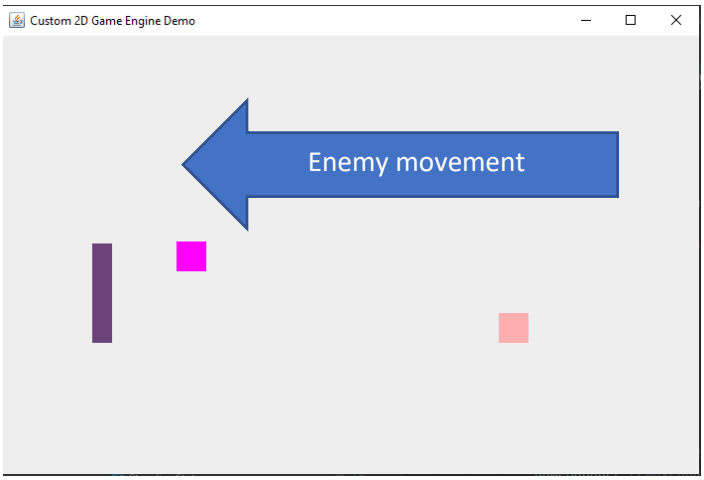
Furkan Gençtürk
21703067

Contents

Furkan Gençtürk
21703067

1. Project Description



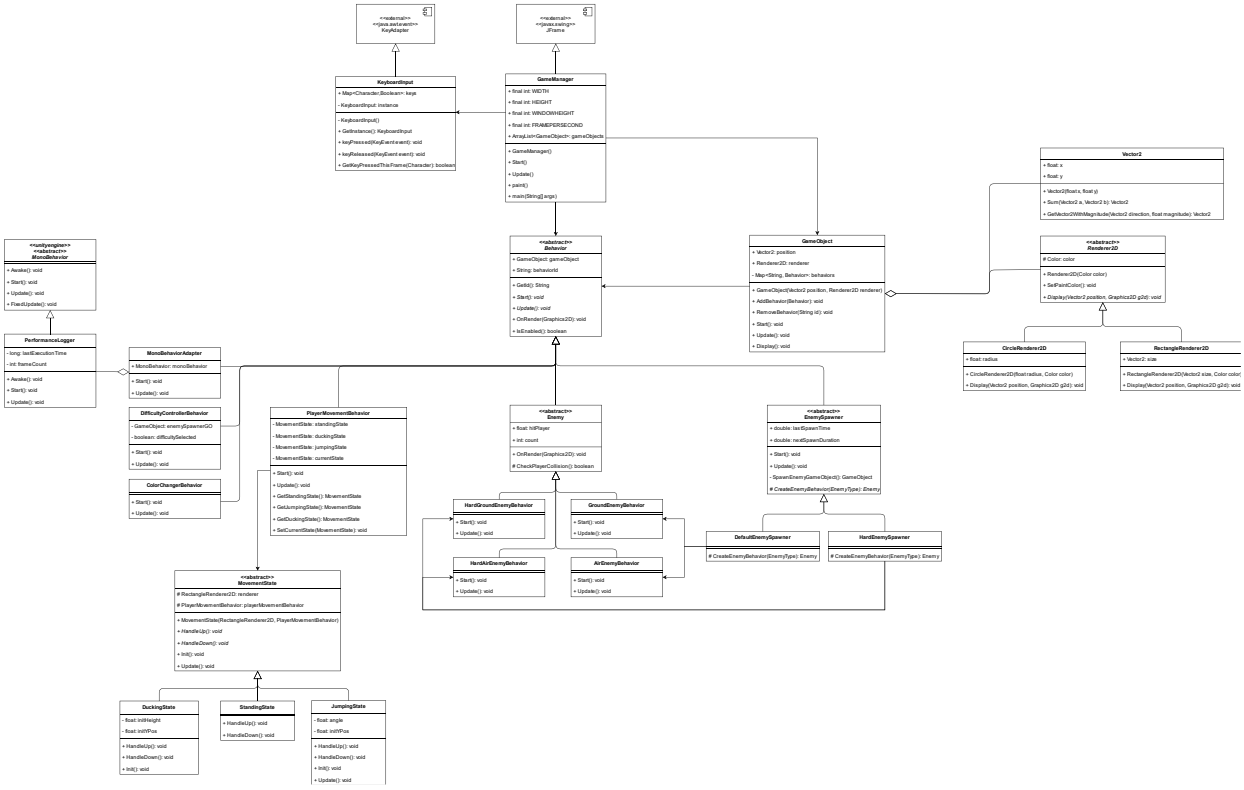
The project aims to provide a very primitive game engine and a demo game. In demo game, enemies move from right to left and the player aims to dodge them by either jumping(pressing W key) or ducking(pressing S key).

When game first starts, pressing E key starts the game in the NORMAL mode where enemies can be dodged easily.

Pressing H key starts the game in the HARD mode where enemies move in a weird pattern and they are harder to dodge.

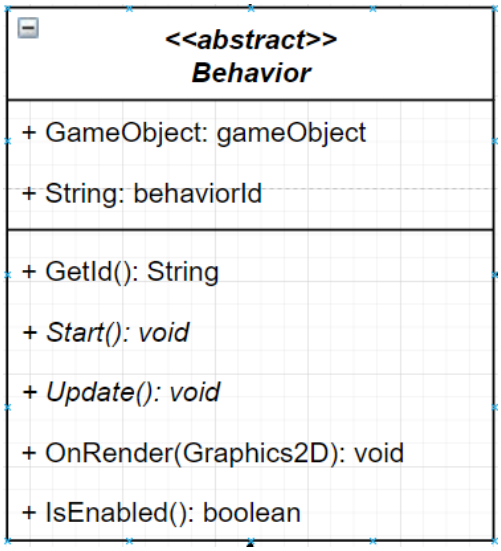
2. Class Diagram

You can also find this class diagram as a SVG file in the root folder of the project.



3. Design Patterns

3.1. Template Method



Behavior abstract class provides the algorithm for game loop. Every concrete class extending Behavior class will go through the same steps in each gameplay session.

Start function is triggered when a behavior is attached to a GameObject in the scene for the first time.

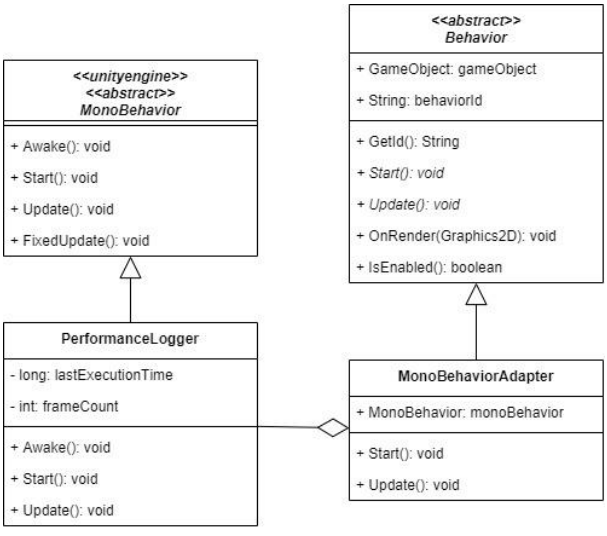
Update function is triggered in each frame to calculate new state of the game before rendering.

InEnabled returns true by default. Concrete classes extending Behavior can change this behavior to opt out being called in each Update and OnRender states.

OnRender lifecycle hook is triggered when a new frame is started to be drawn on the screen. Behaviors willing to contribute this frame, can use this optional function to draw things on the screen.

There are lots of concrete classes extending Behavior and benefiting from Template method design pattern. Each custom behavior uses Start and Update functions to change the game state. For example, PlayerMovementBehavior moves the player in these functions. Additionally, Enemy class draws a huge red rectangle in OnRender hook when it hits the player.

3.2. Adapter Pattern



I used AdapterPattern to adapt MonoBehavior abstract class in Unity Engine to Behavior abstract class in my custom 2D engine. MonoBehavior in UnityEngine has slightly different lifecycle hooks compared to mine.

Awake is called before application starts, therefore before the Start function. However, since we do not have this functionality, I simply call both Awake and Start functions in MonoBehavior in my Start function is Behavior class.

```
public void Start() {
    monoBehavior.Awake();
    monoBehavior.Start();
}
```

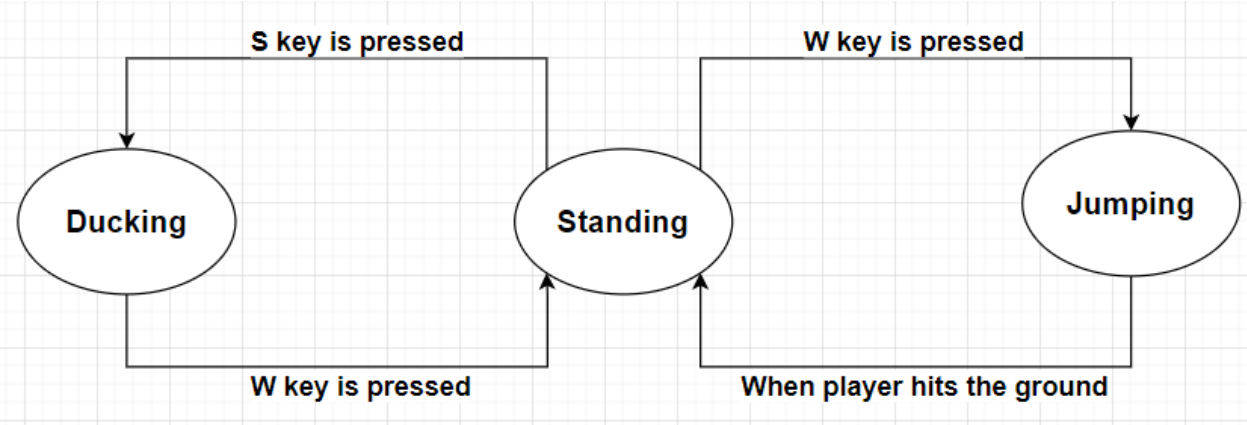
Additionally, FixedUpdate in UnityEngine is called whenever there is a physics calculation is done in the app. Since my custom engine does not have a Physics Engine yet, I do not have this functionality. Therefore, I simply call this FixedUpdate in my Update function.

```
public void Update() {
    monoBehavior.Update();
```

```
    monoBehavior.FixedUpdate();
}
```

Therefore, PerformanceLogger, which logs duration passed until the application opens and FPS (frame per second) in each second, extending MonoBehavior can also work in our engine as a Behavior Object, wrapped inside MonoBehaviorAdapter.

3.3. State Pattern



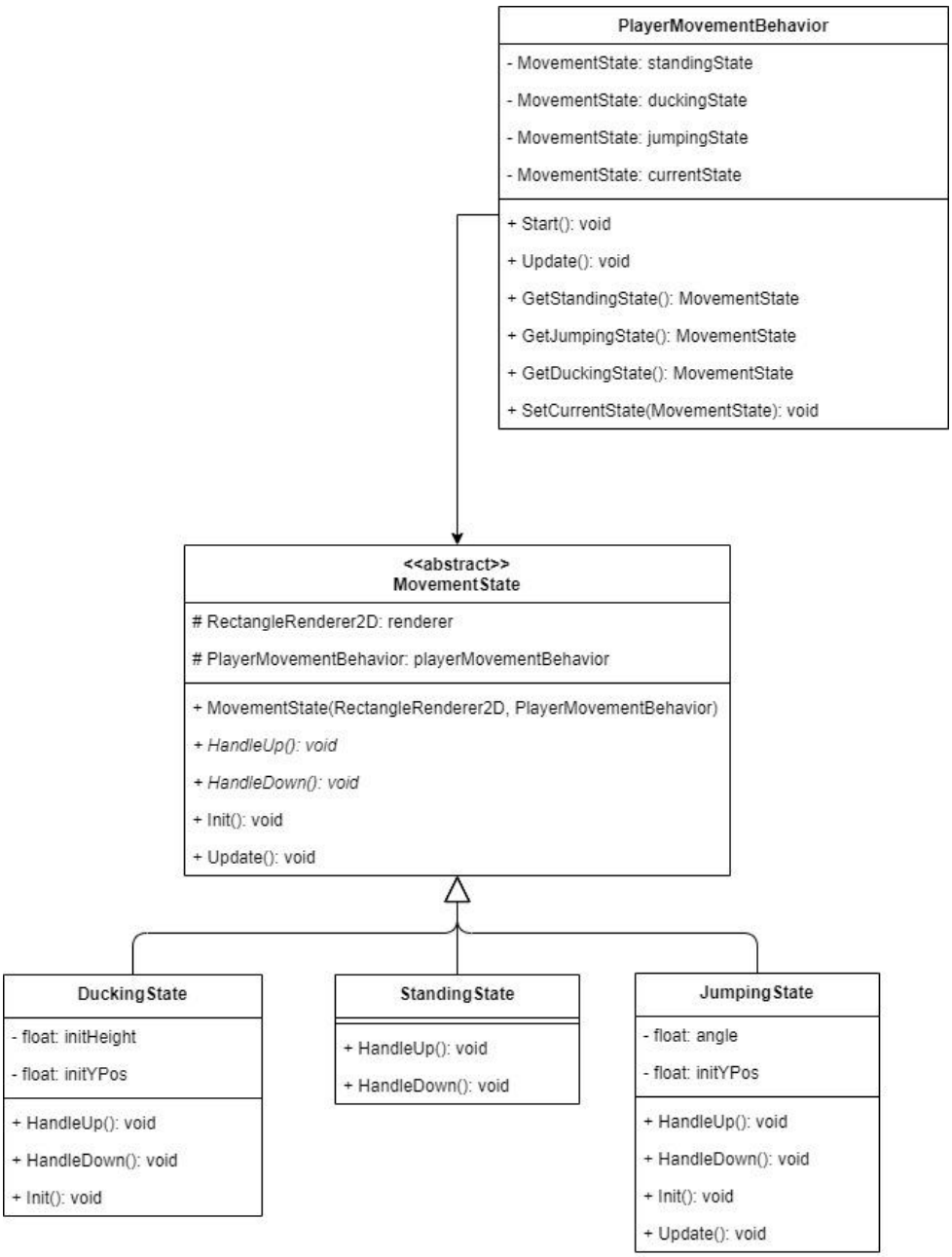
I used StatePattern in my game for the player object. Player can jump or duck to dodge enemies which are coming towards to the player. You can see above how a player can transit between these states and to achieve this, I used state pattern with

```
HandleUp        // When W is pressed
HandleDown      // When S is pressed
Init            // When state is changed to this state
Update          // When Update method called in game loop and current state is this state
```

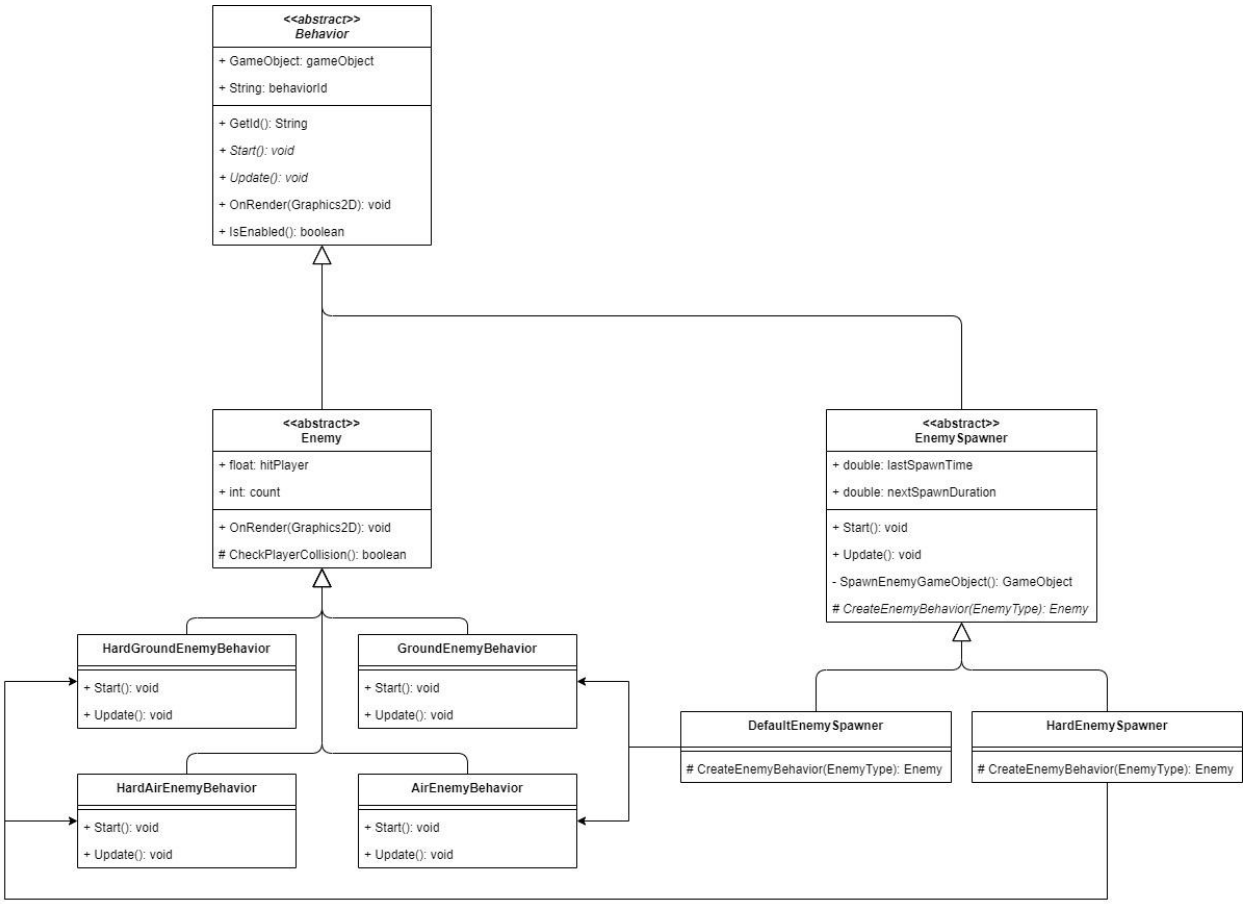
is called. Each concrete state implements required functions and transit between these states.

Here is the class diagram showing only this design pattern:

Furkan Gençtürk
21703067



3.4. Factory Method



Our game has two different enemy type: GROUND and AIR enemies. Our game has also two difficulty settings: NORMAL and HARD. For each difficulty setting, we have a different enemy concrete behavior.

Normal Enemies: GroundEnemyBehavior, AirEnemyBehavior

Hard Enemies: HardGroundEnemyBehavior, HardAirEnemyBehavior

There is a EnemySpawner class which spawns random enemy for every 2 (+/- 0.5) seconds. It spawns a game object but lets its subclasses decide which EnemyBehavior this enemy has.

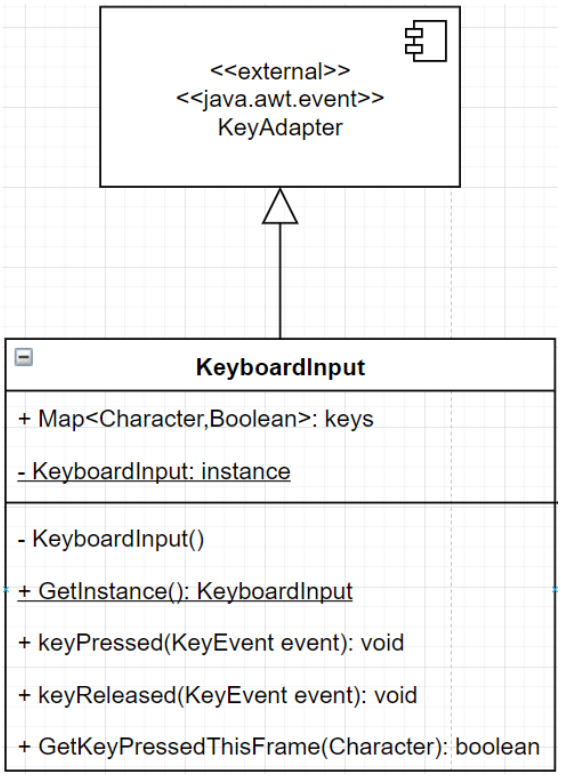
```
public abstract class EnemySpawner extends Behavior {
    enum EnemyType {
        AIR,
        GROUND
    }

    protected abstract Enemy CreateEnemyBehavior(EnemyType type);

    private GameObject SpawnEnemyGameObject() {
        Renderer2D enemyRenderer = new RectangleRenderer2D(new Vector2(30,
30));
        EnemyType type =Math.random() > 0.5 ? EnemyType.AIR :
EnemyType.GROUND;
        Behavior enemyBehavior = CreateEnemyBehavior(type); // SUBCLASSES
```

```
DECIDE
        GameObject enemyGameObject = new GameObject(new
Vector2(GameManager.WIDTH + 100, GameManager.HEIGHT / 2), enemyRenderer);
        enemyGameObject.AddBehavior(enemyBehavior);
        GameManager.spawnedGameObjects.add(enemyGameObject);
        return enemyGameObject;
    }
```

3.5. Singleton Pattern



KeyboardInput class implements the Singleton Pattern behavior. Only one instance of KeyboardInput can exist in the app and it provides a global access for other classes.

Why do we have to ensure there is only one KeyboardInput?

The reason is the keyboard customization. Every GameEngine filters input via only one class to ensure key-mapping is stable throughout the whole app.

Key-mapping is letting player chose which key will trigger the `w` key. That means you can develop the game where ‘w’ key jumps. However, later player can customize this to map ‘j’ to ‘w’ key. That means whenever player presses ‘j’ key, the whole application behaves as if ‘w’ was pressed. Therefore, this key-mapping functionality can be implemented in this unique instance of KeyboardInput class.

Many behaviors in my game engine will need to know about player inputs and global access to this unique KeyboardInput instance is very helpful in that. For example, in player movement, we check if player clicks ‘w’ or ‘s’ keys.

```
public void Update() {
    if(KeyboardInput.GetInstance().IsKeyPressedThisFrame('w')) {
        currentState.HandleUp();
    }
    if(KeyboardInput.GetInstance().IsKeyPressedThisFrame('s')) {
        currentState.HandleDown();
    }
}
```

When selecting game difficulty, player presses ‘e’ for normal mode and ‘h’ for hard mode, where enemies moves more unpredictable.

```
public void Update() {
    if(difficultySelected) {
        return;
    }
```

```
    if(KeyboardInput.GetInstance().IsKeyPressedThisFrame('e')) {
        Behavior enemyBehavior = new DefaultEnemySpawner();
        difficultySelected = true;
    } else if(KeyboardInput.GetInstance().IsKeyPressedThisFrame('h')) {
        Behavior enemyBehavior = new HardEnemySpawner();
        difficultySelected = true;
    }
}
```