COMP-302: SOFTWARE ENGINEERING

NEED FOR SPEAR

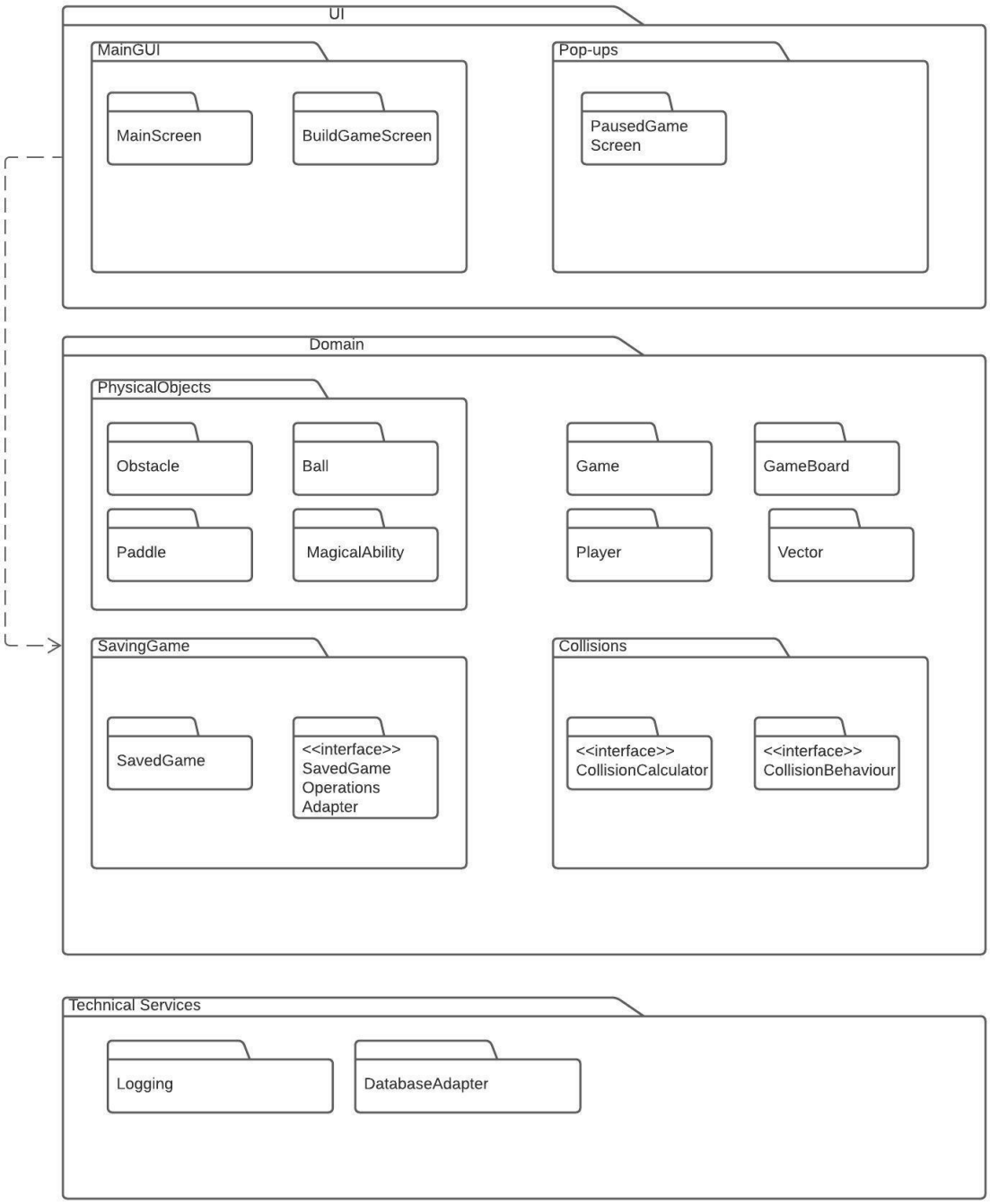
Phase1: DESIGN

SyntacticSugar:

İrem Demir
Atakan Kara
Asu Tutku Gökçek
Farrin Marouf Sofian
Eren Yenigül

Table of Contents

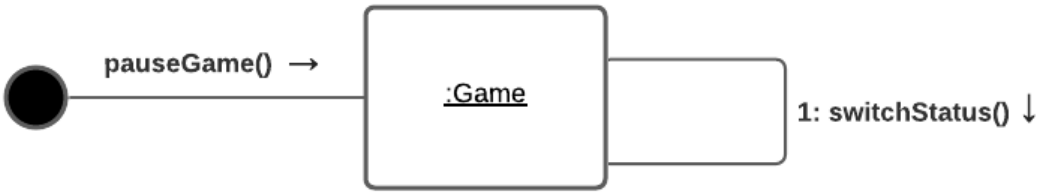
Logical Architecture



Interaction Diagrams

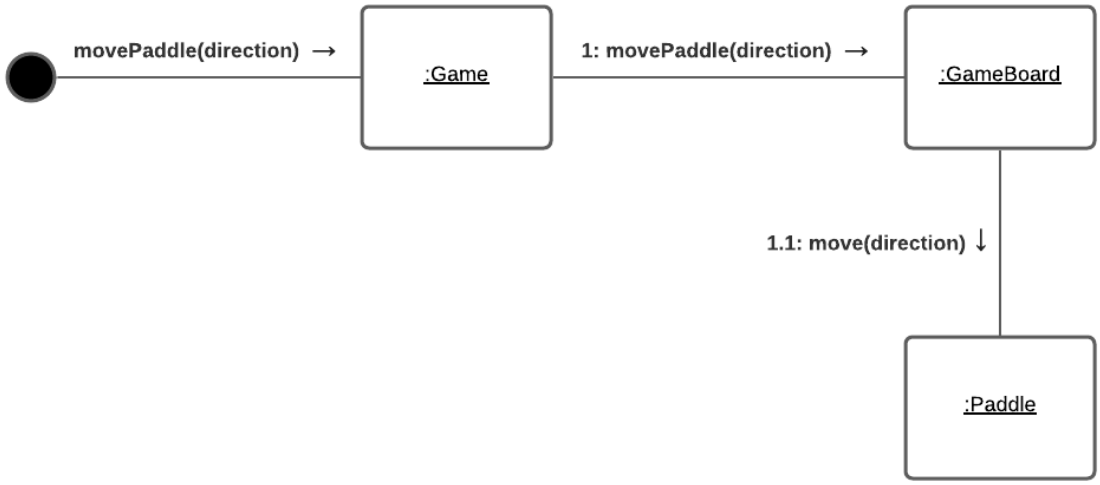
UML Communication Diagrams

CD.1 : Pause Game

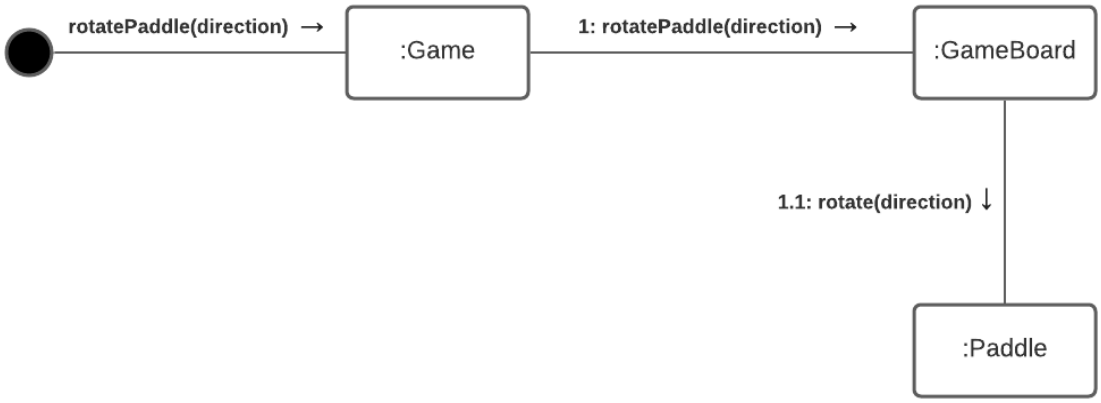


CD.2: Resume Game

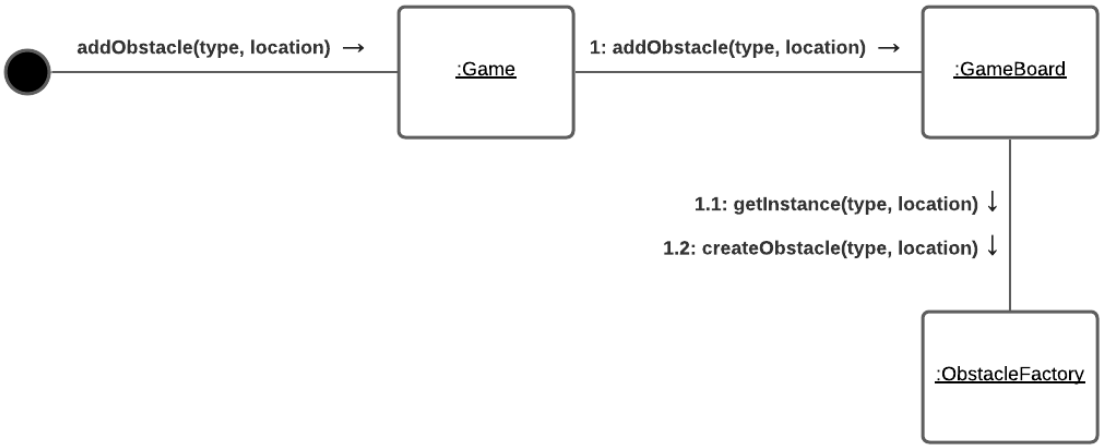
CD.3: Move Paddle



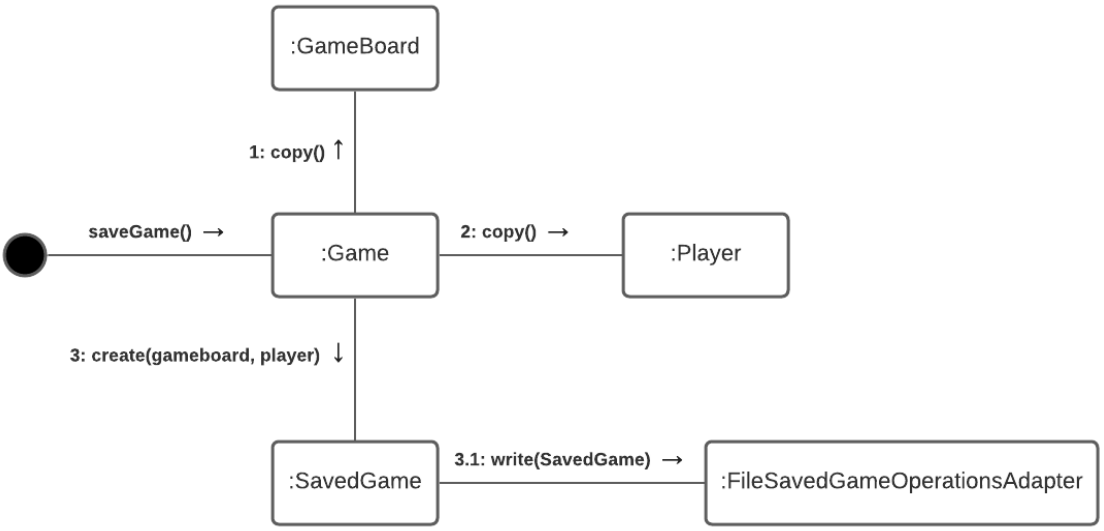
CD.4: Rotate Paddle



CD.5: Add Obstacle

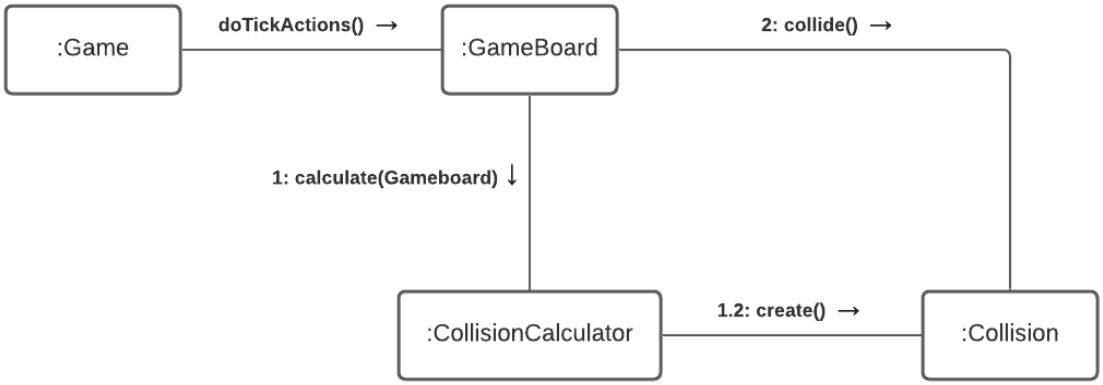


CD.6: Save Game

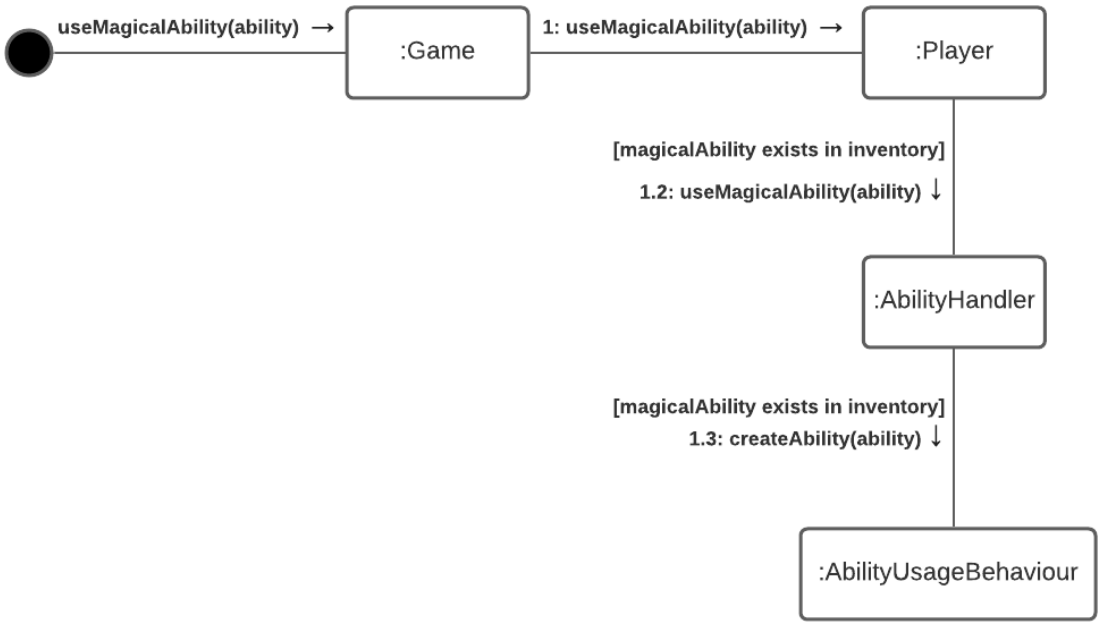


CD.7: Load Game

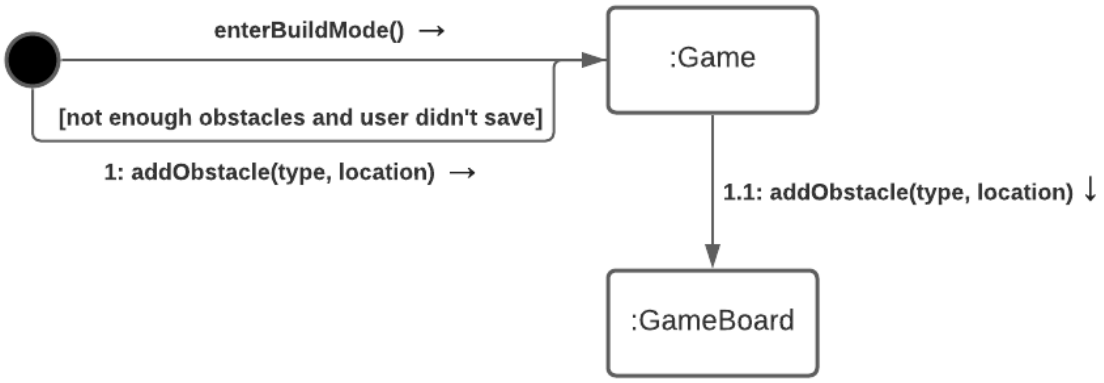
CD.8: Do Tick Actions



CD.9: Use Magical Ability

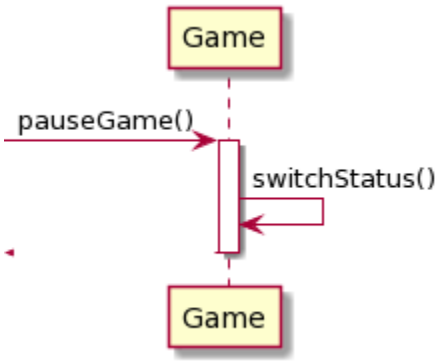


CD.10 : Enter Build Mode

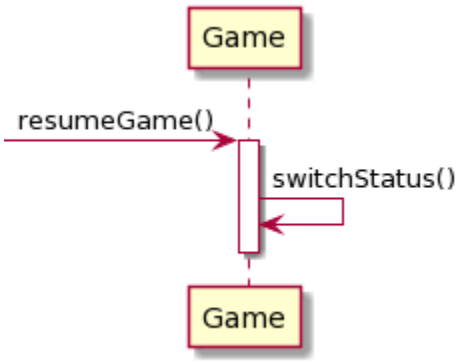


UML Sequence Diagrams

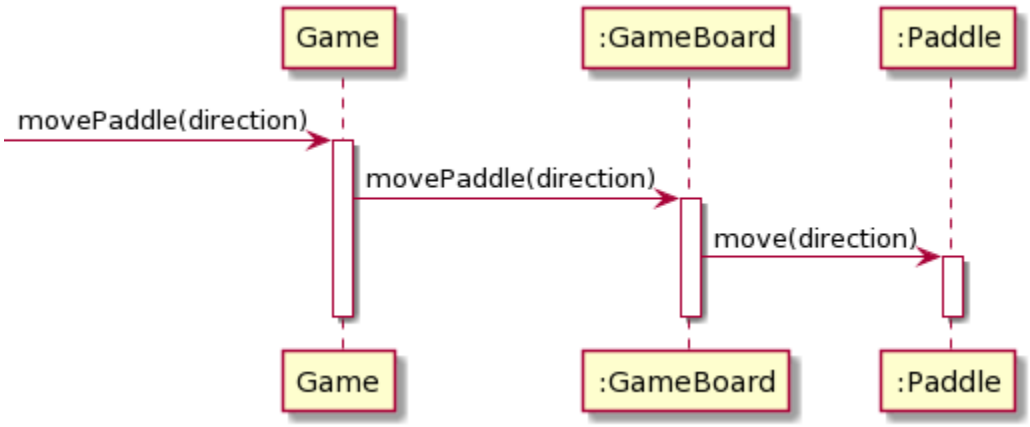
SD.1: Pause Game



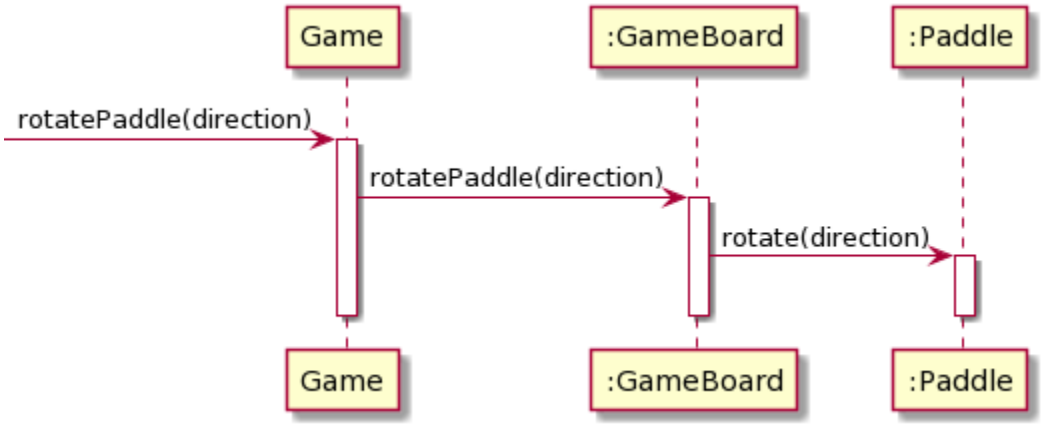
SD.2: Resume Game



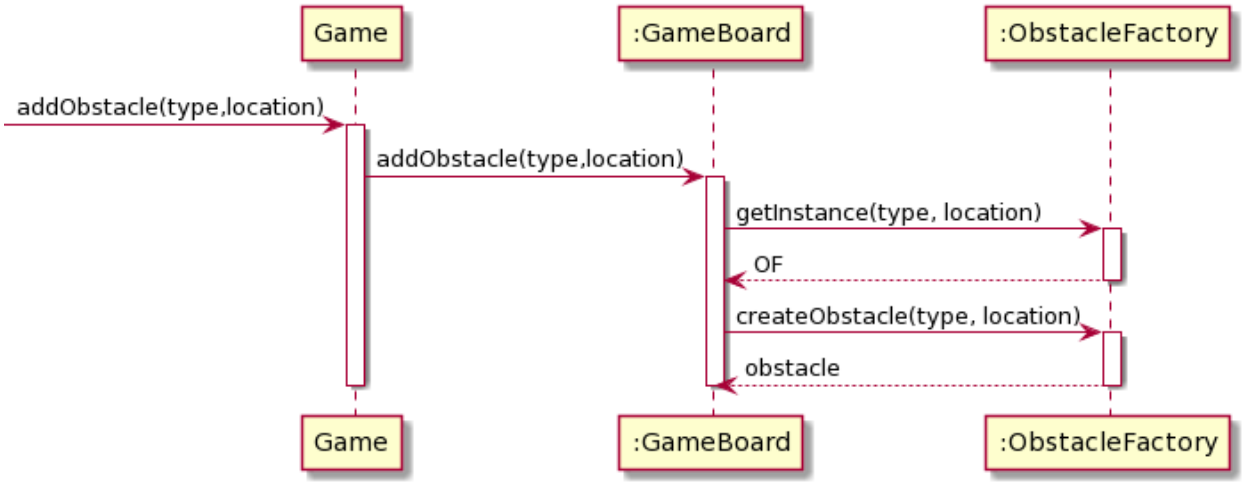
SD.3: Move Paddle



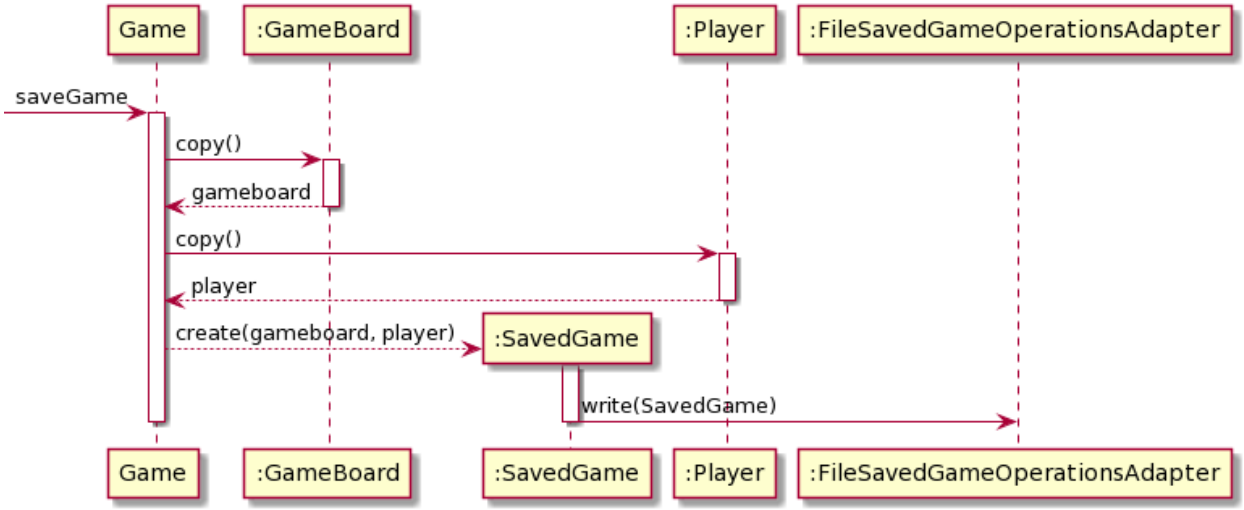
SD.4: Rotate Paddle



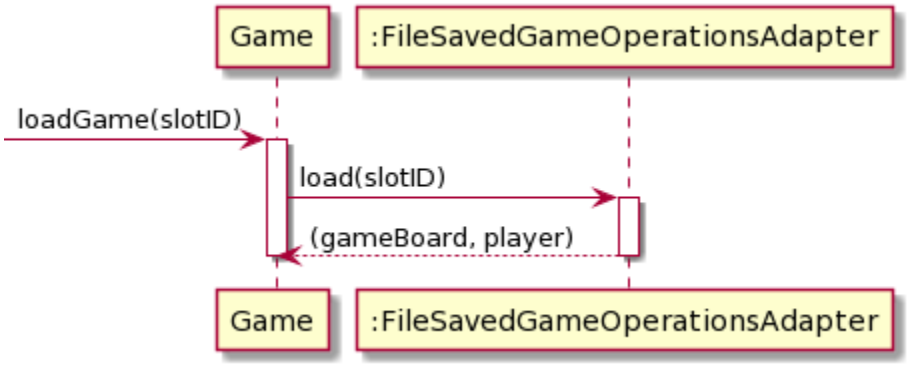
SD.5: Add Obstacle



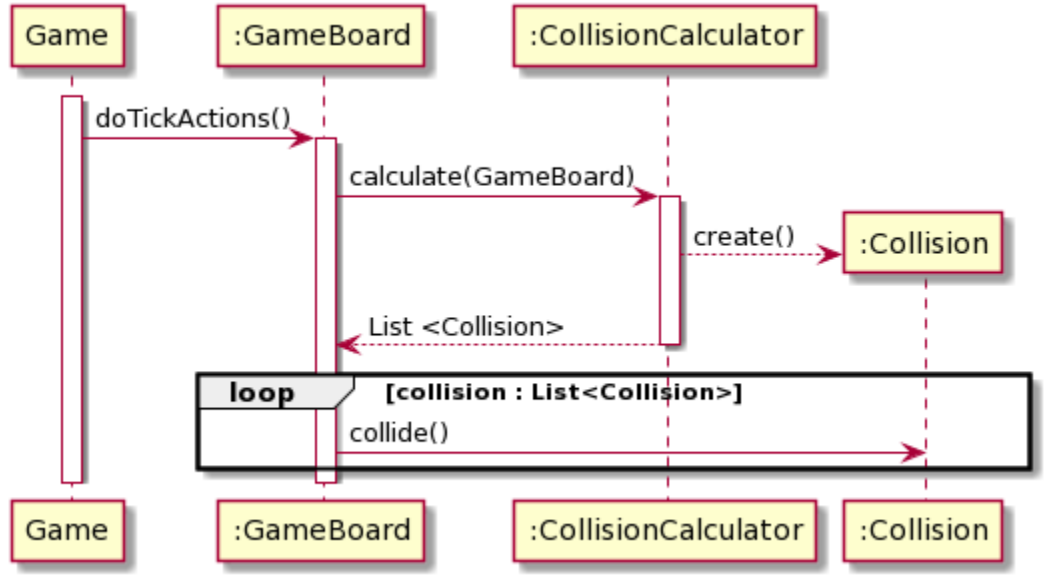
SD.6: Save Game



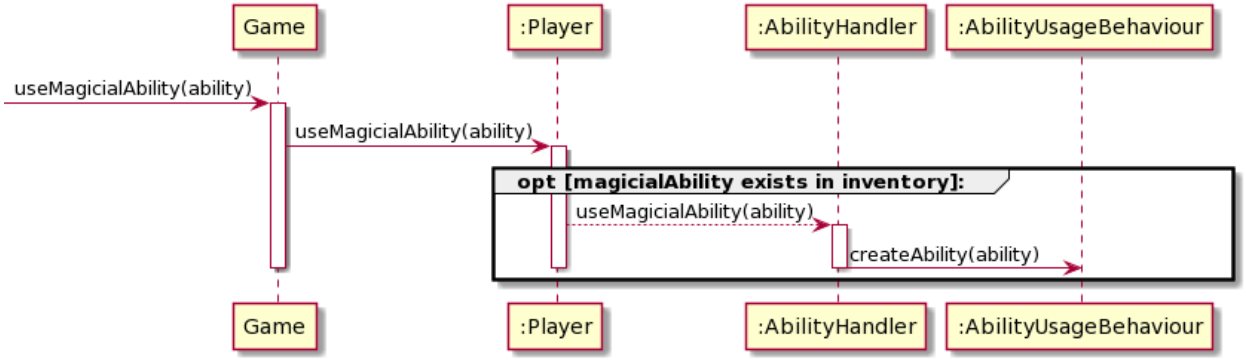
SD.7: Load Game



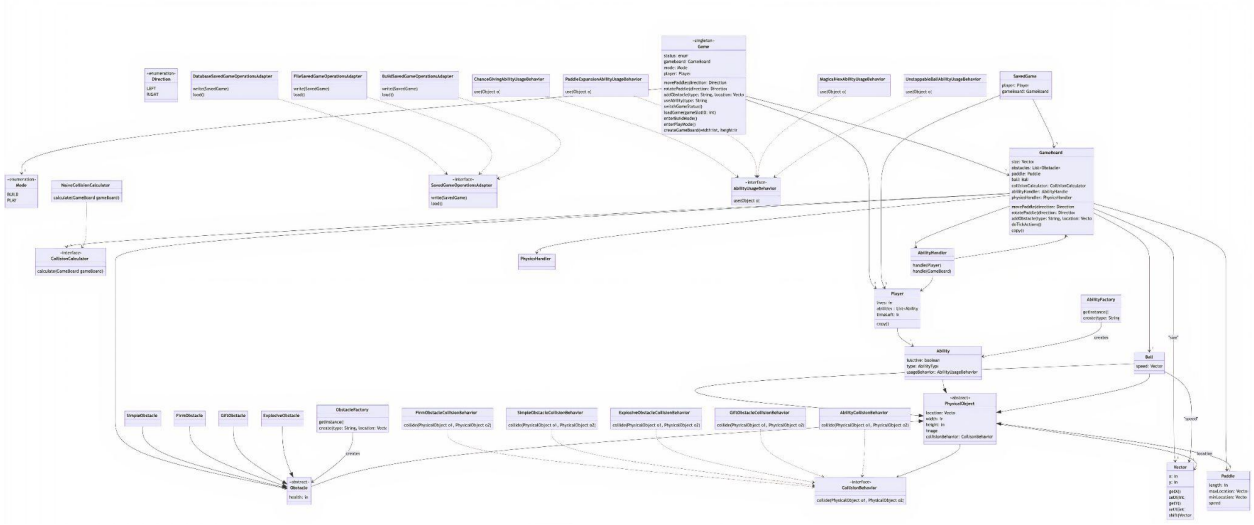
SD.8: Do Tick Actions

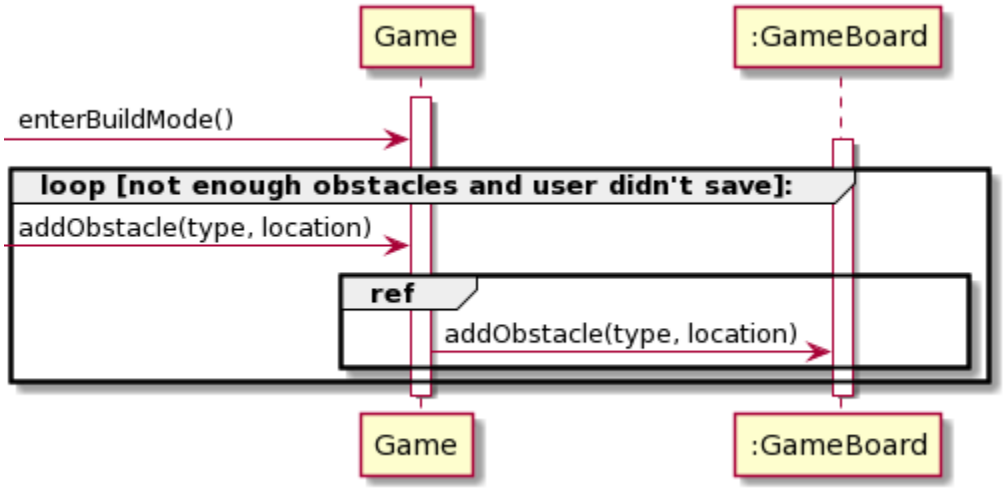


SD.9: Use Magical Ability



SD.10: Enter to Build Mode

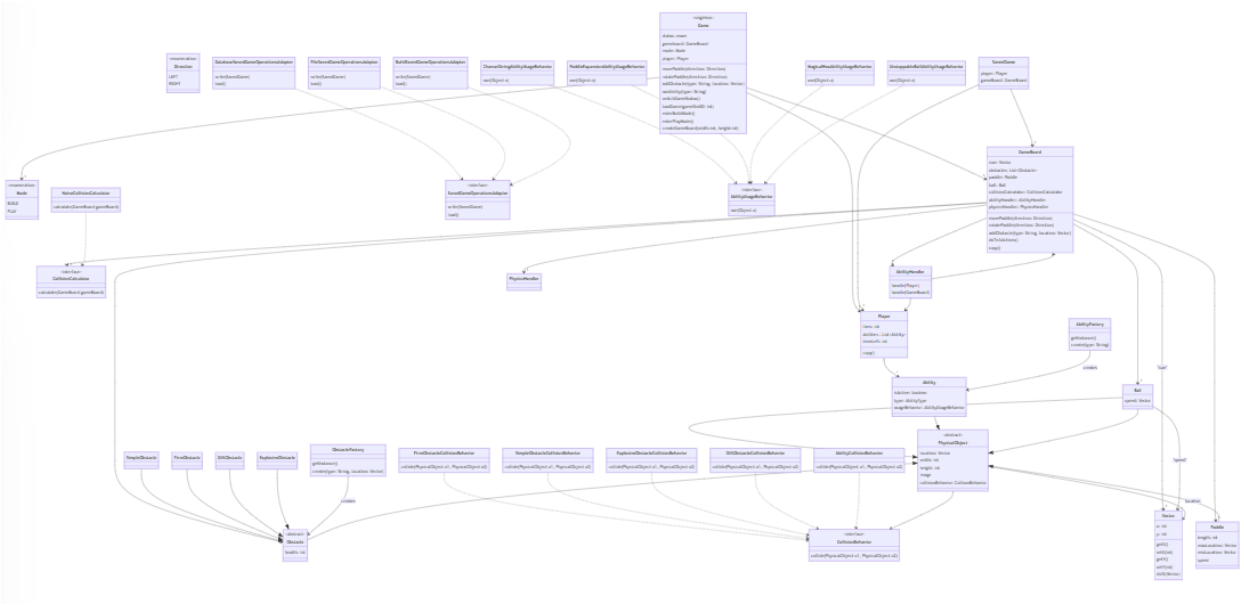


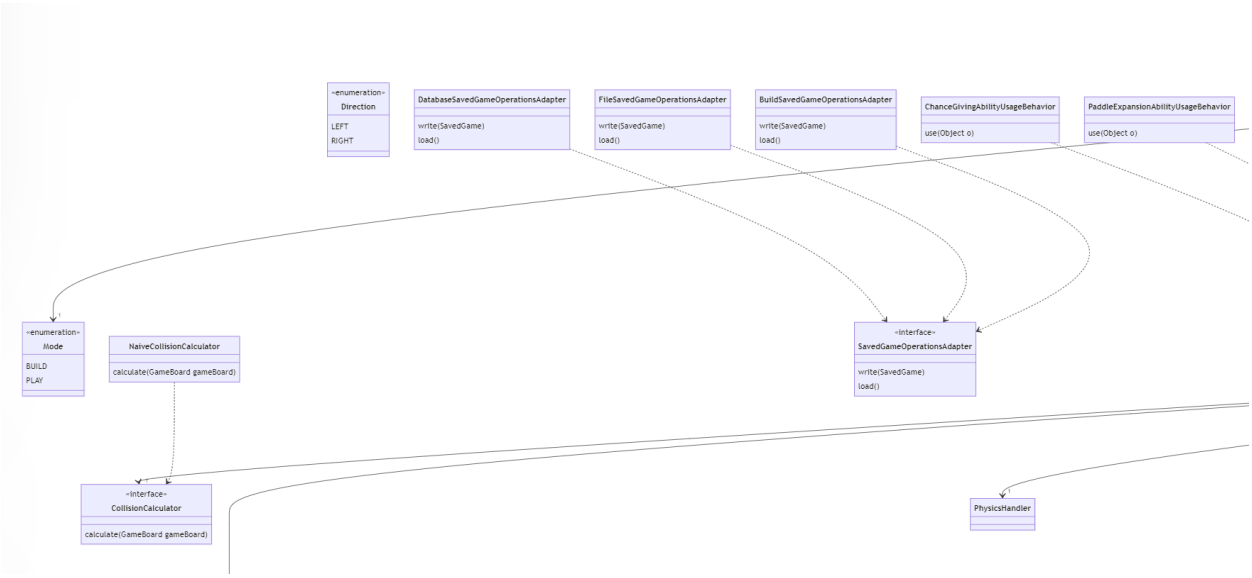


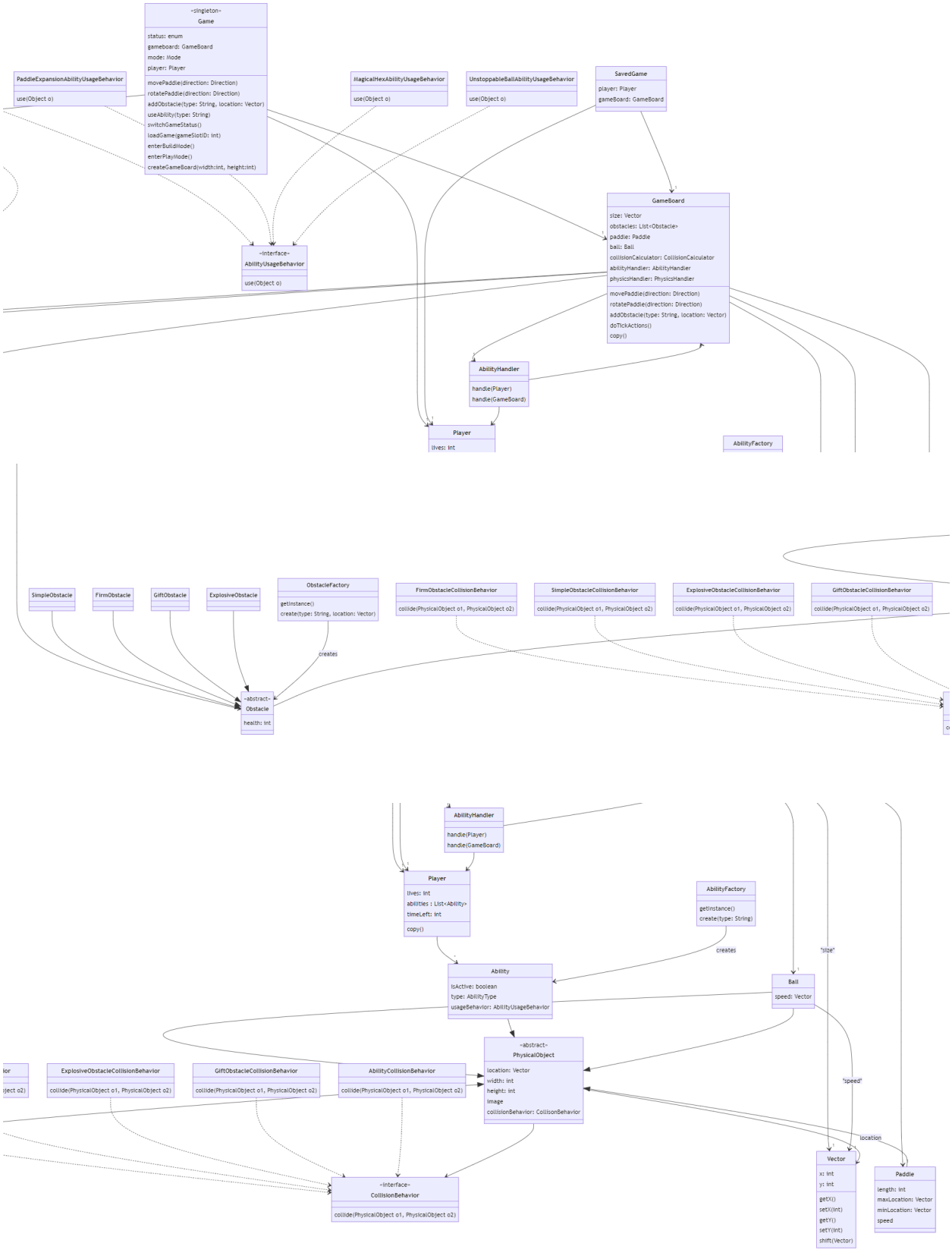
UML Class Diagram

Class Diagram

(First image is the entire UML Class Diagram. We added all parts of the entire diagram separately so that it could be read better.)




«enumeration»
Direction
LEFT
RIGHT
DatabaseSavedGameOperationsAdapter
write(SavedGame)
load()
FileSavedGameOperationsAdapter
write(SavedGame)
load()
BuildSavedGameOperationsAdapter
write(SavedGame)
load()
ChanceGivingAbilityUsageBehavior
use(Object o)
PaddleExpansionAbilityUsageBehavior
use(Object o)
«enumeration»
Mode
BUILD
PLAY
NaiveCollisionCalculator
calculate(GameBoard gameBoard)
«interface»
SavedGameOperationsAdapter
write(SavedGame)
load()
«interface»
CollisionCalculator
calculate(GameBoard gameBoard)
PhysicsHandler


«singleton»
Game
status: enum
gameboard: GameBoard
mode: Mode
player: Player
movePaddle(direction: Direction)
rotatePaddle(direction: Direction)
addObstacle(type: String, location: Vector)
useAbility(type: String)
switchGameStatus()
loadGame(gameSlotID: int)
enterBuildMode()
enterPlayMode()
createGameBoard(width:int, height:int)
PaddleExpansionAbilityUsageBehavior
use(Object o)
MagicalHexAbilityUsageBehavior
use(Object o)
UnstoppableBallAbilityUsageBehavior
use(Object o)
SavedGame
player: Player
gameBoard: GameBoard
«interface»
AbilityUsageBehavior
use(Object o)
GameBoard
size: Vector
obstacles: List<Obstacle>
paddle: Paddle
ball: Ball
collisionCalculator: CollisionCalculator
abilityHandler: AbilityHandler
physicsHandler: PhysicsHandler
movePaddle(direction: Direction)
rotatePaddle(direction: Direction)
addObstacle(type: String, location: Vector)
doTickActions()
copy()
AbilityHandler
handle(Player)
handle(GameBoard)
Player
lives: int
AbilityFactory
SimpleObstacle
FirmObstacle
GiftObstacle
ExplosiveObstacle
ObstacleFactory
getInstance()
create(type: String, location: Vector)
creates
FirmObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
SimpleObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
ExplosiveObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
GiftObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
«abstract»
Obstacle
health: int
AbilityHandler
handle(Player)
handle(GameBoard)
Player
lives: int
abilities : List<Ability>
timeLeft: int
copy()
AbilityFactory
getInstance()
create(type: String)
creates
"size"
Ability
isActive: boolean
type: AbilityType
usageBehavior: AbilityUsageBehavior
Ball
speed: Vector
«abstract»
PhysicalObject
location: Vector
width: int
height: int
image
collisionBehavior: CollisonBehavior
"speed"
location
ExplosiveObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
GiftObstacleCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
AbilityCollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
«interface»
CollisionBehavior
collide(PhysicalObject o1, PhysicalObject o2)
Vector
x: int
y: int
getX()
setX(int)
getY()
setY(int)
shift(Vector)
Paddle
length: int
maxLocation: Vector
minLocation: Vector
speed

Discussion of Design Patterns

Controller Pattern

Our Game class is designed for getting all the information and interaction from UI then distributing it to all other domain objects. With this implementation, Game class becomes the one place in which UI and domain layers get in touch. This helps us to evaluate and decide how to distribute inputs of UI to domain. One advantage of this is that this reduces coupling between UI and domain layers as the interaction happens in one place. Another advantage is that this helps us to think how we will integrate the model view separation. A key point in using this pattern is to carefully assign the message received from the UI to responsible classes and avoid handling the tasks solely in Controller pattern as it can result in low cohesion or in some cases, result in high coupling with other classes.

Creator Pattern

We used a creator pattern in several places. We used this pattern in GameBoard because GameBoard has the responsibility of creating our game objects like obstacles, paddle, and ball. We also used the creator pattern for the Game class because it creates the GameBoard. Applying the creator pattern makes us think in terms of responsibilities and create instances of classes in the most relevant class. Even though in our project it was easier to identify which class should be based on the creator pattern, in some cases, if there are several candidate classes it may be difficult to choose the best one without violating high cohesion and low coupling principles.

Factory Pattern

For creating Obstacles and Magical Abilities, we used factories. That made it possible to easily create those objects with a given type. An advantage of this pattern is that if at some point we decide to improve the game or add new features to the classes which use factory design patterns, we can do so easily, since the desired features will only have to implement the interface class and its methods based on the features functionality. A minor disadvantage but worth mentioning in this approach may be the fact that while creating the objects if there are many different obstacles or magical abilities, the code will consist of an inevitable amount of if and else statements.

Singleton Pattern

As a team we planned our game project to have one Game object only, which will be used constantly through the instances of itself. Therefore, for the “Game” class as well as our factory

class (ObstacleFactory) we used the Singleton Pattern. This helps us achieve high cohesion and low coupling. However, one of the disadvantages of this approach is that later in our project we are going to use unit testing to test our code, and the singleton pattern makes it difficult to recognize dependency chains which leads to a more difficult unit testing process.

Strategy Pattern

Collision handling is an important part of our project. However, it is not just a simple case in which we can apply the same logic in possible scenarios. Basically, our way of treating collisions may change in the presence of different magical abilities although the ways are closely related. Therefore, we decided to implement CollisionBehaviour following the strategy pattern. In this way we can have control over collision in the usage of magical ability differently.

Adapter Pattern

For saving and loading the game, we realized that there might be different interfaces like from or to a file or database. Moreover, building a game or loading from a building mode can be treated as another interface. Although there are different options, they all have the same responsibilities. Therefore, we decided to use an adapter.

Low Coupling and High Cohesion

During all our discussions of design and how to assign responsibilities, low coupling and high cohesion are the two things we always have in consideration. We try to make our design to have low dependency, tolerance to change, and objects that are focused. We applied other patterns to reduce coupling and increase cohesion.