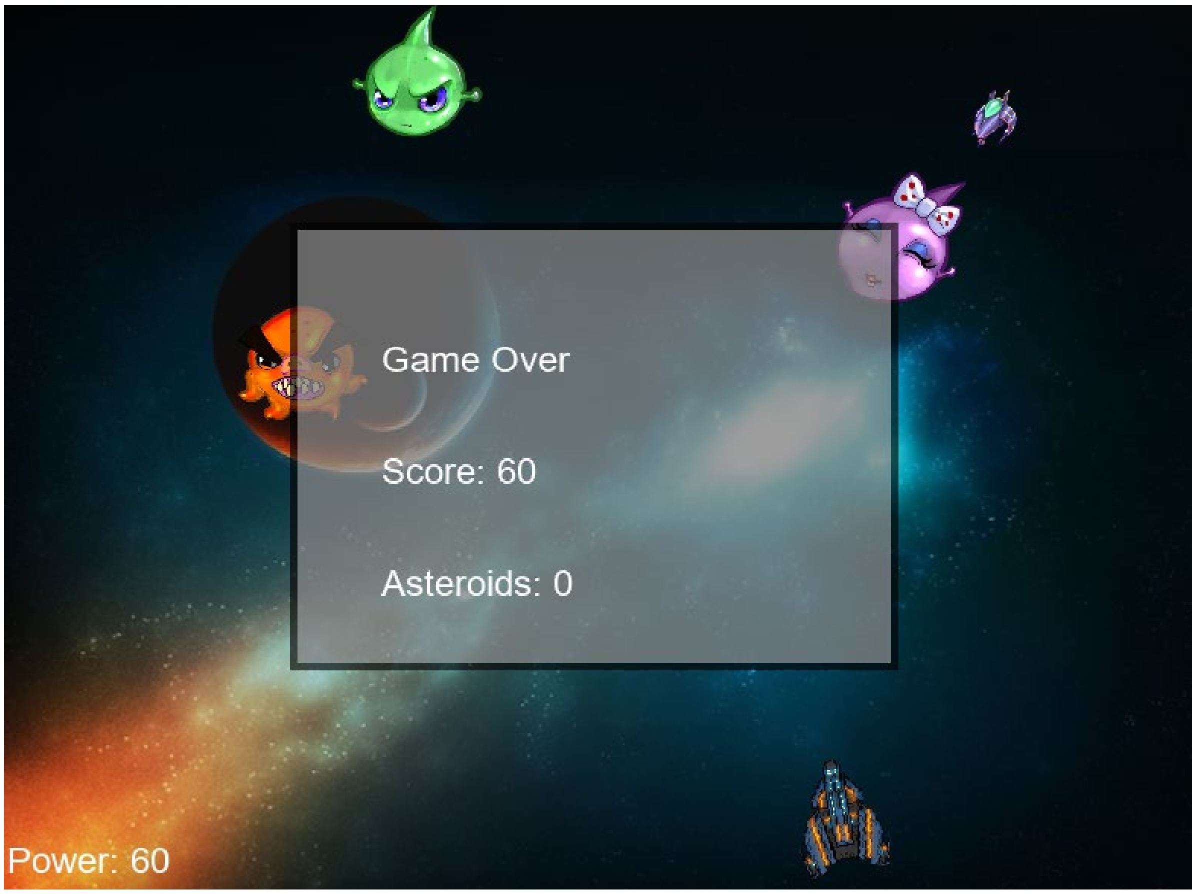
SpaceShooter

A game made by implementing Design Patterns

Team Members: - Akash Nemade
Riya Patel

1. INTRODUCTION: -

Our Spaceshooter game is somewhat different from regular Spaceshooter game. We have added some extra features and have changed the story of the game making it more interesting. The story of our game is that there is a scientist who needs to collect 6 asteroids for his research, so he takes his spaceship and goes into space to search for the asteroids in his space explorations he encounters various kinds of enemies and once he collects 6 asteroids he wins. His spaceship the Spaceshooter can fire laser and ammo that is how the score increases once he hits enemies and collects asteroids. The game is running perfect and targets anyone who wants to play and have fun. It is a 2d game so there are not any specific high requirements. We created this game using Greenfoot visual tools which provides an interface to run our game. Of course, we needed to code the logic but adding graphics was much simpler in this interface. One of the disadvantages of using Greenfoot is the complications to integrate our logic into Greenfoot. One of the key features is to develop the enemy Spaceshooter which would make it more interesting. This could be done using Strategy design pattern (using different algorithm) so that enemy Spaceshooter could counterattack or Command Design Pattern to save the history of current player and use it for enemy Spaceshooter to counterattack.



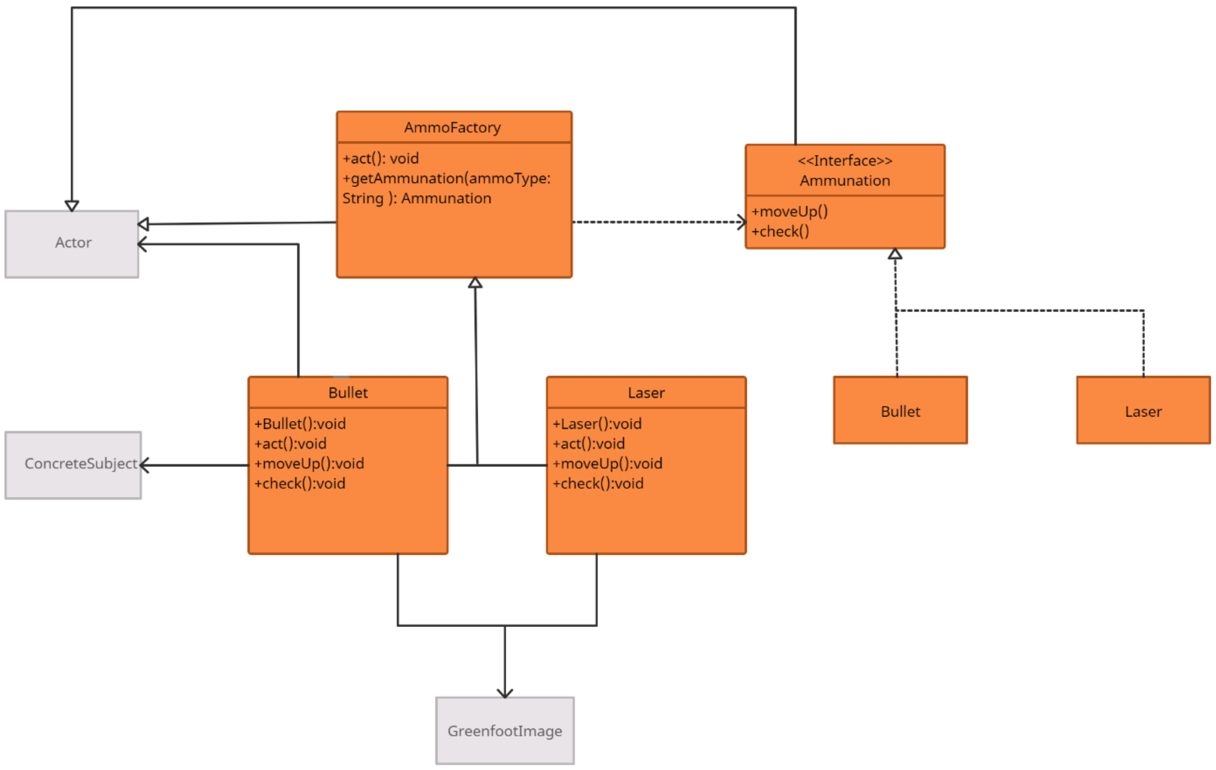
2. System Design: -

We have used four Design Patterns to create the game.

- Factory Design Pattern
- Composite Design Pattern
- State Design Pattern
- Observer Design Pattern

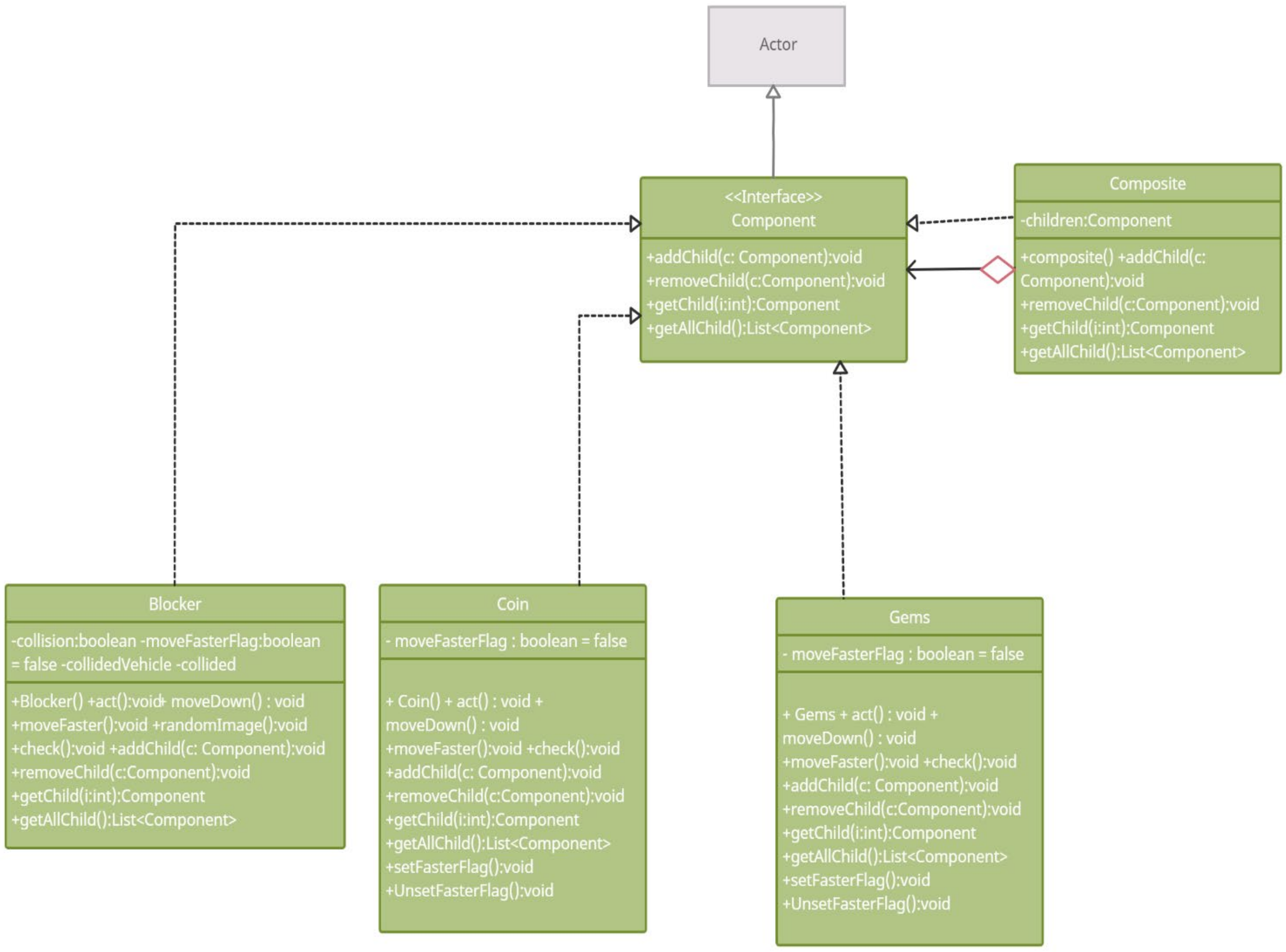
i) Factory Design Pattern: -

It is a creational Design Pattern that provides an interface for creating objects in superclass but allows subclasses to alter the type of objects that will be created. In our game we are using this pattern to create Ammunition. In our case Factory Method in Bullet class returns bullet object whereas the factory method in Laser class returns laser object. The client treats all the ammos as abstract Ammunation. The client knows that all transport objects are supposed to have moveUp and check method, but exactly how it works isn't important to the client. The orange-colored classes are factory method implementation and rest of them are of Greenfoot visual tools.

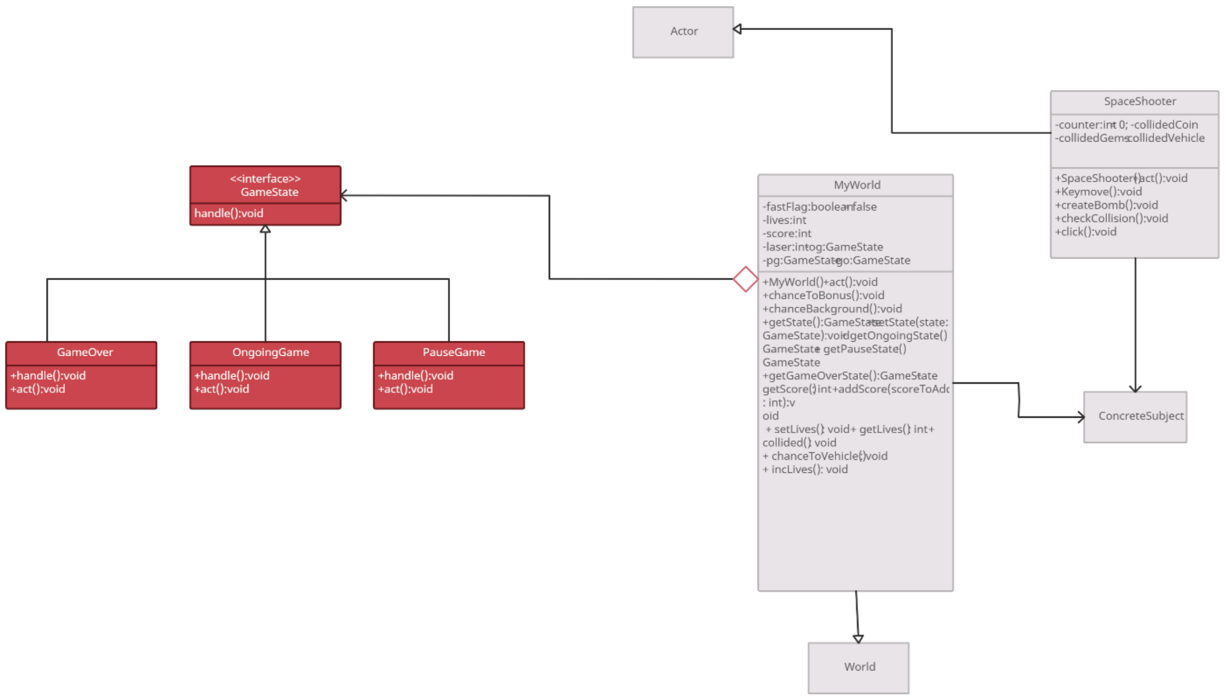


ii) Composite Design Pattern: -

It is a structural Design Pattern that allows you compose objects into tree structures and then work with these structures as if they were individual objects. This design pattern is implemented for coins, asteroids, and enemies(aliens). If any of the given object collides with spaceshooter score increases, life increases, spaceshooter dies or loses a life, respectively. The Component interface describes operations that are common to both simple (Coin and Gems) and complex (Blocker) elements of the tree. Composite is a container which has sub-elements (leaves or other containers). Composite does not know the concrete classes of its children. It works with all sub-elements only via the component interface. The Client works with all elements through the component interface. As a result, the client can work in the same way with both simple (Coin and Gems) and complex (Blocker - responsible for aliens) elements of the tree. The greatest benefit of this approach is that we do not need to care about the concrete classes of objects that compose the tree. We do not need to know whether an object is a simple Coin or a complex Blocker. We can treat them all the same via the common interface. When we call a method, the objects themselves pass the request down the tree.


Actor
<<Interface>>
Component
+addChild(c: Component):void
+removeChild(c:Component):void
+getChild(i:int):Component
+getAllChild():List<Component>
Composite
-children:Component
+composite() +addChild(c: Component):void
+removeChild(c:Component):void
+getChild(i:int):Component
+getAllChild():List<Component>
Blocker
-collision:boolean -moveFasterFlag:boolean = false -collidedVehicle -collided
+Blocker() +act():void+ moveDown() : void
+moveFaster():void +randomImage():void
+check():void +addChild(c: Component):void
+removeChild(c:Component):void
+getChild(i:int):Component
+getAllChild():List<Component>
Coin
- moveFasterFlag : boolean = false
+ Coin() + act() : void + moveDown() : void
+moveFaster():void +check():void
+addChild(c: Component):void
+removeChild(c:Component):void
+getChild(i:int):Component
+getAllChild():List<Component>
+setFasterFlag():void
+UnsetFasterFlag():void
Gems
- moveFasterFlag : boolean = false
+ Gems + act() : void + moveDown() : void
+moveFaster():void +check():void
+addChild(c: Component):void
+removeChild(c:Component):void
+getChild(i:int):Component
+getAllChild():List<Component>
+setFasterFlag():void
+UnsetFasterFlag():void

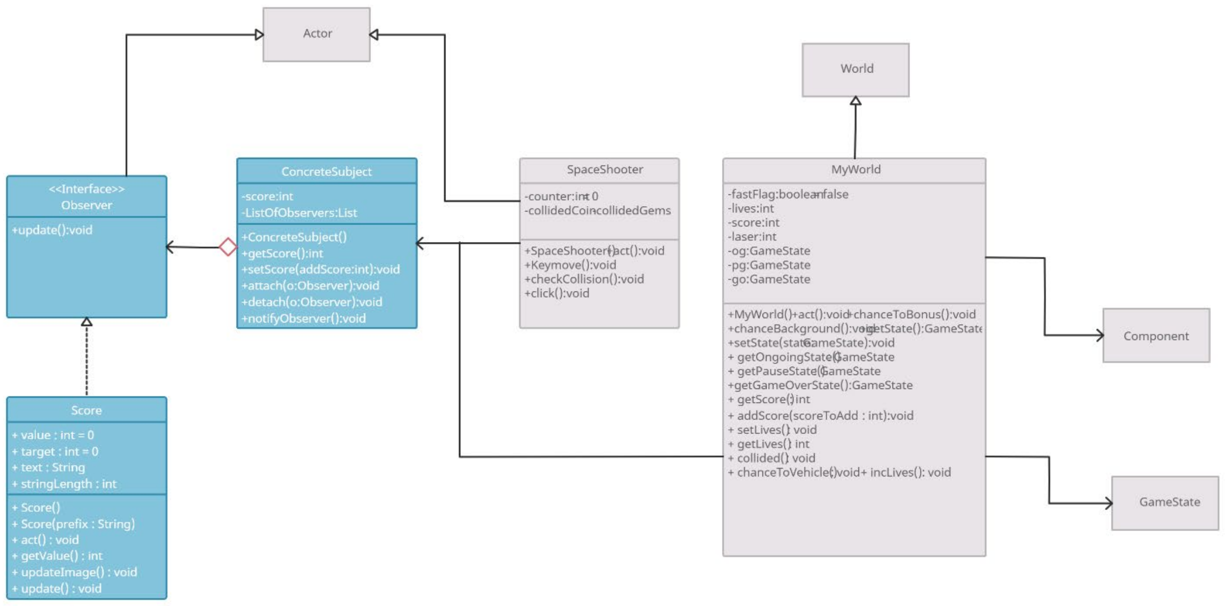
iii) **State Design Pattern: -** State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



As the name suggests this pattern is used for altering the state of the game. Ongoing Game paused game and Game Over are the three states which we alter using this pattern in this game. MyWorld stores a reference to one of the concrete state objects and delegates to it all state-specific work. The MyWorld (context) communicates with the state object via the state interface. The MyWorld (context) exposes a setter for passing it a new state object. The GameState interface declares the state-specific methods. Concrete States provide their own implementations for the state-specific methods.

iv) Observer Design Pattern: -

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they are observing. We are using this design pattern to update the score. The ConcreteSubject class is Observable which issues events to other objects. These events occur when ConcreteSubject changes its state due to collision. (coin spaceshooter or aliens spaceshooter or gems spaceshooter). ConcreteSubject contains a list of Observers that lets new Observers join and current Observers leave the list. When collision happens, the Observable goes over the list and calls the notification method declared in the Observer interface on each Observer object.



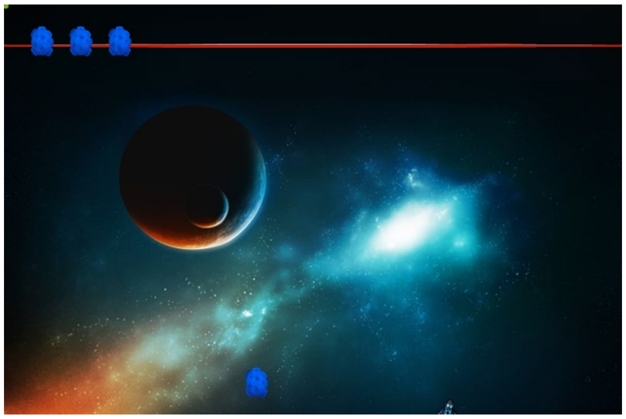
3) System Implementation: -

Features of the game:

i) Ammo

a) Bullet

b) Laser

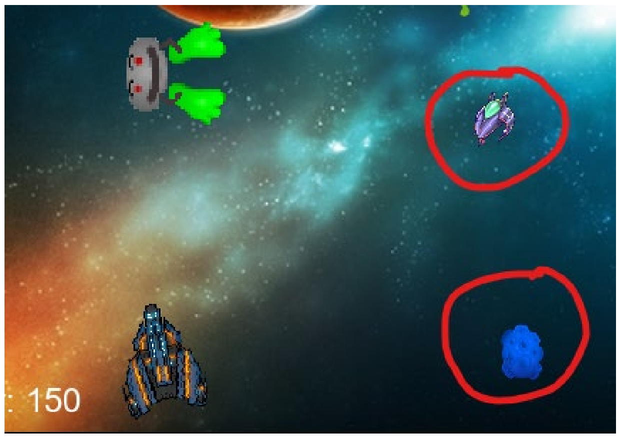


The red line seen is the laser emitted by the spaceshooter to kill all the aliens. When power level reaches 100 Spaceshooter can fire laser.

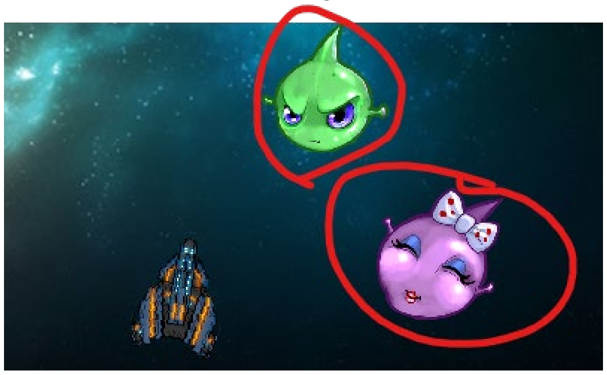
ii) Coins and Gems

Graphics of coin and gem classes have changed to get more space feel.

Image of Gem class is asteroids and image of coin class is a small jet used to increase the power when collected by spaceship.

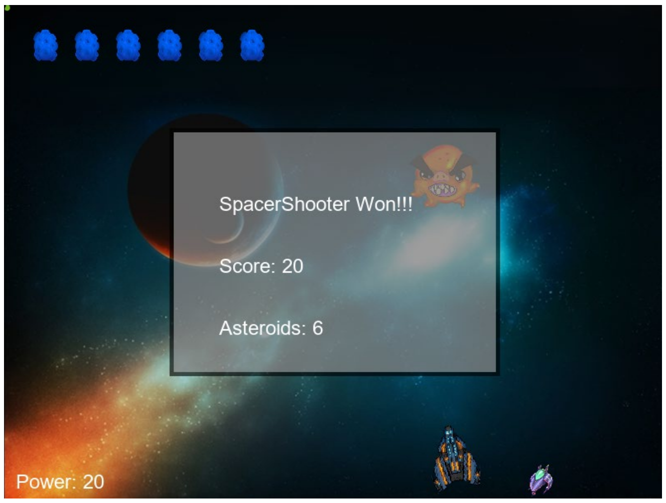


iii) Enemies as aliens in the game



iv) Game Modes

If 6 asteroids are collected player wins the game. By collecting an asteroid player gets a life and colliding with alien, player loses one life if lost all lives then Game Over. As we can see below 6 asteroids are collected so Player (Spaceshooter) wins the game.



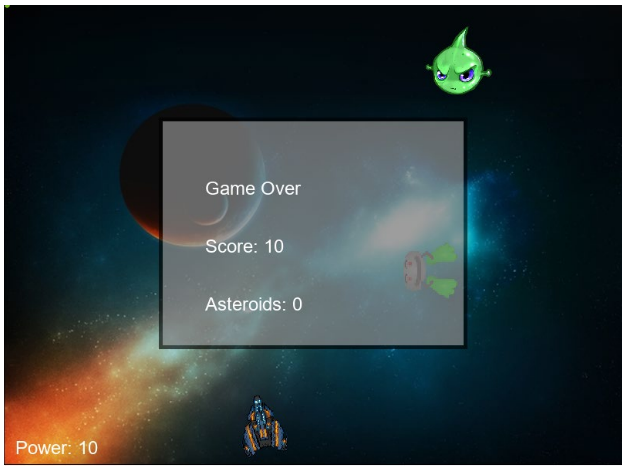
SpacerShooter Won!!!

Score: 20

Asteroids: 6

Power: 20

In below screenshot there are no asteroids remaining so player lost the game. At the bottom we can see power level which is our score which allows us to fire laser once reached 100.



4) Lessons learned:

Riya:
This course of design patterns was a good opportunity to learn Java programming. It was a challenge as well as difficult for me to learn Java, but this course is the reason I got fluent in this programming language. Also, I was not much familiar with design patterns earlier and how can they be used to ease the code implementation and reuse it for better efficiency of program. But I learned about different design patterns and their uses in this course. This is for the first time I have built a game in my life and it was a good experience building a game with my teammate using design patterns.

Akash:
It is fascinating to know that such techniques exist in computer science world. When I say such techniques, I mean Design Patterns. I believe true meaning of education is the ability to think and while implementing design patterns in this game my ability was challenged at each step so trying to solve these challenges made me know a lot of things. I understood the importance of implementing design patterns in a project although it is difficult to design a design pattern initially, but it always helps and saves a lot of time eventually for eg. Adding or removing feature becomes so easy without any hassle. I had no experience with java or any gui programming so in creating this

game I learnt java and Greenfoot visual tools. And lastly never procrastinate to start working on project especially when it is a game.

DemoLink:https://drive.google.com/drive/folders/16iXHSGTsvchKkCsq2xeQ--ygvW8XHD-g?usp=sharing