Game Development Patterns with Unity 2021

Second Edition

Explore practical game development using software design patterns and best practices in Unity and C#

David Baron

Game Development Patterns with Unity 2021
Second Edition

Explore practical game development using software design patterns and best practices in Unity and C#

David Baron

BIRMINGHAM - MUMBAI

Game Development Patterns with Unity 2021
Second Edition

Group Product Manager: Ashwin Nair
Publishing Product Manager: Pavan Ramchandani
Senior Editor: Keagan Carneiro
Content Development Editor: Divya Vijayan
Technical Editor: Saurabh Kadave
Copy Editor: Safis Editing
Project Coordinator: Manthan Patel
Proofreader: Safis Editing
Indexer: Subalakshmi Govindhan
Production Designer: Aparna Bhagat

First published: March 2019
Second edition: July 2021

Production reference: 1290721

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80020-081-4

www.packt.com

This book is dedicated to my mother, Katia Galano, for her continuous words of encouragement and support and for always helping me through difficult times.

– David Baron

Contributors

About the author

David Baron is a game developer with over 15 years of experience in the industry. He has worked for some well-known AAA, mobile, and indie game studios in Montreal, Canada. His skill set includes programming, design, and 3D art. As a programmer, he has worked on various games for various platforms, including virtual reality, mobile, and consoles.

I would like to first and foremost thank my family for their continued support, patience, and encouragement throughout the long process of writing this book. I also want to give special thanks to the following individuals who offered me valuable feedback while writing this book: Nicolas Eypert, Guillaume Leroy, Dave Sirois, and Frédéric Lévesque.

About the reviewers

Lucas Bertolini has 10+ years' experience as a video game software developer. He has worked on three major projects: for Pollux Ltd. (Hong Kong) as a game developer and designer; for Schell Games (Pittsburgh, US) where he moved and worked as a developer until the project was completed; and for Globant as a developer.

He has worked in technical education for 5+ years and has taught a variety of programming courses. He is the cofounder of NGA and Bytenarchy Studios, both digital services development companies that use Unity as their main technology.

Lucas has written *Hands-On Game Development without Coding*, available from Packt.

Mark Bonasoro grew up in the late 80s in a family with computers from a young age. In the mid-90s, Mark started building his own PCs. His interest in game development started when he used Softimage on a Silicon Graphics machine in high school. Mark's programming skills ramped up in the 2000s when he started college, using C++ to make business applications, and worked in a small indie game company using a 3D multiplatform game engine, still in circulation, called Intrinsic Alchemy. In 2006, Mark started a game studio, as an excuse to learn with his classmates to make mobile games for Java 2 Micro Edition-featured phones for Sony Ericsson and Nokia. He graduated in software engineering in 2010 from Concordia University and worked as a Unity developer for Affordance Studios producing educational games.

> *I would like to thank David for all the opportunities he offered and am honored to be included as a technical reviewer for his latest book.*

Table of Contents

Preface

First principles, Clarice: simplicity. Read Marcus Aurelius.

"Of each particular thing, ask: What is it in itself? What is its nature?"

– Hannibal Lecter

The preceding quote is from one of my favorite films and sums up my approach to learning. Following over a decade working in the gaming industry, I have found that the only proper way to gain mastery over a complex system is by breaking it down into its most basic components. In other words, I try to understand the core ingredients before mastering the final form. Throughout this book, you will see that I'm taking a simplistic, but contextual, approach in presenting each pattern.

The goal is not to dumb down the subject matter but learn by isolating the core concepts behind each design pattern so that we can observe them and learn their intricacies. I've learned this approach in the gaming industry while working as a designer and programmer. We will often build components and systems for our game in isolated levels that we call *gyms*. We would spend weeks iterating, testing, and adjusting each ingredient of our game individually until we understood how to make them work as a whole. I wrote this book in a way that's consistent with how I approach game development so you, as a reader, can immerse yourself in the subject matter while adopting some good habits along the way that will help you in your career.

However, it is also important to state that the content of this book is not the ultimate reference regarding patterns in Unity. It's just an introduction to the subject matter, not the final destination of the learning process. I'm not presenting myself as the foremost expert and do not wish my words to become gospel among developers. I'm just a developer trying to find an elegant way of using standard software design patterns in Unity and want to share what I discovered. Therefore, as the reader, I encourage you to critique, research, customize, and improve upon everything presented throughout this book.

Who this book is for

While writing this book, I decided on a specific mental model of my target audience, for the main reason that it's almost impossible to write a book about game development and satisfy every potential type of reader, primarily because game development is a diverse industry and there are so many types of platforms and genres, each with their specific characteristics, that I cannot take into account in a single book. So I decided on focusing the content on a particular kind of audience, which I can describe as follows:

The target audience is game programmers who are currently working on a mobile or indie game project in the Unity engine and who are in the process of refactoring their code to make it more maintainable and scalable. The reader should have a basic understanding of Unity and the C# language.

Senior programmers working on large-scale AAA or MMO games might find particular examples in this book limited compared to the architectural challenges they usually face daily. However, on the other hand, the content of this book might offer another perspective on the use of design patterns in Unity. So feel free to skip any chapter if you already know the theory and want to see how I implemented a specific pattern.

What this book covers

`Chapter 1`, *Before We Begin*, is a brief introduction to the contents of this book.

`Chapter 2`, *The Game Design Document*, presents the design document behind the fully playable prototype of a racing game.

`Chapter 3`, *A Short Primer to Programming in Unity*, reviews some basic C# and Unity concepts.

`Chapter 4`, *Implementing a Game Manager with the Singleton*, covers the implementation of a globally accessible game manager with the infamous Singleton pattern.

`Chapter 5`, *Managing Character States with the State Pattern*, reviews the classic State pattern and how to encapsulate the stateful behaviors of a character.

`Chapter 6`, *Managing Game Events with the Event Bus*, covers the basic principles of the Event Bus pattern and how to use it to manage global game events.

`Chapter 7`, *Implementing a Replay System with the Command Pattern*, reviews how to use the Command pattern to build a replay system for a racing game.

`Chapter 8`, *Optimizing with the Object Pool Pattern*, covers how to use Unity's native implementation of the Object Pool pattern for performance optimization.

`Chapter 9`, *Decoupling Components with the Observer Pattern*, reviews how to decouple core components with the observer.

`Chapter 10`, *Implementing Power-Ups with the Visitor Pattern*, explains how to use the Visitor pattern to implement a customizable power-up game mechanic.

`Chapter 11`, *Implementing a Drone with the Strategy Pattern*, covers how to dynamically assign attack behaviors to enemy drones with the Strategy pattern.

`Chapter 12`, *Using the Decorator to Implement a Weapon System*, explains how to use the Decorator pattern as the foundation of a weapon attachment system.

`Chapter 13`, *Implementing a Level Editor with Spatial Partition*, reviews how to use the general concepts of Spatial Partition to build a level editor for a racing game.

`Chapter 14`, *Adapting Systems with an Adapter*, covers the Adapter pattern basics and how to use it to adapt a third-party library for reuse with a new system.

`Chapter 15`, *Concealing Complexity with a Façade Pattern*, uses the Façade pattern to hide complexity and establish a clean front-facing interface for a complex arrangement of interacting components.

`Chapter 16`, *Managing Dependencies with the Service Locator Pattern*, reviews the basics of the Service Locator pattern and how to use it to implement a system that allows the registration and location of specific services at runtime.

To get the most out of this book

To get the most out of this book, you need a basic understanding of the Unity engine. You will also need to be familiar with C# and have general knowledge of object-oriented programming. If you wish to reproduce the code presented in the upcoming chapters, you will need to download the latest version of Unity to your computer.

You can get the most recent build of Unity at the following link: `https://unity3d.com/get-unity/download`

The system requirements to run Unity can be found here: `https://docs.unity3d.com/Manual/system-requirements.html`

If you wish to download the source code made available on our GitHub repository, you will need a Git client; we recommend GitHub Desktop as it's the easiest one to use. You can download it at the following link: `https://desktop.github.com/`

Besides these tools, there are no other libraries or dependencies to download to run the code examples presented in this book.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Code in action videos for this book can be viewed at `https://bit.ly/2UNCZX1`.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800200814_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "This class initializes the `Context` object and the states, and it also triggers state changes."

A block of code is set as follows:

```
namespace Chapter.State
{
    public interface IBikeState
    {
        void Handle(BikeController controller);
```

```
        }
    }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
namespace Chapter.State
{
    public enum Direction
    {
        Left = -1,
        Right = 1
    }
}
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "These objects can publish specific types of events declared by the **Event Bus** to **Subscribers**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Game Development Patterns with Unity 2021*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Sections 1: Fundamentals

In this section of the book, we will review some design and programming fundamentals before moving on to the book's hands-on sections.

This section comprises the following chapters:

- `Chapter 1`, *Before We Begin*
- `Chapter 2`, *The Game Design Document*
- `Chapter 3`, *A Short Primer to Programming in Unity*

1
Before We Begin

Welcome to the second edition of *Hands-On Game Development Patterns with Unity*; this edition is not merely a revision of the previous version, but a complete upgrade of the original book. After the first edition came out, I was fortunate to get a lot of constructive feedback, which inspired me to improve the structure of this new edition. As we will review in the following sections, this book concentrates on the "hands-on" aspect of the title; in other words, we are going to get our hands dirty and work on implementing systems and features for a fully playable prototype of a game with design patterns. This new approach to the book's structure will be more tangible and also more enjoyable. It's more fun to work on a playable game than with random code examples. So, before we begin, in the following sections, I will establish specific parameters to the book's content and overall approach.

Let's quickly review the topics we are going to see in this chapter, as follows:

- Notes about the new edition
- The philosophy of the book
- What are design patterns?
- Which subjects aren't covered in this book?
- The game project

Notes about the new edition

As mentioned, I redesigned this edition entirely based on reader feedback from the previous version, and so, in consequence, I decided to cut some content from the last edition that readers considered to be trivial, such as the following:

- **Game loop** and **update pattern** chapters: These chapters focused too much on theory and didn't align themselves with the book's "hands-on" approach.
- **Anti-patterns** chapter: Anti-patterns is a complex and deep subject that deserves a book in itself to do it justice.

The end goal of this edition is not to cut content but to instead redesign the book to focus on practical game development uses of software design patterns in order to build a complete project. In other words, unlike the first edition, in which I took the approach of presenting each design pattern in isolation and with a self-contained code example, we will use them unitedly this time.

I added some chapters in this edition that were lacking in the previous version, such as the following:

- ***The Game Design Document***: The beginning of a new game project often starts with writing a GDD. A GDD is a document that will help us understand the design intention behind the game systems and mechanics that we will build throughout the book.
- ***A Short Primer to Programming in Unity***: We will use several advanced Unity engine concepts, C# features, and **object-oriented programming** (**OOP**) techniques throughout the book. In this chapter, we will take the time to review them to establish a shared knowledge base.

The philosophy of the book

This book is not a technical bible or the ultimate authority on how to use design patterns with Unity, and it's best described as a guide filled with design propositions to resolve some game programming challenges in Unity. The code examples included in each chapter are not flawless implementations because the art of design and programming is a continuous refinement process, and so the core goal of this book is to introduce you to potential solutions and inspire you to find better ones.

I always recommend seeking at least two sources of information on any technical subject, particularly concerning design patterns. It's essential to avoid getting too influenced by a single perspective of a complicated subject matter to the point that it becomes dogma instead of knowledge.

What are design patterns?

For those that are very new to programming, design patterns might be a novel concept. The simplest way of explaining design patterns is that they are reusable solutions to common software development problems. An architect named Christopher Alexander originated the notion of design patterns to describe reusable design ideas. In the late 1980s, inspired by the concept, software engineers started experimenting with applying concepts of reusable design patterns to software development, and over the years several books were written on the subject, such as the classic *Design Patterns: Elements of Reusable Object-Oriented Software* by the so-called "Gang of Four".

But in this book, I will be avoiding covering the academic side of software design patterns, focusing instead on their practical use for programming game mechanics and systems in Unity. I will present a recurrent game programming problem in each chapter and propose resolving it using a specific design pattern adapted for the Unity **application programming interface** (**API**).

Which subjects aren't covered in this book?

There are many facets to game programming, and a single book cannot cover them all with the depth they deserve. This book has a specific focus: design patterns and the Unity engine. So, if you are starting your journey into becoming a professional game programmer, this book will not be enough to complete your education. But luckily, some very talented individuals in our industry have taken the time to write very specialized books on core topics of game development. I recommend that anyone interested in joining the game industry reads each of the following reference books:

- Physics programming: *Real-Time Collision Detection, Christer Ericson*
- Engine programming: *Game Engine Architecture, Jason Gregory*
- **Three-dimensional** (**3D**) programming: *Mathematics for 3D Game Programming and Computer Graphics, Eric Lengyel*
- **Artificial intelligence** (**AI**) programming: *Programming Game AI By Example, Mat Buckland*

I have focused the content of this book on a specific aspect of game programming, but I will mention concepts from other domains of game development throughout the book. So, if you feel unfamiliar with some of the topics mentioned, take the time to explore them in depth; the time invested in researching will make you a better game programmer.

The game project

Throughout this book, we will be working continuously on a single game project example. The working title of the game is *Edge Racer*. As the title may indicate, it's a racing game; to be more specific, it's a futuristic racing game in which the player drives high-speed motorcycles. We will review the core concepts of the game in more detail in `Chapter 2`, *The Game Design Document*. But before continuing, I wish to list the reasons I decided on a racing game instead of another type of game—for example, a **role-playing game** (**RPG**)—as follows:

- **Simplicity**: Racing games have a simple premise—get to the finish line as fast as possible without crashing. Because this book is not about game design but game programming, I wanted a simple type of game that will permit us to focus on learning about software design patterns and not get bogged down with the implementation details of complex game mechanics.
- **Fun**: I've worked on various games of many different genres, and I always found that racing games are the most fun to develop because they are enjoyable to test. In racing games, you can speed-run to specific parts of the game and quickly reproduce bugs or test new features. Unlike other games with deep game mechanics and large maps, such as RPGs, racing games are usually quicker to debug.
- **Performance**: The main challenge of programming a racing game is maintaining a consistent frame rate as you add more features and content. So, I find working on racing games forces you to maintain good game-programmer habits by always keeping an eye on how fast your code is running and not just making it more readable.
- **Personal**: There's also a personal reason for me choosing a racing game—it's because it's my favorite genre. I love playing racing games and I love making them.

In conclusion, the game industry produces various products in many genres and sub-genres, but a racing game is a good reference point for us to start learning about design patterns in Unity because it's a simple context and forces us to keep an eye on keeping code clean and fast.

Summary

In this chapter, we reviewed the book's structure so that we can start with a clear understanding of its content and purpose. The key takeaway is that we will be using design patterns to build the mechanics and systems of a new racing game named *Edge Racer*.

In the next chapter, we will review the GDD to have a solid understanding of the game project that we will be working on in the upcoming chapters. I would not recommend skipping it because it's always a good practice to get to know as much as possible about a game project before starting writing code, as this helps in understanding how the parts fit the whole.

2
The Game Design Document

Before we start typing a single line of code, we need to complete a crucial step in a game's development cycle, which is creating a **Game Design Document** (**GDD**). A GDD is a blueprint of our entire project; its primary purpose is to put on paper the overall vision of our game's core concepts and act as a guide to a multidisciplinary team of developers during the arduous journey of a lengthy production cycle.

A GDD can include detailed descriptions of the following elements:

- Lists of core visual, animation, and audio ingredients
- Synopsis, character biographies, and narrative structures
- Marketing research material and monetization strategies
- Descriptions and diagrams that illustrate systems and mechanics

In most cases, a game designer is responsible for writing and maintaining the GDD, while a game programmer is responsible for implementing the described systems and mechanics in the document. Therefore, programmers and designers must give each other feedback throughout the entire process of formulating the GDD. If both sides fail to work together, the game designer will write up systems that are impossible to implement within a reasonable timeframe. Or programmers will waste time writing code for game systems that are defective in their overall design.

Considering that this book's focus is limited to programming and not game design, I will present a simplified and short version of the GDD for the game project we will be working on throughout this book. *The document section will also include some tips and notes for those who are not familiar with the process of analyzing a GDD.*

For reasons of brevity, I did not mention the participation of artists, animators, and audio engineers in the process of drafting the GDD. But their participation is essential because they need to know how many assets they will need to produce to realize the overall vision of the game.

In this chapter, we will cover the following topics:

- An overview of the game's synopsis and core mechanics
- An explanation of the core game loop and objectives
- A list of the game's ingredients and systems

The design document

The following game design document is divided into parts representing the individual point of interest in the overall project. For example, the game's synopsis section might be of interest to those working on the game's narrative elements. At the same time, the parts about minimum requirements and systems are geared toward the programmers. But whatever our specialization, it's good practice to read the entire design document and gain a complete mental model of the project before working on it.

Game overview

Blade Racer (*working title*) is a futurist arcade racing game in which the player attempts to control a turbo-charged motorcycle across an obstacle course to get to the finish line with the highest score. The race track is built on a railway system in which the player can steer a vehicle from one rail to the other to avoid obstacles. Across the pathways, the track system spawns pickup items in strategic positions that reward players with speed and shield boosts. Only players with razor-sharp reflexes will cross the finish line with the fastest time and without damaging their bike.

A working title is a temporary name for a project used during production. It's not final and can be changed when the project is ready for release.

Unique selling points

The following are potential unique selling points:

- International leaderboard with which players worldwide can compete to get their name on the top of the list
- The most challenging racing game on the market
- The game can be played on any device, from mobile to PC

Unique selling points are good to have in a GDD if you work with a publisher. It showcases that you have a vision of the final product and how you will market it. Take note that the examples above are placeholders and not conclusive.

Minimum requirements

The following are the minimum requirements for the current target platforms:

For **Mobile**:

- **OS**: Android 10
- **Model**: Samsung S9

For **PC**:

- **OS**: Windows 10
- **Graphics card**: Nvidia GTX 980
  ** We will support the equivalent ATI card from the same generation of the specified GTX card.*

Specifying minimum platform requirements right at the beginning of a project is very beneficial to game programmers. A platform with limited resources will demand more optimization of the visual assets and codebase for the game to run at a steady frame rate on all targeted devices.

Game synopsis

It's 2097. The human race has achieved mastery over the planet and now has set its sights on exploring beyond the edge of the solar system in the hope of facing new challenges and discovering new worlds. As peace prevails globally, those who have not left the planet experience a new form of collective boredom. Thanks to technology and biohacking, all the challenges of the past are gone. Anyone can be a top athlete, beautiful, and talented without any struggle.

Sensing the growing unrest in the global population, Neo Lusk, an eccentric tech entrepreneur and quadrillionaire, decides to invent a new sport involving turbo-charged motorcycles that can reach speeds most humans have problems handling. To make it even more challenging, he designs a unique race track system involving rails and obstacles.

This new sport invigorates the global population, and millions decide to participate. But few can control these high-tech bikes and maneuver them through a deadly layout of obstacles. Only the genuinely gifted with nerves of steel and reflexes as sharp as a blade can master this sport, but at the moment, the world is waiting to know who will be the first to be crowned its champion.

Even though our current project is an arcade racing game, which is a genre that's usually very light on story but heavy on gameplay, it's still good to have some narrative elements that provide some context to the game's world. It triggers the imagination of the player and gives an overall meaning to the game. Even classic arcade games such as Pac-Man and Missile Command have a certain degree of narrative ingredients that produce a feeling that you are playing a character inside a virtual world.

Game objectives

The list of core objectives (*a.k.a. player goals*) are the following:

- The main objective (*a.k.a.* ***win state***) is to cross the finish line without any fatal crashes.
- The secondary objective is to get the best score by earning "risk" points by taking risky racing lines when swerving between obstacles.
- As a bonus objective, a player can get extra points by picking up a specific amount of collectible items (*a.k.a.* ***pickups***) that are spawned across the tracks. *This objective is aimed at "completionists."*
- The ultimate objective of the game is to get the highest score while finishing the race with the fastest time.

A good game designer will always design several objectives to satisfy the various kinds of players to keep them engaged as long as possible. For example, a player with a "completionist" mindset doesn't simply want to finish a game but complete it by attaining every possible objective and getting every collectible item.

The player can fail (*a.k.a.* ***fail state***) at achieving the core objectives with the following actions:

- The main "fail state" is triggered when a player doesn't arrive at the finish line because of a fatal crash.
- If the checkpoint game mode is activated, the player can fail by not crossing each checkpoint in time.

It's good practice to also clearly define all possible "fail conditions" of a game early on because it helps analyze how many distinct wins and fail state conditions we will need to handle.

Game rules

The game level is an open environment where the player controls a motorcycle that runs at high speed on a rail system composed of four tracks. The player can navigate the vehicle sideways to avoid obstacles or capture "pickup" items spawned along the tracks. The player must race to the finish line while reaching the checkpoints that are set along the paths within the minimum required time while avoiding damaging the vehicle by steering clear of obstacles. The player can score points by choosing risky racing lines such as turning to avoid an impediment within inches of colliding with it. The player can also get bonus points by capturing a specific number of "pickup" items in the correct order.

Game loop

The following diagram illustrates the core game loop:

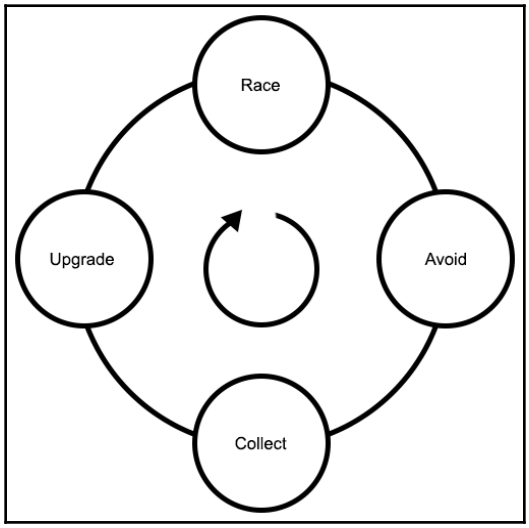


Figure 2.1 – Diagram of the core game loop

The game has four pillars in its core game loop:

- **Race**: The player races against the clock toward the finish line.
- **Avoid**: The player must avoid obstacles while going as fast as possible while taking risks to gain points.
- **Collect**: The player can pick up items and rewards placed on the race track during the race.
- **Upgrade**: During or after the race, the player can upgrade the vehicle with the collected items.

Game environment

The primary setting for each level is a futuristic deserted space with four rails that span the horizon. There are metallic light polls and barriers on the side of the railway. The sky is cloudy, and a constant fog obscures the skyline while rainstorms often occur. On the tracks, there are various obstacles and items for the player to avoid or capture. The level design is focused on punishing slow players and rewarding those that go fast and take risks.

Camera, control, character (3Cs)

3Cs (**camera**, **control**, and **character**) is an industry-standard term that is used in the AAA industry in studios such as *Ubisoft*. It basically states that the core of a player's experience is defined by the balance between the harmonious relationship of the camera, the player character, and the control scheme. If a player feels that the character responds to inputs as expected while the camera seamlessly positions itself at an optimal angle, then the overall experience will be more immersive. Let's look at the 3Cs in detail.

Camera

The main camera will follow the vehicle from the rear in a third-person view. It will automatically position and rotate itself on the nearest track opposite the vehicle's direction during a turn. During a turbo boost, the camera will pull back and start shaking slightly to give the sense of having difficulty following the vehicle as it speeds away. And when the motorbike slows back down to normal speed, the distance between the camera and motorcycle will snap back to its default distance, as we can see in the following diagram:

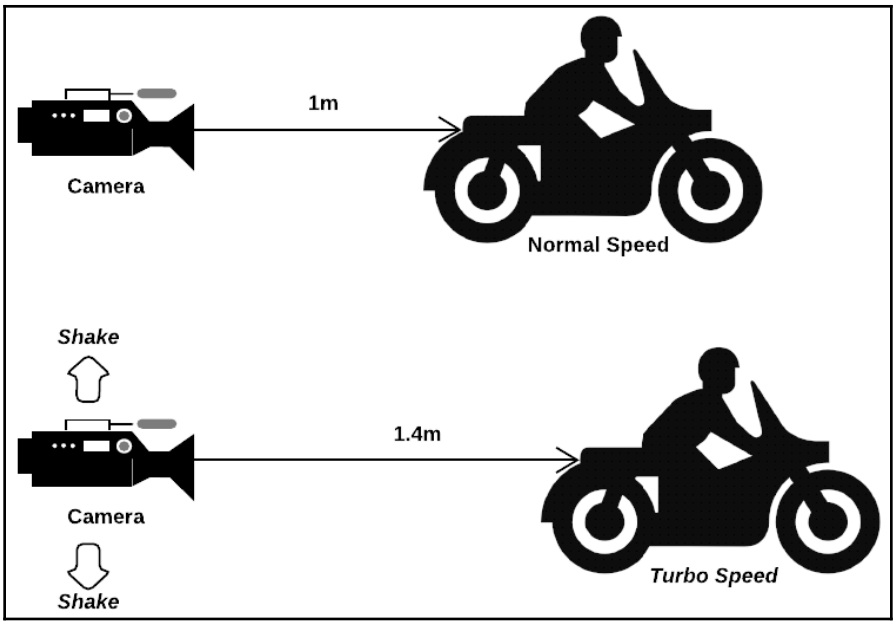


Figure 2.2 – Illustration of the camera positions

In this current iteration of the game, there will be no first-person or rear-view camera.

Character

In this section, we will define the main player-controlled character.

Character description

The main playable character is a human pilot that drives a futurist motorcycle vehicle that runs on a railway system and is limited to maneuvering from one rail to another at very high speeds. Considering that the main playable character is inside the vehicle's cockpit at all times, we can consider the vehicle and the pilot as one playable character during gameplay segments. The game will not have any other playable humanoid characters or types of controllable vehicles.

Character metrics

The following section outlines the primary character/vehicle metrics:

- **Min Speed**: 500 km/h
  The vehicle is always moving during the race and can never completely stop.
- **Max Speed**: 6,500 km/h
  The vehicle can only stay at max speed for a couple of seconds and its handling capacity will reduce to its minimum amount.
- **Max Health**: 100%
  The vehicle has a damage meter calculated as a percentage.
- **Attack Damage**: *Depends on the weapon.*
  The vehicle has no default weapon system but has an attachment slot for upgrades.
- **Handling**: *Depends on the current speed.*
  The vehicle's handling capacity is reduced as it reaches its top speed.

Character states

The following section outlines the primary character/vehicle states:

- **Idle**: The idle state of the vehicle is an animation cycle that shows the chassis vibrating from the roaring engine and the taillights blinking.
- **Speeding**: The speeding state has an animation cycle of the wheels turning.
- **Slowing**: The slowing animation slows down the wheels and flashes the taillights.
- **Turning**: The turning state has an animation of the vehicle tilting and moving toward the direction of the nearest track.
- **Braking:** The braking animation will make the vehicle tilt and slide sideways on a forward trajectory. This state is activated only once the player crosses the finishing line.
- **Shaking:** The shaking animation rises in intensity as the handling capacity of the vehicle is reduced at top speeds.
- **Flipping**: The flipping animation flips the vehicle forward when it collides with a roadblock obstacle.
- **Sliding**: The sliding animation turns the vehicle sideways and slides in a forward direction.

It's important right from the start that the design and programming team define each type of playable controllable character, device, or vehicle in the game. Mainly because implementing 3Cs for a car or a two-legged humanoid character demands different approaches. Like in the real world, driving and walking are different types of experiences with specific mechanical intricacies.

Controller

The following section outlines the primary controller scheme, which is the keyboard:

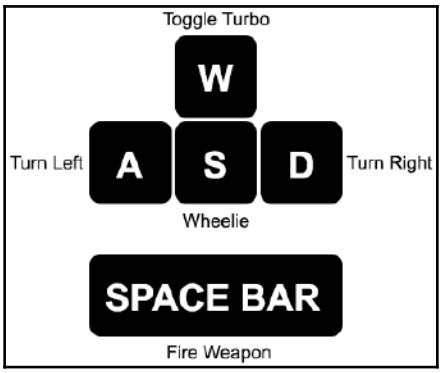


Figure 2.3 – Diagram of the keyboard input keys

We will only support the keyboard as the primary input device for this book's game project for reasons of simplicity.

Game ingredients

The following section outlines the core game ingredients:

- **Bikes** are the main vehicle of the game and can be controlled by the player or **AI** (**Artificial Intelligence**).
- **Pickups** are items that are spawned on the tracks and that the player can pick up by colliding with them. Each type of item will have a distinct shape and will float a couple of inches above the track.

- **Obstacles** are environment ingredients that are spawned on the tracks. Certain types of obstacle entities can damage the vehicle while others block the vehicle and cause it to crash.
- **Weapons** are ingredients that can be attached to an available slot on the player's vehicle. The player's weapon ingredients are laser-based and permit them to destroy obstacles.
- **Drones** are robotic entities that fly around in flight patterns and attack the player with laser beams.

Superbikes

Name	Description	Upgrade Slots	Handling	Top Speed
Venom	The entry-level model for beginner racers.	1	80%	1,900 km/h
Silver Bullet	The advanced model for intermediate racers.	2	60%	4,000 km/h
Death Rider	The most advanced model, only accessible to top players.	4	40%	6,500 km/h

The bike's handling is defined by how quickly the vehicle reacts to the player inputs. As the speed increases, the overall stability decreases. Therefore, the bike will start sliding while turning and will take longer to stabilize and respond to the player's commands.

Pickups

Name	Description	Effect	Value
Collectible	A collectible item is spawned on the length of a track in numbered order.	Increments the risk score	10 risk points
Turbo Boost	This item is spawned on the tracks at specified locations.	Fills up the turbo meter	10% turbo boost
Damage Repair * a.k.a. Health Restore	This item is spawned on the tracks at specific locations.	Repairs damage	10% of heath

Obstacles

Name	Description	Effect	Damage
Track Barrier	A tall and flat barrier that blocks the track but breaks on impact.	Causes damage but doesn't stop the vehicle	20% of health
Road Block	A short barricade stops the vehicle by making it flip forward and crash.	Stops the vehicle	10o% of heath

Weaponry

Name	Description	Effect	Range
Laser Shooter	Singular ray cast weapon	Destroys everything within range	10 meters
Plasma Blaster	Multiple ray cast weapons	Destroys everything within range and a field of view of 60 degrees	20 meters

Game systems

The following section outlines the core game systems:

- **Rail system**
  The rail system permits the player's vehicle to navigate sideways on four individual tracks. Each track can spawn obstacles and pickups along its length.

- **Risk system**
  The risk system rewards a player that takes risks, for example, by swerving away from obstacles in the nick of time at top speed. The number of points given is based on the distance between the vehicle and the barrier when the player avoided it. A sensor installed in front of the bike with a range of 5 meters is constantly monitoring for any obstructions on the path of the current track. Once a potential collision is detected, the sensor calculates the distance between the obstacles and the vehicle. Points are determined based on the distance between the player and when the hindrance was avoided.

The following diagram illustrates the forward-facing sensor:

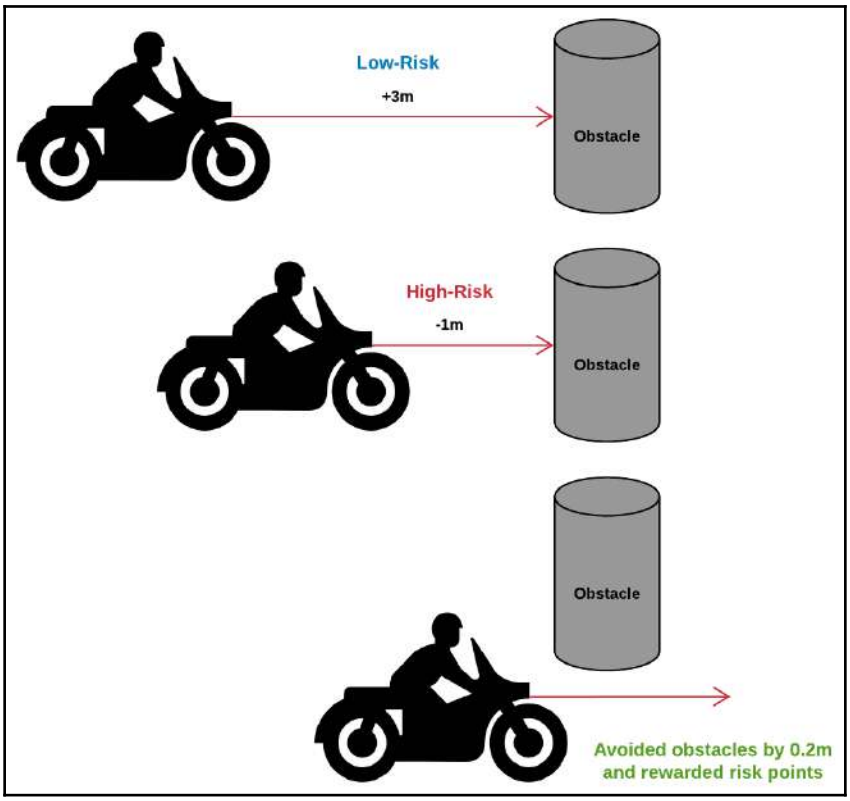


Figure 2.4 – Diagram of the forward-facing sensor detecting obstacles

- **Turbo Boost System**
  The **Turbo Boost System** (**TBS**) permits the vehicle to reach its top speeds. To activate the system, the player must fill up the turbo meter by picking up turbo boost items. The TBS system has a gauge on the **Head-Up Display** (**HUD**) that permits the player to visualize the amount of turbo boost available. When the TBS is engaged, the vehicle handling is lessened and it is more vulnerable to damage.
- **Vehicle upgrade system**
  The player's vehicle has item slots that permit them to attach weapons and other upgrades. Depending on the type of vehicle, the number of slots will vary.

The difference between a game mechanic and a system is sometimes subject to confusion. In this book, I will define a game system as a collection of game mechanics and ingredients. For example, weaponry pickups are ingredients, but the actions of picking them up on the battlefield and then selecting the right weapon in the inventory menu to win a battle are individual mechanics. But all these elements are distinct components of an entire weapon system.

Game menu

In this section, we'll outline the game's menus:

Main Menu: The main menu will be accessible at the start of the game. The menu options are as follows:

- **Start Race**: This option loads the race track scene.
- **Scores**: This option displays the latest top 10 scores.
- **Exit**: This option closes the game's window.

In-Game Menu: The in-game menu is only accessible during the race. The menu options are as follows:

- **Restart Race**: Restarts the race from the beginning
- **Exit Race**: Returns the player back to the lobby scene

Game HUD

The following section outlines the game's **Head-up Display** (**HUD**):

The HUD will only be displayed during the race and will include the following interface components:

- **Turbo Meter**
- **Speed Meter**
- **Damage Meter**
- **Countdown Timer**
- **Track Progress Bar**

Often, the final design of a game's menus and HUDs is completed in later stages of production. Mostly because the overall design, art style, and structure of a game can change several times during the earlier stages of production, therefore, the flow of the menus and the layout of the components of the HUDs are subject to constant change.

Summary

In this chapter, we took the time to review the current draft of the design document of the book's game project. This draft of the GDD might not be complete or perfect, but often, game projects start with incomplete design documentation and ideas. So it's still good that we took the time to review it. We will not implement every system, mechanic, and ingredient described in this book because our primary goal is to learn design patterns by building a game, not necessarily a complete one.

Even though game design is not the main subject of this book, learning to think like a designer is a skill that can only help anyone become a better game programmer. Reading design documentation helps us understand the game's vision, structure, mechanics, and scope. And these are things to keep in mind when making software architecture decisions.

The next chapter includes a short primer on basic Unity programming fundamentals. If you are already an experienced programmer, you can skip to the hands-on section of the book. But if you are a beginner, I would recommend reading the primer before implementing the upcoming chapters' design patterns.

Further reading

For more information, you can refer to the following books:

- *The Art of Game Design* by Jesse Schell
- *Game Design: Secrets of the Sages* by Marc Saltzman
- *Level Up! The Guide to Great Video Game Design* by Scott Rogers
- *Rules of Play: Game Design Fundamentals* by Katie Salen and Eric Zimmerman

3
A Short Primer to Programming in Unity

This chapter is a short primer to help you get familiar with advanced C# language and Unity engine features. I'm not going to attempt to explain in depth any of the subject matter presented here because it's beyond the scope of this book. Still, I will at least introduce you to some core concepts to avoid confusion when they are referenced in the upcoming chapters. I encourage those that have advanced knowledge of C# and Unity programming to skip this chapter. But I recommend beginners and intermediate developers take the time to review the content of this chapter to get a general idea of the language and engine features we will use to implement and design our game systems.

In all cases, complete mastery over C# and Unity is not needed to comprehend this book, just general awareness and familiarity with some advanced critical concepts.

In this chapter, we will go through the following topics:

- What you should already know
- C# language features
- Unity engine features

The code showcased in this chapter is for educational purposes only. It has not been optimized and is not meant to be used in production code.

What you should already know

In this section, I'm listing some core C# language and Unity engine features that you should already be familiar with before continuing to more advanced parts of this book.

The following are some core features of C#:

- Familiarity with class access modifiers such as public and private
- Fundamental knowledge of basic primitive data types (int, string, bool, float, and arrays)
- A conceptual understanding of inheritance and the relation between a base class and a derived class

The following are some core features of Unity:

- A basic understanding of how to write a `MonoBehaviour` script and attach it to a `GameObject` as a component
- Ability to create a new Unity Scene from scratch and manipulate GameObjects inside the editor
- Familiarity with Unity's basic event functions (`Awake`, `Start`, `Update`) and their sequence of execution

If you are not familiar with the concepts listed previously, I would recommend reading the books and documentation that are listed in the *Further reading* section of this chapter.

C# language features

Languages features such as events and delegates might be too advanced for beginners, so if you consider yourself in that category, don't worry; you can still enjoy this book. Just read the beginner-level chapters such as the ones that explain patterns such as Singleton, State, Facade, and Adapter.

The following C# advanced language features are fundamental to the optimal implementation of some design patterns that we will be implementing in the upcoming chapters:

- **Static**: Methods and members of a class with the `static` keyword can be accessed directly with its name and without initializing an instance. Static methods and members are helpful because they are easily accessible from anywhere in your code. The following example showcases a class that uses the keyword to establish a globally accessible event bus:

```
using UnityEngine.Events;
using System.Collections.Generic;

namespace Chapter.EventBus
{
    public class RaceEventBus
    {
        private static readonly
            IDictionary<RaceEventType, UnityEvent>
            Events = new Dictionary<RaceEventType, UnityEvent>();

        public static void Subscribe(
            RaceEventType eventType, UnityAction listener)
        {
            UnityEvent thisEvent;
            if (Events.TryGetValue(eventType, out thisEvent))
            {
                thisEvent.AddListener(listener);
            }
            else
            {
                thisEvent = new UnityEvent();
                thisEvent.AddListener(listener);
                Events.Add(eventType, thisEvent);
            }
        }

        public static void Unsubscribe(
            RaceEventType eventType, UnityAction listener)
        {
            UnityEvent thisEvent;
            if (Events.TryGetValue(eventType, out thisEvent))
            {
                thisEvent.RemoveListener(listener);
            }
        }
```

```
        public static void Publish(RaceEventType eventType)
        {
            UnityEvent thisEvent;
            if (Events.TryGetValue(eventType, out thisEvent))
            {
                thisEvent.Invoke();
            }
        }
    }
}
```

- **Events**: An event permits an object that acts as a publisher to send out a signal that other objects can receive; these objects that listen for a particular event are called subscribers. Events are useful when you want to build an event-driven architecture. The following is an example of a class that publishes events:

```
using UnityEngine;
using System.Collections;

public class CountdownTimer : MonoBehaviour
{
    public delegate void TimerStarted();
    public static event TimerStarted OnTimerStarted;
    public delegate void TimerEnded();
    public static event TimerEnded OnTimerEnded;
    [SerializeField]
    private float duration = 5.0f;
    void Start()
    {
        StartCoroutine(StartCountdown());
    }

    private IEnumerator StartCountdown()
    {
        if (OnTimerStarted != null)
            OnTimerStarted();
        while (duration > 0)
        {
            yield return new WaitForSeconds(1f);
            duration--;
        }
        if (OnTimerEnded != null)
            OnTimerEnded();
    }
    void OnGUI()
    {
        GUI.color = Color.blue;
        GUI.Label(
```

```
                new Rect(125, 0, 100, 20), "COUNTDOWN: " + duration)
        }
    }
```

- **Delegates**: The concept behind delegates is simple when you understand their underlying low-level mechanism. A high-level definition of delegates is that they hold references to functions. But this is a very abstract definition of what delegates actually do behind the scenes. They're function pointers, which means that they hold the memory address to other functions. So, we could visualize them as an address book that contains a list of locations of functions. That is why a delegate can hold multiple of them and call them all at once. The following is an example of a class that subscribes to events triggered by a publisher class by assigning specific local functions to the publisher's delegates:

```
using UnityEngine;

public class Buzzer : MonoBehaviour
{
    void OnEnable()
    {
        // Assigning local functions to delegates defined in the
        // Timer class
        CountdownTimer.OnTimerStarted += PlayStartBuzzer;
        CountdownTimer.OnTimerEnded += PlayEndBuzzer;
    }

    void OnDisable()
    {
        CountdownTimer.OnTimerStarted -= PlayStartBuzzer;
        CountdownTimer.OnTimerEnded -= PlayEndBuzzer;
    }

    void PlayStartBuzzer()
    {
        Debug.Log("[BUZZER] : Play start buzzer!");
    }

    void PlayEndBuzzer()
    {
        Debug.Log("[BUZZER] : Play end buzzer!");
    }
}
```

- **Generics**: A relevant C# feature that permits the deferred specification of a type until the class is declared and instantiated by the client. When we say a class is generic, it doesn't have a defined object type. The following is an example of a generic class and can act as a template:

```
// <T> can be any type.
public class Singleton<T> : MonoBehaviour where T : Component
{
    // ...
}
```

- **Serialization**: Serialization is the process of converting an instance of an object into a binary or textual form. This mechanism permits us to preserve the state of an object in a file. The following is an example of a function that serializes an instance of an object and saves it as a file:

```
private void SerializePlayerData(PlayerData playerData)
{
    // Serializing the PlayerData instance
    BinaryFormatter bf = new BinaryFormatter();
    FileStream file = File.Create(Application.persistentDataPath +
        "/playerData.dat");
    bf.Serialize(file, playerData);
    file.Close();
}
```

C# is a programming language with a lot of depth so it would be impossible for me to explain each of its core features within the scope of this book. Those presented in this section of this book are very useful and will help us implement the game systems and patterns described in the upcoming chapters.

Unity engine features

Unity is a fully featured engine, including a comprehensive scripting API, an animation system, and many additional features for game development. We can't cover them all in this book, so I will only list the core Unity components that we will be using in the upcoming design pattern chapters:

- **Prefabs**: A prefab is a prefabricated container of assembled GameObjects and components. For example, you can have individual prefabs for each type of vehicle in your game and dynamically load them in your scene. Prefabs permit you to construct and organize reusable game entities as building blocks.

- **Unity Events and Actions**: Unity has a native event system; it's very similar to the C# event system but with extra engine-specific features, such as the ability to view and configure them in the Inspector.
- **ScriptableObjects**: A class that derives from the `ScriptableObject` base class can act as a data container. The other native Unity base class, named MonoBehaviour, is used to implement behaviors. Therefore, it's recommended to use MonoBehaviours to contain your logic and ScriptableObjects to contain your data. An instance of a `ScriptableObject` can be saved as an asset and is often used for authoring workflows. It permits non-programmers to create new variations of a specific type of entity without a single line of code.

The following is an example of a simple `ScriptableObject` that permits creating new configurable `Sword` instances:

```
using UnityEngine;

[CreateAssetMenu(fileName = "NewSword", menuName =
"Weaponary/Sword")]
public class Sword: ScriptableObject
{
    public string swordName;
    public string swordPrefab;
}
```

- **Coroutines**: The concept of coroutines is not limited to Unity but is an essential tool of the Unity API. The typical behavior of a function is to execute itself from start to finish. But a coroutine is a function with the extra ability of waiting, timing, and even pausing its execution process. These additional features permit us to implement complex behaviors that are not easy to do with conventional functions. Coroutines are similar to threads but provide concurrency instead of parallelism. The following code example showcases the implementation of a countdown timer using coroutines:

```
using UnityEngine;
using System.Collections;

public class CountdownTimer : MonoBehaviour
{
    private float _duration = 10.0f;

    IEnumerator Start()
    {
        Debug.Log("Timer Started!");
        yield return StartCoroutine(WaitAndPrint(1.0F));
        Debug.Log("Timer Ended!");
```

```
        }

        IEnumerator WaitAndPrint(float waitTime)
        {
            while (Time.time < _duration)
            {
                yield return new WaitForSeconds(waitTime);
                Debug.Log("Seconds: " + Mathf.Round(Time.time));
            }
        }
    }
```

As we can see, Unity has some rich but straightforward features that permit us to implement systems and organize data. We can't cover in depth each of Unity's core features because it's beyond the intended scope of the book. If you feel that you need more information about the engine before moving along, I recommend reviewing the material linked under the *Further reading* section.

Summary

This chapter is intended to be used as a primer to set up a shared knowledge base before moving along to the book's hands-on section. But mastery over the features presented in the previous sections is not required to start using design patterns in Unity. We will review some of them again in more detail in upcoming chapters, this time in a practical context.

In the next chapter, we are going to review our first pattern, the infamous Singleton. We will use it to implement a Game Manager class responsible for initializing the game.

Further reading

For more information, you can refer to the following material:

- *Learning C# by Developing Games with Unity 2020* by Harrison Ferrone
- *Unity User Manual* by Unity Technologies: `https://docs.unity3d.com/Manual/index.html`

Section 2: Core Patterns

In this section of the book, we will review some fundamental patterns and use them to build our game's core systems and mechanics.

This section comprises the following chapters:

- `Chapter 4`, *Implementing a Game Manager with the Singleton*
- `Chapter 5`, *Managing Character States with the State Pattern*
- `Chapter 6`, *Managing Game Events with the Event Bus*
- `Chapter 7`, *Implementing a Replay System with the Command Pattern*
- `Chapter 8`, *Optimizing with the Object Pool Pattern*
- `Chapter 9`, *Decoupling Components with the Observer Pattern*
- `Chapter 10`, *Implementing Power-Ups with the Visitor Pattern*
- `Chapter 11`, *Implementing a Drone with the Strategy Pattern*
- `Chapter 12`, *Using the Decorator to Implement a Weapon System*
- `Chapter 13`, *Implementing a Level Editor with Spatial Partition*

4
Implementing a Game Manager with the Singleton

In this first hands-on chapter, we will review one of the most infamous software design patterns in the field of programming, the **Singleton**. It could be argued by many that the Singleton is the most widely used pattern among Unity developers, maybe because it's the most straightforward pattern to learn. But it can also quickly become the "duct tape" in our programming toolbox that we reach for every time we need a quick fix for a complex architectural problem.

For instance, when using this pattern, we can quickly establish a simple code architecture revolving around wrapping and managing all the core systems of our game in individual manager classes. Then we could have these managers expose clean and straightforward interfaces that will conceal the inner complexity of the systems. Also, to make sure that these managers are easily accessible and only a single instance runs at a time, we would implement them as Singletons. This approach might sound solid and beneficial, but it's full of pitfalls as it will create strong coupling between core components and make unit testing very difficult.

In this book, we will attempt to move away from this type of architecture and use design patterns to establish a more robust, modular, and scalable code base. But this doesn't mean that we will ignore the Singleton and judge it as inherently faulty. Instead, in this chapter, we will explore a use case in which this pattern is well suited.

The following topics will be covered in this chapter:

- The basics of the Singleton pattern
- Writing a reusable Singleton class in Unity
- Implementing a globally accessible GameManager

Technical requirements

This is a hands-on chapter; you will need to have a basic understanding of Unity and C#.

We will be using the following specific Unity engine and C# language concept: **Generics**.

If unfamiliar with this concept, please review `Chapter 3`, *A Short Primer to Programming in Unity*.

The code files of this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter04`.

Check out the following video to see the Code in Action:
`https://bit.ly/3wDbM6W`

Generics is a compelling C# feature that permits us to defer the type for a class at runtime. When we say a class is generic, it means that it doesn't have a defined object type. This approach is advantageous because we can assign it a specific type when we initialize it.

Understanding the Singleton pattern

As its name implies, the Singleton pattern's primary goal is to guarantee singularity. This approach means if a class implements this pattern correctly, once initialized, it will have only one instance of itself in memory during runtime. This mechanism can be helpful when you have a class that manages a system that needs to be globally accessible from a singular and consistent entry point.

The design of the Singleton is quite simple. When you implement a Singleton class, it becomes responsible for making sure there's only a single occurrence of itself in memory. Once a Singleton detects an instance of an object of the same type as itself, it will destroy it immediately. Therefore, it's pretty ruthless and doesn't tolerate any competition. The following diagram illustrates the process to a certain degree:

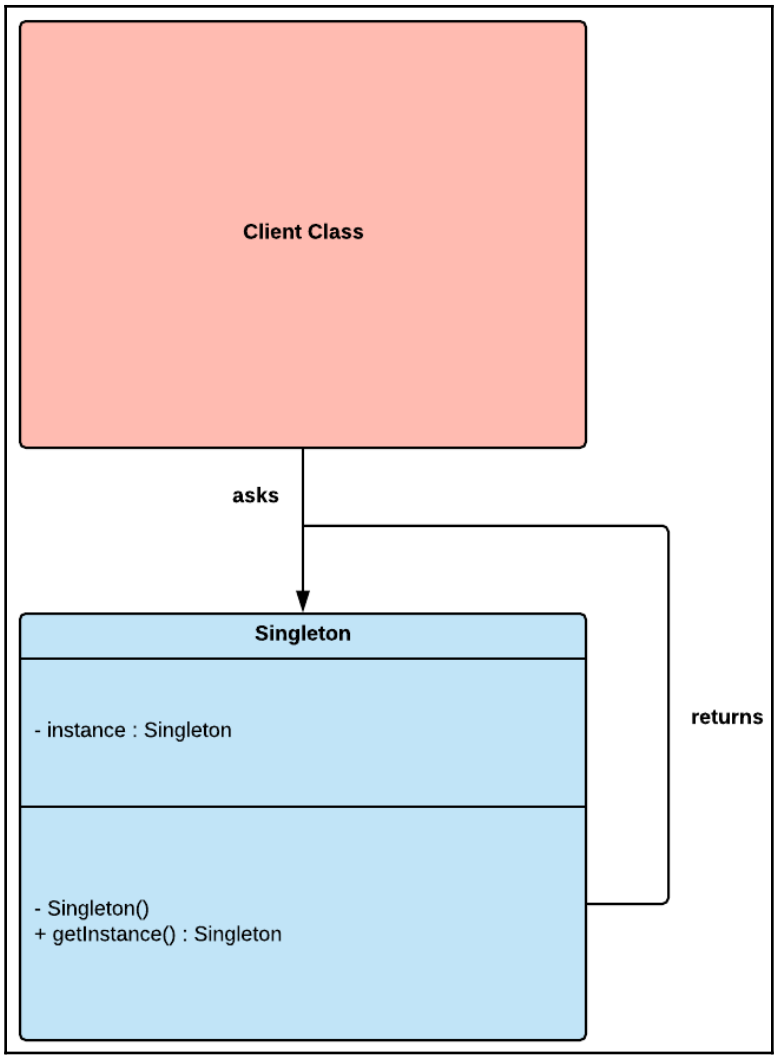


Figure 4.1 – UML diagram of the Singleton pattern

The most important takeaway from this description of the Singleton pattern is that if well implemented, it makes sure that there can only be one of itself; if not, it fails at its purpose.

Benefits and drawbacks

These are some of the benefits of the Singleton pattern:

- **Globally accessible**: We can use the Singleton pattern to create a global access point to resources or services.
- **Control concurrency**: The pattern can be used to limit concurrent access to shared resources.

These are some of the drawbacks of the Singleton pattern:

- **Unit testing**: If overly used, the Singleton can make unit testing very difficult. We might end up with Singleton objects being dependent on other Singletons. If one is missing at any moment, the chain of dependency gets broken. This issue often happens when combining Facade and Singleton to set up front-facing interfaces to core systems. We end up with an array of manager classes, each managing a specific core component of the game, all dependent on each other to function. Therefore, it becomes impossible to test and debug in isolation.
- **Laziness**: Because of its ease of use, the Singleton is a pattern that can quickly instill faulty programming habits. As mentioned in the *Unit testing* drawback, we can easily make everything accessible from anywhere with the Singleton. The simplicity it offers can also make us unwilling to test out more sophisticated approaches when writing code.

When making design choices, it's essential to always keep in mind whether your architecture is maintainable, scalable, and testable. When it comes to testable, I often ask myself whether I can easily test my core systems, components, and mechanics individually and in isolation. If not, then I know I made some potentially unwise decisions.

Designing a Game Manager

A standard class we often see in Unity projects is the Game Manager. It's usually implemented as a Singleton by developers, but its responsibility varies from one code base to another. Some programmers use it to manage top-level game states or as a globally accessible front-facing interface to core game systems.

In the context of this chapter, we will give it the singular responsibility of managing a game session. Similar to the concept of a game master in board gaming, it will be responsible for setting up the game for the player. It can also take on additional responsibilities, such as communicating with backend services, initializing global settings, logging, and saving the player's progress.

The critical thing to keep in mind is that Game Manager will be alive for the entire lifespan of the game. Therefore, there will be a singular but persistent instance of it in memory at all times.

The following diagram illustrates the overall concept:

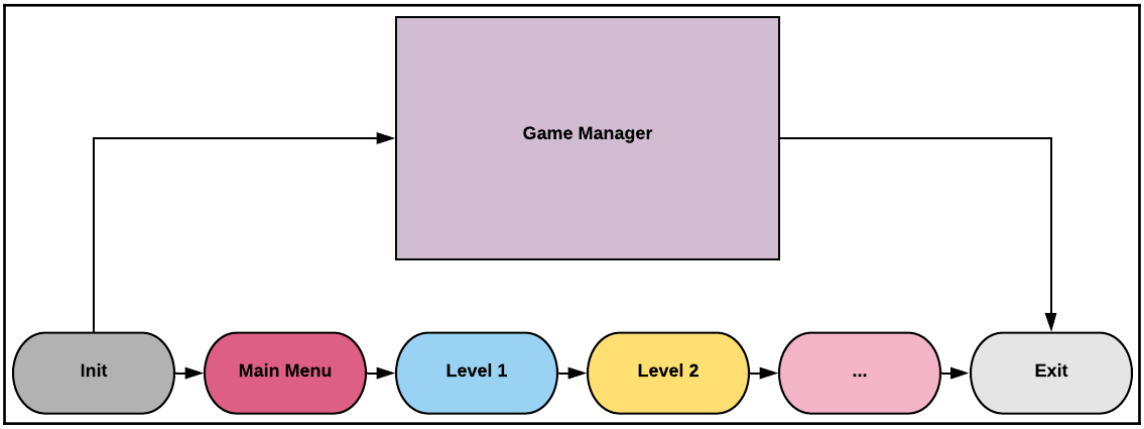


Figure 4.2 – Diagram that illustrates the lifespan of the Game Manager

In the next section, we are going to take the design we just reviewed and translate it into code.

Implementing the Game Manager

In this section, we will implement a Singleton and Game Manager class. We will attempt to utilize some core Unity API features to adapt the pattern for use in the engine:

1. For the first step of the process, we will implement the `Singleton` class. To make it easier to understand its intricacies, we will split it up into two distinct segments:

```
using UnityEngine;
```

```
namespace Chapter.Singleton
{
    public class  Singleton<T> :
        MonoBehaviour where T : Component {

        private static T _instance;

        public static T Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = FindObjectOfType<T>();

                    if (_instance == null)
                    {
                        GameObject obj = new GameObject();
                        obj.name = typeof(T).Name;
                        _instance = obj.AddComponent<T>();
                    }
                }

                return _instance;
            }
        }
```

In the first segment of the `Singleton<T>` class, we can see that we implemented a `public static` property with a `get` accessor. In this accessor, we are making sure that there's no existing instance of this object before initializing a new one. `FindObjectOfType<T>()` searches for the first loaded object of a specified type. If we can't find one, then we create a new `GameObject`, rename it, and add a component to it of a non-specified type.

This process will be more evident when we implement the `GameManager` class.

2. Let's implement the final segment of the `Singleton` class:

```
        public virtual void Awake()
        {
            if (_instance == null)
            {
                _instance = this as T;
                DontDestroyOnLoad(gameObject);
            }
            else
            {
```

```
                Destroy(gameObject);
            }
        }
    }
}
```

For the last segment of the class, we have an `Awake()` method that we marked as `virtual`, which means it can be overridden by a derived class. What is essential to understand is that when the `Awake()` method gets called by the engine, the Singleton component will check whether there's already an instance of itself initialized in memory. If not, then it will become the current instance. But if one already exists, it will destroy itself to prevent duplication.

Therefore, there can only be one instance of a specific type of Singleton in a Scene at once. If you try to add two, one will get automatically destroyed.

Another important detail to review is the following line:

```
DontDestroyOnLoad(gameObject);
```

`DontDestroyOnLoad` is a public static method that is included in the Unity API; it prevents a target object from being destroyed when a new scene is loaded. In other words, it makes sure that the current instance of an object persists even when switching between scenes. This API feature is handy for our Singleton because it guarantees the object will be available throughout the application's lifespan, in this context, the game.

3. For the final steps of our implementation, we will write a skeleton version of the `GameManager` class. We will focus only on code that will validate our `Singleton` implementation for reasons of brevity:

```
using System;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Chapter.Singleton
{
    public class GameManager : MonoBehaviour
    {
        private DateTime _sessionStartTime;
        private DateTime _sessionEndTime;

        void Start() {
            // TODO:
            // - Load player save
            // - If no save, redirect player to registration scene
```

```
                // - Call backend and get daily challenge and rewards

                _sessionStartTime = DateTime.Now;
                Debug.Log(
                    "Game session start @: " + DateTime.Now);
            }
            void OnApplicationQuit() {
                _sessionEndTime = DateTime.Now;

                TimeSpan timeDifference =
                    _sessionEndTime.Subtract(_sessionStartTime);

                Debug.Log(
                    "Game session ended @: " + DateTime.Now);
                Debug.Log(
                    "Game session lasted: " + timeDifference);
            }

            void OnGUI() {
                if (GUILayout.Button("Next Scene")) {
                    SceneManager.LoadScene(
                        SceneManager.GetActiveScene().buildIndex + 1);
                }
            }
        }
    }
```

To give more context to `GameManager`, we left a `TODO` list of potential tasks for the class to accomplish. But we also added a timer and a GUI button. Both will help us validate whether our `Singleton` is working when we start the testing phase.

But at the moment, our `GameManager` is not a `Singleton`; to make it one, we just need to make one change to a single line of code, as we can see here:

```
public class GameManager : Singleton<GameManager>
```

It's as simple as that; we took a regular `MonoBehaviour` class and converted it into a `Singleton` with one line of code. This is made possible because we are using Generics. Hence, our `Singleton` class can be anything until we assign it a specific type.

4. So, for our last step, we took our `GameManager` class and converted it into `Singleton`, as seen here:

```
using System;
using UnityEngine;
```

```
using UnityEngine.SceneManagement;

namespace Chapter.Singleton
{
    public class GameManager : Singleton<GameManager>
    {
        private DateTime _sessionStartTime;
        private DateTime _sessionEndTime;

        void Start() {
            // TODO:
            // - Load player save
            // - If no save, redirect player to registration scene
            // - Call backend and get daily challenge and rewards

            _sessionStartTime = DateTime.Now;
            Debug.Log(
                "Game session start @: " + DateTime.Now);
        }

        void OnApplicationQuit() {
            _sessionEndTime = DateTime.Now;

            TimeSpan timeDifference =
                _sessionEndTime.Subtract(_sessionStartTime);

            Debug.Log(
                "Game session ended @: " + DateTime.Now);
            Debug.Log(
                "Game session lasted: " + timeDifference);
        }

        void OnGUI() {
            if (GUILayout.Button("Next Scene")) {
                SceneManager.LoadScene(
                    SceneManager.GetActiveScene().buildIndex + 1);
            }
        }
    }
}
```

Now that we have prepared all our ingredients, it's time to start the testing phase, which we will do next.

Testing the Game Manager

If you wish to test the classes you just wrote in your instance of Unity, then you should go through the following steps:

1. Create a new empty Unity scene called `Init`.
2. In the `Init` scene, add an empty `GameObject` and attach the `GameManager` class to it.
3. Create several empty Unity scenes, as many as you wish.
4. In **Build Settings** under the **File** menu, add the **Init** scene at index **0**:

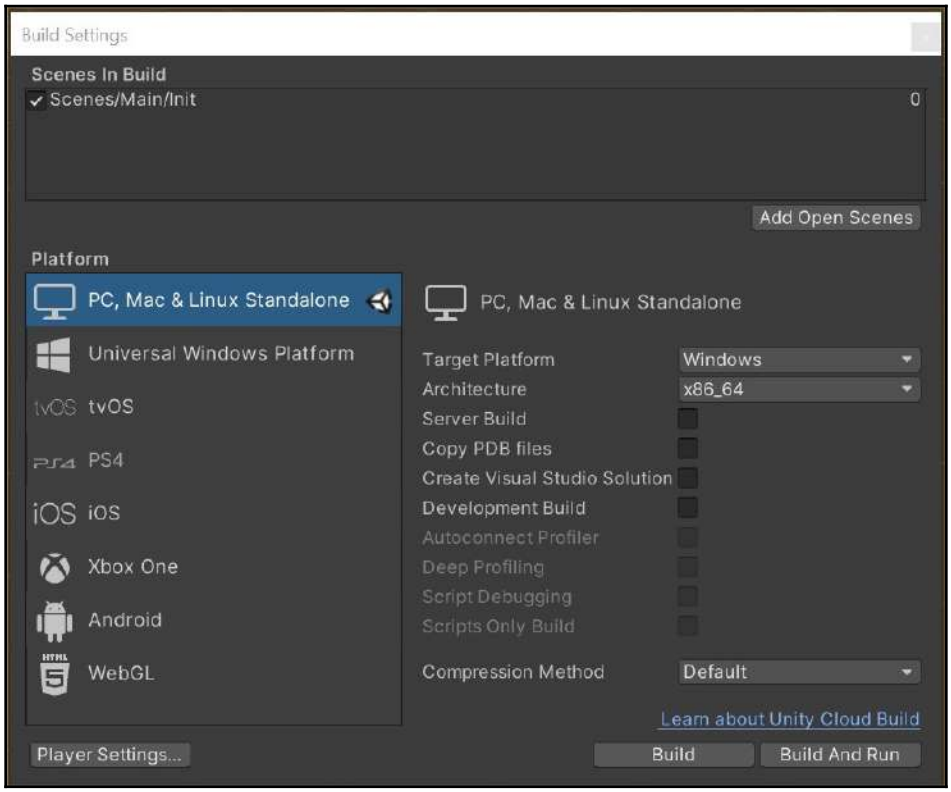


Figure 4.3 – Build Settings

5. Then add your new empty Unity scenes to the **Build Settings** list, as many as you wish.

If you now start the `Init` scene, you should see a GUI button named **Next Scene** as in the following screenshot:

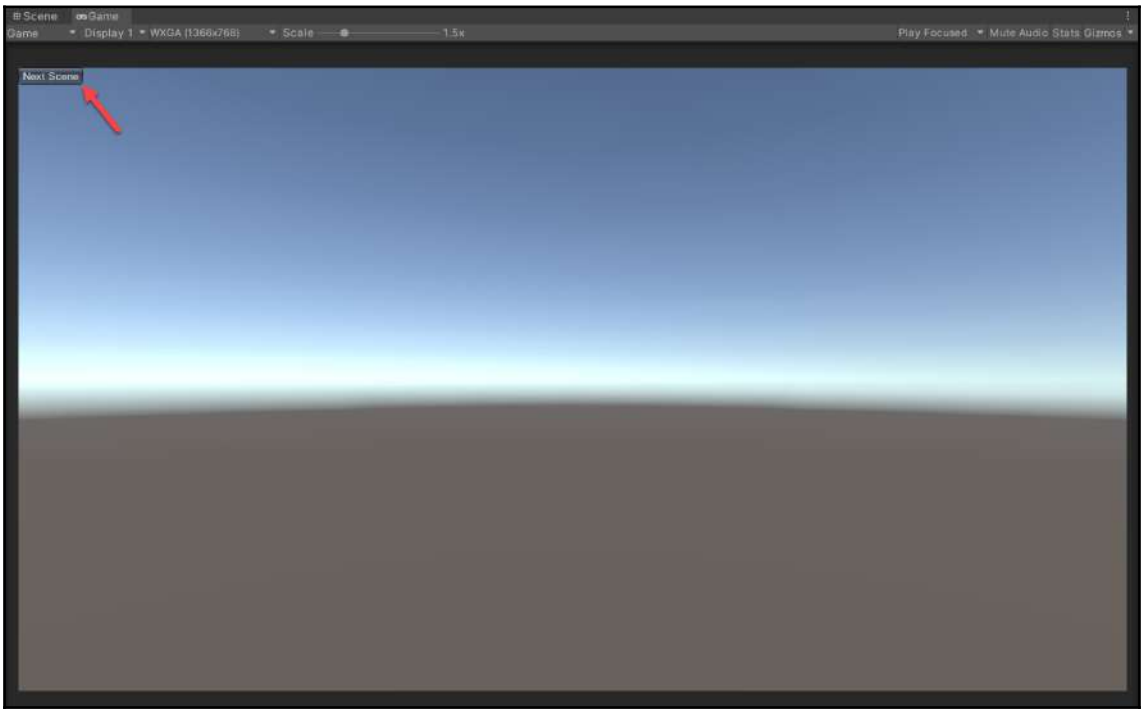


Figure 4.4 – Screenshot of the code example in action

If you click on the **Next Scene** button, you will cycle through each of the scenes you added in **Build Settings**, and the GUI will persist onscreen. If you stop running the game, you should see in the console log the duration of your session. If you try to add additional GameManagers to GameObjects in any scene, you will notice that they get destroyed, as only one can exist during the entire lifespan of the game.

This concludes our tests; we now have the first draft of a GameManager class and a reusable Singleton implementation.

Summary

In this chapter, we tackled one of the most controversial design patterns out there. But we found a way to implement it with a consistent and reusable approach. The Singleton is a pattern that's perfectly suited to Unity's coding model but overusing it can lead you to become too dependent on it.

In the next chapter, we will review the State pattern, which we will use to implement a controller class for the main ingredient of our game, the racing bike.

5
Managing Character States with the State Pattern

In video games, entities continually transition from one state to another depending on player inputs or events. An enemy character might go from an idle state to an attack state depending on whether it spots the player moving across the map. A player character is constantly blending from one animation to another as it responds to the player inputs. In this chapter, we will review a pattern that permits us to define the individual states of an entity and its stateful behaviors.

To start, we will use the traditional State pattern to manage the individual finite states of our main character. In the context of our racing game project, the main character is a motorcycle, so it has a set of mechanical behaviors and animations. As we progress in our implementation of the State pattern, we will soon see its limitations, which we will overcome by introducing FSM (Finite State Machine) concepts.

We will not write an FSM by hand, but instead explore the native state machine implementation in Unity's native animation system. Therefore, this chapter will provide a two-fold approach, an introduction to the core concepts of the State pattern, and how to repurpose Unity's animation system to manage character states and animations.

In this chapter, we will cover the following topics:

- An overview of the State pattern
- Implementation of the State pattern to manage the main character's states

Technical requirements

This chapter is hands-on. You will need to have a basic understanding of Unity and C#.

The code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter05`.

Check out the following video to see the code in action: `https://bit.ly/36EbbHe`

An overview of the State pattern

We use the State design pattern to implement a system that will permit an object to change its behavior based on its internal state. Thus, a change of context will bring on a change of behavior.

The State pattern has three core participants in its structure:

- The `Context` class defines an interface that permits a client to request a change in the internal state of an object. It also holds a pointer to the current state.
- The `IState` interface establishes an implementation contract for the concrete state classes.
- The `ConcreteState` classes implement the `IState` interface and expose a public method named `handle()` that the `Context` object can call to trigger the state's behavior.

Let's now review a diagram of this pattern definition, but in the context of an actual implementation:

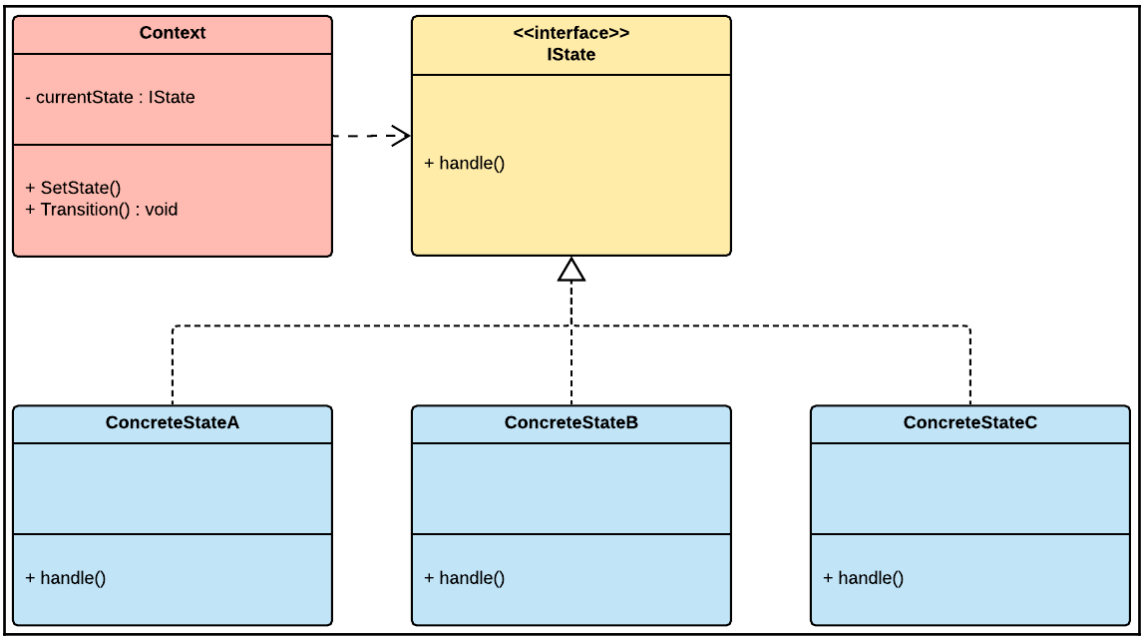


Figure 5.1 – UML diagram of the State pattern

To update an object's state, the client can set the expected state through the `Context` object and request a transition to the new state. Thus, the context is always aware of the current state of the object it handles. However, it doesn't need to be acquainted with each of the concrete state classes that exist. We can add as many state classes as we wish without modifying a single line of code in the `Context` class.

For example, this approach scales better than defining all our states in a single class and managing the transition between them with a switch case, as we can see in the following pseudo-code example:

```
public class BikeController
{
  ....
    switch (state)
    {
        case StopState:
            ...
            break;
        case StarState:
            ...
            break;
        case TurnState:
```

```
            ...
            break;
    }
```

Now that we have a basic understanding of the structure of the State pattern, we can start defining the states and behaviors of our main character, in this case, the bike, as we are going to see in the next section.

Defining character states

In our game, the entity that will transition between states the most is our racing bike. It's under the player's control; it interacts with almost every element in the environment, including obstacles and enemy drones. Therefore, it will never stay in the same state for long.

The following diagram showcases a shortlist of finite states for our bike:

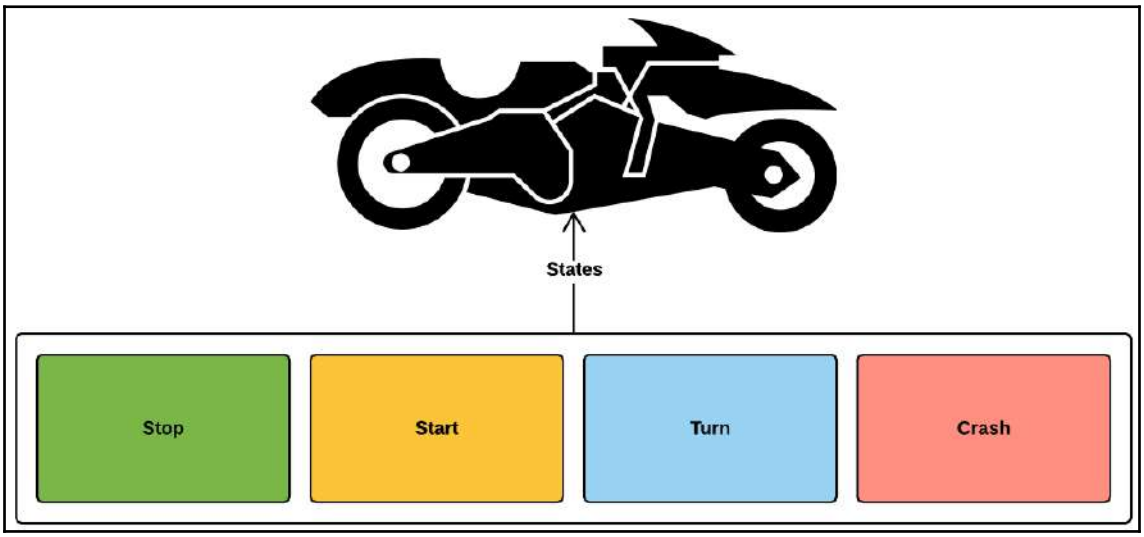


Figure 5.2 – Diagram illustrating the finite states of the bike

And now let's define some expected behaviors for some of the listed states:

- **Stop**: In this state, the bike is not moving. Its current speed is at zero. Its gears are set to neutral. If we decide that the engine should be on, but running in idle mode in that state, we could have an animation of the chassis of the bike vibrating to show that the motor is running.
- **Start**: In this state, the bike moves at full speed, and the wheels turn to match the forward motion.
- **Turn**: In the turning state, the bike turns to the left or to the right, depending on the player's input.
- **Crash**: If the bike is in a crash state, it means it's on fire and on its side. The vehicle will no longer respond to the player's input.

These are just unpolished definitions of potential states for our vehicle. We could go as granular and specific as we wish, but for our use case, it's enough.

It is necessary to keep in mind that each definition of a state contains the description of behavior and animations. These requirements will be essential to consider when we write and review our implementation of the State pattern in the next section of this chapter.

Implementing the State pattern

In this part of the chapter, we will implement the State pattern with the explicit goal of encapsulating the expected behaviors of each of the vehicle's finite states that we defined in the previous section.

We are going to focus on writing minimalist skeleton classes for reasons of brevity and clarity. This approach will permit us to focus on the pattern's structure without being bogged down by implementation details.

It's also important to note that the version of the State pattern presented in this book might be slightly unorthodox as it deviates somewhat from traditional approaches. Therefore, it should be considered a permutation of the State pattern adapted to the context of a Unity project and for a specific use case.

Implementing the State pattern

We are going to review our code example in several steps:

1. Let's start by writing down the main interface that each of our concrete state classes will implement:

```
namespace Chapter.State
{
    public interface IBikeState
    {
        void Handle(BikeController controller);
    }
}
```

   We should note that we are passing an instance of `BikeController` in the `Handle()` method. This approach permits state classes to access public properties of `BikeController`. This approach might deviate slightly from tradition as usually, it's the `Context` object that gets passed to the states.

   However, nothing stops us from passing both the `Context` object and the instance of `BikeController` to the state classes. Alternatively, we could set an instance reference to `BikeController` when we initialize each state class.

   However, it was just simpler to do, as we are going to do for this use case.

2. Now that we have our interface, let's implement the context class:

```
namespace Chapter.State
{
    public class BikeStateContext
    {
        public IBikeState CurrentState
        {
            get; set;
        }

        private readonly BikeController _bikeController;

        public BikeStateContext(BikeController bikeController)
        {
            _bikeController = bikeController;
        }

        public void Transition()
        {
```

```
            CurrentState.Handle(_bikeController);
        }

        public void Transition(IBikeState state)
        {
            CurrentState = state;
            CurrentState.Handle(_bikeController);
        }
    }
}
```

As we can see, the `BikeStateContext` class exposes a public property that points to the current state of the bike; thus, it's aware of any state change. Therefore, we could update the current state of our entity through its property and transition into it by calling the `Transition()` method.

For instance, this mechanism is beneficial if we wanted to link states together by letting each state class declare the next one in the chain. Then, we could cycle through the linked states by simply calling the `Transition()` method of the `Context` object. However, this approach won't be necessary for our use case, as we will call the overloaded `Transition()` method and simply pass the state in which we want to transition.

3. Next up is `BikeController`. This class initializes the `Context` object and the states, and it also triggers state changes:

```
using UnityEngine;

namespace Chapter.State {
    public class BikeController : MonoBehaviour {

        public float maxSpeed = 2.0f;
        public float turnDistance = 2.0f;

        public float CurrentSpeed { get; set; }

        public Direction CurrentTurnDirection {
            get; private set;
        }

        private IBikeState
            _startState, _stopState, _turnState;

        private BikeStateContext _bikeStateContext;

        private void Start() {
```

```
            _bikeStateContext =
                new BikeStateContext(this);

            _startState =
                gameObject.AddComponent<BikeStartState>();
            _stopState =
                gameObject.AddComponent<BikeStopState>();
            _turnState =
                gameObject.AddComponent<BikeTurnState>();

            _bikeStateContext.Transition(_stopState);
        }

        public void StartBike() {
            _bikeStateContext.Transition(_startState);
        }

        public void StopBike() {
            _bikeStateContext.Transition(_stopState);
        }

        public void Turn(Direction direction) {
            CurrentTurnDirection = direction;
            _bikeStateContext.Transition(_turnState);
        }
    }
}
```

If we didn't encapsulate the behaviors of the bike inside individual state classes, we would probably implement them inside `BikeController`. This approach would have likely resulted in a bloated controller class that's difficult to maintain. Therefore, by using the State pattern, we are making our classes smaller and easier to maintain.

And we are also giving back to `BikeController` its original responsibilities of controlling the bike's core components. It exists to offer an interface to control the bike, expose its configurable properties, and manage its structural dependencies.

4. The following three classes are going to be our states; they are quite self-explanatory. Notice that each implements the `IBikeState` interface. Let's start with `BikeStopState`:

```
using UnityEngine;

namespace Chapter.State
{
    public class BikeStopState : MonoBehaviour, IBikeState
```

```
    {
        private BikeController _bikeController;

        public void Handle(BikeController bikeController)
        {
            if (!_bikeController)
                _bikeController = bikeController;

            _bikeController.CurrentSpeed = 0;
        }
    }
}
```

5. The next state class is `BikeStartState`:

```
using UnityEngine;

namespace Chapter.State
{
    public class BikeStartState : MonoBehaviour, IBikeState
    {
        private BikeController _bikeController;

        public void Handle(BikeController bikeController)
        {
            if (!_bikeController)
                _bikeController = bikeController;

            _bikeController.CurrentSpeed =
                _bikeController.maxSpeed;
        }

        void Update()
        {
            if (_bikeController)
            {
                if (_bikeController.CurrentSpeed > 0)
                {
                    _bikeController.transform.Translate(
                        Vector3.forward * (
                            _bikeController.CurrentSpeed *
                            Time.deltaTime));
                }
            }
        }
    }
}
```

6. And lastly, there is `BikeTurnState`, which makes the bike turn left or right:

```
using UnityEngine;

namespace Chapter.State
{
    public class BikeTurnState : MonoBehaviour, IBikeState
    {
        private Vector3 _turnDirection;
        private BikeController _bikeController;

        public void Handle(BikeController bikeController)
        {
            if (!_bikeController)
                _bikeController = bikeController;

            _turnDirection.x =
                (float) _bikeController.CurrentTurnDirection;

            if (_bikeController.CurrentSpeed > 0)
            {
                transform.Translate(_turnDirection *
                                    _bikeController.turnDistance);
            }
        }
    }
}
```

7. For our final class, `BikeController` references an enum named `Direction`, which we will implement here:

```
namespace Chapter.State
{
 public enum Direction
 {
 Left = -1,
 Right = 1
 }
}
```

We now have all our ingredients ready to test our State pattern implementation.

Testing the State pattern implementation

To quickly test our implementation of the State pattern in your own instance of Unity, you need to follow these steps:

1. Copy all the scripts we just reviewed inside your Unity project.
2. Create a new empty scene.
3. Add a 3D GameObject to the scene, such as a cube, ensuring that it's visible to the main camera.
4. Attach the `BikeController` script to the GameObject.
5. Also attach the following client script to the GameObject:

```
using UnityEngine;

namespace Chapter.State
{
    public class ClientState : MonoBehaviour
    {
        private BikeController _bikeController;

        void Start()
        {
            _bikeController =
                (BikeController)
                FindObjectOfType(typeof(BikeController));
        }

        void OnGUI()
        {
            if (GUILayout.Button("Start Bike"))
                _bikeController.StartBike();
            if (GUILayout.Button("Turn Left"))
                _bikeController.Turn(Direction.Left);
            if (GUILayout.Button("Turn Right"))
                _bikeController.Turn(Direction.Right);
            if (GUILayout.Button("Stop Bike"))
                _bikeController.StopBike();
        }
    }
}
```

Once you start the scene, you should see the following GUI buttons on your screen, which you can use to control the GameObject by triggering state changes:

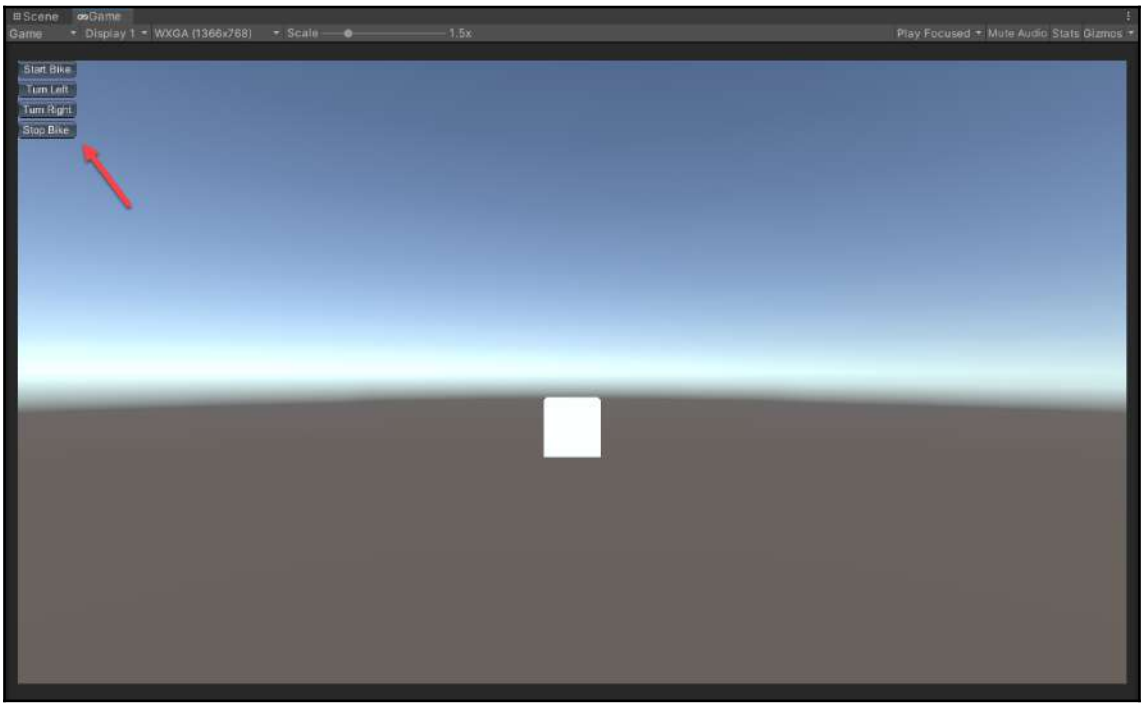


Figure 5.3 – Screenshot of the code example in action

In the next section of the book, we are going to review the benefits of the State pattern, but also its limitations.

Benefits and drawbacks of the State pattern

The following are the benefits of using the State pattern:

- **Encapsulation**: The State pattern allows us to implement an entity's stateful behaviors as a collection of components that can be assigned dynamically to an object when it changes states.
- **Maintenance**: We can easily implement new states without having to modify long conditional statements or bloated classes.

However, the State pattern does have its limitations when we use it to manage an animated character.

Here's a shortlist of potential limitations:

- **Blending**: In its native form, the State pattern doesn't offer a solution to blend animations. This limitation can become an issue when you want to achieve a smooth visual transition between the animated states of a character.
- **Transitioning**: In our implementation of the pattern, we can easily switch between states, but we are not defining the relation between them. Therefore, if we wish to define transitions between states based on relationships and conditions, we will have to write a lot more code; for instance, if I want the idle state to transition to the walk state, and then the walk state to transition to the run state. And this happens automatically and smoothly, back and forth, depending on a trigger or condition. This could be time-consuming to do in code.

However, the limitations listed above can be overcome by using the Unity animation system and its native state machine. We can easily define animation states and attach animation clips and scripts to each of the configured states. But its more important feature is that it lets us define and configure a set of transitions between states with conditions and triggers through a visual editor, as we can see here:

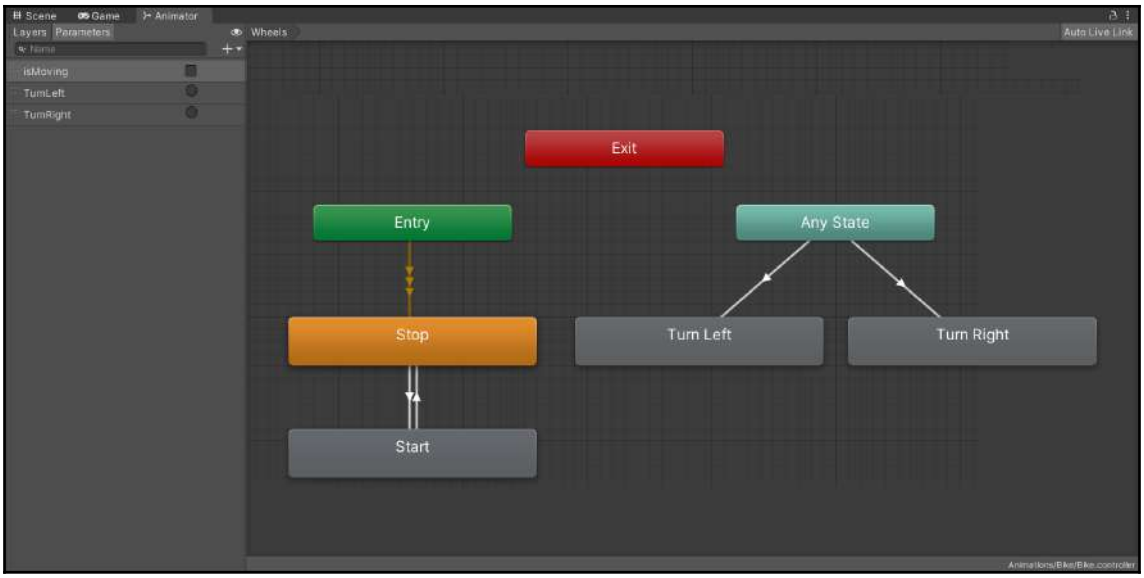


Figure 5.4 – Screenshot of the Unity animation system editor

The rectangles present represent individual animation states, and the arrows indicate relationships and transitions. A deep dive into how to use Unity animation is beyond the scope of this book. The goal of this chapter was to get introduced to the State pattern in the context of the Unity engine. As we see, Unity offers us a native solution that we can leverage to manage the animation states of our characters.

It's essential to keep in mind that this tool is not limited to animating humanoid characters. We could use it for mechanical entities such as cars or even background ingredients, such as a vending machine; hence, anything that has states, animations, and behaviors.

In the next section, we will review a shortlist of alternative patterns to consider before using the State pattern.

For more information on Unity's animation system, you can read the official documentation at the following link: `https://docs.unity3d.com/Manual/AnimationSection.html`.

Reviewing alternative solutions

The following is a list of patterns that are related or alternatives to the State pattern:

- **Blackboard/Behavior Trees**: If you are planning to implement complex AI behaviors for NPC characters, I would recommend considering patterns such as the Blackboard or concepts such as **Behavior Trees** (**BT**). For example, if you need to implement AI with dynamic decision-making behaviors, then BT is a more appropriate approach because it permits you to implement behavior using a tree of actions.
- **FSM**: A question that often arises when discussing the State pattern is the core difference between an FSM and the State pattern. The quick answer is that the State pattern is concerned with encapsulating an object's state-dependent behaviors. However, FSM is more deeply involved with transitioning between finite states based on specific input triggers. And so, FSM is often considered more suited for the implementation of automaton-like systems.
- **Memento**: Memento is similar to the State pattern, but with an extra feature that gives objects the ability to roll back to a previous state. This pattern could be useful in implementing a system that needs the ability to undo a change made to itself.

Summary

In this chapter, we were able to leverage the State pattern to define and implement stateful behaviors of our main character. In our case, the character is a vehicle. After reviewing its structure in the context of a specific code example, we saw its limits when dealing with animated entities. However, Unity offers us a native solution that permits us to manage the states of animated characters with a sophisticated state machine and a visual editor.

However, it doesn't mean that the State pattern in itself is useless in Unity. We could easily use it as a foundation to build stateful systems or mechanics.

In the next chapter, we will define the global states of our racing game and manage them with Event Bus.

6
Managing Game Events with the Event Bus

The Event Bus acts as a central hub that manages a specific list of global events that objects can choose to subscribe to or publish. It's the most straightforward pattern related to event managing that I have in my toolbox. It reduces the process of assigning the role of subscriber or publisher to an object into a single line of code. As you can imagine, this can be beneficial when you need results quickly. But like most simple solutions, it has some drawbacks and limitations, which we will explore further on.

In the code example presented in this chapter, we will use the Event Bus to broadcast specific race events to components that need to listen for changes in the overall state of the race. But it's essential to keep in mind; I'm proposing using the Event Bus as a solution for managing global race events because of its simplicity and not its scalability. So it might not be the best solution in all cases, but it is one of the most straightforward patterns to implement, as we will see in the following sections.

The following topics will be covered in this chapter:

- An overview of the Event Bus pattern
- The implementation of a Race Event Bus

This chapter includes simplified skeleton code examples for the sake of brevity and clarity. If you wish to review a complete implementation of the pattern in the context of an actual game project, open the `FPP` folder in the GitHub project. You can find the link under the *Technical requirements* section.

Technical requirements

We will also be using the following specific Unity engine API features:

- `Static`
- `UnityEvents`
- `UnityActions`

If you are unfamiliar with these concepts, please review `Chapter 3`, *A Short Primer to Programming in Unity*.

The code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter06`.

Check out the following video to see the code in action:
`https://bit.ly/2U7wrCM`.

If you find yourself having problems understanding the mechanism behind delegates, keep in mind that they are similar to function pointers in C/C++. In simple terms, a delegate points to a method. And delegates are often used to implement event handlers and callbacks. An action is a type of delegate that you can use with a method that has a void return type.

Understanding the Event Bus pattern

When an event is raised by an object (publisher), it sends out a signal that other objects (subscribers) can receive. The signal is in the form of a notification that can indicate the occurrence of an action. In an event system, an object broadcasts an event. Only those objects that subscribe to it will be notified and choose how to handle it. So, we can imagine it as having a sudden burst of a radio signal that only those with antennas tuned to a specific frequency can detect.

The Event Bus pattern is a close cousin of the Messaging system and Publish-Subscribe patterns, the latter being the more accurate name for what the Event Bus does. The keyword in the title of this pattern is the term *bus*. In simple computing terms, a bus is a connection between components. In the context of this chapter, the components will be objects that can be publishers or listeners of events.

Hence, the Event Bus is a way to connect objects through events by using a publish-subscribe model. It's possible to accomplish a similar model with a pattern such as the Observer and native C# events. However, those alternatives have some drawbacks. For example, in a typical implementation of the Observer pattern, a tight coupling might occur, as observers (listeners) and subjects (publishers) might become dependent and aware of each other.

But the Event Bus, at least in the way we will implement it in Unity, abstracts and simplifies the relations between publishers and subscribers, so they are entirely unaware of one another. Another advantage is that it reduces the process of assigning the role of publisher or subscriber to a single line of code. Therefore, the Event Bus is a valuable pattern to learn and have in your back pocket. As you can see in the following diagram, it does act as a middleman between publishers and subscribers:

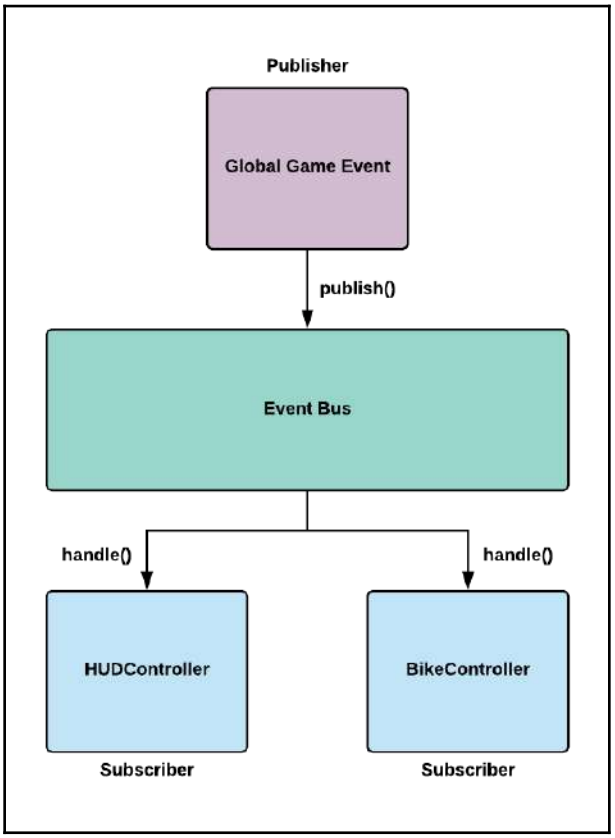


Figure 6.1 – Diagram of the Event Bus pattern

As you can see from the diagram, there are three main ingredients:

- **Publishers**: These objects can publish specific types of events declared by the Event Bus to subscribers.
- **Event Bus**: This object is responsible for coordinating the transmission of events between publishers and subscribers.
- **Subscribers**: These objects register themselves as subscribers of specific events through the Event Bus.

Benefits and drawbacks of the Event Bus pattern

The **benefits** of the Event Bus pattern are as follows:

- **Decoupling**: The main benefit of using an event system is that it decouples your objects. Objects can communicate through events instead of directly referencing each other.
- **Simplicity**: The Event Bus offers simplicity by abstracting the mechanism of publishing or subscribing to an event from its clients.

The **drawbacks** of the Event Bus pattern are as follows:

- **Performance**: Under the hood of any event system, there's a low-level mechanism that manages messaging between objects. And so there might be a slight performance cost when using an event system, but depending on your target platform, it could be minuscule.
- **Global**: In this chapter, we implement the Event Bus with static methods and properties to make it easier to access from anywhere in our code. There is always a risk when using globally accessible variables and states as they can make debugging and unit testing more difficult. However, this is a very contextual drawback and not an absolute.

UnityEvents can actually accept up to four generic-type arguments. This can make it possible to pass event-specific data to subscribers. Review the following Unity API documentation section for more details: `https://docs.unity3d.com/2021.2/Documentation/ScriptReference/Events.UnityEvent.html`.

When to use the Event Bus

I have used the Event Bus in the past for the following:

- **Rapid prototyping**: I use the Event Bus pattern often when rapidly prototyping new game mechanics or features. With this pattern, I can easily have components that trigger each other's behaviors with events while keeping them decoupled. This pattern permits us to add and remove objects as subscribers or publishers with a single line of code, which is always helpful when you want to prototype something quickly and easily.
- **Production code**: I use the Event Bus in production code if I can't find a justifiable reason to implement a more sophisticated approach to managing game events. It's a pattern that does the job well if you don't need to handle complex event types or structures.

I would avoid using a globally accessible Event Bus like the one presented in this chapter to manage events that don't have a "global scope." For instance, if I have a UI component in the HUD that displays a damage alert when the player collides with an obstacle, it would be inefficient to publish a collision event through the Event Bus as it's a localized interaction between the bike and an object on the track. Instead, I would use something like an Observer pattern because, in that scenario, I only need one UI component to observe a specific change in the state of an object, in this case, the bike. As a rule of thumb, every time you have to implement a system that uses events, go through the list of known patterns, choose one suitable for your use case, but don't always fall back to the easiest one, which, in my opinion, is the Event Bus.

Managing global race events

The project we are working on is a racing game, and most races are structured in stages. The following is a shortlist of typical racing stages:

- **Countdown**: At this stage, the bike is stopped behind the start line while a countdown timer is running down.
- **Race start**: Once the clock hits zero, the green light signal is turned on, and the bike moves forward on the track.
- **Race finish**: The moment the player crosses the finish line, the race is over.

In between the start and finish of the race, certain events can be triggered that could change the current state of the race:

- **Race pause**: The player could pause the game while still racing.
- **Race quit**: The player could quit the race at any time.
- **Race stop**: The race could stop suddenly if the player is involved in a fatal crash.

So we want to broadcast a notification that signals the occurrence of each stage of the race and any other important event in between that will change the overall state of the race. This approach will permit us to trigger specific behaviors of components that need to behave in certain ways depending on the current context of the race.

For example, here are some components that will get updated depending on a specific state of the race:

- **HUD**: The race status indicator will change depending on the context of the race **HUD** (**heads-up display**).
- **RaceTimer**: The race timer will only start at the start of the race and stops when the player crosses the finish line.
- **BikeController**: The bike controller releases the brakes of the bike once the race starts. This mechanism prevents the player from launching onto the track before the green light.
- **InputRecorder**: At the start of the race, the input replay system will start recording the player's inputs so that they can be replayed later on.

All these components have specific behaviors that have to be triggered at particular stages of the race. So we will use the Event Bus to implement these global race events.

Implementing a Race Event Bus

We are going to implement the Race Event Bus in two easy steps:

1. To start, we need to expose the specific race event types that we support, which we will do with the following enum:

```
namespace Chapter.EventBus
{
    public enum RaceEventType
    {
        COUNTDOWN, START, RESTART, PAUSE, STOP, FINISH, QUIT
    }
}
```

It's important to note that the preceding enum values represent specific events outlining the stages of a race, from start to finish. So we are restricting ourselves to handling events with just a global scope.

2. The next part is the core component of the pattern, the actual game event bus class, which we will call `RaceEventBus`, to be more domain-specific in the naming convention of our classes:

```
using UnityEngine.Events;
using System.Collections.Generic;

namespace Chapter.EventBus
{
    public class RaceEventBus
    {
        private static readonly
            IDictionary<RaceEventType, UnityEvent>
            Events = new Dictionary<RaceEventType, UnityEvent>();

        public static void Subscribe
            (RaceEventType eventType, UnityAction listener) {
            UnityEvent thisEvent;

            if (Events.TryGetValue(eventType, out thisEvent)) {
                thisEvent.AddListener(listener);
            }
            else {
                thisEvent = new UnityEvent();
                thisEvent.AddListener(listener);
                Events.Add(eventType, thisEvent);
            }
        }

        public static void Unsubscribe
            (RaceEventType type, UnityAction listener) {

            UnityEvent thisEvent;

            if (Events.TryGetValue(type, out thisEvent)) {
                thisEvent.RemoveListener(listener);
            }
        }

        public static void Publish(RaceEventType type) {

            UnityEvent thisEvent;
```

```
                    if (Events.TryGetValue(type, out thisEvent)) {
                        thisEvent.Invoke();
                    }
                }
            }
        }
```

The key ingredient of our class is the `Events` dictionary. This acts as a ledger in which we maintain a list of relations between event types and subscribers. By keeping it `private` and `readonly`, we are ensuring that it can't be overwritten by another object directly.

Therefore, a client must call the `Subscribe()` public static method to add itself as a subscriber of a specific event type. The `Subscribe()` method takes two parameters; the first is the race event type, and the second one is the callback function. Because `UnityAction` is a delegate type, it provides us with a way to pass a method as an argument.

Hence, when a client object calls the `Publish()` method, the registered callback methods of all the subscribers of a specific race event type will get called at the same time.

The `Unsubscribe()` method is self-explanatory as it permits objects to remove themselves as subscribers of a specific event type. Thus, their callback methods will not be called by the Event Bus when an object publishes an event.

If this still looks abstract, it will be made clear as we implement client classes in the next section and see how we can use the Event Bus to trigger behaviors of objects at specific moments in the proper sequence.

Testing the Race Event Bus

Now that we have the core elements of the pattern in place, we can write some code to test our `RaceEventBus` class. For reasons of brevity, I have removed all the behavior code in each client class to focus on the use of the pattern:

1. For starters, we are going to write a countdown timer that subscribes to the `COUNTDOWN` race event type. Once the `COUNTDOWN` event is published, it will trigger a 3-second countdown to the start of the race. And at the exact moment the count reaches its end, it will publish the `START` event to signal the beginning of the race:

```
using UnityEngine;
using System.Collections;
```

```
namespace Chapter.EventBus
{
    public class CountdownTimer : MonoBehaviour
    {
        private float _currentTime;
        private float duration = 3.0f;

        void OnEnable() {
            RaceEventBus.Subscribe(
                RaceEventType.COUNTDOWN, StartTimer);
        }

        void OnDisable() {
            RaceEventBus.Unsubscribe(
                RaceEventType.COUNTDOWN, StartTimer);
        }

        private void StartTimer() {
            StartCoroutine(Countdown());
        }

        private IEnumerator Countdown() {
            _currentTime = duration;

            while (_currentTime > 0) {
                yield return new WaitForSeconds(1f);
                _currentTime--;
            }

            RaceEventBus.Publish(RaceEventType.START);
        }

        void OnGUI() {
            GUI.color = Color.blue;
            GUI.Label(
                new Rect(125, 0, 100, 20),
                "COUNTDOWN: " + _currentTime);
        }
    }
}
```

The most significant lines of code in the preceding class are the following:

```
void OnEnable() {
    RaceEventBus.Subscribe(
        RaceEventType.COUNTDOWN, StartTimer);
}

void OnDisable() {
    RaceEventBus.Unsubscribe(
        RaceEventType.COUNTDOWN, StartTimer);
}
```

Every time the `CountdownTimer` object is enabled, the `Subscribe()` method is called. And when the opposite happens, and it gets disabled, it unsubscribes itself. We are doing this to ensure that the object is listening to an event when it's active or doesn't get called by `RaceEventBus` when disabled or destroyed.

The `Subscribe()` method takes two arguments – the event type, and a callback function. This means that the `StartTimer()` method of `CountdownTimer` will be called by `RaceEventBus` every time the `COUNTDOWN` event is published.

2. Next up, we will implement a skeleton of the `BikeController` class to test out the `START` and `STOP` events:

```
using UnityEngine;

namespace Chapter.EventBus
{
    public class BikeController : MonoBehaviour
    {
        private string _status;

        void OnEnable() {
             RaceEventBus.Subscribe(
                 RaceEventType.START, StartBike);

             RaceEventBus.Subscribe(
                 RaceEventType.STOP, StopBike);
        }

        void OnDisable() {
             RaceEventBus.Unsubscribe(
                 RaceEventType.START, StartBike);

             RaceEventBus.Unsubscribe(
                 RaceEventType.STOP, StopBike);
        }
```

```
        private void StartBike() {
             _status = "Started";
        }

        private void StopBike() {
             _status = "Stopped";
        }

        void OnGUI() {
             GUI.color = Color.green;
             GUI.Label(
                 new Rect(10, 60, 200, 20),
                 "BIKE STATUS: " + _status);
         }
     }
}
```

3. And lastly, we are going to write an `HUDController` class. This doesn't do much except display a button to stop the race once it starts:

```
using UnityEngine;

namespace Chapter.EventBus
{
    public class HUDController : MonoBehaviour
    {
        private bool _isDisplayOn;

        void OnEnable() {
            RaceEventBus.Subscribe(
                RaceEventType.START, DisplayHUD);
        }

        void OnDisable() {
            RaceEventBus.Unsubscribe(
                RaceEventType.START, DisplayHUD);
        }

        private void DisplayHUD() {
            _isDisplayOn = true;
        }

        void OnGUI() {
            if (_isDisplayOn)
            {
                if (GUILayout.Button("Stop Race"))
                {
                    _isDisplayOn = false;
```

```
                        RaceEventBus.Publish(RaceEventType.STOP);
                    }
                }
            }
        }
    }
```

4. To test the sequence of events, we need to attach the following client class to a GameObject in an empty Unity scene. With this, we will be able to trigger the countdown timer:

```
using UnityEngine;

namespace Chapter.EventBus
{
    public class ClientEventBus : MonoBehaviour
    {
        private bool _isButtonEnabled;

        void Start()
        {
            gameObject.AddComponent<HUDController>();
            gameObject.AddComponent<CountdownTimer>();
            gameObject.AddComponent<BikeController>();

            _isButtonEnabled = true;
        }

        void OnEnable()
        {
            RaceEventBus.Subscribe(
                RaceEventType.STOP, Restart);
        }

        void OnDisable()
        {
            RaceEventBus.Unsubscribe(
                RaceEventType.STOP, Restart);
        }

        private void Restart()
        {
            _isButtonEnabled = true;
        }

        void OnGUI()
        {
            if (_isButtonEnabled)
```

```
                    {
                        if (GUILayout.Button("Start Countdown"))
                        {
                            _isButtonEnabled = false;
                            RaceEventBus.Publish(RaceEventType.COUNTDOWN);
                        }
                    }
                }
            }
        }
```

The preceding example might seem overly simplified, but its purpose is to showcase how we can trigger individual object behaviors in a specific sequence with events while keeping them decoupled. None of the objects communicate with one another directly.

Their only common point of reference is `RaceEventBus`. Therefore, we could easily add more subscribers and publishers; for instance, we could have a `TrackController` object listen for the `RESTART` event to know when to reset the race track. And thus, we can now sequence the triggering of behaviors of core components with events, each presenting a particular stage of the race.

Action is a delegate that points to a method that accepts one or more arguments but returns no value. For instance, you should use Action when your delegate points to a method that returns void. UnityAction behaves like Actions in native C#, except UnityActions was designed to be used with UnityEvents.

Reviewing the Event Bus implementation

By using the Event Bus, we can trigger behaviors while keeping core components decoupled. It's straightforward for us to add or remove objects as subscribers or publishers. We also defined a specific list of global events that represent every stage of a race. Therefore, we can now start sequencing and triggering behaviors of core components, from the start to the end of a race, and anything in between.

In the next section, we will review some alternative solutions to the Event Bus, with each solution offering a different approach that might be a better solution depending on context.

Reviewing some alternative solutions

Event systems and patterns are a vast topic, and it's a subject matter we can't cover in depth in this book. Therefore, we have prepared a shortlist of patterns to consider when implementing an event system or mechanism, but keep in mind that there's a lot more out there, and we encourage you as a reader to continue exploring the topic beyond the limited scope of this book:

- **Observer**: An oldie but goodie pattern in which an object (subject) maintains a list of objects (observers) and notifies them of an internal state change. It's a pattern to consider when you need to establish a one-to-many relationship between a group of entities.
- **Event Queue**: This pattern permits us to store events generated by publishers in a queue and forward them to their subscribers at a convenient time. This approach decouples the temporal relationship between publishers and subscribers.
- **ScriptableObjects**: It's possible to create an event system with ScriptableObjects in Unity. The key benefit of this approach is that it makes it easier to author new custom game events. If you need to build a scalable and customizable event system, this might be the way to go.

If you are asking yourself why we are not showcasing more advanced event systems in this book, including those implemented with ScriptableObjects, the answer is that the core intent of this book is to introduce readers to design patterns, not overwhelm them with complexity. We offer a first-step introduction to core concepts, but encourage readers to seek more advanced material as they progress.

Summary

In this chapter, we reviewed the Event Bus, a simple pattern that simplifies the process of publishing and subscribing to events in Unity. It's a tool that helps during rapid prototyping or when you have a defined list of global events to manage. However, it has its limits of use, and it's always wise to explore other options before committing to using a globally accessible event bus.

In the next chapter, we will implement a system that will allow us to replay player inputs. Many racing games have replay and rewind features, and with the Command pattern, we will attempt to build one from scratch.

7
Implement a Replay System with the Command Pattern

In this chapter, we will use a classic design pattern named Command to implement a replay system for our racing game. Its mechanism will record the player's controller inputs and the corresponding timestamp. This approach will permit us to play back the recorded inputs in the proper order and with the correct timing.

Replay features are often a must-have in racing games. With the Command pattern, we are going to implement this system in a scalable and modular manner. This pattern proposes a mechanism that permits encapsulating information needed to perform an "action" or trigger a state change. It also decouples the requester of an "action" from the object that will perform it. This encapsulation and decoupling permit us to queue action requests so we can execute them at a later time.

All of this might sound very abstract, but once we implement the system's core components, it will be made clear.

The following topics will be covered in this chapter:

- Understanding the Command pattern
- Designing a replay system
- Implementing a replay system
- Reviewing alternative solutions

Technical requirements

The following chapter is hands-on, so you will need to have a basic understanding of Unity and C#.

The code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter07`.

Check out the following video to see the code in action: `https://bit.ly/3wAWYpb`.

Understanding the Command pattern

Imagine a platform game in which you can jump over obstacles when you press the space bar. In this scenario, every time you press that input key, you are asking the character on the screen to change states and perform a jump action. In code, we could implement a simple `InputHandler`, which would listen for a space bar input from the player, and when the player hits it, we would call `CharacterController` to trigger the jump action.

The following very simplified pseudo-code sums up what we have in mind:

```
using UnityEngine;
using System.Collections;

public class InputHandler : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown("space"))
        {
            CharacterController.Jump();
        }
    }
}
```

As we can see, this approach can get the job done, but if we wanted to record, undo, or replay an input from the player at a later time, it could become complicated. However, the Command pattern permits us to decouple the object that invokes the operation from the one that knows how to execute it. In other words, our `InputHandler` doesn't need to know what exact action needs to be taken when the player presses the space bar. It just needs to make sure that the correct *command* is executed and let the Command pattern mechanism do its magic behind the scenes.

The following pseudo-code shows the difference in the way we implement `InputHandler` when we are using the Command pattern:

```
using UnityEngine;
using System.Collections;

public class InputHandler : MonoBehaviour
{
    [SerializedField]
    private Controller _characterController;

    private Command _spaceButton;

    void Start()
    {
        _spaceButton = new JumpCommand();
    }

    void Update()
    {
        if (Input.GetKeyDown("space"))
            _spaceButton.Execute(_characterController);
    }
}
```

As we can see, we are not just invoking `CharacterController` directly when the player presses the space bar. We are actually encapsulating all the information we need to perform the jump action into an object that we could put in a queue and re-invoke at a later time.

Let's review the following diagram, which illustrates an implementation of the Command pattern:

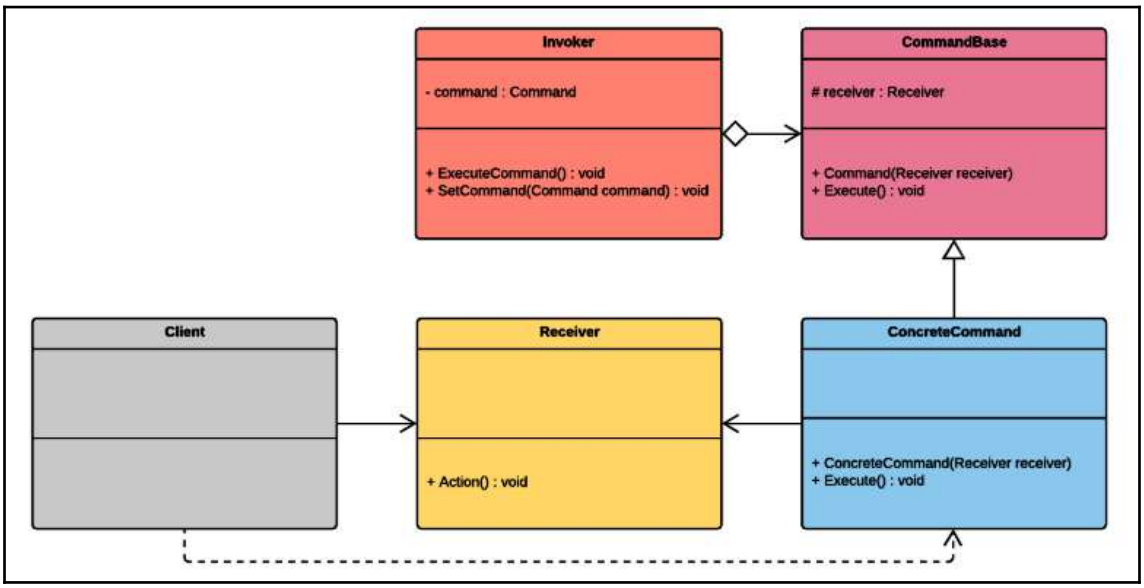


Figure 7.1 – UML diagram of the Command pattern

Trying to learn about the Command pattern by looking at a diagram is not the right approach, but it does help us isolate the fundamental classes that are participating in this pattern:

- `Invoker` is an object that knows how to execute a command and can also do the bookkeeping of executed commands.
- `Receiver` is a type of object that can receive commands and execute them.
- `CommandBase` is an abstract class that an individual `ConcreteCommand` class must inherit, and it exposes an `Execute()` method that `Invoker` can call to execute a specific command.

Each participant in the pattern has a distinct responsibility and a role to play. A solid implementation of the Command pattern should allow us to encapsulate action requests as an object that can be queued and executed immediately or at a later time.

The Command pattern is a part of the Behavioral pattern family; its close cousins are Memento, Observer, and Visitor. These types of patterns are often concerned with the assignment of responsibilities and how objects communicate with one another.

Benefits and drawbacks of the Command pattern

These are some of the benefits of the Command pattern:

- **Decoupling**: The pattern permits the decoupling of the object that invokes the operation from the one that knows how to execute it. This layer of separation allows the addition of an intermediary that will be able to bookkeep and sequence operations.
- **Sequencing**: The Command pattern facilitates the process of queuing user inputs, which permits the implementation of undo/redo features, macros, and command queues.

This is a potential drawback of the Command pattern:

- **Complexity**: It takes numerous classes to implement this pattern as each command is a class in itself. And it takes a good understanding of the pattern to maintain code built with it. In most cases, this is not an issue, but if you are using the Command pattern without a specific goal in mind, it can become an unnecessary layer of complexity and verbosity in your code base.

Benefits and drawbacks are usually contextual; the ones presented in this book are general, and not absolutes. Therefore, it's essential when choosing a pattern to analyze its benefits and drawbacks in the context of your own project, and to not consider or reject one based on general statements.

When to use the Command pattern

Here's a shortlist of possible uses for the Command pattern:

- **Undo**: Implementing an undo/redo system that you find in most text and image editors.
- **Macro**: A macro recording system with which players can record a sequence of attack or defensive combos. Then, assign them on an input key to execute them automatically.

- **Automation:** Automate processes or behaviors by recording a set of commands that a bot will automatically and sequentially execute.

In conclusion, it's a good pattern for features related to storing, timing, and sequencing user inputs. And if you get very creative with it, you could create some compelling game systems and mechanics.

Design patterns are fun to use if you don't worry too much about staying true to the original academic descriptions. If you don't lose the original intent of the pattern while experimenting with it, you should retain its core benefits.

Designing a replay system

Before describing the replay system design that we will be implementing in this chapter, we have to declare some specifications about our game that can influence the way in which we will implement it.

Specifications to keep in mind are as follows:

- **Deterministic**: Everything in our game is deterministic, which means we don't have any entities with random behaviors, and that makes our replay system simpler to implement because we don't have to worry about recording positions or states of entities that move in our scene like enemy drones. We know they will move and behave the same way during the replay sequence.
- **Physics**: We are minimizing the use of the physics features of the Unity engine as our entities' movements are not determined by any physical properties or interactions. Therefore, we do not have to worry about unexpected behaviors when objects collide.
- **Digital**: All our inputs are digital, so we do not bother to capture or handle granular analog input data from a joystick or trigger button.
- **Precision**: We tolerate a lack of precision in the timing of when we replay inputs. Therefore, we do not expect the inputs to replay precisely within the same timeframe as they were recorded. *This tolerance level is subject to change depending on several factors related to the desired level of precision of the replay feature.*

Considering all these specifications, the replay system we are going to implement will only record the player's inputs but not the bike's current position. Because there are no in-between locations that the bike might be at, depending on the player's input, it will be on one rail or the other. Also, another essential detail to note is the fact that the bike does not move forward. It is the track that moves toward the player's position to give the illusion of movement and speed.

In theory, if we record the player's controller inputs from the start and end of a race, then we could simulate a replay of the player's gameplay by replaying the recorded inputs at the beginning of a new race. We have a diagram that illustrates the mechanism behind the replay system:

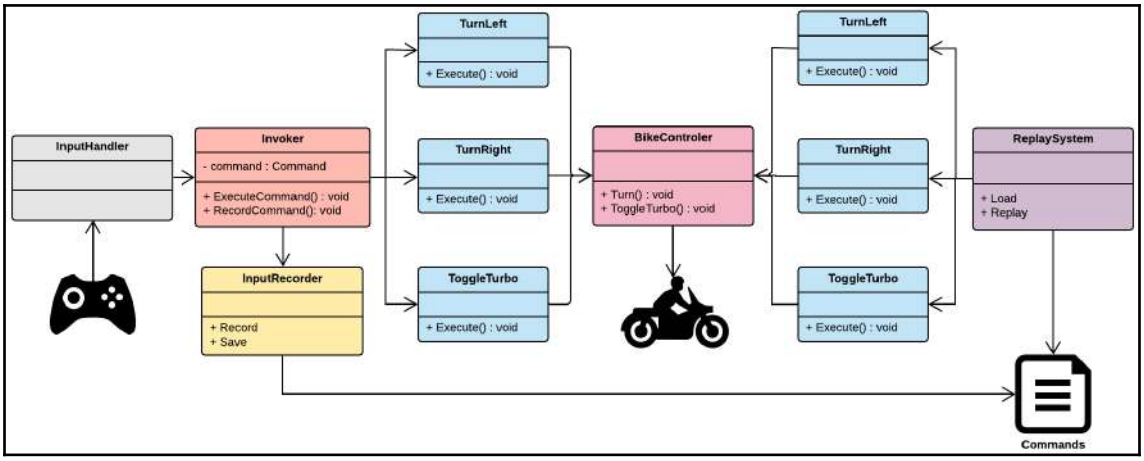


Figure 7.2 – Diagram of the replay system

As we can see in the diagram, `InputRecorder` records and serializes the inputs so that `ReplaySystem` can replay them later on. During a replay sequence, `ReplaySystem` acts similarly to a bot as it takes control of the bike and automatically maneuvers it by replaying the player's inputs. It is a simple form of automation that gives the illusion that we are watching a replay video.

In the next section, we will implement a simplified version of the system we just reviewed, and we will use the Command pattern as its foundation.

Implementing a replay system

This section includes pseudo-code for the sake of simplicity and readability. If you wish to review a complete implementation in the context of an actual game project, open the `FPP` folder in the GitHub project. The link can be found under the *Technical requirements* section.

In this section, we are going to build up a simple input replay system prototype using the Command pattern as the foundation.

Implementing the replay system

The implementation will be done in two parts. In the first part, we will code the core components of the Command pattern and then we will integrate the elements that are necessary to test the replay system:

1. To start, we are implementing a base abstract class named `Command`, which has a singular method named `Execute()`:

```
public abstract class Command
{
    public abstract void Execute();
}
```

2. Now we are going to write three concrete command classes that will derive from the `Command` base class, and then we will implement the `Execute()` method. Each of them encapsulates an action to execute.

   The first one toggles the turbocharger on `BikeController`:

```
namespace Chapter.Command
{
    public class ToggleTurbo : Command
    {
        private BikeController _controller;

        public ToggleTurbo(BikeController controller)
        {
            _controller = controller;
        }

        public override void Execute()
        {
```

```
                _controller.ToggleTurbo();
            }
        }
    }
```

3. The following two commands are the `TurnLeft` and `TurnRight` commands. Each represents a different action and is mapped to a specific input key, as we are going to see when we implement `InputHandler`:

```
namespace Chapter.Command
{
    public class TurnLeft : Command
    {
        private BikeController _controller;

        public TurnLeft(BikeController controller)
        {
            _controller = controller;
        }

        public override void Execute()
        {
            _controller.Turn(BikeController.Direction.Left);
        }
    }
}
```

4. The following command represents the right turn action and, as its name implies, this turns the bike to the right:

```
namespace Chapter.Command
{
    public class TurnRight : Command
    {
        private BikeController _controller;

        public TurnRight(BikeController controller)
        {
            _controller = controller;
        }

        public override void Execute()
        {
            _controller.Turn(BikeController.Direction.Right);
        }
    }
}
```

Now that we have completed the encapsulation of each command into individual classes, it is time to code the critical ingredient that will make our replay system work – `Invoker`.

`Invoker` is an attentive bookkeeper; it keeps track of the commands that have been executed in a ledger. We represent this ledger in code in the form of `SortedList`, a native C# sorted collection with a key/value structure. This list will keep track of when a specific command was executed.

5. Because this class is very long, we will review it in two segments. The following is the first part:

```
using UnityEngine;
using System.Linq;
using System.Collections.Generic;

namespace Chapter.Command
{
    class Invoker : MonoBehaviour
    {
        private bool _isRecording;
        private bool _isReplaying;
        private float _replayTime;
        private float _recordingTime;
        private SortedList<float, Command> _recordedCommands =
            new SortedList<float, Command>();

        public void ExecuteCommand(Command command)
        {
            command.Execute();

            if (_isRecording)
                _recordedCommands.Add(_recordingTime, command);

            Debug.Log("Recorded Time: " + _recordingTime);
            Debug.Log("Recorded Command: " + command);
        }

        public void Record()
        {
            _recordingTime = 0.0f;
            _isRecording = true;
        }
```

In this part of the `Invoker` class, we add a command to the `_recordedCommands` sorted list every time `Invoker` executes a new one. However, we only do this when we start recording because we want to record the player inputs at specific moments, such as at the start of the race.

6. For the next section of the `Invoker` class, we are going to implement the replay behavior:

```
public void Replay()
{
    _replayTime = 0.0f;
    _isReplaying = true;

    if (_recordedCommands.Count <= 0)
        Debug.LogError("No commands to replay!");

    _recordedCommands.Reverse();
}

void FixedUpdate()
{
    if (_isRecording)
        _recordingTime += Time.fixedDeltaTime;

    if (_isReplaying)
    {
        _replayTime += Time.deltaTime;

        if (_recordedCommands.Any())
        {
            if (Mathf.Approximately(
                _replayTime, _recordedCommands.Keys[0])) {

                Debug.Log("Replay Time: " + _replayTime);
                Debug.Log("Replay Command: " +
                          _recordedCommands.Values[0]);

                _recordedCommands.Values[0].Execute();
                _recordedCommands.RemoveAt(0);
            }
        }
        else
        {
            _isReplaying = false;
        }
    }
}
```

```
        }
    }
```

As you may have already noticed, we are using `FixedUpdate()` to record and replay commands. This might seem strange as we usually use `Update()` to listen for player inputs. However, `FixedUpdate()` has the advantage of running in fixed time steps and is helpful for time-dependent but frame rate-independent tasks.

Therefore, we know that the default engine time step is 0.02 seconds, and our timestamps will be in similar increments as we use `Time.fixedDeltaTime` to record the time of executed commands.

However, this also means that we lose precision during the recording phase, as our timestamp is bound to Unity's time-step settings. This loss of precision is tolerable in the context of this example. However, it could become an issue if there are significant inconsistencies between the game play and replay sequences.

In that case, we might need to consider a solution that includes `Update()`, `Time.deltaTime`, and a value that will permit us to set the degree of precision when comparing the recording and replay time. However, this is beyond the scope of this chapter.

We should take note that we are giving `Invoker` the responsibility of bookkeeping but also replaying. It can be argued that we have broken the single responsibility principle by giving `Invoker` too many responsibilities. In this context, this is not an issue; it is just a code example for educational purposes. Still, it would be wise to encapsulate the responsibilities of recording, saving, and replaying commands in a separate class.

Testing the replay system

Now that we have the core ingredients of the Command pattern and our replay system in place, it is time to test whether it works:

1. The first class we will implement is `InputHandler`. Its primary responsibility is to listen for the player's inputs and invoke the appropriate commands. However, because of its length, we will review it in two parts:

```
using UnityEngine;

namespace Chapter.Command
{
    public class InputHandler : MonoBehaviour
    {
```

```
private Invoker _invoker;
private bool _isReplaying;
private bool _isRecording;
private BikeController _bikeController;
private Command _buttonA, _buttonD, _buttonW;

void Start()
{
    _invoker = gameObject.AddComponent<Invoker>();
    _bikeController = FindObjectOfType<BikeController>();

    _buttonA = new TurnLeft(_bikeController);
    _buttonD = new TurnRight(_bikeController);
    _buttonW = new ToggleTurbo(_bikeController);
}

void Update()
{
    if (!_isReplaying && _isRecording)
    {
        if (Input.GetKeyUp(KeyCode.A))
            _invoker.ExecuteCommand(_buttonA);
        if (Input.GetKeyUp(KeyCode.D))
            _invoker.ExecuteCommand(_buttonD);
        if (Input.GetKeyUp(KeyCode.W))
            _invoker.ExecuteCommand(_buttonW);
    }
}
```

In this section of the class, we are initializing our commands and mapping them to specific inputs. Notice that we are passing an instance of `BikeController` in the command's constructor. `InputHandler` is only aware that `BikeController` exists, but does not need to know how it functions. It is the responsibility of the individual command classes to call the proper public methods of the bike's controller depending on the desired action. In the `Update()` loop, we listen for specific key inputs and call on `Invoker` to execute a command associated with a particular input.

This segment of code is what makes it possible to record the inputs of the player. We do not call `BikeController` directly, and we do not execute the commands. Instead, we allow `Invoker` to act like an intermediary and do all the work. This approach permits it to keep records of the player inputs in the background for later use.

2. For the final part of the `InputHandler` class, we are adding some GUI debug buttons that will help us test the replay system. This segment of code is for debugging and testing purposes only:

```
        void OnGUI()
        {
            if (GUILayout.Button("Start Recording"))
            {
                _bikeController.ResetPosition();
                _isReplaying = false;
                _isRecording = true;
                _invoker.Record();
            }

            if (GUILayout.Button("Stop Recording"))
            {
                _bikeController.ResetPosition();
                _isRecording = false;
            }

            if (!_isRecording)
            {
                if (GUILayout.Button("Start Replay"))
                {
                    _bikeController.ResetPosition();
                    _isRecording = false;
                    _isReplaying = true;
                    _invoker.Replay();
                }
            }
        }
    }
}
```

3. For our final class, we will implement a skeleton version of the `BikeController` class for testing purposes. It acts like a receiver in the context of the Command pattern:

```
using UnityEngine;

public class BikeController : MonoBehaviour
{
    public enum Direction
    {
        Left = -1,
        Right = 1
    }
```

```
        private bool _isTurboOn;
        private float _distance = 1.0f;

        public void ToggleTurbo()
        {
            _isTurboOn = !_isTurboOn;
            Debug.Log("Turbo Active: " + _isTurboOn.ToString());
        }

        public void Turn(Direction direction)
        {
            if (direction == Direction.Left)
                transform.Translate(Vector3.left * _distance);
            if (direction == Direction.Right)
                transform.Translate(Vector3.right * _distance);
        }

        public void ResetPosition()
        {
            transform.position = new Vector3(0.0f, 0.0f, 0.0f);
        }
    }
```

The overall purpose and structure of the class are self-explanatory. `ToggleTurbo()` and `Turn()` are public methods that get called by the command classes. However, `ResetPosition()` is for debugging and testing purposes only and can be ignored.

To test this code in your instance of Unity, you need to complete the following steps:

1. Start a new empty Unity scene that includes at least one light and camera.
2. Add a 3D GameObject to the new scene, such as a cube, and make it visible from the scene's main camera.
3. Attach the `InputHandler` and `BikeController` classes to the new GameObject.

4. At runtime, if you have copied all the classes we have just reviewed in your project, you should see the following on your screen:

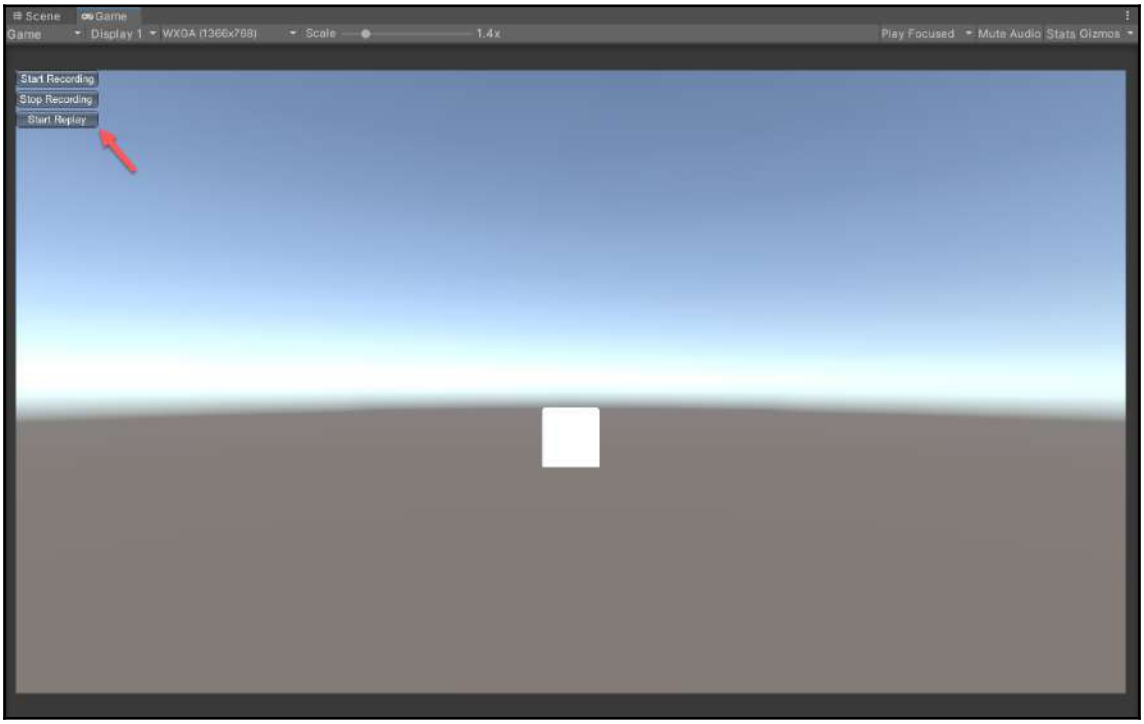


Figure 7.3 – Screenshot of the code in action inside Unity

When you start recording, you will be able to move the cube on a horizontal axis. Each input entered can be replayed in the same sequence and timing they were recorded.

Reviewing the implementation

We completed the process of building a quick input replay system by using the Command pattern as our foundation. Of course, the code example in this chapter is not production-ready and very limited. Nevertheless, the goal of this chapter was to learn how to use the Command pattern with Unity, and not design an entire replay system.

In the game prototype project in the `FPP` folder of the GitHub project, we did implement a more advanced example of a replay system that includes serialization and a rewind feature. We recommend checking it out and, of course, modify it at your discretion.

In the next section, we will review some alternatives solutions and approaches that we could have used to build our replay system.

Reviewing alternative solutions

Even if the Command pattern was perfectly suited for our use case, there are some alternative patterns and solutions we could have considered:

- **Memento:** The Memento pattern provides the ability to roll back an object to a previous state. This was not our first choice for our replay system because we are focusing on recording inputs and queuing them for replay at a later time, which is very compatible with the design intention of the Command pattern. However, if we implemented a system with a rollback to the previous state feature, the Memento pattern will probably be our first choice.
- **Queue/Stack**: Queues and stacks are not patterns, but data structures, but we could simply encode all our inputs and store them in a queue directly in our `InputHandler` class. It would have been more straightforward and less verbose than using the Command pattern. The choice between implementing a system with or without conventional design patterns is very contextual. If a system is simple enough, then the added verbosity and complexity that a design pattern might bring might exceed the potential benefits of using it.

Summary

In this chapter, we implemented a simple, but functional, replay system by using the Command pattern. Our goal with this chapter was not to show how to build a robust replay system, but instead showcase how to use the Command pattern to create something in Unity that might be useful for a game project.

I hope you will research alternative ways of implementing the Command pattern that might be better than shown in this book because, like most things in programming, there is no single way of doing things. Nevertheless, at least this chapter offers a first approach to using the Command pattern with Unity.

In the next part of the book, we will start optimizing our code with the Object pool. An essential aspect of a good racing game is consistent performance and frame rate. Everything must run smoothly at every moment, or it might cause our game to feel slow and sluggish.

8
Optimizing with the Object Pool Pattern

In most video games, there are a lot of things happening on the screen. Bullets are flying around, enemies are spawning around the map, particles are popping up around the player, and these various objects are loaded and rendered on the screen in the blink of an eye. So, to avoid putting strain on the **Central Processing Unit** (**CPU**) while maintaining a consistent frame rate, it's a good practice to reserve some memory for our frequently spawned entities. So, instead of releasing recently destroyed enemies from memory, we add them to an object pool to recycle them for later use. With this technique, we avoid the initial initialization cost of loading a new instance of an entity. In addition, because we are not destroying reusable entities, the **Garbage Collector** (**GC**) won't waste cycles cleaning a set of regularly reinitialized objects.

And this is what we are going to do in this chapter, but we are fortunate as, since Unity version 2021, object pooling has been natively integrated into the **Application Programming Interface** (**API**). Therefore, we will not need to implement the pattern by hand as we have done in previous chapters; instead, we will focus on learning how to use it and let the engine do all the work.

The following topics will be covered in this chapter:

- Understanding the Object Pool pattern
- Implementing the Object Pool pattern
- Reviewing alternative solutions

A GC functions as an automated memory manager and is an essential component of most modern object-oriented languages such as C#. It's not necessary to understand how it works to continue with this chapter, but if you are curious, you can get more information on the subject matter here: `https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/`.

Technical requirements

This chapter is hands-on, so you will need to have a basic understanding of Unity and C#.

The code files of this chapter can be found on GitHub, at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter08`.

Check out the following video to see the code in action:

`https://bit.ly/3yTLcI7`

It's important to note that the code example in the next section will not work in a version of Unity that's below 2021.1, as we are using recently added API features.

Understanding the Object Pool pattern

The core concept of this pattern is simple—a pool in the form of a container holds a collection of initialized objects in memory. Clients can request an Object Pool for an object instance of a specific type; if one is available, it will be removed from the pool and given to the client. If there are not enough pooled instances at a given time, new ones will be dynamically created.

Objects that exit the pool will attempt to return to it once they are not used anymore by the client. If the Object Pool has no more space, it will destroy instances of objects that attempt to return. Therefore, the pool constantly gets refilled, can only be temporarily drained, but never overflows. Hence, its memory usage is consistent.

The following diagram illustrates the back and forth between a client and an Object Pool:

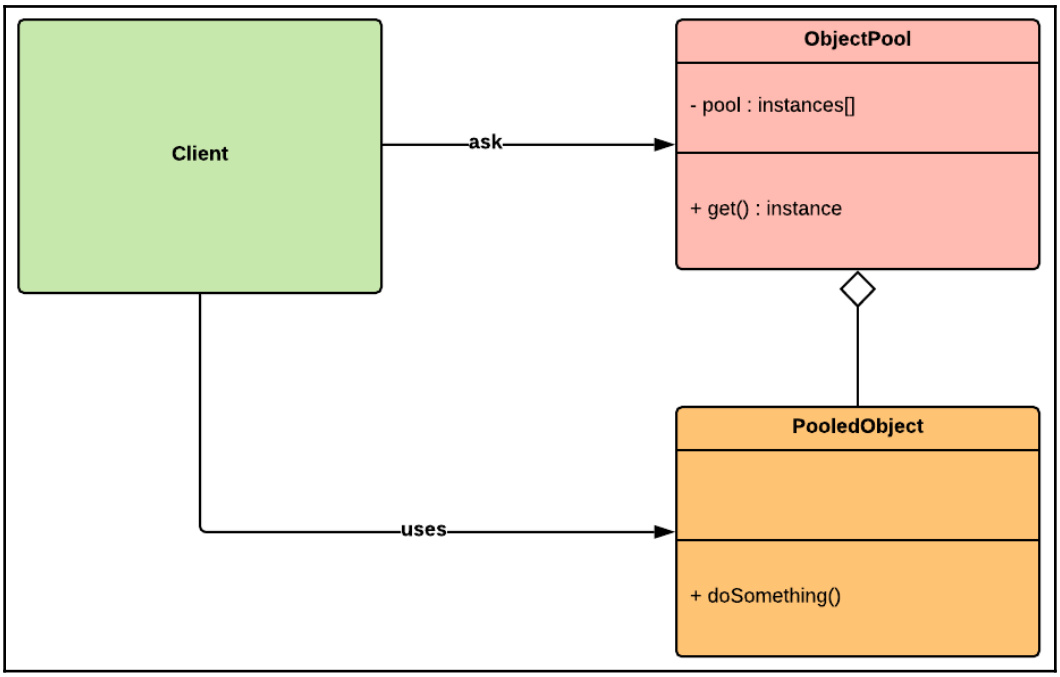


Figure 8.1 – Unified Modeling Language (UML) diagram of the Object Pool pattern

In the diagram, we can see that the Object Pool serves the client by offering it access to a pool of object instances of a specific type—so, for example, a client could be a spawner asking the Object Pool for instances of a particular enemy type.

Benefits and drawbacks of the Object Pool pattern

These are some of the benefits of the Object Pool pattern:

- **Predictable memory usage**: With the Object Pool, we can allocate in a predictable manner some memory to hold a specific amount of instances of certain type of object.
- **Performance boost**: By having objects already initialized in memory, you avoid the loading cost of initializing new ones.

These are some potential drawbacks of the Object Pool pattern:

- **Layering on already managed memory**: Some people criticize the Object Pool pattern as being unnecessary in most cases because modern managed programming languages such as C# already optimally control memory allocation. However, this statement might be true in some contexts but false in others.
- **Unpredictable object states**: A potential pitfall of the Object Pool pattern is that if it is incorrectly handled, objects can be returned to the pool in their current state instead of their initial one. This situation can be an issue when the pooled entity is damageable or destructible. For instance, if you have an enemy entity that just got killed by the player, if you return it to the pool without restoring its health, and when the Object Pool pulls it back out for a client, it will spawn back into the scene in an already damaged state.

When to use the Object Pool pattern

To better understand when to use the Object Pool pattern, let's review when not to use it. For instance, if you have entities that need to be spawned once on your map, such as a final boss, putting it in an Object Pool is a waste of memory that you could use for something more useful.

Also, we should keep in mind that an Object Pool is not a cache. It has a similar purpose—the reuse of objects. The core difference is that an Object Pool has a mechanism in which entities are automatically returned to the pool after being used, and an Object Pool, if well implemented, handles the creation and deletion of objects depending on the available size of the pool.

But suppose we have entities such as bullets, particles, and enemy characters that are frequently spawned and destroyed during a gameplay sequence. In that case, the Object Pool can relieve some of the strain we are putting on the CPU by reducing repetitious lifespan calls such as create and destroy, thus the CPU will be able to reserve processing power for more critical tasks.

In the next section, we are going to take the concepts just learned and translate them into code.

Implementing the Object Pool pattern

Before starting this section, it would be a good idea to read the official API documentation for the `IObjectPool<T0>` class under the `UnityEngine.Pool` namespace at the following link:

`https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Pool.ObjectPool_1.html`

We will try to avoid getting bogged down by API specifications while implementing the following code example. Instead, we will focus on critical elements directly related to the core concepts of object pooling. Also, native object pooling is a relatively new Unity API feature, hence it might be subject to changes and updates. Thus, it would be wise to keep an eye on the documentation in the short term.

Steps for implementing the Object Pool pattern

This section's code example should be relatively straightforward and we should be able to complete it in two steps, as follows:

1. Let's start by implementing our drone, as this is the entity that we will pool. Because it is such a lengthy class, we are going to split it into two segments. You can see the first segment here:

```
using UnityEngine;
using UnityEngine.Pool;
using System.Collections;

namespace Chapter.ObjectPool
{
    public class Drone : MonoBehaviour
    {
        public IObjectPool<Drone> Pool { get; set; }

        public float _currentHealth;

        [SerializeField]
        private float maxHealth = 100.0f;

        [SerializeField]
        private float timeToSelfDestruct = 3.0f;

        void Start()
        {
```

```
        _currentHealth = maxHealth;
    }

    void OnEnable()
    {
        AttackPlayer();
        StartCoroutine(SelfDestruct());
    }

    void OnDisable()
    {
        ResetDrone();
    }
```

It's essential to observe that we call the `ResetDrone()` method in the `OnDisable()` event function in this class segment. We are doing this because we want to reset the drone back to its initial state before returning it to the pool.

And as we are going to see when we implement the Object Pool pattern, when a GameObject returns to the pool it gets disabled, including all its child components. Therefore, if we have any reinitialization code to execute, we can do this at the `OnDisable()` call.

In the context of this chapter, we are keeping things simple; we are only restoring the health of the drone. But in an advanced implementation, we might have to reset visual markers, such as removing damaged decals.

2. In the final segment of our `Drone` class, we will implement the core behaviors, as follows:

```
    IEnumerator SelfDestruct()
    {
        yield return new WaitForSeconds(timeToSelfDestruct);
        TakeDamage(maxHealth);
    }

    private void ReturnToPool()
    {
        Pool.Release(this);
    }

    private void ResetDrone()
    {
        _currentHealth = maxHealth;
    }

    public void AttackPlayer()
```

```
        {
            Debug.Log("Attack player!");
        }

        public void TakeDamage(float amount)
        {
            _currentHealth -= amount;

            if (_currentHealth <= 0.0f)
                ReturnToPool();
        }
    }
}
```

Our drone has two key behaviors, outlined as follows:

- **Self-destruction**: Our drone has a short lifespan; when it's enabled, the `SelfDestruct()` coroutine is called. After several seconds, the drone self-destructs by depleting its health meter and returning itself to the pool by calling the `ReturnToPool()` method.
- **Attack**: The logic inside the method is not implemented, for brevity reasons. But imagine that once the drone is spawned, it seeks and attacks the player.

3. Next up is our `ObjectPool` class, which has the responsibility of managing a pool of drone instances. Because it's a lengthy class, we will review it in two segments, with the first segment available to view here:

```
using UnityEngine;
using UnityEngine.Pool;

namespace Chapter.ObjectPool
{
    public class DroneObjectPool : MonoBehaviour
    {
        public int maxPoolSize = 10;
        public int stackDefaultCapacity = 10;

        public IObjectPool<Drone> Pool
        {
            get
            {
                if (_pool == null)
                    _pool =
                        new ObjectPool<Drone>(
                            CreatedPooledItem,
                            OnTakeFromPool,
                            OnReturnedToPool,
```

```
                                OnDestroyPoolObject,
                                true,
                                stackDefaultCapacity,
                                maxPoolSize);
                    return _pool;
                }
            }

            private IObjectPool<Drone> _pool;
```

In this first part of the script, we are setting a critical variable named `maxPoolSize`; as its name implies, this sets the maximum number of drone instances we will keep in the pool. The `stackDefaultCapacity` variable sets the default stack capacity; this is a property related to the stack data structure we are using to store our drone instance. We can ignore it for the moment as it's not critical to our implementation.

In the following code snippet, we are initializing the object pool, which is the most crucial part of our class:

```
public IObjectPool<Drone> Pool
{
 get
 {
 if (_pool == null)
             _pool =
                 new ObjectPool<Drone>(
                     CreatedPooledItem,
                     OnTakeFromPool,
                     OnReturnedToPool,
                     OnDestroyPoolObject,
                     true,
                     stackDefaultCapacity,
                     maxPoolSize);
        return _pool;
    }
}
```

It's important to note that we are passing callback methods in the constructor of the `ObjectPool<T>` class, and it's in these callbacks that we will implement the logic that will drive our Object Pool.

4. In the last segment of the `DroneObjectPool` class, we will implement the callbacks we declared in the `ObjectPool<T>` constructor, as follows:

```
            private Drone CreatedPooledItem()
            {
                var go =
                    GameObject.CreatePrimitive(PrimitiveType.Cube);

                Drone drone = go.AddComponent<Drone>();

                go.name = "Drone";
                drone.Pool = Pool;

                return drone;
            }

            private void OnReturnedToPool(Drone drone)
            {
                drone.gameObject.SetActive(false);
            }

            private void OnTakeFromPool(Drone drone)
            {
                drone.gameObject.SetActive(true);
            }

            private void OnDestroyPoolObject(Drone drone)
            {
                Destroy(drone.gameObject);
            }

            public void Spawn()
            {
                var amount = Random.Range(1, 10);

                for (int i = 0; i < amount; ++i)
                {
                    var drone = Pool.Get();

                    drone.transform.position =
                        Random.insideUnitSphere * 10;
                }
            }
        }
    }
```

Here's a short explanation of each callback that the `ObjectPool` class will call at specific times:

- `CreatedPooledItem()`: In this callback, we are initializing our drone instances. In the context of this chapter, we are creating a GameObject from scratch for simplicity reasons, but in a more practical context, we would probably just load a prefab.
- `OnReturnedToPool()`: The name of the method implies its use. Notice we are not destroying the GameObject; we simply deactivate it to remove it from the scene.
- `OnTakeFromPool()`: This is called when the client requests an instance of the drone. The instance is not actually returned—the GameObject is enabled.
- `OnDestroyPoolObject()`: This an important method to understand. It's called when there's no more space in the pool. In that case, the returned instance is destroyed to free up memory.

Our `DroneObjectPool` class has taken on extra responsibilities and acts as a spawner, as we can see in the `Spawn()` method. When requested, it will get a `drone` instance from the pool and spawn it at a random location in the scene within a specific range.

Testing the Object Pool implementation

To test our implementation of the Object Pool in your own instance of Unity, you need to carry out the following steps:

1. Create a new empty Unity scene.
2. Copy all the scripts we just reviewed and save them in your project.
3. Add an empty GameObject to the scene.
4. Attach the following client script to your empty GameObject:

```
using UnityEngine;

namespace Chapter.ObjectPool
{
    public class ClientObjectPool : MonoBehaviour
    {
        private DroneObjectPool _pool;

        void Start()
        {
            _pool = gameObject.AddComponent<DroneObjectPool>();
```

```
                }

                void OnGUI()
                {
                    if (GUILayout.Button("Spawn Drones"))
                        _pool.Spawn();
                }
            }
        }
```

Once you start your scene, you should see a **graphical user interface** (**GUI**) button named **Spawn Drones** in the top-left corner, as we can see in the following screenshot:

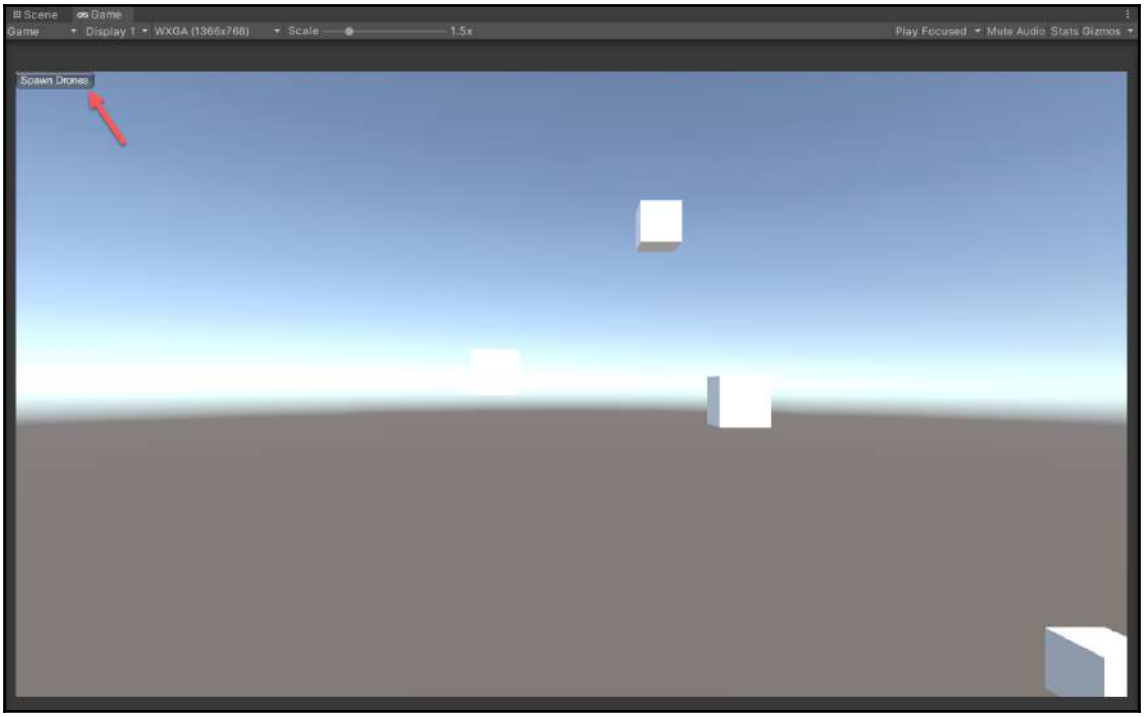


Figure 8.2 – Screenshot of the code example in action

By pressing the **Spawn Drones** button, you can now spawn drones at random positions in the scene. If you want to see the Object Pool mechanism in action, keep an eye on the scene hierarchy—you will be able to see Drone entities get enabled and disabled as they come in and out of the pool.

Reviewing the Object Pool implementation

By using the Object Pool pattern, we automated the process of creating, destroying, and pooling drone instances. We can now reserve an amount of memory to spawn waves of drones while avoiding burdening the CPU. We added optimization and scalability to our code by implementing this pattern without losing readability or adding complexity.

In the next section, we will review some alternative solutions to consider; it's always a good practice to consider other options before deciding on a specific pattern.

Reviewing alternative solutions

A close cousin of the Object Pool pattern is the **Prototype** pattern; both of these are considered creational patterns. With the Prototype pattern, you can avoid the inherent cost of creating new objects by using a cloning mechanism. Therefore, instead of initializing new objects, you clone them from a reference object called a prototype. But in the context of the use case presented in this chapter, object pooling offers better optimization benefits.

Creational-type patterns are concerned with the mechanism of object creation. **Factory**, **Build**, **Singleton**, **Object Pool**, and **Prototype** are all creational design patterns.

Summary

We have just added the Object Pool pattern to our toolkit—this is one of the most valuable patterns for Unity developers. As we saw in our code examples, we can easily recycle instances of frequently used objects. When dealing with a large set of entities that need to spawn quickly and repeatedly, this pattern can help us avoid CPU spikes and lags. These benefits can only help make our game better, as players do enjoy a game that runs smoothly.

In the next chapter, we are going to decouple components from each other with the Observer pattern.

9
Decoupling Components with the Observer Pattern

A common challenge in Unity development is to find elegant ways to decouple components from each other. It's a significant hurdle to overcome when writing code in the engine as it offers us so many ways to reference components through the API and the Inspector directly. But this flexibility can come with a price, and it can make your code fragile since, in some instances, it can take a single missing reference to break your game.

So, in this chapter, we will use the Observer pattern to set up relationships with core components. These relationships will be mapped by assigning objects the role of the Subject or Observer. This approach will not altogether remove coupling between our components but will loosen it up and organize it logically. It will also establish an event handling system with a one-to-many structure, which is precisely what we need to implement in the use case presented in this chapter.

If you are looking for a way to decouple objects from each other with events in a many-to-many relationship, please check out `Chapter 6`, *Managing Game Events with the Event Bus*.

The following topics will be covered in this chapter:

- Understanding the Observer pattern
- Decoupling core components with the Observer pattern
- Implementing the Observer pattern
- Reviewing alternative solutions

Technical requirements

The code files for this chapter can be found on GitHub: `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter09`.

Check out the following video to see the code in action: `https://bit.ly/3xDlBDa`.

Understanding the Observer pattern

The core purpose of the Observer pattern is to establish a one-to-many relationship between objects in which one acts as the subject while the others take the role of observers. The subject then assumes the responsibility of notifying the observers when something inside itself changes and might concern them.

It's somewhat similar to a publisher and subscriber relationship, in which objects subscribe and listen for specific event notifications. The core difference is that the subject and observers are aware of each other in the Observer pattern, so they are still lightly coupled together.

Let's review a UML diagram of a typical implementation of the Observer pattern to see how this might work when implemented in code:

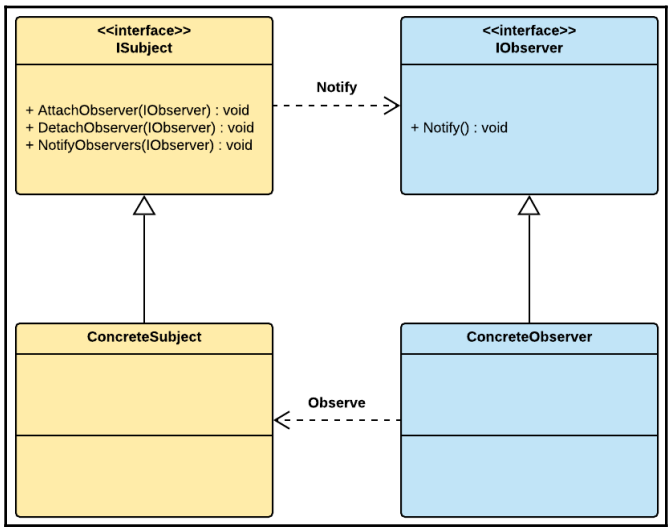


Figure 9.1 – UML diagram of the Observer pattern

As you can see, the Subject and the Observer both have respective interfaces that they implement, but the most important one to analyze is `ISubject`, which includes the following methods:

- `AttachObserver()`: This method allows you to add an observer object to the list of observers to notify.
- `DetachObserver()`: This method removes an observer from the list of observers.
- `NotifyObservers()`: This method notifies all the objects that have been added to the subject's list of observers.

An object that takes the role of Observer must implement a public method called `Notify()`, which will be used by the subject to notify it when it changes states.

Benefits and drawbacks of the Observer pattern

These are some of the benefits of the Observer pattern:

- **Dynamism**: The Observer permits the Subject to add as many objects as it necessitates as Observers. But it can also remove them dynamically at runtime.
- **One-to-Many**: The main benefit of the Observer pattern is that it elegantly solves the problem of implementing an event handling system in which there's a one-to-many relationship between objects.

The following are some potential drawbacks of the Observer pattern:

- **Disorder**: The Observer pattern doesn't guarantee the order in which Observers get notified. So, if two or more Observer objects share dependencies and must work together in a specific sequence, the Observer pattern, in its native form, is not designed to handle that type of execution context.
- **Leaking**: The Observer can cause memory leaks as the subject holds strong references to its observers. If it's implemented incorrectly and the Observer objects are not correctly detached and disposed of when they are not needed anymore, it could cause issues with garbage collection, and some resources will not be liberated.

To understand the potential memory leaking drawback indicated here, I recommend reading the following Wikipedia article on the subject matter: `https://en.wikipedia.org/wiki/Lapsed_listener_problem`.

But take into account that, like anything related to optimization, it's contextual, so you should profile your code before optimizing for potential performance issues.

When to use the Observer pattern

The advantage of the Observer pattern is that it solves specific problems related to one-to-many relationships between objects. So, if you have a core component that often changes states and has many dependencies that need to react to those changes, then the Observer pattern permits you to define a relationship between those entities and a mechanism that enables them to be notified.

Therefore, if you are unsure when to use the Observer pattern, you should analyze the relationship between your objects to determine if this pattern is well-suited for the problem you are trying to solve.

Decoupling core components with the Observer pattern

The core ingredient of our game is the racing bike. It's the entity in our scene that changes states and updates its properties the most often as it's under the player's control while traveling around the world and interacting with other entities. It has several dependencies to manage, such as the main camera that follows it and the HUD that displays its current speed.

The racing bike is the main subject of our game, and many systems must observe it so that they can update themselves when it changes states. For instance, every time the bike collides with an obstacle, the HUD must update the current value of the shield's health, and the camera displays a full screen shader that darkens the edges of the screen to showcase the diminishing endurance.

This type of behavior is easy to implement in Unity. We could have `BikeController` tell `HUDController` and `CameraController` what to do when it takes damage. But for this approach to work, `BikeController` must be aware of the public methods to call on each controller.

As you can imagine, this doesn't scale well since as the complexity of `BikeController` grows, the more calls to its dependencies we will have to manage. But with the Observer pattern, we are going to break this coupling between the controllers. First, we will give each component a role; `BikeController` will become a subject and be responsible for managing a list of dependencies and notifying them when necessary.

The HUD and Camera controllers will act as observers of `BikeController`**.** Their core responsibility will be to listen for notifications coming from `BikeController` and act accordingly. `BikeController` doesn't tell them what to do; it just tells them something has changed and lets them react to it at their discretion.

The following diagram illustrates the concept we just reviewed:

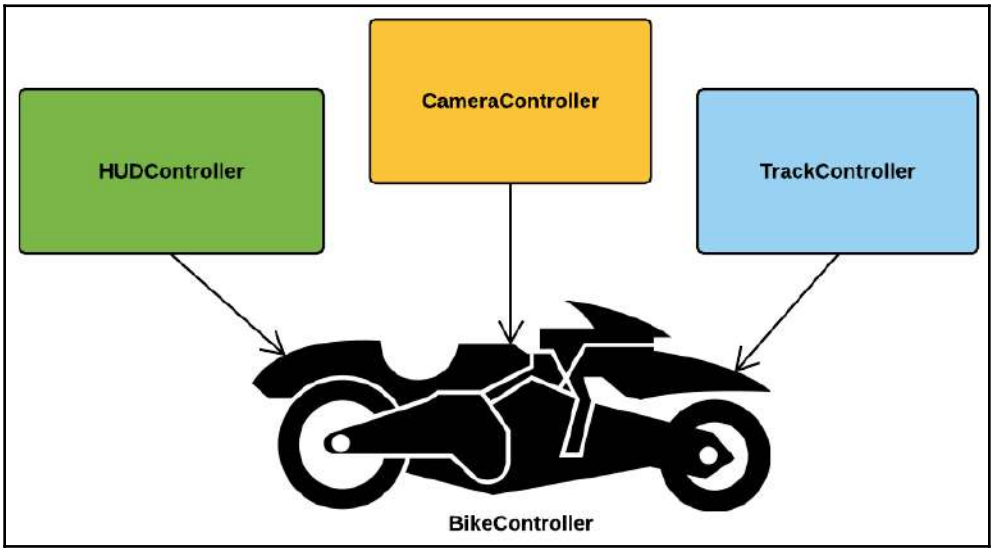


Figure 9.2 – Illustration of the controllers observing a subject

As we can see, we can have as many controllers observing the bike (subject) as we need. In the next section, we will take these concepts and translate them into code.

Implementing the Observer pattern

Now, let's implement the Observer pattern in a simple way that can be reused in various contexts:

1. We are going to start this code example by implementing the two elements of the pattern. Let's begin with the `Subject` class:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Observer
{
    public abstract class Subject : MonoBehaviour
    {
        private readonly
            ArrayList _observers = new ArrayList();

        public void Attach(Observer observer)
        {
            _observers.Add(observer);
        }

        public void Detach(Observer observer)
        {
            _observers.Remove(observer);
        }

        public void NotifyObservers()
        {
            foreach (Observer observer in _observers)
            {
                observer.Notify(this);
            }
        }
    }
}
```

The `Subject` abstract class has three methods. The first two, `Attach()` and `Detach()`, are responsible for adding or removing an observer object from a list of observers, respectively. The third method, `NotifyObservers()` has the responsibility of looping through the list of observer objects and calling their public method, called `Notify()`. This will make sense when we implement concrete observer classes in the upcoming steps.

2. Next up is the `Observer` abstract class:

```
using UnityEngine;

namespace Chapter.Observer
{
    public abstract class Observer : MonoBehaviour
    {
        public abstract void Notify(Subject subject);
    }
}
```

Classes that wish to become observers have to inherit this `Observer` class and implement the abstract method named `Notify()`, which receives the subject as a parameter.

3. Now that we have our core ingredients, let's write a skeleton `BikeController` class that will act as our subject. However, because it's so long, we will split it into three segments. The first segment is just initialization code:

```
using UnityEngine;

namespace Chapter.Observer
{
    public class BikeController : Subject
    {
        public bool IsTurboOn
        {
            get; private set;
        }

        public float CurrentHealth
        {
            get { return health; }
        }

        private bool _isEngineOn;
        private HUDController _hudController;
        private CameraController _cameraController;

        [SerializeField]
        private float health = 100.0f;

        void Awake()
        {
            _hudController =
                gameObject.AddComponent<HUDController>();
```

```
        _cameraController =
            (CameraController)
            FindObjectOfType(typeof(CameraController));
    }

    private void Start()
    {
        StartEngine();
    }
```

The following segment is important because we are attaching our observers when `BikeController` is enabled but also detaching them when it's disabled; this avoids us having to hold on to references we don't need anymore:

```
    void OnEnable()
    {
        if (_hudController)
            Attach(_hudController);

        if (_cameraController)
            Attach(_cameraController);
    }

    void OnDisable()
    {
        if (_hudController)
            Detach(_hudController);

        if (_cameraController)
            Detach(_cameraController);
    }
```

And for the final part, we have some basic implementations of core behaviors. Note that we only notify the observers when the bike's parameters get updated when it takes damage, or when its turbocharger is activated:

```
    private void StartEngine()
    {
        _isEngineOn = true;
        NotifyObservers();
    }

    public void ToggleTurbo()
    {
        if (_isEngineOn)
            IsTurboOn = !IsTurboOn;

        NotifyObservers();
```

```
        }

        public void TakeDamage(float amount)
        {
            health -= amount;
            IsTurboOn = false;

            NotifyObservers();

            if (health < 0)
                Destroy(gameObject);
        }
    }
}
```

`BikeController` never calls `HUDController` or `CameraController` directly; it only notifies them that something has changed – it never tells them what to do. This is important because the observers can independently choose how to behave when being notified. Therefore, they have been decoupled from the subject to a certain degree.

4. Now, let's implement some observers and see how they behave when the subject signals them. We'll begin with `HUDController`, which has the responsibility of displaying the user interface:

```
using UnityEngine;

namespace Chapter.Observer {
    public class HUDController : Observer {

        private bool _isTurboOn;
        private float _currentHealth;
        private BikeController _bikeController;

        void OnGUI() {
            GUILayout.BeginArea (
                new Rect (50,50,100,200));
            GUILayout.BeginHorizontal ("box");
            GUILayout.Label ("Health: " + _currentHealth);
            GUILayout.EndHorizontal ();

            if (_isTurboOn) {
                GUILayout.BeginHorizontal("box");
                GUILayout.Label("Turbo Activated!");
                GUILayout.EndHorizontal();
            }
```

```
                if (_currentHealth <= 50.0f) {
                    GUILayout.BeginHorizontal("box");
                    GUILayout.Label("WARNING: Low Health");
                    GUILayout.EndHorizontal();
                }

                GUILayout.EndArea ();
            }

            public override void Notify(Subject subject) {
                if (!_bikeController)
                    _bikeController =
                        subject.GetComponent<BikeController>();

                if (_bikeController) {
                    _isTurboOn =
                        _bikeController.IsTurboOn;
                    _currentHealth =
                        _bikeController.CurrentHealth;
                }
            }
        }
    }
```

The `Notify()` method of `HUDController` receives a reference to the subject that notified it. Therefore, it can access its properties and choose which one to display in the interface.

5. Lastly, we are going to implement `CameraController`. The expected behavior of the camera is to start shaking when the turbo booster of the bike is activated:

```
using UnityEngine;

namespace Chapter.Observer
{
    public class CameraController : Observer
    {
        private bool _isTurboOn;
        private Vector3 _initialPosition;
        private float _shakeMagnitude = 0.1f;
        private BikeController _bikeController;

        void OnEnable()
        {
            _initialPosition =
                gameObject.transform.localPosition;
        }
```

```
                void Update()
                {
                    if (_isTurboOn)
                    {
                        gameObject.transform.localPosition =
                            _initialPosition +
                            (Random.insideUnitSphere * _shakeMagnitude);
                    }
                    else
                    {
                        gameObject.transform.
                            localPosition = _initialPosition;
                    }
                }

                public override void Notify(Subject subject)
                {
                    if (!_bikeController)
                        _bikeController =
                            subject.GetComponent<BikeController>();

                    if (_bikeController)
                        _isTurboOn = _bikeController.IsTurboOn;
                }
            }
        }
```

`CameraController` checks the public Boolean property of the subject that just notified it and if it's true, it starts shaking the camera until it gets notified again by `BikeController` and confirms that the turbo toggle is off.

The main takeaway of this implementation to keep in mind is that `BikeController` (**subject**) is not aware of how the HUD and Camera controller (**observers**) will behave once they are notified. Hence, it's at the discretion of the observers to choose how they will react as the subject alerts them of a change.

This approach decouples these controller components from each other. Therefore, it's way easier to implement and debug them individually.

Testing the Observer pattern implementation

To test our implementation, we have to do the following:

1. Open an empty Unity scene but make sure it includes at least one camera and one light.
2. Add a 3D GameObject, such as a cube, to the scene and make it visible to the camera.
3. Attach the `BikeController` script as a component to the new 3D object.
4. Attach the `CameraController` script to the main scene camera.
5. Create an empty GameObject, add the following `ClientObserver` script to it, and then start the scene:

```
using UnityEngine;

namespace Chapter.Observer
{
    public class ClientObserver : MonoBehaviour
    {
        private BikeController _bikeController;

        void Start()
        {
            _bikeController =
                (BikeController)
                FindObjectOfType(typeof(BikeController));
        }

        void OnGUI()
        {
            if (GUILayout.Button("Damage Bike"))
                if (_bikeController)
                    _bikeController.TakeDamage(15.0f);

            if (GUILayout.Button("Toggle Turbo"))
                if (_bikeController)
                    _bikeController.ToggleTurbo();
        }
    }
}
```

We should see GUI buttons and labels on the screen, similar to the following:

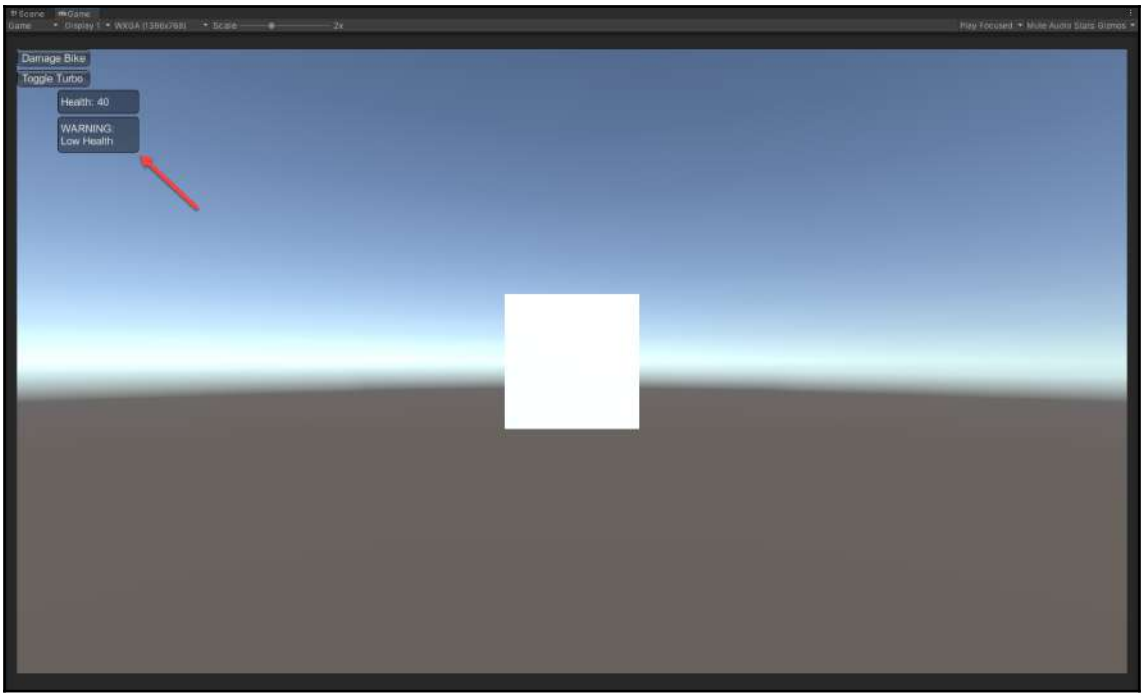


Figure 9.3 – The GUI elements in a running Unity scene

If we press the **Toggle Turbo** button, we will see the camera shake and the HUD display the status of the turbo booster. The **Damage Bike** button will reduce the health of the bike.

There are many different ways you can implement the Observer pattern, each with its intrinsic benefits. I can't cover them all in this chapter. Due to this, I wrote the code example in this chapter for educational purposes in mind. Hence, it's not the most optimized approach but one that's easy to understand.

Reviewing alternative solutions

An alternative to the Observer pattern is the native C# event system. One of the significant advantages of this event system is that it's more granular than the Observer pattern because objects can listen to specific events that another emits, instead of getting a general notification from a subject.

A native event system should always be considered if you need to have components interact through events, especially if you don't need to establish a specific relationship between them.

Unity has its own native event system; it's very similar to the C# version but with added engine features, such as the ability to wire events and actions through the Inspector. To learn more, go to `https://docs.unity3d.com/2021.2/Documentation/Manual/UnityEvents.html`.

Summary

In this chapter, we learned how to use the Observer pattern to decouple `BikeController` from its dependencies by assigning them the roles of subject or observer. Our code is now easier to manage and extend as we can easily have `BikeController` interact with other controllers with minimal coupling.

In the next chapter, we will explore the Visitor pattern, one of the most challenging patterns to learn. We will use it to build power-ups, a core mechanic and ingredient of our game.

10
Implementing Power-Ups with the Visitor Pattern

In this chapter, we are going to implement a power-up mechanic for our game. Power-ups have been a core ingredient of video games since their early inception. One of the first games that implemented power-ups is *Pac-Man* from 1980. In the game, you could eat Power Pellets that would give Pac-Man temporary invincibility. Another classic example is the mushrooms in *Mario Bros.*, which made Mario taller and more robust.

The power-up ingredients that we will build will be similar to the classics but with a little more granularity. Instead of having a power-up boost a singular ability of an entity, we can create combos that give out multiple benefits at once. For instance, we could have a power-up named "**Protector**" that adds durability to the front-facing shield and increments the strength of the primary weapon.

And so, in this chapter, we are going to implement a power-up mechanic that's scalable and configurable, not just for us programmers but also for designers that might be assigned the responsibility of creating and adjusting unique power-up ingredients. We will achieve this using a combination of the Visitor pattern and a unique Unity API feature named ScriptableObjects.

The following topics will be covered in this chapter:

- The basic principles behind the Visitor pattern
- The implementation of a power-up mechanic for a racing game

This section includes simplified code examples for the sake of simplicity and readability. If you wish to review a complete implementation in the context of an actual game project, open the `FPP` folder in the GitHub project, the link for which can be found under the *Technical requirements* section.

Technical requirements

This chapter is hands-on. You will need to have a basic understanding of Unity and C#. We will be using the following Unity engine and C# language concepts:

- Interfaces
- ScriptableObjects

If you are unfamiliar with these concepts, please review them before starting this chapter. The code files for this chapter can be found on `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter10`.

Check out the following video to see the code in action: `https://bit.ly/3eeknGC`.

We often use ScriptableObjects in the code examples of this book because when building game systems and mechanics, it is essential to make them easily configurable by non-programmers. The process of balancing systems and authoring new ingredients usually falls under the responsibility of game and level designers. Therefore, we use ScriptableObjects because it offers a consistent way of establishing an authoring pipeline to create and configure in-game assets.

Understanding the Visitor pattern

The primary purpose of the Visitor pattern is simple once you grasp it; a *Visitable* object permits a *Visitor* to operate on a specific element of its structure. This process allows the visited object to acquire new functionality from visitors without being directly modified. This description might seem very abstract at first, but it is easier to visualize if we imagine an object as a structure instead of a closed-off container of data and logic. Therefore, it is possible with the Visitor pattern to traverse an object's structure, operate on its elements, and extend its functionality without modifying it.

Another way to imagine the Visitor pattern in action is by visualizing the bike in our game colliding with a power-up. Like an electronic current, the power-up flows through the inner structure of the vehicle. Components marked as visitable get visited by the power-up, and new functionalities are added, but nothing is modified.

In the following diagram, we can visualize those principles:

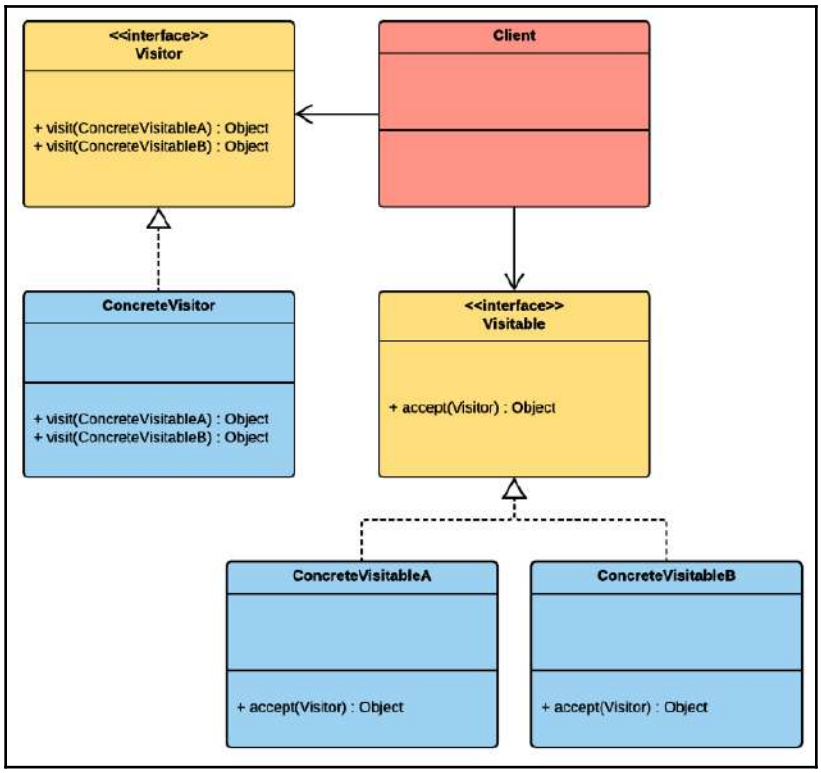


Figure 10.1 - UML diagram of the Visitor pattern

There are two key participants in this pattern that we need to know well:

- **IVisitor** is the interface that a class that wishes to be a visitor must implement. The visitor class will have to implement a visitor method per visitable element.
- **IVisitable** is the interface that classes that wish to become visitable must implement. It includes an `accept()` method that offers an entry point to a visitor object to come and visit.

Before proceeding, it is essential to declare that the code example that we will review in the upcoming sections breaks a potential rule of the Visitor pattern. In the example, the visitor object changes some of the properties of the visited objects. However, whether this transgresses the integrity of the pattern and invalidates its original design intention is open to debate.

Nevertheless, in this chapter, we focus more on how the Visitor pattern permits us to traverse the elements that compose the structure of visitable objects. In our use case, this structure represents the core elements of our bike, which include the engine, shield, and primary weapon.

The Visitor is considered by some to be one of the hardest patterns to understand. So don't feel bad if you don't grasp its core concepts at first. I believe one reason it's a difficult pattern to learn is that it's one of the hardest patterns to explain.

Benefits and drawbacks of the Visitor pattern

I've written a short list of the benefits and drawbacks of using this pattern.

The following are the benefits:

- **Open/Closed**: You can add new behaviors that can work with objects of different classes without modifying them directly. This approach follows the object-oriented programming principle of Open/Closed that states that entities should be open for extension but closed for modification.
- **Single Responsibility**: The Visitor pattern can adhere to the single responsibility principle in the sense you can have an object (visitable) that holds the data and another object (visitor) is responsible for introducing specific behaviors.

The following are some of the potential drawbacks:

- **Accessibility**: Visitors could lack the necessary access to specific private fields and methods of the elements that they are visiting. Therefore, we might need to expose more public properties in our classes than we usually would if we didn't use the pattern.
- **Complexity**: We could argue that the Visitor pattern is structurally more complex than straightforward patterns such as the Singleton, State, and Object Pool. Therefore, it could bring a degree of complexity to your code base that other programmers might find confusing if they are not familiar with the structure and intricacies of the pattern.

The Visitor uses a software engineering concept in its fundamental design named **Double Dispatch**. The simplest definition of the concept is that it's a mechanism that relegates a method call to different concrete methods depending on the types of two objects implicated in the call at runtime. *It's not essential to comprehend this concept completely to follow the example of the pattern presented in this chapter.*

Designing a power-up mechanic

As mentioned at the beginning of this chapter, the power-up is a staple of video games. And it's a core ingredient of our game. But first, we will review some of the key specifications of our mechanic:

- **Granularity**: Our power-up entities will have the ability to boost multiple properties at the same time. For example, we could have a power-up that increases offensive capabilities such as the primary weapon's range while repairing the front-facing shield.
- **Time**: The power-up effects are not temporal so they don't expire after a certain amount of time. And the benefits of the next power-up are added on top of the previous one until they hit the maximum settings of the boosted properties.

Take note that these specifications are limited to the following code example, but are not final. We could easily make the benefits of a power-up temporary or change the overall design of the mechanic with slight changes to the code example presented in the next section.

Our level designers will position the power-ups in strategic points across the race track; they will have 3D shapes that are easy to spot at high speed. The player will have to collide with a power-up to activate the abilities and benefits it contains.

We will use a combination of the Visitor pattern and ScriptableObjects to implement this game mechanic so designers can author new variations of power-ups without needing to write a single line of code.

In video games, the core difference between an item and a power-up is that a player can collect items, store them, and choose when to use their benefits. But in contrast, a power-up takes effect immediately after the player touches it.

Implementing a power-up mechanic

In this section, we will write the necessary skeleton code to implement a power-up system with the Visitor pattern. Our goal is to have a valid proof of concept by the end of this section.

Implementing the power-up system

Let's look at the steps for implementation:

1. We'll start by writing a core element of the pattern, the `Visitor` interface:

```
namespace Pattern.Visitor
{
    public interface IVisitor
    {
        void Visit(BikeShield bikeShield);
        void Visit(BikeEngine bikeEngine);
        void Visit(BikeWeapon bikeWeapon);
    }
}
```

2. Next up, we are going to code an interface that each visitable element will have to implement:

```
namespace Pattern.Visitor
{
    public interface IBikeElement
    {
        void Accept(IVisitor visitor);
    }
}
```

3. Now that we have our primary interfaces, let's implement the main class that makes our power-up mechanism work; because of its length, we will review it in two parts:

```
using UnityEngine;

namespace Pattern.Visitor
{
    [CreateAssetMenu(fileName = "PowerUp", menuName = "PowerUp")]
    public class PowerUp : ScriptableObject, IVisitor
    {
        public string powerupName;
```

```
public GameObject powerupPrefab;
public string powerupDescription;

[Tooltip("Fully heal shield")]
public bool healShield;

[Range(0.0f, 50f)]
[Tooltip(
    "Boost turbo settings up to increments of 50/mph")]
public float turboBoost;

[Range(0.0f, 25)]
[Tooltip(
    "Boost weapon range in increments of up to 25 units")]
public int weaponRange;

[Range(0.0f, 50f)]
[Tooltip(
    "Boost weapon strength in increments of up to 50%")]
public float weaponStrength;
```

The first thing to notice is that this class is a `ScriptableObject` with a `CreateAssetMenu` attribute. Therefore, we will be able to use it to create new power-up assets from the Asset menu. And then, we will be able to configure the parameters of each new power-up in the engine's Inspector. But another important detail is that this class implements the `IVisitor` interface, which we will review in the following part:

```
public void Visit(BikeShield bikeShield)
{
    if (healShield)
        bikeShield.health = 100.0f;
}

public void Visit(BikeWeapon bikeWeapon)
{
    int range = bikeWeapon.range += weaponRange;

    if (range >= bikeWeapon.maxRange)
        bikeWeapon.range = bikeWeapon.maxRange;
    else
        bikeWeapon.range = range;

    float strength =
        bikeWeapon.strength +=
            Mathf.Round(
                bikeWeapon.strength
```

```
                            * weaponStrength / 100);

                if (strength >= bikeWeapon.maxStrength)
                    bikeWeapon.strength = bikeWeapon.maxStrength;
                else
                    bikeWeapon.strength = strength;
            }

            public void Visit(BikeEngine bikeEngine)
            {
                float boost = bikeEngine.turboBoost += turboBoost;

                if (boost < 0.0f)
                    bikeEngine.turboBoost = 0.0f;

                if (boost >= bikeEngine.maxTurboBoost)
                    bikeEngine.turboBoost = bikeEngine.maxTurboBoost;
            }
        }
    }
```

As we can see, for each visitable element, we have a unique method associated with it; inside each of them, we implement the operation we want to execute when visiting a specific element.

In our case, we are changing specific properties of the visited object while taking into account the defined maximum values. Therefore, we encapsulate the expected behavior of a power-up when it visits a specific visitable element of the bike's structure inside individual `Visit()` methods.

We need to change specific values, modify operations, and add new behaviors for a particular visitable element; we can do these things inside this single class.

4. Next up is the `BikeController` class, responsible for controlling the bike's key components that comprise its structure:

```
using UnityEngine;
using System.Collections.Generic;

namespace Pattern.Visitor
{
    public class BikeController : MonoBehaviour, IBikeElement
    {
        private List<IBikeElement> _bikeElements =
            new List<IBikeElement>();

        void Start()
```

```
            {
                _bikeElements.Add(
                    gameObject.AddComponent<BikeShield>());

                _bikeElements.Add(
                    gameObject.AddComponent<BikeWeapon>());

                _bikeElements.Add(
                    gameObject.AddComponent<BikeEngine>());
            }

            public void Accept(IVisitor visitor)
            {
                foreach (IBikeElement element in _bikeElements)
                {
                    element.Accept(visitor);
                }
            }
        }
    }
```

Notice that the class is implementing the `Accept()` method from the `IBikeElement` interface. This method will get called automatically when the bike collides with a power-up item positioned on the race track. And through this method, a power-up entity will be able to pass a visitor object to the `BikeController`.

The controller will proceed to forward the received visitor object to each of its visitable elements. And the visitable elements will get their properties updated as configured in the instance of the visitor object. Hence, this is how the power-up mechanism is triggered and operates.

5. It's time to implement our individual visitable elements, starting with our skeleton `BikeWeapon` class:

```
using UnityEngine;

namespace Pattern.Visitor
{
    public class BikeWeapon : MonoBehaviour, IBikeElement
    {
        [Header("Range")]
        public int range = 5;
        public int maxRange = 25;

        [Header("Strength")]
        public float strength = 25.0f;
```

```
            public float maxStrength = 50.0f;

            public void Fire()
            {
                Debug.Log("Weapon fired!");
            }

            public void Accept(IVisitor visitor)
            {
                visitor.Visit(this);
            }

            void OnGUI()
            {
                GUI.color = Color.green;

                GUI.Label(
                    new Rect(125, 40, 200, 20),
                    "Weapon Range: " + range);

                GUI.Label(
                    new Rect(125, 60, 200, 20),
                    "Weapon Strength: " + strength);
            }
        }
    }
```

6. The following is the `BikeEngine` class; take note that in a complete implementation, this class would have the responsibility of simulating some of the behaviors of an engine, including activating the turbocharger, managing the cooling system, and controlling the speed:

```
using UnityEngine;

namespace Pattern.Visitor
{
    public class BikeEngine : MonoBehaviour, IBikeElement
    {
        public float turboBoost = 25.0f; // mph
        public float maxTurboBoost = 200.0f;

        private bool _isTurboOn;
        private float _defaultSpeed = 300.0f; // mph

        public float CurrentSpeed
        {
            get
            {
```

```
                if (_isTurboOn)
                    return _defaultSpeed + turboBoost;
                return _defaultSpeed;
            }
        }

        public void ToggleTurbo()
        {
            _isTurboOn = !_isTurboOn;
        }

        public void Accept(IVisitor visitor)
        {
            visitor.Visit(this);
        }

        void OnGUI()
        {
            GUI.color = Color.green;

            GUI.Label(
                new Rect(125, 20, 200, 20),
                "Turbo Boost: " + turboBoost);
        }
    }
}
```

7. And finally, `BikeShield`, the name of which implies its main function, is done as follows:

```
using UnityEngine;

namespace Pattern.Visitor
{
    public class BikeShield : MonoBehaviour, IBikeElement
    {
        public float health = 50.0f; // Percentage

        public float Damage(float damage)
        {
            health -= damage;
            return health;
        }

        public void Accept(IVisitor visitor)
        {
            visitor.Visit(this);
        }
```

```
            void OnGUI()
            {
                GUI.color = Color.green;

                GUI.Label(
                    new Rect(125, 0, 200, 20),
                    "Shield Health: " + health);
            }
        }
    }
```

As we can see, each individual visitable bike element class implements the `Accept()` method, thus making themselves visitable.

Testing the power-up system implementation

To quickly test our implementation in your own instance of Unity, you need to follow these steps:

1. Copy all the scripts we just reviewed into your Unity project.
2. Create a new scene.
3. Add a GameObject to the scene.
4. Attach the following `ClientVisitor` script to the new GameObject:

```
using UnityEngine;

namespace Pattern.Visitor
{
    public class ClientVisitor : MonoBehaviour
    {
        public PowerUp enginePowerUp;
        public PowerUp shieldPowerUp;
        public PowerUp weaponPowerUp;

        private BikeController _bikeController;

        void Start()
        {
            _bikeController =
                gameObject.
                    AddComponent<BikeController>();
        }

        void OnGUI()
        {
```

```
                if (GUILayout.Button("PowerUp Shield"))
                    _bikeController.Accept(shieldPowerUp);

                if (GUILayout.Button("PowerUp Engine"))
                    _bikeController.Accept(enginePowerUp);

                if (GUILayout.Button("PowerUp Weapon"))
                    _bikeController.Accept(weaponPowerUp);
            }
        }
    }
```

5. Go to the **Assets/Create** menu option and create three `PowerUp` assets.
6. Configure and name the new `PowerUp` assets with your desired parameters.
7. Name the new `PowerUp` assets and adjust their parameters in the Inspector.
8. Add the new `PowerUp` assets to the public properties of the `ClientVisitor` component.

Once you start the scene, you should see the following GUI buttons and debug output on your screen:

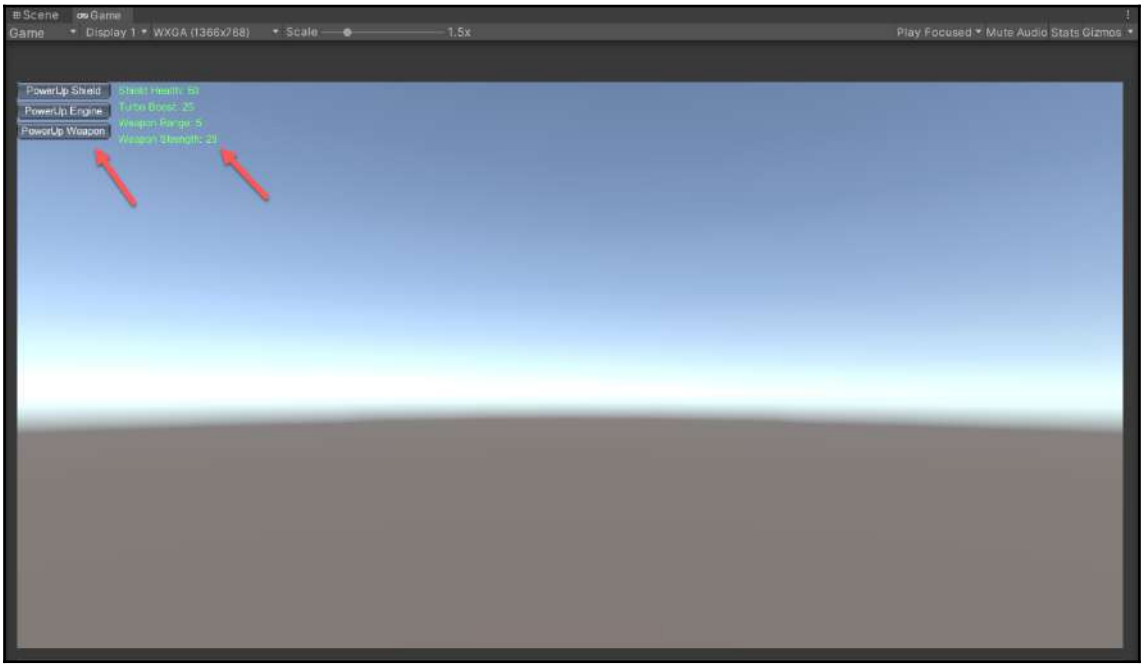


Figure 10.2 - Screenshot of the code example in action

But you might be asking yourself, how do I create an actual pickup entity that I can spawn on the race track?

A quick way to do this is simply to create a `Pickup` class like the following:

```
using System;
using UnityEngine;

public class Pickup : MonoBehaviour
{
    public PowerUp powerup;

    private void OnTriggerEnter(Collider other)
    {
        if (other.GetComponent<BikeController>())
        {
            other.GetComponent<BikeController>().Accept(powerup);
            Destroy(gameObject);
        }
    }
}
```

By attaching this script to a GameObject with a collider component configured as a trigger, we can detect when an entity with the `BikeController` component enters the trigger. Then we just need to call its `Accept()` method and pass the `PowerUp` instance. Setting up triggers is beyond the scope of this chapter, but I recommend looking at the FPP project in the Git repo to review how we set it up in a playable prototype of the game.

Reviewing the power-up system implementation

We were able to combine the structure of the Visitor pattern and the API features of ScriptableObjects to create a power-up mechanic that permits anyone on our project to author and configure new power-ups without writing a single line of code.

If we need to adjust how power-ups affect various components of our vehicle, we can do so by modifying a single class. So, in conclusion, we achieved a degree of scalability while keeping our code easily maintainable.

The implementations of the standard software design patterns in this book are experimental and adapted in creative ways. We are adapting them to utilize Unity API features and adjusting them for game development use cases. So we should not consider the examples to be academic or standardized references, just interpretations.

Summary

In this chapter of the book, we built a power-up mechanic for our game using the Visitor pattern as our foundation. We also established a workflow to create and configure power-ups. Therefore, we combined the technical and creative mindsets, which is the core of game development.

In the next chapter, we are going to design and implement attack maneuvers for enemy drones. We are going to use the Strategy pattern as the foundation of our system.

11
Implementing a Drone with the Strategy Pattern

In this chapter, we are going to implement enemy drones that fly around the race track and attack the player by shooting laser beams. They are little annoying robotic pests that will test the player's reflexes. Our drones will have a single attack that consists of firing a continuous laser beam at a 45-degree angle. To create the illusion of autonomous intelligence, the drones can be assigned three distinct attack maneuvers at runtime. Each maneuver is a repetitive series of predictable movements. Individually, the drone's behavior may look robotic, but when they are placed in a formation at specific positions on the race track, it could look like they are forming a strategy to outmaneuver the player.

And so, I'm proposing that we use the Strategy pattern to implement the various drone behaviors. The main reason for this choice is that this pattern allows us to assign specific behavior to an object at runtime. But first, let's break down the specifications of the pattern and the design intentions of our enemy drone.

This chapter includes simplified skeleton code examples for the sake of brevity and clarity. If you wish to review a complete implementation of the pattern in the context of an actual game project, open the `FPP` folder in the GitHub project. You can find the link in the *Technical requirements* section.

In this chapter, we will cover the following topics:

- An overview of the Strategy pattern
- Implementing enemy drone attack behaviors

Technical requirements

This chapter is hands-on, so you will need to have a basic understanding of Unity and C#.

The code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter11`.

Check out the following video to see the code in action: `https://bit.ly/2TdeoL4`.

Understanding the Strategy pattern

The primary goal of the Strategy pattern is to defer the decision of which behavior to use at runtime. This is made possible because the Strategy pattern lets us define a family of behaviors that are encapsulated in individual classes that we call strategies. Each strategy is interchangeable and can be assigned to a target context object to change its behavior.

Let's visualize the key elements of the pattern with this UML diagram:

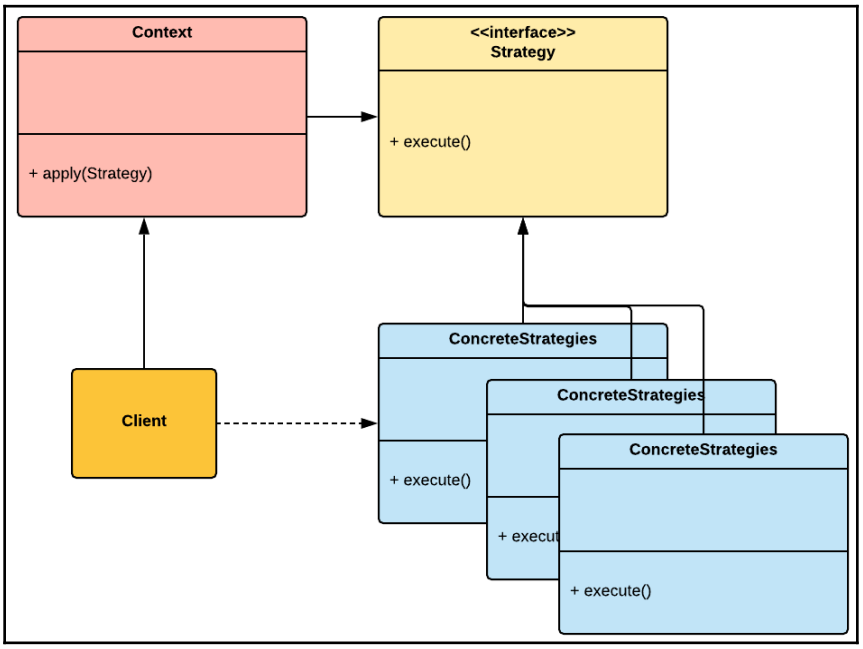


Figure 11.1 – UML diagram of the Strategy pattern

Here's a breakdown of the key players of the pattern:

- **Context** is the class that uses the various concrete strategy classes and interacts with them through the Strategy interface.
- The **Strategy** interface is common to all concrete strategy classes. It exposes a method that the `Context` class can use to execute a strategy.
- **Concrete Strategy** classes, also known as **strategies**, are concrete implementations of variants of algorithms/behaviors that can be applied to a `Context` object at runtime.

At the moment, these concepts might sound very abstract, but in practice, they are quite easy to understand, as we are going to see later in this book.

The Strategy pattern is a behavioral software design pattern; its closest cousin is the State pattern. We can use both to encapsulate a set of behaviors in individual classes. You should use the Strategy pattern when you want to select a behavior at runtime and apply it to an object. You can also use the State pattern when you want an object to change its behavior when its internal state changes.

Benefits and drawbacks of the Strategy pattern

These are some of the benefits of the Strategy pattern:

- **Encapsulation**: A clear benefit of this pattern is that it enforces variations of algorithms to be encapsulated in individual classes. Hence, this helps us avoid using long conditional statements while keeping our code structured.
- **Runtime**: The main benefit of this pattern is that it implements a mechanism that permits us to swap algorithms that an object is using at runtime. This approach makes our objects more dynamic and open for extension.

The following are some potential drawbacks of the Strategy pattern:

- **Client**: The client class must be aware of the individual strategies and the variations in the algorithm they implement so as to know which one to select. Therefore, the client becomes responsible for making sure that an object is behaving as expected during its lifespan.

- **Confusion**: Because the Strategy and State patterns are so similar in structure but have different intents, confusion could arise when you're choosing which one to use and in what context. In most cases, it's not an issue, but if you are working with a team of programmers, depending on the various levels of knowledge of the subject matter, some colleagues might not understand your choice of pattern.

I believe it's essential to have regular open discussions about architecture, patterns, and best practices with your colleagues. If you can agree as a team on common approaches when using a specific set of design patterns, you will end up with a more consistent overall architecture and cleaner code.

When to use the Strategy pattern

When I get tasked with implementing behaviors for an enemy character, the first options I consider are the State pattern or a **finite state machine** (**FSM**) since most of the time, characters are stateful.

But sometimes, I might use the Strategy pattern if the following conditions are met:

- I have an entity with several variants of the same behavior, and I want to encapsulate them in individual classes.
- I want to assign specific behavior variants to an entity at runtime, without the need to take its current internal state into consideration.
- I need to apply a behavior to an entity so that it can accomplish a specific task based on selection criteria that are defined at runtime.

The third point is probably the main reason I chose to use the Strategy pattern over the State pattern to implement the enemy drone presented in this chapter. The behavior of the drone is robotic; it has a singular task: attack the player. It doesn't make any alterations to its actions based on internal state changes. It only needs to be assigned an attack behavior at runtime to accomplish its task of attacking the player, which makes it the right candidate for the Strategy pattern in its current design.

It's important to note that potential use cases of the Strategy pattern are not limited to implementing enemy characters. For example, we could use it to encapsulate various encryption algorithms to apply to a saved file, depending on the target platform. Or, if we are working on a fantasy game, we could use it to encapsulate the individual behaviors of a family of spells that players can apply to a target entity. Therefore, the potential use cases for this pattern are broad and can be applied to various contexts, ranging from core systems to gameplay mechanics.

In the next section, we are going to review the design intentions of our enemy drone.

Designing an enemy drone

The enemy drones in our game are not very smart; there is no artificial intelligence running behind the scene. These are robots with robotic behaviors, and it's common in video games to have enemies with predictable automated behaviors running in a loop. For instance, the Goombas in the original Super Mario Bros just walk in one direction; they are not aware of the presence of Mario or react to him. They are simply running an algorithm to make them wander in a path until they collide with an obstacle. Alone, they are not a threat, but if they are put in a formation or positioned at a point in the map in which navigation is difficult, they can become challenging to avoid.

We will be using the same approach for our enemy drones. Individually, they are easy to defeat because they can't change their behaviors based on the player's movements, but in a squad, they can be challenging to avoid.

Our drone has three distinct attack maneuvers; each revolves around a specific set of movements that are predictable but still challenging to counter when the drones are in a squad formation.

Let's look at each maneuver:

- **Bobbing Maneuver**: In a bobbing maneuver, the drone moves up and down at high speed while shooting a laser beam.
- **Weaving Maneuver**: For the weaving maneuver, the drone moves horizontally at high speed, while shooting. The weaving maneuver is limited to the distance between the two rails of a track.
- **Fallback Maneuver**: For the fallback maneuver, the drone moves backward while shooting. The top speed of the drone can match that of the player's bike, but can only move back for a limited amount of time.

The following diagram illustrates the preceding maneuvers:

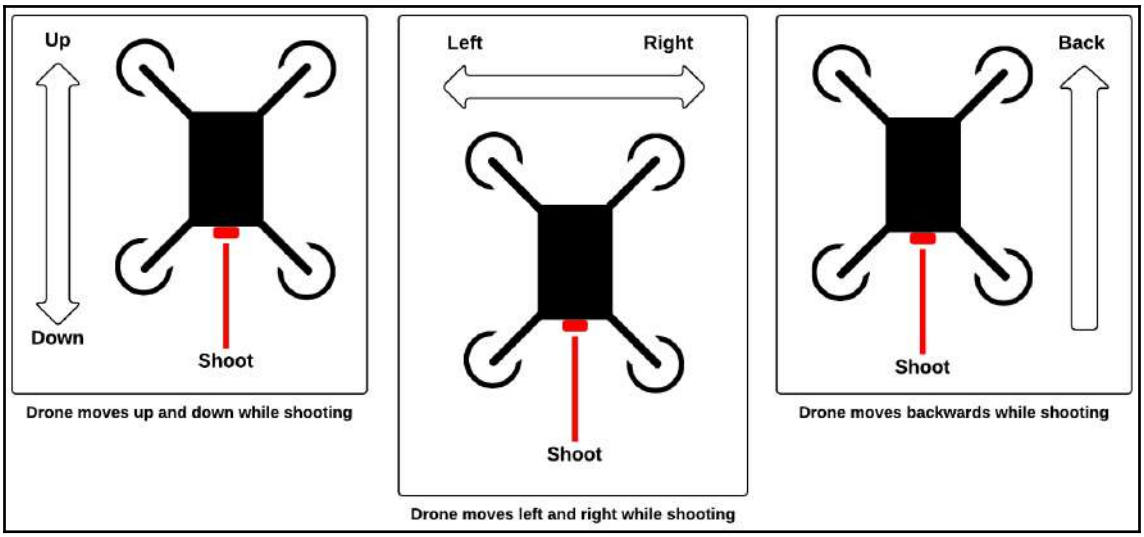


Figure 11.2 – Illustration of the drone's attack maneuvers

The enemy drone has a single weapon: a front-facing laser beam that's fired at a 45-degree angle toward the ground. The following diagram illustrates the laser weapon of the drone:

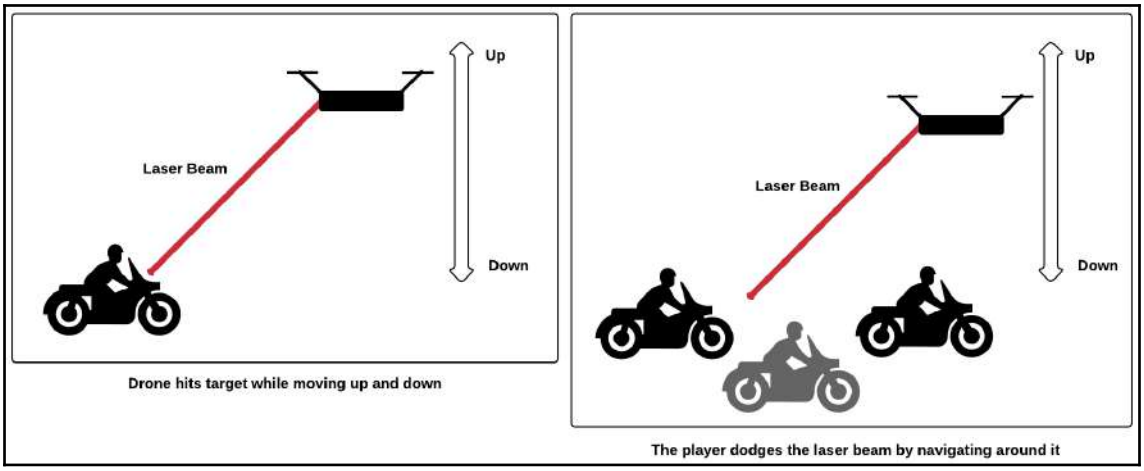


Figure 11.3 – Illustration of the drone's weapon attack

As we can see, the player must avoid the attack by navigating around the drone at high speed. If hit by the beam, the bike's front-facing shield will lose a certain amount of power. If the shield becomes depleted, the vehicle will explode on the next hit, and it will be game over.

In the next section, we are going to translate this design into code.

Implementing an enemy drone

In this section, we will write a skeleton implementation of the Strategy pattern and the individual attack behaviors of the drone enemy. The code in this section may seem oversimplified in certain aspects. Still, the end goal is not to have a complete implementation of the enemy drone but to understand the basics of the Strategy pattern.

Steps to implementing an enemy drone

Let's start by implementing the main ingredients of the Strategy pattern:

1. Our first element is the Strategy interface; all our concrete strategies will use it:

```
namespace Chapter.Strategy
{
    public interface IManeuverBehaviour
    {
        void Maneuver(Drone drone);
    }
}
```

   Note that we are passing a parameter of the `Drone` type to the `Maneuver()` method. This is an important detail we will review later.

2. Next up is our `Drone` class; it's going to use our concrete strategies, so in the overall structure of the Strategy pattern, we will consider it to be our `Context` class:

```
using UnityEngine;

namespace Chapter.Strategy {
    public class Drone : MonoBehaviour {

        // Ray parameters
        private RaycastHit _hit;
        private Vector3 _rayDirection;
```

```
        private float _rayAngle = -45.0f;
        private float _rayDistance = 15.0f;

        // Movement parameters
        public float speed = 1.0f;
        public float maxHeight = 5.0f;
        public float weavingDistance = 1.5f;
        public float fallbackDistance = 20.0f;

        void Start() {
            _rayDirection =
                transform.TransformDirection(Vector3.back)
                * _rayDistance;

            _rayDirection =
                Quaternion.Euler(_rayAngle, 0.0f, 0f)
                * _rayDirection;
        }

        public void ApplyStrategy(IManeuverBehaviour strategy) {
            strategy.Maneuver(this);
        }

        void Update() {
            Debug.DrawRay(transform.position,
                _rayDirection, Color.blue);

            if (Physics.Raycast(
                transform.position,
                _rayDirection, out _hit, _rayDistance)) {

                if (_hit.collider) {
                    Debug.DrawRay(
                        transform.position,
                        _rayDirection, Color.green);
                }
            }
        }
    }
}
```

Most of the lines of code in this class are for raycasting debugging information; we can safely ignore them. However, the following section is essential to understand:

```
public void ApplyStrategy(IManeuverBehaviour strategy)
{
    strategy.Maneuver(this);
}
```

The `ApplyStrategy()` method contains the core mechanism of the Strategy pattern. If we look closely, we can see that the method in question accepts a concrete strategy of the `IManeuverBehaviour` type as a parameter. And this is where things get very interesting. A `Drone` object can communicate with the concrete strategies it received through the `IManeuverBehaviour` interface. Thus, it only needs to call `Maneuver()` to execute a strategy at runtime. Hence, a `Drone` object doesn't need to know how a strategy's behavior/algorithm is executed – it just needs to be aware of its interface.

Now, let's implement the concrete strategy classes:

1. The following class implements the bopping maneuver:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Strategy {
    public class BoppingManeuver :
        MonoBehaviour, IManeuverBehaviour {

        public void Maneuver(Drone drone) {
            StartCoroutine(Bopple(drone));
        }

        IEnumerator Bopple(Drone drone)
        {
            float time;
            bool isReverse = false;
            float speed = drone.speed;
            Vector3 startPosition = drone.transform.position;
            Vector3 endPosition = startPosition;
            endPosition.y = drone.maxHeight;

            while (true) {
                time = 0;
                Vector3 start = drone.transform.position;
                Vector3 end =
```

```
                        (isReverse) ? startPosition : endPosition;

                    while (time < speed) {
                        drone.transform.position =
                            Vector3.Lerp(start, end, time / speed);
                        time += Time.deltaTime;
                        yield return null;
                    }

                    yield return new WaitForSeconds(1);
                    isReverse = !isReverse;
                }
            }
        }
    }
```

2. The following class implements the weaving maneuver:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Strategy {
    public class WeavingManeuver :
        MonoBehaviour, IManeuverBehaviour {

        public void Maneuver(Drone drone) {
            StartCoroutine(Weave(drone));
        }

        IEnumerator Weave(Drone drone) {
            float time;
            bool isReverse = false;
            float speed = drone.speed;
            Vector3 startPosition = drone.transform.position;
            Vector3 endPosition = startPosition;
            endPosition.x = drone.weavingDistance;

            while (true) {
                time = 0;
                Vector3 start = drone.transform.position;
                Vector3 end =
                    (isReverse) ? startPosition : endPosition;

                while (time < speed) {
                    drone.transform.position =
                        Vector3.Lerp(start, end, time / speed);
                    time += Time.deltaTime;
                    yield return null;
```

```
                }

                yield return new WaitForSeconds(1);
                isReverse = !isReverse;
            }
        }
    }
}
```

3. Finally, let's implement the fallback maneuver:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Strategy
{
    public class FallbackManeuver :
        MonoBehaviour, IManeuverBehaviour {

        public void Maneuver(Drone drone) {
            StartCoroutine(Fallback(drone));
        }

        IEnumerator Fallback(Drone drone)
        {
            float time = 0;
            float speed = drone.speed;
            Vector3 startPosition = drone.transform.position;
            Vector3 endPosition = startPosition;
            endPosition.z = drone.fallbackDistance;

            while (time < speed)
            {
                drone.transform.position =
                    Vector3.Lerp(
                        startPosition, endPosition, time / speed);

                time += Time.deltaTime;

                yield return null;
            }
        }
    }
}
```

You may have noticed that the code of each class is quite similar, even repetitive in certain parts. This accounts for one of the reasons we are using the Strategy pattern – we want to encapsulate variations of similar behaviors so that they are easier to maintain individually. But also, imagine how messy our `Drone` class would be if we tried to implement the bopping, weaving, and fallback behaviors in a single class. We would find ourselves in a bloated `Drone` class that's potentially filled to the brim with conditional statements.

I would not recommend using coroutines to animate non-humanoid entities. Instead, I suggest using a Tween engine such as DOTween since you can animate objects with less code while getting better results. We are using coroutines in this chapter to avoid external dependencies and make our code easy to port. To learn more about DOTween, go to `http://dotween.demigiant.com`.

Testing the enemy drone implementation

And now for the fun part – testing our implementation. It's going to be an easy one since all we need to do is attach the following client class to an empty `GameObject` in a Unity scene:

```
using UnityEngine;
using System.Collections.Generic;

namespace Chapter.Strategy {
    public class ClientStrategy : MonoBehaviour {

        private GameObject _drone;

        private List<IManeuverBehaviour>
            _components = new List<IManeuverBehaviour>();

        private void SpawnDrone() {
            _drone =
                GameObject.CreatePrimitive(PrimitiveType.Cube);

            _drone.AddComponent<Drone>();

            _drone.transform.position =
                Random.insideUnitSphere * 10;

            ApplyRandomStrategies();
        }

        private void ApplyRandomStrategies() {
            _components.Add(
```

```
                _drone.AddComponent<WeavingManeuver>());
            _components.Add(
                _drone.AddComponent<BoppingManeuver>());
            _components.Add(
                _drone.AddComponent<FallbackManeuver>());

            int index = Random.Range(0, _components.Count);

            _drone.GetComponent<Drone>().
                ApplyStrategy(_components[index]);
        }

        void OnGUI() {
            if (GUILayout.Button("Spawn Drone")) {
                SpawnDrone();
            }
        }
    }
}
```

In your instance of Unity, if you included all the scripts we wrote in the preceding sections in your project, you should see a single button on the screen when you start it called **Spawn Drone**, as shown in the following screenshot:

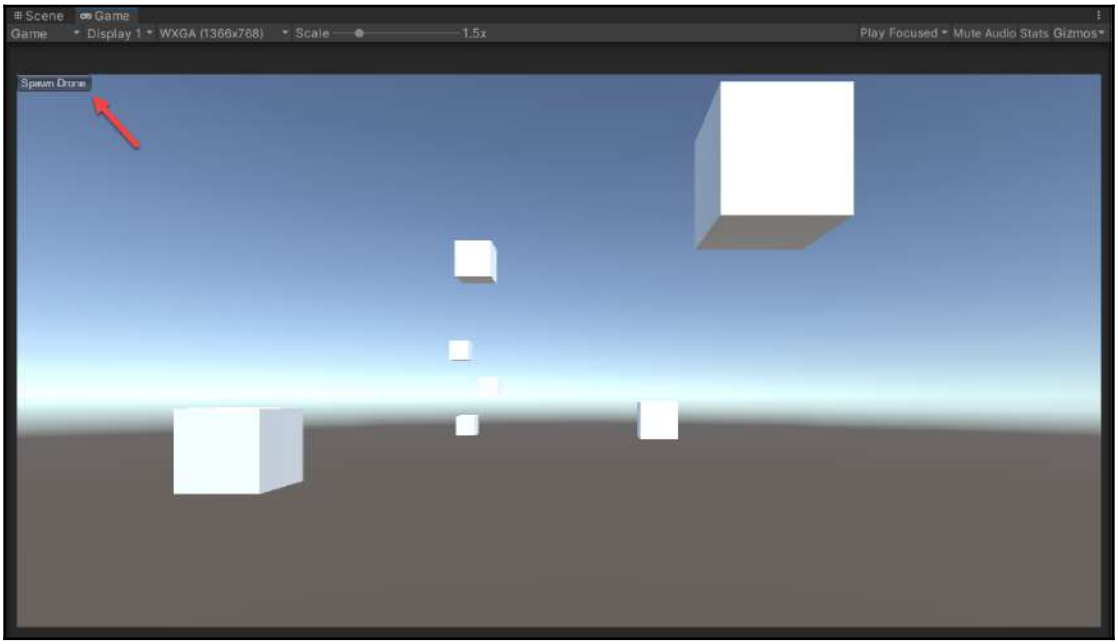


Figure 11.4 – The code example in action inside Unity

If you click on the scene's primary button, a new cube representing a Drone entity should appear at a random position while executing a randomly selected attack maneuver.

Reviewing the enemy drone implementation

In the preceding code example, the client class acts like a spawner that randomly assigns a strategy to a new drone instance. This could be an interesting approach for a real-world use case. But there are many other approaches we could have used to choose which strategy to assign to a drone. It could be based on specific rules and factors that are only known at runtime. Therefore, it's not limited to randomness but can also be deterministic and rule-based.

Reviewing alternative solutions

There's one glaring issue with the code examples presented in this chapter. We encapsulated attack maneuver behaviors into distinct strategy classes, but each maneuver is nothing more than a single animation running on a loop. So, in an actual game project that's been built by a production team that includes animators, I would not have animated the enemy drones in code by using coroutines or even a Tween animation engine. Instead, I would ask an animator to author some detailed attack maneuver animations in an external authoring tool and then import them into Unity as animation clips. I would then have used Unity's native animation system and its state machine feature to assign attack maneuver animations to a drone dynamically.

Using this approach, I would have gained quality in the animations and the flexibility of transitioning smoothly from one attack behavior to another, if I decide that the drones can switch attacks when an internal state changes. Therefore, I would have moved away from the idea of encapsulating each attack behavior into a strategy class and instead defined them as finite states. This switch would not be a dramatic change in design as the concepts that drive the FSM, State, and Strategy patterns are closely related.

Even though the Strategy pattern's implementation in the example presented in this chapter is valid, it's wise to consider what can be achieved natively with Unity's animation system first when you're managing an entity's animation set. But imagine another use case in which we need to implement variations of a motion detection algorithm and assign them to a drone at runtime. In that context, the Strategy pattern would be an excellent choice to build that system.

You can read the official documentation on Unity's native animation system at `https://docs.unity3d.com/2021.2/Documentation/Manual/AnimationOverview.html`.

Summary

In this chapter, we used the Strategy pattern to implement our game's first enemy ingredient, a flying, laser-shooting drone. By using this pattern, we encapsulated each variant of the drone's attack maneuvers in individual classes. This approach made our code easier to maintain by avoiding having bloated classes filled with lengthy conditional statements. Now, we can quickly write new attack maneuver variations or adjust existing ones. Hence, we have given ourselves the flexibility to be creative and test out new ideas quickly, which is a vital part of game development.

In the next chapter, we will start working on a weapon system and explore the Decorator pattern.

12
Using the Decorator to Implement a Weapon System

In this chapter, we are going to build a customizable weapon system. Throughout the game, the player will be able to upgrade its bike's primary weapon by purchasing attachments that will augment specific properties, such as range and strength. The primary weapon is mounted on the front of the bike and has two expansion slots for attachments that the player can use to build various combinations. To build this system, we are going to use the Decorator pattern. It should not be a surprise because its name implies its use, as we will see further in the chapter.

The following topics will be covered in this chapter:

- The basic principles behind the Decorator pattern
- The implementation of a weapon system with attachments

Technical requirements

You will need to have a basic understanding of Unity and C#.

We will be using the following Unity engine and C# language concepts:

- Constructors
- ScriptableObjects

If you are unfamiliar with these concepts, please review `Chapter 3`, *A Short Primer to Programming in Unity*.

The code files for this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter12`.

Check out the following video to see the code in action: `https://bit.ly/3r9rvJD`.

We often use **ScriptableObjects** in the code examples in this book because we want to establish an authoring workflow for our designers to create new weapon attachments or configure existing ones without modifying a single line of code. It's good practice to make your systems, ingredients, and mechanics easy to configure for non-programmers.

Understanding the Decorator pattern

In short, the Decorator is a pattern that permits the addition of new functionalities to an existing object without altering it. And this is made possible by creating a decorator class that wraps the original class. And with this mechanism, we easily attach but also detach new behaviors to an object.

Let's review the following diagram to visualize the Decorator's structure before diving deeply into the subject matter:

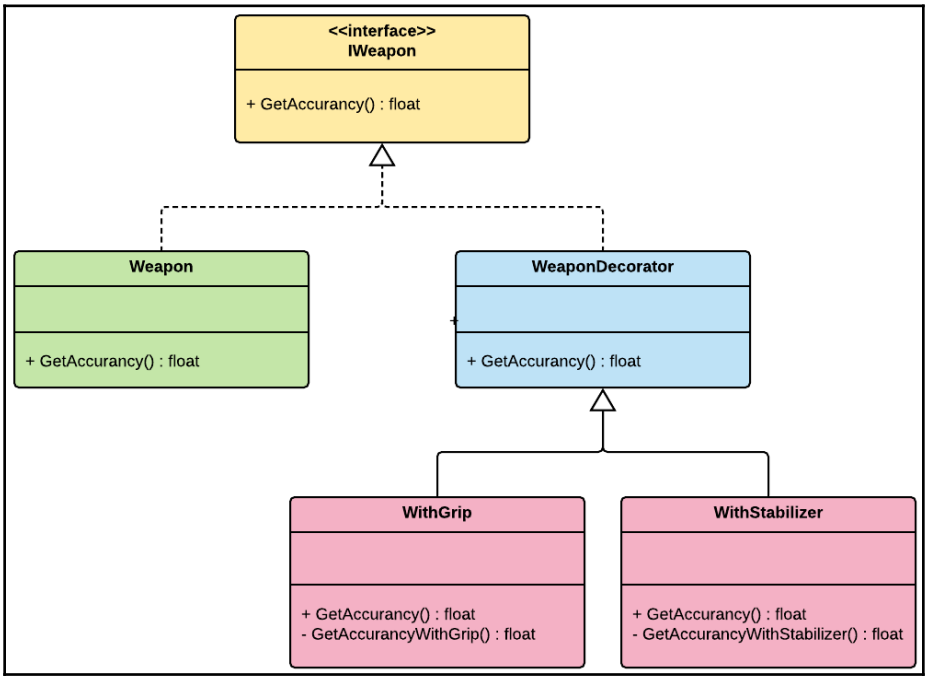


Figure 12.1 – UML diagram of the Decorator pattern

The `IWeapon` interface establishes an implementation contract that will maintain a consistent method signature between the decorated object and its decorators. `WeaponDecorator` wraps the target object, and the concrete decorator classes (`WithGrip` and `WithStabilizer`) decorate it by enhancing or overriding its behaviors.

The method signatures and the overall structure of the *decorated* object are not modified during the process, just its behaviors or property values. Thus, we can easily remove decorations from an object and return it to its initial form.

Most textbook examples of this pattern are heavily dependent on the class constructor. However, native Unity API base classes, such as `MonoBehaviour` and `ScriptableObject`, don't use the concept of a constructor to initialize an instance of an object. Instead, in the case of MonoBehaviours, the engine takes care of initializing the classes that are attached to GameObjects. And any initialization code is expected to be implemented in the `Awake()` or `Start()` callbacks.

Therefore, we are challenged to find a way to adapt the Decorator pattern so that it will use core Unity API features while not losing its primary benefits.

Benefits and drawbacks of the Decorator pattern

The following are some of the benefits of the Decorator pattern:

- **Alternative to subclassing**: Inheritance is a static process. Unlike the Decorator pattern, it doesn't permit extending an existing object's behavior at runtime. You can only replace an instance with another with the same parent class that has the desired behavior. Therefore, the Decorator pattern is a more dynamic alternative to subclassing and overcomes the limits of inheritance.
- **Runtime dynamics**: The Decorator pattern permits us to add functionality to an object at runtime by attaching decorators to it. But this also means that the reverse is possible, and you can restore an object back to its original form by removing its decorators.

The following are some of the potential drawbacks of the Decorator pattern:

- **Relationship complexity**: Keeping track of the chain of initialization and the relationships between decorators can become very complicated if there are multiple layers of decorators around an object.

- **Code complexity**: Depending on how you implement the Decorator pattern, it can add complexity to your code base as you might need to maintain several little decorator classes. But when and how this becomes an actual drawback is very contextual and not a constant. In the upcoming code example that we will review, this is not an issue as each decorator is a `ScriptableObject` instance that we save as a configurable asset.

The Decorator pattern is a part of the structural design pattern family; some of its members include the Adapter, Bridge, Façade, and Flyweight patterns.

When to use the Decorator pattern

In this chapter, we are implementing a weapon system with attachments. We have a couple of specifications to consider, such as the following:

- We need to be able to attach multiple attachments to a weapon.
- We need to be able to add and remove them at runtime.

The Decorator pattern offers us a way to fulfill these two core requirements. Therefore, it's a pattern to consider when implementing a system in which we need to support the ability to add and remove behaviors to an individual object in a dynamic manner.

For instance, if we are assigned to work on a **CCG** (**collectible card game**), we might have to implement a mechanism in which the player can augment a base card's powers with artifact cards stacked on top of each other. Another use case is imagining having to implement a wardrobe system in which players can decorate their armor with accessories to buff specific stats.

In both cases, using the Decorator pattern could be a good starting point to build these types of mechanics and systems. Now that we have a basic understanding of the Decorator pattern, we will review the design of our weapon attachment system before writing code.

Designing a weapon system

By default, every bike model in our game comes equipped with a front-mounted gun that fires laser beams. The player can use this weapon to destroy obstacles that might be in the way, and very skilled players can shoot at flying drones while doing wheelies. The player can also purchase various attachments that boost the basic stats of the bike's weapon, as illustrated in the following diagram:

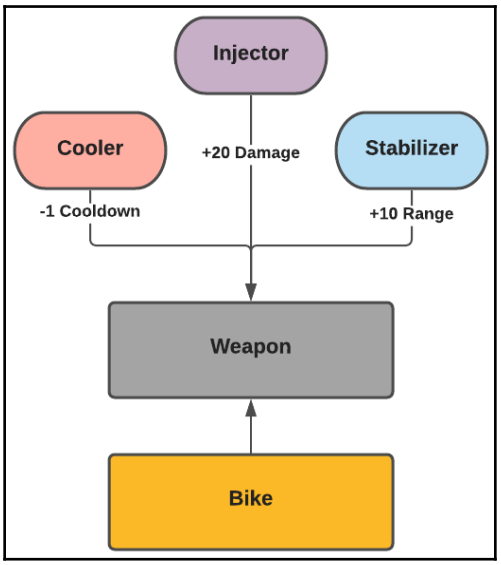


Figure 12.2 – Diagram of the weapon attachment system

Let's review a shortlist of potential attachments that the player could buy:

- **Injector**: A plasma injector that amplifies the weapon's damage capacity.
- **Stabilizer**: An attachment that reduces vibrations caused when the bike hits its top speed. It stabilizes the weapon's shooting mechanism and expands its range.
- **Cooler**: A water-cooling system that attaches itself to the weapon's firing mechanism. It enhances the rate of fire and reduces the cool-down duration.

These examples are high-level concepts of potential attachment variations. Each of these attachments modifies or, in other words, "decorates" specific properties of the weapon.

Another requirement to keep in mind while implementing our system is that our weapon will have a maximum of two expansion slots for attachments. The player must be able to add and remove them dynamically when adjusting the settings on their vehicle.

Now that we have a good overview of the requirements of the design of our system, it's time to translate this into code, which we will do in the next section.

Implementing a weapon system

In this section, we are going to implement the core classes of our weapon system. However, please note that we are not going to write the weapon's core behaviors for reasons of brevity. We want to keep the focus on understanding how the Decorator pattern works. But if you wish to review a more advanced version of this code example, check out the `FPP` folder in this book's GitHub repo. The link can be found in the *Technical requirements* section.

Implementing the weapon system

Many steps need to be performed as we have a lot of code to review together:

1. To start, we are going to implement the `BikeWeapon` class. Because it's quite long, we will split it up into three segments:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Decorator
{
    public class BikeWeapon : MonoBehaviour
    {
        public WeaponConfig weaponConfig;
        public WeaponAttachment mainAttachment;
        public WeaponAttachment secondaryAttachment;

        private bool _isFiring;
        private IWeapon _weapon;
        private bool _isDecorated;

        void Start()
        {
            _weapon = new Weapon(weaponConfig);
        }
```

The first segment is the initialization code. Note that we are setting the weapon's configuration at the `Start()`.

For the second segment of the class, it is just `GUI` labels that will help us with debugging:

```
void OnGUI()
{
    GUI.color = Color.green;

    GUI.Label (
        new Rect (5, 50, 150, 100),
        "Range: "+ _weapon.Range);

    GUI.Label (
        new Rect (5, 70, 150, 100),
        "Strength: "+ _weapon.Strength);

    GUI.Label (
        new Rect (5, 90, 150, 100),
        "Cooldown: "+ _weapon.Cooldown);

    GUI.Label (
        new Rect (5, 110, 150, 100),
        "Firing Rate: " + _weapon.Rate);

    GUI.Label (
        new Rect (5, 130, 150, 100),
        "Weapon Firing: " + _isFiring);

    if (mainAttachment && _isDecorated)
        GUI.Label (
            new Rect (5, 150, 150, 100),
            "Main Attachment: " + mainAttachment.name);

    if (secondaryAttachment && _isDecorated)
        GUI.Label (
            new Rect (5, 170, 200, 100),
            "Secondary Attachment: " + secondaryAttachment.name);
}
```

However, it is in the last segment that things start to get interesting:

```
        public void ToggleFire() {
            _isFiring = !_isFiring;

            if (_isFiring)
                StartCoroutine(FireWeapon());
        }

        IEnumerator FireWeapon() {
```

```
            float firingRate = 1.0f / _weapon.Rate;

            while (_isFiring) {
                yield return new WaitForSeconds(firingRate);
                Debug.Log("fire");
            }
        }

        public void Reset() {
            _weapon = new Weapon(weaponConfig);
            _isDecorated = !_isDecorated;
        }

        public void Decorate() {
            if (mainAttachment && !secondaryAttachment)
                _weapon =
                    new WeaponDecorator(_weapon, mainAttachment);

            if (mainAttachment && secondaryAttachment)
                _weapon =
                    new WeaponDecorator(
                        new WeaponDecorator(
                            _weapon, mainAttachment),
                                secondaryAttachment);

            _isDecorated = !_isDecorated;
        }
    }
}
```

The `Reset()` method resets the weapon to its initial configurations by initializing a new `Weapon` with its default weapon configurations. This is a quick and simple way to remove the attachments that we set in the `Decorate()` method. It's not always the best approach, but it works in the context of this example.

However, `Decorate()` is where the magic happens, and the Decorator pattern mechanism is triggered. You might notice that we can stack attachments over one another by chaining them in the constructor, as we see here:

```
if (mainAttachment && secondaryAttachment)
    _weapon =
        new WeaponDecorator(
            new WeaponDecorator(
                _weapon, mainAttachment),
                    secondaryAttachment);
```

That's a little trick that the Decorator and constructors permit us to do. How it works will be made more apparent when we start implementing the other classes.

2. The next class to implement is the `Weapon` class. Notice that it's not a `MonoBehaviour`. Therefore, we will initialize it with its constructor:

```
namespace Chapter.Decorator
{
    public class Weapon : IWeapon
    {
        public float Range
        {
            get { return _config.Range; }
        }

        public float Rate
        {
            get { return _config.Rate; }
        }

        public float Strength
        {
            get { return _config.Strength; }
        }

        public float Cooldown
        {
            get { return _config.Cooldown; }
        }

        private readonly WeaponConfig _config;

        public Weapon(WeaponConfig weaponConfig)
        {
            _config = weaponConfig;
        }
    }
}
```

Unlike `BikeWeapon`, the `Weapon` class doesn't implement any behaviors; it's just a representation of the configurable properties of a weapon. This is the object we will decorate with attachments. In its constructor, we are passing an instance of a `WeaponConfig` object. As we are going to see further on, this is a `ScriptableObject`.

3. We need a common interface between the decorator class and the weapon, so we are going to implement one named `IWeapon`:

```
namespace Chapter.Decorator
{
    public interface IWeapon
    {
        float Range { get; }
        float Duration { get; }
        float Strength { get; }
        float Cooldown { get; }
    }
}
```

4. Now that we have a standard interface that defines a set of properties that we can decorate, we can implement a `WeaponDecorator` class:

```
namespace Chapter.Decorator
{
    public class WeaponDecorator : IWeapon
    {
        private readonly IWeapon _decoratedWeapon;
        private readonly WeaponAttachment _attachment;

        public WeaponDecorator(
            IWeapon weapon, WeaponAttachment attachment) {

            _attachment = attachment;
            _decoratedWeapon = weapon;
        }

        public float Rate {
            get { return _decoratedWeapon.Rate
                         + _attachment.Rate; }
        }

        public float Range {
            get { return _decoratedWeapon.Range
                         + _attachment.Range; }
        }

        public float Strength {
            get { return _decoratedWeapon.Strength
                         + _attachment.Strength; }
        }

        public float Cooldown
        {
```

```
            get { return _decoratedWeapon.Cooldown
                        + _attachment.Cooldown; }
        }
    }
}
```

Notice how `WeaponDecorator` is wrapping itself around an instance of an object that shares a similar interface, in this case, the `IWeapon` interface. It never modifies directly the object that's its wrapping; it's only decorating its public properties with those of a `WeaponAttachment`.

Keep in mind that in our code example, we are just modifying the values of the properties of a weapon. And this fits with our design because the attachments don't alter the weapon's core mechanism. Instead, they just enhance specific properties through an attachment slot.

One last detail to keep in mind is the fact that we are defining the weapon's behavior inside the `BikeWeapon` class and its behavior is determined by the properties set in an instance of a `Weapon` object. Therefore, by decorating the `Weapon` object, we are modifying how the bike weapon behaves.

5. The following step is where we are starting to deviate from a traditional implementation of the Decorator pattern. Instead of defining individual concrete decorator classes, we are going to implement a `ScriptableObject` named `WeaponConfig`. This approach will permit us to author separate attachments and configure their properties through the inspector. Then, we will use these attachment assets to decorate the weapon, as we can see here:

```
using UnityEngine;

namespace Chapter.Decorator
{
    [CreateAssetMenu(fileName = "NewWeaponAttachment",
        menuName = "Weapon/Attachment", order = 1)]
    public class WeaponAttachment : ScriptableObject, IWeapon
    {
        [Range(0, 50)]
        [Tooltip("Increase rate of firing per second")]
        [SerializeField] public float rate;

        [Range(0, 50)]
        [Tooltip("Increase weapon range")]
        [SerializeField] float range;

        [Range(0, 100)]
```

```
        [Tooltip("Increase weapon strength")]
        [SerializeField] public float strength;

        [Range(0, -5)]
        [Tooltip("Reduce cooldown duration")]
        [SerializeField] public float cooldown;

        public string attachmentName;
        public GameObject attachmentPrefab;
        public string attachmentDescription;

        public float Rate {
            get { return rate;  }
        }

        public float Range {
            get { return range; }
        }

        public float Strength {
            get { return strength;  }
        }

        public float Cooldown {
            get { return cooldown; }
        }
    }
}
```

It's important to note that `WeaponAttachment` implements the `IWeapon` interface. This keeps it consistent with `WeaponDecorator` and `Weapon` classes as all three of them share the same interface.

6. And for our last step, we are going to implement the `ScriptableObject` class named `WeaponConfig`. We are using to it create various configurations for our `Weapon` object that we can then configure with attachments:

```
using UnityEngine;

namespace Chapter.Decorator
{
    [CreateAssetMenu(fileName = "NewWeaponConfig",
        menuName = "Weapon/Config", order = 1)]
    public class WeaponConfig : ScriptableObject, IWeapon
    {
        [Range(0, 60)]
        [Tooltip("Rate of firing per second")]
```

```
        [SerializeField] private float rate;

        [Range(0, 50)]
        [Tooltip("Weapon range")]
        [SerializeField] private float range;

        [Range(0, 100)]
        [Tooltip("Weapon strength")]
        [SerializeField] private float strength;

        [Range(0, 5)]
        [Tooltip("Cooldown duration")]
        [SerializeField] private float cooldown;

        public string weaponName;
        public GameObject weaponPrefab;
        public string weaponDescription;

        public float Rate {
            get { return rate;  }
        }

        public float Range {
            get { return range; }
        }

        public float Strength {
            get { return strength;  }
        }

        public float Cooldown {
            get { return cooldown; }
        }
    }
}
```

We now have all the key ingredients for our weapon attachment system, and it's time to test them inside the Unity engine.

Testing the weapon system

If you wish to test the code we just reviewed in your own instance of Unity, you need to follow these steps:

1. Copy all the classes we just reviewed in your Unity project.
2. Create an empty Unity scene.

3. Add a new GameObject to the scene.
4. Attach the following client script to the new GameObject:

```
using UnityEngine;

namespace Chapter.Decorator
{
    public class ClientDecorator : MonoBehaviour
    {
        private BikeWeapon _bikeWeapon;
        private bool _isWeaponDecorated;

        void Start() {
            _bikeWeapon =
                (BikeWeapon)
                FindObjectOfType(typeof(BikeWeapon));
        }

        void OnGUI()
        {
            if (!_isWeaponDecorated)
                if (GUILayout.Button("Decorate Weapon")) {
                    _bikeWeapon.Decorate();
                    _isWeaponDecorated = !_isWeaponDecorated;
                }

            if (_isWeaponDecorated)
                if (GUILayout.Button("Reset Weapon")) {
                    _bikeWeapon.Reset();
                    _isWeaponDecorated = !_isWeaponDecorated;
                }

            if (GUILayout.Button("Toggle Fire"))
                _bikeWeapon.ToggleFire();
        }
    }
}
```

5. Add to the new GameObject the `BikeWeapon` script we implemented in the previous section.
6. Under the **Assets** | **Create** | **Weapon** menu option, create a new **Config** and configure it as you wish.
7. Under the **Assets** | **Create** | **Weapon** menu option, create several **Attachment** variations with various configurations.

8. You can now add `WeaponConfig` and `WeaponAttachment` assets to the `BikeWeapon` component properties in `Inspector`, as seen in the following screenshot:

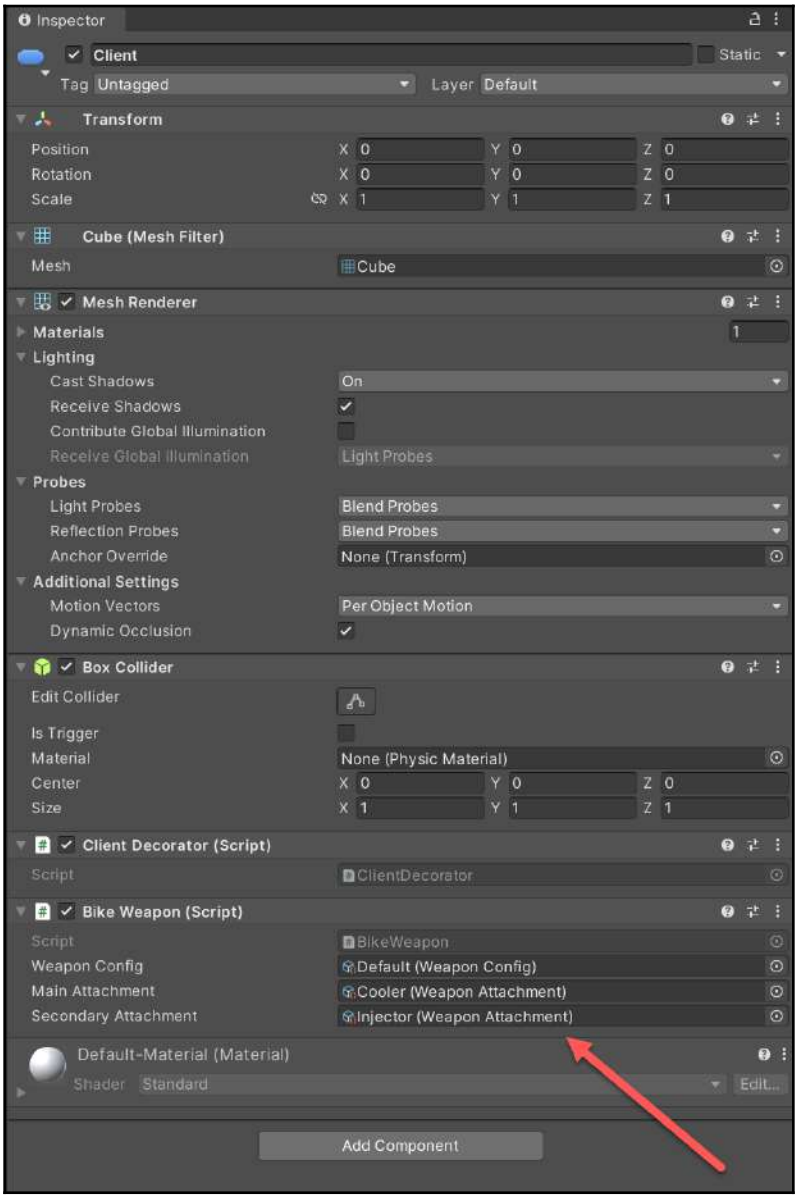


Figure 12.3 – BikeWeapon component properties

When you start your scene, you should see the following buttons in the top-left corner of the screen, as shown in the following screenshot:

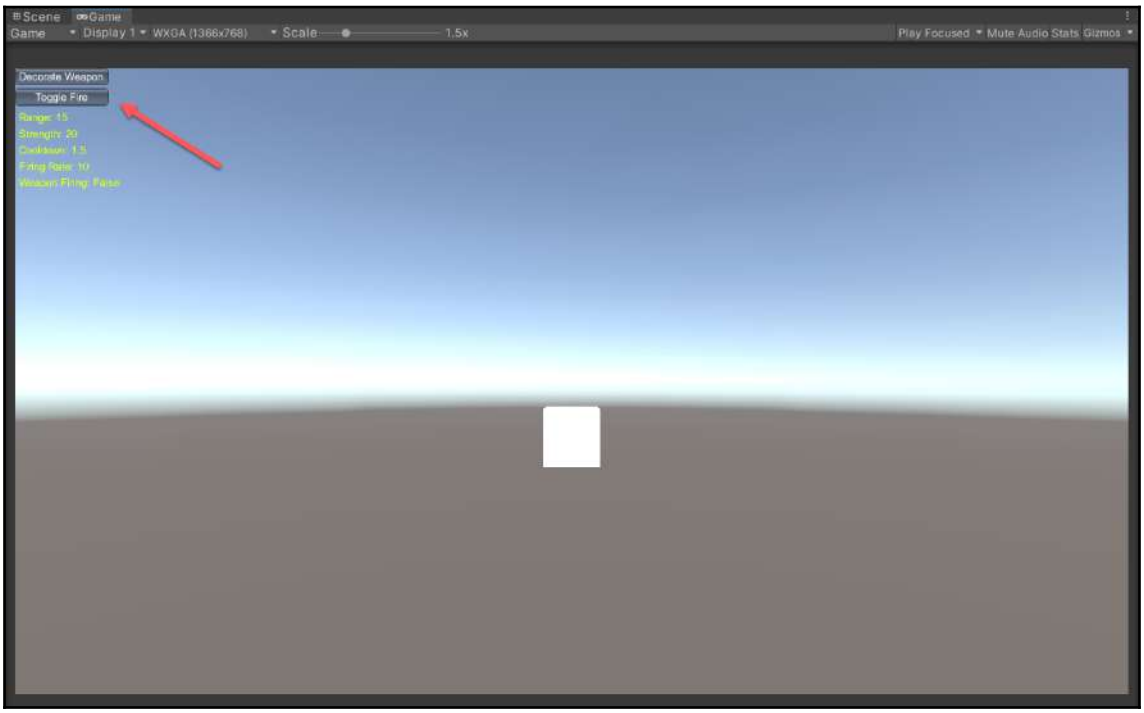


Figure 12.4 – Screenshot of the code example in action inside Unity

You now test the weapon attachment system and adjust it as you wish by pressing the buttons and look at the weapon stats change as you modify the decorator's settings.

Reviewing the weapon system

Our Decorator implementation is unorthodox because we are mixing native C# language features such as constructors while trying to utilize the best parts of the Unity API. By converting decorator classes into configurable `ScriptableObject` assets, we gain the ability to make our weapon attachments authorable and configurable by non-programmers. And under the hood, our attachment system is constructed on the foundation of a solid design pattern.

Therefore, we endeavored to strike a balance between usability and maintainability. However, as we are going to see in the next section, there are always alternative approaches.

Design patterns are guidelines, not commandments. Therefore, it's OK to experiment with patterns and test out different ways of implementing them. The code example in this chapter is experimental because it's not a traditional implementation of the Decorator pattern. But we encourage you, as the reader, to continue experimenting with the patterns showcased in this book and find new ways of using them in Unity.

Reviewing alternative solutions

In the context of the use case presented in this chapter, we could have implemented the weapon system without the Decorator pattern and with ScriptableObjects only. We could have iterated through a list of acquired weapon attachments and apply each of their properties to those of a `Weapon` class. We would lose the ability of chaining decorators, but our code will be more straightforward.

The core benefit of using the Decorator pattern in our case was that it gave us a structured and repeatable approach to implementing our system. Still, in consequence, we added additional complexity to our code base.

Summary

In this chapter, we implemented a weapon attachment system that's authorable and configurable. Non-programmers will be able to create and adjust new attachments without writing a single line of code. Therefore, we can focus on building systems while designers work on balancing them. The Decorator pattern has proven itself to be a handy pattern for game development, and so it's a pattern to keep in our programmer's toolbox.

In the next chapter, we will explore the Spatial Partition pattern—a subject matter that's quite important to understand when building games with large maps.

13
Implementing a Level Editor with Spatial Partition

In this chapter, we will explore the concept of spatial partitioning. Unlike in previous chapters, the main subject is not traditionally defined as a software design pattern but more as a process and a technique. But because it offers us a reusable and structured approach to solving recurrent game-programming problems, we will treat it as a design pattern in the context of this chapter.

The approach we are going to take in this chapter is different from previous chapters for the following specific reasons:

- We are taking a hands-off approach; in other words, we will not attempt to implement a code example but will instead review some code segments.
- We will not try to stay faithful to any academic definition but will instead use the general concept of spatial partitioning to build a level editor for our racing game.

The simplest definition of spatial partitioning is a process that offers an efficient way of locating objects by collating them in a data structure ordered by their positions. The level editor we are implementing in this chapter will be built with a stack data structure, and the type of object we will keep in a specific order on a stack is a race track segment. These individual segments of the race track will be spawned or deleted in a particular order, depending on their relation to the player's position on the map.

All of this might sound very abstract, but it's pretty easy to achieve this with the Unity **application programming interface** (**API**), as we will see in this chapter.

The system we are implementing in this chapter is too elaborate to boil down into a skeleton code example. So, unlike in previous chapters, the code presented is not meant to be reproduced or used as a template. We instead recommend reviewing a complete code example of the level editor in the `/FPP` folder of the Git project. The link can be found in the *Technical requirements* section of this chapter.

In this chapter, we will cover the following topics:

- Understanding the Spatial Partition pattern
- Designing a level editor
- Implementing a level editor
- Reviewing alternative solutions

Technical requirements

We will also be using the following specific Unity engine API features:

- Stack
- ScriptableObjects

If unfamiliar with these concepts, please review `Chapter 3`, *A Short Primer to Programming in Unity*.

The code files of this chapter can be found on GitHub at the following link: `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter13`.

A Stack is a linear data structure with two primary operations: **Push**, which adds an element on top of the stack, and **Pop**, which removes the most recent element from the top.

Understanding the Spatial Partition pattern

The Spatial Partition pattern name comes from the process known as space partitioning, which plays an integral part in computer graphics and is often used in ray-tracing rendering implementations. The process is utilized to organize objects in virtual scenes by storing them in a space-partitioning data structure such as a **binary space partitioning** (**BSP**) tree; this makes it faster to perform geometry queries on a large set of **three-dimensional** (**3D**) objects. In this chapter, we will use the general concept of spatial partitioning without being faithful to how it's usually implemented in computer graphics.

A very high-level and conceptual way of visualizing spatial partitioning is with the following diagram:

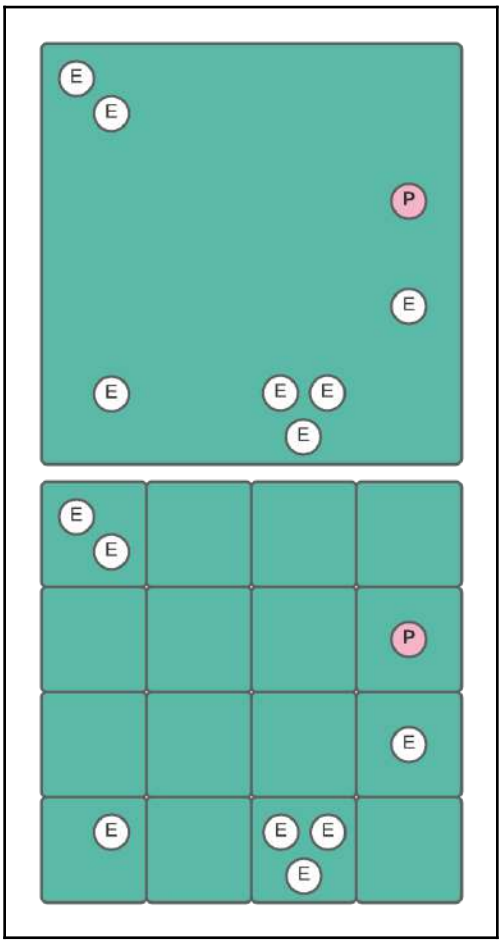


Figure 13.1 – A diagram that illustrates spatial partitioning on a map

The first example represents the position of enemies on the map without any partitioning. If we want to quickly look up the location of enemies in relation to the player, it could be challenging to do this efficiently. Of course, we could use ray casts to calculate the distance between entities, but this could become inefficient as the number of enemies grows.

The next part of the diagram shows that if we partition the map, we can now easily visualize the clusters of enemies in relation to the player's location. In code, we could now quickly look up which enemy is the closest to the player and where there's the largest cluster of them, and because we don't need the exact position of each enemy but just their general geographical relation to the player, just knowing which approximate cell of the grid they are in is good enough.

BSP is a 3D programming technique that recursively subdivides space into convex pairs using a series of hyperplanes. The method is implemented with a binary-tree data structure. John Carmack has famously used BSP to develop games such as *Doom* and *Quake*.

When to use the Spatial Partition pattern

3D programming is beyond the scope of this book, yet the most important takeaway of the description of spatial partitioning is that it offers a solution to organizing a large set of objects in a scene in an optimal manner. Therefore, if you find yourself needing a quick way to query an extensive collection of objects in a scene while keeping track of their spatial relations, keep in mind the principles of spatial partitioning.

In the next section, we will review the overall design of the level editor and examine its technical requirements.

Designing a level editor

When working on a multi-disciplinary game development team, the responsibilities of a game programmer are not limited to implementing cool game mechanics and features. We are often tasked with building asset-integration pipelines and editing tools. The most common tool we might need to implement early on in a production cycle is a custom level editor for the level designers on our team.

Before writing a single line of code, we need to keep in mind that our game has no randomness integrated into its core game systems. It's a game of skill in which players have the primary goal of reaching the top of the leaderboard by memorizing each track's intricacies and getting to the finish line as fast as possible.

Thus, based on these core design pillars, we can't use a solution such as procedural generation maps that spawn random obstacles based on specific rules and constraints. Consequently, the level designers in our team will have to design the layout of each race track by hand.

And this brings us to our most significant challenge: in our game, a bike travels through a 3D world in a straight line at very high speeds. If we wish to have a race that lasts more than a dozen seconds, we will need a massive amount of assets in the memory, and our designers will have to deal with editing massive levels in the editor.

This approach is neither efficient during the editing phase nor at runtime. So, instead of managing a single race track as one entity, we will divide it into segments, and each segment will be editable individually and then assembled in a specific order to form a single track.

The following diagram illustrates this high-level concept:

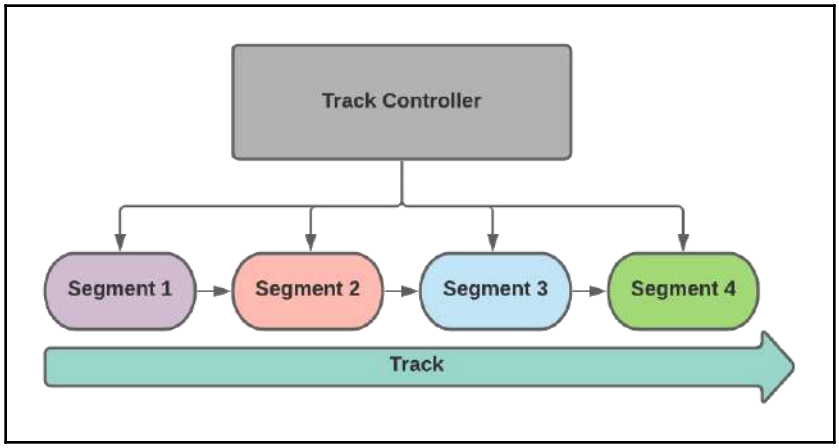


Figure 13.2 – A diagram of the sequencing of track segments

We gain two key benefits from this system, outlined as follows:

1. Our level designers can author new tracks by creating new segments and sequencing them in various layouts.
2. We don't need to load the entire content of the race track into memory, just to spawn the segments we need at the right moment in relation to the player's current position.

The next diagram illustrates how the track controller uses a stack data structure to manage which track to unload and which ones to spawn in relation to the current position of the player:

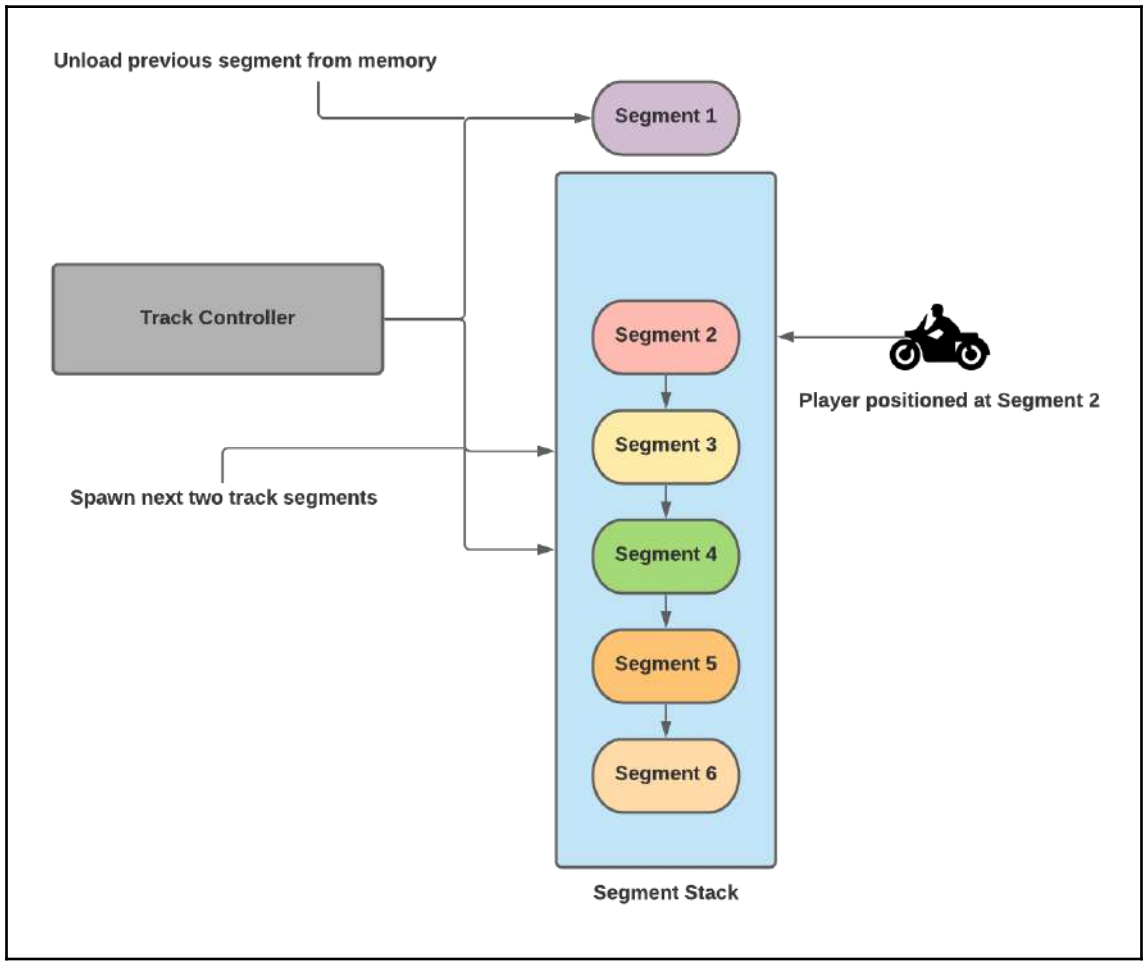


Figure 13.3 – A diagram of the segment stack

There are two distinct characteristics of our game that we must keep in mind while implementing this system, outlined as follows:

1. The bike never moves from its initial position. It's the track segments that move toward the player. Therefore, the sense of speed and movement is simulated and provides a visual illusion.

2. The player can only see forward. There are no rear-view windows or look-back cameras. This camera-view limitation means we can unload track segments immediately after they are behind the player's field of view.

In summary, with a single system, we have resolved two potential core issues for our project. Foremost, we are establishing a level design pipeline, and lastly, we have a mechanism to load our levels dynamically with a degree of optimization embedded in the design.

I was inspired by miniature electric-toy slot cars when designing the racing game we are building in this book. One unique aspect of this toy is that you can assemble individual track segments in various configurations. It was sometimes more fun thinking of new and unique track layouts than actually racing the toy cars.

Implementing a level editor

In this section, we will review some code that will implement the core components of our level editor. Unlike in previous chapters, we will not try to make this code runnable or testable. Instead, we will review the implementations to understand how we use the general idea of spatial partitioning to build a functional level editor for designers while optimizing the way we load levels at runtime.

The code presented in the next section is to be reviewed but not compiled, as it's not a complete self-contained example.

Steps for implementing a level editor

1. To start, we are going to write a `ScriptableObject` class named `Track`, as follows:

```
using UnityEngine;
using System.Collections.Generic;

namespace Chapter.SpatialPartition
{
    [CreateAssetMenu(fileName = "New Track", menuName = "Track")]
    public class Track : ScriptableObject
    {
```

```
        [Tooltip("The expected length of segments")]
        public float segmentLength;

        [Tooltip("Add segments in expected loading order")]
        public List<GameObject> segments = new List<GameObject>();
    }
}
```

With this `ScriptableObject` class, our level designers will be able to design new variations of race tracks by adding segments into a list and then sequencing them in a specific order. Each track asset will be fed to the `TrackController` class, which will spawn each segment automatically and in the order that the designers sequenced them.

For the player, this process is seamless as it runs in the background, and segments are spawned before they are in the camera's field of view. So, from the point of view of the player, it looks like the whole level is loaded.

2. Next up is the `TrackController` class. In it, we are going to implement the segment-loading mechanism, but because it's an extensive class, we are going to divide it up and look at it in sections, as follows:

```
using UnityEngine;
using System.Linq;
using System.Collections.Generic;

namespace Chapter.SpatialPartition
{
    public class TrackController : MonoBehaviour
    {
        private float _trackSpeed;
        private Transform _prevSeg;
        private GameObject _trackParent;
        private Transform _segParent;
        private List<GameObject> _segments;
        private Stack<GameObject> _segStack;
        private Vector3 _currentPosition = new Vector3(0, 0, 0);

        [Tooltip("List of race tracks")]
        [SerializeField]
        private Track track;

        [Tooltip("Initial amount of segment to load at start")]
        [SerializeField]
        private int initSegAmount;

        [Tooltip("Amount of incremental segments to load at run")]
```

```
        [SerializeField]
        private int incrSegAmount;

        [Tooltip("Dampen the speed of the track")]
        [Range(0.0f, 100.0f)]
        [SerializeField]
        private float speedDampener;

        void Awake()
        {
            _segments =
                Enumerable.Reverse(track.segments).ToList();
        }

        void Start()
        {
            InitTrack();
        }
```

The first section is just the initialization code and is self-explanatory, but the following region of the code gets more interesting:

```
void Update()
{
    _segParent.transform.Translate(
        Vector3.back * (_trackSpeed * Time.deltaTime));
}

private void InitTrack()
{
    Destroy(_trackParent);

    _trackParent =
        Instantiate(
            Resources.Load("Track", typeof(GameObject)))
            as GameObject;

    if (_trackParent)
        _segParent =
            _trackParent.transform.Find("Segments");

    _prevSeg = null;

    _segStack = new Stack<GameObject>(_segments);

    LoadSegment(initSegAmount);
}
```

As we can see, in the `Update()` loop, we are moving the track parent object toward the player to simulate movement. And in the `InitTrack()` method, we instantiate a track `GameObject`, which will act as the container of the track segments. But there's one significant line of code in the function that's a critical component to our segment-loading mechanism, and this is illustrated here:

```
_segStack = new Stack<GameObject>(_segments);
```

On this line, we are injecting the list of segments into a new Stack container. As mentioned at the beginning of the chapter, an essential part of the spatial-partitioning technique is the organization *of environment objects in a data structure* so that they are easier to query.

In the next code snippet, we are going to see how we use the Stack data structure to load segments in the correct order:

```
private void LoadSegment(int amount)
{
    for (int i = 0; i < amount; i++)
    {
        if (_segStack.Count > 0)
        {
            GameObject segment =
                Instantiate(
                    _segStack.Pop(), _segParent.transform);

            if (!_prevSeg)
                _currentPosition.z = 0;

            if (_prevSeg)
                _currentPosition.z =
                    _prevSeg.position.z
                    +
                    track.segmentLength;

            segment.transform.position = _currentPosition;

            segment.AddComponent<Segment>();

            segment.GetComponent<Segment>().
                trackController = this;

            _prevSeg = segment.transform;
        }
    }
}
```

```
        public void LoadNextSegment()
        {
            LoadSegment(incrSegAmount);
        }
    }
}
```

The `LoadSegment()` private method is at the heart of the system. It accepts as a parameter a specific amount of segments. This value will determine the number of segments that it will load when called. If there are enough segments remaining on the stack, it pops one from the top and initializes it behind the previously loaded segment. It continues this circular process until it has loaded the expected amount.

You might be asking yourself: *How do we destroy segments that have passed behind the player?* There are many ways we can calculate or detect if one entity is behind another, but for our context, we are going to use a twofold solution. Every segment prefab has an entity called a segment marker loaded at its edge; this is composed of two pillars and an invisible trigger.

Once the bike goes through the trigger, the segment marker deletes its parent `GameObject`, as we can see here:

```
using UnityEngine;

public class SegmentMarker : MonoBehaviour
{
    private void OnTriggerExit(Collider other)
    {
        if (other.GetComponent<BikeController>())
            Destroy(transform.parent.gameObject);
    }
}
```

When an entity with the `BikeController` component exits a segment marker's trigger, it requests the destruction of its parent `GameObject`, which in this case would be a `Segment` entity.

When an entity with the `BikeController` component exits a segment marker's trigger, it requests the destruction of its parent `GameObject`, which would be a `Segment` entity in this context.

As seen in the `LoadSegment()` method from the `TrackController` class, every time we pop a new segment from the top of the stack, we attach to it a script as a component named `Segment`, as seen here:

```
segment.transform.position = _currentPosition;

segment.AddComponent<Segment>();

segment.GetComponent<Segment>().trackController = this;
```

Because we are passing the current instance of the `TrackController` class to its `trackController` parameter, the `Segment` object can call back the `TrackController` class and request the loading of the following sequence of segments just before it gets destroyed, as we can see here:

```
using UnityEngine;

public class Segment : MonoBehaviour
{
    public TrackController trackController;
    private void OnDestroy()
    {
        if (trackController)
            trackController.LoadNextSegments();
    }
}
```

This approach creates a circular mechanism that loads and unloads a controlled amount of segments automatically at specific intervals. With this approach, we are managing the number of spawned entities in the scene at a given time. In theory, this will result in a more consistent frame rate.

Another benefit of this approach, which is more gameplay-related, is that the segment markers can act as landmarks for a checkpoint system. Checkpoints are often used in time-limit racing game modes in which a player must reach several points on the track within a specific timeframe.

An excellent example of a checkpoint-based racing game is *Rad Racer* from 1987.

Using the level editor

You can play with the level editor by opening up the `/FPP` folder in the Git repository and then do the following:

- Under the `/Scenes/Gyms` folder, you should find a scene named `Segment`. In this scene, you will be able to edit and create new segment prefabs.
- Under the **Assets-> Create-> Track** menu, you have an option to create new track assets.
- And finally, you can modify and attach new tracks to the `TrackController` class by opening the `Track` scene under the `Scenes/Main` folder.

Feel free to improve the code and, more importantly, have fun!

Reviewing the level-editor implementation

The implementations in this chapter are simplified versions of the code of a more complex system, but if you take the time to review an advanced version of the level editor in the `/FPP` folder of the Git project, we will see some improvements, such as the following:

- **Segments**: There's an authoring pipeline for segments that uses ScriptableObjects.
- **Object pooling**: The `TrackController` class is using an object pool to optimize the loading time of individual segments.

I didn't include these optimizations in the chapter to keep the code examples short and simple, for educational purposes.

Reviewing alternative solutions

In an actual production context, and if time permits, I would build our game's level editor differently. I would instead design a top-down track editor that would allow the level designers to draw rails and drag and drop obstacles on them. The designers would then be able to save their work in a serialized format.

Then, using spatial-partitioning principles, the tracks would be automatically divided into segments by the `TrackController` class and put into an object pool. This approach would automate the process of generating individual segments while optimizing the spawning process.

Consequently, the designers would not have to author individual segments as prefabs, and they could design new tracks while visualizing the entire layout in an editor.

When I'm building tools and setting up integration pipelines, my end goal is always automation. I always try to automate myself out of a job so that I don't waste time on manual tasks.

Summary

In this chapter, we took a hands-off approach and reviewed how to build a basic level editor while using the broad ideas of the Spatial Partition pattern. Our goal wasn't to be faithful to standard definitions of the pattern. Instead, we use it as a starting point to build our system. I encourage you to take the time to review the code in the `/FPP` folder and refactor it to make it better.

In the next chapter, we will review some alternative patterns that are good to know but have general use cases. Therefore, compared to the previous chapters, the use cases will have a broader scope without being specific to a game mechanic or system. The first pattern that we will tackle is the Adapter pattern. As its name implies, we will use it to integrate an adapter between two incompatible systems.

Section 3: Alternative Patterns

In this section of the book, we will review some notable patterns that we didn't use while programming our game as they are designed for more general use cases. These patterns are good to know and could help Unity developers in solving various architectural issues.

This section comprises the following chapters:

- `Chapter 14`, *Adapting Systems with an Adapter*
- `Chapter 15`, *Concealing Complexity with a Façade Pattern*
- `Chapter 16`, *Managing Dependencies with the Service Locator Pattern*

14
Adapting Systems with an Adapter

In a world full of different types of cables and plugs, we have all become accustomed to the concept of adapters. The Adapter pattern will be one of those patterns that will be easy for you to grasp because it correlates so perfectly with our real-world experiences with technology. The Adapter pattern's name perfectly reveals its core purpose; it offers us a way to seamlessly use old code with new code by adding an interface between the code that will act as an adapter.

The following topics will be covered in this chapter:

- Understanding the Adapter pattern
- Implementing the Adapter pattern

The examples presented in this chapter are skeleton code. It's simplified for learning purposes so we can focus on the structure of the pattern. It might not be optimized or contextualized enough to be used as is in your project.

Technical requirements

The following chapter is hands-on, so you will need to have a basic understanding of Unity and C#.

The code files of this chapter can be found on GitHub: `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter14`

Check out the following video to see the Code in Action:
`https://bit.ly/3wBHqkX`

Understanding the Adapter pattern

As its name implies, the Adapter pattern adapts two incompatible interfaces; like a plug adapter, it doesn't modify what it adjusts but bridges one interface with another. This approach can be beneficial when dealing with legacy code that you cannot refactor due to its fragility, or when you need to add features to a third-party library but don't want to modify it to avoid issues when upgrading it.

Here's a quick breakdown of the two main approaches to implementing the Adapter pattern:

- **Object Adapter**: In this version, the pattern uses object composition, and the adapter acts as a wrapper around the adapted object. It's helpful if we have a class that doesn't have the methods we require, but we can't modify it directly. The Object Adapter adopts the methods of the original class and adapts them to what we need.
- **Class Adapter**: In this version of the pattern, the adapter uses inheritance to adapt the interface of an existing class to that of another. It's useful when we need to adjust a class so it can work with others but can't modify it directly.

In our code example, we will use a permutation of the Class Adapter as it's more challenging to learn. If we understand the Class Adapter version, then we can extrapolate an understanding of the Object Adapter version.

Let's take a look at a side-by-side diagram of the Object and Class Adapters; the core differences are subtle, but the similarities are apparent:

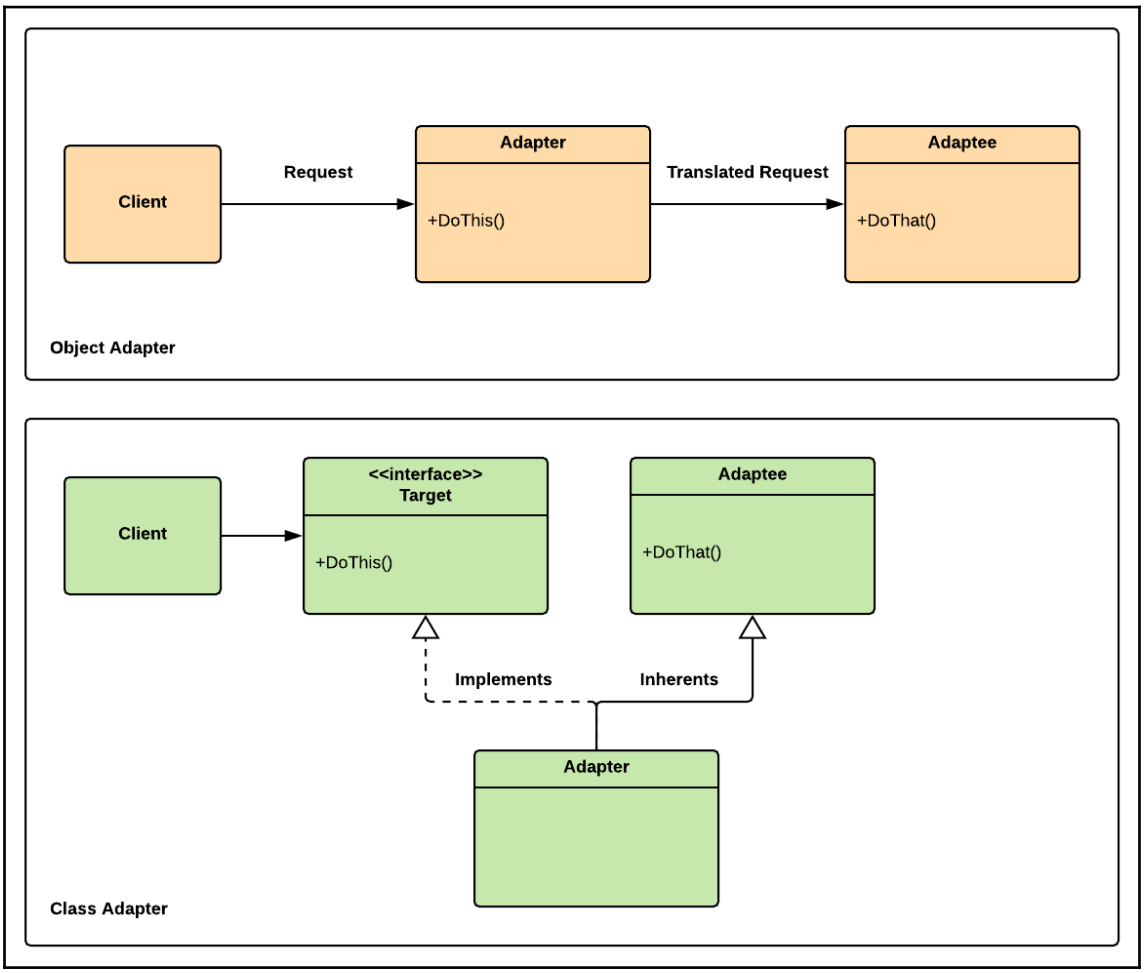


Figure 14.1 – Diagram of a replay system

As you can see, in both cases, the **Adapter** class is positioned between the **Client** and the adapted entity (**Adaptee**). But the Class Adapter establishes a relationship with the Adaptee through inheritance. In contrast, the Object Adapter uses composition to wrap an instance of the Adaptee for adaption.

In both cases, the entity that we are adapting is not modified. And the client is not aware of what we are adapting; it just knows that it has a consistent interface to communicate with the adapted object.

Composition and inheritance are ways to define relationships between objects; they describe how they relate to each other. And that difference in the relationship structure determines in part the difference between the Object and Class Adapters.

Another point to address is that the Adapter can sometimes be confused with the Facade pattern. We must understand that the core difference between them is that the Facade pattern establishes a simplified front-facing interface to a complex system. But the Adapter adapts incompatible systems while maintaining a consistent interface for a client.

Both patterns are related as they are both structural patterns but have entirely different purposes.

Composition is one of the core concepts of **Object-Oriented Programming** (**OOP**); it refers to the concept of combining simple types to make more complex ones. For instance, a motorbike has wheels, an engine, and a handlebar. So composition establishes a "has a" relationship while inheritance sets an "is a" relationship.

Benefits and drawbacks of the Adapter pattern

The following are some of the benefits of the Adapter pattern:

- **Adapting without modifying**: The main benefit of the Adapter pattern is that it offers a standard approach to adapting old or third-party code without modifying it.
- **Reusability and flexibility**: This pattern permits the continued use of legacy code on new systems with minimal changes; this has an immediate return on investment.

The following are some potential drawbacks of the Adapter pattern:

- **Persisting legacy**: The ability to use legacy code with new systems is cost-effective, but in the long term, it can become an issue, because the old code might limit your upgrade options as it becomes deprecated and incompatible with new versions of Unity or third-party libraries.
- **Slight overhead**: Because, in some instances, you are redirecting calls between objects, there might be a slight performance hit, usually too small to become an issue.

The Adapter is part of the structural pattern family, including the Facade, Bridge, Composite, Flyweight, and Decorator.

When to use the Adapter pattern

A potential use case for the Adapter in Unity is when you have a third-party library that you downloaded from the Unity Asset Store, and you need to modify some of its core classes and interfaces to add new features. But when changing third-party code, you risk having merge issues every time you pull an update from the library owners.

Hence, you find yourself in a situation where you choose to wait for the third-party library owners to integrate the changes you need or modify their code and add the missing features. Both choices have their risks versus rewards. But the Adapter pattern gives us a solution to this dilemma by letting us place an adapter between existing classes so they can work together without modifying them directly.

Let's imagine we are working on the code base of a project that uses an inventory system package downloaded from the Unity Asset Store. The system is excellent; it saves the player's purchased or gifted inventory items to a secure cloud backend service. But there's one problem, it doesn't support local disk saves. This limitation has become an issue in our project because we need both local and cloud saves for redundancy purposes.

In this scenario, we could easily modify the vendor's code and add the feature we need. But when they release their next update, we will have to merge our code with theirs. This approach can be an error-prone process. Instead, we will use the Adapter pattern and implement a wrapper that will maintain a consistent interface to the inventory system while adding local-save support. And in the process, we will not have to modify any of the existing classes. Thus, we will be able to avoid changing the vendor's code and still have your local save system handle the saving of inventory items.

We will implement an example of this use case in the next section. But in conclusion, the Adapter pattern is handy in situations in which you have incompatible systems that need to interface with each other. Still, you want to avoid modifying any existing code.

Implementing the Adapter pattern

The code example will be simple; we will not attempt to write an entire local disk save system because that's not the focus of this chapter. Instead, we will write the skeleton of the system to concentrate on the use of the Adapter pattern and not be impeded by unrelated implementation details.

Implementing the Adapter pattern

To start off, we will implement a placeholder class that will mock the third-party inventory system as presented in the scenario of the previous section:

1. The `InventorySystem` class provided by our fictional provider has three methods, `AddItem()`, `RemoveItem()`, and `GetInventory()`. All of these methods are hardcoded to use cloud storage and we can't modify them:

```
using UnityEngine;
using System.Collections.Generic;

namespace Chapter.Adapter
{
    public class InventorySystem
    {
        public void AddItem(InventoryItem item)
        {
            Debug.Log(
                "Adding item to the cloud");
        }

        public void RemoveItem(InventoryItem item)
        {
            Debug.Log(
                "Removing item from the cloud");
        }

        public List<InventoryItem> GetInventory()
        {
            Debug.Log(
                "Returning an inventory list stored in the cloud");

            return new List<InventoryItem>();
        }
    }
}
```

2. Next up is the class that will act as our adapter in this scenario. It adds the ability to save inventory items to a local disk. But it also exposes a new functionality that permits the merging and syncing of the player's local and cloud inventories:

```
using UnityEngine;
using System.Collections.Generic;

namespace Chapter.Adapter {
    public class InventorySystemAdapter:
        InventorySystem, IInventorySystem {

        private List<InventoryItem> _cloudInventory;

        public void SyncInventories() {
            var _cloudInventory = GetInventory();

            Debug.Log(
                "Synchronizing local drive and cloud inventories");
        }

        public void AddItem(
            InventoryItem item, SaveLocation location) {

            if (location == SaveLocation.Cloud)
                AddItem(item);

            if (location == SaveLocation.Local)
                Debug.Log("Adding item to local drive");

            if (location == SaveLocation.Both)
                Debug.Log(
                    "Adding item to local drive and on the cloud");
        }

        public void RemoveItem(
            InventoryItem item, SaveLocation location) {

            Debug.Log(
                "Remove item from local/cloud/both");
        }

        public List<InventoryItem>
            GetInventory(SaveLocation location) {

            Debug.Log(
                "Get inventory from local/cloud/both");

            return new List<InventoryItem>();
```

```
        }
    }
}
```

Notice that by inheriting the third party's `InventorySystem` class, we have access to all its properties and methods. Thus we can continue to use its core functionalities while adding our own. We are modifying nothing in the process, just adapting it.

3. We are going to expose an interface to our new inventory system:

```
using System.Collections.Generic;

namespace Chapter.Adapter
{
    public interface IInventorySystem
    {
        void SyncInventories();

        void AddItem(
            InventoryItem item, SaveLocation location);

        void RemoveItem(
            InventoryItem item, SaveLocation location);

        List<InventoryItem> GetInventory(SaveLocation location);
    }
}
```

The client who uses this interface is not aware that it's communicating with a system adapted from another. The Adaptee is also not aware that we are adapting it. Like a charger adapter, the phone and cable are unaware of which plug they are connected to, only that current is passing through the system and charging the battery.

4. To complete our implementation, we need to add an `enum` that exposes the save locations:

```
namespace Chapter.Adapter
{
    public enum SaveLocation
    {
        Disk,
        Cloud,
        Both
    }
}
```

5. And for our last step, we will implement a placeholder `InventoryItem` class:

```
using UnityEngine;

namespace Chapter.Adapter
{
    [CreateAssetMenu(
        fileName = "New Item", menuName = "Inventory")]
    public class InventoryItem : ScriptableObject
    {
        // Placeholder class
    }
}
```

Testing the Adapter pattern implementation

To test our implementation in your instance of Unity, copy all the classes we just reviewed into your project and attach the following client class to an empty GameObject in a new Unity scene:

```
using UnityEngine;

namespace Chapter.Adapter
{
    public class ClientAdapter : MonoBehaviour
    {
        public InventoryItem item;
        private InventorySystem _inventorySystem;
        private IInventorySystem _inventorySystemAdapter;

        void Start()
        {
            _inventorySystem = new InventorySystem();
            _inventorySystemAdapter = new InventorySystemAdapter();
        }

        void OnGUI()
        {
            if (GUILayout.Button("Add item (no adapter)"))
                _inventorySystem.AddItem(item);

            if (GUILayout.Button("Add item (with adapter)"))
                _inventorySystemAdapter.
                    AddItem(item, SaveLocation.Both);
```

```
            }
        }
    }
```

We now can build a new inventory system that uses the functionality of an older one provided by a third party. We can continue to pull library updates from the third-party website with confidence and without the need to worry about merge issues. Our system can grow in features while we continue to adapt theirs, and if one day we want to remove their system from our code base and just use our own, we could start the deprecation process by updating the `adapter` class.

Summary

In this chapter, we added the Adapter pattern to our toolbox. It's a type of pattern that's very beneficial to have in our back pocket. One of the biggest challenges for a professional programmer is dealing with incompatible systems that are often developed by external vendors or other teams inside an organization. A consistent approach to adapting existing classes can only be helpful, especially when time becomes an issue and it's just faster to reuse old code for new purposes.

In the next chapter, we will review a close cousin of the Adapter, the Facade pattern, which we will use to manage the growing complexity in our code.

15
Concealing Complexity with a Facade Pattern

The Facade pattern is an easy pattern to grasp because its name implies its purpose. The primary intent of the Facade pattern is to offer a simplified front-facing interface that abstracts the intricate inner workings of a complex system. This approach is beneficial for game development because games are composed of complex interactions between various systems. As a use case, we will write code that simulates the behavior and interactions of a vehicle's engine core components and then offers a simple interface to interact with the overall system.

The following topics will be covered in this chapter:

- Understanding the Facade pattern
- Designing a bike engine
- Implementing a bike engine
- A basic implementation of a vehicle's engine with the Facade pattern

This section includes a simplified version of the implementation of an engine for simplicity and brevity reasons. A complete implementation of this code example can be found in the `/FPP` folder of the GitHub project—the link is available under the *Technical requirements* section.

Technical requirements

This is a hands-on chapter, so you will need to have a basic understanding of Unity and C#.

The code files of this chapter can be found on GitHub at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter15`.

Check out the following video to see the code in action:

`https://bit.ly/36wJdxe`

Understanding the Facade pattern

The Facade pattern's name is analogous to the facade of a building—as the name implies, it's an exterior face that hides a complex inner structure. Contrary to building architecture, in software development, the goal of a facade is not to beautify; instead, it is to simplify. As we are going to see in the following diagram, an implementation of the Facade pattern is usually limited to a single class that acts as a simplified interface to a collection of interacting subsystems:

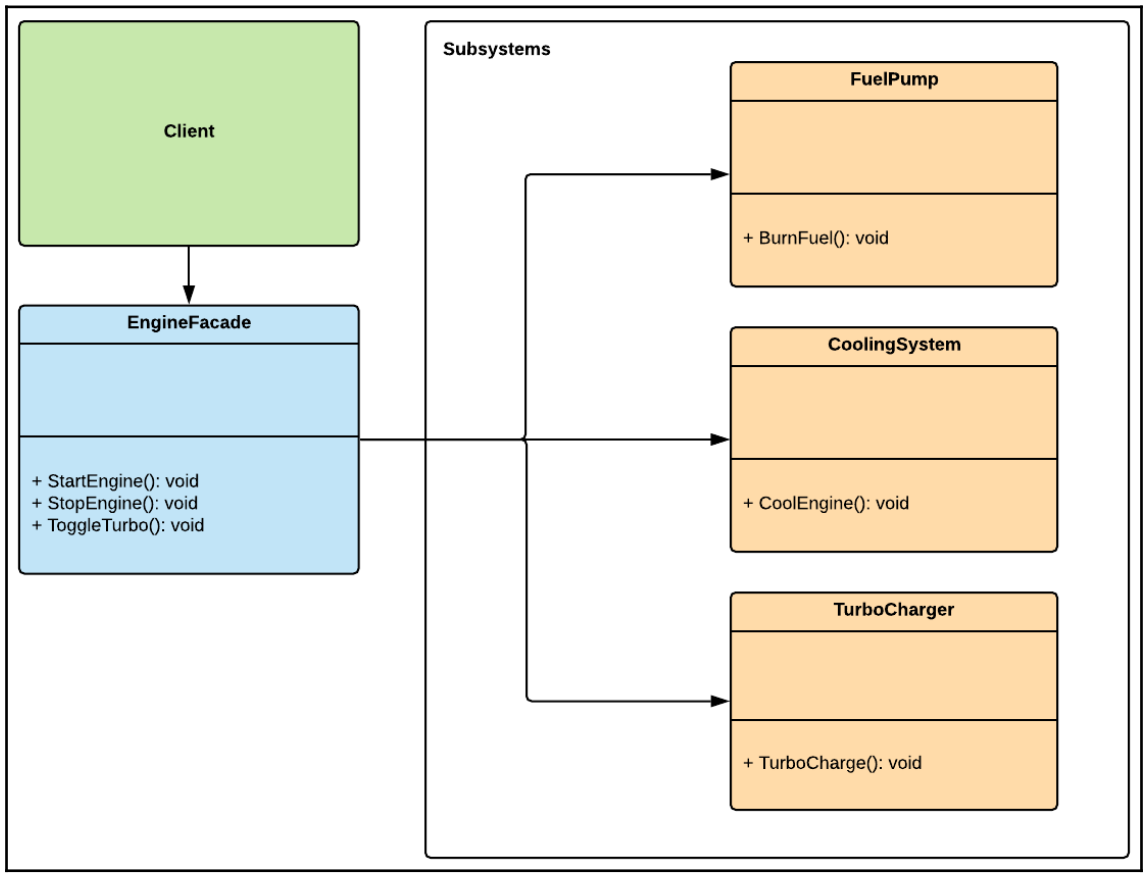


Figure 15.1 – Unified Modeling Language (UML) diagram of the Facade pattern

As we can see in the preceding diagram, `EngineFacade` acts as an interface to the various components of the engine, thereupon the client is unaware of what is happening behind the scenes when calling `StartEngine()` on `EngineFacade`. It's unaware of the components that make up the engine and how to reach them; it only knows what it needs to know. This is similar to what happens when you turn the ignition key in your car—you don't see what's happening under the hood, and you don't need to; your only concern is that the engine starts. And so, the Facade pattern offers this same level of abstraction in your code, keeping the details of the system under the hood.

The Facade pattern is part of the structural pattern category. Patterns in this classification are concerned with how classes and objects are composed to form larger structures.

Benefits and drawbacks

Here are some **benefits** of the Facade pattern:

- **Simplified interface to a complex body of code**: A solid Facade class will conceal complexity from a client while providing a simplified interface to interact with an intricate system.
- **Easy refactoring**: It's easier to refactor code that's isolated behind a Facade because the system's interface remains consistent to the client while its components are being modified behind the scenes.

The following are some **drawbacks** to watch out for:

- **It makes it easier to hide the mess**: Using the Facade pattern to hide messy code behind a clean front-facing interface will defeat the pattern's core benefits in the long run, but this pattern does offer a way to mask some code smells until you have time to refactor them. However, expecting to have enough time later to fix stuff is a trap in itself because we rarely have enough time to refactor things correctly.
- **Too many facades**: Globally accessible manager classes that act as facades to core systems are popular among Unity developers; they often implement them by combining the Singleton and Facade patterns. Unfortunately, it's too easy to abuse this combination and end up with a code base comprising too many manager classes, each dependent on the other to function. As a consequence, debugging, refactoring, and unit testing components becomes very difficult.

The Facade pattern establishes a new interface, whereas the Adapter pattern adapts an old interface. Therefore, when implementing patterns that might look and sound similar, it's essential to keep in mind that they're not necessarily identical in their purpose.

Designing a bike engine

We are not aiming to implement a complete simulation of an actual gas engine for a bike; this will take too long and demands an in-depth understanding of the physics and mechanics of a real-world engine. But we will aim to simulate, to a minimal degree, some standard components of a high-speed vehicle's motor. So first, let's break down the expected behavior of each part of our engine, as follows:

- **Cooling system**: The cooling system is responsible for making sure the engine is not overheating. When the turbocharger is activated, the cooling system shuts down during a turbocharge. This behavior means that if the player overuses the turbocharger, this can overheat the engine, and in consequence, the engine will stop or explode and the bike will stop moving.
- **Fuel pump**: This component is responsible for managing the fuel consumption of the bike. It knows the amount of gas remaining and stops the engine if it runs out of it.
- **Turbocharger**: If the turbocharger is activated, the vehicle's top speed is increased, but the cooling system shuts down temporarily so that the engine's circuitry can relay power to the charger.

In the next section, we will implement a skeleton class for each of these components and establish an engine `Facade` class so that clients can start and stop the engine.

The design intention of having the cooling system shut down when the turbocharger is activated is to create a sense of risk versus reward. The player must balance the desire to go faster with the potential consequence of overheating the engine.

Implementing a bike engine

As we are going to see, the Facade pattern is straightforward, so we will keep the following code example simple and straight to the point. To start, we will write the classes for each of the core components that make up the bike's engine, as follows:

1. We will start with the fuel pump; the purpose of this component is to simulate the consumption of fuel so that it knows the amount remaining and shuts down the engine when it runs out. Here's the code you'll need:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Facade
{
    public class FuelPump : MonoBehaviour
    {
        public BikeEngine engine;
        public IEnumerator burnFuel;

        void Start()
        {
            burnFuel = BurnFuel();
        }

        IEnumerator BurnFuel()
        {
            while (true)
            {
                yield return new WaitForSeconds(1);
                engine.fuelAmount -= engine.burnRate;

                if (engine.fuelAmount <= 0.0f) {
                    engine.TurnOff();
                    yield return 0;
                }
            }
        }

        void OnGUI()
        {
            GUI.color = Color.green;
            GUI.Label(
                new Rect(100, 40, 500, 20),
                "Fuel: " +  engine.fuelAmount);
        }
```

```
        }
    }
```

2. Next up is the cooling system, which is responsible for preventing the engine from overheating but gets deactivated if the turbocharger is activated. The code is illustrated in the following snippet:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Facade {
    public class CoolingSystem : MonoBehaviour {
        public BikeEngine engine;
        public IEnumerator coolEngine;
        private bool _isPaused;

        void Start() {
            coolEngine = CoolEngine();
        }

        public void PauseCooling() {
            _isPaused = !_isPaused;
        }

        public void ResetTemperature() {
            engine.currentTemp = 0.0f;
        }

        IEnumerator CoolEngine() {
            while (true) {
                yield return new WaitForSeconds(1);

                if (!_isPaused) {
                    if (engine.currentTemp > engine.minTemp)
                        engine.currentTemp -= engine.tempRate;
                    if (engine.currentTemp < engine.minTemp)
                        engine.currentTemp += engine.tempRate;
                } else {
                    engine.currentTemp += engine.tempRate;
                }

                if (engine.currentTemp > engine.maxTemp)
                    engine.TurnOff();
            }
        }

        void OnGUI() {
            GUI.color = Color.green;
```

```
            GUI.Label(
                new Rect(100, 20, 500, 20),
                "Temp: " +  engine.currentTemp);
        }
    }
}
```

3. And finally, the turbocharger, when activated, increases the bike's top speed, but for it to function, it needs to deactivate the cooling system temporarily. Here's the code to accomplish this:

```
using UnityEngine;
using System.Collections;

namespace Chapter.Facade
{
    public class TurboCharger : MonoBehaviour
    {
        public BikeEngine engine;
        private bool _isTurboOn;
        private CoolingSystem _coolingSystem;

        public void ToggleTurbo(CoolingSystem coolingSystem)
        {
            _coolingSystem = coolingSystem;
            if (!_isTurboOn)
                StartCoroutine(TurboCharge());
        }

        IEnumerator TurboCharge()
        {
            _isTurboOn = true;
            _coolingSystem.PauseCooling();

            yield return new WaitForSeconds(engine.turboDuration);

            _isTurboOn = false;
            _coolingSystem.PauseCooling();
        }

        void OnGUI()
        {
            GUI.color = Color.green;
            GUI.Label(
                new Rect(100, 60, 500, 20),
                "Turbo Activated: " +  _isTurboOn);
        }
```

```
    }
}
```

4. Now that we have the core components of our engine ready, we need to implement a class that will permit a client to interact with them seamlessly. So, we will implement a Facade class called `BikeEngine` that will offer an interface for a client to start and stop the engine and toggle the turbocharger. But because the following code example is so long, we will review it in two parts. Here's the first part:

```
using UnityEngine;

namespace Chapter.Facade
{
    public class BikeEngine : MonoBehaviour
    {
        public float burnRate = 1.0f;
        public float fuelAmount = 100.0f;
        public float tempRate = 5.0f;
        public float minTemp = 50.0f;
        public float maxTemp = 65.0f;
        public float currentTemp;
        public float turboDuration = 2.0f;

        private bool _isEngineOn;
        private FuelPump _fuelPump;
        private TurboCharger _turboCharger;
        private CoolingSystem _coolingSystem;

        void Awake() {
            _fuelPump =
                gameObject.AddComponent<FuelPump>();
            _turboCharger =
                gameObject.AddComponent<TurboCharger>();
            _coolingSystem =
                gameObject.AddComponent<CoolingSystem>();
        }

        void Start() {
            _fuelPump.engine = this;
            _turboCharger.engine = this;
            _coolingSystem.engine = this;
        }
```

The first part of this class is initialization code and is self-explanatory, but the following segment is the important part:

```
        public void TurnOn() {
            _isEngineOn = true;
            StartCoroutine(_fuelPump.burnFuel);
            StartCoroutine(_coolingSystem.coolEngine);
        }

        public void TurnOff() {
            _isEngineOn = false;
            _coolingSystem.ResetTemperature();
            StopCoroutine(_fuelPump.burnFuel);
            StopCoroutine(_coolingSystem.coolEngine);
        }

        public void ToggleTurbo() {
            if (_isEngineOn)
                _turboCharger.ToggleTurbo(_coolingSystem);
        }

        void OnGUI() {
            GUI.color = Color.green;
            GUI.Label(
                new Rect(100, 0, 500, 20),
                "Engine Running: " +  _isEngineOn);
        }
    }
}
```

As we can see, the `EngineFacade` class exposes the available functionality that the bike engine offers but, at the same time, conceals the interaction between its components. If we want to start the engine, we just need to call the `StartEngine()` method. If we didn't have a Facade pattern such as the one we just implemented, we would have to initialize each engine component individually and know each of their parameters to set and the methods to call. The Facade pattern permits us to tuck all the complexity away behind a clean interface.

But suppose we wished to add another engine component, such as a nitro injector; in that case, we would only need to modify the `BikeFacade` class and expose a new public method to allow us to trigger the injector.

Testing the engine facade

We can quickly test the code we just implemented by adding the following client script to a GameObject in an empty Unity scene:

```
using UnityEngine;

namespace Chapter.Facade
{
    public class ClientFacade : MonoBehaviour
    {
        private BikeEngine _bikeEngine;

        void Start()
        {
            _bikeEngine =
                gameObject.AddComponent<BikeEngine>();
        }

        void OnGUI()
        {
            if (GUILayout.Button("Turn On"))
                _bikeEngine.TurnOn();

            if (GUILayout.Button("Turn Off"))
                _bikeEngine.TurnOff();

            if (GUILayout.Button("Toggle Turbo"))
                _bikeEngine.ToggleTurbo();
        }
    }
}
```

In the client class, we see that it's not aware of the engine's inner workings, and this is the effect we want to achieve when using the Facade pattern. The only thing the client class knows is that it can start and stop the engine and toggle on a turbocharged feature by calling the public methods made available by the `BikeEngine` class. To put it another way, just as in real life, we don't need to open the hood to start the engine; we turn the ignition key, and the components start working together without us knowing how they are interacting with each other.

In the next section, we will review alternative solutions to consider before deciding on using the Facade pattern.

In a potentially more advanced version of this code example, the engine would calculate the current **revolutions per minute** (**RPM**)—also known as the engine's speed—and we could connect it to a gear system regulated by a shifter input with which the player could control the bike's speed. Thus, we could easily enhance the level of realism at any time.

Reviewing alternative solutions

There are several alternatives to keep in mind before considering the Facade pattern, depending on what you are actually trying to accomplish. These are listed here:

- **Abstract Factory pattern:** If you only want to conceal how subsystem objects are initialized from the client code, you should consider using the Abstract Factory pattern instead of the Facade pattern.
- **Adapter**: If you are intending to write a "wrapper" over existing classes with the intent to bridge two incompatible interfaces, then you should consider using the Adapter pattern.

Summary

Even though the Facade pattern is sometimes used to hide messy code, when you use it as intended, it can enhance your code base's readability and usability by masking underlying complex interactions of subsystems behind a singular front-facing interface. Thus, it's a pattern that can be very beneficial for game programming, but one to use wisely and with good intent.

In the upcoming chapter, we will explore a pattern named Service Locator, which we will use to manage global dependencies and expose core services.

16
Managing Dependencies with the Service Locator Pattern

This chapter will be brief because the Service Locator pattern we will review is simple and efficient. The core idea of this pattern is straightforward: it revolves around having a central registry of initialized dependencies. But to be more precise, these dependencies are components that offer specific services that we can expose with interfaces we call "service contracts". Hence, when a client needs to call upon a particular service, it doesn't need to know how to localize and initialize it; it just needs to ask the Service Locator pattern, and this will do all the legwork to fulfill the service contract.

As we will see in this chapter, it's quite a simple design and is easy to implement.

In this chapter, we will cover the following topics:

- Understanding the Service Locator pattern
- Implementing a Service Locator pattern
- Reviewing alternative solutions

We simplified this chapter's code example for learning purposes, to present the pattern's core concepts without being distracted by the implementation details. So, the code shown is neither optimized nor contextualized enough to be used as-is in a project.

Technical requirements

The following chapter is hands-on, so you will need to have a basic understanding of Unity and C#.

We will be using the following Unity-specific engine and C# language concepts:

- Statics
- Generics

If unfamiliar with these concepts, please review `Chapter 3`, *A Short Primer to Programming in Unity*.

The code files of this chapter can be found on GitHub, at `https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter16`.

Check out the following video to see the code in action: `https://bit.ly/36AKWli`

Static is a keyword modifier. A method declared as static in a class can be called without instantiating an object.

Understanding the Service Locator pattern

Compared to more traditional patterns, the Service Locator pattern has less academic theory behind it and is very pragmatic in its overall design. As its name implies, its purpose is to locate services for a client. It achieves this by maintaining a central registry of objects that offer specific services.

Let's review a diagram of a typical Service Locator implementation:

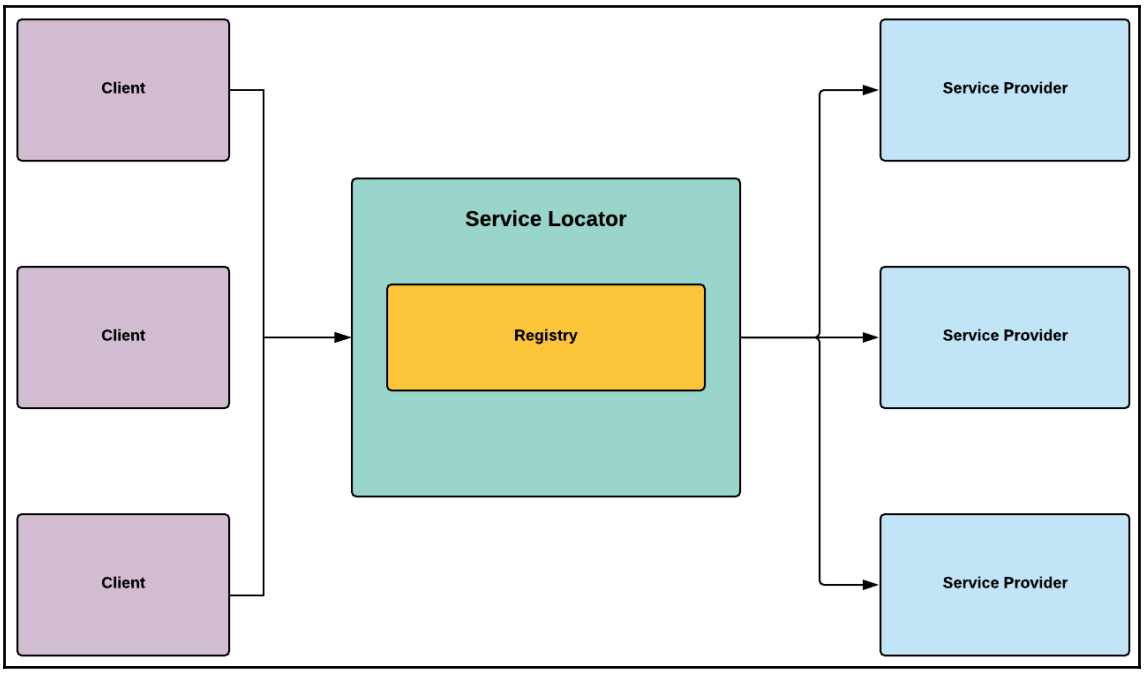


Figure 16.1 – Diagram of the Service Locator pattern

As we can see, we could easily say that the Service Locator pattern is acting as a proxy between the clients (requestors) and the service providers, and this approach decouples them to a certain degree. A client will only need to call the Service Locator pattern when it has a dependency to resolve and needs access to a service. We could say that the Service Locator pattern is acting similarly to a waiter in a restaurant, taking orders from clients and acting as an intermediary between the various services that the restaurant offers to its clientele.

The Service Locator pattern has a bad reputation in some circles; experts often criticize it for being an anti-pattern. The core reason for this criticism is that it violates several coding best practices as it hides class dependencies instead of exposing them. In consequence, this could make your code harder to maintain and test.

Benefits and drawbacks of the Service Locator pattern

Here are some of the potential benefits of using the Service Locator pattern:

- **Runtime optimization**: The Service Locator pattern can optimize an application by dynamically detecting more optimized libraries or components to complete a specific service, depending on the runtime context.
- **Simplicity**: Service Locator is one of the most straightforward dependency management patterns to implement and doesn't have the steep learning curve of a **dependency injection** (**DI**) framework. Hence, you can quickly start using it in a project or teach it to colleagues.

Here are some drawbacks of using the Service Locator pattern:

- **Black boxing**: The Service Locator pattern's registry obfuscates class dependencies. Consequently, some issues might pop up at runtime instead of during compilation if dependencies are missing or incorrectly registered.
- **Globally dependent**: If overly used and with the wrong intentions, the Service Locator pattern can become a global dependency in itself that's arduous to manage. Your code will become excessively dependent on it, and eventually, it won't be easy to decouple it from the rest of your core components.

The Service Locator pattern was popular among Java developers; it was defined in part by Martin Fowler in a blog post published in 2004, which you can read here:

`https://martinfowler.com/articles/injection.html`

When to use the Service Locator pattern

The question on when to use the Service Locator pattern is self-explanatory, based on its description. For example, if you have a list of services that you dynamically need to access but want to encapsulate the process involved in obtaining them, then this pattern can offer a solution.

But another aspect we should consider when contemplating using the Service Locator pattern is when not to use it. Because a Service Locator pattern is usually globally accessible, as its name implies, it should locate and provide access to services. Then, we should use it only to expose services that have a global scope.

For instance, we need to access the **heads-up display** (**HUD**) to update one of its **user interface** (**UI**) components. Should we consider the HUD a global service that should be accessible through the Service Locator pattern? The answer should be no, as the HUD only appears during certain parts of the game and should be accessible only by particular components in a specific context. But if we design a custom logging system, we could justify exposing it through a Service Locator pattern, as we might need to log information from anywhere in our code independently of context and scope.

Now that we have gone through the theory, let's get our hands dirty and write a Service Locator pattern to provide access to a logger, an analytics system, and an **advertising network** (**ad network**) provider.

Implementing a Service Locator pattern

We are going to implement a basic Service Locator pattern to expose three specific services, as follows:

- **Logger**: A service that acts as a facade to a centralized logging system
- **Analytics**: A service that sends custom analytical information to a backend to provide insight on player behavior
- **Advertisement**: A service that pulls video **advertisements** (**ads**) from a network and displays them to monetize the game's content at specific moments

We are adding these services to the registry of the Service Locator pattern because of their following characteristics:

- They offer a specific service.
- They need to be accessible from anywhere in the code base.
- They can be mocked or removed without causing any regression in the gameplay code.

As we are going to see from the following code example, implementing a basic Service Locator pattern is a straightforward process. These are the steps we'll take:

1. Let's start by implementing the most important ingredient—the `ServiceLocator` class, as follows:

```
using System;
using System.Collections.Generic;

namespace Chapter.ServiceLocator
{
```

```
        public static class ServiceLocator
        {
            private static readonly
                IDictionary<Type, object> Services =
                    new Dictionary<Type, Object>();

            public static void RegisterService<T>(T service)
            {
                if (!Services.ContainsKey(typeof(T)))
                {
                    Services[typeof(T)] = service;
                }
                else
                {
                    throw new
                        ApplicationException
                        ("Service already registered");
                }
            }

            public static T GetService<T>()
            {
                try
                {
                    return (T) Services[typeof(T)];
                }
                catch
                {
                    throw new
                        ApplicationException
                        ("Requested service not found.");
                }
            }
        }
    }
```

This version of a Service Locator pattern has three primary responsibilities, outlined as follows:

- It manages a registry of services in the form of a `Dictionary`.
- It offers a static function named `RegisterService()` that permits an object to be registered as a service.

- It returns an instance of `service` of a specific type when requested through the `GetService()` function.

It has to take into account that both `RegisterService()` and `GetService()` are static functions, hence they are accessible directly without needing to initialize the `ServiceLocator` class.

The `Services` dictionary holds a list of available services, and we flagged it as `readonly` and `private`; thus, we protect it from being overridden or accessed directly. Instead, a client will have to go through the public methods that the Service Locator pattern exposes to add or get a service.

Now that we have our Service Locator class ready, we can now start implementing some service contracts in the form of interfaces.

2. Our first interface is for the `Logger` service, as illustrated in the following code snippet:

```
namespace Chapter.ServiceLocator
{
    public interface ILoggerService
    {
        void Log(string message);
    }
}
```

3. The next interface is for the `Analytics` service, as we can see here:

```
namespace Chapter.ServiceLocator
{
    public interface IAnalyticsService
    {
        void SendEvent(string eventName);
    }
}
```

4. And lastly, we implement the code for our `Advertisement` service's interface, as follows:

```
namespace Chapter.ServiceLocator
{
    public interface IAdvertisement
    {
        void DisplayAd();
    }
}
```

5. And now, we are going to implement concrete service classes, starting with the `Logger` class. The code to accomplish this is illustrated in the following snippet:

```
using UnityEngine;

namespace Chapter.ServiceLocator
{
    public class Logger: ILoggerService
    {
        public void Log(string message)
        {
            Debug.Log("Logged: " + message);
        }
    }
}
```

6. Next up is the `Analytics` class. Here's the code you'll need to implement this:

```
using UnityEngine;

namespace Chapter.ServiceLocator
{
    public class Analytics : IAnalyticsService
    {
        public void SendEvent(string eventName)
        {
            Debug.Log("Sent: " + eventName);
        }
    }
}
```

7. And lastly, we implement our concrete `Advertisement` service class, as follows:

```
using UnityEngine;

namespace Chapter.ServiceLocator
{
    public class Advertisement : IAdvertisement
    {
        public void DisplayAd()
        {
            Debug.Log("Displaying video advertisement");
        }
    }
}
```

We now have a Service Locator pattern with services that we can register and access from anywhere.

Testing the Service Locator pattern

To test our implementation of the Service Locator pattern, let's write a client class that we will attach as a component to a GameObject in an empty Unity scene, as follows:

```
using UnityEngine;

namespace Chapter.ServiceLocator
{
    public class ClientServiceLocator : MonoBehaviour
    {
        void Start() {
            RegisterServices();
        }

        private void RegisterServices() {
            ILoggerService logger = new Logger();
            ServiceLocator.RegisterService(logger);

            IAnalyticsService analytics = new Analytics();
            ServiceLocator.RegisterService(analytics);

            IAdvertisement advertisement = new Advertisement();
            ServiceLocator.RegisterService(advertisement);
        }

        void OnGUI()
        {
            GUILayout.Label("Review output in the console:");

            if (GUILayout.Button("Log Event")) {
                ILoggerService logger =
                    ServiceLocator.GetService<ILoggerService>();
                logger.Log("Hello World!");
            }

            if (GUILayout.Button("Send Analytics")) {
                IAnalyticsService analytics =
                    ServiceLocator.GetService<IAnalyticsService>();
                analytics.SendEvent("Hello World!");
            }

            if (GUILayout.Button("Display Advertisement")) {
                IAdvertisement advertisement =
                    ServiceLocator.GetService<IAdvertisement>();
                advertisement.DisplayAd();
            }
        }
```

```
        }
    }
```

It's important to note that in an actual implementation of the Service Locator pattern in a Unity project, we would register our services as early as possible during our game's lifespan so that they are made available at all times—for example, this task could be given to a `GameManager` object that we initialized in the first scene of our project. If we know that the scene and the Game Manager object will always be loaded when the player starts the game, we are sure the registry of the Service Locator pattern will be updated before clients start requesting access to services.

A key benefit of our approach is that we are registering services by referring to their interfaces, which means that at the moment we register services, we can choose which concrete implementation to use. Thus, we could easily have mock versions of each service running in a debug build. Furthermore, this approach will avoid adding noise to the logs and analytics during the **quality assurance** (**QA**) phase.

Hence, this is one of the cool features of this pattern; you can dynamically inject various service versions depending on the runtime context.

One of the main benefits of using Unity as your engine is that it offers a range of integrated services, including ads and analytics services, therefore most of the time, you will not have to implement them by hand. You can read about the range of available Unity services at the following link: `https://docs.unity3d.com/Manual/UnityServices.html`.

Reviewing alternative solutions

If you are having issues with the management of dependencies in your code base, it might be time to start investigating the use of a DI framework. DI is a technique in which an object receives dependencies it needs through an "injection mechanism." There are several ways an object can receive its dependencies—through the constructor, a setter, or even with an interface that provides an injector method.

The best way to start using DI in a structured manner is through a framework because this gives you a helping hand in managing complex relationships between objects, the initialization process, and the lifespan of dependencies. In conclusion, you should start considering using a DI framework when you see tight coupling between classes and when their dependencies are becoming a bottleneck to writing consistent, testable, and maintainable code.

Extenject is a free DI framework for Unity that can be downloaded from the Asset Store:

`https://assetstore.unity.com/packages/tools/utilities/extenject-dependency-injection-ioc-157735`

Summary

In this chapter, we reviewed the Service Locator pattern. This pattern is a simple solution to resolving a recurring challenge of managing dependencies between objects relying on services (functionalities) that other objects offer. In its simplest form, the Service Locator pattern decouples the relationship between a client (requester) and a service provider.

We have arrived at the end of our journey, as this is the last chapter of the book. We hope you enjoyed the content of each chapter. Please remember that the concepts presented throughout this book are just introductions, not the final word on the subject matter. There's a lot more to learn about design patterns, Unity, and game development—so much that we can't define it in a single book. Thus, we encourage you to continue learning, take what we reviewed together in each chapter, and make it better because there's always room for improvement.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

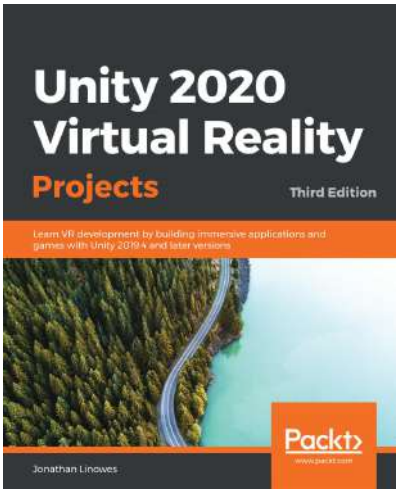
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

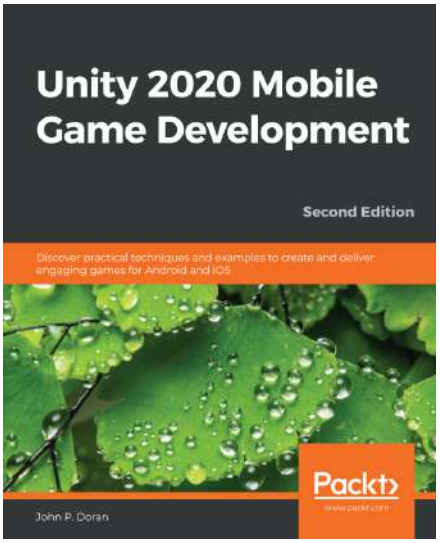
If you enjoyed this book, you may be interested in these other books by Packt:



Unity 2020 Virtual Reality Projects - Third Edition
Jonathan Linowes

ISBN: 978-1-83921-733-3

- Understand the current state of virtual reality and VR consumer products
- Get started with Unity by building a simple diorama scene using Unity Editor and imported assets
- Configure your Unity VR projects to run on VR platforms such as Oculus, SteamVR, and Windows immersive MR
- Design and build a VR storytelling animation with a soundtrack and timelines
- Implement an audio fireball game using game physics and particle systems
- Use various software patterns to design Unity events and interactable components
- Discover best practices for lighting, rendering, and post-processing



Unity 2020 Mobile Game Development - Second Edition
John P. Doran

ISBN: 978-1-83898-733-6

- Design responsive user interfaces for your mobile games
- Detect collisions, receive user input, and create player movements for your mobile games
- Create interesting gameplay elements using inputs from your mobile device
- Explore the mobile notification package in Unity game engine to keep players engaged
- Create interactive and visually appealing content for Android and iOS devices
- Monetize your game projects using Unity Ads and in-app purchases

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Game Development Patterns with Unity 2021*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

F

G

H

L

O

V

W