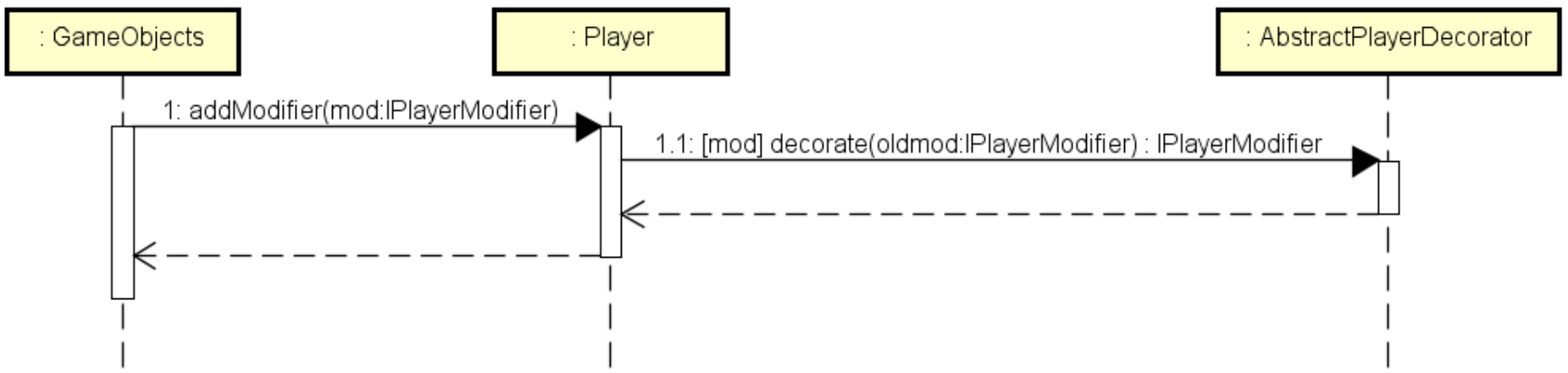
The AbstractPlayerDecorator and AbstractBubbleDecorator maintain a reference to an IPlayerModifier/IBubbleModifier object.

- **playerMods and bubbleMods**

The playerMods and bubbleMods packages contain the classes which extend the abstract decorator classes and implement the methods of the IPlayerModifier/IBubbleModifier interface.

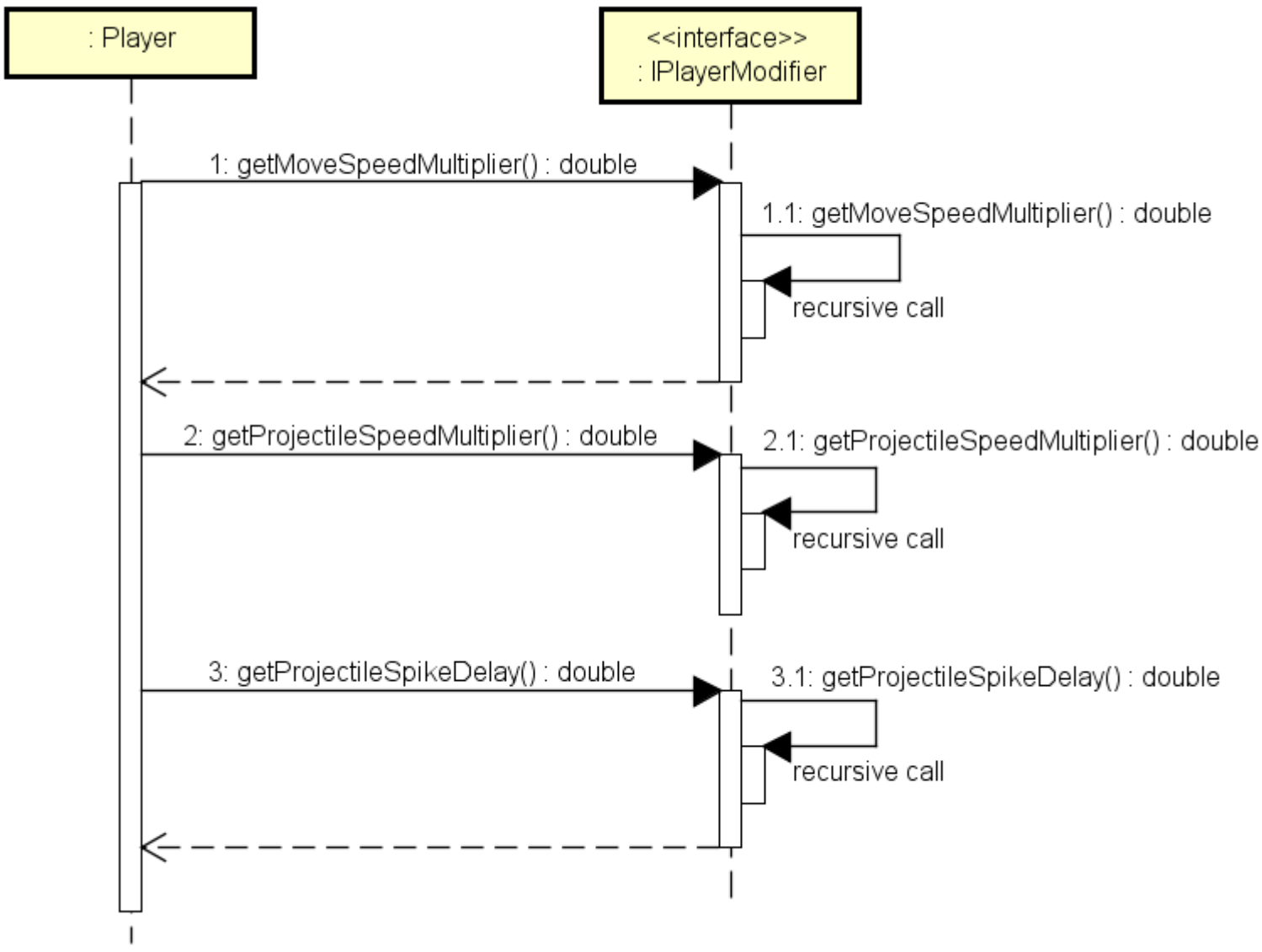
3. Make a sequence diagram of how the pattern works dynamically in your code (5 pts)

We will make 2 separate sequence diagrams, because the full functionality of the decorator pattern is hard to show with just one sequence diagram. The first sequence diagram will show what will happen when the player picks up a new powerup and the second sequence diagram will show how the player can get the effects powerups.



When the player collides with the pickup, GameObjects will be notified by the pickup.

When the “addModifier()” method of the player is added, the player decorates the received modifier with his current modifier, thus wrapping the new modifier around the old modifier and then replaces the old modifier with the new one.



This sequence diagram shows which methods the player will call when he will fire a projectile.

The player will always call the “getMoveSpeedMultiplier()” method during every update to determine his speed. If the player fires a projectile he will call the “getProjectileSpeedMultiplier()” and “getProjectileSpikeDelay()” methods and pass the received values to the constructor of the projectile.