Brian Larosiliere

Mr.Brown

December 9, 2022

Design Mini-Paper

Flyweight Design Pattern

As a programmer, time and space constraints are prioritized over most things. That being how fast a program can run and how much space is necessary for a program to run. Therefore it is important that the programs we write are as compact and wasteless as possible. To achieve these practices, it is commonplace to use fitting design patterns whenever possible. One such design pattern that proved beneficial to my project is the Flyweight Design Pattern. The Flyweight Design Pattern decreases the amount of space a program requires by having similar objects share certain characteristics.

As earlier noted, memory space is detrimental to creating a program. A program that goes over the amount of memory the CPU can allocate will crash and is basically worthless. This issue is capable of arising when we have multiple objects that share multiple characteristics but differentiate slightly. Take the circumstance given by Refractoring.Guru for instance, “The actual problem was related to your particle system. Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing plenty of data. At some point, when the carnage on a player’s screen reached its climax, newly created particles no longer fit into the remaining RAM, so the program crashed.(Refactoring.guru)”[1] The particles noted share characteristics like sprites, colors, and methods, but they are all represented by their own object. The Flyweight Design Pattern combats this issue specifically by having objects share characteristics.

When having multiple similar objects it's easy to see how they all overlap. The Flyweight Design pattern makes use of these overlapping parts to reduce storage using a hashmap. Geekforgeeks does a great job of explaining this, "In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.(Geekforgeeks)"[2] There are two parts to an object when using the Flyweight Design Pattern, the intrinsic and extrinsic state. The intrinsic state are the parts that are shared and overlap among similar objects. While the extrinsic state is what is independent and is unique among objects. By using a hashmap, whenever we create a new object we can link the intrinsic state by passing a hash key for that trait.

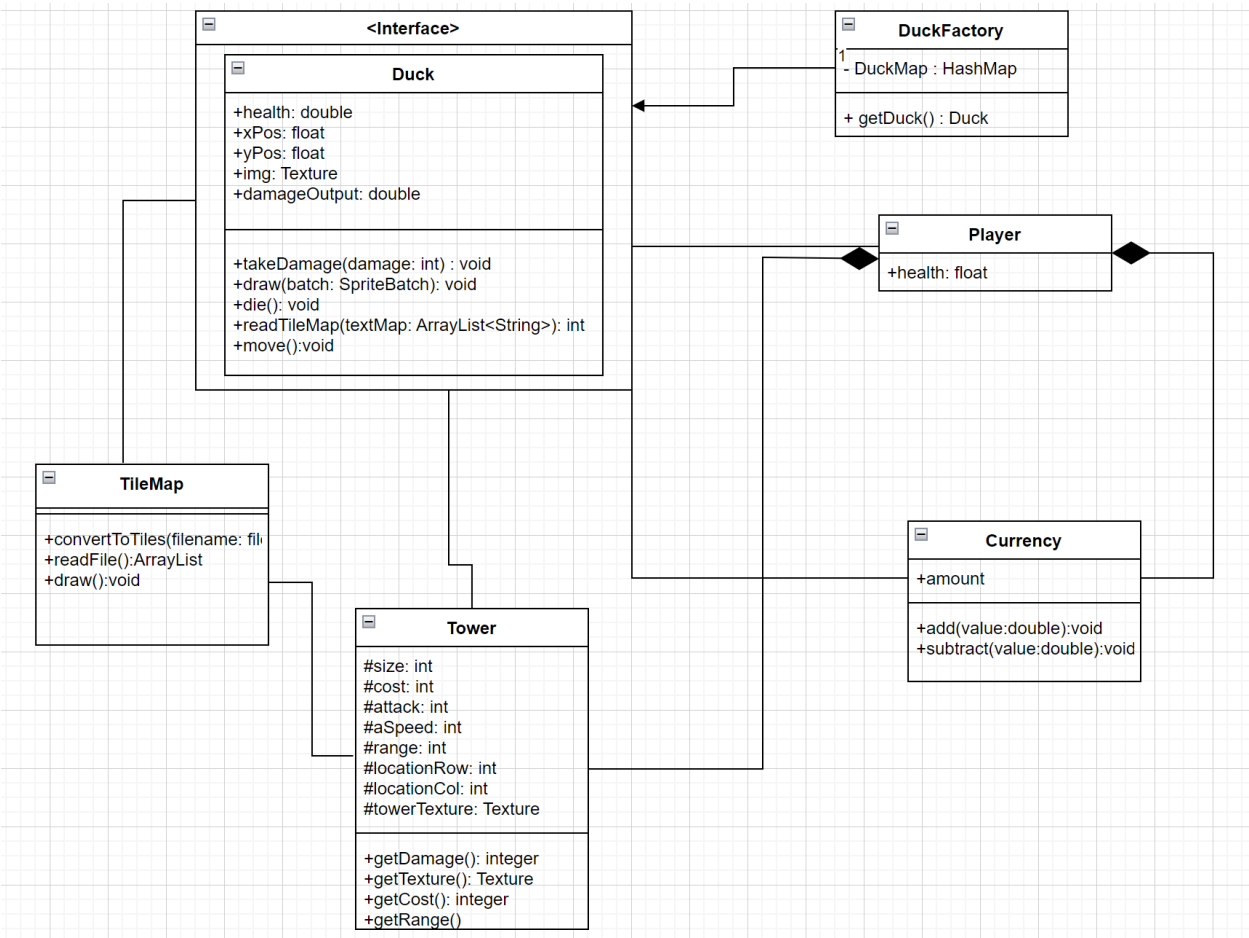
For my project, my group and I are creating a tower defense game where we will destroy ducks that will act as our enemies. As such there will be a large amount of duck objects that will need a lot of space to manage. However, these ducks share common traits such as sprites, health, damage capable of inflicting, starting position, path, etc. These traits will all be part of the intrinsic state. By using the Flyweight Design Pattern, I can greatly decrease the amount of space allocated by the ducks.

Ultimately, the Flyweight Design Pattern decreases the amount of space a program requires by having similar objects share certain characteristics. By using this Design Pattern, I can greatly decrease the amount of space required by my project. Essentially allowing the game to run much smoother and allow computers with less amount of space to run the game as well. This design pattern would prove to be extremely beneficial for me as well as numerous other programmers.

Works Cited

"Flyweight." *Refactoring.Guru*, https://refactoring.guru/design-patterns/flyweight.

"Flyweight Design Pattern." *GeeksforGeeks*, 1 Sept. 2021, https://www.geeksforgeeks.org/flyweight-design-pattern/.


<Interface>
Duck
+health: double
+xPos: float
+yPos: float
+img: Texture
+damageOutput: double
+takeDamage(damage: int) : void
+draw(batch: SpriteBatch): void
+die(): void
+readTileMap(textMap: ArrayList<String>): int
+move():void
DuckFactory
1
- DuckMap : HashMap
+ getDuck() : Duck
Player
+health: float
TileMap
+convertToTiles(filename: fil
+readFile():ArrayList
+draw():void
Currency
+amount
+add(value:double):void
+subtract(value:double):void
Tower
#size: int
#cost: int
#attack: int
#aSpeed: int
#range: int
#locationRow: int
#locationCol: int
#towerTexture: Texture
+getDamage(): integer
+getTexture(): Texture
+getCost(): integer
+getRange()