Module Code: CS2JA16
Assignment Report Title: Java GUI Project
Student Number: 27022145
Date: 23/01/20
Actual Hours Spent: 23
Assignment Evaluation:

- This Assignment has been extremely enjoyable.
- I have learnt a myriad of things throughout this task.
- Most notably I have learnt how to organise large scale applications.

Abstract

This report explains the design and development decisions taken when creating a 2D graphical game. The game has been implemented using the programming language Java, however javafx and css have also been used to help achieve the game like functionality. This report is compromised of several sections, these include a description of the class organisation/hierarchy, a description of the individual classes, a collection of UML diagrams illustrating the program visually, screenshots showing the classes in action, the tests performed, and a final section where the report is concluded.

Initial Design Stage

Before developing this game, it was important that there was a clear plan for how the game would operate and consequently, how the code-base would be structured. Therefore, prior to making the program, I conducted research into what the best game/simulation design would be. After analysing existing simulations, I decided that the game would be a ‘drone shooter game’. Additionally, it was decided that there should be a menu stage, where the user would initiate a game/simulation. Once in the game there should be a singular drone which the user can move, however there should also be other drones, which will interact in various ways with the environment and/or player drone. The user will be able to ‘destroy’ other drones in the arena, all while progressing through various levels of the game. Information on the drone/arena will be displayed on the screen. Specific objects that will be included in the simulation will be a player drone, an enemy drone, a satellite, a space station, a meteor, and an obstacle.

Description of Class Organisation / Hierarchy

As mentioned prior, the programming language used is java, therefore thanks to the languages advanced capabilities, I decided to structure the code-base in an object oriented manner. Object orientated programming is a paradigm based on the concept of “objects," which contain data, in the form of fields or methods. It was decided that an object orientated approach would be beneficial for this project as the simulation is, and will encapsulate objects (these being drones, enemies, obstacles). Using this approach also allowed me to harness features like encapsulation, data abstraction, polymorphism, and inheritance.

Collectively the program is made up of 15 classes, each class representing an object in the simulation, and each performing a myriad of tasks. The general design of the program is derived from the possible views, for example the entire program is managed by two classes, these being ‘MainViewManager’ and ‘GameViewManager’. The MVM manages

the objects within the main menu, and the GVM manages the objects within the main game simulation. Approaching the design this way allowed for the structure to be easily conceptualised before and during development, in addition, this structure ensured optimal readability.

The hierarchy of the class's are as follows, The 'main file' for the program is stored in the 'QuantumVoyage' file, this starts the execution of the program. Associated with the main class are MainViewManager and GameViewManager. Inside GVM is then associated the core game classes, the most important of these being 'Entity,' which is the base abstract class that all game objects inherit from. The abstract class droneObject extends 'Entity,' which in turn is then extended from 'Player'. In addition to this, the 'Bullet' class provides a bullet object, as well as the 'Enemy' class providing an enemy. There are also several other classes including 'Meteor,' 'Satellite,' 'Obstacle,' and 'Portal' which all extend from the abstract class Entity. The final class 'SoundManager' allows for the game sounds to be added and managed. A visual representation of this class structure can be seen Appendix section A. Arranging the classes in this manner allowed for many OOP principles to be harnessed. Inheritance is the core backbone of the program. For example, because 'Player' extends droneObject (which extends Entity), the Player object inherits a multitude of fields and methods. This is also an example of data abstraction, as the Player class does not contain the method bodies, however can still execute the code encapsulated in the method. Polymorphism is also a key technique, static polymorphism is used by the Meteor class when the class uses an overloaded constructor to create a varied object of type Entity. Dynamic polymorphism is also used in the 'Player' class, this is done by overriding a function contained in 'Entity.' This means that when a 'Player' object is created in runtime, it reverts to an overridden method, apposed to its parent class's method.

Description of Individual Classes and Functions

As mentioned earlier, the program's entry point is in the 'QuantumVoyage' class. This class extends 'Application' which allows for JavaFX to be implemented. Inside this class I first declare a MVM object and create a 'new' instance. The two included methods are 'main' and 'start,' the 'main' function calls the 'start' function, which subsequently initialises the stage to the value of the function, 'getStage'. This function is associated with the MVM class, it's return value is of type 'Stage'. Finally the 'show' function Is used, this attempts to show the window by setting the visibility to true.

The corresponding class MVM now takes over control. Inside this class is first declared several fields. These include a StackPane, a Scene, and a Stage. In the MVM class constructor these fields are then initialised. Notable methods used in the constructor are 'screenWidth,' and 'screenHeight,' these methods return a double value corresponding to the height/width of the screen that the game has been loaded onto. I decided to use these methods as it allowed for the window to be dynamically sized to the optimum

height and width, instead of determining a fixed size which may be perfect for one screen, but too small/big for another. The first of the final two methods executed In the constructor sets the background of this scene. And the second method creates the buttons to be displayed on the main menu. When created, each button is added to the arrayList, ‘buttons’. These buttons are also added to a ‘vBox,' and the vBox is then added to the StackPane. It was decided to use a vBox for the sole reason that a vBox object allows you to centre it inside the screen. In order to manage these buttons, I employ the added function ‘handleButtonAction’ which perpetually listens for a button click that corresponds to any button in the button ArrayList. Encapsulated inside these conditions, is code to create a ‘new game,' to ‘load game,' and to ‘exit game’. We will first discuss the execution associated with creating a new game. Firstly, the GV object previously declared is instantiated, the screen dimensions are passed into the GVM constructor. The method ‘createNewGame’ is then called, taking the parameter stage (being the menu view stage). Finally the gameLoop method is called which will be discussed next.

Instantiating the GVM object first triggers the constructor. Inside the constructor several fields are instantiated. Most notably using the parameters passed in, the boundaries of the arena are set in the form of maxUp, maxDown, maxLeft, and maxRight. A collection of methods are then called which initialise the environment. Key methods include the creation of backgrounds, game information, player objects, enemy objects, and a key listening function. The method used to switch the screen from the MV stage to the GV stage is named ‘createNewGame,' as mentioned earlier this is called in the MV, essentially this function method hides the MV, and shows the GV.

Certain objects are by default, always loaded, hence there ‘create’ functions are nested inside the class constructor. For example a ‘Player’ object is always created, this is done by calling the function ‘createPlayer’. This method consequently creates an object of type Player, and adds it to the current pane. The private function ‘createKeyListeners’ monitors for ‘KeyEvents’, and if a particular key is pressed, executes its corresponding code. In this case pressing either W,A,S,D will move the player drone. This movement is handled in the ‘Entity’ class. A notable function to complement the KeyListener is the onKeyReleased method, this adjusts the players speed/velocity when keys are released. This combination of functions services a smooth simulation of the player drone.

The entire class’s function is controlled/modulated by the ToolBar menu which holds buttons corresponding to a single action. These actions include adding enemy drones, satellites, meteors, obstacles, cannons, portals, and several more objects. Therefore when the button is pressed, said button, calls the corresponding function to create the object. An example of this can be explained using ‘createSatellite’ method. When the ‘create new satellite’ button is pressed, this function is called, which instantiates an ‘Entity’ object whilst passing in three parameters to configure the image, and size of the object. Details of this class will be discussed later in the report, however, to briefly outline,

this class creates an object of a size given in the parameters, and then translates the object to a random position in the window. The object is then added to both the current ‘gamePane’ and the ‘satellites’ array list.

In order to move objects such as the satellite, the ‘moveSatellite’ method is used. Essentially this method iterates through the objects arrayList, and for each object, calls the method ‘moveSatellite,' which is associated with the ‘Entity’ class. This function, as for all ‘move’ functions is included in the method ‘gameLoop’ which runs an AnimationTimer, this is a loop that executes 60 times per second, thus executing the move methods thousands of times a minute, resulting in a smooth animation. The AnimationTimer loop also allows for the player drone to fire bullets. This is achieved by calling the ‘fireBullet’ method when the space bar is pressed. This method first creates a new object of type ‘Bullet,' this bullet object is then ‘initialised,' this gives the Bullet class information associated with the position of the player in the arena, the velocity of the player, and the rotation of the player. The bullet is then added to the bullet ArrayList. Finally the method ‘updateBullets’ (which is executed repeatedly inside of the timer), updates each bullet in the bullets ArrayList to the described next position.

Also encapsulated in this class are methods that determine how the objects interact with each other. For example when the player and a satellite collide, or when a satellite and a obstacle collide, there needs to be a true physics reactions. To achieve this function, a method ‘checkAllCollisions’ is included inside the animation timer. This methods encapsulates several for loops, these for loops iterate though all entities in the object array lists, if any object touches another object, a corresponding reaction is initiated. To detect if two objects touch each other, the ‘intersects’ method is used, this is a built in method that evaluates this condition relative to the ‘Bounds’ of said objects. In some cases this reaction is to destroy the object, however in others it is to ‘bounce’ the object. This implementation allows for the player drone to fire bullets and destroy enemies, while enemies/satellites also being able to bounce of objects in the environment.

As well as offering a rich simulation, the simulation can follow a game like set of rules. This is done by adding a ‘portal’ to the scene (when a minimum amount of enemies have been killed) , which when the player goes through, transports the player drone to the next level, where more enemies/obstacles are added. This function is implemented by executing the ‘createPortal’ function, only when the ‘currentKills’ is equal to the ‘desiredKills’, once this condition has been met, the ‘desiredKills’ variable is incremented by 5. When the player enters the portal, all obstacles/enemies are removed, a method ‘setNewBackground’ is called, and an increased amount of enemies are re-spawned in the new level. This loop is continued until the player drone runs out of health, at this point the gameLoop is stopped, and you are presented with your game statistics.

Entity should be seen as the base class from which all on screen objects extend. Encapsulated within ‘Entity’ is several fields and methods which allow for a basic object

to be quickly created. Entity is declared as an abstract class that all objects derive from. In the class constructor, using the supplied parameter variables, an image is assigned to the object, in addition to a height and width. From this size, the radius of the object is then calculated for future reference. Using the randomX and randomY methods in the class, the position of the entity is set. In addition to this constructor, there is also an overloaded constructor, this is designed to initialise an object with slightly different characteristics. All other object class's, including obstacle, meteor, portal and satellite, extend this abstract entity class. This allows each class to inherit different methods, and take on different characteristics held in Entity, however while still maintaining the same parent class.

The remaining methods in the class body are tasked around controlling/changing the position of an entity. For example there are methods such as getRotate, setRotate, setVelocity ect. Methods like rotateLeft and moveForward use these functions to update the players position based of current information like velocity, rotation, and speed. Another notable method in this class is 'isInArena,' this method contains a set of conditions that if met, perform an action. In this case if the entity has hit the edge of the boundary, stop the drone. This results in the containment of the entity within the game arena. The final method in this class is 'getCenter,' this method is extremely valuable, and used regularly, by using several built in functions, the x and y values of the centre of the object are returned as a Point type, allowing me perform more accurate calculations on the object.

Another class used heavily is 'Droneobject,' this is an abstract class that extends Entity. This class is used for all drone-like objects in the arena, therefore it was decided to make the class abstract to allow for both 'Players' and 'Enemies' to extend the class. The droneObject class inherits from entity as all droneObjects are entities. Inside of the class constructor, several values are instantiated, these include the image value, the size of the drone, the health of this individual drone, and the amount of 'kills' this drone has. This class also has the function 'getNode' which returns the node of this object. Due to the fact that this class extends the base class entity, there are a minimal amount of methods, however all extended droneObjects have access to these.

Having now discussed the parent classes, we can understand the child classes. The class 'Player' extends droneObject, as all players are droneObjects. By extending droneObject, the 'player' class inherits all functions associated with drone movement. Inside the class constructor the super class constructor is called, providing the object with an image and size. Other fields are also instantiated including the speed of the drone. Despite inheriting all entity/droneObject methods, it was required to 'override' a function that is found in the 'entity' class. This method is overridden to change the way a player object navigates, in comparison to other entities. This overrides method can be seen in Appendix A.

The class ‘Enemy’ configures the characteristics and movement of enemy objects. This class extends circle. Meaning that in the constructor I first call the super method, thus determining the size and colour of the circle. The circle is then translated to a random position within the arena. Inside this class is several key methods, the most important of which being ‘moveEnemy,' this function takes three parameters, one being the enemy object, and the final two being the x and y position of the player. These parameters are then passed into a path method which move the enemy object towards the player. This moveEnemy function is called frequently in the main gameLoop, resulting in a smooth animation of the enemies.

The ‘Bullet’ class allows for bullet objects to be initialised by the player. The core method ‘init’ contained in this class is called in GVM when the space bar is pressed, this method first translates the bullet to the position of the player, the bullet then uses the ‘setRotate’ function to rotate the bullet in the direction that the player is facing. The velocity of the player is then assigned to a field within the class. As discussed previously the movement of these bullets is handled by the GVM method ‘updateBullets’ which increments the position of each bullet according to its rotation and velocity. The combination of these methods creates a rapidly fireable player gun. As explained the bullet collisions are handled in the GVM

The final class used in the program is ‘SoundManager,' this class manages all the sounds used in the game. Methods included in this class each manage there own sound, for example the method ‘gunSound’ declares a string address to the unsound mp3, and then passes this into the ‘playSound’ method, which plays the sound.

Analysis of Key Functions

The first notable feature of this program would be the smooth movement of the player. This is achieved by an overridden function ‘moveUp,' this method simply moves the player forward according to a ‘speed’ variable, if this speed variable is high, the drone will travel fast, and if the speed variable is low, it will travel slow. When the user presses ‘W,' the players speed is increased, thus moving the player forward. When the player is rotated, the ‘rotSpeed’ field is either increased or decreased, this results in the player rotating. In order for this to function correctly, there is also an event handler that reduces the speed and rotation variables back to zero, when all buttons are released.

The next notable function of my program is the ability for all object drones to be kept within the bounds of the arena. This is achieved by implementing the function ‘isInArena’ which is located in the ‘Entity’ class. This method is passed five parameters, these being maxUp, maxDown, maxLeft, maxRight, and radius. Inside the function, if the said entity goes over the arena bounds, reset the position to the max bounds. When applied to all corners, this method restricts all objects from leaving the arena. The animation of the enemies is also a key feature. This is achieved by repeatedly calling the method

‘moveEnemy’, which initialises a path transition object, this object plots a path toward the inputted coordinates. When updated in succession, this creates a smooth animation of the enemies.

One feature which all objects use is an entities ability to adjust, and understand its speed, rotation, and velocity. This is allowed by using the object Point2D, this object holds two points, in this case it represents the velocity of the object. Whenever any movement is conducted on a drone, the velocity of the object is quickly updated. This serves as a base, from which all other measurements can be derived.

UML Diagrams

UML diagrams can be seen at Appendix B . Both two diagrams are class diagrams, the first being an abstract version, and the second providing more information on the methods, constructors inside.

Explanation of Interface / User Manual

As can be seen from the included screenshots (Appendix A), the game simulation has a rich number of possible views and included options. There are two ways in which the program can be used, one is in a pure simulation, and the other is in a game survival way. The survival mode of this game works by the player drone having to avoid all enemies, and shoot at least 10 enemies, at this point, a ‘portal’ will open, when entered the ‘next level’ will be reached, this changes the background image, adds more enemy drone objects, and increases the speed of the drones. This cycle continuous up until the point that your health (displayed in the top right) runs out, which is where the ‘GAME OVER’ text is displayed and the simulation is stopped. In order to return to the main menu, the user must press the ‘Esc’ key. The program can also be used a simulation, this is aided by the toolbar menu, this menu allows the user to add enemy drones, obstacles, satellites, space stations, and meteors. In addition to this by pressing ‘Add Info’, information can be displayed on each individual drone. And lastly the simulation can be ‘paused’ and ‘played’ using the toolbar buttons. From the main menu, previously saved games can also be retrieved by pressing the ‘Load Game’ button.

W —> Move Forward	**Esc** - Return Home
A —> Move Left	**Space** —> Shoot
S —> Move Back	**R** — Open Toolbar
D — Move Right	**T** — Close Toolbar

Testing

In order to ensure maximum efficiency, and reliability, the game/program was put through several tests. It is vitally important to test code for subtle vulnerabilities that could cause serious errors.

The first stage of testing was ‘basic functionality’ testing, this involved exploring the interface, and making sure that there were no obvious errors with the program. Upon conducting this first level of testing I discovered that there were several buttons which did not have any actions linked to them, I then discovered that this was due to a mix up with indexing. The second stage of testing was ‘code review,' this is where a fellow software engineer looked over my code base. This step did not reveal any functional errors, however it did help to clean up the formatting and layout of the code. Static code analysis was then completed. To do this I employed the IDE IntelliJ, this software tool performs static tests on your code. The results of such tests can be seen in Appendix B. The final, and most important test stage was then completed, this was ‘unit testing’. In order to perform the unit tests I used testFX, this is a java testing framework that has been optimised for JavaFX. Results of the unit tests can be seen below.

Test Description	Expected Output	Actual Output	Comment
Drone Collision Test	No exceptions Found	1 Exception Found	Satellites do not collide with Player drones.
Out Of Bounds Drone	No exceptions Found	0 Exceptions Found	Drone is constrained within the canvas.
GUI Performance Test	No errors found	Performance exceptions found.	Non fixable.
Serialisable Test	No Errors	No Errors	

Discussion / critical analysis

If I was to re-do this project, there would be several things done differently. First of all, I would have designed the enemy movement’s from the ground up, this is instead of using existing JavaFX methods. By building animation code from the ground up, I would have had a greater scope of control, this would have created a richer simulation.I also would have designed the components with the expectation that they needs to be serialised. When coming to serialise all the classes towards the end of the project, I encountered several errors which required the structure of the code to be refactored. This caused a significant delay, and could have been avoided. In contrast however, I beileve my frequent use of version control and Kanban software allowed myself to effectively manage deadlines.

Conclusion

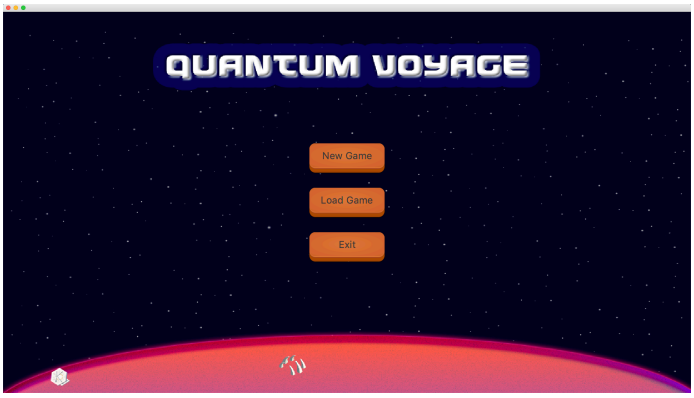
To sum up. Through harnessing OOP via the language of Java, a multi-functional graphical user interface game has been created. This game has been fully tested, and thus proven to be free of any major bugs or errors. The final product is rich with features, these include the ability to control a drone object in an arena, shoot at other drone objects, save arena configurations, and progress through game levels which automatically increase the difficulty of the game. The game also has the ability to act as a 'simulator', this is controlled by the toolbar, inside the toolbar are buttons that can be pressed to add enemies, various other objects, list information on each drone, and pause or play the simulation. Despite there being several structural inconsistencies, the program operates efficiently, an excessive amount of objects on the screen does not effect the performance of the program.

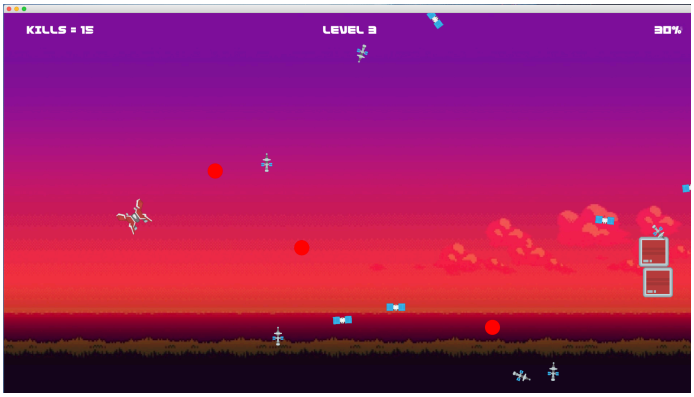
Personal Reflection

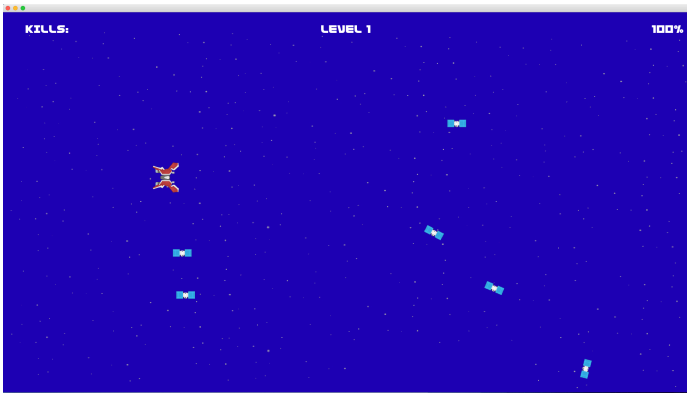
In reflection, I have enjoyed the process of designing and then developing this program. As mentioned earlier there are several things that I think could have been done different. One massive thing that I have learnt from this process Is how to structure large and complex javafx files. When first starting to write this program, the file structure and arrangement of classes was extremely messy. However, upon learning more about how javafx works, I began to understand how the program should be structured. I have also massively increased my understanding of how to effectively test your program. This is an extremely important component of the software development cycle, I now feel very confident I can write test cases for a wide variety of programs.

Appendix

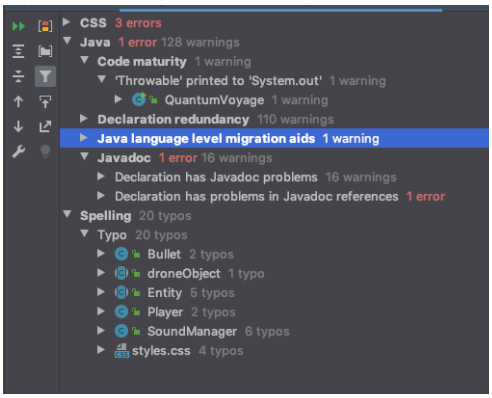
(A)

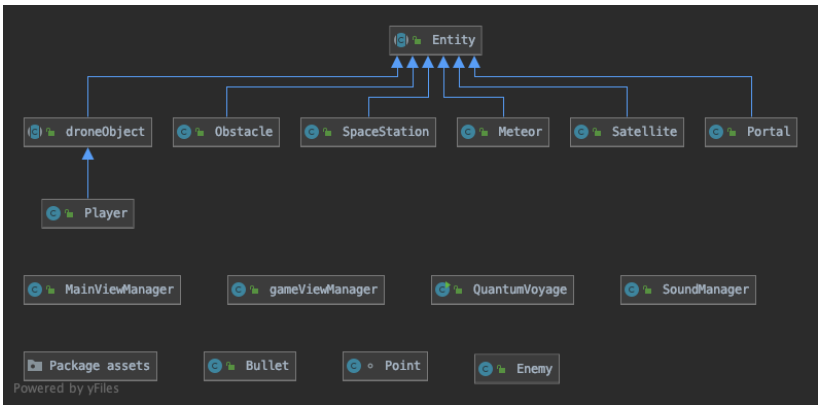




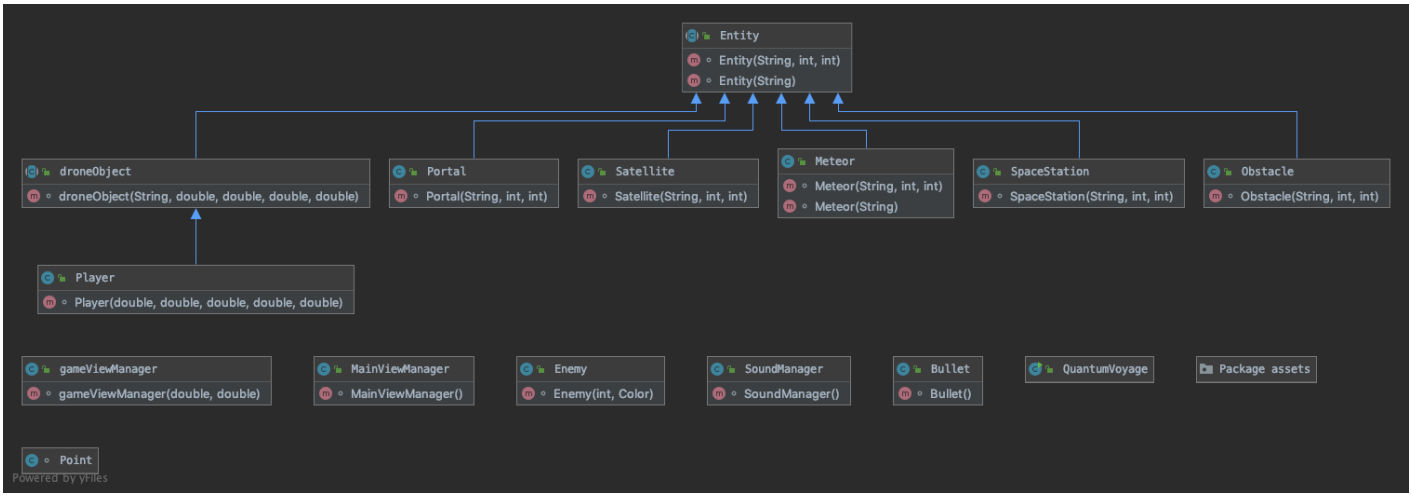

KILLS:
LEVEL 1
100%


KILLS = 28
LEVEL 4
0%
GAME OVER
KILLS :28


CSS 3 errors
Java 1 error 128 warnings
Code maturity 1 warning
'Throwable' printed to 'System.out' 1 warning
QuantumVoyage 1 warning
Declaration redundancy 110 warnings
Java language level migration aids 1 warning
Javadoc 1 error 16 warnings
Declaration has Javadoc problems 16 warnings
Declaration has problems in Javadoc references 1 error
Spelling 20 typos
Typo 20 typos
Bullet 2 typos
droneObject 1 typo
Entity 5 typos
Player 2 typos
SoundManager 6 typos
styles.css 4 typos


Entity
droneObject
Obstacle
SpaceStation
Meteor
Satellite
Portal
Player
MainViewManager
gameViewManager
QuantumVoyage
SoundManager
Package assets
Bullet
Point
Enemy
Powered by yFiles

(B)


Entity
Entity(String, int, int)
Entity(String)
droneObject
droneObject(String, double, double, double, double)
Portal
Portal(String, int, int)
Satellite
Satellite(String, int, int)
Meteor
Meteor(String, int, int)
Meteor(String)
SpaceStation
SpaceStation(String, int, int)
Obstacle
Obstacle(String, int, int)
Player
Player(double, double, double, double, double)
gameViewManager
gameViewManager(double, double)
MainViewManager
MainViewManager()
Enemy
Enemy(int, Color)
SoundManager
SoundManager()
Bullet
Bullet()
QuantumVoyage
Package assets
Point
Powered by yFiles

(C)