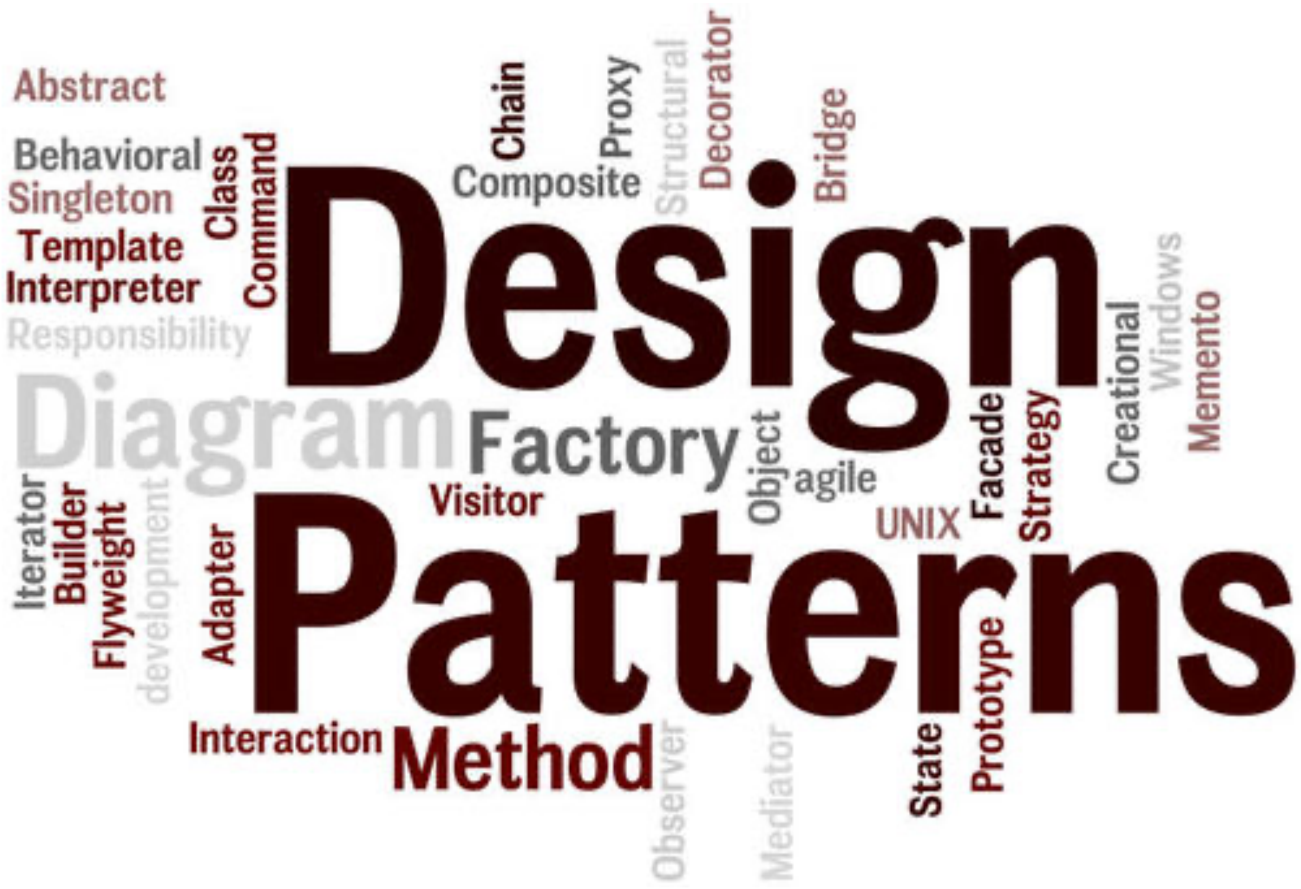


A javaFX app utilising Design Patterns

Ayub Aden Ahmed

@00596190

AGD861

Set by Dr Ian Drumm

TABLE OF CONTENTS

Executive Summary ***3***

Introduction ***3***

Design ***3***

game Design **3**

uml class Diagram **3**

Use Case Diagram **5**

Design Patterns ***5***

MVC (model view controller) **6**

Factory **7**

Singleton **8**

Evaluation ***8***

References ***8***

EXECUTIVE SUMMARY

"Building object-oriented software is difficult, and designing reusable object-oriented software is considerably more difficult," according to (Gamma, Helm, Johnson and Vlissides, 1994). Design patterns are reusable solutions to reoccurring challenges (Drumm, 2022) that software developers face during the development process. These solutions were discovered through trial and error by software engineers over a long period of time. The usage of patterns improves productivity and performance while also making object-oriented software design and development easier. In this report, I'll show my understanding of design patterns and how I applied them to my project. And to demonstrate thoughtful 'responsibility driven design', I've included a use case diagram, and an uml class diagram.

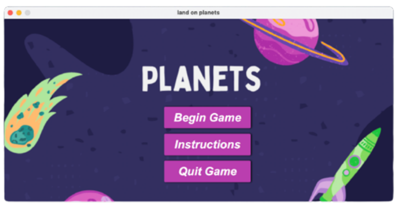
INTRODUCTION

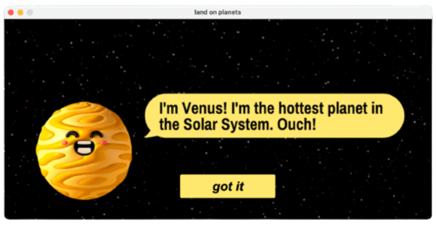
The purpose of this project is to create a single interactive game that utilises design patterns to teach primary school children about science in a colourful and enjoyable way. To create the game, I used eclipse, maven, and javaFX2. Factory, Singleton, and MVC were the design patterns used in this game. Each one will be discussed in greater detail in subsequent sections.

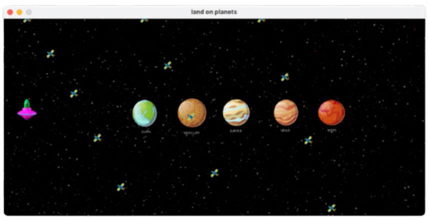
DESIGN

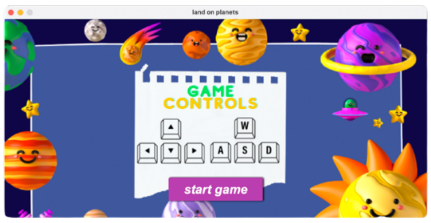
GAME DESIGN

The game's design and development took a long time. The goal of the game is to arrive on a planet without colliding with satellites. When you arrive on a planet, you will be given information about it. I made sure the information is presented in a simple and child-friendly style so that the kids could understand and even memorise it. I created the game in a child-friendly and entertaining way. The design of the game can be seen below.




land on planets
I'm Venus! I'm the hottest planet in the Solar System. Ouch!
got it





UML CLASS DIAGRAM

The class diagram below depicts an overview of the game, including its classes and their relationships. Designing the class diagram helped me see the application's functioning and grasp the many classes needed and their relationships. It also made game development simple and quick.

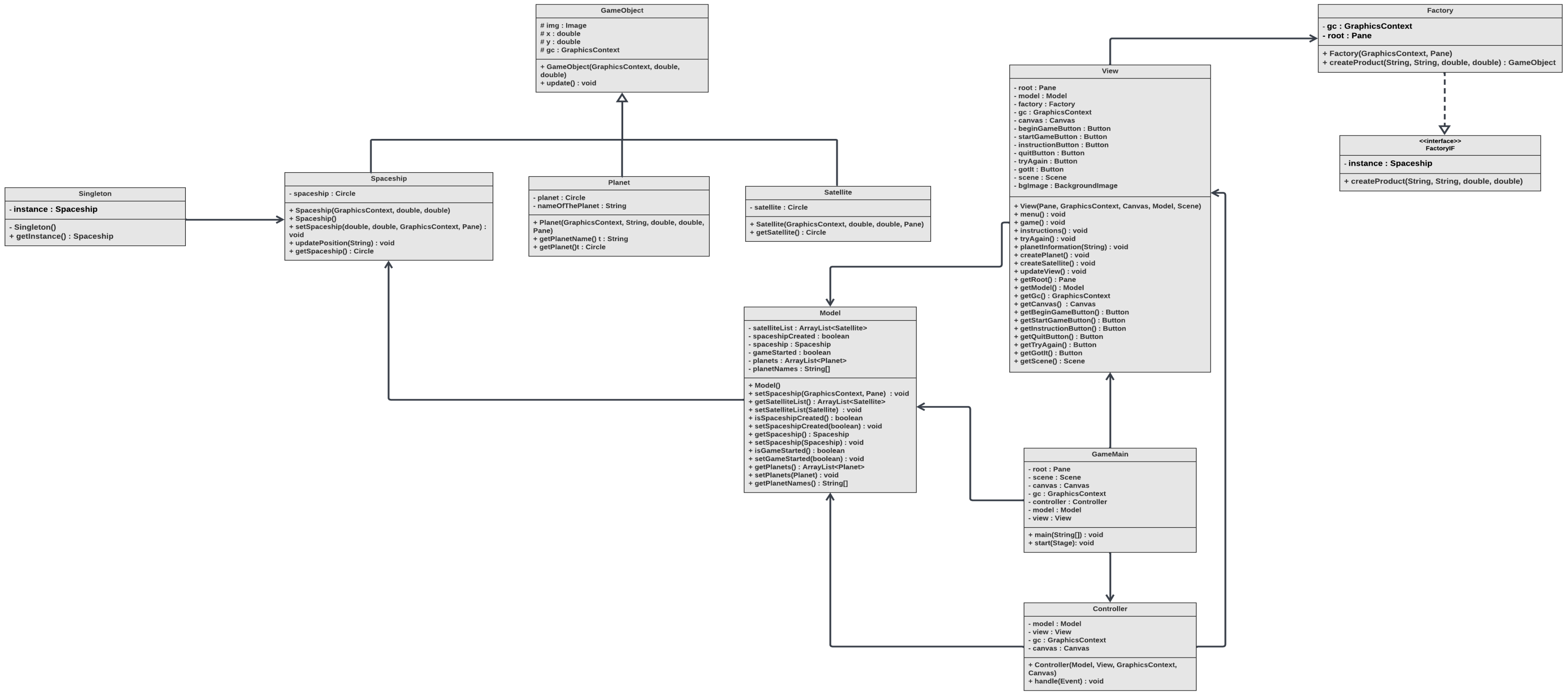

GameObject
# img : Image
# x : double
# y : double
# gc : GraphicsContext
+ GameObject(GraphicsContext, double, double)
+ update() : void
Factory
- gc : GraphicsContext
- root : Pane
+ Factory(GraphicsContext, Pane)
+ createProduct(String, String, double, double) : GameObject
<<interface>> FactoryIF
- instance : Spaceship
+ createProduct(String, String, double, double)
View
- root : Pane
- model : Model
- factory : Factory
- gc : GraphicsContext
- canvas : Canvas
- beginGameButton : Button
- startGameButton : Button
- instructionButton : Button
- quitButton : Button
- tryAgain : Button
- gotIt : Button
- scene : Scene
- bgImage : BackgroundImage
+ View(Pane, GraphicsContext, Canvas, Model, Scene)
+ menu() : void
+ game() : void
+ instructions() : void
+ tryAgain() : void
+ planetInformation(String) : void
+ createPlanet() : void
+ createSatellite() : void
+ updateView() : void
+ getRoot() : Pane
+ getModel() : Model
+ getGc() : GraphicsContext
+ getCanvas() : Canvas
+ getBeginGameButton() : Button
+ getStartGameButton() : Button
+ getInstructionButton() : Button
+ getQuitButton() : Button
+ getTryAgain() : Button
+ getGotIt() : Button
+ getScene() : Scene
Singleton
- instance : Spaceship
- Singleton()
+ getInstance() : Spaceship
Spaceship
- spaceship : Circle
+ Spaceship(GraphicsContext, double, double)
+ Spaceship()
+ setSpaceship(double, double, GraphicsContext, Pane) : void
+ updatePosition(String) : void
+ getSpaceship() : Circle
Planet
- planet : Circle
- nameOfThePlanet : String
+ Planet(GraphicsContext, String, double, double, Pane)
+ getPlanetName() t : String
+ getPlanet()t : Circle
Satellite
- satellite : Circle
+ Satellite(GraphicsContext, double, double, Pane)
+ getSatellite() : Circle
Model
- satelliteList : ArrayList<Satellite>
- spaceshipCreated : boolean
- spaceship : Spaceship
- gameStarted : boolean
- planets : ArrayList<Planet>
- planetNames : String[]
+ Model()
+ setSpaceship(GraphicsContext, Pane) : void
+ getSatelliteList() : ArrayList<Satellite>
+ setSatelliteList(Satellite) : void
+ isSpaceshipCreated() : boolean
+ setSpaceshipCreated(boolean) : void
+ getSpaceship() : Spaceship
+ setSpaceship(Spaceship) : void
+ isGameStarted() : boolean
+ setGameStarted(boolean) : void
+ getPlanets() : ArrayList<Planet>
+ setPlanets(Planet) : void
+ getPlanetNames() : String[]
GameMain
- root : Pane
- scene : Scene
- canvas : Canvas
- gc : GraphicsContext
- controller : Controller
- model : Model
- view : View
+ main(String[]) : void
+ start(Stage): void
Controller
- model : Model
- view : View
- gc : GraphicsContext
- canvas : Canvas
+ Controller(Model, View, GraphicsContext, Canvas)
+ handle(Event) : void

Figure 1 Class diagram for the game

USE CASE DIAGRAM

A primary actor is shown on the left-hand side of the diagram in the diagram below. There is no secondary because the game does not have one. The game is represented by the box in the middle, while the ellipses indicate the user's actions within the game. Use case diagrams are used to show all of the game's actors and Use Cases, as well as how they interact with one another.

Creating a use case diagram allowed me to see the game's purpose and what it was supposed to accomplish. It allowed me to concentrate on the game's most essential aspects.

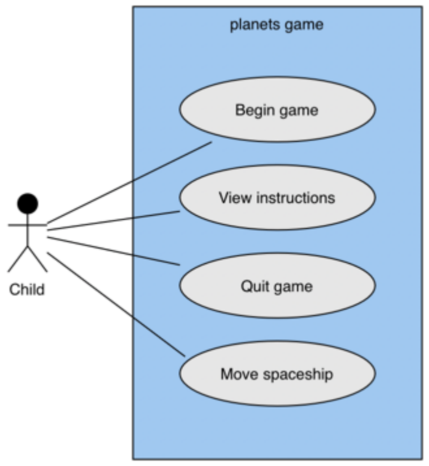


Figure 2 use case diagram for the game

DESIGN PATTERNS

Design patterns are reusable solutions to common challenges in software development (Drumm, 2022). The "Gang of Four" (Gamma, Helm, Johnson and Vlissides, 1994) published a book called Design Patterns: elements of reusable object-oriented software in 1994. It described 23 patterns based on the concept of patterns in software design. These patterns were not created by the authors (Shalloway & Trott, 2001). Rather, they identified patterns existed in at least three actual software systems. An American-Austrian-British architect Christopher Alexander and his colleagues may have been the first to suggest utilising a pattern language to design structures and towns (Gamma, Helm, Johnson and Vlissides, 1994). Alexander's work was discovered by these Gang of four developers. They questioned if the same was true for software design patterns as it was for architectural patterns.

There are four essential elements in a pattern.

1: Name: the pattern name is a one- or two-word description of a design problem, its solutions, and results. A pattern name is a set of descriptive words that we can use to communicate with others, document them, and reference them in software designs. A pattern always has a name (Shalloway & Trott, 2001)

2: Context: the problems the pattern handles. (Shalloway & Trott, 2001) “It's crucial to describe explicitly the problem being handled, which is the reason for having the pattern in the first place, even if it may seem obvious at times”.

3: Solution: a general-purpose solution to a problem. The solution explains the relationships between objects and classes, as well as their responsibilities and collaborations. A pattern is a template that may be used in a variety of scenarios; therefore, the solution does not specify a particular implementation.

4: Consequences: These are the drawbacks and advantages of using the patterns. It enables

developers to better comprehend and evaluate them.

Design patterns are categorised in three groups when it comes to their goal or purpose. These three groups are:

1: Creational patterns: Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented (Gamma, Helm, Johnson and Vlissides, 1994). Creational patterns concentrate on the object-initialisation process. Creational patterns involve object instantiation, and all provide a way to decouple a client from the objects it needs to instantiate (Freeman & Freeman, 2004). Some examples of creational patterns include singleton, builder, factory method, and prototype.

2: Structural patterns: the focus of structural Design patterns is to structure objects and classes while maintaining the flexibility and efficiency of these structures. Structural class patterns use inheritance to compose interfaces (Gamma, Helm, Johnson and Vlissides, 1994). Some examples of structural patters are proxy, adapter, flyweight, bridge, and decorator.

3: Behavioral patterns: Concentrates on how the objects and classes interact with each other to fulfil their duties. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them (Gamma, Helm, Johnson and Vlissides, 1994). Some examples of behavioral patterns are strategy, state, visitor, template method, iterator, and command.

Patterns can also be classified by their scope, or whether they apply to classes or objects.

Object: some examples of object scope patterns are singleton, proxy, strategy, and builder

Class: some examples of class scope patterns are adapter, factory method, and template method.

Below are the design patterns used in the project.

MVC (MODEL VIEW CONTROLLER)

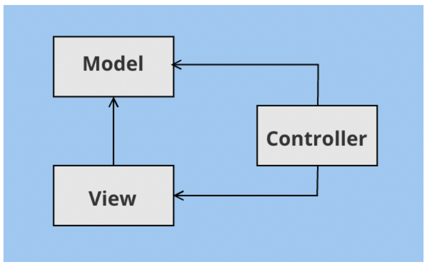
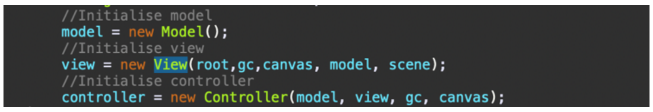


Figure 3 mvc

The Model-View-Controller (MVC) pattern separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes (Drumm, 2022). figure 3 describes how mvc works. Data is accepted, processed, and returned by the model. There are no references to the view or the controller in the model. The user interface is defined by the view. The model is visible to the view. The view and model are visible to the controller. Parallel development is possible with

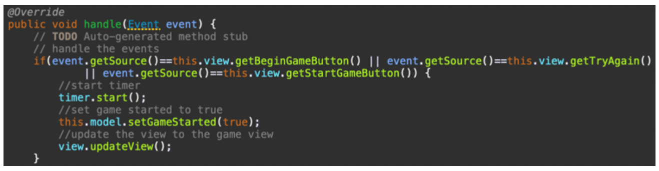
MVC. If used, the software can be decoupled such that one developer can work on the view while the other works on the controller.



```
    //Initialise model
    model = new Model();
    //Initialise view
    view = new View(root,gc,canvas, model, scene);
    //Initialise controller
    controller = new Controller(model, view, gc, canvas);
```

Figure 4 initilising the three mvc objects

I have used mvc throughout the project. As seen in figure 4, I am creating the three objects needed for mvc. The Model which contains all the data of the game and methods to access and modify those data, the View which manages the display of data, and the Controller to handle user inputs such when user clicks on a button on the screen.



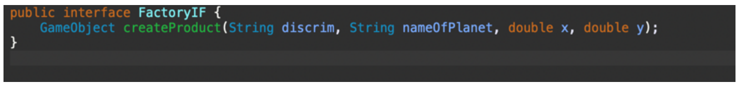
```
@Override
public void handle(Event event) {
    // TODO Auto-generated method stub
    // handle the events
    if(event.getSource()==this.view.getBeginGameButton() || event.getSource()==this.view.getTryAgain()
            || event.getSource()==this.view.getStartGameButton()) {
        //start timer
        timer.start();
        //set game started to true
        this.model.setGameStarted(true);
        //update the view to the game view
        view.updateView();
    }
```

Figure 5 controller handling user button clicks

As seen in figure 5 my controller class is handling user events for example when the user clicks a button on the screen the controller and updates the view.

FACTORY

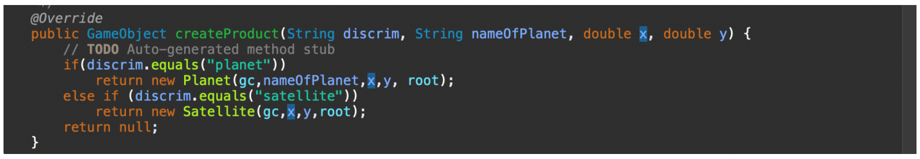
The Factory design pattern is a creational object-oriented design pattern that allows us to create objects. It implements the code that allows objects to be created in the same way that products are created in factories. When a project has a superclass with several subclasses and one of those subclasses needs to be instantiated, Factory can be utilised. As described by (Drumm, 2022), “A class that can create instances of other classes without depending on them. The reusable class delegates the creation to another object and accesses it via a common interface”.



```
public interface FactoryIF {
    GameObject createProduct(String discrim, String nameOfPlanet, double x, double y);
}
```

Figure 6 factory interface

The factory interface is depicted in Figure 6. There is an abstract method in the interface that of course has no implementation. The Factory class (Figure 7) implements the Factory interface, which also implements the abstract method.



```
@Override
public GameObject createProduct(String discrim, String nameOfPlanet, double x, double y) {
    // TODO Auto-generated method stub
    if(discrim.equals("planet"))
        return new Planet(gc,nameOfPlanet,x,y, root);
    else if (discrim.equals("satellite"))
        return new Satellite(gc,x,y,root);
    return null;
}
```

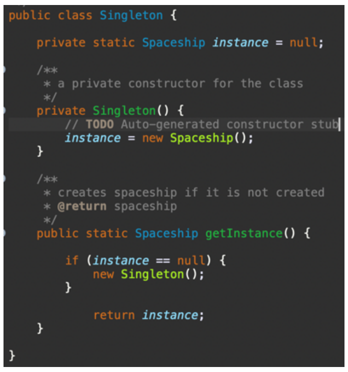
Figure 7 factory class implementing factory interface abstract method

The factory class implements the abstract method of the factory interface, as seen in figure 7. The method returns an object based on the discrim or string provided. For example, if it is provided with “planet” it will return a new planet object. All objects created by this method are all children of the game object class or they all extend the game object class.

Because there were multiple types of classes (planets, satellites) that were all sub-classes of the game object, the factory method proved really useful for this game.

SINGLETON

"Ensure a class only has one instance, and provide a global point of access to it" (Gamma, Helm, Johnson and Vlissides, 1994). Singleton is a creational pattern that offers one of the most effective way of creating an object. The singleton design allows us to only have one instance of a class at any one given time. Constructor are made private. As a result, we are unable to create objects outside of the class.



```java
public class Singleton {

    private static Spaceship instance = null;

    /**
     * a private constructor for the class
     */
    private Singleton() {
        // TODO Auto-generated constructor stub
        instance = new Spaceship();
    }

    /**
     * creates spaceship if it is not created
     * @return spaceship
     */
    public static Spaceship getInstance() {

        if (instance == null) {
            new Singleton();
        }

            return instance;
    }

}
```

Figure 8 Singleton class

As shown in figure 8, I have used single pattern for the spaceship. As there is only one spaceship needed at one given time throughout the game. I have made the contractor private. The constructor initialises the static instance. When the get instance static method is called, it creates a new spaceship instance if one doesn't exist. If one does exist, it just returns a pointer to that instance.

EVALUATION

While the project was tough, it provided me with valuable experience. I was able to learn about design patterns and experience the benefits and cons of various patterns. Despite the project's complexity, the usage of design patterns and uml diagrams like use case diagrams and class diagrams made the game development process simple and controllable. Overall, I had a great time making the game.

Throughout the game, I made good use of OOP. For example, I've made the fields private and used getters and setters to access and modify them(encapsulation). I've also followed correct naming conventions and given meaningful names to everything.

Making the satellite move around is one improvement I'd like to make, as it would make the game more challenging and enjoyable. The satellites are currently stationed in one location, making it incredibly simple to pass through them and arrive on a planet. another improvement I would like to make is in terms of design patterns, I think using behavioural patterns would be a nice addition to this game, given I haven't utilised one.

REFERENCES

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1994. Design patterns. 1st ed. Addison Wesley.

Drumm, I. 2022. Design patterns lecture slides. University of Salford

Shalloway, A., & Trott, J. (2001). Design patterns explained. Addison-Wesley.

Freeman, E., & Freeman, E. (2004). Head first design patterns. O'Reilly.