2024

Escape the Planet: Revolutionizing Game Design with Novel OOP Techniques

Qusai Kamal Fannoun
Minnesota State University, Mankato

Escape The Planet

By

Qusai Fannoun

A An Alternate Plan Paper Submitted in Partial Fulfillment of the

Requirements for the Degree of

Masters of Science

In

Information Technology

Minnesota State University, Mankato

Mankato, Minnesota

April 2024

April 12th, 2024

Escape the Planet

Qusai Fannoun

This Alternate Plan Paper has been examined and approved by the following members of the student's committee.

John C Burke
Advisor

Flint Million
Committee Member

Naseef Manseor
Committee Member

Table of Contents

List of Figures

List of Tables

Escape the Planet

Qusai Fannoun

AN ALTERNATE PLAN PAPER SUBMITTED IN PARTIAL FULFILLMENT OF
THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

MINNESOTA STATE UNIVERSITY, MANKATO
MANKATO, MINNESOTA
APRIL, 2024

ABSTRACT

Mobile devices are continuously evolving and greater computing power and graphics capabilities are being introduced every year. As a result, there is an increasing demand for challenging and engaging mobile games that leverage these advanced features. This project explores best design practices using the development of Escape the Planet, which is an intricate maze game for mobile devices in which players navigate using a spaceship that is trapped in a hostile planet's maze while avoiding obstacles and enemy attacks. The goal is to safely guide the spaceship out of the maze without colliding into walls or taking bullets from defensive cannons. Players can use shields and collect power-ups such as extra fuel or shield boosters to help their chances but must demonstrate skill, strategy, and patience to succeed. With multiple maze configurations, escalating difficulty levels, collectible rewards, and threatening enemies, Escape the Planet aims to provide an exciting, suspenseful gameplay experience challenging enough to appeal to hardcore gamers. The inspiration comes from observing the expanding mobile games market revenue, especially for ports of popular console/PC game franchises once considered too computationally intensive for mobile. By focusing on efficient coding, complex gameplay elements, and leveraging newer device capacities, this project utilizes the full potential of modern mobiles for an engaging, high-quality game worthy of monetization. With sufficient polish and testing, Escape the Planet will be released on the Android store and hopefully to the iOS store to contribute an innovative title to the maturing mobile gaming landscape.

1. Introduction

1.1. Evolution of Mobile Gaming

In this modern world, people are witnessing revolutionary technology with extraordinary performance capability. Among these advancements, mobile phones are a prime example of innovation that has drastically transformed our daily lives. Nowadays, smartphones offer greater computing power and advanced graphics capabilities. As a result, the landscape of mobile gaming has expanded significantly, which attracts a broader audience. According to the Games - Worldwide | Statista Market Forecast (2023), the game industry has witnessed significant revenue growth during the past few years. This evolution has caught the attention of business owners such as Netflix, Ubisoft, and many others. They are considering mobile platforms as effective marketing and revenue-generating platforms.

1.2. Project Overview

Escape the Planet is a mobile game app intended to be released for Android devices, featuring a variety of challenges and entertaining gameplay and will illustrate best design practices according to industry. The game includes multiple levels with increasing difficulty. Each level presents a unique maze, offering a new challenge to players. Also, the players must avoid obstacles that move around, including defensive cannons that shoot at the spaceship. Colliding with any of these obstacles will destroy the spaceship, which means the level will be restarted from the beginning. Players can collect power-ups, such as shields and fuel collectibles, throughout the game to enhance their spaceship’s capabilities.

1.3. Technical Innovation

Escape the Planet utilizes advanced object-oriented programming (OOP) techniques, ensuring efficient coding and maintainable projects. The game builds on best practices discussed in the "Design Patterns: Elements of Reusable Object-Oriented Software," "Heads First," and "C# Design Patterns" books. The development process could not be more enjoyable. Gamma et al. (1994), Freeman et al. (2013), and Bisho's (2007) contributions to the field of software development have been incredible sources that discussed multiple design patterns, such as abstract factory, command, observer, adaptor, and many more patterns that solve design issues. Which allow the implementation of new game features to be integrated more efficiently and effectively. The goal of the project is to explore the best practices that a developer could adopt to improve the code and keep it maintainable.

In conclusion, Escape the Planet is not just a showcase of technical skill. It is a demonstration of object-oriented principles that help preserve a program. These techniques provide a solid structure that allows the developer to extend the project in the future without going through unreliable code, as it puts up a robust structure that can be modified with no trouble.

1.4. Problem Statement

Developing a native mobile app would allow these apps to utilize the full potential of the mobile device. The general problem is that mobile devices are small computers with compact components, which provide technical challenges to overcome performance issues (Ahmad et al., 2017; Arnomo et al., 2021). The specific problem is that the memory in

mobile phones is limited, and the CPU could be easily overloaded. Therefore, that could overheat the devices as they do not contain cooling mechanisms and cannot cool off, causing the mobile device to perform poorly (Arnomo et al., 2021).

2. Literature Review

2.1 Game Development History

The video game development history goes back to the 1950s when a physicist called William Higinbotham created the first video game, a rudimentary tennis game on an oscilloscope (*Early History of Video Games - Wikipedia*, n.d.; *This Month in Physics History - October 1958: Physicist Invents First Video Game*, 2008). However, video games started gaining popularity in the 1970s and 1980s, when arcade video games, gaming consoles, and home computer games were introduced to the general public (*Early History of Video Games - Wikipedia*, n.d.).

2.2 Mobile Gaming Market

Mobile gaming is the dominant sector of the video game industry. *Figure 1* highlights the gaming market revenue among different platforms, and by looking at mobile phones, the dominance is almost 50% of the entire gaming market, reaching almost $93 billion in revenue in 2022 (Knezovic, 2023). The growth of mobile gaming has emerged from the widespread expansion of mobile phones and the new capabilities they have introduced (*Gapminder Tools*, n.d.). The proliferation of smartphones and advancements in mobile technologies have encouraged developers to develop sophisticated, high-quality

games that can be played instantly from users' pockets wherever and whenever the user wants.

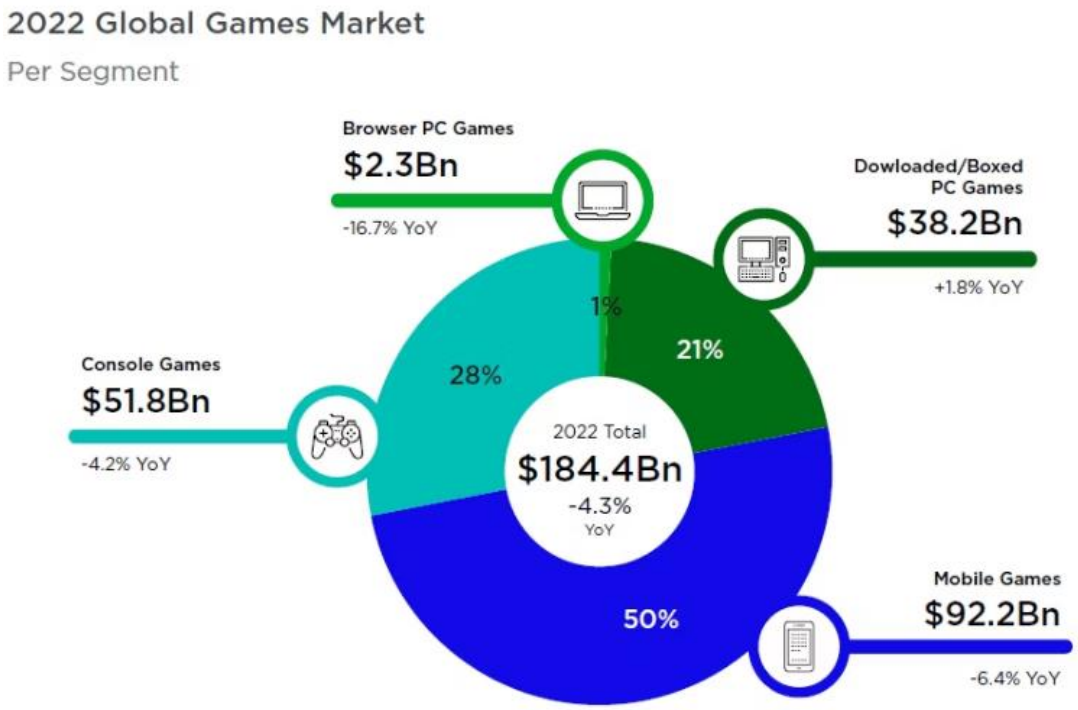


Figure 1. Mobile gaming revenue statistics for 2022 (Retrieved from Knezovic, 2022).

2.3 Mobile Game Development Challenges

While the exponential advancement of mobile technologies has brought new innovations in mobile gaming, developers still face notable challenges and technical difficulties in building games for smartphones and tablets. A key challenge arises from the diversity of mobile devices (Szeja, 2023). The wide variety of screen sizes, display resolutions, aspect ratios, and hardware components like CPU and GPU must be considered during development. Games must preserve extensive optimization and testing across various devices to provide a consistent experience. Control limitations of touchscreens compared to controllers with multiple buttons and keys also pose interface and gameplay

design constraints (Arabaine, 2023). Additional challenges arise from battery life restrictions, as mobile games drain power significantly faster than other apps (Mitchell, 2023). As a result, the developers must carefully balance performance and energy efficiency.

2.4 Game Engines

The concept of game engines was first introduced by ID Software in the mid-1990s when they introduced a first-person shooter game called Doom. ID Software was able to separate the game's core components, such as the collision detection system, the audio system, and the graphics renderer system, into their own independent systems. Developers have recognized this separation of the game components and allowed them to reuse these components to create new games with different visuals and mechanics with minimal changes and in the cheapest way possible (Christopoulou & Xinogalos, 2017; Gregory, 2018).

Gregory (2018) stated that when other developers started using and modifying these components to create new games, the engine's creators started charging them for using these components by making engine licensing agreements to help themselves and start gaining revenue. Nowadays, game engines are complex and complicated tools with multiple utilities that provide a solid and robust environment to create game mechanics. They are the core components or tools for developing a game for different platforms. For example, if a developer wants to make a character jump or fly, they do not need to work on the whole functionality and physics of jumping or flying because the game engine already

has this functionality created. The developer could use these built-in mechanics and customize them to make the character perform the desired action.

Multiple companies around the world invest millions in making their own private game engines, such as EA Games, RockStars, CD Project Red, and many others, to have their own customized mechanisms and functionalities. These companies do not allow individuals to use their engines. As a result, developers tend to use some of the free-to-use game engines, such as Unity, Unreal, Godot, CryEngine, etc. Unity and Unreal are two game engines that allow developers with a limited budget to explore their talents. To understand the capability of these engines, some creative and innovative games, such as Rust, Angry Bird, Temple Run, Fortnite, Batman: Arkham, and many more other games, have been developed using the powerful tools provided by these engines (Drake, 2023; *List of Unreal Engine Games | Fandom*, n.d.).

Despite being free to use, Unity and Unreal engines provide their own unique and distinct features. Šmíd (2017) compared these two game engines by going in depth about each engine's functionality, highlighting the engines' distinct features and weaknesses. Christopoulou and Xinogalos (2017) covered a broader range of open-source game engines that are best used for developing a mobile game. The authors organized their data in a tabular form, making their comparison easier to follow and understand. However, their comparison was mainly based on each engine's documentation and included various options that could make a choice harder.

Table 1 provides a simple comparison between both engines highlighted by Christopoulou and Xinogalos (2017) and Šmíd (2017).

Table 1. Unity vs. Unreal

Feature	**Unity**	**Unreal Engine**
Programming Language	C#	C++
Primary Use	Indie games, mobile, VR/AR	AAA titles, high-end graphics
Graphics	Good, with support for various styles	Excellent, with advanced visual fidelity
Ease of Use	User-friendly with a gentle learning curve	Slower learning curve but provides powerful tools
Pricing Model	Free tier available, subscription for Pro	Free to start, royalty after revenue threshold
Community and Support	Large community, extensive tutorials	Large community, professional-grade support
Performance	Optimized for a wide range of devices	High performance, especially in complex scenes
Supported Platforms	Support a broader range of platforms.	Focused mainly on the big platforms.
Asset Store	Extensive library of assets and tools	High-quality assets, especially for graphics

2.5 Design Patterns

The design patterns implemented in this project are based on the selected books. Going through these books has provided a general understanding of each implemented

pattern in the game. Unfortunately, the examples provided by “Design Patterns: Elements of Reusable Object-Oriented Software” are illustrated in C++, while “Heads First” is in Java programming language. However, that does not prevent applying the idea to any programming language (Unity Technologies, 2022). On the other hand, “C# Design Pattern” covered the programming language used in Escape the Planet, which helped understand the implementation in Unity as it was directly relevant to the context.

Even though these three books are considered great resources for understanding advanced object-oriented techniques, they are not explicitly used for modern game engines such as Unity. Therefore, several sources, such as tutorials, blogs, and documentation, that focus on implementing design patterns in real-world applications were checked and analyzed to find the best practices for implementing each pattern in a game project.

Table 2 provides a summary of the primary key features of each book to help provide a deeper understanding of each book’s context and traits.

Table 2. Summary of the primary used sources

Book	**Main Traits**	**Advantages**	**Disadvantages**
Gang of Four	It provides a foundational explanation of design patterns and covers 23 patterns divided into creational, structural, and behavioral. The book examples are implemented in	It provides a deep theoretical foundation. The patterns can be adapted for game development architecture and design, offering long-term benefits in understanding software design principles.	It does not cover applications related to Unity or game development. Also, the boos require an effort to understand the concepts and the implementation of the patterns.

	C++ programming language.		
Head First Design Patterns	It provides an introductory guide to design patterns, using a visually rich format and an engaging, conversational style to make complex concepts accessible. The book examples are implemented in Java programming language.	The book is suitable for beginners. The concepts can be quickly grasped and applied to Unity game development, making understanding and implementing design patterns in game projects easier.	The book might feel like a childish book. It may not cover the depth of some patterns as they apply to complex game development scenarios. The book is more suitable for beginners.
C# Design Patterns by Judith Bishop	It focuses on implementing design patterns in C#, providing practical examples, and demonstrating the language's capabilities for software development.	The book directly applies to Unity development, focusing mainly on C#. Also, it offers practical examples that can be easily translated into game development practices, enhancing code structure and reusability.	Specific to C# and may not cover design patterns outside those applicable to C# programming, it could limit exposure to broader design pattern applications in game development.

One of the chosen sources is "Level up your programming with game programming patterns," an eBook by Unity Technologies (2022) covering various game development aspects, including best practices in C# programming. The eBook referenced the "Design Patterns: Elements of Reusable Object-Oriented Software" book, making it an invaluable resource. It explains the concept of each pattern by tailoring it to fit the context of game development. The document included an explanation for the observer, state, command,

singleton, and some other patterns with code samples and templates that anyone could use as a reference. The document provided simple images, as shown in *Figure 2*, explaining each pattern's concept without including complex UML diagrams as they are directly related to game development.

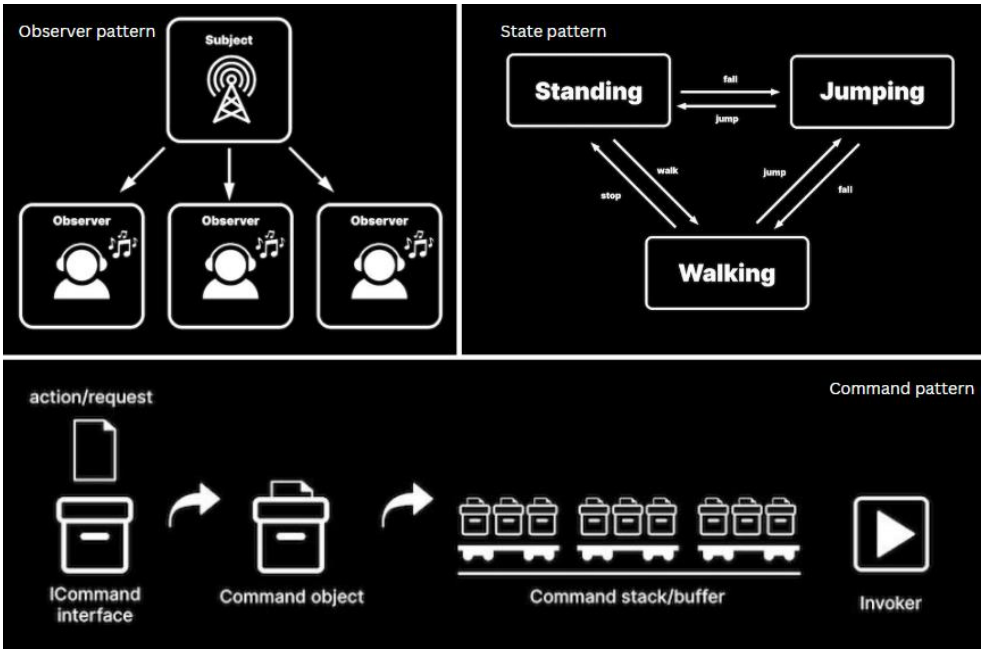


Figure 2. Concept explanation (retrieved from Unity Technologies, 2022*)*

The e**B**ook by Unity is a reliable free resource to help understand design patterns used explicitly for game development in C# due to using Unity-specific examples. Unfortunately, the documentation lacks some design patterns that might be helpful in the scope of game development. Additionally, the document does not show how to assign the scripts to the correct object in the editor window, which might lead to confusion.

As for another source, Tutorials Point is an online website that was taken into consideration. It provides a wide range of tutorials and explanations for multiple

programming languages, including C#, C++, Java, Python, and many more. The website provides well-organized documentation for each supported programming language, from basic to advanced programming techniques. The website also includes well-documented tutorials for over thirty design patterns, from the definition of each pattern to the implementation steps. Even though Tutorials Point supports multiple programming languages, it uses only Java to explain the implementation of the design patterns.

2.6 Design Patterns Used

2.6.1 Introduction

Design patterns provide better solutions to problems in object-oriented software design. They capture expert approaches to designing flexible, reusable systems by identifying class roles and critical relationships. The "Design Patterns: Elements of Reusable Object-Oriented Software" book categorized patterns into creational, structural, and behavioral categories (Gamma et al., 1994). Creational patterns handle object creation, structural patterns deal with class composition, and behavioral patterns address communication between objects. Examples include the Singleton, which restricts the instantiation of a class to one instance, and the Observer, which allows event-based communication between objects. Therefore, using advanced design patterns would improve the quality of the software code, aiding in better comprehension and maintenance (Antoniol et al., 1998; Nikolaeva et al., 2019).

Table 3. Design patterns type comparison

Pattern Type	Purpose	Focus	Examples

Creational	It focuses mainly on object-creation mechanisms and aims to create objects that suit the situation.	Object creation	Singleton, Abstract Factory, Builder, Prototype, Factory
Structural	It explains how to assemble objects and classes into larger structures while keeping the structures flexible and efficient.	Class or object composition	Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
Behavioral	Define how objects interact in a way that increases flexibility in carrying out communication.	Object interaction and responsibility	Observer, Strategy, Command, State, Template Method, Iterator, Mediator, Memento, Chain of Responsibility, Visitor

2.6.2 Singleton Design Pattern

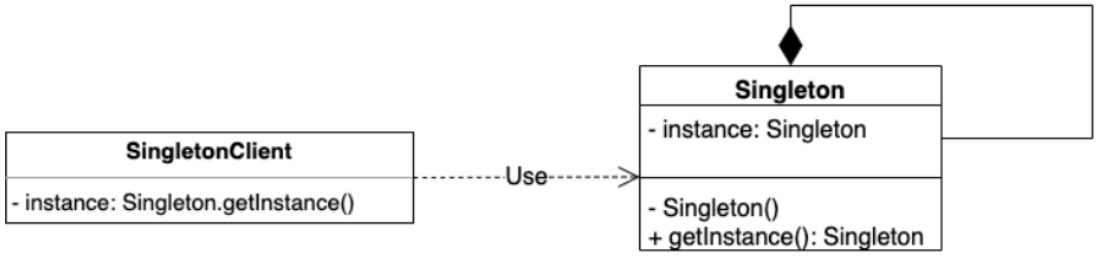


Figure 3. Singleton pattern (Retrieved from Kushwaha, n.d.).

The singleton design pattern is one of the most controversial design patterns in the world of software development. Some developers consider it an anti-pattern, making the code more complex and a real pain to reuse or test (Safyan, n.d.). However, the pattern is

one of the most straightforward design patterns to implement as it does not involve complex coding. The pattern will ensure the existence of only one instance of an object throughout the application by destroying the duplicates when created (Freeman et al., 2013; Gamma et al., 1994).

The pattern helps simplify the process of connecting different scripts as it makes the variable accessible globally across the application (Amat, 2020; French, 2023; Kushwaha.). Additionally, the global variables reduce the number of consumed resources, eliminating the need to store more data in memory (Finch, 2020). However, the singleton could make the code more difficult to read and trace, especially when your project gets larger (French, 2023). French (2023) explained the issue with a direct example of using an audio manager and how it becomes more challenging to modify the manager when more audio clips and settings are involved.

2.6.3 Observer Pattern

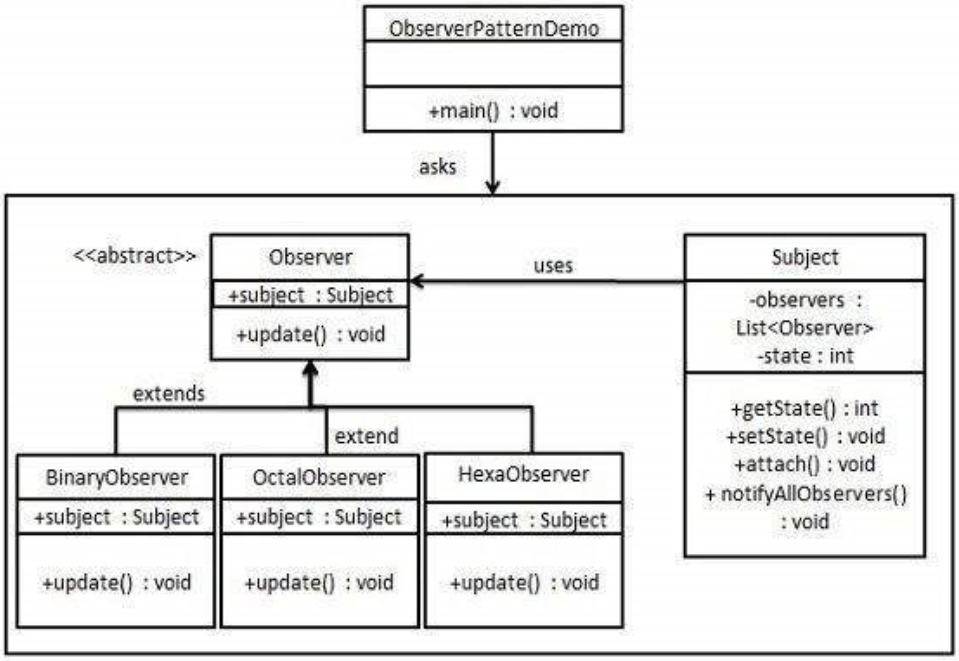


Figure 4. Observer pattern (retrieved from TutorialsPoint: Design Patterns - Observer Pattern, n.d.)

The observer design pattern is used heavily in enormous aspects of software development, which defines a one-to-many relation between objects (*Design Patterns - Observer Pattern*, n.d.; Freeman et al., 2013). The pattern allows some classes to observe and monitor some objects so that when the subject changes state, all observers are notified and updated automatically (Gamma et al., 1994). Imagine it as if it was the monitoring bracelet used in house arrests. The police officers (the observers) will use this bracelet to register an observer on the suspect (the observable). When the suspect leaves the house, the bracelet will notify the policemen (the observers) that the suspect's location (the observable) has changed.

According to Unity, the bigger the code gets, the more unnecessary dependencies lead to inflexibility and excess overhead (*Create Modular, More Maintainable Code with the Observer Pattern | Unity*, n.d.). As a result, they suggested using the observer pattern to overcome this issue. The pattern is ideal for connecting different aspects of the application "without being tightly coupled" (Hache, 2023). The pattern's advantage is that it separates the subject from the observer, who is uninterested in the observer's actions after receiving the signal.

2.6.4 State Pattern

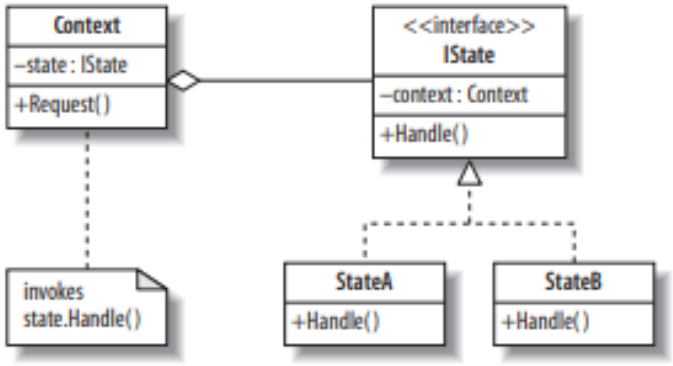


Figure 5. State pattern (retrieved from Bishop J, 2007)

The State pattern can be seen as a dynamic version of the Strategy pattern. It allows an object to alter its behavior when its internal state changes while appearing to change its class (Bishop, 2007; Freeman et al., 2013; Gamma et al., 1994). It encapsulates state-specific behaviors into separate state classes and delegates them to the current state object to handle requests rather than relying on conditionals. According to Gamma et al. (1994), this pattern promotes loose coupling by avoiding conditional logic related to states within

objects. It provides a cleaner way to implement state-dependent behavior than using polymorphism (Bishop, 2007).

The pattern applies when an object's behavior depends on its state, and it must change behavior at run-time based on that state, like a traffic light (Tutor, 2023). The state of the light acts differently when in the green state versus the red or yellow states, so each color has its own state. For example, writing multiple if statements based on different color conditions would make the code very messy and difficult to read, and having so many conditions will increase the difficulty in modifying the code. As a result, using the state pattern could solve this problem by creating different classes for each color state, making the code clean and easier to read as each state class handles its own specialized behavior.

For example, the RedState class would handle flashing the light and knowing when to transition to green. This separation of concerns makes the code more modular and easier to understand. The end benefit is that the traffic light class does not get bogged down with all the specific state rules. It just tells the current state object to handle any actions. This State pattern can apply to other objects with different modes or states.

2.6.5 Strategy Pattern

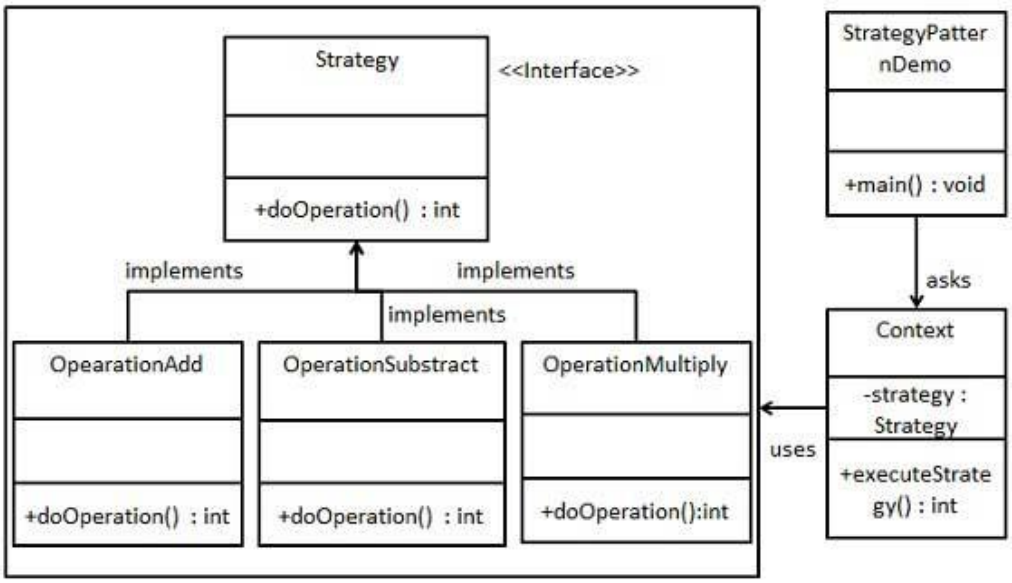


Figure 6. Strategy pattern (retrieved from TutorialsPoint: Design Patterns - Strategy Pattern, n.d.)

The strategy pattern defines a set of algorithms and encapsulates each to make them interchangeable. A simple example would be imagining a clothing store switching strategies to maximize sales during different seasons. For instance, the store would use a summer strategy focusing on selling lightweight clothing, including summer sales. While in winter, the strategy would change from selling lightweight clothes to something warmer and heavier (Tim R, n.d.).

The strategy enables flexible run-time logic selection using interchangeable, lightweight strategy objects, while the state pattern localizes state-based conditionals using polymorphic state objects to avoid conditional complexity (Freeman et al., 2013). This process allows the application to switch to different strategies or behaviors at run-time without the need to modify the context. The pattern provides robust mechanisms where an

application needs to adapt to varying conditions or requirements, enabling developers to choose the most appropriate algorithm or process dynamically.

2.6.6 Command Pattern

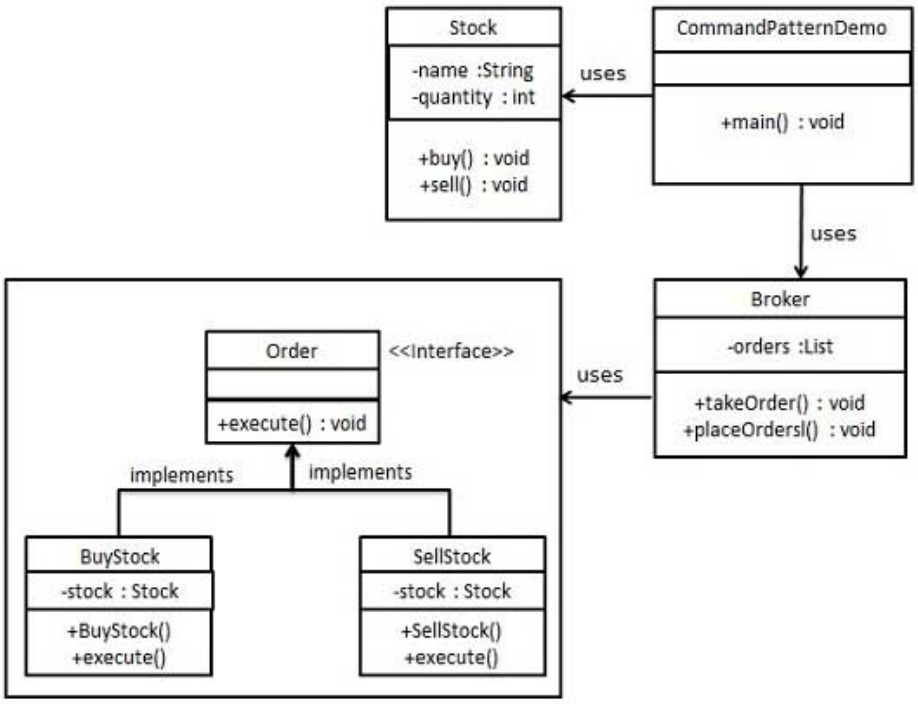


Figure 7. Command pattern (retrieved from TutorialsPoint: Design Patterns - Command Pattern, n.d.)

Command pattern separates the sender and receiver from each other by turning the requested action into a stand-alone object (*Command Design Pattern - GeeksforGeeks*, n.d.). Freeman et al. (2013) demonstrated the separation of the sender and receiver in a straightforward example that simplifies understanding of the pattern. The book's writers discussed this pattern as a food order. The customer would send their food order request to the chief through the waiter. The waiter will take that order and pass it to the chief, who will execute the instructions provided by the upcoming order.

The pattern brings several advantages to software development, making it a powerful choice for multiple scenarios. For instance, Bishop (2007) highlighted that the command pattern allows the application to make a list of the executed commands to allow the object to undo or redo the tracked commands. This process is possible as the command itself is an object that contains all the required instructions to be executed. Additionally, the command pattern allows the developer to add new commands without modifying and altering the existing code.

3. Technical Implementation

3.1 Introduction

The technical implementation section will discuss the design, tools, and coding methodology utilized to develop Escape the Planet. It will provide an overview of the game engine, programming languages, software architecture patterns, and other technical elements involved in constructing the game. Key aspects to be discussed include the object-oriented design patterns leveraged to enhance the game.

The choice of Unity as the game engine, along with C# for scripting, methods for optimized mobile graphics rendering, approaches for efficient physics simulations, strategies to handle mobile device limitations, and the modular scripting structure, enable rapid prototyping and iterative development. This section will give insights into the technical considerations and best practices that guided the development of Escape the Planet to meet performance, responsiveness, robustness, and maintainability goals while executing within mobile platform constraints.

3.2 Game Engine Choice

After taking the literature and the documentation into consideration, the chosen engine was decided to be Unity3D as it meets the requirements of developing Escape the Planet. The engine provides a straightforward UI design that makes learning its functionalities easier. In addition, the engine supports C# programming language, which has a simpler learning curve when compared to C++ in Unreal Engine. Escape the Planet does not count on high-quality graphics or high performance. As a result, going with Unity

seems more reliable since it is better used for developing games for mobile devices (Sarkar, 2023).

3.3 Icons and Sound Effects

Every game requires elements that make it attractive, exciting, and entertaining. As a result, sound and visual effects and icons are essential to enhance the game's user interface and audio experience. Escape the Planet provides multiple UI elements and various sound effects, making the game more intuitive, fun, and easy to play. However, designing these sounds and icons requires a creative mind and a sense of art, and lacking these skills led to third-party providers providing thousands of free sounds and icons for the community.

3.3.1 Sound Effects Source

The game without sounds feels boring and might lead the players to lose interest in the game. Also, it does feel great to have sound feedback when an action is being performed or to use it as a hint when you get closer to an object. Additionally, the sound must also be related to the action. For example, when the spaceship starts moving, the engine's sound must be relevant to the spaceship. Otherwise, playing a sports car engine when the ship moves will not make sense. However, these sounds needed to be recorded and engineered professionally for the perfect experience.

Due to the complex challenges of recording these sounds, free resources were reviewed. After going through multiple websites, "Zapsplat.com" provided various audio effects that matched the scenarios where the clips were played. The website provides

thousands of audio clips that could be used in games, movies, software, and many more applications. Their license allows any user with a free account to use these sounds as long as the user credits and references them properly. For instance, “Sound effects obtained from https://www.zapsplat.com.“

3.3.2 Icons

The buttons in the game are an essential part of controlling multiple functionalities, such as thrust, rotation, controlling the shield, and more. Some of these basic icons were designed and created using Adobe Photoshop. Other complex designs were retrieved from free source websites such as “Flaticon.com.” The website offers more than 14 million free icons. The icons are well-designed and provide user-friendly visuals. Flacion.com allows any user with a free account to use the icons as PNGs and other forms for the paid account. However, the website license states that the used icons must reference the authors appropriately. For instance, “Teleport icons created by Mihimihi – Flaticon,“ or “ image: retrieved from Flaticon.com.”

3.4 Design Pattern Implementation

Escape the Planet mainly counts on behavioral operations. After reviewing the literature, articles, videos, blogs, and more resources, some design patterns were selected to be implemented in the code to improve the game performance and the code quality, such as singleton pattern, state pattern, strategy pattern, observer pattern, and finally, command pattern.

3.4.1 Singleton

Escape the Plant has data to be saved in order to showcase the player's performance in the game. The data includes the total number of deaths across the game. The process of saving the data in multiple scripts and scenes is complex. For that reason, the singleton has helped simplify the data management process by creating a DontDestroyOnLoad object that will be carried on to all levels *(Figure 8)*. The object will give access to other scripts to save the data and give them permission to modify and alter it without connecting and referencing them directly.

C# provides an extremely powerful feature to unbound the class to a specific type, which will make the code applicable to different types with the same underlying behavior (C# Programming Guide - C# | Microsoft Learn, 2022; Generics - Unity Learn, n.d.). C# generics is a similar feature to C++ templates that was used in multiple scenarios.

A tutorial by Solo Game Dev (2022), "Unity Design Pattern - SINGLETON 2022," and Unity Technologies (2022), discuss how Generics help generalize the class and make it reusable with different types of objects or variables. Additionally, both discussed how implementing generics with the pattern has provided significant benefits, allowing the code to be more flexible and reusable across the application. Following the tutorial, the pattern was successfully implemented in the game. The singleton class will be generic, where T is a placeholder for the type parameter *(Figure 9)*, allowing other classes to be inherent for the main singleton class without creating different singleton classes in the game.

Gamma et al. (1994) introduced the singleton pattern with the primary goal of ensuring that a class can only have one instance. In Escape the Planet, the singleton class is created once, and other classes inherit from it, allowing the reusability of the singleton class in other classes without having them implement their singleton logic.

Bishop (2007) explained the logic behind using generics with a singleton class to make the code reusable across the application. The writer's example is focused on a wide range of C# applications. However, it does not work perfectly with the Unity project. In Escape the Planet, the pattern is coded to be used in the Unity application only due to some Unity-specific features, such as MonoBehaviour and the DontDestroyOnLoad method. Additionally, the pattern in Escape the Planet does not rely on a default constructor to ensure the existence of one instance in the application. It uses the Awak method provided by Unity to manage the singleton creation in the scene *(Figure 9)*.

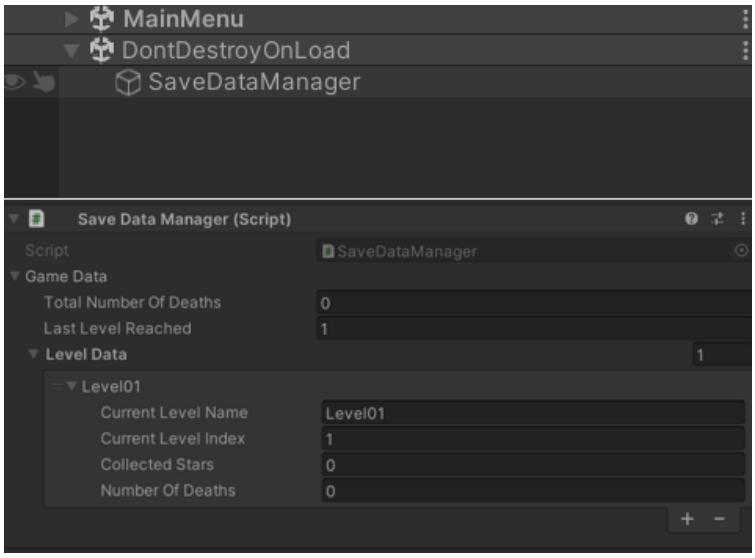
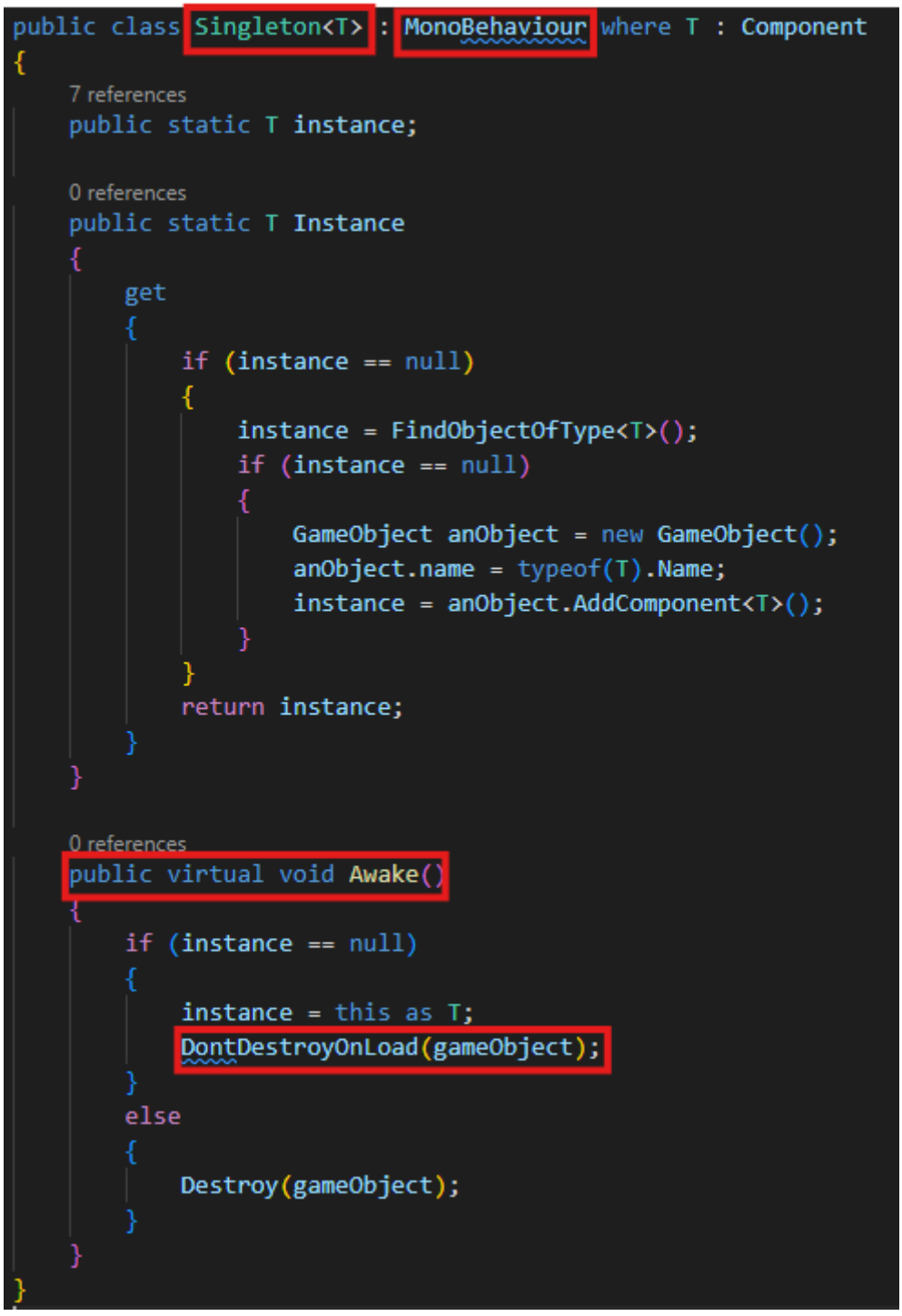


Figure 8. SaveDataManager as a singleton



```
public class Singleton<T> : MonoBehaviour where T : Component
{
    7 references
    public static T instance;

    0 references
    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                instance = FindObjectOfType<T>();
                if (instance == null)
                {
                    GameObject anObject = new GameObject();
                    anObject.name = typeof(T).Name;
                    instance = anObject.AddComponent<T>();
                }
            }
            return instance;
        }
    }

    0 references
    public virtual void Awake()
    {
        if (instance == null)
        {
            instance = this as T;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

Figure 9. Making the Singleton class generic to have different types of singletons

```
12 references | Codeium: Explain
public class SaveDataManager : Singleton<SaveDataManager>
{
    7 references
    [SerializeField] GameData gameData;

    4 references
    int currentCollectedStars = 0;
    3 references
    int tempOldCollectedStars = 0;
```

Figure 10. Marking the SaveDataManager Class as Singleton

```
// This is the collision handler script updating
// the number of deaths when the player collides with an object
1 reference | Codeium: Refactor | Explain | ×
void UpdateDataOnLosing()
{
    GameData gameData = SaveManager.Load();
    var currentLevelData = GetLevelData(gameData);
    currentLevelData.numberOfDeaths++;
    SaveManager.ResetTempCollectedStars();
    gameData.totalNumberOfDeaths++;
    SaveManager.Save(gameData);
}
```

Figure 11. CollisionHandler script accessing the SaveDataManager script to update the data

3.4.2 Observer

The observer pattern came in handy to modify the UI elements. It has been used to keep the players updated with fuel state, shield time, collected starts, and the collected key to unlock the doors. Whenever the player collects fuel barrels or starts pushing the thrust button, the observer pattern will ensure that the fuel bar level matches the current fuel amount. The process is implemented on all other UI elements to make this process easier.

In Escape the Planet, the pattern follows the core principles of the design pattern as described in the books which is to define a one-to-many relationship between the objects. However, the UI management includes multiple elements that depend on different variables and game objects, and implementing the observer pattern for each object is inefficient and would lead to so much redundant code. Fortunately, the tutorial by Solo Game Dev has inspired the implementation of generics to the observer pattern. In Escape the Planet, the implementation of generics has improved the way the observer pattern works. In *Figure 14,* both interfaces take generic parameters to avoid creating multiple interfaces for each UIState.

What distinguishes the implementation of the observer pattern in Escape the Planet is using UIState, an object of type enum, as shown in *Figure 12*, to enhance the flexibility and clarity of understanding the UI management logic. In *Figure 13,* each enum will help the observer pattern notify the corresponding observer when the state of the UI changes.

```
public enum UIState
{
    11 references
    FuelChanged,
    14 references
    ShieldChanged,
    5 references
    KeyState,
    4 references
    StarsState,
}
```

Figure 12. UI States

```
public void NotifyObservers(UIState state)
{
    foreach (var observer in observers)
    {
        observer.OnStateChange(this, state);
    }
}
```

Figure 13. NotifyObservers method

```
public interface IUIObservable<T> where T : UIManager
{
    23 references
    void NotifyObservers(UIState state);

    9 references
    void AddObserver(IUIObserver<T> anObserver);

    5 references
    void RemoveObserver(IUIObserver<T> anObserver);
}
```

```
public interface IUIObserver<T> where T : UIManager
{
    9 references
    void OnStateChange(T manager, UIState state);
}
```

Figure 14. IUIObservable and IUIObserver interfaces with a generic type parameter

The FuelUIObserver will monitor the FuelManager, which will act as the observable in this case.

```
public class FuelUIObserver : MonoBehaviour, IUIObserver<FuelManager>
{
    4 references
    private FuelManager fuelManager;
    2 references
    [SerializeField] Slider FuelSlider;
```

Figure 15. Implementing the observer pattern on the Fuel UI element

```
public class FuelManager : UIManager, IUIObservable<FuelManager>
{
    [Header("Fuel Settings")]
    2 references
    [SerializeField] bool isUsingFuel = false;
    2 references
    [SerializeField] float fuelAmount;
    3 references
```

Figure 16. FuelManager

3.4.3 State Pattern

Implementing the state pattern was not as easy as the other used patterns in this game. The pattern was complicated and confusing, leading to watching multiple tutorials and going through several documents related to this pattern. A tutorial video called "*How to Code a Simple State Machine (Unity Tutorial)"* by Amat (2020b) on his YouTube channel explained the pattern using a question-answer approach, making the understanding of the pattern more straightforward. Amat (2020b) explained the pattern using a Unity example, which helped understand the pattern concept as it was directly connected to game development using Unity. However, the developer provided some pre-coded examples with multiple classes and methods and then started modifying the code to explain the pattern's

concept, which some people might find confusing and difficult to keep up with the implementation.

A tutorial series called "*Finite State Machines in Unity"* by Thompson (2020) on his YouTube channel explained the pattern using a sample project. The developer explains the pattern using a baby-steps approach to help other developers understand the concept. The example that Thompson (2020) used was simpler than that of Amat (2020b); however, the tutorial series takes longer and more time. Both tutorials are considered reliable resources for understanding the state pattern in Unity.

The pattern is used in two of the game's main functionalities: the collision state and the fuel pad state. Since the pattern allows the control of the entire object's behavior, it comes in handy in managing the player's behavior. Both tutorials by Tommy Thompson and Charles Amat inspired how the pattern was implemented in this game. For example, Thompson (2020) used Scriptable Objects to present the state in Unity. Scriptable Objects is one of Unity's features that stores shared data independently from class instances. The unique implementation by Thompson (2020) was not discussed in any of the books, leading to a new way of implementing the pattern.

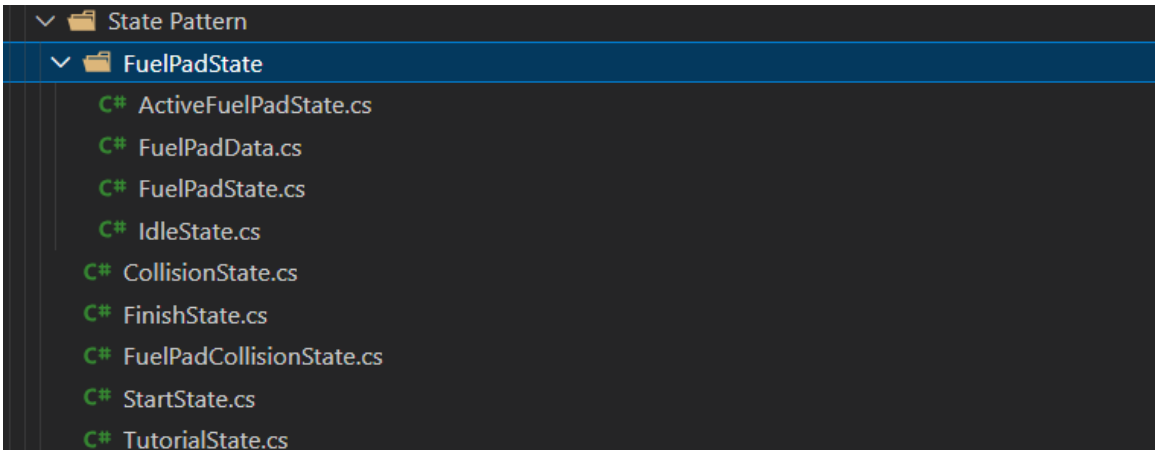


Figure 17. State pattern scripts folder

In Escape the Planet, the design pattern does not use Switch case and If statements in a client class to control the state machine. In *Figure 17,* multiple scripts were created to implement the pattern in the game, and since the pattern is being implemented into different functionalities, different approaches were created and coded in two different ways. The first implementation of the project used an abstract class to specify the fuel pad states, as highlighted in *Figure 18*. Meanwhile, in the CollisionState script, an interface class was created to handle the different collision states, as highlighted in *Figure 19*.

```
4 references | Codeium: Explain
public abstract class FuelPadState
{
    3 references
    public abstract void ActivateState(FuelPad pad);
    3 references
    public abstract void DeactivateState(FuelPad pad);
}
```

Figure 18. FuelPadState Abstract Class

```
public interface CollisionState
{
    6 references
    void Handle(CollisionHandler context);
}
```

Figure 19. ColliosnState Interface

When the player interacts with the fuel pad, the state changes according to the collision with the trigger area, as shown in *Figure 20*. The logic behind it is demonstrated and highlighted in *Figure 21**Error! Reference source not found.*** This approach helps eliminate any conditional statements such as IF or Switch.

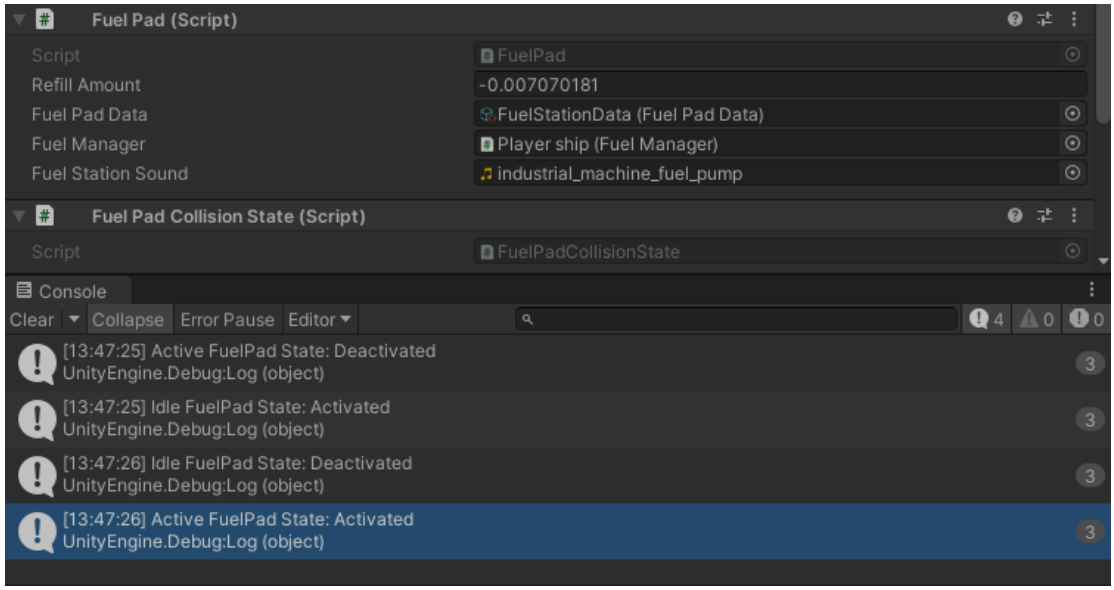


Figure 20. Fuel Pad State Implementation

```
public class FuelPad : MonoBehaviour
{
    10 references
    public float refillAmount;
    3 references
    float refillSpeed;

    3 references
    [SerializeField] FuelPadState currentState;
    2 references
    public FuelPadData fuelPadData;
    6 references
    public FuelManager fuelManager;
    1 reference
    [SerializeField] AudioClip FuelStationSound;

    8 references
    AudioSource AS;

    0 references | Codeium: Refactor | Explain | Generate Documentation | X
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            ChangeState(new ActiveFuelPadState());
        }
    }

    0 references | Codeium: Refactor | Explain | Generate Documentation | X
    private void OnTriggerExit(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            ChangeState(new IdleState());
        }
    }


    3 references | Codeium: Refactor | Explain | Generate Documentation | X
    public void ChangeState(FuelPadState newState)
    {
        currentState?.DeactivateState(this);
        currentState = newState;
        currentState.ActivateState(this);
    }
```

Figure 21. FuelPad Client Class

Implementing the pattern using an abstract class or an interface may provide some changes to how it is coded. Weimann (2017) created a game programming course on

YouTube to help new developers gain the required skills to work with Unity. He has discussed the differences between using an interface and an abstract class to implement different features in Unity.

In an interface, it is allowed to define the methods that any class needs to implement, and according to Microsoft, all versions above C#8 can contain a body code for any of its methods, as shown in *Figure 24,* and in C#10 the interface can declare a static variable; however, it cannot declare a variable as other classes. The code editor would return an error, as shown in *Figure 23 and Figure 24*, and the code would not work. (*Default Interface Methods - C# Feature Specifications | Microsoft Learn*, n.d.).

```
public class FinishState : MonoBehaviour, CollisionState
{
    2 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public void Handle(CollisionHandler context)
    {
        context.StartSuccessSequence();
    }
}
```

Figure 22. FinishState implementing CollisionState

```
public interface CollisionState
{
    0 references
    string CollisionType;
    6 references
    void Handle(CollisionHandler context);
}
```

Figure 23. Interface error

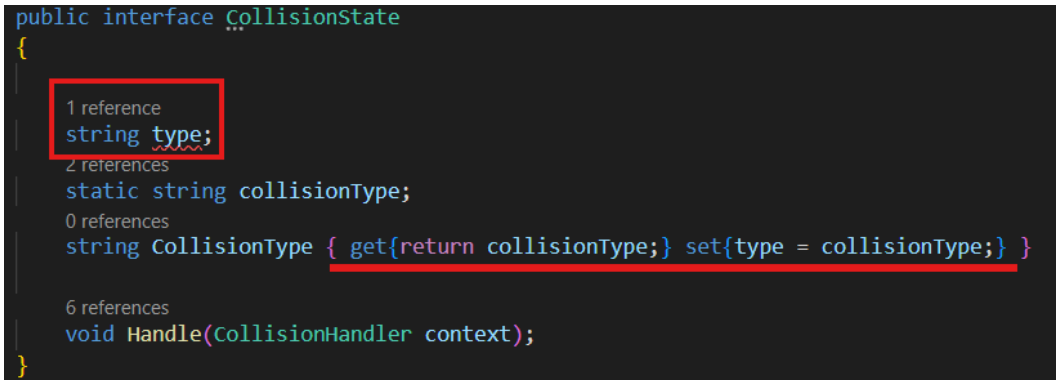


Figure 24. Interface new features

On the other hand, an abstract class could provide a default behavior to any object inherited from it or provide some other methods that the inherited classes could use. For example, in *Figure 25*, a method was defined to set the amount of refill fuel a station could provide, and in *Figure 26*, the method was used to set the amount of fuel to 100 every time the player landed on it. Such a behavior could not be implemented using an Interface.

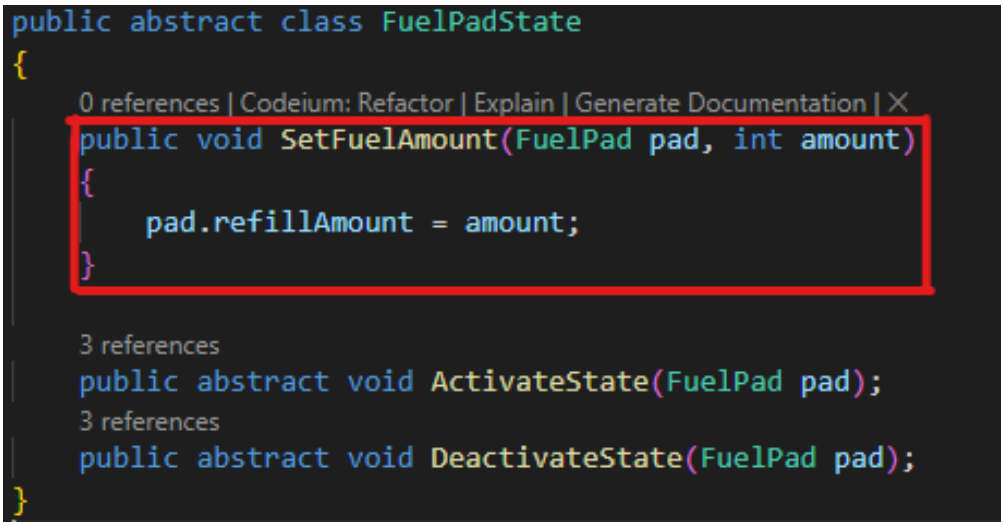


Figure 25. FuelPadState Class with a setter method

```
public class ActiveFuelPadState : FuelPadState
{
    2 references | Codeium: Refactor | Explain | Generate Documentation | X
    public override void ActivateState(FuelPad pad)
    {
        SetFuelAmount(pad, 100);
        pad.StartFuelRefill();
        Debug.Log("Active FuelPad State: Activated");
    }

    2 references | Codeium: Refactor | Explain | Generate Documentation | X
    public override void DeactivateState(FuelPad pad)
    {
        pad.StopFuelRefill();
        Debug.Log("Active FuelPad State: Deactivated");
    }
}
```

Figure 26. ActiveFuelPadState Class

After defining all the states, a client class must use the pattern in order to implement the pattern entirely. In *Figure 27*, the client references the FuelPadState to distinguish the different behavior of the fuel pad when the player interacts with it. Now, in the FuelPad Script, the behavior of the fuel pad changes when the player gets into the range and when they exit it. In *Figure 21*, the OnTriggerEnter method checks if the player got in range, and then it will call the ChangeState method to set the behavior of the fuel pad to be activated and start providing the player with fuel. On the other hand, the OnTriggerExit method will do the opposite. When the player leaves the range, the state will be changed to an idle state, and the player will no longer be provided with fuel.

```
public class FuelPad : MonoBehaviour
{
    10 references
    public float refillAmount;
    3 references
    float refillSpeed;

    3 references
    [SerializeField] FuelPadState currentState;
    2 references
    public FuelPadData fuelPadData;
    6 references
    public FuelManager fuelManager;
    1 reference
    [SerializeField] AudioClip FuelStationSound;
```

Figure 27. FuelPad Script

The implementation of the collision state followed a different approach than the approach used in the fuel state. Each state is created as a MonoBehavior class, which provides the ability to attach the script to an object, as shown in *Figure 28*. Using this approach has eliminated the use of any conditional statement. In *Figure 29,* the old code was commented out, and the new one implemented the new approach. In the old approach, the code used the OnCollisionEnter method to check for the game object tag using a switch case to decide what behavior to use. However, the pattern changed the way the game functions. The OnCollisionEnter method will now get the object's state after being hit and then execute the corresponding behavior. If the object does not have any specific behavior, the default behavior will be executed, which in this case is StartCrushSequence.

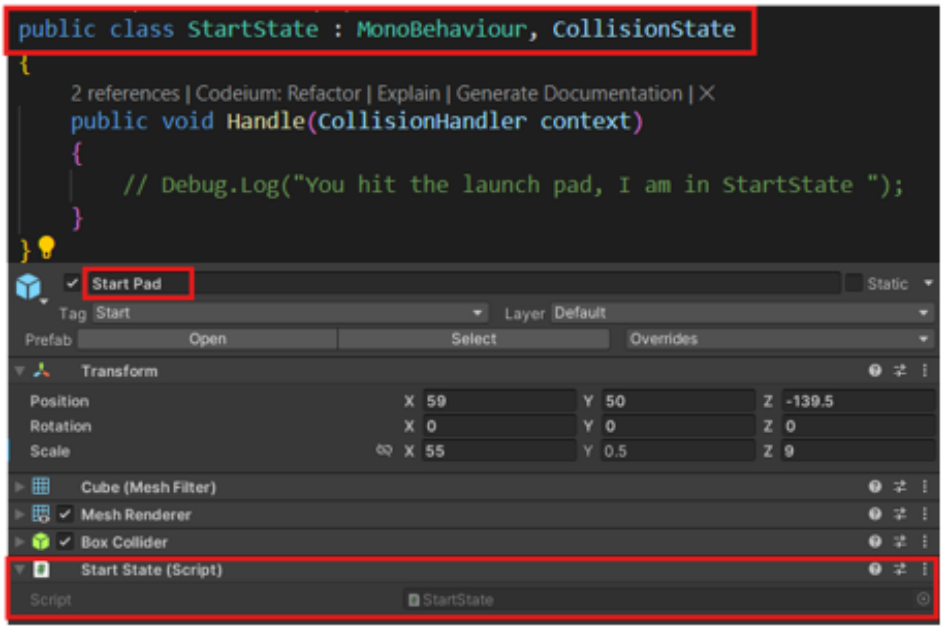


Figure 28. Attached state to an object

```
void OnCollisionEnter(Collision other)
{
    if (isTransitioning || CollisionDisabled)
    {
        return;
    }

    CollisionState state = other.gameObject.GetComponent<CollisionState>();
    if (state != null)
    {
        state.Handle(this);
    }
    else
    {
        Debug.Log($"{other.gameObject.name} has no CollisionState");
        StartCrashSequence();
    }

    // the switch statement is not needed since we have a CollisionState that being attached to game objects
    // switch (other.gameObject.tag)
    // {
    //     case "Start":
    //         currentState = new StartState();
    //         // Debug.Log("You hit the Start pad, I am in collision handler");
    //         break;
    //     case "FuelPad":
    //         //Debug.Log("You hit the Fuel pad");
    //         currentState = new FuelPadState();
    //         break;
    //     case "Finish":
    //         StartSuccessSequence();
    //         break;
    //     default:
    //         StartCrashSequence();
    //         break;
    // }
    // currentState.Handle(this);

}
```

Figure 29. Collision State Implementation

3.4.4 Strategy Pattern

Implementing the strategy pattern was much simpler than the state pattern. The pattern is used to define multiple in-game collectible behaviors to help the player during their journey. The strategy pattern was used instead of the state pattern in this case, as the behavior of each collectible is almost the same but still slightly different. For example, the state pattern is used when there is a significant change in the object's behavior, similar to what might happen when the player wins or loses the game. In a winning condition, the player must go to the next level, and the data will be saved. While in a losing condition,

the player's location will be reset, the data will not be saved, and the whole behavior will change.

On the other hand, in the strategy pattern, the behavior does not change significantly. For instance, the collectibles in the game have similar behavior where they get destroyed, instantiate an explosion effect, and play a sound effect when collected. Still, the type of power-up they provide or the sound effects are different.

In Escape the Planet, the collectibles come in different shapes and types. Each one of them has its own unique purpose. The FuelBarrle is used to boost the player's fuel to help them finish the level, and the ShieldPowerUp collectible helps extend the shield uptime to protect the ship from any enemy cannons around the level's map. Also, the Key collectible is used to give the player the ability to unlock doors that hold valuable game objects behind them, such as the Star collectible, which is used as a scoring system for each level.

In *Figure 30,* multiple scripts were created to implement the pattern in the game. Each script serves as an essential part and provides the required behavior for each collectible. Similar to the observer patterns, the strategy pattern implements generic interface to ensure code fixability and reusability.

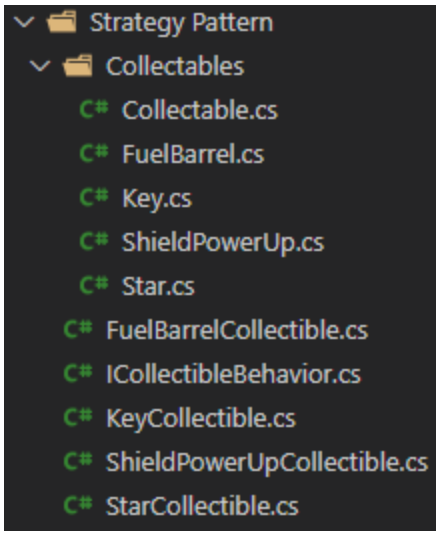


Figure 30. Strategy Pattern Scripts

Starting with the first major part of the pattern, the IcollectibleBehavior is a generic interface, as shown in *Figure 31,* that defines a signature method that must be defined in any class that implements the interface. Unlike the books, Escape the planet introduce the strategy patterns alongside with the observer pattern. That allowed the creation of a fully functional system that updates the UI as soon as the player collects one of the collectibles.

```
public interface ICollectibleBehavior<T> where T : UIManager
{
    9 references
    void ExecutePowerUp(T manager);
}
```

Figure 31. IcollectibleBehavior interface with a generic parameter

Going back to the observer pattern, a UIManager class was created, and multiple children managers were inherited from it, such as the FuelManatger, StarsManager,

ShieldManager, and KeyManager. *Figure 32* highlights some of the collectibles' behavior classes. The collectible implements the interface with the specific UI manager type responsible for updating the corresponding element on the screen.

```
public class KeyCollectible : ICollectibleBehavior<KeyManager>
{
    2 references
    private bool hasKey;

    1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
    public KeyCollectible(bool hasKey)
    {
        this.hasKey = hasKey;
    }

    6 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public void ExecutePowerUp(KeyManager keyManager)
    {
        keyManager.PlayerHasKey = hasKey;
    }
```

```
public class FuelBarrelCollectible : ICollectibleBehavior<FuelManager>
{
    2 references
    private int amount;

    1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
    public FuelBarrelCollectible(int amount)
    {
        this.amount = amount;
    }

    6 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public void ExecutePowerUp(FuelManager fuelManager)
    {
        Debug.Log($"fuelManager.FuelCounter " + fuelManager.FuelAmount);
        if (fuelManager.FuelAmount < fuelManager.MaxFlightTime)
        {
            fuelManager.FuelBarrel(amount);
        }
    }
```

Figure 32. Some of the Collectable Behaviors

Additionally, *Figure 32* shows how the collectibles are similar but provide a different algorithm to handle their own behavior.

In *Figure 33,* a client class that also uses a generic parameter was created to call the collect method as soon as the player gets in the range of any of the collectibles on the map. The Collectible class will act as a parent class for multiple children. The scripts will be attached to a game object to make it a collectible object. The OnTriggerEnter method will decide what algorithm to execute. Each collectible uses its own strategy that provides a unique power-up for the player, as shown in *Figure 34*. However, the process of collecting the object is still the same, as all the collectibles will spawn an explosion effect and then destroy that object when collected.

```
public abstract class Collectable<T> : MonoBehaviour where T : UIManager
{
    1 reference
    [SerializeField] protected ParticleSystem ExplosionEffect;

    // This is a part of the strategy pattern, it will be used to determine the type of collectable
    8 references
    protected ICollectibleBehavior<T> collectibleBehavior;

    0 references | Codeium: Refactor | Explain | Generate Documentation | ×
    private void OnTriggerEnter(Collider other)
    {
        var manager = other.GetComponent<T>();

        if (manager != null)
        {
            Collect(manager);

            Instantiate(ExplosionEffect, transform.position + new Vector3(0, 1f, 0), Quaternion.identity);
            Destroy(gameObject);
        }
    }

    5 references
    protected abstract void Collect(T manager);
}
```

Figure 33. The Collectible Parent Class

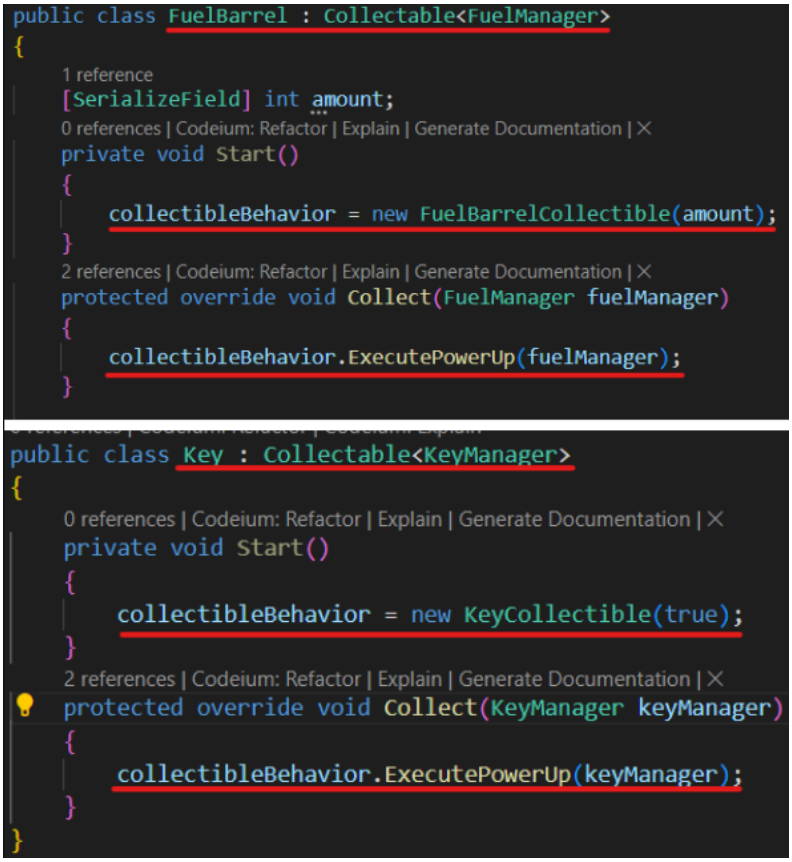


Figure 34. Children Collectible Classes

3.4.5 Command Pattern

The command pattern solved an issue that the MobileController script failed to solve. The MobileController was used to handle all the player input, which is against one of the essential principles of programming, the separation of code logic. Implementing the pattern made the player input system more flexible and easier to modify when a new command is wanted, and refactoring the MobileController script has led to the creation of multiple scripts to handle each input independently, as shown in *Figure 35*. The pattern consists of multiple key components responsible for processing different player inputs.

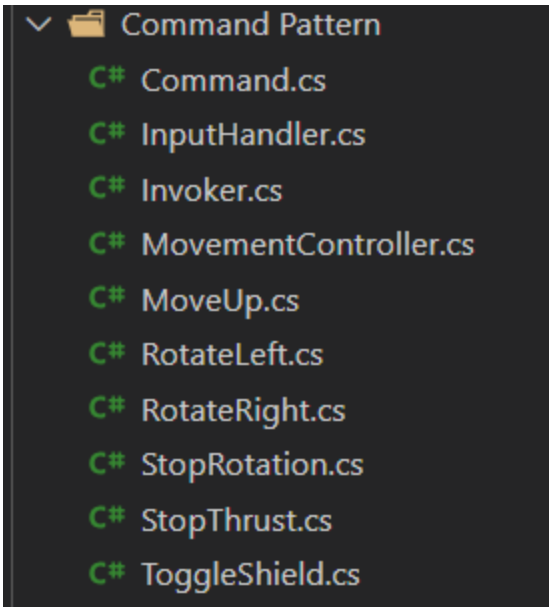


Figure 35. Command Pattern Scripts

The pattern is divided into four main parts that are combined with each other to implement the pattern correctly. The first part of the command pattern is the Command abstract class. This class defines an execute method that all the other commands must implement.

```
public abstract class Command
{
    7 references
    public abstract void Execute(Rigidbody rigidbody, AudioSource audioSource);

}
```

Figure 36. Command Abstract Class

The second part of the pattern is the Invoker class, which is responsible for the execution of the player commands when they press a button. It calls the command object

and then runs the Execute method. The class is not connected to the pattern directly as it is not unaware of the concrete class that implements the Command interface.

```
public class Invoker
{
    6 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public void ExecuteCommand(Command command, Rigidbody rigidbody, AudioSource audioSource)
    {
        if (command != null)
        {
            command.Execute(rigidbody, audioSource);
        }
        else
        {
            Debug.Log($"Command is null");
        }
    }

}
```

Figure 37. The Invoker Class

Escape the Planet allows players to move up, stop thrusting, rotate left, rotate right, stop the rotation, and finally toggle the shield. These commands are the next part of the pattern, consisting of multiple concrete classes inherited from the Command abstract class. Each command provides a parameterized constructor that will be used by the client class to create a new object of the command, and these classes will also implement the execute method as a part of the pattern to carry out a specific action similar to the MoveUp command in *Figure 38* and the RotateLeft command in *Figure 39.*

```
public class MoveUp : Command
{
    2 references
    private float movementSpeed;
    2 references
    private AudioClip mainEngine;
    4 references
    private ParticleSystem rocketBoostParticles;
    4 references
    private FuelManager fuelManager;

    1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
    public MoveUp(float movementSpeed, AudioClip mainEngine, ParticleSystem rocketBoostParticles, FuelManager fuelManager)
    {
        this.movementSpeed = movementSpeed;
        this.mainEngine = mainEngine;
        this.rocketBoostParticles = rocketBoostParticles;
        this.fuelManager = fuelManager;
    }

    2 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public override void Execute(Rigidbody rigidbody, AudioSource audioSource)
    {
        FuelConsumption(audioSource);
        AudioControl(audioSource);
        rigidbody.AddRelativeForce(Vector3.up * Time.deltaTime * movementSpeed);
    }
```

Figure 38. MoveUp Command

```
public class RotateLeft : Command
{
    2 references
    private float rotationSpeed;
    3 references
    private ParticleSystem leftThrustParticles;

    1 reference | Codeium: Refactor | Explain | Generate Documentation | ×
    public RotateLeft(float rotationSpeed, ParticleSystem leftThrustParticles)
    {
        this.rotationSpeed = rotationSpeed;
        this.leftThrustParticles = leftThrustParticles;
    }

    2 references | Codeium: Refactor | Explain | Generate Documentation | ×
    public override void Execute(Rigidbody rigidbody, AudioSource audioSource)
    {
        ApplyRotation(rigidbody, 1);
        if (!leftThrustParticles.isPlaying)
        {
            leftThrustParticles.Play();
        }
    }
```

Figure 39. RotateLeft Command

Finally, the last part of the pattern is the client class, which will be responsible for creating the command objects and assigning each command to a specific button that the player uses to perform an action. When the player presses a button, the client class will ask the invoker class to execute the corresponding command.

```
public class InputHandler : MonoBehaviour
    private Invoker invoker;

    2 references | 2 references | 2 references
    private Command moveUpCommand, rotateLeftCommand, rotateRightCommand,
                   2 references | 2 references | 2 references
                   stopThrustCommand, stopRotationCommand, toggleShieldCommand;
    0 references | Codeium: Refactor | Explain | Generate Documentation | X
    private void Awake()
    {
        invoker = new Invoker();
        FuelManager fuelManager = FindObjectOfType<FuelManager>();
        ShieldManager shieldManager = FindObjectOfType<ShieldManager>();
        movementController = FindObjectOfType<MovementController>();

        this.moveUpCommand = new MoveUp(movementSpeed, mainEngine, rocketBoostParticles, fuelManager);
        this.rotateLeftCommand = new RotateLeft(rotationSpeed, leftThrustParticles);
        this.rotateRightCommand = new RotateRight(rotationSpeed, rightThrustParticles);
        this.stopThrustCommand = new StopThrust(mainEngine, rocketBoostParticles);
        this.stopRotationCommand = new StopRotation(leftThrustParticles, rightThrustParticles);
        this.toggleShieldCommand = new ToggleShield(shieldManager, shieldActivationSound);
    }
private void Update()

    if (movementController == null)
    {
        Debug.LogError("MovementController is not assigned in the InputHandler script.");
        return;
    }
    if (Input.GetKey(KeyCode.Space) || CrossPlatformInputManager.GetButton("Thrust"))
    {
        invoker.ExecuteCommand(moveUpCommand, movementController.Rigidbdy, movementController.AS);
    }
    if (Input.GetKeyUp(KeyCode.Space) || CrossPlatformInputManager.GetButtonUp("Thrust"))
    {
        invoker.ExecuteCommand(stopThrustCommand, movementController.Rigidbdy, movementController.AS);
    }
    if (Input.GetKey(KeyCode.A) || CrossPlatformInputManager.GetButton("Left"))
    {
        invoker.ExecuteCommand(rotateLeftCommand, movementController.Rigidbdy, movementController.AS);
    }
    if (Input.GetKey(KeyCode.D) || CrossPlatformInputManager.GetButton("Right"))
    {
        invoker.ExecuteCommand(rotateRightCommand, movementController.Rigidbdy, movementController.AS);
    }
    if (Input.GetKeyUp(KeyCode.D) || CrossPlatformInputManager.GetButtonUp("Right")
    || Input.GetKeyUp(KeyCode.A) || CrossPlatformInputManager.GetButtonUp("Left"))
    {
        invoker.ExecuteCommand(stopRotationCommand, movementController.Rigidbdy, movementController.AS);
    }
    if (Input.GetKeyDown(KeyCode.E) || CrossPlatformInputManager.GetButtonDown("Shield"))
    {
        invoker.ExecuteCommand(toggleShieldCommand, movementController.Rigidbdy, movementController.AS);
    }
}
```

Figure 40. InputHandler Client Class

The command pattern in Escape the Planet did not implement the option to track the player input or to give them the possibility to redo and undo the previously executed commands due to the game's nature and mechanics. The game depends on the player's reaction to the game's obstacles, and allowing them to redo and undo the executed command would not help them complete the game.

3.5 Conclusion

In conclusion, Escape the Planet is a game that aims to make the players challenge themselves. The game demonstrates various programming principles and design patterns that were discussed by multiple developers and scholars in the literature. The development process was covered with complex and difficult challenges that required well-structured research to be able to solve the problems.

All the books discussed the implementation of patterns using different approaches. Gamma et al. (1994) focused on describing and introducing the patterns using a language-agnostic approach. The main goal of his book is to introduce the solution and solve the problem by providing a deep theoretical foundation. The book used UML diagrams and simple examples that helped understand each pattern's logic. The book was written over 20 years ago, making it an outdated but reliable source for understanding design patterns.

Bishop (2007) and Freeman et al. (2013) both referred to Gamma et al. (1994) book due to its valuable information. Each of the books has its unique traits. Bishop (2007) focused on tailoring the patterns around C# applications to help developers learn more about object-oriented programming in C#. The book discussion of Generics helped to

create a reusable code and logic across Escape the Planet. The book was introduced back in 2007 when C#3 was released. As a result, some of its examples can be improved by the new C# features. Freeman et al. (2013) have a unique graphical representation of the patterns. The authors focused on tailoring the patterns around real-life examples to make it suitable for beginners who want to learn more about advanced OOP.

The utilization of the Unity3D engine came along due to its simplicity, the massive amount of community support and tutorials available online, and the C# programming language, which is a powerful programming language used for multiple applications. The engine also proved to be a robust environment for mobile platforms. The engine provides a variety of external tools and free assets for limited-budget projects. The engine's packages are solid and robust and could be easily used. Moving forward, implementing multiple design patterns, such as the Singleton, Observer, State, Strategy, and Command patterns to implement some of the game core mechanics, has provided a solid code quality and foundation that could be easily maintained and modified even if new features is planned to be added later. In the end, the technical choices that came up through the developing progress have led to successful implementation and a fully functional game. This solid foundation also sets up solid foundations for future development and potential expansions to the game.

The books do not provide a direct example of how to make your code reusable. Escape the Planet, on the other hand, implements the patterns using a unique approach. The game focused on tailoring the patterns around the Unity project, making it a valuable resource for understanding how the singleton, observer, state, strategy, and command

patterns work in this environment. The project's primary goal concentrated on making the game functional using less code and practical solutions. The game followed all the practices discussed in the books and tutorials and sufficient techniques inspired by the literature. It ensures the code reusability and maintainability, as discussed in Technical Implementation. Moreover, Escape the Planet visualizes the logic of the implemented patterns since it is a game. The person who wants to learn more about the chosen patterns can go through the code and watch the logic being implemented in front of their eyes.

4. Software Requirements and Specification

4.1 Introduction

Creating an immersive and interactive experience is paramount in the evolving landscape of mobile gaming. This section outlines the software requirements for Escape the Planet. The requirements are segmented into distinct functionalities and interface design elements, each crafted to enhance the user's engagement and enjoyment.

4.2 General Functionalities

- The game shall allow the player to control the spaceship using on-screen buttons.
- The game shall provide a three-star scoring system for each level.
- Players could collect power-ups like shields and fuel boosts.
- The game shall track the player's performance at each level. (The number of deaths and the number of collected stars).
- The game shall display the fuel amount in the ship as a slider if enabled.
- The game shall display the shield's remaining uptime as a slider if enabled.
- The game shall allow the player to restart each level at any point.
- The game shall allow the player to reset the game.
- The game shall allow the player to replay previously finished levels if the player has not restarted the game.

4.3 Win Condition Functionalities

- The game shall proceed to the next level when the player hits the finish pad.
- The game shall store the number of collected stars for each level.

- The game shall change the last level reached variable to match the latest unlocked level.

4.4 Lose Condition Functionalities

- The game shall restart the level when hitting any obstacle on the map.
- The game shall restart the counter of collected stars when losing.
- The game shall increase the number of deaths on losing.

4.5 Interface

- The game shall provide a user-friendly interface.
- The game shall provide appropriate visuals.
- The game shall use the English language for all of its written text.
- The game might provide slightly different views on different screens depending on the aspect ratio.

4.5.1 Main menu wireframe

- The game shall provide a main menu to navigate between the scenes.
- The game shall provide a list of levels.
    - The game shall enable the unlocked level buttons.
    - The game shall disable the locked level buttons.
    - The game shall display the number of collected stars for each level.
    - The game shall provide a reset game button.
    - The game shall provide a confirmation message for the reset option.
- The game shall provide a list of tutorials.

- The game shall provide a performance panel.
    - The game shall display the total number of deaths in the game.
    - The game shall display the total number of deaths for each level independently.
- The game shall provide a support panel.
    - The game shall provide a contact email.

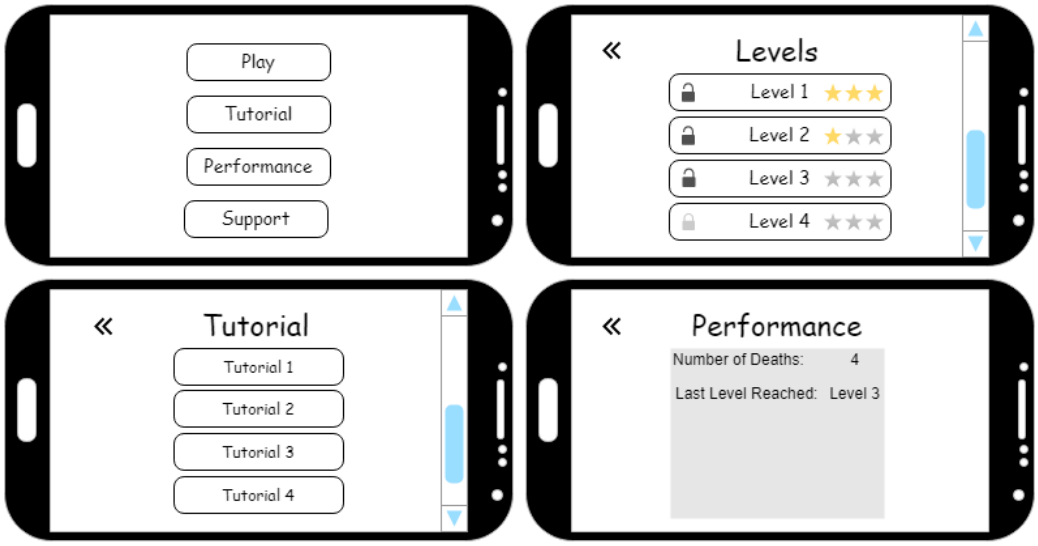


Figure 41. Main menu wireframe

4.5.2 HUD Wireframe

- The game shall display rotation buttons on the bottom-left side of the screen.
- The game shall display the thrusting button on the bottom-right side of the screen.
- The game shall display the pause menu button on the top-left side of the screen.
- The game shall display the shield button next to the thrusting button on the left if needed.

- The game shall display the teleport button on the bottom-mid side of the screen if needed.
- The game shall display the fuel slider on the top-left side of the screen next to the pause menu button if needed.
- The game shall display the shield slider on the top-right side of the screen if needed.
- The game shall display the stars' states on the top-mid side of the screen.
- The game shall display the key's state below the stars' state.

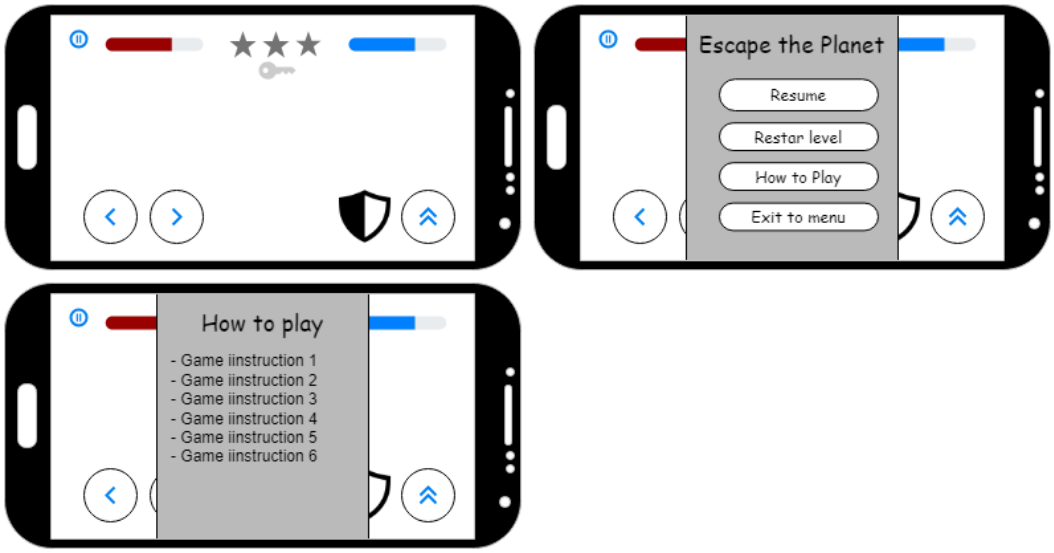


Figure 42. HUD Wireframe

4.6 Data and Information

4.6.1 Storage

- The game shall not save or store any private information.
- The game shall use the players' local space for storage.
    - The game shall store the player status in a binary file.

- The player shall not modify or edit the binary file (it may lead to data loss).

4.6.2 Data Security

- The game shall not ask players to provide any confidential information.
- The game shall not ask players to connect to the internet except for new updates.

4.7 In-game Purchase

- The game shall not provide any in-game purchases.
- The game shall not ask the player for Credit/Debit card information.

4.8 System Requirements

- The game shall work on Android devices compatible with a minimum API level of 24 and above. (Android 7.0 “Nougat” and above).

References

Ahmad, A., Feng, C., Ma, T., Yousif, A., & Shi, G. (2017). Challenges of mobile applications development: Initial results. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, *2017-November*, 464–469. https://doi.org/10.1109/ICSESS.2017.8342956

Amat, C. (2020). *Everything You Need to Know About Singletons in Unity - YouTube*. https://www.youtube.com/watch?v=mpM0C6quQjs&ab_channel=InfallibleCode

Amat, C. (2020b). *How to Code a Simple State Machine (Unity Tutorial) - YouTube*. https://www.youtube.com/watch?v=G1bd75R10m4&ab_channel=InfallibleCode

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. *Program Comprehension, Workshop Proceedings*, 153–160. https://doi.org/10.1109/WPC.1998.693342

Arabaine, I. (2023, January 5). *The State of Mobile UX: Challenges and Opportunities | by Ilyass Arabaine | Bootcamp*. https://bootcamp.uxdesign.cc/the-state-of-mobile-ux-challenges-and-opportunities-71ea1af077ad

Arnomo, S. A., Simanjuntak, P., & Nur Sadikan, S. F. (2021). Overheating Analysis of Mobile Phone Temperature Based on Multitasking Process. *Proceedings - 2nd International Conference on Computer Science and Engineering: The Effects of the Digital World After Pandemic (EDWAP), IC2SE 2021*. https://doi.org/10.1109/IC2SE52832.2021.9792125

Bishop, J. (2007). *C# 3.0 Design Patterns* (J. Osborn, L. Dimant, & R. Wheeler, Eds.; First Edition). O'Reilly Media, Inc.

Christopoulou, E., & Xinogalos, S. (2017). Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices. *International Journal of Serious Games*, *4*(4). https://doi.org/10.17083/ijsg.v4i4.194

Command Design Pattern - GeeksforGeeks. (n.d.). Retrieved February 9, 2024, from https://www.geeksforgeeks.org/command-pattern/

Constraints on type parameters - C# Programming Guide - C# | Microsoft Learn. (2022, November 15). https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

Create Modular, More Maintainable Code with the Observer Pattern | Unity. (n.d.). Retrieved December 26, 2023, from https://unity.com/how-to/create-modular-and-maintainable-code-observer-pattern

Default interface methods - C# feature specifications | Microsoft Learn. (n.d.). Retrieved April 6, 2024, from https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods

Design Patterns - Observer Pattern. (n.d.). Retrieved January 11, 2024, from https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

Drake, J. (2023, September 30). *The Best Games That Use The Unity Game Engine*. https://www.thegamer.com/unity-game-engine-great-games/#night-in-the-woods

Early history of video games - Wikipedia. (n.d.). Retrieved December 26, 2023, from https://en.wikipedia.org/wiki/Early_history_of_video_games

Finch, D. (2020, April 28). *Singleton Pattern Explained - Creational Design Patterns*. https://darrenfinch.com/singleton-pattern-explained-creational-design-patterns/

Freeman, E., Freeman, E., Bates, B., & Sierra, K. (2013). Head First Design Patterns. *Carcinogenesis*, *34*(2), 619.

French, J. (2023, May 22). *Singletons in Unity (done right) - Game Dev Beginner*. https://gamedevbeginner.com/singletons-in-unity-the-right-way/

Games - Worldwide | Statista Market Forecast. (2023). https://www.statista.com/outlook/amo/media/games/worldwide

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*.

Gapminder Tools. Retrieved December 5, 2023, from https://www.gapminder.org/tools/#uichart$cursorMode=minus;;&model$markers$bubble$encoding$size$data$concept=net_users_num&space@=country&=time;;&scale$domain:null&type:null&zoomed:null;;&y$data$concept=cell_phones_total&source=sg&space@=country&=time;;&scale$domain:null&zoomed@:0&:1746238000;&type:null;;&x$data$concept=pop&space@=country&=time;;&scale$domain:null&zoomed@:651&:1696976687;&type:null;;&frame$value=1990;;;;;&chart-type=bubbles&url=v1

Generics - Unity Learn. (n.d.). Retrieved January 26, 2024, from https://learn.unity.com/tutorial/generics#

Gregory, J. (2018). *Game Engine Architecture, Third Edition*. A K Peters/CRC Press. https://doi.org/10.1201/9781315267845

Hache, C. (2023, May 17). *Top 7 Design Patterns Every Unity Game Developer Should Know | LinkedIn*. https://www.linkedin.com/pulse/top-7-design-patterns-every-unity-game-developer-should-charles-hache/

Knezovic, A. (2023, December 13). *200+ Mobile Game Statistics: Market Report [2023] - Udonis*. https://www.blog.udonis.co/mobile-marketing/mobile-games/mobile-gaming-statistics#h2-0

Kushwaha, N. . *Learn the Singleton Design Pattern - LEARNCSDESIGN*. Retrieved January 11, 2024, from https://www.learncsdesign.com/learn-the-singleton-design-pattern/

List of Unreal Engine games | Fandom. (n.d.). Retrieved February 7, 2024, from https://thecoolestvideogames.fandom.com/wiki/List_of_Unreal_Engine_games

Mitchell, P. (2023, May 19). *Does Gaming Destroy Phone Battery? – TechCult*. https://techcult.com/does-gaming-destroy-phone-battery/

Nikolaeva, D., Bozhikova, V., & Stoeva, M. (2019). A simple approach to design patterns identification in programming code. *2019 28th International Scientific Conference Electronics, ET 2019 - Proceedings*. https://doi.org/10.1109/ET.2019.8878506

Safyan, M. (n.d.). *Singleton Anti-Pattern - Michael Safyan*. Retrieved January 8, 2024, from https://www.michaelsafyan.com/tech/design/patterns/singleton

Sarkar, S. (2023, August 30). *Unity vs Unreal: Ending the Endless Debate in Mobile vs. PC/Console Game Development | LinkedIn*. https://www.linkedin.com/pulse/unity-vs-unreal-ending-endless-debate-mobile-pcconsole-sarkar/

Šmíd, A. (2017). *Comparison of Unity and Unreal Engine*.

Szeja, R. (2023, August 3). *14 Biggest Challenges in Mobile App Development in 2022*. https://www.netguru.com/blog/mobile-app-challenges

This Month in Physics History - October 1958: Physicist Invents First Video Game. (2008, October). https://www.aps.org/publications/apsnews/200810/physicshistory.cfm

Thompson, T. (2020). *Building an Idle State | Finite State Machines in Unity (#2) | Table Flip Games - YouTube*.

https://www.youtube.com/watch?v=e8m6yXDIx9U&ab_channel=TableFlipGames

Tim R. (n.d.). *The Strategy Pattern*. Retrieved February 3, 2024, from https://www.topcoder.com/tc?module=Static&d1=tutorials&d2=strategyPattern

Tutor, L. (2023, August 17). *State Pattern in C#: From Basics to Advanced | by Laks Tutor | Medium*. https://medium.com/@lexitrainerph/state-pattern-in-c-from-basics-to-advanced-de3a0dc526af

Unity Design Pattern - SINGLETON (2022) - YouTube. (2022). https://www.youtube.com/watch?v=F6Y8q9H3UZI&t=3s&ab_channel=SoloGameDev

Unity Technologies. (2022). *Level up your programming with game programming patterns | Unity*. https://unity.com/resources/level-up-your-code-with-game-programming-patterns?ungated=true

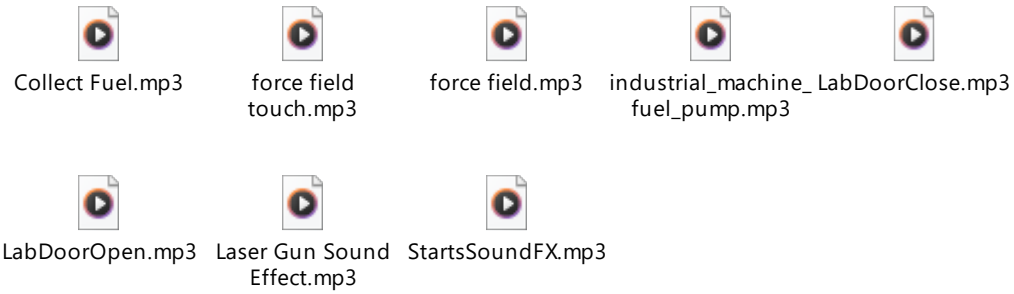
Weimann, J. (2017, September 10). *Unity Interfaces vs Abstract Classes - Part 1 - Interfaces - YouTube*. https://www.youtube.com/watch?v=kYJRIWjoeFA&ab_channel=JasonWeimann

Appendices

Sound Effects

Below are the sound effects that are being used in the game. The file name has been changed for easier reference in the code. The file name is usually named similar to this: “zapsplat_warfare_missile_incoming_whizz_by_then_explosion_004_31165.mp3”

Sounds form Zapsplat



Sounds form Freesound.org

Icons

Uncollected key Icon retrieved from Flaticon.com.

Shield Button Icon retrieved from Flaticon.com.

Teleport Icon retrieved from Flaticon.com.

Uncollected Star Icon retrieved from Flaticon.com.

Collected Star Icon retrieved from Flaticon.com.

Pause Menu Icon retrieved from Flaticon.com.