

# 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

УХ ТЫ  
электронная книга [www.wowebook.org](http://www.wowebook.org)



# Шаблоны разработки игр с Unity

## 2021

### Второе издание

Изучите практическую разработку игр с использованием шаблонов проектирования программного обеспечения и лучших практик в Unity и C#.

Дэвид Барон

Packt

БИРМИНГЕМ - МУМБАЙ

УХ ТЫ

электронная книга [www.wowebook.org](http://www.wowebook.org)

# Шаблоны разработки игр с Unity 2021

## Второе издание

Copyright © 2021 Packt Publishing.

Все права защищены. Никакая часть этой книги не может быть воспроизведена, скопирована в поисковой системе или передана в любой форме и любыми средствами без предварительного письменного разрешения издателя, за исключением случаев кратких цитат, включенных в критические статьи или обзоры.

При подготовке этой книги были приложены все усилия, чтобы обеспечить точность предоставленной информации. Однако информация, содержащаяся в этой книге, продается без каких-либо гарантий, явных или подразумеваемых. Ни автор, ни компания Packt Publishing, ни ее дилеры и дистрибуторы не несут ответственности за любой ущерб, причиненный или предположительно причиненный этой книгой, прямым или косвенным.

Packt Publishing старалась представить информацию о товарных знаках всех компаний и продуктов, упомянутых в этой книге, используя соответствующие прописные буквы. Однако Packt Publishing не может гарантировать точность этой информации.

Менеджер группы продуктов: Эшвин Нэйр

Менеджер по издательству продукту: Паван Рамчандани

Старший редактор: Киган Карнейро

Редактор по разработке контента: Дивья Виджаян

Технический редактор: Сараб Кадре

Редактор копий: Safis Editing

Координатор проекта: Мантан Патель

Корректор: Сафис Редактирование

Индексатор: Субалакши Говиндуан.

Художник-постановщик: Апарна Бхагат

Первая публикация: март 2019 г.

Второе издание: июль 2021 г.

Код производства: 1290721

Опубликовано Packt Publishing Ltd.

Livery Place 35

Livery Street

Birmingham B3

2PB, Великобритания.

ISBN 978-1-80020-081-4

[www.packt.com](http://www.packt.com)

Эта книга посвящается моей матери Кате Галаноза ее постоянные слова  
ободрения и поддержки, а также зато, что она всегда помогала мне в трудные времена.

— Дэвид Бэрон

# Авторы

## Об авторе

Дэвид Барон — разработчик игр с более чем 15-летним опытом работы в отрасли. Он работал в некоторых известных студиях AAA, мобильных и инди-игр в Монреале, Канада.

Его набор навыков включает программирование, дизайн и 3D-искусство. Как программист, он работал над различными играми для разных платформ, включая виртуальную реальность, мобильные устройства и консоли.

Прежде всего я хотел бы поблагодарить своих друзей из постиндустриальной поддержки, терпение и поддержку на протяжении всего долгого процесса написания этой книги. Я также хочу выразить свою благодарность следующим людям, которые предоставили мне ценные отзывы во время написания этой книги: Николя Эйперту, Гийому Леруа, Дэйву Сируа и Фредерику Левеску.

## Орец ензентах

Лукас Бертолини имеет более чем 10-летний опыт работы разработчиком программного обеспечения для видеигр. Он работал над тремя крупными проектами: для Pollux Ltd. (Гонконг) в качестве разработчика и дизайнера игр; для Shell Games (Питтсбург, США), куда он переехал и работал разработчиком до завершения проекта; и для Глобанта как разработчика.

Он работал в сфере технологического образования более 5 лет и преподавал различные курсы по программированию. Он являлся соучредителем NGA и Bytenarchy Studios, компаний по разработке цифровых услуг, ис пользующих Unity в качестве основной технологии.

Лукас написал книги по «Практическая разработка игр без программирования», доступную на Packt.

Марк Бонас орос в конце 80-х в семье, где с юных лет были компьютеры. В середине 90-х Марк начал с обирать собственные компьютеры. Его интерес к разработке игр начался, когда он ис пользовал Softimage на машине Silicon Graphics в старшей школе. Навыки программирования Марка возросли в 2000-х годах, когда он поступил в колледж, ис пользовал C++ для создания бизнес-приложений и работал в небольшой инди-компании, занимающейся играми, ис пользуя мультиплатформенный 3D-игровой движок, который все еще находился в обращении, под название Intrinsic Alchemy. В 2006 году Марк основал игруюю студию чтобы вместе с со своими одноклассниками научиться создавать мобильные игры для телефонов с поддержкой Java 2 Micro Edition для Sony Ericsson и Nokia. В 2010 году он окончил Университет Конкордия по специальности «программное обеспечение» и работал разработчиком Unity в Affordance Studios, производя щей обучающие игры.

Я хотел бы поблагодарить Дэвида за все возможности, которые он предложил, и для меня большая честь стать техническим редактором его последней книги.

# Оглавление

Предисловие	1
<b>Разделы 1: Основы</b>	
Глава 1: Прежде чем мы начнем	8
Заметки о новом издании	8
Философия книг и что такое	9
шаблоны проектирования?	10
Какие темы не освещены в этой книге?	10
Краткое описание	11
игрового	12
Глава 2. Дизайн-документ игры Дизайн-	13
документ Обзор игры	14
Уникальные	14
преимущества	14
Минимальные	15
требования	15
Краткое описание	16
игры Цели	17
игры	17
Правила игры	18
Игровой цикл Игровая среда Камера, управление, персонаж (3Cs)	18
Камера	18
Персонаж	19
Описание персонажа	19
Показатели	20
персонажа	20
Состояния	21
персонажа Контроллер	21
Игровые	22
ингредиенты	22
Супербайки	23
Пикапы Прягательства	23
Оружие Игровые	25
системы	25
Игровое меню	26
Игровой интерфейс Краткое описание Дальнейшее чтение	26
Глава 3. Краткое введение в программирование в Unity	27
Чтобы уже должны знать	28

## Оглавление

---

Возможности языка C#	28
Возможности движка Unity	32
Краткое содержание	34
дальнейшее чтение	34
<b>Раздел 2: Основные шаблоны</b>	
Глава 4. Реализация игрового менеджера с помощью Singleton	36
Технические требования	37
Понимание шаблона Singleton	37
Преимущества и недостатки	39
Проектирование игрового менеджера	39
Реализация игрового менеджера	40
Тестирование игрового менеджера	45
Краткое содержание	47
Глава 5: Управление состояниями с помощью шаблона состояния	48
Технические требования	49
Обзор шаблона «Состояние»	49
Определение состояний персонажа	51
Реализация шаблона State	52
Реализация шаблона State	53
Тестирование реализации шаблона состояния	58
Преимущества и недостатки шаблона State	59
Рассмотрение альтернативных решений	61
Краткое содержание	62
Глава 6. Управление игровыми событиями с помощью шин событий	63
Технические требования	64
Понимание шаблона шины событий	64
Преимущества и недостатки шаблона Event Bus	66
Когда использовать шину событий	67
Управление глобальными игровыми событиями	67
Реализация шин игровых событий	68
Тестирование игрового автобуса	70
Обзор реализации шины событий	75
Рассмотрение некоторых альтернативных решений	76
Краткое содержание	76
Глава 7. Реализация системы воспроизведения с помощью шаблона команды	77
Технические требования	78
Понимание шаблона команды	78
Преимущества и недостатки шаблона «Команда»	81
Когда использовать шаблон команды	81
Проектирование системы повторов	82

## Оглавление

---

Реализация системы повторов	84
Реализация системы повторов	84
Тестирование системы повторов	88
Обзор реализации	92
Расмотрение альтернативных решений	93
Краткое содержание	93
<b>Глава 8. Оптимизация с помощью шаблона пул объектов</b>	94
Технические требования	95
Понимание шаблона пул объектов	95
Преимущества и недостатки шаблона пул объектов	96
Как да использовать шаблон пул объектов	97
Реализация шаблона пул объектов	98
Шаг по реализации шаблона пул объектов	98
Тестирование реализации пул объектов	103
Обзор реализации пул объектов	105
Расмотрение альтернативных решений	105
Краткое содержание	105
<b>Глава 9: Развделение компонентов с помощью шаблона наблюдателя</b>	106
Технические требования	107
Понимание шаблона наблюдателя	107
Преимущества и недостатки шаблона Observer	108
Как да использовать шаблон Observer	109
Развделение основных компонентов с помощью шаблона Observer	109
Реализация шаблона наблюдателя	111
Тестирование реализации шаблона Observer	117
Расмотрение альтернативных решений	119
Краткое содержание	119
<b>Глава 10: Реализация усилений с помощью шаблона пос етителя</b>	120
Технические требования	121
Понимание шаблона «Пос етитель»	121
Преимущества и недостатки шаблона «Пос етитель»	123
Проектирование механизма включения питания	124
Реализация механизма включения питания	125
Реализация с системы включения питания	125
Тестирование реализации с системы включения питания	131
Обзор реализации с системы включения питания	133
Краткое содержание	134
<b>Глава 11. Реализация дрона с помощью шаблона стратегии</b>	135
Технические требования	136
Понимание шаблона стратегии	136
Преимущества и недостатки паттерна «Стратегия»	137
Как да использовать паттерн «Стратегия»	138

## Оглавление

---

Проектирование вражеского дрона	139
Реализация вражеского дрона	141
Шаг по внедрению вражеского дрона	141
Тестирование реализации вражеского дрона	146
Обзор реализации вражеских дронов	148
Рассмотрение альтернативных решений	148
Краткое содержание	149
<b>Глава 12. Использование диктора для реализации системы оружия</b>	150
Технические требования	150
Понимание шаблона Диктор	151
Преимущества и недостатки шаблона «Диктор»	152
Как использовать шаблон «Диктор»	153
Проектирование системы вооружения	154
Реализация системы вооружения	155
Реализация системы вооружения	155
Тестирование системы вооружения	162
Обзор системы вооружения	165
Рассмотрение альтернативных решений	166
Краткое содержание	166
<b>Глава 13: Реализация редактора уровней с пространственным разделением</b>	167
Технические требования	168
Понимание шаблона пространственного разделения	168
Как использовать шаблон пространственного разделения	170
Проектирование редактора уровней	170
Реализация редактора уровней	173
Шаг по реализации редактора уровней	173
Использование редактора уровней	179
Обзор реализации редактора уровней	179
Рассмотрение альтернативных решений	179
Краткое содержание	180
<b>Раздел 3: Альтернативные шаблоны</b>	
<b>Глава 14: Адаптация с системой помощника адаптера</b>	182
Технические требования	183
Понимание шаблона адаптера	183
Преимущества и недостатки шаблона адаптера	185
Как использовать шаблон адаптера	186
Реализация шаблона адаптера	187
Реализация шаблона адаптера	187
Тестирование реализации шаблона адаптера	190
Краткое содержание	191
<b>Глава 15: Скрытие сложности с помощью зора фасада</b>	192

## Оглавление

---

Технические требования	192
Понимание шаблона Фасад	193
Преимущества и недостатки	194
Проектирование велосипедного двигателя	195
Реализация велосипедного двигателя	196
Тестирование фасада двигателя	201
Рассмотрение альтернативных решений	202
Краткое содержание	202
<b>Глава 16. Управление зависимостями с помощью шаблона локатора сервисов</b>	<b>203</b>
Технические требования	204
Понимание шаблона локатора сервисов	204
Преимущества и недостатки шаблона Service Locator	206
Как использовать шаблон Service Locator	206
Реализация шаблона локатора сервисов	207
Тестирование шаблона Service Locator	211
Рассмотрение альтернативных решений	212
Краткое содержание	213
<b>О пакете</b>	<b>214</b>
<u>Другие книги, которые могут вам понравиться</u>	215
<b>Индекс</b>	<b>218</b>

## Предисловие

Первые принципы Кларисы: просите Марка Аврелия.

«О каждой конкретной вещи спрашивайте: что она сама по себе? Какова ее природа?»

- Ганнибал Лектор

Предыдущая глава взята из одного из моих любимых фильмов и резюмирует мой подход к обучению. Проработав более десяти лет в игровой индустрии, я обнаружил, что единственный правильный способ обладать сложной системой — это разбить ее на самые основные компоненты. Другими словами, я пытаюсь понять основные ингредиенты, прежде чем освоить окончательную форму. На протяжении всей этой книги вы увидите, что я использую прошенный, но контекстуальный подход к представлению каждого шаблона.

Цель состоит не в том, чтобы упрощать предмет, а в том, чтобы учиться, изолируя основные концепции, лежащие в основе каждого шаблона проектирования, чтобы мы могли наблюдать за ними и изучать их тонкости. Я научился этому подходу в игровой индустрии, работая дизайнером и программистом. Мы часто создаем компоненты и системы для нашей игры на изолированных уровнях, которые мы называем противными залами. Мы тратили недели на iteration, тестирование и корректировку каждого компонента нашей игры по отдельности, пока не поняли, как заставить их работать как единое целое. Я написал эту книгу в соответствии со своим подходом к разработке игр, чтобы вы, как читатель, могли погрузиться в предмет, одновременно приобретая некоторые хорошие привычки, которые помогут вам в вашей карьере.

Однако важно также отметить, что содержание этой книги не является окончательным справочником по шаблонам в Unity. Это всего лишь введение в предмет, а не конечная цель процесса обучения. Я не позиционирую себя как главного эксперта и не хочу, чтобы мои слова стали единственной сферой интересов среди разработчиков. Я всего лишь разработчик, пытающийся найти легкий способ использования стандартных шаблонов проектирования программного обеспечения в Unity, и хочу поделиться тем, что обнаружил. Поэтому, как читатель, я призываю вас критиковать, исследовать, адаптировать и улучшать все, что представлено в этой книге.

## Для кого эта книга

При написании этой книги я определился с конкретной ментальной моделью моей целевой аудитории по той основной причине, что практически невозможно написать книгу о разработке игр и удовлетворить каждый потенциальный тип читателей, прежде всего потому, что разработка игр — это разнообразная отрасль, и существует множество различных платформ и жанров, каждый со своими специфическими характеристиками, чтобы не мог уместить их в одной книге. Поэтому я решил сократить контент на определенной аудитории, которую могу описать следующим образом:

Целевая аудитория — игровые программисты, которые в настоящее время работают над мобильным или инди-игровым проектом на движке Unity и находятся в процессе рефакторинга своего кода, чтобы сделать его более удобным в оправождении и масштабируемости. Читатель должен иметь базовое представление о Unity и языке C#.

Старшие программисты, работающие над крупномасштабными играми AAA или MMO, могут счесть конкретные примеры в этой книге ограниченными по сравнению с архитектурными проблемами, с которыми они обычно сталкиваются ежедневно. Однако, с другой стороны, с одержание этой книги может предложить другой взгляд на использование шаблонов проектирования в Unity. Так что с мело пропускайте любую лаву, если вы уже знакомы с теорией и хотите увидеть, как я реализовал конкретный шаблон.

## О чем эта книга

Глава 1 «Прежде чем мы начнем» представляет собой краткое введение в содержание этой книги.

Глава 2 «Документ по дизайну игры», представляет проектный документ, лежащий в основе полностью игрового прототипа очной игры.

Глава 3 «Краткий курс программирования в Unity» рассмотривает некоторые базовые концепции C# и Unity.

Глава 4 «Реализация игрового менеджера с помощью Singleton» описывает реализацию глобально доступного игрового менеджера с использованием пресловутого шаблона Singleton.

В главе 5 «Управление состояниями персонажа с помощью шаблона состояния» рассматривается классический шаблон «Состояние» и способы инкапсуляции поведения персонажа с учетом состояния.

Глава 6 «Управление игровыми событиями с помощью шаблона команд» описывает новые принципы шаблона «Шина событий» и способы его использования для управления глобальными играми с событиями.

В главе 7 «Реализация системы повторов с помощью шаблона команд» рассматривается, как использовать шаблон «Команда» для создания системы повторов для очной игры.

---

Предисловие

---

Глава 8 « Оптимизация с помощью шаблона пул объектов » описывает, как использовать с обобщенной реализацией шаблона пул объектов в Unity для оптимизации производительности.

В главе 9 « Разворачивание компонентов с помощью шаблона наблюдателя » рассказывается, как отделить основные компоненты от наблюдателя .

Глава 10 « Реализация усилений с помощью шаблона посредника » объясняет, как использовать шаблон «Посредник» для реализации настраиваемой игровой механики и усилий .

В главе 11 « Реализация дрона с помощью шаблона стратегии » рассказывается, как динамически назначать поведение атаки вражеским дронам с помощью шаблона стратегии .

Глава 12 « Использование декоратора для реализации системы оружия » объясняет, как использовать шаблон «Декоратор» в качестве новой системы крепления оружия .

Глава 13 « Реализация редактора уровней с помощью Spatial Partition » описывает, как использовать общие концепции Spatial Partition для создания редактора уровней для гоночной игры .

Глава 14 « Адаптация с системой с помощью адаптера » описывает новые шаблона адаптера и способа его использования для адаптации с сторонней библиотекой для повторного использования в новой системе .

В главе 15 « Скрытие сложности с помощью шаблона фасада » шаблон «Фасад» используется для скрытия сложности и создания чистого внешнего интерфейса для сложного расположения взаимодействующих компонентов .

В главе 16 « Управление зависимостями с помощью шаблона локатора с службой » рассказывается о новом шаблоне «Локатор с службой» и способа его использования для реализации системы, которая позволяет регистрировать и определять места расположения определенных служб во время выполнения .

## Чтобы получить максимальную пользу от этой книги

Чтобы получить максимальную пользу от этой книги, вам необходимо иметь базовое представление о движке Unity. Вам также необходимо знать C# и иметь общие знания объектно-ориентированного программирования . Если вы хотите воспроизвести код, представленный в следующих главах , вам необходимо загрузить на свой компьютер исходники Unity .

Вы можете получить самую последнюю версию Unity по следующей ссылке: <https://unity3d.com/get-unity/download> .

Системные требования для запуска Unity можно найти здесь: <https://docs.unity3d.com/Manual/system-requirements.html> .

## Предисловие

Если вы хотите загрузить исходный код, доступный в нашем репозитории GitHub, вам понадобится клиент Git; мы рекомендуем GitHub Desktop, поскольку его проще всего использовать. Скачать его можно по следующей ссылке: <https://desktop.github.com/>.

Помимо этих инструментов, для запуска примеров кода, представленных в этой книге, не требуется загрузка других библиотек или зависимостей.

## Загрузите файлы примеров кода

Вы можете загрузить файлы примеров кода для этой книги с GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition>. В случае обновления кода он будет обновлен в соответствующем репозитории GitHub.

У нас также есть другие пакеты кода из нашего каталога книг и видео, доступных на <https://github.com/PacktPublishing/>. Проверь их!

## Код в действии

Видео «Код в действии» для этой книги можно посмотреть по адресу <https://bit.ly/2UNCZX1>.

## Загрузите цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями экрана и диаграмм, использованных в этой книге. Вы можете скачать его здесь: [https://static.packt-cdn.com/downloads/9781800200814\\_ColorImages.pdf](https://static.packt-cdn.com/downloads/9781800200814_ColorImages.pdf).

## Используемые конвенции

В этой книге используется ряд текстовых соглашений.

**CodeInText:** указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и дескрипторы Twitter. Вот пример: «Этот класс инициализирует объект Context и состояния, а также запускает изменения состояний».

Блок кода задается следующим образом:

```
protected void Chapter.State {
    // общедоступный интерфейс IBikeState {
        void Handle(контроллер BikeController);
```

Предисловие

---

```
    }  
}
```

Когда мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
просмотр трансформированный Chapter.State {  
  
    публичное перечисление Направление  
    {  
        Слева = -1,  
        Справа = 1  
    }  
}
```

Жирный шрифт: обозначает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах показываются в тексте вот так. Вот пример: «Эти объекты могут публиковать определенные типы событий, объединенные шиной событий подписчикам».



Предупреждения или важные примечания выглядят следующим образом.



Советы и подсказки выглядят так.

## Связаться

Обратная связь от наших читателей всегда приветствуется.

Общий отзыв: Если у вас есть вопросы по какому-либо аспекту этой книги, укажите название книги в теме сообщения и напишите нам по адресу [customercare@packtpub.com](mailto:customercare@packtpub.com).

Опечатка: Мы приложили все усилия, чтобы обеспечить точность нашего контента, ошибки все же случаются. Если вы нашли ошибку в этой книге, мы будем признательны, если вы сообщите нам об этом. Пожалуйста, посетите [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata), выберите свою книгу, щелкните ссылку «Форма отправки исправлений» и введите данные.

---

Предисловие

---

Пиратство. Если вы встретите в Интернете незаконные копии наших работ в любой форме, мы будем признательны, если вы предоставите нам адрес или название веб-сайта. Пожалуйста, свяжитесь с нами по адресу [Copyright@packt.com](mailto:Copyright@packt.com) и укажите ссылку на материал.

Если вы заинтересованы в том, чтобы стать автором: если есть тема, в которой вы разбираетесь, и вы заинтересованы в написании или написании книги, поговорите с [авторы.packtpub.com](http://authors.packtpub.com).

## Поделитесь с вашими мыслами

После того, как вы прочитали «Шаблоны разработки игр с Unity 2021», мы будем рады услышать ваши мысли! Нажмите здесь, чтобы перейти прямо на страницу обзора Amazon для этой книги и поделиться своим мнением.

Ваш отзыв важен для нас и технического сообщества и поможет нам убедиться, что мы предоставляем контент отличного качества.

# Разделы 1: Основы

В этом разделе книг и мы рассмотрим некоторые основы проектирования и программирования, прежде чем перейти к практическим разделам книги.

Этот раздел состоит из следующих глав:

- Глава 1. Прежде чем мы начнем
- Глава 2. Документ по дизайну игры
- Глава 3. Краткое введение в программирование в Unity

# 1

## Прежде чем мы начнем

Добро пожаловать во второе издание книг и «Практические шаблоны разработки игр с помощью Unity»; это издание предстavляется с обойнею простотой переработку предыдущей версии, а полное обновление оригинальной книги. После выхода первой версии издания мне по частотивилось получить множество конструктивных отзывов, которые вдохновили меня на улучшение структуры нового издания. Как мы увидим в следующих разделах, эта книга акцентируется на «практическом» аспекте названия; другими словами, мы с обирайемся запачкать руки и поработать над реализацией систем и функций для полноценного рабочего прототипа игры с шаблонами проектирования. Этот новый подход к структуре книги будет более осозаемым и более привычным. Работать над игрой рабочей версии будет легче, чем над лучшими примерами кода. Итак, прежде чем мы начнем, в следующих разделах я установлю конкретные параметры содержания книги и общего подхода.

Давайте быстрее смотрим темы, которые мы увидим в этой главе, а именно:

- Заметки о новом издании Философия
- книг и что такое шаблоны
- проектирования?
- Какие темы не освещены в этой книге?
- Игровой проект

## Заметки о новом издании

Как уже упоминалось, я полностью переработал это издание, основываясь на отзывах читателей о предыдущей версии, и поэтому решил вырезать из последней часть содержания, которую читателисличили тривиальной, например следующее:

- Главы игрового цикла и шаблонов обновления. Эти главы с лишком сосредоточены на теории и не соответствуют «практическому» подходу книги.
- Глава об антипаттернах. Антипаттерны — это сложная и глубокая тема, которая заслуживает отдельной книги, чтобы отдать ей должное.

Конечная цель этого издания — не сократить содержание, а переопределить книгу, сосредоточив внимание на практическом использовании шаблонов проектирования программного обеспечения при разработке игр для создания полноценного проекта. Другими словами, в отличие от первого издания, в котором я ис用车ал под собой, заключающийся в представлении каждого шаблона проектирования изолированно и с помощью автономного примера кода, на этот раз мы будем использовать их вместе.

В это издание я добавил несколько глав, отсутствовавших в предыдущей версии, например следующие:

- Документ по игровому дизайну. Начало нового игрового проекта часто начинается с написания GDD. GDD — это документ, который поможет нам понять замысел разработки игровых системщиков, которые мы будем сдавать на протяжении всей книги.
- Краткое введение в программирование в Unity. На протяжении всей книги мы будем использовать несколько передовых концепций движка Unity, функции C# и методы объектно-ориентированного программирования (ООП). В этой главе мы уделим время их рассмотрению чтобы создать общую базу знаний.

## Философия книги

Эта книга не является технической библией или высшим авторитетом в области использования шаблонов проектирования в Unity. Лучше всего можно описать как руководство, наполненное дизайнерскими предложениями для решения некоторых проблем программирования игр в Unity. Примеры кода, включенные в каждую главу, не являются безупречными реализациями, поскольку искустство проектирования и программирования — это непрерывный процесс совершенствования, и поэтому основная цель этой книги — познакомить вас с потенциальными решениями и вдохновить вас на поиск лучших.



Я всегда рекомендую кратко как минимум два источника информации по любому техническому вопросу, особенно если имеются шаблонов проектирования. Очень важно избегать чрезмерного влияния одной точки зрения на слишкомый предмет до такой степени, что она становится для меня, а не знанием.

## Что такое шаблоны проектирования?

Для новичков в программировании шаблоны проектирования могут оказаться новой концепцией. Самый простой способ обьяснить шаблоны проектирования состоит в том, что они представляют собой многократно используемые решения распространенных проблем разработки программного обеспечения. Архитектор по имени Крис Тофер Александер придумал понятие шаблонов проектирования для описания идей многократного использования. В конце 1980-х годов, вдохновленные этой концепцией, инженеры-программисты начали экспериментировать с применением концепций многократного использования шаблонов проектирования в разработке программного обеспечения, и эти годы на эту тему было написано несолько книг, таких как классические «Шаблоны проектирования: элементы многократного использования объектно-ориентированных систем». Программное обеспечение так называемой «Банды четырех».

Но в этой книге я не буду освещать академическую сторону шаблонов проектирования программного обеспечения, а вместо этого я сосредоточусь на их практическом использовании для программирования игровых механик и систем в Unity. В каждой главе я буду представлять повторяющуюся проблему программирования игр и предлагать ее решение с использованием определенного шаблона проектирования, адаптированного для интерфейса прикладного программирования (API) Unity.

## Какие темы не освещены в этой книге?

Программирование игр имеет множество аспектов, и одна книга не может охватить их все с той глубиной, которой они заслуживают. Этакий аспект конкретной темы: шаблонам проектирования и движку Unity. Итак, если вы только начинаете свой путь к карьере профессионального программиста игр, эта книга будет недостаточно для завершения вашего образования. Но, к счастью некоторые очень талантливые люди в нашей отрасли нашли время написать очень специализированные книги по новым темам разработки игр. Я рекомендую вам, кто хочет присоединиться к игровой индустрии, прочитать каждый из следующих справочников:

- Физическое программирование: обнаружение с толкованием в реальном времени, Крис Тер Эриксон
- Программирование движка Game Engine Architecture, Джейсон Грегори
- Трехмерное (3D) программирование: математика для программирования 3D-игр и компьютерной графики, Эрик Лендель
- Программирование искусственного интеллекта (ИИ): программирование игр о ИИ на примере, Матем. Бакленд

Я сосредоточил содержание этой книги на конкретном аспекте программирования игр, но на протяжении всей книги я буду упоминать концепции из других областей разработки игр. Итак, если вы чувствуете, что незнакомы с некоторыми из упомянутых тем, найдите время, чтобы изучить их более подробно; время, потраченное на исследования, сделает вас лучшим программистом игр.

## Игровой проект

На протяжении всей этой книги мы будем постоянно работать над одним примером игрового проекта.

Рабочее название игры — Edge Racer. Как следует из названия, это очная игра, если быть более конкретным, то это футуристическая очная игра, в которой игра управляется высокоскоростными мотоциклами. Мы рассмотрим новые концепции игры более подробно в главе 2 «Документ по дизайну игры». Но прежде чем продолжить, я хотел бы перечислить причины, по которым я решил выбрать очную игру вместо другого типа игры, например ролевой игры (RPG), именно:

- ПРОСТОТА: очные игры имеют простую идею — доберитесь до финиша как можно быстрее и не разбившись. Поскольку эта книга посвящена игровому дизайну, а игровому программированию мне нужен был простой тип игры, который позволил бы нам сосредоточиться на изучении шаблонов проектирования программного обеспечения и не увязнуть в деталях реализации сложной игровой механики.
- ВСЕЛЬЕ: я работал над различными играми самых разных жанров и всегда обнаруживал, что разрабатывать очные игры интереснее всего, потому что их приятно тестировать. В очных играх вы можете быстро перейти к определенным частям игры и быстро воспроизводить ошибки или тестировать новые функции. В отличие от других игр с глубокой игровой механикой и большими картами, таких как ролевые игры, очные игры обычно отлаживаются быстрее.
- ПРОИЗВОДИТЕЛЬНОСТЬ: основная задача программирования очной игры — поддержание постоянной частоты кадров при добавлении новых функций и контента. Итак, я считаю, что работа над очными играми заставляет вас поддерживать хорошие привычки разработчика игр, всегда следя за тем, как быстро работает ваш код, а не просто делая его более читабельным.
- ЛИЧНОЕ: у меня есть личная причина, по которой я выбрал очную игру: это мой любимый жанр. Я люблю играть в очные игры и люблю их создавать.

В заключение, игровая индустрия производит различные продукты во многих жанрах и поджанрах, но очная игра — это хороший ориентир для нас, чтобы начать изучать шаблоны проектирования в Unity, потому что это простой контекст, заставляющий нас следить за код чистый и быстрый.

## Краткое сведение

В этой главе мы рассмотрели структуру книги, чтобы начать с четкого понимания ее содержания и цели. Ключевой вывод заключается в том, что мы будем использовать шаблоны проектирования для создания механизики и систем новой игровой игры под названием Edge Racer.

В следующей главе мы рассмотрим GDD, чтобы иметь четкое представление об игровом проекте, над которым мы будем работать в следующих главах. Я бы не советовал пропускать его, потому что если вы хотите больше узнать об игровом проекте, прежде чем приступить к написанию кода, поскольку это помогает понять, как отдельные части соответствуют целому.

# 2

## Документ дизайна игры

Прежде чем мы начнем вводить одну строку кода, нам необходимо выполнить важный шаг в цикле разработки игры — создание документа дизайна игры (GDD). GDD — это план всего нашего проекта: его основная цель — изложить на бумаге общее видение особых концепций нашей игры и служить руководством для многочленной команды разработчиков на трудном пути длительного производственного цикла.

GDD может включать подробное описание следующих элементов:

- Список особых визуальных, анимационных и звуковых элементов.
- Синопсис, биографии персонажей и повествовательные структуры.
- Материалы маркетинговых исследований и стратегии монетизации
- Описания и диаграммы, иллюстрирующие системы и механизмы.

В большинстве случаев за написание и поддержку GDD отвечает главный дизайнер, а реализацию описанных систем и механизмов в документе отвечает программисты. Поэтому программы и дизайнеры должны давать друг другу обратную связь на протяжении всего процесса формирования GDD. Если обе стороны не могут работать вместе, главный дизайнер напишет системы, которые невозможно реализовать в разумные сроки. Или программы будут тратить время на написание кода для игровых систем, которые имеют дефекты в своей общей конструкции.

Учитывая, что эта книга посвящена программированию игрому дизайну, я предстаю проще и кратко версия GDD для игрового проекта, над которым мы будем работать на протяжении всей книги. Раздел документов также будет включать несколько советов и примечаний для тех, кто не знаком с процессом анализа GDD.



Для краткости не упомяну об участии художников, аниматоров и звукоинженеров в процессе составления GDD. Но их участие необходимо, потому что им нужно знать, сколько активов им нужно будет произвести, чтобы реализовать общее видение игры.

В этой главе мы рассмотрим следующие темы:

- Обзор краткого описания и базовой механики игры.
- Описание новых игровых объектов и целей.
- Список ингредиентов и систем игры.

## Проектный документ

Следующий документ по дизайну игры разделен на части, представляющие отдельные точки интереса в общем проекте. Например, раздел с инструкциями может представлять интерес для тех, кто работает над элементами повествования игры. В то же время разделы о минимальных требованиях и системах ориентированы на программистов. Но какой бы ни была наша специализация, хороший практик обязательно прочитает всю проектную документацию и получит полную картину линейной модели проекта, прежде чем приступить к нему.

## Обзор игры

Blade Racer (рабочее название) — футуристическая аркадная головная игра, в которой игрок пытается управлять мотоциклом с турбонаддувом, преодолевая полосу препятствий, чтобы добраться до финиша с наибольшим количеством очков. Головная трасировка на основе железнодорожной системы, в которой игрок может перемещать транспортное средство с одного рельса на другой, чтобы избежать препятствий. Повсему систматреков горючает предметы на стратегических позициях, которые награждают игроков коростью или щитом. Только игроки с невероятно острыми рефлексами могут пересечь финишную линию быстрее всех и не повредить свой мотоцикл.



Рабочее название — это временное название проекта, используемое во время производства. Оно не является окончательным и может быть изменено, когда проект будет готов к выпуску.

## Уникальные преимущества

Ниже приведены потенциальные уникальные преимущества продажи:

- Международная таблица лидеров, с помощью которой игроки со всего мира могут соревноваться за свое имя в верхней части списка.
- Самая сложная головная игра на рынке
- В игре можно играть на любом устройстве: от мобильного до ПК.



Если вы работаете с издателем, в GDD полезно иметь уникальные преимущества. Это демонстрирует, что у вас есть видение конечного продукта и того, как вы будете его продавать. Обратите внимание, что приведенные выше примеры являются окончательными.

## Минимальные требования

Ниже приведены минимальные требования для текущих целевых платформ:

Для мобильных устройств:

- ОС: Андроид 10
- Модель: Samsung S9

Для ПК:

- ОС: Windows 10
- Видеокарта: Nvidia GTX 980

Мы будем поддерживать эквивалентную карту ATI того же поколения, что и указанная карта GTX.



Определение минимальных требований к платформе в самом начале проекта очень полезно для программистов игр. Платформа с различными ресурсами потребует большей оптимизации визуальных ресурсов и кодовой базы, чтобы игра работала с постоянной частотой кадров на всех целевых устройствах.

## Краткое описание игры

На дворе 2097 год. Человечество досчитало над планетой и теперь настало время с ледование границ Солнечной системы в надежде столкнуться с новыми проблемами и открыть новые миры. Поскольку во всем мире царит мир, те, кто еще не покинул планету, пытаются новую форму коллективной скуки. Благодаря технологиям и биоактивингу все проблемы прошлого исчезли. Любой может стать лучшим спортсменом, красивым и талантливым без каких-либо усилий.

Чувствуя расущее волнение среди населения мира, Нэй Ласк, эксцентричный технологический предприниматель и квадриллионер, решает изобрести новый вид спорта, включающий мотоциклы с турбонаддувом, которые могут развивать скорость, с которыми большинство людей трудно справляться. Чтобы сделать задачу еще более сложной, он проектирует уникальную систему гоночных трасс, включающую горы и преграды.

Этот новый вид спорта воодушевляет наследие планеты, и миллионы людей решают принять в нем участие. Но немногие могут управлять этими высокотехнологичными мотоциклами и преодолевать на них смертоносные препятствия. Только опытный водитель одаренные, обладающие стальными нервами и острыми, как лезвие, рефлексами, могут овладеть этим видом спорта, но на данный момент мир ждет, чтобы узнать, кто первым станет его чемпионом.



Несмотря на то, что наш текущий проект предстает с общей аркадной очной игрой, в жанре которой обычно очень мало сюжета, но много игрового контекста, все же хорошо иметь некоторые элементы повествования, которые придают некоторый контекст миру игры. Это активизирует воображение игрока и придает игре общий смысл. Даже в классических аркадных играх, таких как Pac-Man и Missile Command, есть определенная степень повествовательных компонентов, которые создают ощущение, что вы играете персонажем в виртуальном мире.

## Цели игры

Список основных целей (так называемых целей игры) следующий:

- Основная цель (так называемое состояния победы) — пересечь финишную линию без каких-либо фатальных аварий.
- Вторичная цель — получить лучший результат, зарабатывая очки «приз», выбирая рискованные очные траектории при повороте между препятствиями.
- В качестве бонусной цели игра может получить дополнительные очки, собрав определенное количество коллекционных предметов (также известных как пикапы), которые лежат на земле. Эта цель назначена на «завершающих».
- Конечная цель игры — набрать наибольшее количество очков и финишировать в течение самого быстрого времени.



Хороший геймдизайнер всегда разрабатывает несолько задач, чтобы удовлетворить различные типы игр и поддерживать их как можно дольше. Например, игра с «завершающим» мышлением не просит закончить игру, но и завершить ее, достигнув всех возможных целей и получив все коллекционные предметы.

Игра может потерпеть неудачу (так называемое состояние неудачи) в достижении основных целей, выполнив следующие действия:

- Основное «состояние провала» срабатывает, когда игра не доехает до финиша из-за фатальной аварии.
- Если активирован режим игры с контрольными точками, игра может потерпеть неудачу, если не пересечет каждую контрольную точку вовремя.



Хорошей практикой также является четкое определение всех возможных «с состояний с боя» игры на раннем этапе, поскольку это помогает проанализировать, с какими различных условий выигрыша и состояний с боя нам придется обработать.

## Правила игры

Игровой уровень предстает с собой открытую среду, где игрок управляет мотоциклом, который на высокой скорости движется по железнодорожной системе, состоящей из четырех путей. Игрок может управлять транспортным средством боком, чтобы избегать препятствий, или захватывать предметы, встречающиеся на пути. Игрок должен дойти до финиша, достигнув контрольно-пунктовых пунктов, расположенных вдоль дорожек, за минимально необходимое время, избегая при этом повреждения транспортного средства, уклоняясь от препятствий. Игрок может набирать очки, выбирая рискованные гоночные траектории, например поворачивая, чтобы избежать препятствия в нескольких долях от столкновения с ним. Игрок также может получить бонусные очки, захватив определенное количество «обращенных» предметов в правильном порядке.

## Игровой цикл

Следующая диаграмма иллюстрирует основной игровой цикл:

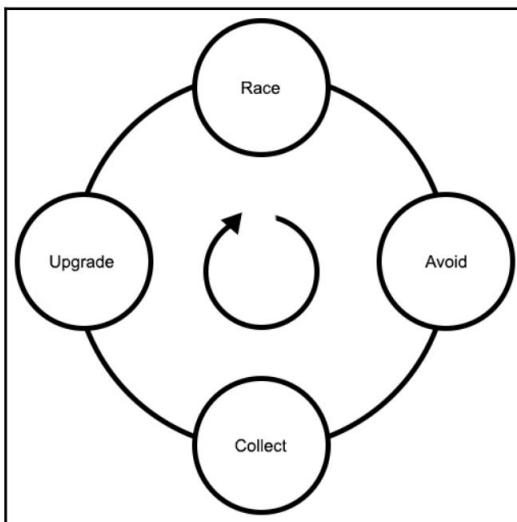


Рисунок 2.1 – Схема основного игрового цикла

В основном игровом цикле игры есть четыре стадии:

- Гонка: игрок мчится на время к финишу.
- Избегайте: игрок должен избегать препятствий, двигаясь как можно быстрее и рискуя, чтобы набрать очки.
- Сбор: игрок может собирать предметы и награды, находящиеся на гоночной трассе во время гонки.
- Обновление: во время или после гонки игрок может модернизировать автомобиль с помощью собранных предметов.

## Игровая среда

Основным местом действия каждого уровня является футуристическое пустынное пространство с четырьмя рельсами, простирающимися до горизонта. По обочинам железной дороги расположены металлические с ветофоры и шлагбаумы. Небо облачное, постоянный туман закрывает горизонт, часто случаются ливни. На трассах есть различные препятствия и предметы, которые игроку следует избегать или захватывать. Дизайн уровня ориентирован на наказание медленных игроков и вознаграждение тех, кто идет быстрее.

## Камера, управление, персонаж (3Cs)

3Cs (камера, управление и персонаж) — это стандартный термин, который используется в индустрии AAA в таких студиях, как Ubisoft. Поступи, в нем говорится, что суть опыта игрока определяется балансом между гармоничным взаимодействием камеры, персонажа и схемой управления. Если игрок чувствует, что персонаж реагирует на команды так, как ожидалось, а камера главно позиционируется под оптимальным углом, тогда общее впечатление будет более захватывающим. Давайте посмотрим на 3Cs подробно.

## Камера

Основная камера будет следить за автомобилем сзади с видом от третьего лица. Во время поворота она автоматически позиционируется и вращается на ближайшей трассе, противоположной направлению движения автомобиля. Во время турбонаддува камера отодвигается назад и начинает слегка кататься с легкостью, с оздавая ощущение, что вам трудно следить за автомобилем, который ускоряется. А когда моторчик снова замедлится до нормальной скорости, расположение между камерой и моторчиком вернется к значению по умолчанию, как мы видим на следующей диаграмме:

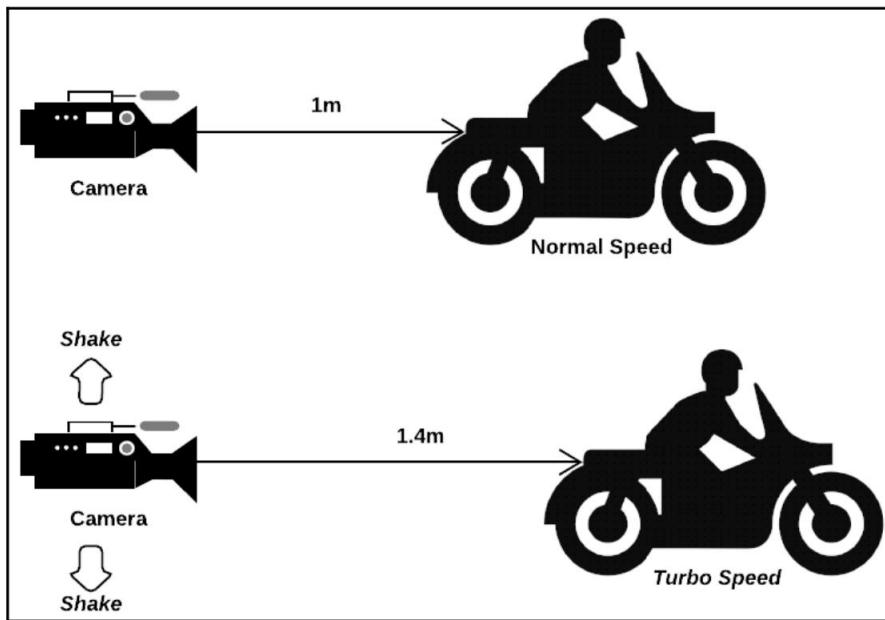


Рисунок 2.2 – Иллюстрация положений камеры

В текущей версии игры не будет камеры от первого лица или камеры заднего вида.

## Характер

В этом разделе мы определим главного персонажа, управляющего игровым процессом.

### Описание персонажа Главный

Главный персонаж — человек-пилот, который управляет футуристическим мотоциклом, курсирующем по железнодорожной системе и ограниченным маневрированием с одногорельсовой на другой на очень высоких скоростях. Учитывая, что главный игровой персонаж всегда время находится внутри кабинки транспортного средства, мы можем рассматривать транспортное средство как одного игрового персонажа во время игровых сегментов. В игре не будет других игровых персонажей гуманоидных персонажей или типов управляемых транспортных средств.

## Метрики персонажей

В следующем разделе описаны основные показатели персонажа/транспортного средства:

- Минимальная скорость: 500 км/ч. Во время гонки автомобиль всегда движется и никогда не может полностью становиться.
- Максимальная скорость: 6500 км/ч. Автомобиль может оставаться на максимальной скорости в пару секунд, и его грузоподъемность снизится до минимального значения.
- Максимальное здоровье: 100%. Транспортное средство имеет счетчик урона, расчетываемый в процентах.
- Урон от атаки: зависит от оружия. У машины нет системы вооружения, поэтому есть слот для установки обновлений.
- Управляемость: Зависит от текущей скорости. Грузоподъемность автомобиля снижается по мере достижения максимальной скорости.

## Состояния персонажа

В следующем разделе описаны основные состояния персонажа/транспорта:

- Холосстойкость: состояние простого автомобиля предстаёт с обой анимационный цикл, в котором показано, как шасси вибрирует от рева двигателя и мигают задние фонари.
- Превышение скорости: состояние превышения скорости имеет анимационный цикл вращения колес.
- Замедление: анимация замедления замедляет движение колес и мигает задними фонарями.
- Поворот: в состоянии поворота отображается анимация наклона автомобиля и его движения в направлении ближайшего пути.
- Торможение: анимация торможения заставит автомобиль наклоняться и скользить вбок по прямой траектории. Это состояние активируется только после того, как игрок пересекает финишную линию.
- Тряска: анимация тряски усиливается по мере снижения управляемости автомобиля на максимальной скорости.
- Переворот: анимация переворота переворачивает автомобиль вперед, когда он сталкивается с препятствием на дороге.
- Скользжение: анимация скольжения поворачивает автомобиль вбок и скользит вперед.



Очень важно с самого начала, чтобы команда дизайнера и программистов определила каждый тип игрового управления для каждого персонажа, устроившегося в транспортном средстве в игре. Главным образом потому, что реализация ЗС для автомобиля или двухногого гуманоидного персонажа требует разных подходов. Как и в реальном мире, вождение и ходьба — это разные виды опыта с определенными механическими тонкостями.

## Контроллер

В следующем разделе описывается схема основного контроллера, то есть клавиатуры:

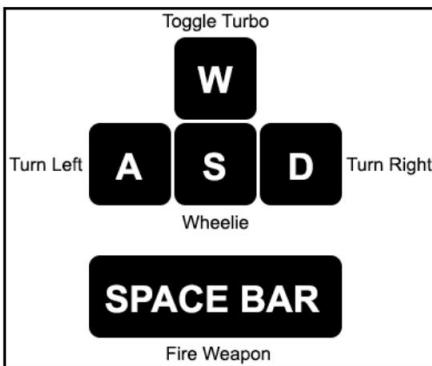


Рисунок 2.3 – Схема клавиш ввода клавиатуры



Мы будем поддерживать клавиатуру в качестве основного устройства ввода в игровом проекте этой книги и только изображений прости.

## Ингредиенты игры

В следующем разделе описаны основные ингредиенты игры:

- Велосипеды — основное транспортное средство в игре, которым может управлять игрок или искусственный интеллект.
- Пикапы — это предметы, которые являются на пути и которые игрок может подобрать, столкнувшись с ними. Каждый тип предметов будет иметь различную форму и будет парить на пару дюймов над дорожкой.

- Препятствия — это компоненты окружающей среды, которые погружаются на трассах. Определенный типы объектов-препятствий могут повредить транспортное средство, в то время как другие блокируют транспортное средство и вызывать его остановку.
- Оружие — это ингредиенты, которые можно прикрепить к досуговому салоту на игровом поле игра. Транспортное средство. Компоненты оружия играют оснащены лазерами и позволяют ему разрушать препятствия.
- Дроны — это роботизированные устройства, которые летают посекундно полета и атакуют игра. лазерными лучами.

## Супербаки

Имя	Описание	Обновление Слоты	Управление максимальной скоростью
Яд	Модель начального уровня для начинающих гонщиков. 1		80% 1900 км/ч
Серебряная пуля	Усовершенствованная модель для среднего уровня гонщиков.	2	60% 4000 км/ч
Смертельный вадник	Самая совершенная модель, досуговая только тогда игра.	4	40% 6500 км/ч



Управляемость мотоцикла определяется тем, насколько быстро автомобиль реагирует на волны игры. С увеличением скорости общая устойчивость снижается. Поэтому велосипед начнет скользить при повороте, и на него поворот уйдет больше времени, стабилизироваться и реагировать на команды игры.

## Пикапы

Имя	Описание	Эффект	Ценить
Коллекционный	Коллекционный предмет погружаются на длине трех в порядке номерации.	Увеличивает риск счет	10 точек риска
Турбоускорение	Этот предмет погружаются на пути в указанном месте.локации.	Заполняет турбометр	10% турбо
Ремонт повреждений*	Этот предмет погружаются на пути в определенные моменты.локации.	Восстанавливает повреждения	10% здоровья

## Препятствия

Имя	Описание	Эффект	Повреждать
Трековый барьер	Высокий и плоский барьер, блокирующий путь. Но ломается при ударе.	Наносит ущерб, но не сбивает машину	20% здоровья
Дорожный блок	Короткая баррикада сбивает машину. Заставляя ее перевернуться вперед и разбитьсь.	Сбивает автомобиль	10% здоровья

## Вооружение

Имя	Описание	Эффект	Диапазон
Лазерный стрелок	Оружие с единичным лучом Уничтожает все в пределах досядимости.		10 метров
Плазменный блaster	Многолучевое оружие	Уничтожает все в радиусе действия и поле зрения 60 градусов	20 метров

## Игровые системы

В следующем разделе описаны основные игровые системы:

- Железнодорожная система

Рельсовая система позволяет транспортировать игрука двинуться вперед на четырех отдельных колесах. Каждая траектория может сдавать препятствия и предметы по всей своей длине.

- Система рисков

Система риска позволяет игроку, который рискует, например, уклоняясь от игры. От препятствий в самый последний момент на максимальной скорости. Количество набранных баллов равно в зависимости от состояния между транспортом и барьером, когда игрок избегает этого. Датчик установленный перед велосипедом, с радиусом действия 5 метров. Постоянный мониторинг наличия препятствий текущего пути. Однажды обнаружено потенциальное сближение, датчик вычисляет расстояние между препятствием и автомобиль. Очки определяются исходя из состояния между игроком и препятствием удалось избежать.

На следующей схеме показан датчик, обращенный вперед:

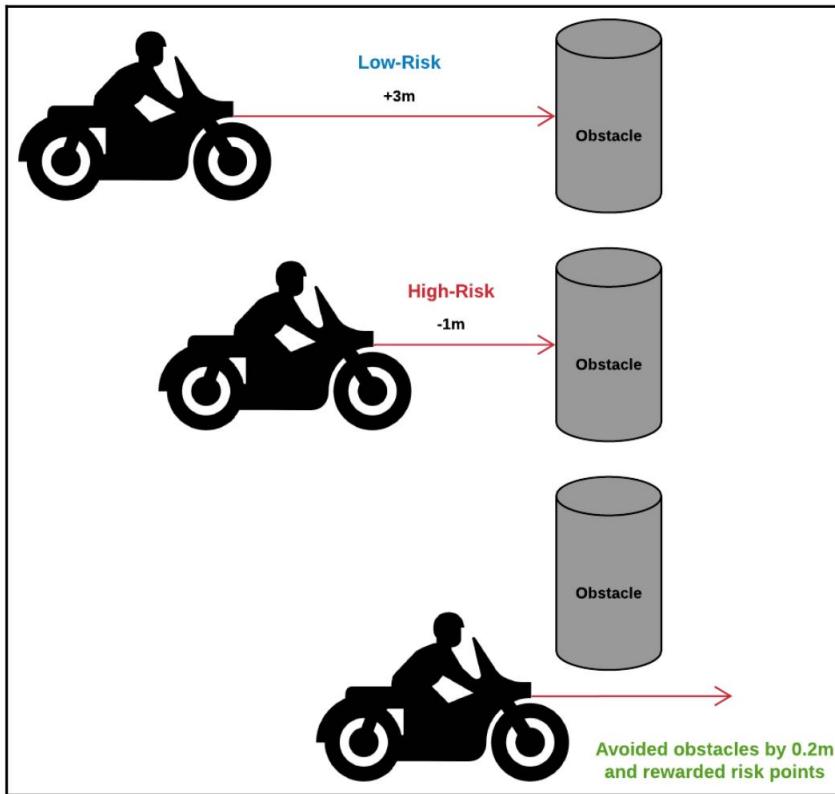


Рисунок 2.4 – Схема работы фронтального датчика обнаружения препятствий

- Система Turbo Boost Система

Turbo Boost (TBS) позволяет автомобилю развивать максимальную скорость. Чтобы активировать систему, игрок должен заполнить турбосчетчик, собирая предметы с турбонаддувом. Система TBS имеет датчик на проекционном дисплее (HUD), который позволяет игроку визуализировать достигнутый уровень турбонаддува. Когда TBS включена, управляемость автомобиля улучшается, и он становится более уязвимым к повреждениям.

- Система улучшения транспортного средства

С транспортом средства в транспортном средстве игрока есть слоты для предметов, которые позволяют прикрепить оружие и другие улучшения. В зависимости от типа транспортного средства количество слотов будет различным.



Разница между игровой механикой и системой иногда приводит к путанице. В этой книге я буду определять игровую систему как совокупность игровых механик и игровых ресурсов. Например, предметы, подбираемые для оружия, являются ингредиентами, но действия по их подбору на поле боя, а затем выбору правильного оружия в меню инвентаря для победы в битве — это индивидуальная механика. Но все эти элементы являются отдельными компонентами всей системы вооружений.

## Менюигры

В этом разделе мы опишем менюигры:

Главное меню Главное меню будет доступно в начале игры. Опции меню следующие:

- Начать игру: эта опция загружает сцену игровой трассы.
- Результаты: эта опция отображает 10 последних лучших результатов.
- Выход: эта опция закрывает окно игры.

Внутриигровое меню Внутриигровое меню доступно только во время гонки. Опции меню следующие:

- Перезапуск гонки: возобновляет гонку с самого начала.
- Выход из гонки: возвращает игрока обратно в сцену лобби.

## Игровой интерфейс

В следующем разделе описывается проекционный дисплей игры (HUD):

HUD будет отображаться только во время гонки и будет включать в себя следующие компоненты интерфейса:

- Турбометр
- Измеритель скорости
- Счетчик урона
- Таймер обратного отсчета
- Отслеживать индикатор выполнения



Часто окончательный дизайн меню HUD игры завершается на более поздних стадиях производства. Главным образом потому, что общий дизайн, художественный стиль и структура игры могут меняться несколько раз на разных этапах производства, поэтому порядок меню или расположение компонентов HUD могут постоянно меняться.

### Краткое содержание

В этой главе мы нашли время просмотреть текущий проект дизайна-документа игрового проекта книги. Этот черновой вариант GDD может быть неполным или несовершенным, но зачастую игровые проекты начинаются с неполной проектной документации идей. Так что все равно хорошо, что мы нашли время его просмотреть. Мы не будем реализовывать каждую систему, механику и ингредиент, описанные в этой книге, поскольку наша основная цель — изучить шаблоны проектирования путем создания игры, не обязательно полной.

Хотя дизайн игр не является основной темой этой книги, научиться думать как дизайнер — это навык, который может помочь любому стать лучшим программистом игр. Чтение проектной документации помогает нам понять концепцию структуру, механику и масштаб игры. И это следует учитывать при принятии решений по архитектуре программного обеспечения.

Следующая глава включает краткое введение в базовые основы программирования Unity. Если вы уже опытный программист, вы можете перейти к практическому разделу книги.

Но если вы новичок, я бы рекомендовал прочитать руководство, прежде чем приступить к реализации шаблонов проектирования следующих глав.

### дальнейшее чтение

Для получения дополнительной информации вы можете обратиться к следующим книгам:

- Искусство игрока Джесси
- Шелла Дизайн игр: Тайны мудрецов Марка Зальцмана
- Повышайте уровень! «Руководство по великому дизайну видеоигр»
- Скотта Роджерса «Правила игры: основы игрового дизайна» Кэти Сален и Эрика Циммермана

# 3

## Краткое руководство по программированию в Unity

Эта глава предстает с собой краткое введение, которое поможет вам познакомиться с расширенными возможностями языка C# и движком Unity. Я не буду пытаться подробно объяснять какие-либо темы, представленные здесь, поскольку они выходят за рамки этой книги. Тем не менее, я покрайней мере познакомлю вас с некоторыми новыми понятиями, чтобы избежать путаницы при упоминании о них в следующих главах. Я советую тем, кто обладает глубокими знаниями программирования на C# и Unity, пропустить эту главу. Но я рекомендую новичкам и разработчикам среднего уровня потратить время на просмотр содержания этой главы, чтобы получить общее представление о языке и функциях движка, которые мы будем использовать для реализации и проектирования наших игровых систем.

В любом случае для понимания этой книги не требуется полного владения C# и Unity, достаточно общего понимания и знакомства с некоторыми продвинутыми критическими концепциями.

В этой главе мы рассмотрим следующие темы:

- Что вы уже должны знать
- Возможности языка C#
- Возможности движка Unity



Код, представленный в этой главе, предназначен только для образовательных целей. Он не оптимизирован и не предназначен для ис пользования в рабочем коде.

## Что вы уже должны знать

В этом разделе я перечисляю некоторые основные функции языка C# и движка Unity, с которыми вам следует уже ознакомиться, прежде чем перейти к более сложным частям этой книги.

Ниже приведены некоторые основные возможности C#:

- Знакомство с модификаторами доступа к классам, такими как публичный и закрытый.
- Фундаментальные знания основных примитивных типов данных (int, string, bool, float и т. д.).
- Концептуальное понимание наследования и связи между базовым классом и производным классом.

Ниже приведены некоторые функции Unity:

- Базовое понимание того, как написать сценарий MonoBehaviour и присоединить его к GameObject как компонент
- Возможность создавать новую сцену Unity с нуля и манипулировать игровыми объектами внутри редактора.
- Знакомство с новыми функциями событий Unity (Пробуждение, Запуск, Обновление) и последовательностью выполнения.

Если вы не знакомы с концепциями, перечисленными ранее, я бы рекомендовал прочитать книгу и документацию по перечисленные в разделе «Дополнительная литература» этой главы.

## Возможности языка C#



Функции языка, такие как события и делегаты, могут быть слишком сложными для новичков, поэтому, если вы относите себя к этой категории, не волнуйтесь; вы все еще можете наслаждаться этой книгой. Просмотрите главы для начинающих, например те, в которых объясняются такие шаблоны, как Singleton, State, Facade и Adaptor.

Следующие расширенные возможности языка C# имеют особенное значение для оптимальной реализации некоторых шаблонов проектирования, которые мы будем реализовывать в следующих главах:

- Статический: доступ к методам и членам класса с ключевым словом static можно получить напрямую по его имени и без инициализации экземпляра. Статические методы и члены полезны, поскольку они легкодоступны из любого места вашего кода. В следующем примере показан класс, который использует статическое слово для создания глобально доступной шины событий:

```
ис пользование UnityEngine.Events;
ис пользование System.Collections.Generic;

прос транс тво имен Chapter.EventBus {

общественный класс с RaceEventBus {

частный статический только для чтения
IDictionary<RaceEventType, UnityEvent>
События = новый словарь<RaceEventType, UnityEvent>();

public static void Подписаться (
    RaceEventType eventType, прослушиватель UnityAction)
{
    UnityEvent thisEvent; if
    (Events.TryGetValue(eventType, out thisEvent)) {

        thisEvent.AddListener(прослушиватель);

    } еще
    {
        thisEvent = новый UnityEvent();
        thisEvent.AddListener(прослушиватель);
        Events.Add(eventType, thisEvent);
    }
}

public static void Отписаться (
    RaceEventType eventType, прослушиватель UnityAction)
{
    UnityEvent thisEvent; if
    (Events.TryGetValue(eventType, out thisEvent)) {

        thisEvent.RemoveListener(прослушиватель);
    }
}
}
```

```
public static void Publish (RaceEventType eventType) {
```

```
    UnityEvent thisEvent; if
    (Events.TryGetValue(eventType, out thisEvent)) {

        этоСобытие.Вызвать();
    }
}
```

- События : события позволяют объекту, который действует как издатель, отправлять сигналы, который может получить другие объекты; эти объекты, которые прослушивают определенное событие, называются подписчиками. События полезны, когда вы хотите строить архитектуру, управляющую событиями. Ниже приведен пример класса, публикующий события :

```
использование UnityEngine;
использование System.Collections;
```

```
публичный класс CountdownTimer: MonoBehaviour {
```

```
    публичный делегат void TimerStarted(); общедоступное
    статическое событие TimerStarted OnTimerStarted; публичный делегат
    void TimerEnded(); общедоступное статическое
    событие TimerEnded OnTimerEnded; [SerializeField] продолжительность
    час тного о плавающей
    режима = 5.0f; недействительный старт()
```

```
{
    StartCoroutine(StartCountdown());
}
```

```
час тный IEnumerator StartCountdown() {
```

```
    если (OnTimerStarted != ноль)
        ОнТаймерНачал();
    в то время как (длительность > 0)
    {
        дождь одноточия возврата новых WaitForSeconds (1f);
        продолжительность--;
    }
    } if (OnTimerEnded != ноль)
        OnTimerEnded();
```

```
} void OnGUI() {
```

```
    GUI.цвет = Цвет.синий; GUI.Label(
```

```
    новый Rect(125, 0, 100, 20), "ОБРАТНЫЙ ОСЧЕТ:"  
}  
}  
}
```

- **Делегаты.** Концепция делегатов проста, если вы понимаете их базовый низкоуровневый механизм. Высокуюровневое определение делегатов заключается в том, что они содержат ссылки на функции. Но это очень абстрактное определение того, что на самом деле делают делегаты закулисами. Это указатели на функции, что означает, что они содержат адрес памяти для других функций. Таким образом, мы могли бы визуализировать их как адресную книгу, содержащую список расположений функций. Вот почему делегат может хранить несколько из них и вызывать их все одновременно. Ниже приведен пример класса, который подразумевается на событиях, инициируемые классом издателя, путем назначения определенных локальных функций делегатам издателя:

использование UnityEngine;

```
общественный класс Buzzer: MonoBehaviour {  
  
недействительный OnEnable()  
{  
    // Назначение локальных функций делегатам, определенным в  
    // Класс таймера  
    CountdownTimer.OnTimerStarted += PlayStartBuzzer; CountdownTimer.OnTimerEnded  
    += PlayEndBuzzer;  
}  
  
недействительный OnDisable()  
{  
    CountdownTimer.OnTimerStarted -= PlayStartBuzzer; CountdownTimer.OnTimerEnded -=  
    PlayEndBuzzer;  
}  
  
недействительный PlayStartBuzzer()  
{  
    Debug.Log("[BUZZER]: Включить звуковой сигнал!");  
}  
  
недействительный PlayEndBuzzer()  
{  
    Debug.Log("[BUZZER]: Звук окончания воспроизведения!");  
}  
}
```

- Универсальные шаблоны: соответствующие функции C#, которые позволяют откладывать спецификацию типа до тех пор, пока клиент не объявит и не создаст экземпляра класса. Когда мы говорим, что класс является универсальным, у него нет определенного типа объекта. Ниже приведен пример универсального класса, который может выступать в качестве шаблона:

```
// <T> может быть любого типа.
общедоступный класс Singleton<T>: MonoBehaviour, где T: Component {
    // ...
}
```

- Сериализация: Сериализация — это процесс преобразования экземпляра объекта в двоичную или текстовую форму. Этот механизм позволяет нам сохранять состояния объектов в файле. Ниже приведен пример функции, которая сериализует экземпляра объекта с сохранением его в виде файла.

```
частная пуск SerializePlayerData (PlayerData playerData) {
    // Сериализация экземпляра PlayerData BinaryFormatter
    bf = new BinaryFormatter (); Файл FileStream =
    File.Create (Application.persistentDataPath +
    "/playerData.dat");
    bf.Serialize (файл, playerData); файл.Закрыть ();
}
```

C# — это очень глубокий язык программирования, поэтому мне было бы невозможно объяснить каждую из его основных функций в рамках этой книги. Представленные в этом разделе книги очень полезны и помогут нам реализовать игровые системы и шаблоны, описанные в следующих главах.

## Возможности движка Unity

Unity — это полнофункциональный движок, включающий комплексный API сценарiev, систему анимации и множество дополнительных функций для разработки игр. Мы не можем охватить их все в этой книге, поэтому я перечислю только основные компоненты Unity, которые мы будем использовать в следующих главах о шаблонах проектирования:

- Префабы: Префаб — это сборочный контейнер с обрамленными игровыми объектами и компонентами. Например, вы можете иметь отдельные префабы для каждого типа транспортных средств в вашей игре и динамически загружать их в сцену. Префабы позволяют сдавать и организовывать многократно используемые игровые объекты в качестве строительных блоков.

- События и действия Unity: Unity имеет встроенную систему событий; она очень похожа на систему событий C#, но с дополнительными функциями, специальными для языка, такими как возможность просматривать и настраивать их в Инспекторе.
- ScriptableObjects: класс, производный от базового класса ScriptableObject, может выступать в качестве контейнера для данных. Другой базовый класс Unity, MonoBehaviour, используется для реализации поведения. Поэтому рекомендуется использовать MonoBehaviours для хранения вашей логики и ScriptableObjects для хранения ваших данных. Экземпляр ScriptableObject можно создавать как ресурс и часто использовать для разработки рабочих процессов. Это позволяет не программировать создавать новые варианты объектов определенного типа без единой строки кода.

Ниже приведен пример создания ScriptableObject, который позволяет создавать новые настраиваемые экземпляры Sword:

```
использование UnityEngine;
[CreateAssetMenu(fileName = "NewSword", MenuName =
«Оружие/Меч»)] public class
Sword: ScriptableObject {
    общедоступная строка SwordName;
    общедоступная строка SwordPrefab;
}
```

- Создание раммы. Концепция сопрограмм не ограничивается Unity, но является важным инструментом Unity API. Типичное поведение функции — выполнение с себя от начала до конца. Но с сопрограммами — это функция с дополнительной возможностью ожидания, определения времени и даже установки процессов выполнения. Эти дополнительные функции позволяют нам реализовать ложное поведение, которое не легко реализовать с помощью обычных функций. Сопрограммы похожи на потоки, но обеспечивают параллелизм вместо параллелизма. В следующем примере кода демонстрируется реализация таймера обратного отсчета с использованием сопрограмм:

```
использование UnityEngine;
использование System.Collections;
публичный класс CountdownTimer: MonoBehaviour {
    частный float _duration = 10.0f;
    IEnumerator Start() {
        Debug.Log("Таймер запущен!"); дох одноть
        return StartCoroutine(WaitAndPrint(1.0f)); Debug.Log("Таймер закончился !");
    }
}
```

```

        }

IEnumerator WaitAndPrint (float waitTime) {

    while (Time.time < _duration) {

        // здесь однотипно возврата новых WaitForSeconds (waitTime); +
        Debug.Log("Секунды: " + Mathf.Round(Время .время ));

    }
}

```

Как мы видим, Unity имеет несолько общих, но простых функций, которые позволяют нам реализовывать системы и организовывать данные. Мы не можем подробно рассмотреть каждого из новых функций Unity, поскольку этого однозначно предполагает книга. Если вы чувствуете, что вам нужна дополнительная информация о движении, прежде чем двигаться дальше, я рекомендую просмотреть материалы, ссылки на которые приведены в разделе «Дополнительная литература».

## Краткое содержание

Этот глава предназначена для использования в качестве основы для создания общей базы знаний перед переходом к практическому разделу книги. Но для того, чтобы начать использовать шаблоны проектирования в Unity, необходимо обязательно овладеть функциями, представленными в предыдущих разделах. Некоторые из них мы еще раз рассмотрим более подробно в следующих главах, на этот раз в практическом контексте.

В следующей главе мы рассмотрим наш первый шаблон — печально известный Singleton. Мы будем использовать его для реализации класса Game Manager, отвечающего за анализ игры.

## дальнейшее чтение

Для получения дополнительной информации вы можете обратиться к следующим материалам:

- Изучение C# путем разработки игр на Unity 2020, Харрисон Ферроне. Руководство
- пользователя Unity, Unity Technologies: <https://docs.unity3d.com/Manual/index.html>

# Раздел 2: Основные шаблоны

В этом разделе книги мы рассмотрим некоторые фундаментальные закономерности и будем использовать их для создания основных систем механик нашей игры.

Этот раздел состоит из следующих глав:

- Глава 4. Реализация игрового менеджера с помощью Singleton
- Глава 5. Управление состояниями персонажа с помощью шаблона состояния
- Глава 6. Управление игровыми событиями с помощью шаблона событий
- Глава 7. Реализация системы воспроизведения с помощью шаблона команды
- Глава 8. Оптимизация с помощью шаблона пул объектов
- Глава 9. Разделение компонентов с помощью шаблона наблюдателя
- Глава 10. Реализация усилений с помощью шаблона посредителя
- Глава 11. Реализация дрона с помощью шаблона стратегии
- Глава 12. Использование декоратора для реализации системы оружия
- Глава 13. Реализация редактора уровней с пространственным разделением

# 4

## Реализация игрового менеджера с помощью Singleton

В этой первой практической главе мы рассмотрим один из самых печально известных шаблонов проектирования программного обеспечения в области программирования — Singleton. Многие могут возразить, что Singleton — это наиболее широкий и полезный шаблон среди разработчиков Unity, возможно, потому, что его легче всего изучить. Но он также может быстро стать «клейкой лентой» в нашем наборе инструментов программирования, к которой мы обращаемся каждый раз, когда нам нужно пристроить решение с ложной архитектурой проблемы.

Например, ис пользуя этот шаблон, мы можем быстро создать простую архитектуру кода, основанную на упаковке и управлении всеми новыми системами нашей игры в отдельных классах менеджеров. Тогда мы могли бы добавить этих менеджеров предварительно понятные и понятные интерфейсы, которые скроют внутреннюю ложность системы. Кроме того, чтобы обеспечить легкий доступ к этим менеджерам и одновременное выполнение только одного экземпляра, мы реализуем их какsingletonы. Этот подход может показаться надежным и полезным, но он полон подводных камней, поскольку создает прочную связь между новыми компонентами и очень затрудняет модульное тестирование.

В этой книге мы попытаемся отойти от этого типа архитектуры и использовать шаблоны проектирования для создания более надежной, модульной и масштабируемой базы кода. Но это не означает, что мы будем игнорировать Singleton и считать его ошибочным. Вместо этого в этой главе мы рассмотрим вариант использования, для которого этот шаблон хорошо подходит.

В этой главе будут рассмотрены следующие темы:

- Основы паттерна Singleton
- Написание многоправового класса Singleton в Unity
- Реализация глобального доступа к GameManager

## Технические требования

Это практическая глава; вам потребуется базовое понимание Unity и C#.

Мы будем пользоваться следующий конкретный движок Unity и концепции языка C#: Generics.

Если вы не знакомы с этой концепцией, просмотрите главу 3 «Краткое руководство по программированию Unity».

Файлы кода этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/>.

главная /Активы/Главы/Глава04.

Посмотрите следующее видео, чтобы увидеть Кодекс в действии: <https://bit.ly/3wDbM6W>



Generics — это привлекательная функция C#, которая позволяет нам отложить тип класса во время выполнения. Когда мы говорим, что класс является универсальным, это означает, что у него нет определенного типа объекта. Этот подход выгоден, поскольку мы можем присвоить ему определенный тип при его инициализации.

## Понимание шаблона Singleton

Как следует из названия, основная цель паттерна Singleton — гарантировать уникальность. Этот подход означает, что если класс правильно реализует этот шаблон, после инициализации он будет иметь только один свой экземпляр в памяти во время выполнения. Этот механизм может быть полезен, если у вас есть класс, управляющий системой, которая должна быть доступна глобально из единой и согласованной точки входа.

Конструкция Singleton довольно проста. Когда вы реализуете класс Singleton, он становится ответственным за то, чтобы в памяти было только один объект.

Как только Singleton обнаружит экземпляра объекта того же типа, что и он сам, он немедленно уничтожит его. Поэтому он довольно безжалостен и не терпит никакой конкуренции. Следующая диаграмма в определенной степени иллюстрирует этот процесс:

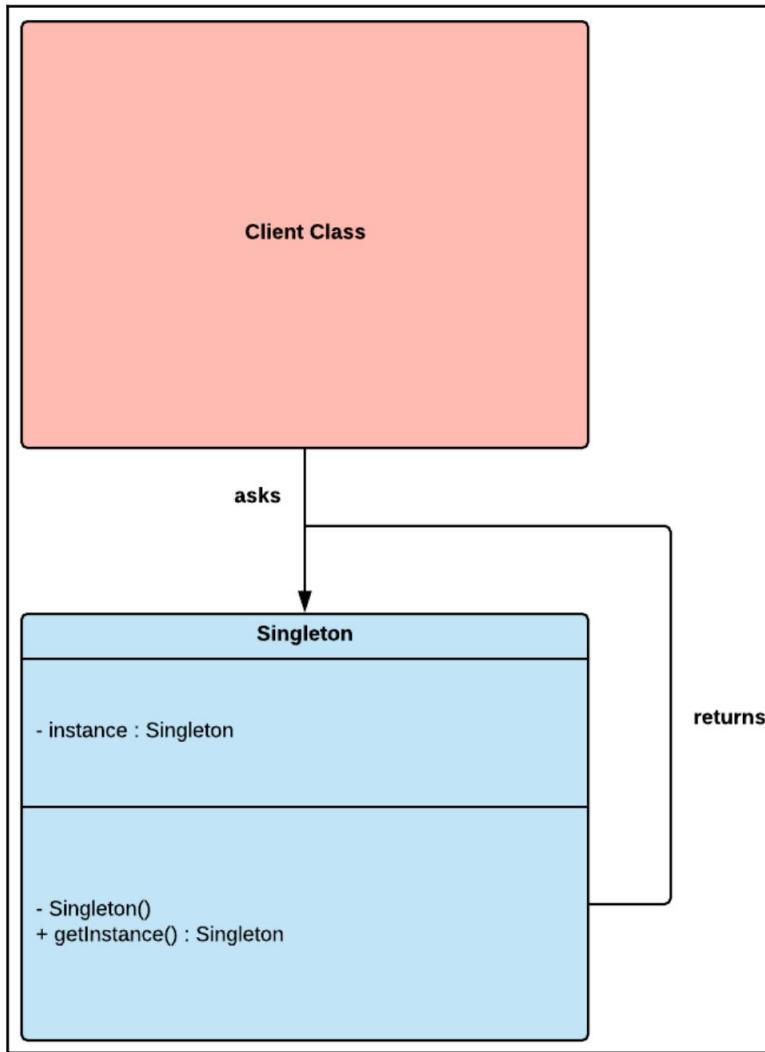


Рисунок 4.1 – UML-диаграмма шаблона Singleton

Самый важный вывод из этого описания паттерна Синглтон заключается в том, что если хорошо реализованный, он гарантирует, что может быть только один из них ; в противном случае он не достигает своей цели.

## Преимущества и недостатки

Вот некоторые преимущества шаблона Singleton:

- Доступность по всему миру. Мы можем использовать шаблон Singleton для создания глобальной точки доступа к ресурсам или сервисам.
- Управление параллелизмом. Этот шаблон можно использовать для ограничения одновременного доступа к общим ресурсам.

Вот некоторые недостатки шаблона Singleton:

- Модульное тестирование. При чрезмерном использовании Singleton может очень затруднить модульное тестирование. В конечном итоге мы можем столкнуться с тем, что объекты Singleton будут зависеть от других Singleton. Если в какой-то момент один из них отсутствует, цепочка зависимости разрывается. Эта проблема часто возникает при объединении Facade и Singleton для настройки интерфейсов с новыми системами. В итоге мы получаем класс с множеством методов, каждый из которых управляет определенным новым компонентом игры, и все они зависят друг от друга в своем функционировании. Поэтому становится невозможным изолированное тестирование и отладку.
- Лень: из-за простоты использования Singleton представляется с собой шаблон, который может быть просто привить ошибочные привычки программирования. Как упоминалось в разделе «Недостаток модульного тестирования», с помощью Singleton мы можем легко делать все доступным из любого места. Простота, которую он предлагает, также может заставить нас не желать тестировать более сложные подсистемы при написании кода.



При выборе дизайна важно всегда помнить, является ли ваша архитектура поддерживаемой, масштабируемой и тестируемой. Когда дело доходит до тестирования, я часто спрашиваю себя, могу ли я легко протестировать свои новые системы, компоненты и механизмы индивидуально и изолированно. Если нет, то я знаю что принял несколько потенциально неразумных решений.

## Проектирование игрового менеджера

Стандартный класс, который мы часто видим в проектах Unity, — это Game Manager. Обычно разработчики реализуют его как Singleton, но его ответственность варьируется от одной базы кода к другой. Некоторые программы используют его для управления игровыми состояниями верхнего уровня или в качестве глобально доступного интерфейса для основных игровых систем.

В контексте этой главы мы возложим на него исключительную ответственность за управление игровой сессией. Подобно концепции мастера игры в настольных играх, он будет отвечать за настройку игры для игрока. Он также может брать на себя дополнительные обязанности, такие как связь с первыми службами, инициализация глобальных настроек, ведение журнала и сохранение прогресса игрока.

Важно помнить, что Game Manager будет работать на протяжении всей жизни игры. Следовательно, в памяти всегда будет единственный, но постоянный экземпляр.

Следующая диаграмма иллюстрирует общую концепцию:

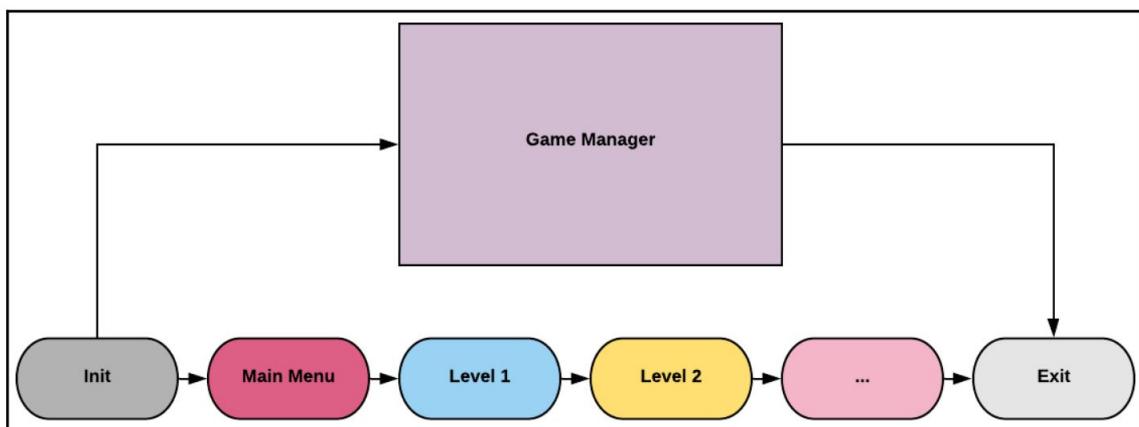


Рисунок 4.2 – Диаграмма, иллюстрирующая срок службы Game Manager

В следующем разделе мы возьмем только что созданный дизайн и переведем его в код.

## Реализация игрового менеджера

В этом разделе мы реализуем класс Singleton и Game Manager. Мы попытаемся использовать некоторые основные функции Unity API, чтобы адаптировать шаблон для использования в движке:

- На первом этапе процесс мы реализуем класс Singleton. К

Чтобы было легче понять его структуру, мы разделим его на два отдельных сегмента:

использование UnityEngine;

```

    protected void OnEnable()
    {
        if (_instance == null)
        {
            _instance = this;
            DontDestroyOnLoad(gameObject);
        }
    }

    public static T GetInstance()
    {
        return _instance;
    }

    protected void Awake()
    {
        if (_instance != null && _instance != this)
        {
            Destroy(gameObject);
        }
        else
        {
            _instance = this;
            DontDestroyOnLoad(gameObject);
        }
    }

    protected void OnDestroy()
    {
        if (_instance == this)
        {
            _instance = null;
        }
    }
}

```

В первом сегменте класса Singleton<T> мы видим, что реализовали общедоступное статическое свойство с помощью метода доступа get. В этом методе доступа мы проверяем отсутствие существующего экземпляра этого объекта перед инициализацией нового. FindObjectOfType<T>() ищет первый загруженный объект указанного типа. Если мы не можем его найти, то создаем новый GameObject, переименовываем его и добавляем к нему компонент неуказанных типов.

Этот процесс станет более очевидным, когда мы реализуем класс GameManager.

## 2. Реализуем последний сегмент класса Singleton :

```

public void Awake()
{
    if (_instance == null)
    {
        _instance = this;
        DontDestroyOnLoad(gameObject);
    }
}

```

```
        Уничтожить (игровой объект);  
    }  
}  
}  
}
```

Для последнего класса у нас есть метод Awake(), который мы пометили как виртуальный, что означает, что он может быть переопределен производным классом. Важно понимать, что когда движок вызывает метод Awake(), компонент Singleton проверит, существует ли уже его экземпляр, инициализированный в памяти. Если нет, то он станет текущим экземпляром. Но если он уже существует, он уничтожит себя, чтобы предотвратить дублирование.

Следовательно, в это же одновременно может быть только один экземпляр определенного типа Singleton. Если вы попытаетесь добавить два, один будет автоматически уничтожен.

Еще одна важная деталь, которую следует рассмотреть, — это следующая строка:

```
DontDestroyOnLoad(gameObject);
```

DontDestroyOnLoad — общий статический метод, включенный в Unity API; он предотвращает уничтожение целевого объекта при загрузке новой сцены. Другими словами, он гарантирует, что текущий экземпляр объекта сюда попадет даже при переключении между сценами. Эта функция API удобна для нашего синглтона, поскольку она гарантирует, что объект будет доступен на протяжении всего срока службы приложения, в данном контексте игры.

3. На заключительных этапах нашей реализации мы напишем скелетную версию класса GameManager. Для краткости мы сосредоточимся только на коде, который будет проверять нашу реализацию Singleton:

```
использование  
системы; использование  
UnityEngine; использование UnityEngine.SceneManagement;  
  
последование имен Chapter.Singleton {  
  
    публичный класс GameManager: MonoBehaviour {  
  
        часный DateTime _sessionStartTime; часный DateTime  
        _sessionEndTime;  
  
        недействительный Старт () {  
            //ДЕЛАТЬ:  
            // - Загрузить сюда ранение игрового  
            // - Если нет сюда ранения, перенаправляем игровое занятие на сцену регистрации
```

```
// - Вызовите серверную часть и получите ежедневные испытания и награды

_sessionStartTime = DateTime.Now; Debug.Log("Начало
игровой
сессии @: "
+ DateTime.Now);

} void OnApplicationQuit () { _sessionEndTime
= DateTime.Now;

TimeSpan timeDifference =
_sessionEndTime.Subtract(_sessionStartTime);

Debug.Log("Игровая сессия завершена @: " + DateTime.Now);

Debug.Log("Игровая сессия длилась: " + TimeSpan.ToString());
}

void OnGUI() { if
(GUIGUILayout.Button("Следующая сцена"))
{ SceneManager.LoadScene(
SceneManager.GetActiveScene().buildIndex + 1);
}
}
}
```

Чтобы дать больше контекста GameManager, мы оставили список TODO потенциальных задач, которые должен выполнить класс. Но мы также добавили таймер и кнопку графического интерфейса. Оба помогут нам проверить, работает ли наш Singleton, когда мы начнем этап тестирования.

Но на данный момент наш GameManager не является Singleton, чтобы сделать его единым, нам просто нужно внести одно изменение в одну строку кода, как мы видим здесь:

общедоступный класс GameManager: Singleton<GameManager>

Этот простой мы взяли обычный класс MonoBehaviour и преобразовали его в Singleton с помощью одной строки кода. Это стало возможным, потому что мы используем Generics. Следовательно, наш класс Singleton может быть чем угодно, пока мы не присвоим ему определенный тип.

4. Итак, на последнем этапе мы взяли наш класс GameManager и преобразовали его в Singleton, как показано здесь:

использование
системы; использование UnityEngine;

```
ис пользование UnityEngine.SceneManagement;

протрансовоимен Chapter.Singleton {

    публичный класс GameManager: Singleton<GameManager> {

        час тный DateTime _sessionStartTime; час тный DateTime
        _sessionEndTime;

        недействительный Старт () {
            //ДЕЛАТЬ:
            // - Загрузить с ох ранение игрока
            // - Если нет с ох ранения, перенаправля ем игрока на сцену регистрации
            // - Вызовите серверную часть и получите ежедневные испытания и награды

            _sessionStartTime = DateTime.Now; Debug.Log( "Начало
игровой сессии
@: " + ДатаВремя .Сейчас );
        }

        void OnApplicationQuit () { _sessionEndTime
= DateTime.Now;

        TimeSpan timeDifference =
        _sessionEndTime.Subtract(_sessionStartTime);

        Debug.Log( "Игровая сессия завершена @: " + ДатаВремя .Сейчас );

        Debug.Log( "Игровая сессия длилась: " + Разница во времени);
    }

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

Теперь, когда мы подготовили все ингредиенты, пришло время приступить к этапу тестирования, которым мы и займемся дальше.

## Тестирование игрового менеджера

Если вы хотите протестировать только что написанные классы в своем экземпляре Unity, вам следует выполнить следующие шаги:

1. Создайте новую пустую сцену Unity под названием Init.
2. В сцене Init добавьте пустой GameObject и привяжите класс GameManager к этому.
3. Создайте несколько пустых сцен Unity, сколько захотите.
4. В настройках сборки в меню «Файл» добавьте сцену Init с индексом 0:

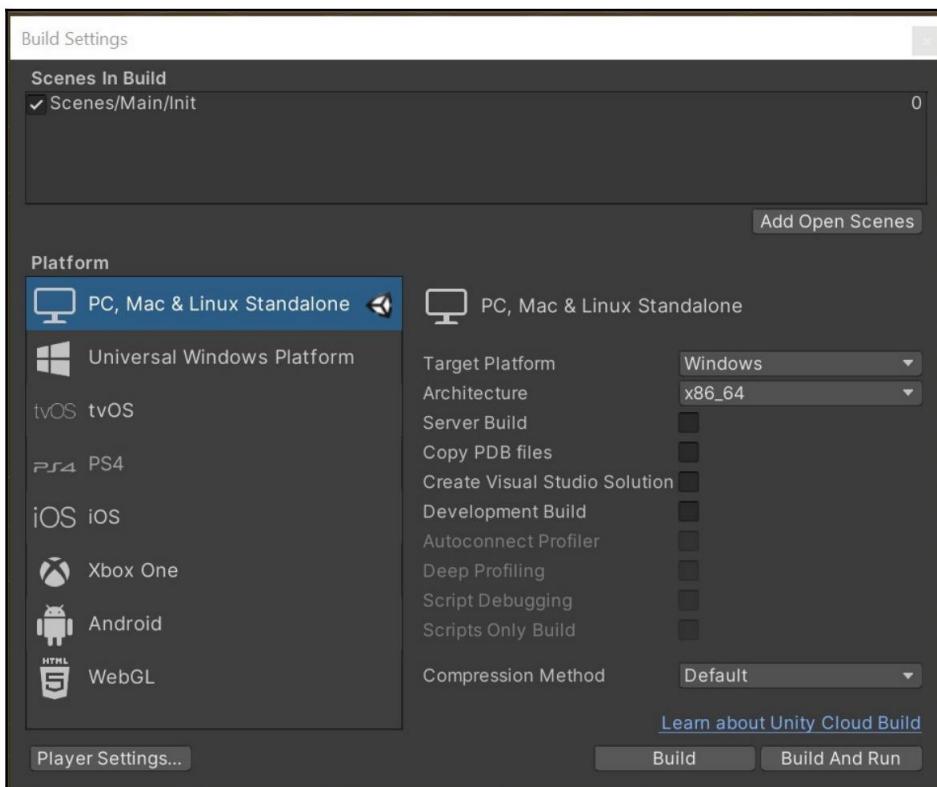


Рисунок 4.3 – Настройки сборки

5. Затем добавьте новые пустые сцены Unity в список «Настойки сборки» в любом количестве. Ты очешь.

Если вы сейчас запустите сцену инициализации, вы должны увидеть кнопку графического интерфейса с именем «Следующая сцена», как показано на следующем скриншете экрана:

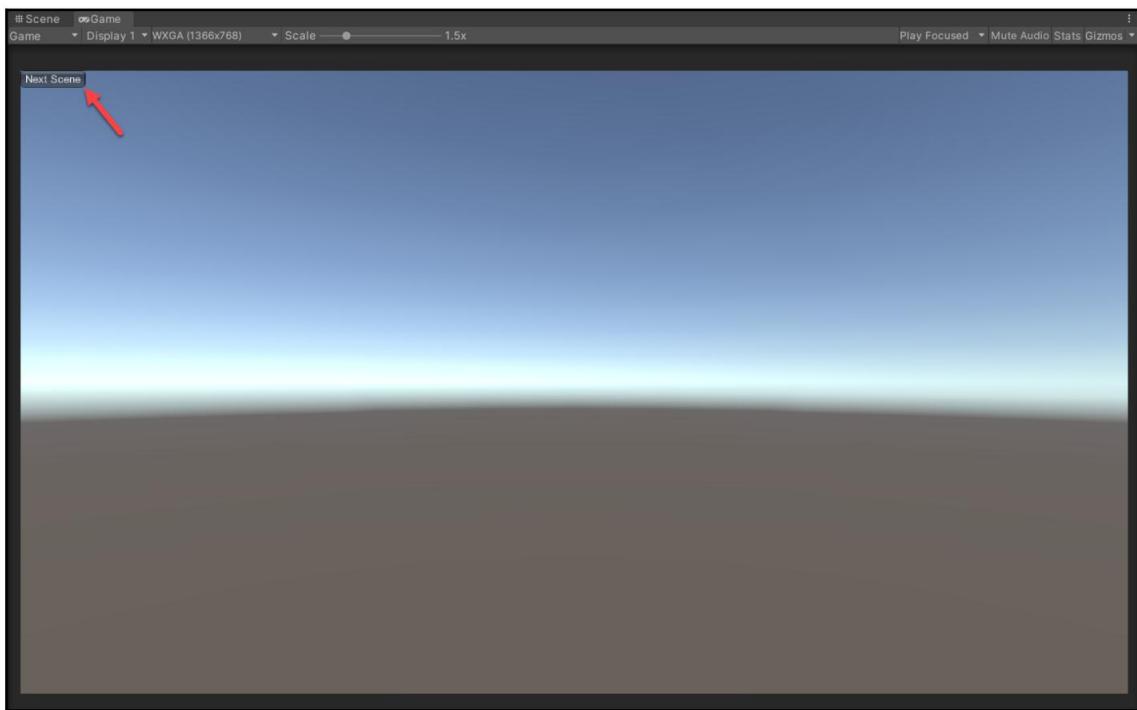


Рисунок 4.4 – Скриншот примера кода в действии

Если вы нажмете кнопку «Следующая сцена», вы прокрутите каждую из сцен, добавленных в настройках сборки, и графический интерфейс станет на экране. Если вы перестанете запускать игру, вы должны увидеть в журнале консоли продолжительность вашего сеанса. Если вы попытаетесь добавить дополнительные GameManager к GameObjects в любой сцене, вы заметите, что они уничтожаются, поскольку в течение всей жизни игры может существовать только один.

На этом наши тесты завершаются; теперь у нас есть первый черновик класса GameManager и многоразовая реализация Singleton.

### Краткое сведение

В этой главе мы рассмотрели один из самых противоречивых шаблонов проектирования. Но мы нашел способ реализовать это с помощью посреднического и многократного подхода. Синглтон — это шаблон, который идеально подходит для модели кодирования Unity, но чрезмерное его использование может привести к статье слишком зависимым от него.

В следующей главе мы рассмотрим шаблон State, который будем использовать для реализации класса контроллера для основного игрока нашего игры — очного матча.

# 5

## Управление состояниями персонажа с помощью шаблона состояния

В видеоиграх сущности состояния непрерывно из одного состояния в другое в зависимости от действий игрока или событий. Вражеский персонаж может перейти из состояния ожидания в состояние атаки в зависимости от того, заметил ли он игрока, движущегося по карте. Персонаж игрока состояния непрерывно от одной анимации к другой, реагируя на действия игрока. В этой главе мы рассмотрим шаблон, который позволяет нам определять отдельные состояния сущности и ее поведение с учетом состояния.

Для начала мы будем использовать традиционный шаблон State для управления отдельными конечными состояниями нашего главного героя. В контексте нашего проекта гонки главным героем является мотоцикл, поэтому у него есть набор механических моделей поведения и анимаций. По мере продвижения в реализации шаблона состояний мы вскоре увидим его ограничения, которые мы преодолеем, представив концепции FSM (конечного автомата).

Мы не будем писать FSM вручную вместе с этим, рассмотрим реализацию этого концепта в собственной системе анимации Unity. Поэтому в этой главе будет представлен двойной подход: введение в новые концепции шаблона State и обычное профилирование системы анимации Unity для управления состояниями персонажей и анимаций.

В этой главе мы рассмотрим следующие темы:

- Обзор шаблона «Состояние»
- Реализация паттерна State для управления состояниями главного героя.

## Технические требования

Этот главный раздел является практической. Вам потребуется базовое понимание Unity и C#.

Файлы кода для этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter05>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/36EbbHe>

## Обзор шаблона «Состояние»

Мы используем шаблон проектирования «Состояние» для реализации системы, которая позволит объекту изменять свое поведение в зависимости от его внутреннего состояния. Таким образом, изменение контекста приведет к изменению поведения.

В структуре модели «Состояние» есть три основных участника:

- Класс `Context` определяет интерфейс, который позволяет клиенту запрашивать изменение внутреннего состояния объекта. Он также содержит указатель на текущее состояние.
- Интерфейс `IState` устанавливает контракт реализации для конкретных классов состояний.
- Классы `ConcreteState` реализуют интерфейс `IState` и предоставляют общий метод с именем `handle()`, который объект `Context` может вызывать для запуска поведения состояния.

Давайте теперь рассмотрим диаграмму определения этого шаблона, но в контексте практической реализации:

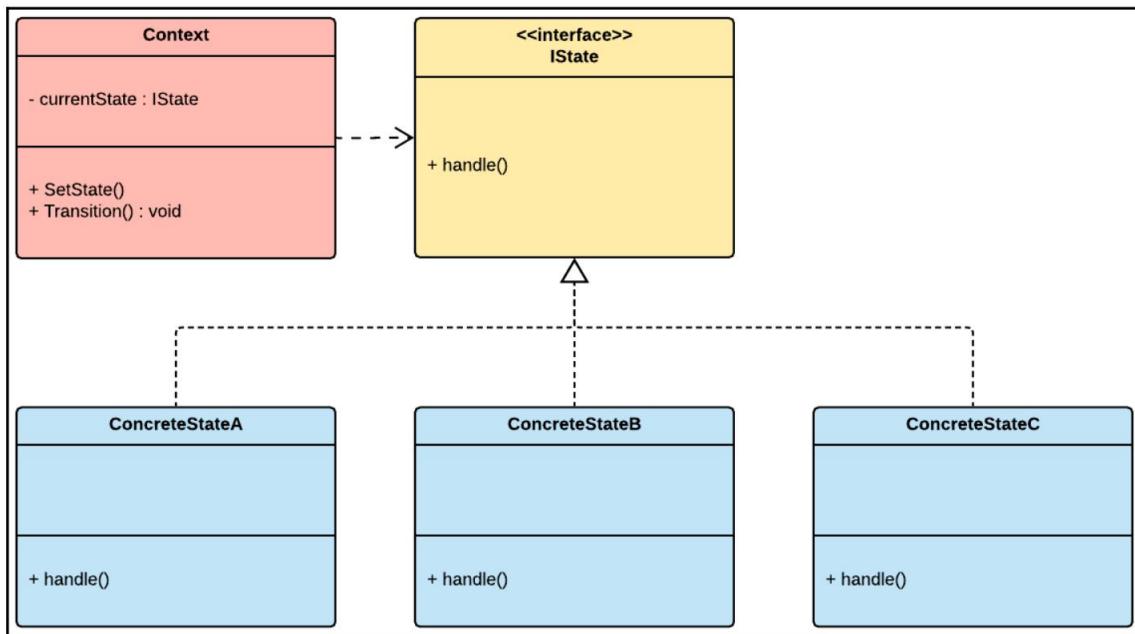


Рисунок 5.1 – UML-диаграмма шаблона «Состояние»

Чтобы обновить состояние объекта, клиент может установить ожидаемое состояние через объект Context и запросить переход в новое состояние. Таким образом, контекст всегда знает текущее состояние объекта, который он обрабатывает. Однако ему не обязательно знать каждый из конкретных существующих классов состояний. Мы можем добавить с только классов состояний, сколько пожелаем, не изменяя ни единой строки кода в классе Context.

Например, этот подход масштабируется лучше, чем определение всех наших состояний в одном классе и управление переходом между ними с помощью лучшей переключения, как мы можем видеть в следующем примере псевдокода:

```

общественный класс BikeController {

    ...
    переключатель (состояние)
    {
        случай StopState:
        ...
        перерыв;
        дело StarState:
        ...
        перерыв;
        случай TurnState:
    }
}
  
```

```

...
перерыв;
}

```

Теперь, когда у нас есть базовое понимание структуры шаблона State, мы можем приступить к определению состояний и поведения нашего главного героя, в данном случае велосипеда, как мы увидим в следующем разделе.

## Определение состояний переключения

В нашей игре объектом, который чаще всего будет переключаться между состояниями, является наш очный мотоцикл. Это под контролем игрока; он взаимодействует практически со всеми элементами окружающей среды, включая препятствия и вражеские дроны. Поэтому он никогда не останется в одном и том же состоянии надолго.

На следующем диаграмме показан краткий список конечных состояний нашего велосипеда:

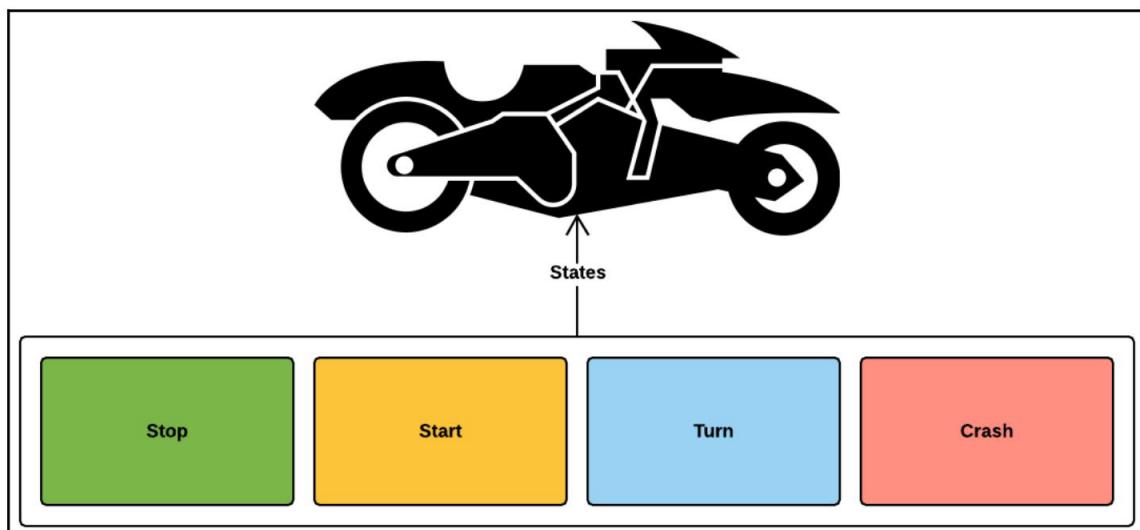


Рисунок 5.2 – Схема, иллюстрирующая конечные состояния велосипеда

А теперь давайте определим ожидаемое поведение для некоторых из перечисленных состояний:

- Старт: В этом состоянии мотоцикл не движется. Его текущая скорость равна нулю. Его шестерни установлены в начальное положение. Если мы решим, что двигатель должен быть включен, но в этом состоянии работать в режиме холостого хода, у нас может быть анимация вибрации мотоцикла, показывающая, что двигатель работает.
- Старт: в этом состоянии мотоцикл движется на полной скорости, а колеса поворачиваются в соответствии с движением вперед.
- Поворот: в состоянии поворота мотоцикл поворачивает влево или вправо, в зависимости от действий игрока.
- Авария: если мотоцикл находится в аварийном состоянии, это означает, что он горит и лежит на боку.

Транспортное средство больше не будет реагировать на действия игрока.

Это всего лишь неотшлифованные определения потенциальных состояний нашего автомобиля. Мы могли бы действовать настолько детализированно и конкретно, насколько пожелаем, но для нашего варианта использования этого достаточно.

Следует иметь в виду, что каждое определение состояния содержит описание поведения и анимацию. Эти требования необходимо будет учитывать, когда мы будем писать и расматривать реализацию шаблона State в следующем разделе этой главы.

## Реализация шаблона State

В этой части главы мы реализуем шаблон State с языком C# для упрощения ожидаемого поведения каждого из конечных состояний транспорта с помощью класса, который мы определили в предыдущем разделе.

Мы собираемся сократить количество написания минималистичных скелетных классов изображений краткости и ясности. Такой подход позволит нам сократить количество написания на структуре шаблона, не увязая в деталях реализации.

Также важно отметить, что версия шаблона State, представленная в этой книге, может быть немного неортодоксальной, поскольку она не сильно отличается от традиционных подходов. Следовательно, если следует рассматривать как пересстановку шаблона State, адаптированную контекст проекта Unity и для конкретного варианта использования.

## Реализация шаблона State

Мы рассмотрим наш пример кода в несколько этапов:

1. Начнем с записи нового интерфейса, который будет реализовывать каждый из наших конкретных классов состояний:

```
прос транс тво имен Chapter.State {  
  
общедоступный интерфейс IBikeState {  
  
void Handle (контроллер BikeController);  
}  
}
```

Следует отметить, что мы передаем экземпляр BikeController в метод Handle(). Этот подход позволяет классам состояний получать доступ к общедоступным свойствам BikeController. Этот подход может немного отклоняться от традиции, поскольку обычно состояние передается объект Context.

Однако ничто не мешает нам передать в классы состояния и объект Context, и экземпляр BikeController. В качестве альтернативы мы могли бы установить ссылку на экземпляр BikeController при инициализации каждого класса состояния.

Однако это было проще сделать, как мы и собираемся делать в данном случае.

2. Теперь, когда у нас есть интерфейс, давайте реализуем класс контекста:

```
прос транс тво имен Chapter.State {  
  
общес твенный класс BikeStateContext {  
  
общес твенный IBikeState currentState {  
  
получать; набор;  
}  
  
частный BikeController только для чтения _bikeController;  
  
public BikeStateContext (BikeController BikeController) {  
  
_bikeController = BikeController;  
}  
  
общес твенный недействительный перех од()  
{
```

```

        CurrentState.Handle(_bikeController);
    }

    public void Transition(состояние IBikeState) {

        ТекущееСостояние =
            состояние; CurrentState.Handle(_bikeController);
    }
}
}
}

```

Как мы видим, класс BikeStateContext предоставляет общедоступное свойство, указывающее на текущее состояние велосипеда; таким образом, он знает о любом изменении состояния. Следовательно, мы могли бы обновить текущее состояние нашей сущности через ее свойство и перейти в него, вызвав метод Transition().

Например, этот механизм полезен, если мы хотим связать состояния вместе, позволяя каждому классу с состоянием обновляться следующий цепочке. Затем мы могли бы циклически переключаться между связанными состояниями, просто вызывая метод Transition() объекта Context. Однако для нашего варианта использования этот подход не подойдет, поскольку мы вызовем перегруженный метод Transition() и просто передадим состояние, в котором хотим перейти.

3. Далее идет BikeController. Этот класс инициализирует объект Context и состояния, а также вызывает изменения состояний:

```

использование UnityEngine;

последование Chapter.State {
    публичный класс BikeController: MonoBehaviour {

        общедоступное число плавающей запятой maxSpeed =
        2.0f; общедоступное число плавающей запятой TurnDistance = 2.0f;

        общественный поплавок CurrentSpeed {get; набор; }

        публичное направление CurrentTurnDirection {
            получить; частный набор;
        }

        частный IBikeState
        _startState, _stopState, _turnState;

        частный BikeStateContext _bikeStateContext;

        частный недействительный Start () {

```

```
_bikeStateContext = новый
    BikeStateContext (это);

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

    _bikeStateContext.Transition(_stopState);
}

общественный недействительный
StartBike () { _bikeStateContext.Transition (_startState);
}

общественный недействительный
StopBike () { _bikeStateContext.Transition (_stopState);
}

общественный недействительный Turn (направление
направления ) {CurrentTurnDirection = направление;
    _bikeStateContext.Transition(_turnState);
}
}

}
```

Если бы мы не инкапсулировали поведение велосипеда внутри отдельных классов состояний, мы, вероятно, реализовали бы их внутри BikeController. Такой подход, с кореей всего, привел бы к раздому классу контроллера, который было бы сложно поддерживать.

Таким образом, используя шаблон State, мы делаем наши классы меньше и проще в обслуживании.

И мы также возвращаем BikeController его первоначальные обязанности по управлению новыми компонентами велосипеда. Он существует для того, чтобы предлагать интерфейс для управления велосипедом, раскрывая его основные траектории с вводом и управлением его структурными зависимостями.

4. Следующие три класса будут нашими состояниями; они вполне самостоятельные  
поля состояния. Обратите внимание, что каждый из них реализует интерфейс IBikeState. Начнем с  
BikeStopState:

```
ис пользование UnityEngine;
просмотр твоим Chapter.State {

общедоступный класс BikeStopState: MonoBehaviour, IBikeState
```

```
{
    частный BikeController _bikeController;

    public void Handle(BikeController BikeController) {

        если (!_bikeController)
            _bikeController = BikeController;

        _bikeController.CurrentSpeed = 0;
    }
}
```

5. Следующий класс с состояния — BikeStartState:

```
использование UnityEngine;

пространствоимен Chapter.State {

    общеслужебный класс BikeStartState: MonoBehaviour, IBikeState {

        частный BikeController _bikeController;

        public void Handle(BikeController BikeController) {

            если (!_bikeController)
                _bikeController = BikeController;

            _bikeController.CurrentSpeed =
                _bikeController.maxSpeed;
        }

        недействительное Обновление
        {
            если (_bikeController)
            {
                если (_bikeController.CurrentSpeed > 0) {

                    _bikeController.transform.Translate(
                        Vector3.forward *
                        (_bikeController.CurrentSpeed * Time.deltaTime));
                }
            }
        }
    }
}
```

6. И наконец, есть BikeTurnState, который заставляет байк поворачивать влево или вправо:

```
использование UnityEngine;
пространство имен Chapter.State {
    публичный класс BikeTurnState: MonoBehaviour, IBikeState {
        частный Vector3 _turnDirection; частный
        BikeController _bikeController;

        public void Handle(BikeController BikeController) {
            если (!_bikeController)
                _bikeController = BikeController;

            _turnDirection.x = (float)
                _bikeController.CurrentTurnDirection;

            если (_bikeController.CurrentSpeed > 0) {
                Transform.Translate(_turnDirection *
                    _bikeController.turnDistance);
            }
        }
    }
}
```

7. В нашем последнем классе BikeController создается набор направлений с именем Direction, который мы реализуем здесь:

```
пространство имен Chapter.State
{
    public enum Direction {
        Слева = -1,
        Вправо = 1
    }
}
```

Теперь у нас есть все ингредиенты, готовые для тестирования реализации шаблона State.

## Тестирование реализации шаблона состояния

Чтобы быстрее протестировать нашу реализацию шаблона State в вашем собственном примере Unity, вам необходимо выполнить следующие шаги:

1. Скопируйте все скрипты, которые мы только что расмотрели, в свой проект Unity.
2. Создайте новую пустую сцену.
3. Добавьте на сцену 3D GameObject, например куб, убедившись, что он виден с новой камеры.
4. Привяжите скрипт BikeController к GameObject.
5. Также привяжите к GameObject следующий клиентский скрипт:

использование UnityEngine;

```
последовательность имен Chapter.State {  
  
общественный класс ClientState: MonoBehaviour {  
  
частный BikeController _bikeController;  
  
недействительный Старт  
(){  
    _bikeController =  
        (Велосипедный Контроллер)  
        FindObjectOfType(typeof(BikeController));  
}  
  
недействительный OnGUI()  
{  
    if (GUILayout.Button("Запустить велосипед"))  
        _bikeController.StartBike(); if  
        (GUILayout.Button("Повернуть налево"))  
            _bikeController.Turn(Направление.Влево);  
    if (GUILayout.Button("Повернуть направо"))  
        _bikeController.Turn(Направление.Вправо);  
    if (GUILayout.Button("Остановить велосипед"))  
        _bikeController.StopBike();  
}  
}
```

После запуска сцены вы должны увидеть на экране следующие кнопки графического интерфейса, которые вы можете использовать для управления GameObject, вызывая изменения состояния:

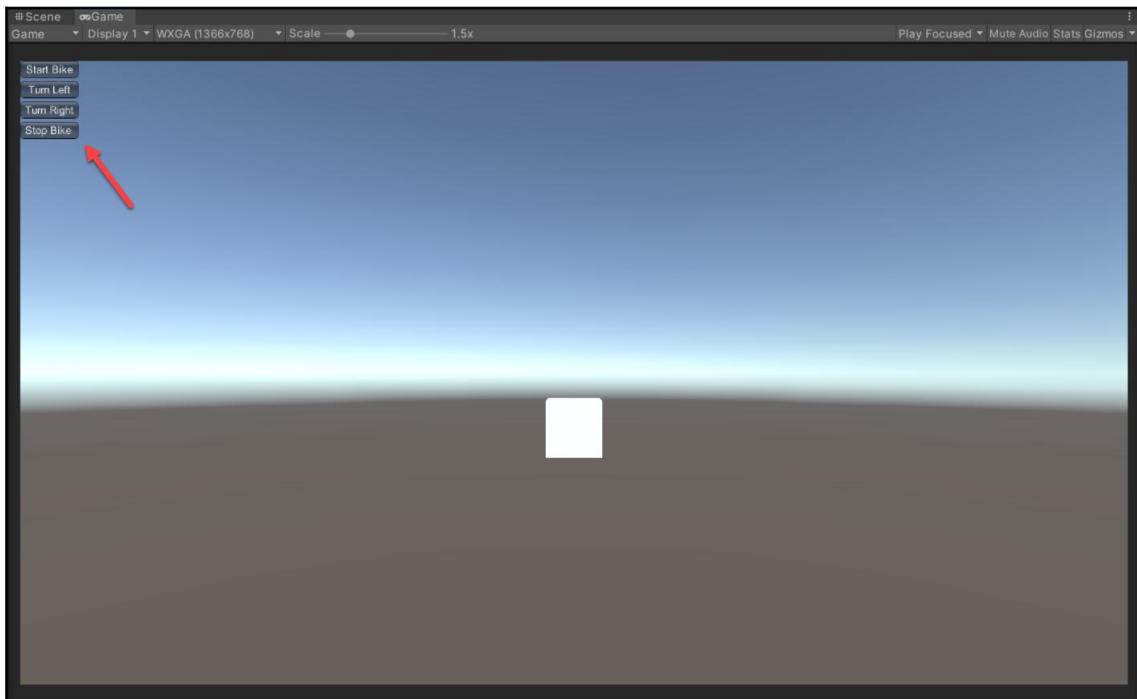


Рисунок 5.3 – Скриншот примера кода в действии

В следующем разделе книги мы с обираемся рассмотреть преимущество шаблона State, но также его ограничения.

## Преимущества и недостатки шаблона State

Ниже приведены преимущества использования шаблона State:

- Инкапсуляция: шаблон State позволяет нам реализовать отслеживание состояния объекта поведения как набор компонентов, которые можно динамически назначать объекту, когда он меняется состояния.
- Сопровождение: мы можем легкореализовать новые состояния без необходимости изменения длинные условные операторы или раздувые классы.

Однако шаблон State имеет следующие ограничения, когда мы используем его для управления анимированным персонажем.

Вот краткий список потенциальных ограничений:

- Смещение. В своей исходной форме шаблон State не предлагает решения для смещивания анимаций. Это ограничение может стать проблемой, если вы хотите добиться плавного визуального перехода между анимированными состояниями персонажа.
- Перехват. В нашей реализации шаблона мы можем легко переключаться между состояниями, но мы не определяем связь между ними. Следовательно, если мы хотим определить переходы между состояниями на основе отношений и условий, нам придется написать гораздо больше кода; например, если я хочу, чтобы состояние ожидания перешло в состояние ходьбы, затем состояния ходьбы перешло в состояние работы. Это происходит автоматически и плавно, туда и обратно, в зависимости от триггеров или состояний. Это может занять много времени в коде.

Однако перечисленные выше ограничения можно преодолеть, используя систему анимации Unity и ее собственный конечный автомат. Мы можем легко определить состояния анимации и пркрепить анимационные клипы и сценарии к каждому из настроенных состояний. Но гораздо важней особенность: я вляется то, что он позволяет нам определять и настраивать набор переходов между состояниями с условиями и триггерами через визуальный редактор, как мы видим здесь:

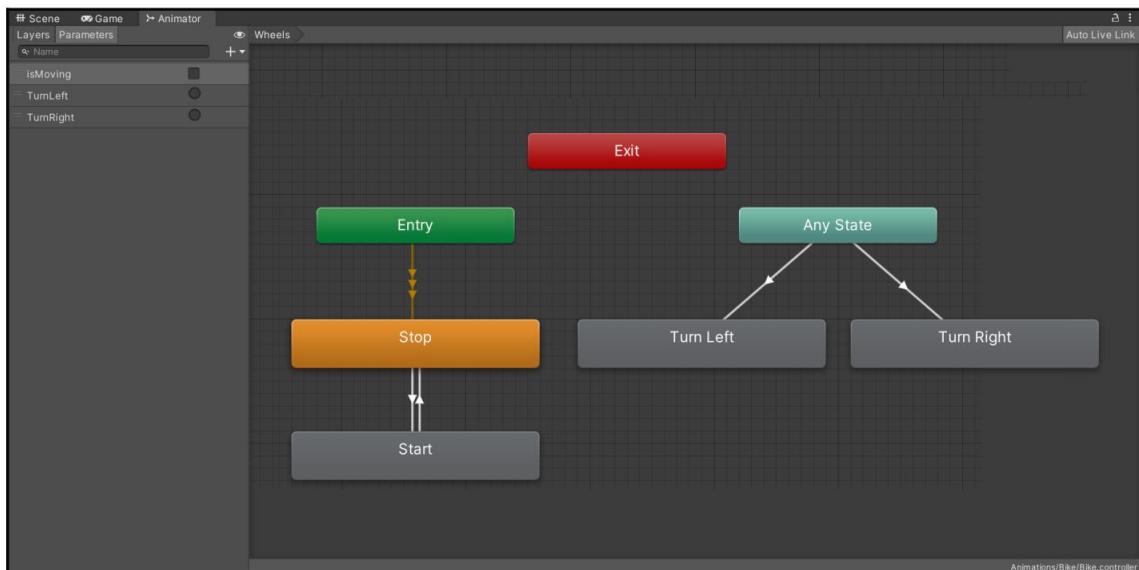


Рисунок 5.4 – Скриншот редактора системы анимации Unity

Присутствующие прямые могут быть предstawлены отдельные состояния анимации, а треки указывающие отношения и переходы. Глубокое погружение в использование анимации Unity выходит за рамки этой книги. Целью этой главы было знакомство с шаблоном State в контексте движка Unity. Как мы видим, Unity предлагает нам собственное решение, которое мы можем использовать для управления состояниями анимации наших персонажей.

Важно помнить, что этот инструмент не ограничивается анимацией гуманоидных персонажей.

Мы могли бы использовать его для механических объектов, таких как автомобили, или даже для фоновых ингредиентов, таких как торговый автомат; следовательно, все, что имеет состояние, анимационное поведение.

В следующем разделе мы рассмотрим краткий список альтернативных шаблонов, которые следует учитывать перед использованием шаблона «Состояние».



Для получения дополнительной информации о системе анимации Unity вы можете прочитать официальную документацию по следующему ссылке: <https://docs.unity3d.com/Manual/AnimationSection.html>.

## Рассмотрение альтернативных решений

Ниже приведен список шаблонов, которые связанны или являются альтернативами шаблону State:

- **Деревья поведения.** Если вы планируете реализовать ложное поведение ИИ для персонажей NPC, я бы рекомендовал рассмотреть такие шаблоны, как дерево, или такие концепции, как деревья поведения (BT). Например, если вам нужно реализовать ИИ с динамическим поведением при принятии решений, то BT — более подходящий подход, поскольку он позволяет реализовать поведение с использованием дерева действий.
- **FSM:** Вопрос, который часто возникает при обсуждении шаблона «Состояние», — это основное различие между FSM и шаблоном «Состояние». Быстрый ответ: шаблон State занимается инкапсуляцией поведения объекта, зависящего от состояния. Однако автомат более глубоко вовлечен в переходы между конечными состояниями на основе определенных вех одних тригеров. Таким образом, FSM часто считается более подходящим для реализации автоматоподобных систем.
- **Memento:** Memento показан шаблон State, но имеет дополнительную функцию, которая дает объектам возможность откатиться к предыдущему состоянию. Этот шаблон может быть полезен при реализации системы, которой требуется возможность отменить внесенные в нее изменения.

## Краткое содержание

В этой главе мы могли использовать шаблон State для определения и реализации поведения нашего главного героя с со временем состояния. В нашем случае персонаж — трансформное представление. Рассмотрев его структуру в контексте конкретного примера кода, мы увидели его ограничения при работе с анимированными объектами. Однако Unity предлагает нам собственное решение, которое позволяет нам управлять состояниями анимированных персонажей с помощью ложного конечного автомата и визуального редактора.

Однако это не означает, что шаблон State сам по себе бесполезен в Unity. Мы могли бы легко использовать его в качестве основы для создания систем или механизмов с отслеживанием состояния.

В следующей главе мы определим глобальные состояния нашей головной игры и будем управлять ими с помощью Event Bus.

# 6

## Управление игровыми событиями с помощью шин с событий

Шина с событий действует как центральный узел, который управляет определенным списком глобальных с событий, на которые объекты могут подписываться или публиковать. Это самый простой шаблон управления с событиями, который есть в моем наборе инструментов. Это скращает процесс назначения роли подписчика или издателя объекту до одной строки кода. Как вы можете себе представить, это может быть полезно, когда вам нужны быстрые результаты. Но, как и большинство простых решений, оно имеет некоторые недостатки и ограничения, которые мы рассмотрим далее.

В примере кода, представленном в этой главе, мы будем использовать шину с событий для трансляции определенных с событий гонки и компонентам, которым необходимо прослушивать изменения в общем состоянии гонки. Поэтому важно иметь в виду; Я предлагаю использовать шину с событий в качестве решения для управления глобальными гоночными событиями из-за ее простоты, а не масштабируемости. Так что, возможно, это не лучшее решение во всех случаях, но это один из наиболее простых в реализации шаблонов, как мы увидим в следующих разделах.

В этой главе будут рассмотрены следующие темы:

- Обзор шаблона шины с событий
- Реализация гоночного автобуса



Для краткости я сноси в эту главу упрощенные примеры скелетного кода. Если вы хотите просмотреть полную реализацию шаблона в контексте реального игрового проекта, откройте папку FPP в проекте GitHub. Ссылку вы найдете в разделе «Технические требования».

## Технические требования

Мы также будем использовать следующие конкретные функции API движка Unity:

- Статический
- UnityEvents
- UnityActions

Если вы не знакомы с этими концепциями, просмотрите главу 3 «Краткое руководство по программированию Unity».

Файлы кода для этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/>.

главная /Активы/Главы/Глава06.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/2U7wrCM>.



Если у вас возникли проблемы с пониманием механизма делегатов, имейте в виду, что они похожи на указатели функций в C/C++. Понятие `delegate`, делегат указывает на метод. Адеделегаты часто используются для реализации обработчиков событий и обратных вызовов. Действие — это тип делегата, который можно использовать с методом, имеющим возвращаемый тип `void`.

## Понимание шаблона шины с событий

Когда событие вызывается объектом (издателем), он отправляет сигнал, который может получить другие объекты (подписчики). Сигнал имеет форму уведомления, которое может указывать на возникновение действия. В системе с событий объект транслирует событие. Только те объекты, которые подписываются на него, будут уведомлены и могут выбрать, как с этим обращаться. Итак, мы можем представить это как внезапный всплеск радиосигнала, который может обнаружить только те, у которых антенны настроены на определенную частоту.

Шаблон «Шина с событий» является близким родственником шаблонов «Система обмена сообщениями» и «Публикация-подписка», причем последний является более точным названием того, что делает шина с событий. Ключевое слово в названии этого шаблона — термин «шина». Говоря простым языком, вычислений, шина — это соединение между компонентами. В контексте этой главы компонентами будут объекты, которые могут быть издателями или прослушивателями с событий.

Следовательно, шина с событий — это способ обединения объектов посредством с событий с использованием модели публикации-подписки. Подобную модель можно реализовать с помощью такого шаблона, как Observer и связанные с событием C#. Однако эти альтернативы имеют некоторые недостатки. Например, в типичной реализации шаблона Observer может возникнуть тесная связь, поскольку наблюдатели (слушатели) и объекты (издатели) могут стать зависимыми и знать друг друга.

Но шина с событий, по крайней мере в том виде, в котором мы ее реализуем в Unity, абстрагирует и упрощает отношения между издателями и подписчиками, поэтому они совершенно не знают друг о друге. Еще одним преимуществом является то, что он скращивает процесс назначения роли издателя или подписчика для одной строки кода. Таким образом, Event Bus — это ценный образец, который стоит изучить и иметь при себе. Как вы можете видеть на следующей диаграмме, он действует как посредник между издателями и подписчиками:

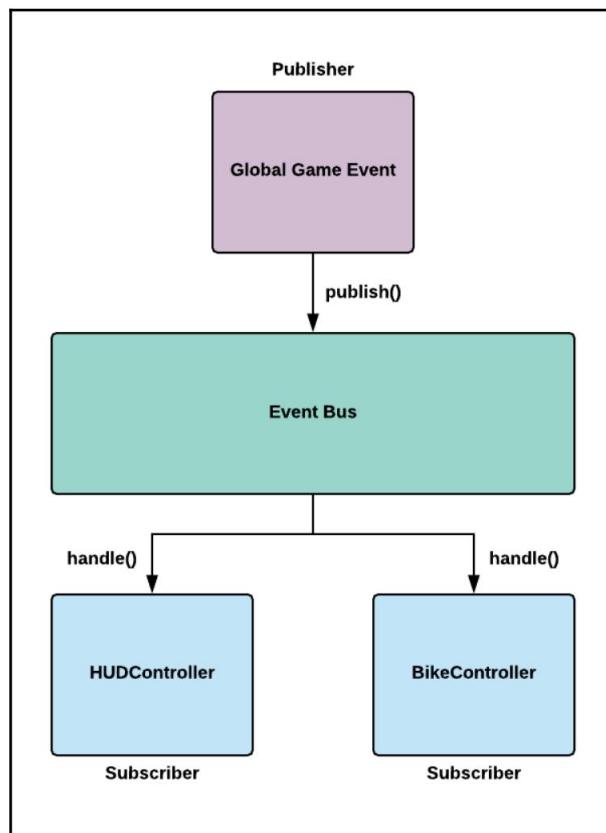


Рисунок 6.1 – Схема шаблона шины с событий

Как видно из диаграммы, есть три основных ингредиента:

- Издатели: эти объекты могут публиковать подписчикам определенные типы с событий, объявленные шиной с событий.
- Шина с событий: этот объект отвечает за координацию передачи с событий между издателями и подписчиками.
- Подписчики: эти объекты регистрируются как подписчики определенных с событий через шину с событий.

## Преимущества и недостатки шаблона Event Bus

Преимущества шаблона Event Bus заключаются в следующем:

- Развязка. Основное преимущество использования системы с событий заключается в том, что она отделяет ваши объекты. Объекты могут взаимодействовать посредством с событий, а не напрямую ссылаясь друг на друга.
- Простота: шина с событий обес печивает простоту, абстрагируя механизм публикации или подписки на событие от своих клиентов.

Недостатки шаблона Event Bus заключаются в следующем:

- Производительность. Под капотом любой системы с событий находится низкоуровневый механизм, который управляет обменом сообщениями между объектами. Таким образом, при использовании системы с событий может возникнуть небольшое снижение производительности, но в зависимости от вашей целевой платформы оно может быть незначительным.
- Глобальное: в этой главе мы реализуем шину с событий сстатическими методами и свойствами, чтобы упростить доступ из любого места нашего кода. При использовании глобально доступных переменных и состояний всегда существует риск, поскольку они могут затруднить отладку и модульное тестирование. Однако это весьма контекстуальный недостаток, а не абсолютный.



UnityEvents фактически может принимать до четырех аргументов универсального типа. Это может позволить передавать подписчикам данные, специфичные для события. Дополнительные сведения см. в следующем разделе документации Unity API: <https://docs.unity3d.com/2021.2/Documentation/ScriptReference/Events.UnityEvent.html>.

## Как использовать шину с событий

Раньше я использовал Event Bus для следующих целей:

- Быстрое прототипирование: я часто использую шаблон Event Bus при быстром прототипировании новой игровой механики или функций. С помощью этого шаблона я могу легко создать компоненты, которые запускают поведение друг друга с помощью событий, с которых я при этом их развяжу.
- Этот шаблон позволяет нам добавлять и удалять объекты в качестве подписчиков или издателей с помощью одной строки кода, что всегда полезно, когда вы хотите быстро и легко создать прототип чего либо.
- Продуктивный код: я использую Event Bus в производственном коде, если не могу найти уважительную причину для реализации более сложного подхода к управлению событиями. Этот шаблон хорошо справляется со своей задачей, если вам нужно обрабатывать ложные типы событий или структуры.

Я бы не стал использовать глобальную шину с событий, подобную той, что представлена в этой главе, для управления событиями, не имеющими «глобальной области действия». Например, если у меня есть компонент пользователя с его интерфейсом в HUD, который отображает предупреждение о повреждении, когда игрок сопткается с препятствием, было бы неэффективно блокировать событие с толкновения через шину с событий, поскольку это локализованное взаимодействие между велосипедом и объектом на треке. Вместо этого я бы использовал что-то вроде шаблона Observer, потому что в этом случае мне нужен только один компонент пользователя с его интерфейсом, чтобы наблюдать за конкретным изменением состояния объекта, в данном случае велосипеда. Как правило, каждый раз, когда вам нужно реализовать систему, использующую события, просмотрите список известных шаблонов, выберите тот, который подходит для вашего варианта использования, но не всегда возвращайтесь к самому простому варианту, который, по моему мнению, это Event Bus.

## Управление глобальными и очочными с событиями

Проект, над которым мы работаем, представляет собой глобальную игру, и большинство его этапов разбиты на этапы.

Ниже приводится краткий список типичных глобальных этапов:

- Обратный отсчет: на этом этапе мотоцикл остается залипанием с тарта, пока идет таймер обратного отсчета.
- Начало гонки: как только часы дойдут до нуля, загорится зеленый световой сигнал, и мотоцикл начнет двигаться вперед потраске.
- Финиш гонки: в тот момент, когда игрок пересекает финишную линию, гонка заканчивается.

Между стартом и финишем гонки могут произойти определенные события, которые могут изменить текущее состояние гонки:

- Паузы в гонке: игрок может пристановить игру, продолжая участвовать в гонке.
- Выход из гонки: игрок может выйти из гонки в любой момент.
- Отстановка гонки: гонка может внезапно прекратиться, если игрок попадет в аварию смертельным исходом.

Поэтому мы хотим транслировать уведомление, если наизириующее о наступлении каждого этапа гонки и любого другого важного события между ними, которое изменит общее состояние гонки. Этот подход позволит нам запускать определенное поведение компонентов, которые должны вести себя определенным образом в зависимости от текущего контекста гонки.

Например, вот некоторые компоненты, которые будут обновляться в зависимости от конкретного состояния гонки:

- HUD: индикатор статуса гонки будет меняться в зависимости от контекста HUD гонки (проектационный дисплей).
- RaceTimer: Таймер гонки запускается только в начале гонки и останавливается, когда игрок пересекает финишную линию.
- BikeController: контроллер велосипеда отпускает тормоза велосипеда после начала гонки. Этот механизм не позволяет игроку выйти на трассу с зеленого света.
- InputRecorder: в начале гонки система воспроизведения ввода начинает записывать действия игрока, чтобы их можно было воспроизвести позже.

Все эти компоненты имеют определенное поведение, которое должно запускаться на определенных этапах гонки. Поэтому мы будем использовать Event Bus для реализации этих глобальных гонок.

## Реализация шины гоночных событий

Мы собираемся реализовать Race Event Bus в два простых шага:

1. Для начала нам нужно представить информацию о конкретных типах событий гонки, которые мы поддерживаем, что мы и сделаем с помощью следующего перечисления:

```
представление имен Chapter.EventBus {
    общесобытийное перечисление RaceEventType {
        ОБРАТНЫЙ_ОСЧЕТ, СТАРТ, ПРЕЗАПУСК, ПАУЗА_СТОП, ОКОНЕНИЕ, ВЫХОД
    }
}
```

Важно отметить, что предыдущие значения перечисления предстают с собой конкретные с события, описывавшие этапы гонки, от начала до конца. Поэтому мы ограничиваемся обработкой с событий только в глобальном масштабе.

2. Следующая часть — это новый компонент шаблона, фактический класс шины игровых с событий, который мы назовем RaceEventBus, чтобы обес печить более специфичное для предметной области соглашение об именах наших классов:

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

прототипы имен Chapter.EventBus {

общественный класс с RaceEventBus
{
    частный статический только для чтения
    IDictionary<RaceEventType, UnityEvent>
    События = новый словарь<RaceEventType, UnityEvent>();

    public static void Подписьться
        (RaceEventType eventType, прослушиватель UnityAction) {
        UnityEvent thisEvent;

        if (Events.TryGetValue(eventType, out thisEvent)) {
            thisEvent.AddListener(прослушиватель);

        } Ещё
            {thisEvent = новый UnityEvent();
            thisEvent.AddListener(прослушиватель);
            Events.Add(eventType, thisEvent);
        }
    }

    public static void Отменить подписку
        (Тип RaceEventType, прослушиватель UnityAction) {
        UnityEvent thisEvent;

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

    public static void Publish (тип RaceEventType) {
        UnityEvent thisEvent;
```

```

если (Events.TryGetValue(type, out thisEvent)) { thisEvent.Invoke();

}

}
}

}
}

```

Ключевым компонентом нашего класса является словарь с событий. Это действует как реестр, в котором мы храним списки отношений между типами событий и подписчиками. Сразу я его конфиденциальным и доступным только для чтения, мы гарантируем, что он не может быть перезаписан другим объектом напрямую.

Таким образом, клиент должен вызывать общий досгупный статический метод `Subscribe()`, чтобы добавить себя в качестве подписчика с событием определенного типа. Метод `Subscribe()` принимает два параметра; первый — это тип с событием гонки, второй — функция обратного вызова. Поскольку `UnityAction` является типом делегата, он дает нам возможность передать метод в качестве аргумента.

Следовательно, когда клиентский объект вызывает метод `Publish()`, зарегистрированные методы обратного вызова всех подписчиков определенного типа с событием гонки будут вызваны одновременно.

Метод `Unsubscribe()` не требует поля `on`, поскольку позволяет объектам удалять себя из числа подписчиков определенного типа с событием. Таким образом, их методы обратного вызова не будут вызываться шиной с событием, когда объект публикует с событием.

Если это все еще выглядит абстрактно, это станет ясно, когда мы реализуем клиентские классы в следующем разделе и посмотрим, как мы можем использовать шину с событием для запуска поведения объектов в определенные моменты в правильной последовательности.

## Тестирование гоночного автобуса

Теперь, когда у нас есть основные элементы шаблона, мы можем написать код для тестирования нашего класса `RaceEventArgs`. Для краткости я удалил весь код поведения в каждом клиентском классе, чтобы сократить внимание на использовании шаблона:

- Для начала мы собираемся написать таймер обратного отсчета, который подписывается на тип с событием гонки `COUNTDOWN`. Как только с событием `COUNTDOWN` будет опубликовано, оно запустит 3-секундный обратный отсчет до начала гонки. И в тот момент, когда отсчет достигнет конца, будет опубликовано событие `START`, сигнализирующее о начале гонки:

```

использование UnityEngine;
использование System.Collections;

```

```
протрансивоимен Chapter.EventBus {  
  
    публичный класс с CountdownTimer: MonoBehaviour {  
  
        частный плавающий _currentTime;  
        продолжительность частного плавающего режима = 3.0f;  
  
        void OnEnable()  
  
            { RaceEventBus.Subscribe( RaceEventType.COUNTDOWN, StartTimer);  
            }  
  
        void OnDisable()  
            { RaceEventBus.Unsubscribe(  
                RaceEventType.COUNTDOWN, StartTimer);  
            }  
  
        частный недействительный StartTimer() {  
            StartCoroutine(Обратный отсчет());  
        }  
  
        частный IEnumerator Countdown () {_currentTime =  
            продолжительность;  
  
            while (_currentTime > 0) { дох односТЬ return  
                new WaitForSeconds(1f); _Текущее время --;  
  
            }  
  
            RaceEventBus.Publish(RaceEventType.START);  
        }  
  
        void OnGUI () { GUI.color  
            = Color.blue; GUI.Label(new Rect(125, 0,  
            100, 20),  
            "ОБРАТНЫЙ ВНИЗ: " + _currentTime);  
  
        }  
    }  
}
```

Наиболее важные строки кода в предыдущем классе следующие:

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

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

Каждый раз, когда объект CountdownTimer включается, вызывается метод Subscribe(). А когда происходит обратное и его отключают, он отписывается сам. Мы делаем это, чтобы гарантировать, что объект прослушивает событие, когда он активен, или не вызывается RaceEventBus при отключении или уничтожении.

Метод Subscribe() принимает два аргумента — тип события и функцию обратного вызова. Это означает, что метод StartTimer() CountdownTimer будет вызываться RaceEventBus каждый раз, когда публикуется событие COUNTDOWN.

2. Далее мы реализуем скелет класса BikeController для тестирования с событием СТАРТ и СТОП:

```
использование UnityEngine;

пространство имён Chapter.EventBus {

    публичный класс BikeController: MonoBehaviour {

        частная строка _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 );
        }
    }
}
```

```
частный недейс твительный StartBike
() { _status = «Начало»;
}

частный недейс твительный StopBike
() { _status = «Остановлен»;
}

недейс твительный OnGUI() {
    GUI.цвет = Цвет.зеленый;
    GUI.Label(new
        Rect(10, 60, 200, 20), "СТАТУС
        ВЕ ЛОС ИГРЫ: + _status);
}
}
```

3. Итак, мы напишем класс HUDController. Это мало что дает, кроме отображения кнопки, позволяющей установить флагок после ее начала:

```
использование UnityEngine;

протранс твоимен Chapter.EventBus {

    публичный класс HUDController: MonoBehaviour {

        частный bool _isDisplayOn;

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

        void OnDisable()

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

        частный недейс твительный DisplayHUD
        () { _isDisplayOn = true;
        }

        недейс твительный OnGUI()
        если (_isDisplayOn) {

            if (GUILayout.Button("Остановить флагок")) {

                _isDisplayOn = ложь;
            }
        }
    }
}
```

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

4. Чтобы проверить последовательность с событий, нам нужно прокомментировать следующий клиентский класс с Gameobject в пусть сцене Unity. Благодаря этому мы можем запустить таймер обратного отсчета

```
использование UnityEngine;

последовательность Chapter.EventBus {

общественный класс ClientEventBus: MonoBehaviour {

частный bool _isEnabled;

недействительный Старт
{
    gameObject.AddComponent<HUDController>();
    gameObject.AddComponent<CountdownTimer>();
    gameObject.AddComponent<BikeController>();

    _isEnabled = правда;
}

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

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

частный недействительный Restart()
{
    _isEnabled = правда;
}

недействительный OnGUI()
{
    если (_isEnabled)
```

```
{  
    if (GUILayout.Button("Начать обратный отсчет")) {  
  
        _isEnabled = false;  
        RaceEventBus.Publish(RaceEventType.COUNTDOWN);  
    }  
}  
}  
}
```

Предыдущий пример может показаться слишком прошвенным, но его цель — продемонстрировать, как мы можем запускать поведение отдельных объектов в определенной последовательности с событиями, с охраняя при этом их развязку. Ни один из объектов не взаимодействует друг с другом напрямую.

Их единственная общая точка отсчета — RaceEventBus. Поэтому мы могли бы легко добавить больше подписчиков и издателей; например, мы могли бы добавить объект TrackController прослушивать событие RESTART, чтобы знать, когда следующий бросок гонки наступает. Таким образом, теперь мы можем упорядочить срабатывание поведения основных компонентов с событиями, каждое из которых представляет определенный этап гонки.



Действие — это делегат, указывающий на метод, который принимает один или несколько аргументов, но не возвращает значения. Например, вам следует использовать Action, когда ваш делегат указывает на метод, который возвращает void. UnityAction ведет себя как Actions в собственном C#, за исключением того, что UnityActions был разработан для использования с UnityEvents.

## Обзор реализации шины с событий

Используя шину с событиями, мы можем инициализировать поведение, с охраняя при этом развязку основных компонентов. Нам легко добавлять или удалять объекты в качестве подписчиков или издателей.

Мы также определили конкретный список глобальных событий, которые представляют каждый этап гонки.

Таким образом, теперь мы можем начать определять последовательность и запускать поведение основных компонентов от начала до конца гонки и все это, что между ними.

В следующем разделе мы рассмотрим некоторые альтернативные решения для шины с событий, причем каждое решение предлагает свой подход, который может быть лучшим решением в зависимости от контекста.

## Рассмотрение некоторых альтернативных решений

Системы и шаблоны с событий — это обширная тема, и мы не можем подробно осветить ее в этой книге. Поэтому мы подготовили краткий список шаблонов, которые следует учитывать при реализации системы или механизма с событий, но имейте в виду, что существует гораздо больше, и мы призываем вас как читателя продолжить изучение этой темы за пределами ограниченных рамок этой книги:

- **Наблюдатель:** стандартный, но полезный шаблон, в котором объект (объект) поддерживает список объектов (наблюдателей) и уведомляет их об изменении внутреннего состояния. Это шаблон, который следует учитывать, когда вам нужно установить связь «один к многим» между группой сущностей.
- **Очень с событий:** этот шаблон позволяет нам хранить события, созданные издателями, в очередь и пересыпать их со временем. Этот подход отделяет временные отношения между издателями и подписчиками.
- **ScriptableObjects:** в Unity можно создать систему с событий с помощью ScriptableObjects. Ключевым преимуществом этого подхода является то, что он упрощает создание новых пользовательских игровых событий. Если вам нужно создать масштабируемую наращивающую систему с событий, возможно, это то, что вам нужно.



Если вы спрашиваете себя, почему мы не демонстрируем в этой книге более продвинутые системы с событий, в том числе реализованные с помощью ScriptableObjects, ответ заключается в том, что основная цель этой книги — познакомить читателей с шаблонами проектирования, а не перегружать их сложностью. Мы предлагаем первый шаг введения в новые концепции, но призываем читателей исследовать более продвинутый материал по мере их продвижения.

## Краткое содержание

В этой главе мы рассмотрели шину с событий — простой шаблон, который упрощает процесс публикации и подписки на события в Unity. Это инструмент, который помогает при быстром создании прототипов или когда у вас есть определенный список глобальных событий, которыми нужно управлять. Однако у него есть свои пределы использования, и всегда полезно изучить другие варианты, прежде чем перейти к использованию лобального дистрибутивной шины с событий.

В следующей главе мы реализуем систему, которая позволит нам производить действия игрока. Многие игровые системы имеют функции повтора и перемотки назад, и с помощью шаблона «Команда» мы попытаемся создать такую игру с нуля.

# 7

## Реализуйте систemu воспроизведения с помощью шаблона к

В этой главе мы будем использовать классический шаблон проектирования под названием Command для реализации системы повторов для нашей гоночной игры. Егомеханизм будет записывать вх одные данные контроллера игрока и соответствующую временную метку. Этот подход позволит нам воспроизводить записианные вх одные данные в правильном порядке и с правильным таймингом.

Функции повтора часто не обходятся в гоночных играх. С помощью шаблона «Команда» мы собираемся реализовать эту систему максимально модульным способом. Этот шаблон предлагает механизм, который позволяет инкапсулировать информацию необходиющую выполнения «действия» или запуска изменения состояния. Он также отделяет запрашивающее «действие» от объекта, который его выполнит. Эта инкапсуляция и развязка позволяют нам ставить запросы на действие в очередь, чтобы мы могли выполнить их позже.

Все это может показаться очень абстрактным, но как только мы реализуем основные компоненты системы, все станет ясно.

В этой главе будут рассмотрены следующие темы:

- Понимание шаблона команды
- Проектирование системы повторов
- Реализация системы повторов
- Рассмотрение альтернативных решений

## Технические требования

Следующая глава имеет практический характер, поэтому вам необходимо иметь базовое представление о Unity и C#.

Файлы кода для этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter07>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/3wAWYpb>.

## Понимание шаблона команды

Представьте себе платформер, в котором вы можете перепрыгивать препятствия, нажимая клавишу пробела. В этом сценарии каждый раз, когда вы нажимаете эту клавишу ввода, вы просите персонажа на экране изменить состояние и выполнить прыжок. В коде мы могли бы реализовать это с помощью InputHandler, который будет прослушивать ввод пробела от игрока, и когда игрок нажимает на него, мы вызываем CharacterController, чтобы вызвать действие прыжка.

Следующий очень упрощенный псевдокод демонстрирует то, что мы имеем в виду:

```
использование UnityEngine;
использование System.Collections;
```

```
публичный класс InputHandler: MonoBehaviour {
```

```
недокументированное Обновление
{
    if (Input.GetKeyDown("пробел")) {
        КонтроллерПерсонажа.Прыжок();
    }
}
```

Как мы видим, этот подход может выполнить свою работу, но если бы мы захотели записать, отменить или воспроизвести ввод от игрока, это могло бы усомниться. Однако шаблон «Команда» позволяет нам отделить объект, который вызывает операцию от объекта, который знает, как ее выполнить. Другими словами, нашему InputHandler не нужно знать, какое именно действие необходимо предпринять, когда игрок нажимает пробел. Ему просто нужно убедиться, что выполняется правильная команда, и позволить механизму шаблона команд творить чудеса за нас.

Следующий код показывает разницу в том, как мы реализуем InputHandler при использовании шаблона Command:

```
использование UnityEngine;
использование System.Collections;
```

```
публичный класс InputHandler : MonoBehaviour {

    [SerializedField] частный
    контроллер_characterController;

    частная команда_spaceButton;

    недействительный старт()
    {
        _spaceButton = новая JumpCommand();
    }

    недействительное обновление
    {
        if (Input.GetKeyDown("пробел"))
            _spaceButton.Execute(_characterController);
    }
}
```

Как мы видим, мы просто вызываем CharacterController напрямую когда игрок нажимает пробел. Фактически мы инкапсулируем всю информацию о необходимости для выполнения действия перед тем, как она попадет в объект, который можем поместить в очередь и повторно вызвать позже.

Давайте рассмотрим следующую диаграмму, которая иллюстрирует реализацию шаблона «Команда»:

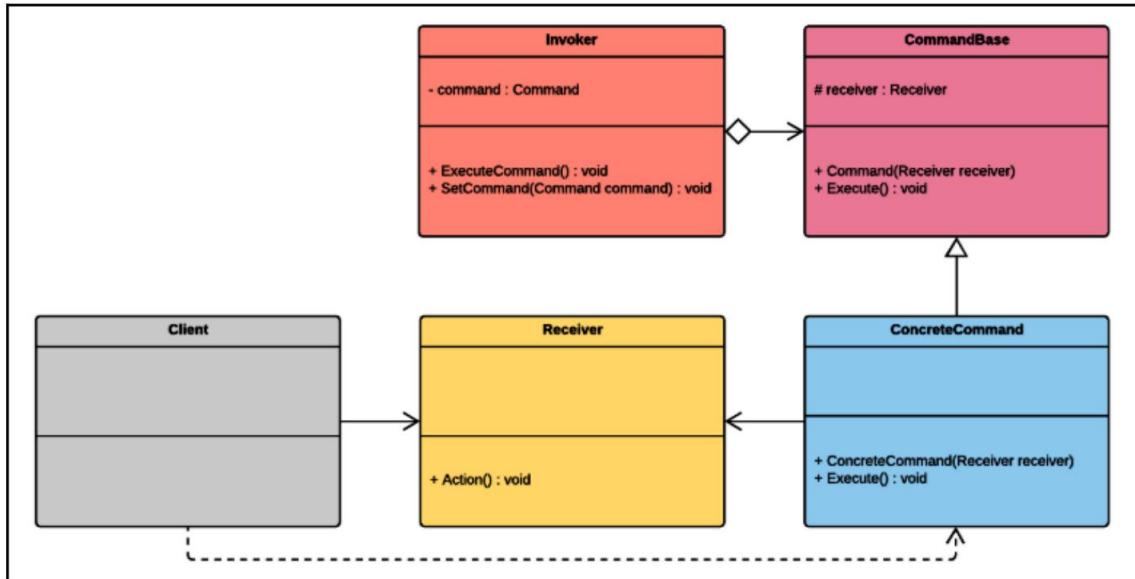


Рисунок 7.1 – UML-диаграмма шаблона «Команда»

Попытка узнать о шаблоне «Команда» путем просмотра диаграммы — не правильный подход, но он помогает нам изолировать функциональные классы, участвующие в этом шаблоне:

- Invoker — это объект, который знает, как выполнить команду, а также может вести учет выполненных команд.
- Ресивер — это тип объекта, который может получать команды и выполнять их.
- CommandBase — это абстрактный класс, который должен наследовать отдельный класс ConcreteCommand, и он предоставляет метод Execute(), который Invoker может вызывать для выполнения определенной команды.

Каждый участник модели имеет определенную ответственность с воротами. Надежная реализация шаблона «Команда» должна позволить нам инкапсулировать запросы действий в виде объекта, который можно поставить в очередь и выполнить немедленно или позже.



Шаблон «Команда» является частью семейства шаблонов «Поведение», его близкими родственниками являются Memento, Observer и Visitor. Эти типы шаблонов часто связаны с распределением обязанностей и тем, как объекты взаимодействуют друг с другом.

## Преимущества и недостатки шаблона «Команда»

Вот некоторые преимущества шаблона «Команда»:

- Развязка: шаблон позволяет отделить объект, который вызывает операцию от объекта, который знает, как ее выполнить. Этот уровень разделения позволяет добавить посредника, который может вести бухгалтерский учет и определять последовательность операций.
- Упорядочение: шаблон «Команда» упрощает процесс установки в очередь вводимых пользователем данных, что позволяет реализовать функции отмены/повтора, макросы и очистки команд.

Это потенциальный недостаток шаблона Command:

- Сложность: для реализации этого шаблона требуется множество классов, поскольку каждая команда сама себе является классом. И требуется хорошее понимание шаблона, чтобы поддерживать код созданный с его помощью. В большинстве случаев это не проблема, но если вы используете шаблон «Команда» без конкретной цели, он может стать ненужным уровнем сложности и мгновенно в вашей кодовой базе.



**Преимущества и недостатки** обычно зависят от контекста; те, что представлены в этой книге, являются общими, а не абсолютными. Поэтому очень важно при выборе шаблона проанализировать его преимущества и недостатки в контексте вашего собственного проекта, а не рассматривать или отвергать тот или иной шаблон на основе общих принципов. Заявления.

## Когда использовать шаблон команды

Вот краткий список возможных вариантов использования шаблона «Команда»:

- Отмена: реализация системы отмены/повтора, которую можно найти в большинстве текстовых и графических редакторов.
- Макрос: система записи макросов, с помощью которой можно записывать последовательность действующих или защитных комбинаций. Затем назначьте их клавиши ввода, чтобы они выполнялись автоматически.

- Автоматизация: автоматизируйте процессы или поведение, записывая набор команд, которые бот будет автоматически последовательно выполнять.

В заключение отметим, что это хороший образец для функций, связанных с хранением, с их реализацией и упорядочиванием вводимых пользователем данных. Если вы подойдете к этому очень творчески, вы сможете создать несколько привлекательных игровых систем механик.



Шаблоны проектирования интересно использовать, если вы не слишком беспокоитесь о том, чтобы оставаться верными оригинальным академическим описаниям. Если вы не потеряете первоначальную цель шаблона, экспериментируя с ним, вы должны со временем освоить его основные преимущества.

## Проектирование систмы повторов

Прежде чем описывать структуру системы повторов, которую будем реализовывать в этой главе, мы должны обсудить некоторые характеристики нашей игры, которые могут повлиять на способ ее реализации.

Технические характеристики, на которые следует обратить внимание, следующие:

- Детерминированность: все в нашей игре детерминировано, что означает, что у нас нет объектов с случайным поведением, и это упрощает реализацию нашей системы воспроизведения, поскольку нам не нужно беспокоиться о записи позиций или состояний объектов, которые движутся в нашей игре. Цена, как правило, дроны. Мы знаем, что они будут двигаться и вести себя одинаково во время повтора.
- Физика: мы сходим к минимуму использование физических возможностей движка Unity, поскольку движения наших объектов не определяются какими-либо физическими силами или взаимодействиями. Поэтому нам не нужно беспокоиться о неожиданном поведении при столкновении объектов.
- Цифровой: все наши виды взаимодействия цифровыми, поэтому мы не утруждаем себя сбором или обработкой детализированных аналоговых входных данных с джойстика или кнопки триггера.
- Точность: мы терпим отсутствие точности во времени воспроизведения входных данных. Поэтому мы не ожидаем, что входные данные будут воспроизводиться точно в тот же период времени, в котором они были записаны. Этот уровень допуска может меняться в зависимости от нескольких факторов, связанных с желаемым уровнем точности функции воспроизведения.

Учитывая все эти характеристики, система повторов, которую мы собираемся реализовать, будет записывать только действия игрока, но не текущее положение мотоцикла. Поскольку нет промежуточных мест, в которых мог бы находиться велосипед, зависящим от действий игрока он будет находиться на одном рельсе или другом. Кроме того, еще одна важная деталь, которую следует обратить внимание, — этот факт, что мотоцикл не движется вперед. Это дорожка, которая движется к положению игрока, сдавая иллюзию движения искорки.

Теоретически, если мы записываем вводимые игроком команды контроллером в начале и в конце гонки, то мы могли бы имитировать повтор игрока проще с игрока, воспроизводя записианные вводы в начале новой гонки. У нас есть диаграмма, иллюстрирующая механизм системы повторов:

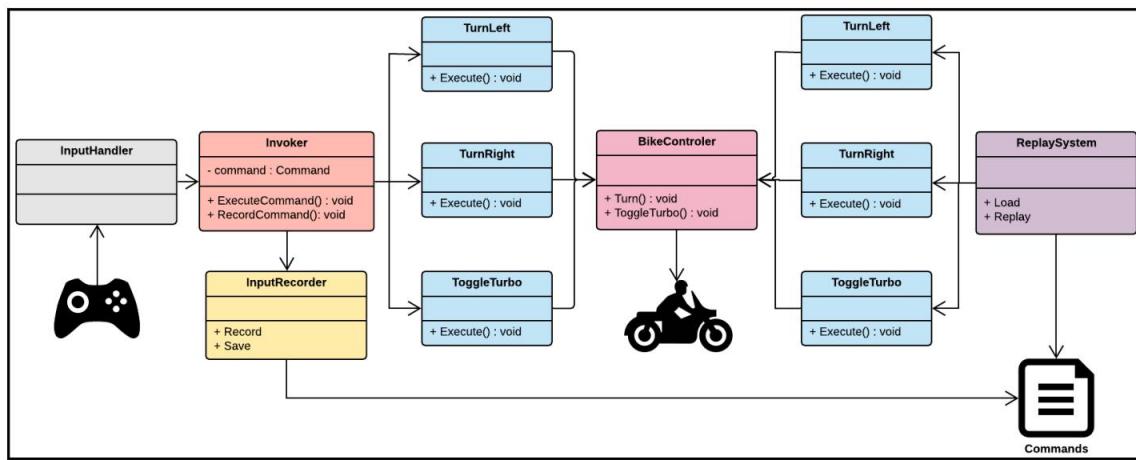


Рисунок 7.2 – Схема системы повторов

Как мы видим на диаграмме, **InputRecorder** записывает и сериализует в одни данные, чтобы **ReplaySystem** мог воспроизвести их позже. Во время воспроизведения **ReplaySystem** действует аналогично тому: он берет на себя управление велосипедом и автоматически маневрирует, воспроизводя действия игрока. Это простая форма автоматизации, которая создает иллюзию будто мы смотрим повтор видео.

В следующем разделе мы реализуем упрощенную версию системы, которую мы только что рассмотрели, и будем использовать шаблон «Команда» в качестве ее основы.

## Реализация систмы повторов



В этот раздел включен пейдж для простоты и читабельности. Если вы хотите просмотреть полную реализацию контексте реального игрового проекта, откройте папку FPP в проекте GitHub. Ссылку можно найти в разделе Технические требования.

В этом разделе мы собираемся создать простой прототип системы ввода произведения ввода, используя в качестве основы шаблон «Команда».

## Реализация систмы повторов

Реализация будет осуществляться в два этапа. В первой части мы закодируем новые компоненты шаблона «Команда», а затем интегрируем элементы, необходимые для тестирования системы ввода произведения:

1. Для начала мы реализуем базовый абстрактный класс с именем Command, который имеет единственный метод с именем Execute():

```
публичный абстрактный класс Command {
    публичная абстрактная недействительность Execute();
}
```

2. Теперь мы собираемся написать три конкретных класса команд, которые будут производными от базового класса Command, а затем реализуем метод Execute(). Каждый из них инкапсулирует действие, которое необходимо выполнить.

Первый включает турбокомпрессор на BikeController:

```
просмотр твоимен Chapter.Command {
    публичный класс ToggleTurbo: Command {
        частный BikeController _controller;
        public ToggleTurbo (контроллер BikeController) {
            _controller = контроллер;
        }
        публичное переопределение void Execute() {
```

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

### 3. Следующие две команды — это команды TurnLeft и TurnRight .

Каждый из них представляет с собой отдельное действие и определен с определенной клавишей ввода, как мы увидим, когда реализуем InputHandler:

```
просмотр твоим Chapter.Command {  
  
    публичный класс с TurnLeft: Command {  
  
        частный BikeController _controller;  
  
        public TurnLeft (контроллер BikeController) {  
  
            _controller = контроллер;  
        }  
  
        публичное переопределение void Execute() {  
  
            _controller.Turn(BikeController.Direction.Left);  
        }  
    }  
}
```

### 4. Следующая команда представляет действие поворота направо и, как следует из названия , это поворачивает велосипед вправо:

```
просмотр твоим Chapter.Command {  
  
    публичный класс с TurnRight: Command {  
  
        частный BikeController _controller;  
  
        public TurnRight (контроллер BikeController) {  
  
            _controller = контроллер;  
        }  
  
        публичное переопределение void Execute() {  
  
            _controller.Turn(BikeController.Direction.Right);  
        }  
    }  
}
```

Теперь, когда мы завершили инкапсуляцию каждого команды в отдельные классы, пришло время закодировать критически важный компонент, который заставит нашу систему воспроизведения работать — Invoker.

Инвокер — внимательный бухгалтер; он отслеживает выполненные команды в реестре. Мы представляем этот реестр в коде в виде SortedList — с обобщенной отсортированной коллекции C# со структурой ключ/значение. Этот список будет отслеживать, когда была выполнена определенная команда.

Поскольку этот урок очень длинный, мы рассмотрим его в двух частях. Следующее это первая часть:

```
использование UnityEngine;
используя System.Linq;
использование System.Collections.Generic;

последование твоим именем Chapter.Command {

    класс Invoker: MonoBehaviour
    {
        частный bool _isRecording; частный
        bool _isReplaying; частный плавающий
        _replayTime; частный плавающий
        _recordingTime; частный SortedList<float,
        Command> _recordedCommands =
        новый SortedList<float, Command>();

        public void ExecuteCommand (Командная команда) {
            команда.Выполнить();
            если (_isRecording)
                _recordedCommands.Add(_recordingTime, команда);
            Debug.Log("Время записи: " + _recordingTime);
            Debug.Log("Записанная команда: " + команда);
        }

        общественная недействительная запись ()
        {
            _recordingTime = 0,0f; _isRecording
            = правда;
        }
    }
}
```

В этой части класса Invoker мы добавляем команду в отсортированный список \_recordedCommands каждый раз, когда Invoker выполняет новую. Однако мы делаем это только тогда, когда начинаем запись, потому что хотим записать действия игрока в определенные моменты, например, в начале гонки.

6. В следующем разделе класса Invoker мы собираемся реализовать воспроизведение поведение:

```

общественный недействительный Replay()
{
    _replayTime = 0.0f; _isReplaying
    = правда

    если (_recordedCommands.Count <= 0)
        Debug.LogError("Нет команд для воспроизведения!");

    _recordedCommands.Reverse();
}

недействительный фиксированный Update()
{
    если (_isRecording)
        _recordingTime += Time.fixedDeltaTime;

    если (_isReplaying) {

        _replayTime += Time.deltaTime;

        если (_recordedCommands.Any()) {

            если (Math.Приблизительно(
                _replayTime, _recordedCommands.Keys[0])) {

                Debug.Log("Время воспроизведения : " + _replayTime);
                Debug.Log("Команда воспроизведения : " +
                    _recordedCommands.Values[0]);

                _recordedCommands.Values[0].Execute();
                _recordedCommands.RemoveAt(0);
            }
        }
    }
}

```

```

        }
    }
}
```

Как вы, возможно, уже заметили, мы используем FixedUpdate() для записи и воспроизведения команд. Это может показаться странным, поскольку мы обычно используем Update() для прослушивания ввода игрока. Однако метод фиксированного обновления () имеет преимущество, заключающееся в том, что он работает с фиксированными временными интервалами и полезен для задач, зависящих от времени, но не зависящих от частоты кадров.

Таким образом, мы знаем, что шаг времени механики по умолчанию составляет 0,02 секунды, и наши временные метки будут иметь такие же приращения, как и мы используем Time.fixedDeltaTime для записи времени выполненных команд.

Однако это также означает, что мы теряем точность на этапе записи, поскольку наша временная метка привязана к настройкам времени шага Unity. Эта потеря точности допустима в контексте этого примера. Однако это может стать проблемой, если между игровым процессом и последовательностью повторов существует значительные несогласования.

В этом случае нам, возможно, придется рассматривать решение, включающее Update(), Time.deltaTime и значение, которое позволит нам установить степень точности при сравнении времени записи и воспроизведения. Однако это выходит за рамки данной главы.

Следует принять во внимание что мы возлагаем на Invoker ответственность за ведение буферов учета, а также за воспроизведение. Можно утверждать, что мы нарушили принцип единой ответственности, возложив на Invoker слишком много обязанностей. В данном контексте это не проблема; это просто пример кода для образовательных целей. Тем не менее, было бы разумно инкапсулировать общую занятость по записи, с охвачением воспроизведению команд в отдельный класс.

## Тестирование системы повторов

Теперь, когда у нас есть новые компоненты шаблона «Команда» и наша система повторов, пришло время проверить, работает ли она:

1. Первый класс, который мы реализуем, — это InputHandler. Его основная обязанность — слушать действия игрока и вызывать соответствующие команды. Однако из-за его длины мы рассмотрим его в двух частях:

```

использование UnityEngine;
{
    прослушивание имен Chapter.Command {
        публичный класс InputHandler: MonoBehaviour {

```

```

частный Invoker _invoker; частный
bool _isReplaying; частный bool
_isRecording; частный BikeController
_bikeController; частная команда _buttonA, _buttonD,
_buttonW;

недействительный стар()
{
    _invoker = gameObject.AddComponent<Invoker>(); _bikeController =
FindObjectOfType<BikeController>();

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

недействительное Обновление
0(
if (!_isReplaying && _isRecording) {

если (Input.GetKeyUp(KeyCode.A))
    _invoker.ExecuteCommand(_buttonA); если
(Input.GetKeyUp(KeyCode.D))
    _invoker.ExecuteCommand(_buttonD); если
(Input.GetKeyUp(KeyCode.W))
    _invoker.ExecuteCommand(_buttonW);
}
)
}

```

В этом разделе классы инициализируются нашими командами и определяются с конкретными значениями. Обратите внимание, что мы передаем экземпляр BikeController в конструктор команды. InputHandler знает только о существовании BikeController, но ему не обязательно знать, как он работает. Отдельные классы команд несут ответственность за вызов соответствующих общедоступных методов контроллера в зависимости от желаемого действия. В цикле Update() мы прослушиваем ввод определенных клавиш и вызываем Invoker для выполнения команды, связанный с конкретным вводом.

Этот сегмент кода позволяет записывать вводимые игроком данные.

Мы не вызываем BikeController напрямую и не выполняем команды.

Вместо этого мы позволяем Invoker действовать как посредник и выполнять всю работу. Такой подход позволяет сократить запись ввода игрока в фоновом режиме для последующего использования.

2. В заключительной части класса InputHandler мы добавляем не сколько кнопок отладки графического интерфейса, которые помогут нам протестировать систему воспроизведения. Этот segment кода предназначен только для целей отладки и тестирования:

```
недействительный OnGUI())
{
    if (GUILayout.Button("Начать запись")) {

        _bikeController.ResetPosition(); _isReplaying = ложь;
        _isRecording = правда;
        _invoker.Запись();

    }

    if (GUILayout.Button("Остановить запись")) {

        _bikeController.ResetPosition(); _isRecording = ложь;

    }

    если (!_isRecording) {

        if (GUILayout.Button("Начать повтор")) {

            _bikeController.ResetPosition(); _isRecording =
            ложь; _isReplaying = правда;
            _invoker.Replay();

        }
    }
}
}
```

3. Для нашего позднего класса мы реализуем скелетную версию класса BikeController для целей тестирования. Он действует как получатель в контексте шаблона «Команда»:

```
использование UnityEngine;

публичный класс BikeController: MonoBehaviour {

    публичное перечисление Направление
    {
        Слева = -1,
        Справа = 1
    }
}
```

```

частный bool _isTurboOn; частный
float _distance = 1.0f;

общественная недействительность ToggleTurbo()
{
    _isTurboOn = !_isTurboOn;
    Debug.Log("Турбо-активен: " + _isTurboOn.ToString());
}

public void Turn (направление направления ) {

    если (направление == Направление.Влево)
        Transform.Translate(Vector3.left * _distance);
    if (направление == Direction.Right)
        Transform.Translate(Vector3.right * _distance);
}

общественный недействительный ResetPosition ()
{
    Transform.position = новый Vector3(0.0f, 0.0f, 0.0f);
}
}

```

Общая цель и структура класса говорят сама за себя.

`ToggleTurbo()` и `Turn()` — общедоступные методы, вызываемые классами команд. Однако `ResetPosition()` предназначен только для целей отладки и тестирования, и его можно игнорировать.

Чтобы протестировать этот код в вашем экземпляре Unity, вам необходимо выполнить следующие шаги:

1. Запустите новую пустую сцену Unity, включаяющуюся в один источник света и камеру.
2. Добавьте в новую сцену 3D GameObject, например куб, и сделайте его видимым с новной камеры сцены.
3. Прикрепите классы `InputHandler` и `BikeController` к новому GameObject.

4. Если во время выполнения вы скопировали все классы, которые мы только что расмотрели, в свой проект, вы должны увидеть на экране следующее:

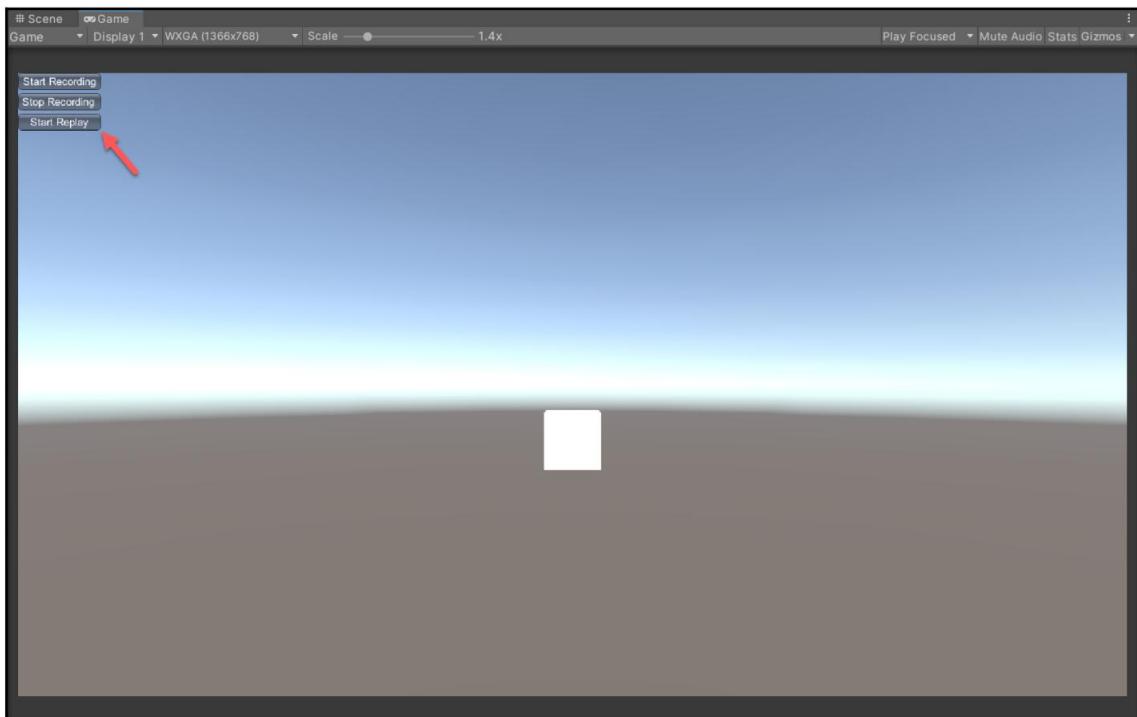


Рисунок 7.3 – Скриншот кода в действии внутри Unity

Когда вы начнете запись, вы сможете перемещать куб по горизонтальной оси. Каждый введенный ввод можно воспроизвести в той же последовательности времени, в котором они были записаны.

## Обзор реализации

Мы завершили процесс создания системы быстрого воспроизведения ввода с помощью команды `yield` в качестве нашей основы. Конечно, пример кода в этой главе не готов к использованию очень ограничен. Тем не менее, целью этой главы было научиться использовать.

Используйте шаблоны команд с Unity, а не проектируйте используя систему повторов.

В проекте прототипа игры в папке FPP проекта GitHub мы реализовали более продвинутый пример систмы повторов, включающей сериализацией функций и перемотки назад. Рекомендуем ознакомиться и, конечно же, доработать по своему усмотрению.

В следующем разделе мы рассмотрим некоторые альтернативные решения и подходы, которые мы могли бы использовать для создания нашей системы повторов.

## Рассмотрение альтернативных решений

Даже если шаблон «Команда» идеально подходит для нашего варианта использования, мы могли бы рассмотреть некоторые альтернативные шаблоны и решения:

- **Memento:** шаблон Memento предоставляет возможность откатить объект к предыдущему состоянию. Это не был наш первый выбор для нашей системы воспроизведения, поскольку мы концентрируемся на записи и хранении данных и постановке их в очередь для воспроизведения в более позднее время, что очень совместимо с замыслом проектирования шаблона «Команда». Однако если мы реализуем систему с функцией отката к предыдущему состоянию то паттерн Memento, вероятно, будет нашим первым выбором.
- **Очередь/стек.** Очереди и стеки — это не шаблоны, а структуры данных, но мы могли бы просто закодировать все наши храненные данные и сюда занести их в очередь непосредственно в нашем классе InputHandler. Это было бы проще и менее сложно, чем использование шаблона «Команда». Выбор между реализацией системы с традиционными шаблонами проектирования или без них очень зависит от контекста. Если система достаточно проста, то дополнительная многолетность и сложность, которые может принести шаблон проектирования, могут превысить потенциальные выгоды от его использования.

## Краткое содержание

В этой главе мы реализовали простую функциональную систему воспроизведения, используя шаблон «Команда». Наша цель в этой главе заключалась не в том, чтобы показать, как создать надежную систему повторов, а в том, чтобы продемонстрировать, как использовать шаблон «Команда» для создания в Unity чего-то, что может быть полезно для игрового проекта.

Я надеюсь, что вы изучите альтернативные способы реализации шаблона «Команда», которые могут оказаться лучше, чем показано в этой книге, поскольку, как и в большинстве случаев в программировании, не существует единого способа выполнения задач. Тем не менее, по крайней мере, в этой главе предлагается первый подход к использованию шаблона «Команда» в Unity.

В следующий части книги мы начнем оптимизировать наш код с помощью пул объектов. Важным аспектом ходящей гонки времени является стабильная производительность частота кадров. В любой момент все должно работать гладко, иначе игра может стать медленной и вязкой.

# 8

## Оптимизация с помощью пул объектов Шаблон

В большинстве видеоигр на экране происходят множества событий. Пули летают, враги появляются на карте, частицы появляются вокруг игрока, эти различные объекты загружаются и отображаются на экране в мгновение ока. Поэтому, чтобы избежать нагрузки на центральный процессор (ЦП) и при этом поддерживать постоянную частоту кадров, рекомендуется зарезервировать немногую память для частот создаваемых объектов. Таким образом, вместе с тем, чтобы высвобождать недавно уничтоженных врагов из памяти, мы добавляем их в пул объектов, чтобы переработать их для дальнейшего использования. С помощью этого метода мы избегаем первоначальных затрат на анализ изображения рисунков нового экземпляра объекта. Кроме того, поскольку мы не уничтожаем повторно используемые объекты, сборщик мусора (GC) не будет тратить циклы на очистку набора регулярно повторно анализируемых объектов.

Именно этим мы и собираемся заняться в этой главе, но нам повезло, поскольку начиная с версии Unity 2021 пул объектов изначально интегрированы в интерфейс прикладного программирования (API).

Следовательно, нам не нужно будет реализовывать шаблон вручную как мы это делали в предыдущих главах; вместе с тем мы сопрототочимся на том, чтобы научиться его использовать и позволить движку выполнять всю работу.

В этой главе будут рассмотрены следующие темы:

- Понимание шаблона пула объектов
- Реализация шаблона пула объектов
- Рассмотрение альтернативных решений



GC функционирует как автоматизированный менеджер памяти и является важным компонентом большинства современных объектно-ориентированных языков, таких как C#. Чтобы продолжить изучение этой главы, не обязательно понимать, как это работает, но если вам интересно, вы можете получить дополнительную информацию по этому вопросу здесь: <https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/>.

## Технические требования

Эта глава является практической, поэтому вам потребуется базовое понимание Unity и C#.

Файлы кода этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter08>.

Посмотрите следующее видео, чтобы увидеть код в действии:

<https://bit.ly/3yTLcI7>



Важно отметить, что пример кода в следующем разделе не будет работать в версии Unity ниже 2021.1, поскольку мы используем недавно добавленные функции API.

## Понимание шаблона пул объектов

Основная концепция этого шаблона проста: пул в форме контейнера хранится в памяти коллекцию инициализированных объектов. Клиенты могут запросить пул объектов для экземпляра объекта определенного типа; если он досутупен, он будет удален из пула и передан клиенту. Если в данный момент в пуле недостаточно экземпляров, новые будут создаваться динамически.

Объекты, покидающие пул, пытаются вернуться в него, как только они больше не будут использоваться клиентом. Если в пуле объектов больше нет места, он уничтожит экземпляры объектов, которые пытаются вернуться. Поэтому басейн постепенно пополняется, может ли временно сливаться, но никогда не переполняется. Следовательно, использование памяти влечется последовательным.

На следующей диаграмме показано взаимодействие между клиентом и пулем объектов:

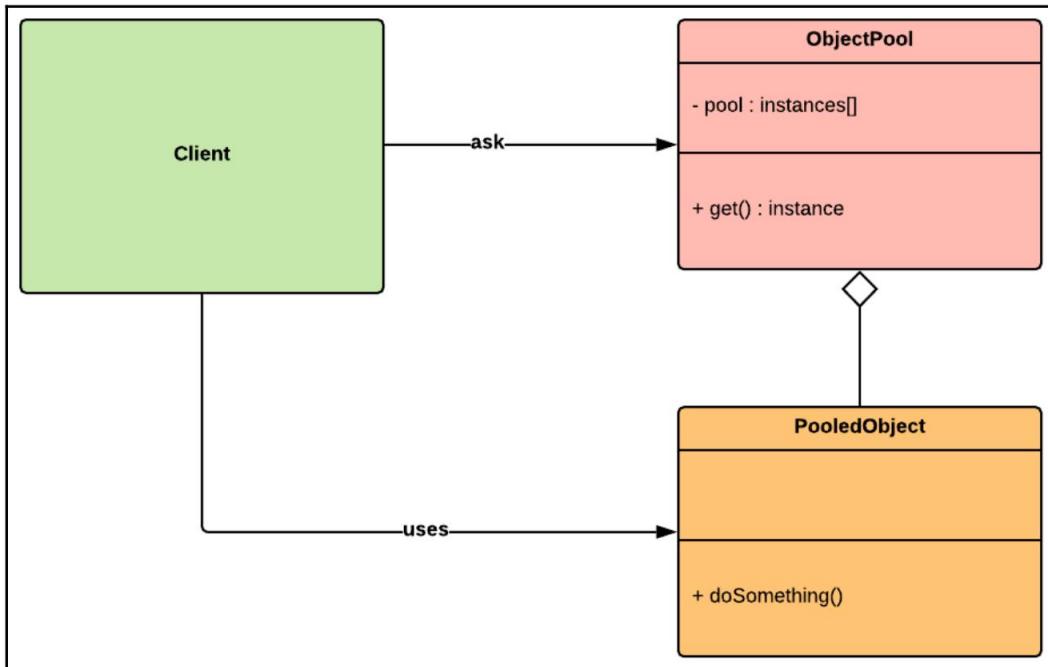


Рисунок 8.1 – Диаграмма унифицированного языка моделирования (UML) шаблона «Пул объектов»

На диаграмме мы видим, что пул объектов обслуживает клиента, предлагая ему доступ к пулу экземпляров объектов определенного типа — так, например, клиент может быть генератором, запрашивающим у пула объектов экземпляры определенного вида.

## Преимущества и недостатки шаблона пула объектов

Вот некоторые преимущества шаблона пула объектов:

- Предсказуемое использование памяти. С помощью пула объектов мы можем предсказуемым образом выделить некоторый объем памяти для хранения определенного количества экземпляров объектов определенного типа.
- Повышение производительности. Имея объекты, уже инициализированные в памяти, вы избегаете затрат на инициализацию новых.

Вот некоторые потенциальные недостатки шаблона пула объектов:

- Наложение на уже управляемую память. Некоторые люди критикуют шаблон пула объектов как ненужный в большинстве случаев, поскольку современные управляемые языки программирования, такие как C#, уже оптимально управляют распределением памяти. Однако это утверждение может быть истинным в одних контекстах и ложным в других.
- Непредсказуемые состояния объекта. Потенциальная ошибка шаблона «Пул объектов» заключается в том, что при неправильной обработке объекты могут быть возвращены в пул в их текущем состоянии вместо исходного. Эта ситуация может стать проблемой, если объединенный объект можно повредить или разрушить. Например, если у вас есть времеческий объект, который только что был убит и рекомендован, если вы вернете его в пул, не восстанавливая его здоровье, когда пул объектов вытащит его обратно для клиента, он может виться обратно на страницу уже испорченное состояние.

## Когда использовать шаблон пула объектов

Чтобы лучше понять, когда использовать шаблон «Пул объектов», давайте рассмотрим, когда его не следует использовать. Например, если у вас есть объекты, которые необходи́мо один раз создать на вашей карте, например финальный босс, помещение их в пул объектов — это пустая траты памяти, которую могли бы использовать для чего-то более полезного.

Также следует помнить, что пул объектов не является кешем. У него есть одна цель — повторное использование объектов. Основное отличие состоит в том, что пул объектов имеет механизм, в котором объекты автоматически возвращаются в пул после использования, а пул объектов, если он хорошо реализован, обрабатывает создание и удаление объектов в зависимости от достаточного размера пула.

Но предположим, что у нас есть такие особенности, как пули, частицы и времеческие персонажи, которые часто сдвигаются и уничтожаются во время игрового процесса. В этом случае пул объектов может облегчить часть нагрузки, которую мы оказываем на ЦП за счет сокращения повторяющихся вызовов жизненного цикла, таких как создание и уничтожение, таким образом ЦП может зарезервировать вычислительную мощность для более важных задач.

В следующем разделе мы возьмем только что изученные концепции и переведем их в код.

## Реализация шаблона пула объектов

Прежде чем начать этот раздел, было бы неплохо прочитать официальную документацию API для класса `IObjectPool<T0>` в пространстве имен `UnityEngine.Pool` по следующей ссылке:

[https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Pool.ObjectPool\\_1.html](https://docs.unity3d.com/2021.1/Documentation/ScriptReference/Pool.ObjectPool_1.html)

Мы постараемся не упомянуть спецификации API при реализации с ледущего примера кода. Вместо этого мы сосредоточимся на критических элементах, не посредственно связанных с основными концепциями пула объектов. Кроме того, объединение с обстоятельных объектов в пулы — это относительно новая функция Unity API, поэтому в нее могут вноситься изменения и обновления. Таким образом, было бы разумно следить за документацией в краткосрочной перспективе.

## Шаги по реализации шаблона пула объектов

Пример кода в этом разделе должен быть относительно простым, и мы можем выполнить его в два этапа, а именно:

- Начнем с реализации нашего дрона, так как это объект, который мы объединим.

Поскольку этот такой длинный класс, мы разделим его на два этапа. Посмотреть первый этап можно здесь:

```
использование UnityEngine;
использование UnityEngine.Pool;
использование System.Collections;

пространство имен Chapter.ObjectPool {

    публичный класс Drone: MonoBehaviour {

        общесостоятельный IObjectPool<Drone> Pool {get; набор; }

        общедоступное плавающее значение _currentHealth;

        [SerializeField] Private
        float maxHealth = 100.0f;

        [Сериалайзифлд]
        частное плавающее время ToSelfDestruct = 3.0f;

        недействительный Старт
        {
    }
}
```

```

        _currentHealth = максимальное здоровье;
    }

недействительный OnEnable()
{
    Атакайг рож();
    StartCoroutine(SelfDestruct());
}

недействительный OnDisable()
{
    Сброс Дрон();
}

```

Важно отметить, что мы вызываем метод ResetDrone() в функции с события OnDisable() в этом сегменте класса. Мы делаем это, потому что хотим вернуть дрон в исходное состояние, прежде чем вернуть его в пул.

И как мы увидим, когда реализуем шаблон пула объектов, когда GameObject возвращается в пул, он отключается, включая все его одочерние компоненты. Следовательно, если нам нужно выполнить какой-либо код повторной инициализации, мы можем сделать это при вызове OnDisable().

В контексте этой главы мы стараемся упростить задачу; мы лишь восстановляем здоровье дрона. Но в более продвинутой реализации нам, возможно, придется сбросить визуальные маркеры, например удалить поврежденные надписи.

2. В последнем сегменте нашего класса Drone мы реализуем новые модели поведения: следуют:

```

IEnumerator SelfDestruct() {

    дох одноть возврата новых WaitForSeconds (timeToSelfDestruct);
    TakeDamage (Макс Здоровье);
}

частная пустота ReturnToPool () {

    Pool.Release(это);
}

частная пустота ResetDrone () {

    _currentHealth = максимальное здоровье;
}

объектная недействительность AttackPlayer()

```

```
{  
    Debug.Log("Атакуйте иг рока!");  
}  
  
public void TakeDamage(плывающая сумма) {  
  
    _currentHealth -= сумма;  
  
    если (_currentHealth <= 0f)  
        ВозвратВПул();  
    }  
}  
}
```

У нашего дрона есть два ключевых поведения :

- Самоуничтожение . У нашего дрона краткий срок службы ; когда он включен , вызывается с опорой рамма SelfDestruct() . Через несколько секунд дрон самоуничтожится , исчерпав свой счетчик здоровья и вернувшись в пул , вызвав метод ReturnToPool() .
- Атака логика внутри метода не реализована изображений краткости . Но представьте , что после падения дрона он идет и атакует игру .

3. Далее идет наш класс ObjectPool , который отвечает за управление пулем экземпляров дронов . Поскольку это длинный урок , мы рассмотрим его в двух сегментах , причем первый сегмент дос тупен для просмотра здесь :

```
использование UnityEngine;  
использование UnityEngine.Pool;  
  
пространство имен Chapter.ObjectPool {  
  
    публичный класс DroneObjectPool : MonoBehaviour {  
  
        общес твенный ИНГ maxPoolSize = 10;  
        общес твенный int stackDefaultCapacity = 10;  
  
        общес твенный пул IObjectPool<Drone> {  
  
            получать  
            {  
                if (_pool == null) _pool = новый  
  
                    ObjectPool<Drone>(  
                        Созданный пул для item,  
                        OnTakeFromPool,  
                        OnReturnedToPool,  
                    );  
            }  
        }  
    }  
}
```

```

        OnDestroyPoolObject, true,
        stackDefaultCapacity, maxPoolSize);

    вернуть _пул;
}
}

час тный IObjectPool<Drone> _pool;

```

В этой первой части сценария мы устанавливаем критическую переменную с именем maxPoolSize; как следует из названия, он устанавливает максимальное количество экземпляров дронов, которые мы будем хранить в пуле. Переменная stackDefaultCapacity устанавливает емкость стека по умолчанию это свойство занесено со структурой данных стека, которую мы используем для хранения экземпляра нашего дрона. На данный момент мы можемignoreировать это, поскольку для нашей реализации это не критично.

В следующем фрагменте кода мы инициализируем пул объектов, который является наиболее важной частью нашего класса:

```

public IObjectPool<Drone> Pool { get { if (_pool ==
    null)

    _pool =
        новый ObjectPool<Дрон>(
            Созданный пулителем,
            OnTakeFromPool,
            OnReturnedToPool,
            OnDestroyPoolObject,
            правда,
            stackDefaultCapacity, maxPoolSize);
        вернуть _пул;

    }
}

```

Важно отметить, что мы передаем методы обратного вызова в конструктор класса ObjectPool<T>, и именно в этих обратных вызовах мы реализуем логику, которая будет управлять нашим пулом объектов.

4. В последнем сегменте класса DroneObjectPool мы реализуем обратные вызовы, объявленные в контракте ObjectPool<T>, следующим образом:

```

    частный дрон CreatedPooledItem() {

        var go =
            GameObject.CreatePrimitive(PrimitiveType.Cube);

        Дрон дрон = go.AddComponent<Drone>();

        go.name = "Дрон";
        дрон.Басейн = Гул;

        возвратный дрон;
    }

    Private void OnReturnedToPool (Дрон дрон) {

        дрон.gameObject.SetActive(ложь);
    }

    Private void OnTakeFromPool (Дрон дрон) {

        дрон.gameObject.SetActive(истина);
    }

    Private void OnDestroyPoolObject (Дрон дрон) {

        Уничтожить(drone.gameObject);
    }

    общественная недействительность Spawn()
    {
        сумма var = Random.Range(1, 10);

        для (int я = 0; я < сумма; ++i) {

            var дрон = Pool.Get();

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

```

Вот краткое описание сущности каждого обратного вызова, который класс с ObjectPool будет вызывать в определенное время:

- `CreatedPooledItem()`: в этом обратном вызове мы инициализируем экземпляры наших дронов. В контексте этой главы мы создаем GameObject с нуля изображений прости, но в более практическом контексте мы, вероятно, просто загрузим префаб.
- `OnReturnedToPool()`: название метода подразумевает его использование. Обратите внимание, что мы не уничтожаем GameObject; мы просто деактивируем его, чтобы удалить его из сцены.
- `OnTakeFromPool()`: вызывается, когда клиент запрашивает экземпляр дрона. На самом деле экземпляр не возвращается — GameObjectключен.
- `OnDestroyPoolObject()`: это важный метод для понимания. Он вызывается, когда в бассейне больше нет места. В этом случае возвращенный экземпляр уничтожается, чтобы освободить память.

Наш класс с DroneObjectPool взял на себя дополнительные обязанности и действует как генератор, как мы видим в методе `Spawn()`. По запросу он получит экземпляр дрона из пула и сдаст его в случайном месте сцены в пределах определенного диапазона.

## Тестирование реализации пула объектов

Чтобы протестировать нашу реализацию пула объектов в вашем собственном экземпляре Unity, вам необходимо выполнить следующие шаги:

1. Создайте новую пустую сцену Unity.
2. Скопируйте все скрипты, которые мы только что рассмотрели, и скопируйте их в свой проект.
3. Добавьте пустую GameObject.
4. Привяжите следующий скриптовый скрипт к пустому GameObject:

```
ис пользование UnityEngine;

прос транс тво имен Chapter.ObjectPool {

    публичный класс ClientObjectPool: MonoBehaviour {

        частный DroneObjectPool _pool;

        недействительный Старт
        {
            _pool = gameObject.AddComponent<DroneObjectPool>();
```

```
    }

    недействительный OnGUI()
{
    if (GUILayout.Button("Создать дронов"))
        _pool.Spawn();
}
}
```

После запуска сцены вы должны увидеть кнопку графического интерфейса пользователя (GUI) с именем Создайте дронов в верхнем левом углу, как мы видим на следующем скриншоте:

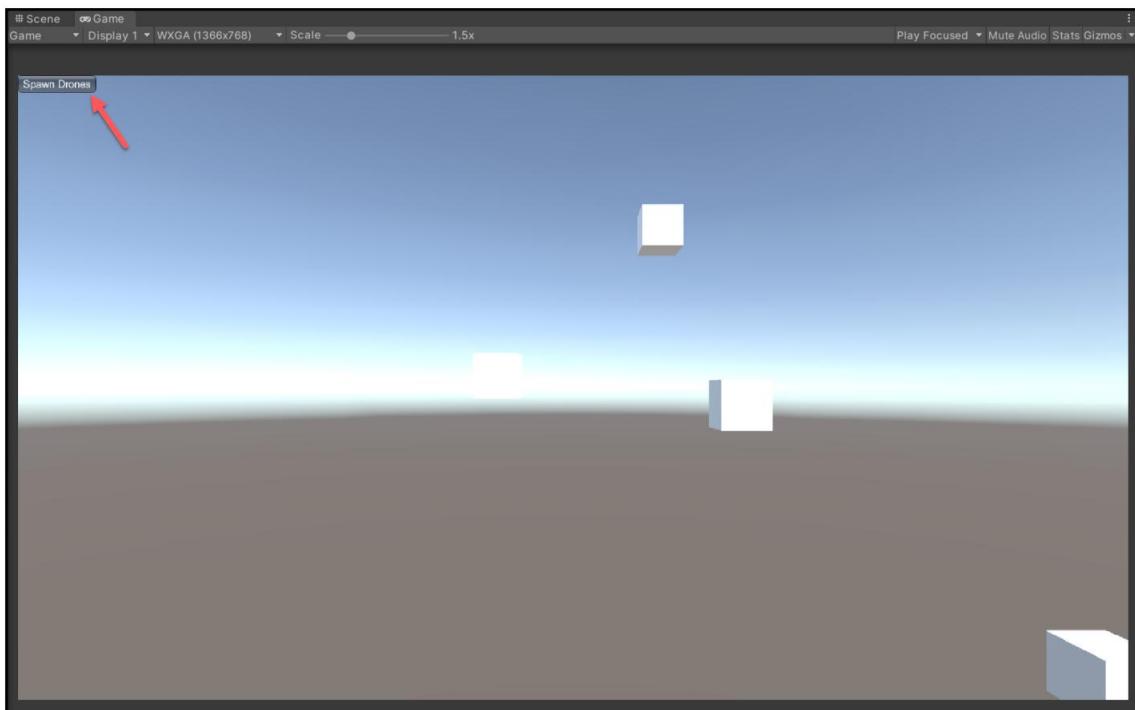


Рисунок 8.2 – Скриншот примера кода в действии

Нажав кнопку «Создать дронов», вы теперь можете создавать дронов в случайных местах сцены. Если вы отите увидеть механизм пула объектов в действии, следите за иерархией сцен — вы сможете увидеть, как объекты дронов включаются и отключаются по мере их выхода из пула.

## Обзор реализации пула объектов

Используя шаблон «Пул объектов», мы автоматизировали процесс создания, удаления и объединения экземпляров дронов. Теперь мы можем зарезервировать объем памяти для создания волн дронов, не нагружая при этом процессор. Мы добавили оптимизацию масштабируемости в наш код, реализовав этот шаблон, не теряя читабельности и не усложняя его.

В следующем разделе мы рассмотрим некоторые альтернативные решения, которые стоит рассмотреть; Всегда полезно рассмотреть другие варианты, прежде чем выбирать конкретный шаблон.

### Рассмотрение альтернативных решений

Близким родственником шаблона пула объектов является шаблон прототипа; оба они считаются творческими моделями. Используя шаблон прототипа, вы можете избежать затрат, связанных с созданием новых объектов, используя механизм клонирования. Поэтому вместо инициализации новых объектов вы клонируете их из эталонного объекта, называемого прототипом. Но в контексте варианта использования, представляемого в этой главе, объединение объектов в пул дает лучшие преимущества оптимизации.



Шаблоны творческого отрасли связаны с механизмом создания объектов. Factory, Build, Singleton, Object Pool и Prototype — все это шаблоны творческого проектирования.

### Краткое содержание

Мы только что добавили в наш набор инструментов шаблон Object Pool — это один из самых ценных шаблонов для разработчиков Unity. Как мы видели в наших примерах кода, мы можем легко перерабатывать экземпляры часто используемых объектов. При работе с большим набором объектов, которые должны создаватьсь быстро и не однократно, этот шаблон может помочь нам избежать сбоев и задержек. Эти преимущества могут только помочь сделать нашу игру лучше, поскольку играющим нравится игра, которая работает гладко.

В следующей главе мы с обирайемся отделить компоненты друг от друга с помощью шаблона Observer.

## 9

# Разделение компонентов с помощью шаблона наблюдателя

Общая задача при разработке Unity — найти элегантные способы отделения компонентов друг от друга. Это серьезное претворение, которое не обошло однажды при написании кода в движке, поскольку он предлагает нам множество способов напрямую взаимодействовать с компонентами через API и инспектор. Но за эту гибкость приходится платить: она может сделать ваш код рукописным, поскольку в некоторых случаях одна недостаточная ссылька может сломать вашу игру.

Итак, в этой главе мы будем использовать шаблон Observer для настройки отношений с основными компонентами. Эти отношения будут отображаться путем присвоения объектам роли Субъекта или Наблюдателя. Этот подход не полностью отстраняет связь между нашими компонентами, но облегчает ее логическую организацию. Он также создает систему обработки событий с структурой «один к многим», а это именно то, что нам нужно реализовать в цепи нации использования, представляемом в этой главе.



Если вы ищете способ отделить объекты друг от друга с помощью событий, находящихся в отношениях «многие ко многим», ознакомьтесь с главой 6 «Управление игровыми событиями с помощью шаблона наблюдателя».

В этой главе будут рассмотрены следующие темы:

- Понимание шаблона наблюдателя
- Разделение основных компонентов с помощью шаблона Observer
- Реализация шаблона наблюдателя
- Рассмотрение альтернативных решений

## Технические требования

Файлы кода для этой главы можно найти на GitHub: <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/главная/Активы/Главы/Глава09>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/3xDIBDA>.

## Понимание шаблона наблюдателя

Основная цель шаблона Observer — установить связь «один-ко-многим» между объектами, в которых один выступает в роли субъекта, а другой — в роли наблюдателей.

Затем субъект берет на себя ответственность уведомлять наблюдателей, когда что-то внутри него меняется и может их беспокоить.

Это чем-то похоже на отношения издателя и подписчика, в которых объекты подписываются и прослушивают определенные уведомления о событиях. Основное отличие состоит в том, что субъект и наблюдатели знают друг друга в шаблоне «Наблюдатель», поэтому они все еще связаны друг с другом.

Давайте рассмотрим UML-диаграмму типичной реализации шаблона Observer, чтобы увидеть, как это может работать при реализации в коде:

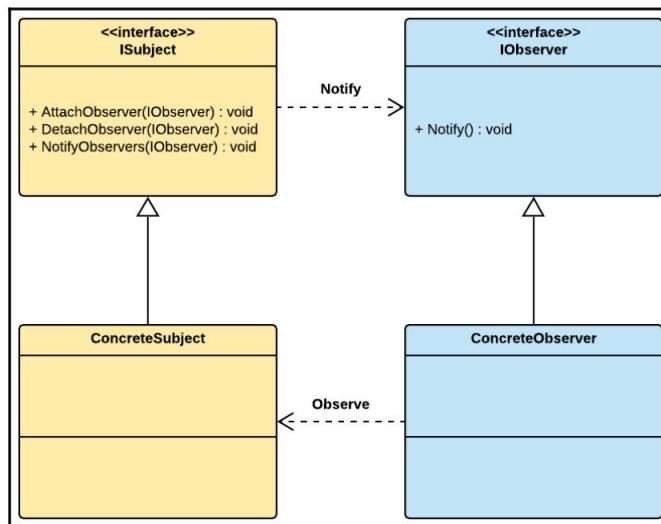


Рисунок 9.1 – UML-диаграмма шаблона Observer

Как видите, субъект и наблюдатель имеют соответствующие интерфейсы, которые они реализуют, но наиболее важным для анализа является `ISubject`, который включает в себя следующие методы:

- `AttachObserver()`: этот метод позволяет вам добавить объект-наблюдатель в список наблюдателей для уведомления.
- `DetachObserver()`: этот метод удаляет наблюдателя из списка наблюдателей.
  
- `NotifyObservers()`: этот метод уведомляет все объекты, которые были добавлены в список наблюдателей субъекта.

Объект, берущий на себя роль наблюдателя, должен реализовать общий метод `Notify()`, который будет использоваться субъектом для уведомления его об изменении состояния.

## Преимущества и недостатки шаблона Observer

Вот некоторые преимущества шаблона `Observer`:

- Динамизм: Наблюдатель позволяет Субъекту добавлять с только объектов, сколько ему необходимо в качестве Наблюдателей. Но он также может удалять их динамически во время выполнения.
- Один-ко-многим: Основное преимущество шаблона `Observer` заключается в том, что он легко решает проблему реализации системы обработки событий, в которой между объектами существует связь «один-ко-многим».

Ниже приведены некоторые потенциальные недостатки шаблона `Observer`:

- Бессердак: шаблон `Observer` не гарантирует порядок, в котором наблюдатели получают уведомления. Таким образом, если два или более объекта `Observer` имеют общие зависимости и должны работать вместе в определенной последовательности, шаблон `Observer` в своей обстановке не предназначен для обработки такого контекста выполнения.
- Утечка памяти: наблюдатель может вызывать утечку памяти, поскольку субъект содержит ссылки на своих наблюдателей. Если он реализован неправильно и объекты `Observer` неправильно отсоединяются и удаляются, когда они больше не нужны, это может вызвать проблемы с ожиданием мусора, и некоторые ресурсы не будут освобождены.



Чтобы понять потенциальный недостаток утечки памяти, указанный здесь, я рекомендую прочитать следующие статьи Википедии по этому вопросу: [https://en.wikipedia.org/wiki/Lapsed\\_listener\\_problem](https://en.wikipedia.org/wiki/Lapsed_listener_problem).

Но учтите, что, как и все, что связано с оптимизацией, оно зависит от контекста, поэтому вам следует проанализировать свой код перед оптимизацией на предмет потенциальных проблем производительности.

## Когда использовать шаблон Observer

Преимущество шаблона Observer в том, что он решает конкретные проблемы, связанные с отношениями один-ко-многим между объектами. Итак, если у вас есть основной компонент, который часто меняется состояния и имеет множество зависимостей, которые должны реагировать на эти изменения, то шаблон Observer позволяет вам определить связь между этими существами механизмо, который позволяет им уведомляться.

Поэтому, если вы не уверены, когда использовать шаблон «Наблюдатель», вам следует проанализировать взаимосвязь между вашими объектами, чтобы определить, подходит ли этот шаблон для проблемы, которую вы пытаетесь решить.

## Развяжка новых компонентов с помощью шаблона наблюдателя

Основным элементом нашей игры является игровой мотоцикл. Это существо в нашей сцене, которое изменяет состояния и обновляет свое состояние, поскольку находясь под контролем игрока во время путешествий по миру и взаимодействия с другими существами. У него есть несколько зависимостей, которыми нужно управлять, например, новая камера, следящая за ним, и HUD, отображающий текущую скорость.

Игровой мотоцикл — главный объект нашей игры, и многие системы должны наблюдать за ним, чтобы иметь возможность обновляться при изменении его состояния. Например, каждый раз, когда мотоцикл сталкивается с препятствием, HUD должен обновлять текущее значение здоровья щита, а камера отображает полноэкранный шейдер, который затемняет края экрана, чтобы продемонстрировать уменьшающуюся видимость.

Этот тип поведения легко реализовать в Unity. Мы могли бы заставить BikeController сообщать HUDController и CameraController, что делать, когда он получает урон. Но чтобы этот подход работал, BikeController должен знать об общедоступных методах, которые можно вызывать на каждом контроллере.

Как вы можете себе представить, это плохой способ шаблируется, поскольку по мере роста сложности BikeController нам придется управлять большим количеством вызовов его зависимостей. Но с помощью шаблона Observer мы собираемся разорвать эту связь между контроллерами. Сначала мы дадим каждому компоненту роль; BikeController станет субъектом и будет отвечать за управление списком зависимостей и уведомление о них при необходимости.

Контроллеры HUD и камеры будут действовать как наблюдатели BikeController. Их основная обязанность будет заключаться в том, чтобы прослушивать уведомления, поступающие от BikeController, и действовать в соответствии с определенным образом. BikeController не говорит им, что делать; он просто сообщает им, что что-то изменилось, и позволяет им реагировать на это по своему усмотрению.

Следующая диаграмма иллюстрирует концепцию которую мы только что рассмотрели:

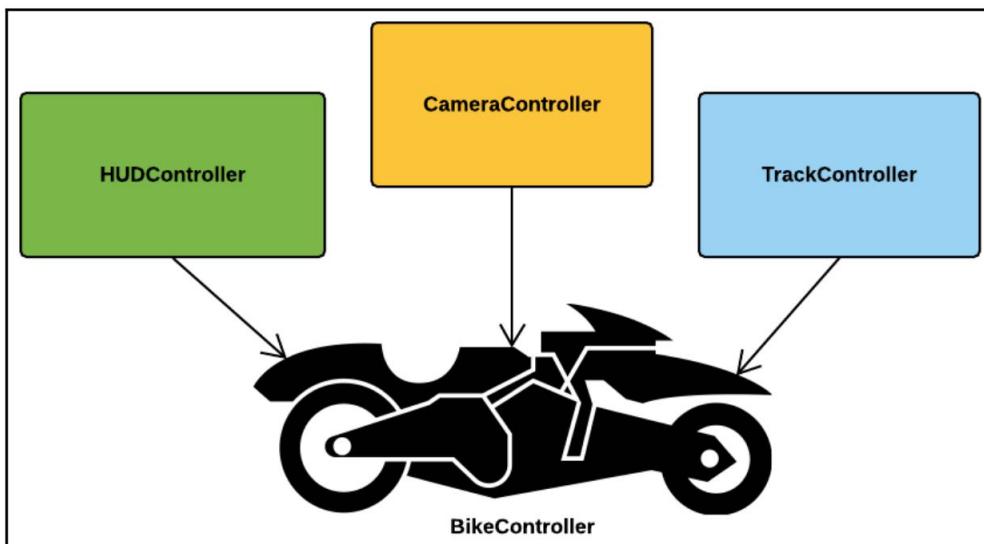


Рисунок 9.2 – Изображение контроллеров, наблюдающих за объектом

Как мы видим, у нас может быть только контроллеров, наблюдающих за велосипедом (субъектом), сколько нам нужно. В следующем разделе мы возьмем эти концепции и переведем их в код.

# Реализация шаблона наблюдателя

Теперь давайте реализуем шаблон Observer простым способом, который можно будет повторно использовать в различных контекстах:

- Мы собираемся начать этот пример кода с реализации двух элементов шаблона. Начнем с класса `Subject`:

```
ис пользование UnityEngine;
ис пользование System.Collections;

последовательность Chapter.Observer {

    публичный абстрактный класс с Тема MonoBehaviour {

        частный только для чтения
        ArrayList _observers = новый ArrayList();

        public void Attach (наблюдатель-наблюдатель) {

            _observers.Add(наблюдатель);
        }

        public void Detach (наблюдатель-наблюдатель) {

            _observers.Remove(наблюдатель);
        }

        общественный недоступный NotifyObservers ()
        {
            foreach (Наблюдатель-наблюдатель в _observers) {

                наблюдатель.Notify(это);
            }
        }
    }
}
```

Абстрактный класс субъекта имеет три метода. Первые два, `Attach()` и `Detach()`, отвечают за добавление или удаление объекта-наблюдателя из списка наблюдателей соответственно. Третий метод, `NotifyObservers()`, отвечает за циклический просмотр списка объектов-наблюдателей и вызов их открытого метода `Notify()`. Это будет иметь смысл, когда мы реализуем конкретные классы наблюдателей на следующих шагах.

2. Далее идет абстрактный класс с Observer :

```
использование UnityEngine;

прос транс тво имен Chapter.Observer {

    публичный абстрактный класс с Observer: MonoBehaviour {

        публичная абстрактная недействительность Notify(субъект темы);
    }
}
```

Классы, желающие стать наблюдателями, должны наследовать этот класс с Observer и реализовать абстрактный метод Notify(), который получает субъект в качестве параметра.

3. Теперь, когда у нас есть основные ингредиенты, давайте напишем скелетный класс с BikeController, который будет нашим субъектом. Однако, поскольку он такой длинный, мы разделим его на три сегмента. Первый сегмент — это простой код инициализации:

```
использование UnityEngine;

прос транс тво имен Chapter.Observer {

    публичный класс с BikeController: Тема {

        общес твенный bool IsTurboOn {

            получать; частный набор;
        }

        public float CurrentHealth {

            получить { вернуть здоровье; }
        }

        частный bool _isEngineOn; частный
        HUDController _hudController; частный CameraController
        _cameraController;

        [SerializeField] частное
        плавающее состояние = 100.0f;

        пустого пробуждение()
        {
            _hudController =
                gameObject.AddComponent<HUDController>();
        }
    }
}
```

```

    _cameraController =
        (Контроллер камеры)
        FindObjectOfType(typeof(CameraController));
    }

    частный недействительный Start()
    {
        Запустить двигатель();
    }
}

```

Следующий момент важен, потому что мы подключаем наших наблюдателей, когда BikeController включен, но также отсоединяется, когда он отключен; это избавляет нас от необходимости сортировать ссыльки, которые нам больше не нужны:

```

недействительный OnEnable()
{
    если (_hudController)
        Прикрепить (_hudController);

    если (_cameraController)
        Прикрепить (_cameraController);
}

недействительный OnDisable()
{
    если (_hudController)
        Отсоединить (_hudController);

    если (_cameraController)
        Отсоединить (_cameraController);
}

```

И в заключительной части у нас есть несколько базовых реализаций основного поведения. Обратите внимание, что мы уведомляем наблюдателей только тогда, когда параметры мотоцикла обновляются, когда он получает повреждения или когда его турбокомпрессор активируется:

```

частный недействительный StartEngine()
{
    _isEngineOn = правда;
    УведомитьОбсерверов();
}

общественная недействительность ToggleTurbo()
{
    если (_isEngineOn)
        IsTurboOn = !IsTurboOn;

    УведомитьОбсерверов();
}

```

```
}

public void TakeDamage(главающая сумма) {

    здоровье = количество;
    ИстурбоOn = ложь;

    УведомитьОбсерверов();

    если (здоровье < 0)
        Уничтожить (игровой объект);
    }
}
}
```

BikeController никогда не вызывает HUDController или CameraController напрямую он только уведомляет их о том, что что-то изменилось, но никогда не говорит им, что делать.

Это важно, поскольку наблюдатели могут самостоятельно выбирать, как вести себя при получении уведомления. Поэтому они в определенной степени оторваны от предмета.

4. Теперь давайте реализуем несколько наблюдателей и посмотрим, как они ведут себя, когда с объектом подает им сигналы. Начнем с HUDController, который отвечает за отображение пользователя с его интерфейсом

```
использование UnityEngine;

протрансивоимен Chapter.Observer
{общественный класс HUDController: Observer {

    частный bool _isTurboOn; частный
    главящий _currentHealth; частный BikeController
    _bikeController;

    недействительный OnGUI() {
        GUILayout.BeginArea (новый Rect
            (50,50,100,200)); GUILayout.BeginHorizontal
            ("коробка"); GUILayout.Label ("Здоровье: "
                + _currentHealth);
        GUILayout.EndHorizontal ();

        if (_isTurboOn)
            { GUILayout.BeginHorizontal("коробка");
            GUILayout.Label("Турбо активирован!");
            GUILayout.EndHorizontal();
        }
    }
}}
```

```

если (_currentHealth <= 50.0f) {
    GUILayout.BeginHorizontal("коробка");
    GUILayout.Label("ВНИМАНИЕ : низкий уровень здоровья ");
    GUILayout.EndHorizontal();
}

GUILayout.EndArea ();
}

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

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

```

Метод `Notify()` `HUDController` получает ссылку на объект, который его уведомил. Таким образом, он может получить доступ к своим свойствам и выбрать, какие из них отображать в интерфейсе.

5. Наконец, мы собираемся реализовать CameraController. Ожидаем поведение камеры — начать трастик при активации турбо с илителя мотоцикла:

ис пользование UnityEngine;

прос транс тво имен Chapter.Observer {

публичный класс CameraController: Observer

```
частный bool _isTurboOn; частный  
Vector3 _initialPosition; частный плавающий  
_shakeMagnitude = 0,1f; частный BikeController  
bikeController;
```

недействительный OnEnable()

```
_initialPosition =  
    gameObject.transform.localPosition;
```

```
недействительное Обновление
0
если (_isTurboOn) {

    gameObject.transform.localPosition = _initialPosition +
        (Random.insideUnitSphere *
        _shakeMagnitude);

} еще
{
    gameObject.transform.
        localPosition = _initialPosition;
}
}

публичное переопределение void Notify (Тематы) {

    if (!_bikeController) _bikeController
        =
        subject.GetComponent<BikeController>();

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

CameraController проверяет общедоступное логическое свойство объекта, который только что уведомил его, и если оно истинно, он начинает трясти камеру, пока не получит повторное уведомление от BikeController, и не подтвердит, что турбо-переключатель выключен.

Главный вывод из этой реализации, который следует иметь в виду, заключается в том, что BikeController (субъект) не знает, как будут вести себя HUD и контроллер камеры (наблюдатели) после получения уведомления. Следовательно, наблюдатели могут выбирать, как они будут реагировать, когда субъект предупреждает их об изменениях.

Такой подход отделяет эти компоненты контроллера друг от друга. Поэтому их гораздо проще реализовать и отлаживать по отдельности.

## Тестирование реализаций шаблона Observer

Чтобы протестировать нашу реализацию нам нужно следующее:

1. Откройте пустую сцену Unity, но убедитесь, что она включает хотя бы одну камеру и один скрипт.
2. Добавьте в сцену 3D-игровой объект, например куб, и сделайте его видимым для камеры.
3. Прикрепите скрипт BikeController как компонент к новому 3D-объекту.
4. Прикрепите скрипт CameraController к камере основной сцены.
5. Создайте пустой GameObject, добавьте к нему следующий скрипт ClientObserver, а затем запустите сцену:

```
использование UnityEngine;
using System.Collections;

public class Chapter.Observer : MonoBehaviour {
    public BikeController _bikeController;
    public CameraController _cameraController;

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

    void OnGUI()
    {
        if (GUILayout.Button("Поврежденный велосипед"))
        {
            _bikeController.TakeDamage(15.0f);
        }

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

На экране мы должны увидеть кнопки и метки графического интерфейса, подобные следующему:

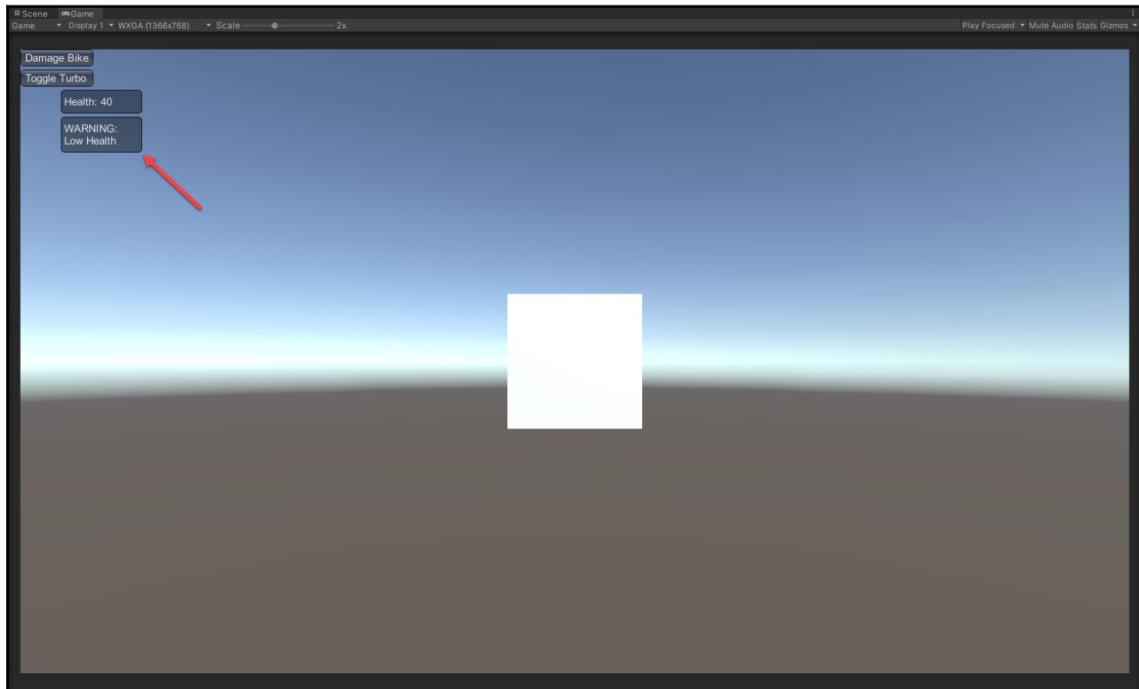


Рисунок 9.3 – Элементы графического интерфейса в работе сцене Unity

Если мы нажмем кнопку `Toggle Turbo`, мы увидим дрожание камеры и отображение HUD. состояния турбоусилителя. Кнопка «Повредить велосипед» уменьшит здоровье велосипеда.



Существует множество разных способов реализации шаблона Observer. Каждый со своими внутренними преимуществами. Я не мог ухватить их все в этой главе. Из-за связи с этим я написал пример кода в этой главе в образовательных целях в разум. Следовательно, это не самый оптимизированный подход, но его легче реализовать. понимать.

## Расмотрение альтернативных решений

Альтернативой шаблону Observer является собственная система с событий C#. Одним из существенных преимуществ этой системы с событий является то, что она более детализирована, чем шаблон Observer, поскольку объекты могут прослушивать определенные события, которые генерирует другой, вместе с тем чтобы получать общее уведомление от объекта.

Всегда следует использовать встроенную систему с событий, если вам нужно, чтобы компоненты взаимодействовали непосредственно с событий, особенно если вам не нужно устанавливать между ними определенные отношения.



Unity имеет собственную систему с событий; он очень похож на версию C#, но с дополнительными функциями движка, такими как возможность связывать события и действия через Инспектор. Чтобы узнать больше, перейдите по адресу <https://docs.unity3d.com/2021.2/Documentation/Manual/UnityEvents.html>.

### Краткое содержание

В этой главе мы узнали, как использовать шаблон Observer для отделения BikeController от его зависимостей, назначая им роли субъекта или наблюдателя. Наши коды теперь легче управлять и расширять, поскольку мы можем легко заставить BikeController взаимодействовать с другими контроллерами с минимальной связью.

Следующей главе мы рассмотрим шаблон «Посетитель», один из самых сложных для изучения шаблонов. Мы будем использовать его для создания усилений, основной механики и ингредиента нашей игры.

# 10

## Реализация усилений с помощью шаблона Posttitle

В этой главе мы собираемся реализовать механизм усиления для нашей игры. Бонусы были основным компонентом видеоигр с момента их появления. Одной из первых игр, в которых были реализованы бонусы, была Pac-Man 1980 года. В игре могли есть Power Pellets, которые давали Pac-Man временную победимость. Еще один классический пример — грибы в Mario Bros., которые делали Марио выше и крепче.

Ингредиенты для усиления, которые мы создадим, будут похожими на классические, но с немногим большей детализацией. Вместе с тем, чтобы усиливать объекты посубъективности, мы можем создавать комбинации, которые дают разные бонусы. Например, мы могли бы иметь усиление под названием «Защитник», которое увеличивает прочность переднего щита и увеличивает силу основного оружия.

Итак, в этой главе мы собираемся реализовать механизм усиления, которую можно масштабировать и настраивать не только для нас, программистов, но и для дизайнеров, на которых может быть возложена ответственность за создание и настройку уникальных ингредиентов усиления. Мы добьемся этого, используя комбинацию шаблона Posttitle и уникальную функцию Unity API под названием ScriptableObjects.

В этой главе будут рассмотрены следующие темы:

- Основные принципы шаблона Posttitle
- Реализация механизма усиления для головной игры.

В этот раздел включены упрощенные примеры кода для простоя и удобства чтения. Если вы хотите просмотреть полную реализацию контексте реального игрового проекта, откройте папку FPP в проекте GitHub, ссылку на которую можно найти в разделе «Технические требования».



## Технические требования

Эта глава вляется практической. Вам потребуется базовое понимание Unity и C#. Мы будем использовать следующий движок Unity и концепцию языка C#:

- Интерфейсы
- ScriptableObjects

Если вы не знакомы с этими понятиями, пожалуйста, ознакомьтесь с ними, прежде чем приступить к этой главе.

Файлы кода для этой главы можно найти по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter10>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/3eeknGC>.



Мы часто используем ScriptableObjects в примерах кода этой книги, потому что при создании игровых систем важно делать их легко настраиваемыми и прогрессивными. Процесс балансировки и системы создания новых ингредиентов обычно лежит на дизайнерах игр и уровней. Поэтому мы используем ScriptableObjects, поскольку он предлагает последовательный способ создания конвейера разработки для создания настройки внутри игровых ресурсов.

## Понимание шаблона «Пос етиль»

Основная цель шаблона «Пос етиль» проста, как только вы ее поймете; Объект Visitable позволяет Постителю работать с определенным элементом его структуры. Этот процесс позволяет посещаемому объекту приобретать новые функциональные возможности от постителей без непосредственного изменения. На первый взгляд это описание может показаться очень абстрактным, но олегчает визуализировать, если представить объект как структуру, а не как закрытый контейнер данных и логики.

Таким образом, с помощью шаблона Поститель можно перемещаться по структуре объекта, работать с его элементами и расширять его функциональность без его изменения.

Другой способ представить паттерн «Поститель» в действии — это представить, как моторикл в нашей игре с тягивается с усилием. Подобно электронному току, мощность протекает через внутреннюю структуру автомобиля. Компоненты, помеченные как доступные для посещения, появляются при включении питания, и добавляются новые функции, но ничего не изменяется.

На следующей диаграмме мы можем визуализировать эти принципы:

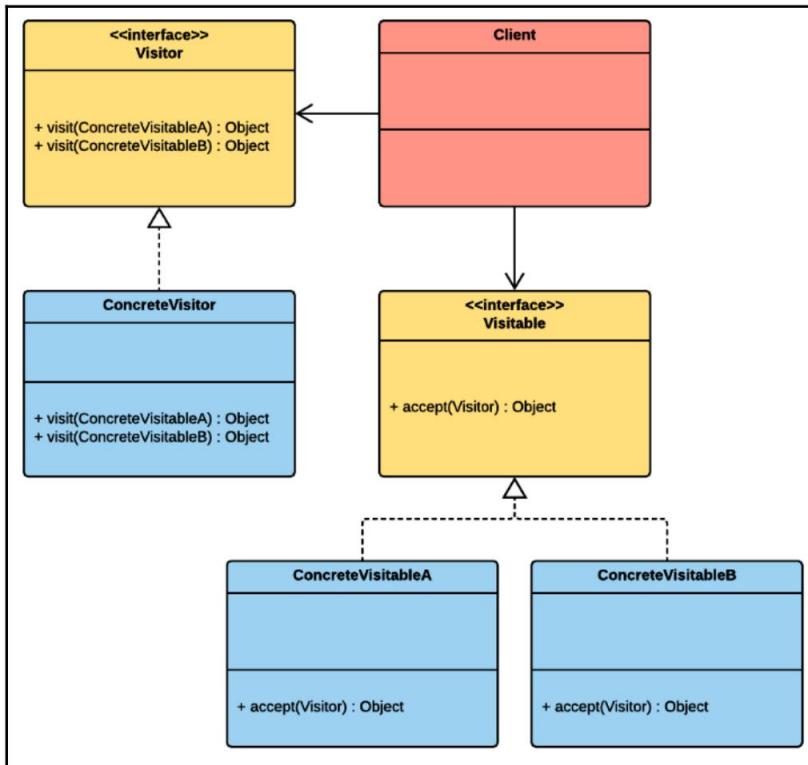


Рисунок 10.1 – UML-диаграмма паттерна «Посетитель»

В этом паттерне есть два ключевых участника, которых нам нужно хорошо знать:

- **IVisitor** — это интерфейс, который должен реализовать класс, желающий стать посетителем. Класс посетителя должен будет реализовать метод посетителя для каждого посещаемого элемента.
- **IVisitable** — это интерфейс, который должны реализовать классы, желающие стать доступными для посещения. Он включает в себя метод `Accept()`, который предлагает точку входа в объект посетителя, чтобы он мог прийти и посетить.

Прежде чем продолжить, важно заметить, что пример кода, который мы рассматриваем в следующих разделах, нарушает потенциальное правило шаблона «Посетитель». В примере объект посетителя изменяется с войсом твои посещенных объектов. Однако вопрос о том, нарушает ли это целесообразность шаблона и лишает законной силы его первоначальный замысел, остается открытым.

Тем не менее, в этой главе мы больше сосредоточимся на том, как шаблон Посетитель позволяет нам перемещаться по элементам, состоящим из структуры посещаемых объектов. В нашем случае эта структура представляет собой новые элементы нашего моторика, в том числе двигатель, щит и новое оружие.



Некоторые считают Посетителя одним из самых сложных для понимания шаблонов. Так что не расстраивайтесь, если поначалу вы не поймете его основные концепции. Я считаю, что одна из причин, по которой эту модель сложно выучить, заключается в том, что ее сложнее всего объяснить.

## Преимущества и недостатки шаблона «Посетитель»

Я написал краткий список преимуществ и недостатков использования этого шаблона.

Ниже приведены преимущества:

- Открытый/закрытый: вы можете добавлять новые варианты поведения, которые могут работать с объектами разных классов, не изменяя их напрямую. Этот подход следует принципу объектно-ориентированного программирования Open/Closed, который гласит, что объекты должны быть открыты для расширения, но закрыты для модификации.
- Единая ответственность: шаблон «Посетитель» может соответствовать принципу единой ответственности в том смысле, что у вас может быть объект (видимый), который содержит данные, другой объект (посетитель) отвечает за введение определенного поведения.

Ниже приведены некоторые из потенциальных недостатков:

- Доступность. Посетителя может нехватать необъектов конкретным частным полями и методами элементов, которые они посещают. Следовательно, нам может потребоваться языке больше общедоступных свойств в наших классах, чем обычно, если бы мы не использовали шаблон.
- Сложность. Можно утверждать, что шаблон «Посетитель» структурно более сложен, чем простые шаблоны, такие как «Одиночка», «Состояние» и «Пул объектов». Следовательно, это может усложнить вашу кодовую базу, что может быть с трудом другим программистом, если они не знакомы со структурой и тонкостями шаблона.



Посетитель использует концепцию иерархии для обработки информации в своей фундаментальной конструкции под названием Double Dispatch. Постепенное определение этой концепции состоит в том, что это механизм, который распределяет вызов метода по различным конкретным методам в зависимости от типов двух объектов, участвующих в вызове во время выполнения. Не обязательно полностью понимать эту концепцию, чтобы следовать примеру шаблона, представленного в этой главе.

## Проектирование механики включения питания

Как упоминалось в начале этой главы, усиление является основным элементом видеоигр. И это основной ингредиент нашей игры. Но сначала мы рассмотрим некоторые ключевые характеристики нашей механики:

- Детализация: наши существа-исследователи могут усиливать не только с войсками одновременно. Например, мы могли бы иметь усиление, которое увеличивает наступательные возможности, такие как дальность действия основного оружия, при ремонте переднего щита.
- Время: эффекты усиления не являются постоянными, поэтому они исчезают через определенное время. Преимущества следующего включения добавляются поверх предыдущего, пока не достигнут максимальных усиленных свойств.

Обратите внимание, что эти спецификации ограничены следующим примером кода, но не являются окончательными. Мы могли бы легко сделать преимущество от включения временными или изменить общую конструкцию механики, внеся небольшие изменения в пример кода, представленный в следующем разделе.

Наши дизайнеры уровней разместят бонусы в стратегических точках игровой трассы; они будут иметь трехмерные формы, которые легко обнаружить на высокой скорости. Игровому придется столкнуться с усилением, чтобы активировать содержащиеся в нем способности преимущества.

Мы будем использовать комбинацию шаблона Visitor и ScriptableObjects для реализации этой игровой механики, чтобы дизайнеры могли создавать новые варианты бонусов без необходимости писать единую строку кода.



В видеоиграх основное различие между предметом и усилением заключается в том, что игрушка может сортировать предметы, хранить их и выбирать, когда использовать их преимущества. Но, напротив, усиление вступает в силу сразу после того, как игрушка к нему приводится.

## Реализация механики включения питания

В этом разделе мы напишем необх одимый скелетный код для реализации системы включения питания с помощью шаблона Постройка. Наша цель — получить действительное доказательство концепции к концу этого раздела.

## Реализация системы включения питания

Рассмотрим этапы реализации:

1. Начнем с написания основного элемента шаблона — интерфейса построителя:

```
просранствоимен Pattern.Visitor {
    общедоступныйинтерфейс IVisitor {
        void Visit(BikeShield BikeShield); недействительный
        визит (BikeEngine BikeEngine); void Visit (BikeWeapon
        BikeWeapon);
    }
}
```

2. Далее мы собираемся написать интерфейс, который будет иметь каждый построимый элемент. реализовать:

```
просранствоимен Pattern.Visitor {
    общедоступныйинтерфейс IBikeElement {
        void Accept (постройтель IVisitor);
    }
}
```

3. Теперь, когда у нас есть основные интерфейсы, давайте реализуем основной класс, который заставляет наш механизм включения работать; Из-за его однородности мы рассмотрим его в двух частях:

использование UnityEngine;

```
просранствоимен Pattern.Visitor {
```

```
[CreateAssetMenu (fileName = «PowerUp», MenuName = «PowerUp»)] общедоступный класс
PowerUp: ScriptableObject, IVisitor {
```

общедоступная строка powerupName;

```

    публичный GameObject powerupPrefab;
    публичная строка powerupDescription;

    [Tooltip("Полностью исцеляя щит")]
    public bool healShield;

    [Диапазон(0,0f, 50f)]

    [Подсказка("Увеличьте настройку до 50/миль в час")]
    общесственный поплавок TurboBoost;

    [Диапазон(0,0f, 25)]

    [Tooltip("Увеличение дальности оружия с шагом до 25 единиц")]
    public int WeaponRange;

    [Диапазон(0,0f, 50f)]

    [Подсказка("Повышает силу оружия с шагом до 50%")]
    публичное плавающее оружие Сила;

```

Первое, на что следует обратить внимание, это то, что этот класс представляет собой ScriptableObject с атрибутом CreateAssetMenu. Таким образом, мы можем использовать его для создания новых ресурсов усиления из меню «Активы». Итог да мы можем настраивать параметры каждого нового оружия в Инспекторе движателя. Но еще одна важная деталь заключается в том, что этот класс реализует интерфейс IVisitor, который мы рассмотрим в следующей части:

```

public void Visit(BikeShield BikeShield) {
    если (healShield)
        BikeShield.health = 100.0f;
}

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

    if (диапазон >= BikeWeapon.maxRange)
        BikeWeapon.range = BikeWeapon.maxRange; еще

    BikeWeapon.range = диапазон;

    силы плавания =
    BikeWeapon.strength +=

    Mathf.Round( BikeWeapon.strength

```

```
*Сила оружия /100);

if (сила >= BikeWeapon.maxStrength)
    BikeWeapon.strength = BikeWeapon.maxStrength; иначе

    BikeWeapon.strength = сила;
}

public void Visit(BikeEngine BikeEngine) {

    float boost = BikeEngine.turboBoost += TurboBoost;

    if (boost < 0,0f)
        BikeEngine.turboBoost = 0,0f;

    если (boost >= BikeEngine.maxTurboBoost)
        BikeEngine.turboBoost = BikeEngine.maxTurboBoost;
    }
}
}
```

Как мы видим, для каждого посетителя у нас есть уникальный метод, связанный с ним, внутри каждого из них мы реализуем операцию которую хотим выполнить при посещении определенного элемента.

В нашем случае мы меняем конкретные свойства посещаемого объекта с учетом заданных максимальных значений. Поэтому мы инкапсулируем ожидаемое поведение усиления посещения определенного посещаемого элемента с помощью структуры велосипеда внутри отдельных методов Visit().

Нам нужно изменить определенные значения, изменить операции и добавить новое поведение для определенного посещаемого элемента, мы можем делать эти вещи внутри этого единственного класса.

4. Следующим идет класс BikeController, отвечающий за управление ключевыми компонентами велосипеда, состоящими его структурой:

```
использование UnityEngine;
использование System.Collections.Generic;

последовательность имен Pattern.Visitor {

    публичный класс BikeController: MonoBehaviour, IBikeElement {

        частный список<IBikeElement> _bikeElements = новый
        список<IBikeElement>();

        недействительный стартер()
    }
}
```

```
{  
  
    _bikeElements.Add( gameObject.AddComponent<BikeShield>());  
  
    _bikeElements.Add( gameObject.AddComponent<BikeWeapon>());  
  
    _bikeElements.Add( gameObject.AddComponent<BikeEngine>());  
}  
  
public void Accept(посетитель IVisitor) {  
  
    foreach (элемент IBikeElement в _bikeElements) {  
  
        элемент.Принять(посетитель);  
    }  
}  
}  
}
```

Обратите внимание, что класс реализует метод Accept() из интерфейса IBikeElement. Этот метод будет вызываться автоматически, когда методикл стартует с предметом усиления, расположенным на очной трассе. Используя этот метод, объект включения может передать объект посетителя велосипед контроллер.

Контроллер продолжит пересылку полученного объекта посетителя каждому из его посещаемых элементов. А посещаемые элементы обновят свои свойства, как это и было в примере объекта посетителя. Следовательно, именно так работает механизм включения питания.

5. Пришло время реализовать наши отдельные посещаемые элементы, начиная с нашего скелета класса BikeWeapon:

```
ис пользование UnityEngine;  
  
прос транс тво имен Pattern.Visitor {  
  
    публичный класс BikeWeapon: MonoBehaviour, IBikeElement {  
  
        [Header("Range")] public  
        int range = 5; общес твенный  
        ИНТ maxRange = 25;  
  
        [Header("Strength")] public  
        float Strength = 25.0f;  
    }  
}
```

```
общедоступное число огня
```

```
    {
        Debug.Log("Оружие выс трелило!");
    }
```

```
public void Accept (посетитель IVisitor) {
```

```
    посетитель.Посетить(это);
}
```

```
недействительный OnGUI()
```

```
{
```

```
    GUI цвет = Цвет.зеленый;
```

```
    GUI.Label(
        новый прямотекущий (125, 40, 200, 20),
        «Диапазон оружия : » + диапазон);

```

```
    GUI.Label(
        новый прямотекущий (125, 60, 200, 20),
        «Сила оружия : » + сила);
}
```

```
}
```

6. Ниже приведен класс с BikeEngine ; обратите внимание, что в полном объеме реализации этот класс будет отвечать за моделирование некоторых режимов работы двигателя , включая включение турбонагнетателя , управление системой охлаждения и контроль сброса топлива:

использование UnityEngine;

прос транс тво имен Pattern.Visitor {

```
public class BikeEngine : MonoBehaviour, IBikeElement {
    // общедоступное плавающее турбоБуст = 25.0f; // миль/ч public
    float maxTurboBoost = 200.0f;
    // частный bool _isTurboOn; частный
    float _defaultSpeed = 300.0f; // миль в час
    // общедоступное плавающее значение CurrentSpeed
    {
        // получать
    }
}
```

```

        if (_isTurboOn) возвращет
            _defaultSpeed + TurboBoost; вернуть _defaultSpeed;

    }

}

общественная недействительность ToggleTurbo()
{
    _isTurboOn = !_isTurboOn;
}

public void Accept (посетитель IVisitor) {

    посетитель.Посетить(это);
}

недействительный OnGUI()
{
    GUI.цвет = Цвет.зеленый;

    GUI.Label(
        новый Rect(125, 20, 200, 20), + TurboBoost);
        "Турбоускорение:
    )
}
}
}

```

7. Итак, BikeShield, название которого о подразумевает его основную функцию — это сведение:

```

использование UnityEngine;

последовательно имен Pattern.Visitor {

общественный класс BikeShield: MonoBehaviour, IBikeElement {

    состояния общественного плавающего состояния = 50.0f; // Процент

    public float Damage (плавающий ущерб) {

        здоровье = урон; вернуть
        здоровье;
    }

    public void Accept (посетитель IVisitor) {

        посетитель.Посетить(это);
    }
}

```

```
недействительный OnGUI()
{
    GUI.цвет = Цвет.зеленый;

    GUI.Label(новый
        Rect(125, 0, 200, 20), + здоровье);
    "Здоровье Штата"
}
}
```

Как мы видим, каждый отдельный посещаемый класс элемента ведет себя реализуя метод Accept(), тем самым делая себя посещаемым.

## Тестирование реализации систмы включения питания

Чтобы было проще протестировать нашу реализацию с помощью собственного экземпляра Unity, вам необходимо выполнить следующие шаги:

1. Скопируйте все скрипты, которые мы только что расмотрели, в свой проект Unity.
2. Создайте новую сцену.
3. Добавьте на сцену GameObject.
4. Присоедините следующий скрипт ClientVisitor к новому GameObject:

```
ис пользование UnityEngine;

пространство имен Pattern.Visitor {

общественный класс ClientVisitor: MonoBehaviour {

объект дружественный движок PowerUpPowerUp;
объект дружественный щит PowerUpPowerUp; публичное
усиление оружия PowerUp;

частный BikeController _bikeController;

недействительный Старт
{
    _bikeController = gameObject.

       .AddComponent<BikeController>();

}

недействительный OnGUI()
{
```

```

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

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

if (GUILayout.Button("Усиление оружия "))
    _bikeController.Accept(weaponPowerUp);
}
}
}
}

```

5. Перейдите в пункт меню «Активы/Создать» и создайте три ресурса PowerUp.

6. Настройте и назовите новые ресурсы PowerUp с нужными параметрами.

7. Назовите новые ресурсы PowerUp и настройте их параметры в Инспекторе.

8. Добавьте новые ресурсы PowerUp в общедоступные свойства ClientVisitor.

компонент.

После запуска сцены вы должны увидеть на экране следующие кнопки графического интерфейса и их одни данные отладки:

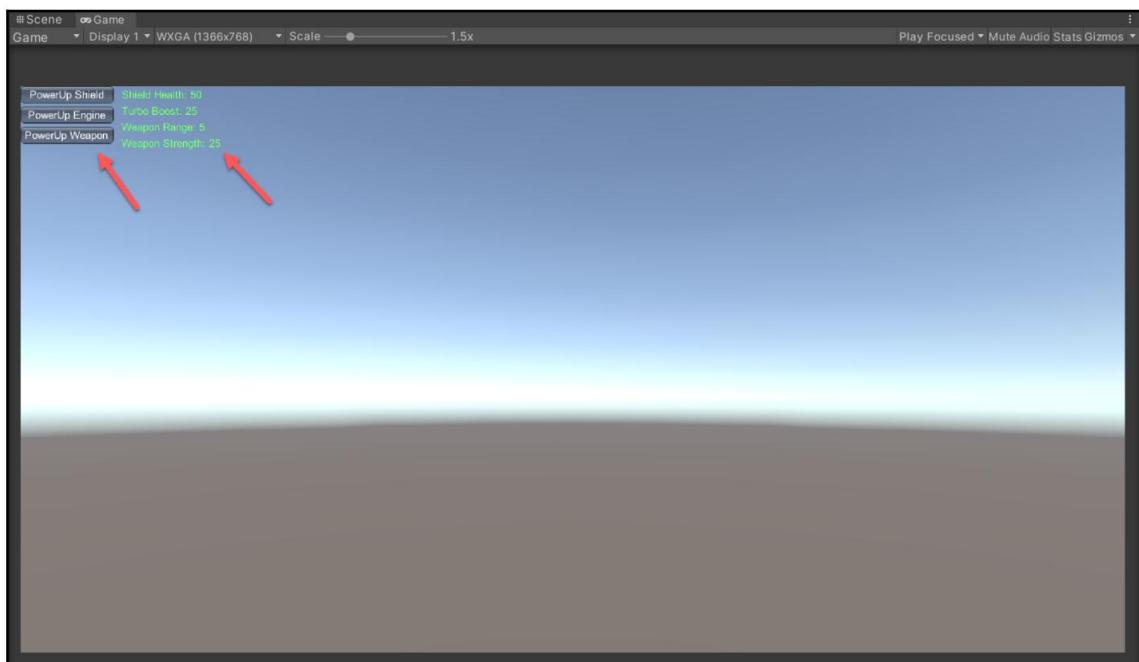


Рисунок 10.2 – Скриншот примера кода в действии

Но вы можете спросить себя, как мне создать настоящий пикап, который можно будет создавать на гоночной трассе?

Быстрый способ сделать это — просто создать класс с Pickup, подобный следующему:

```
ис пользование
системы; ис пользование UnityEngine;

Пикап() : MonoBehaviour {

    public PowerUp powerup;

    void OnTriggerEnter(Collider other) {
        if (other.GetComponent<BikeController>()) {
            другой.GetComponent<BikeController>().Accept(powerup); Уничтожить (игровой
            объект);
        }
    }
}
```

Прикрепив этот скрипт к GameObject с компонентом коллайдера, настроенным как тригер, мы можем определить, когда объект с компонентом BikeController входит в тригер. Затем нам нужно вызвать его метод Accept() и передать экземпляр PowerUp. Настройка триггеров в рамках этой главы, но я рекомендую просмотреть проект FPP в репозитории Git, чтобы узнать, как мы настроили его в игровом прототипе игры.

## Обзор реализации систе мы включения питания

Нам удалось объединить структуру шаблона «Посетитель» и функции API ScriptableObjects, чтобы создать механизм активации, который позволяет любому участнику нашего проекта создавать и настраивать новые бонусы без написания единой строки кода.

Если нам нужно настроить влияние усилий на различные компоненты нашего автомобиля, мы можем сделать это, изменив один класс. Итак, включение мы добились определенной степени масштабируемости, сохранив при этом простоя сопровождения нашего кода.



Реализации с тандемных шаблонов проектирования программного обеспечения в этой книге являются экспериментальными и творчески адаптированными. Мы адаптируем их для использования функций Unity API и адаптируем их для ценарий разработки. Поэтому мы не должны рассматривать примеры как академические или стандартизированные ссылки, а просто как интерпретации.

### Краткое сведение

В этой главе книги мы создали механику усиления для нашей игры с помощью посетителя. Узор в качестве нашей системы. Мы также наладили рабочий процесс для создания и настройки усиления. Поэтому мы объединили технический и творческий подход, который является основой новой игры. разработка.

В следующей главе мы собираемся разработать и реализовать маневры атаки противника. дроны. Мы собираемся использовать шаблон «Стратегия» в качестве новой нашей системы. разработка.

# 11

## Реализация дрона с помощью

### Стратегический шаблон

В этой главе мы собираемся реализовать вражеские дроны, которые летают по гоночной трассе и атакуют игрука, стреляя лазерными лучами. Это маленькие надеждливые роботы-вредители, которые проверяют рефлексы игрока. Наши дроны будут проводить одну атаку, состоящую из непрерывного высева трела лазерным лучом под углом 45 градусов. Чтобы создать иллюзию автономного интеллекта, дронам во время работы можно назначить три различных атакующих маневра. Каждый маневр предстает с обойми повторяющимися серией предсказуемых движений. По отдельности поведение дронов может выглядеть роботизированным, но когда их размещают встроена определенных позициях на гоночной трассе, может показаться, что они формируют стратегию для перехватить игрока.

Итак, я предлагаю использовать шаблон «Стратегия» для реализации различных вариантов поведения дронов. Основная причина этого выбора заключается в том, что этот шаблон позволяет нам назначать объекту определенное поведение во время выполнения. Начиная с этого момента, разберемся в характеристиках модели и цели конструкции нашего вражеского дрона.



Для краткости и сноски в эту главу включены упрощенные примеры скелетного кода. Если вам интересно просмотреть полную реализацию шаблона в контексте реального игрового проекта, откройте папку FPP в проекте GitHub. Ссылку вы можете найти в разделе Технические требования.

В этой главе мы рассмотрим следующие темы:

- Обзор паттерна «Стратегия»
- Реализация поведения атакующих дронов

## Технические требования

Эта глава является практической, поэтому вам потребуется базовое понимание Unity и C#.

Файлы кода для этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter11>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/2TdeoL4>.

## Понимание шаблона с тратегии

Основная цель шаблона «Стратегия» — отложить принятие решения о том, какое поведение использовать во время выполнения. Это стало возможным потому, что шаблон «Стратегия» позволяет нам определить семейство поведений, инкапсулированных в отдельные классы, которые мы называем стратегиями. Каждая стратегия взаимодействует и может быть назначена любому объекту контекста для изменения его поведения.

Давайте визуализируем ключевые элементы шаблона с помощью UML-диаграммы:

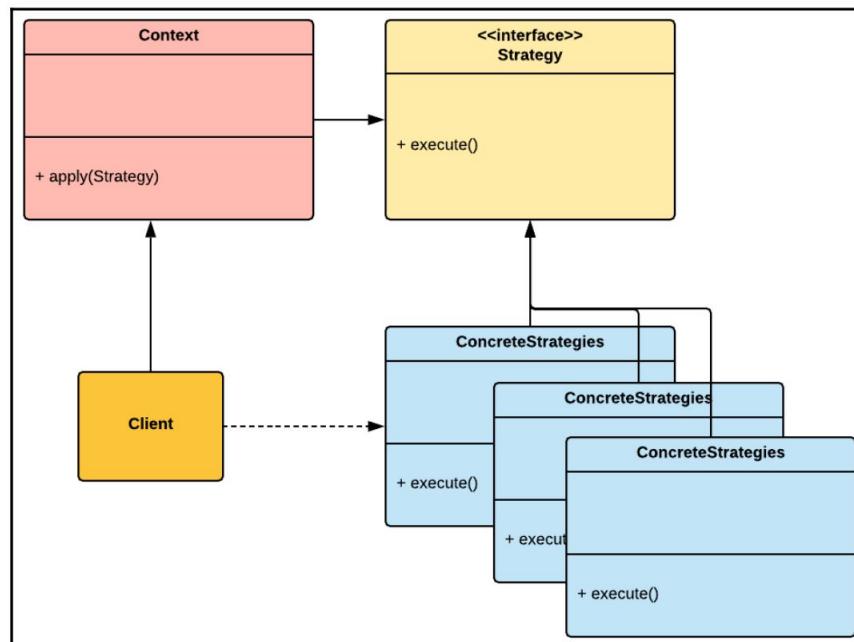


Рисунок 11.1 – UML-диаграмма паттерна «Стратегия»

Вот разбивка ключевых игроков паттерна:

- Контекст — это класс, который использует различные классы конкретных стратегий и взаимодействует с ними через интерфейс Strategy.
- Интерфейс Strategy является общим для всех конкретных классов с тратегией. Он предоставляет метод, который класс Context может использовать для выполнения стратегии.
- Классы конкретных стратегий, также известные как стратегии, представляют собой конкретные реализации алгоритмов/поведений, которые можно применять к объекту Context во время выполнения.

Наданный момент эти концепции могут показаться очень абстрактными, но на практике их довольно легко понять, как мы увидим позже в этой книге.



Шаблон «Стратегия» — это поведенческий шаблон проектирования программного обеспечения; его ближайшим родственником является шаблон State. Мы можем использовать оба, чтобы инкапсулировать набор поведений в отдельных классах. Шаблон «Стратегия» следует использовать, если вы хотите выбрать поведение во время выполнения и применить его к объекту. Вы также можете использовать шаблон «Состояние», если хотите, чтобы объект менялся свое поведение при изменении его внутреннего состояния.

## Преимущества и недостатки паттерна «Стратегия»

Вот некоторые преимущества шаблона «Стратегия»:

- Инкапсуляция. Очевидным преимуществом этого шаблона является то, что он обеспечивает инкапсуляцию различных алгоритмов в отдельные классы. Следовательно, это помогает нам избежать использования длинных условных операторов, с которых раньше при этом структуру нашего кода.
- Время выполнения. Основное преимущество этого шаблона заключается в том, что он реализует механизм, который позволяет нам менять алгоритмы, используемые объектом во время выполнения. Такой подход делает наши объекты более динамичными и открытыми для расширения.

Ниже приведены некоторые потенциальные недостатки шаблона «Стратегия»:

- Клиент: Клиентский класс должен знать об отдельных стратегиях и вариантах реализуемых ими алгоритмов, чтобы знать, какой из них выбрать.  
Таким образом, клиент становится ответственным за то, чтобы объект вел себя должным образом в течение всего срока службы.

- Путаница. Поскольку шаблоны «Стратегия» и «Состояние» очень похожи по структуре, но имеют разные цели, при выборе того, какой из них использовать в каком контексте, может возникнуть путаница. В большинстве случаев это не проблема, но если вы работаете с командой программистов, в зависимости от уровня знаний предмета некоторые коллеги могут не понять ваш выбор шаблона.



Я считаю что очень важно регулярно проводить открытые дискуссии об архитектуре, шаблонах и лучших практиках с вашими коллегами. Если вы как команда сможете договориться об общих подходах при использовании определенного набора шаблонов проектирования, вы получите более согласованную архитектуру и более чистый код.

## Когда использовать паттерн «Стратегия»

Когда мне поручают реализовать поведение вражеского персонажа, первыми вариантами, которые я рассматриваю, являются шаблон состояния или конечный автомат (FSM), поскольку в большинстве случаев персонаж имеет состояние.

Но иногда могу использовать шаблон «Стратегия», если выполняются следующие условия:

- У меня есть сущность с несколькими вариантами одного и того же поведения, и я хочу инкапсулировать их в отдельные классы.
- Я хочу назначить определенные варианты поведения объекту во время выполнения без необходимости учитывать его текущее внутреннее состояние.
- Мне нужно применить поведение к сущности, чтобы она могла выполнять конкретную задачу на основе критерии выбора, определенных во время выполнения.

Третий момент, вероятно, я вляется основной причиной, по которой я решил использовать шаблон «Стратегия» вместе с шаблоном «Состояние» для реализации вражеского дрона, представленного в этой главе. Поведение дрона роботизировано; у него единственная задача: атаковать игрока. Он не вносит никаких изменений в свое действие в зависимости от изменений внутреннего состояния. Ему нужно только назначить поведение атаки во время выполнения, чтобы выполнить задачу атаки на игрока, что делает его подход однозначным кандидатом для шаблона Стратегии в его текущем дизайне.

Важно отметить, что потенциальные варианты использования шаблона «Стратегия» не ограничиваются реализацией вражеских персонажей. Например, мы могли бы использовать его для инкапсуляции различных алгоритмов шифрования для применения к социальному файлу, в зависимости от целевой платформы. Или, если мы работаем над фэнтезийной игрой, мы могли бы использовать ее для инкапсуляции индивидуального поведения с элементами заклинаний, которые игроки могут применять к целевой сущности. Таким образом, потенциальные варианты использования этого шаблона широки и могут применяться в различных контекстах: от основных систем до игровых механик.

В следующем разделе мы рассмотрим конструктивные замыслы нашего вражеского дрона.

## Проектирование вражеского дрона

Вражеские дроны в нашей игре не очень умны; за счет чего не работает искусственный интеллект. Это работы с роботизированным поведением, и в видеонентах часто встречаются враги с предсказуемым автоматическим поведением, работающие по циклу. Например, Гумбы в оригинальном Super Mario Bros просто ходят в одном направлении; они не знают о присутствии Марио и не реагируют на него. Они просто запускаются алгоритмом, который заставляет их блуждать по пути, пока они не столкнутся с препятствием. По отдельности они не представляют угрозы, но если их высаживать в строй или расположить в точке карты, в которой затруднена навигация, их может оказаться сложно избежать.

Мы будем использовать тот же подход для наших вражеских дронов. По отдельности они легко победить, потому что они не могут меняять свое поведение в зависимости от движений игрока, но в отряде их может быть сложно избежать.

У нашего дрона есть три различных маневра атаки; каждый из них вращается вокруг определенного набора движений, которые предсказуемы, но которым все же нужно противостоять, когда дроны высажены в отряд.

Давайте рассмотрим каждый маневр:

- Маневр покачивания: при маневре покачивания дрон движется вверх и вниз на высоте с користи, стреляя лазерным лучом.
- Ткачий маневр: при выполнении маневра дрон движется горизонтально на высоте с користи во время стрельбы. Маневр переплетения ограничен расположением между двумя рельсами пути.
- Откатный маневр: при отступлении дрон движется назад во время стрельбы.

Максимальная скорость дрона может соответствовать скорости велосипеда игрока, но он может двигаться назад только в течение ограниченного времени.

Следующий диаграмма иллюстрирует предыдущие маневры:

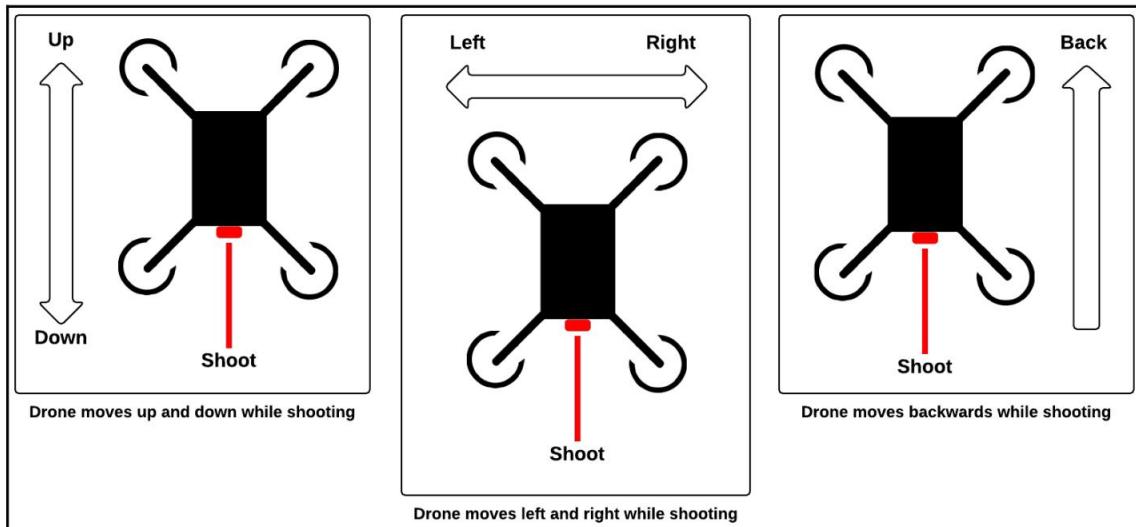


Рисунок 11.2 – Иллюстрация атакующих маневров дрона

У вражеского дрона есть единственное оружие: направленный вперед лазерный луч, стреляющий под углом 45 градусов к земле. На следующей схеме показано лазерное оружие дрона:

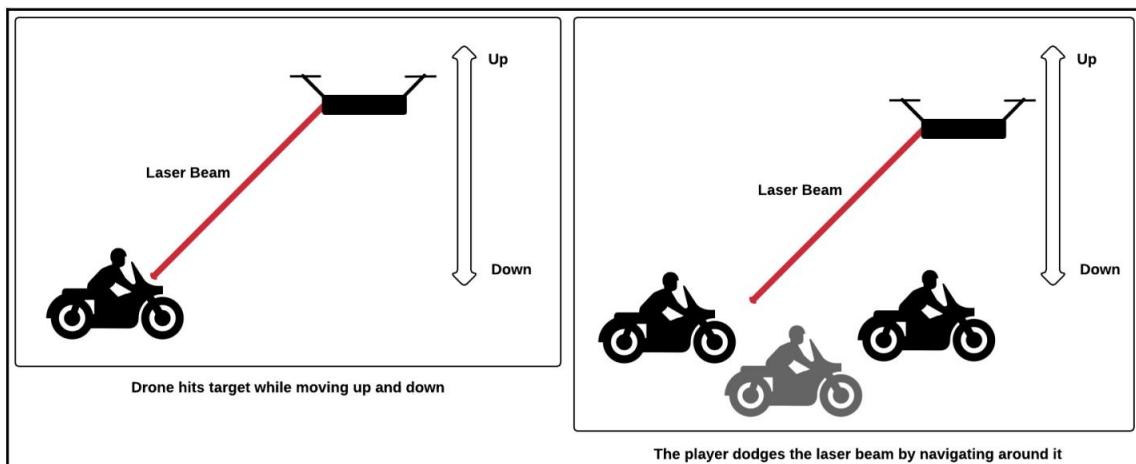


Рисунок 11.3 – Иллюстрация боевой атаки дрона

Как мы видим, игрок должен избегать атаки, перемещаясь вокруг дрона на высокой скорости. При попадании луча передний щит мотоцикла потеряет определенное количество мощности.

Если щит иссякнет, машина взорвётся при следующем попадании, и игра будет завершена.

В следующем разделе мы собираемся перевести этот проект в код.

## Реализация вражеского дрона

В этом разделе мы напишем скелетную реализацию паттерна «Стратегия» и индивидуальное поведение атак и дронов-противников. В некоторых аспектах код в этом разделе может показаться слишком упрощенным. Тем не менее, конечная цель — не полная реализация вражеского дрона, а понимание основ паттерна «Стратегия».

## Шаги по внедрению вражеского дрона

Начнем с реализации новых ингредиентов паттерна «Стратегия»:

1. Наш первый элемент — интерфейс стратегии; все наши конкретные стратегии будут использовать его:

```
прос транс тво имен Chapter.Strategy {
    общедоступный интерфейс IManeuverBehaviour {
        void Maneuver (Дрон-дрон);
    }
}
```

Обратите внимание, что мы передаем в метод Maneuver() параметр типа Drone. Это важная деталь, которую мы рассмотрим позже.

2. Далее идет наш класс дронов; он будет использовать наши конкретные стратегии, поэтому в общую структуру шаблона Стратегии, мы будем считать ее нашим классом Контекста:

```
ис пользование UnityEngine;
прос транс тво имен Chapter.Strategy {
    публичный класс Drone: MonoBehaviour {
        // Параметры луча Private
        RaycastHit _hit; частный Vector3
        _rayDirection;
```

```
час тный float _rayAngle = -45.0f; час тный float
_rayDistance = 15.0f;

// Параметры движения public
float speed = 1.0f; общедоступное число
с главающей земли той maxHeight = 5.0f; public
float weavingDistance = 1.5f; public float backDistance =
20.0f;

void Start()
{
    _rayDirection =
        Transform.TransformDirection(Vector3.back) * _rayDistance;

    _rayDirection =
        Quaternion.Euler(_rayAngle, 0.0f, 0f) * _rayDirection;
}

public void ApplyStrategy(стратегия IManeuverBehaviour) { Strategy.Maneuver(this);

}

void Update()
{
    Debug.DrawRay(transform.position, _rayDirection,
        Color.blue);

    if (Physics.Raycast(transform.position,
        _rayDirection, out _hit,
        _rayDistance)) {

        если (_hit.collider) {

            Debug.DrawRay(transform.position,
                _rayDirection, Color.green);
        }
    }
}
}
```

Большинство строк кода в этом классе предназначены для рассылки отладочной информации; мы можем сюда норировать их. Однако следующий раздел не обходим для понимания:

```
public void ApplyStrategy(стратегия IManeuverBehaviour) {
    стратегия.Маневр(это);
}
```

Метод `ApplyStrategy()` с одержит основной механизм шаблона «Стратегия». Если присмотреться, то можно увидеть, что расматриваемый метод принимает в качестве параметра конкретную стратегию и типа `IManeuverBehaviour`. И здесь все становится очень интересно. Объект `Drone` может взаимодействовать с конкретными стратегиями, которые он получил через интерфейс `IManeuverBehaviour`. Таким образом, для выполнения стратегии во время выполнения достаточно вызвать `Maneuver()`. Следовательно, объекту `Drone` не нужно знать, как выполняется поведение/алгоритм стратегии — ему просто нужно знать его интерфейс.

Теперь давайте реализуем конкретные классы стратегии:

1. Следующий класс реализует маневр «бопинг»:

```
использование UnityEngine;
использование System.Collections;

просмотреть имена Chapter.Strategy {
    публичный класс BoppingManeuver:
        MonoBehaviour, IManeuverBehaviour {

    public void Maneuver(Дрон-дрон) {
        StartCoroutine(Bopple(дрон));
    }

    IEnumerator Bopple(Дрон-дрон) {
        время плавания;
        Болт isReverse = ложь; с коростью
        плавания = дрон.корость; Vector3
        startPosition = drone.transform.position; Vector3 endPosition = startPosition;
        endPosition.y = drone.maxHeight;

        в это время как (истина) {
            время = 0;
            Vector3 start = drone.transform.position; Конец вектора3 =

```

```
(isReverse) ? начальная позиция : конечная позиция ;  
  
while (время < скорость)  
{drone.transform.position =  
    Vector3.Lerp(начало, конец, время /скорость);  
время += Время.дельтавремя ;  
выход одной возврат нулевой;  
}  
  
дох односТЬ возврата новых WaitForSeconds (1); isReverse  
= !isReverse;  
}  
}  
}  
}
```

## 2. Следующий класс с реализует маневр переплетения :

```
использование UnityEngine;  
использование System.Collections;  
  
прототип твоим Chapter.Strategy {  
общедоступный класс с WeavingManeuver:  
MonoBehaviour, IManeuverBehaviour {  
  
    public void Maneuver (Дрон-дрон) {  
        StartCoroutine(Weave(дрон));  
    }  
  
    IEnumerator Weave (Дрон-дрон) {  
        время плавания ;  
        Болл isReverse = ложь; скорость  
        плавания = дрон.скорость; Vector3  
        startPosition =drone.transform.position; Vector3 endPosition = startPosition;  
        endPosition.x =drone.weavingDistance;  
  
        в то время как  
        (истина)  
        {время = 0; Vector3 start =drone.transform.position; Конец  
        вектор3 =  
            (isReverse) ? начальная позиция : конечная позиция ;  
  
            while (время < скорость)  
            {drone.transform.position =  
                Vector3.Lerp(начало, конец, время /скорость); время +=  
                Время.дельтавремя ; выход одной  
                возврат нулевой;  
            }
```

```
        }

        if(дрон.одинаковы возвраты новых WaitForSeconds(1); isReverse
        = !isReverse;
    }
}
}
}
```

### 3. Наконец, реализуем запасной маневр:

```
использование UnityEngine;
использование System.Collections;

пространство имен Chapter.Strategy {

общедоступный класс с FallbackManeuver:
MonoBehaviour, IManeuverBehaviour {

    public void Maneuver (Дрон дрон) {
        StartCoroutine(Fallback(дрон));
    }

    IEnumerator Fallback (Дрон дрон) {

        время плавания = 0;
        скорость плавания = дрон.скорость;
        Vector3 startPosition = drone.transform.position; Vector3 endPosition = startPosition;
        endPosition.z = дрон.fallbackDistance;

        while (время < скорость) {

            drone.transform.position = Vector3.Lerp(
                startPosition, endPosition, время / скорость);

            время += Время.дельтавремя;

            вых однократный возврат нулевой;
        }
    }
}
```

Возможно, вы заметили, что код каждого класса очень похож, даже в некоторых частях повторяется. Это объясняется тем, что мы используем шаблон «Стратегия»: мы хотим инкапсулировать варианты схем поведения, чтобы их было легче поддерживать индивидуально. Но также представьте, насколько беспорядочным был бы наш класс Drone, если бы мы попытались реализовать поведение Bopping, Weaving и Fallback в одном классе. Мы окажемся в раздражении классе Drone, который потенциально до края в заполнен ус ловными операторами.



Я бы не рекомендовал использовать спраги раммы для анимации неуманоидных объектов. Вместо этого я предлагаю использовать движок Tween, такой как DOTween, поскольку вы можете анимировать объекты с меньшим количеством кода, получая при этом лучшие результаты. В этой главе мы используем спраги раммы, чтобы избежать внешних зависимостей и упростить портирование нашего кода. Чтобы узнать больше о DOTween, посетите <http://dotween.demigiant.com>.

## Тестирование реализации вражеского дрона

А теперь самое интересное — тестирование нашей реализации. Это будет несложно, поскольку все, что нам нужно сделать, это прокомпилировать следующий клиентский класс к пустому GameObject в сцене Unity:

```
использование UnityEngine;
использование System.Collections.Generic;

последование Chapter.Strategy {
    общественный класс ClientStrategy: MonoBehaviour {

        частный GameObject _drone;

        частный списки<IManeuverBehaviour>
        _comComponents = новый список<IManeuverBehaviour>();

        частный недействительный SpawnDrone
        () { _drone =
            GameObject.CreatePrimitive(PrimitiveType.Cube);

            _drone.AddComponent<Drone>();

            _drone.transform.position =
                случайный.insideUnitSphere * 10;

            ПрименитьСлучайныеСтратегии();
        }

        частная пустота ApplyRandomStrategies() {
            _компоненты.Добавить(

```

```

        _drone.AddComponent<WeavingManeuver>());

        _comComponents.Add( _drone.AddComponent<BoppingManeuver>());

        _comComponents.Add( _drone.AddComponent<FallbackManeuver>());

        int index = Random.Range(0, _comComponents.Count);

        _drone.GetComponent<Дрон>().
            ApplyStrategy(_comComponents[индекс]);
    }

    void OnGUI() { if
        (GUILayout.Button("Spawn Drone")) {
            СпаунДрон();
        }
    }
}
}
}

```

Если в вашем экземпляре Unity вы включили в свой проект все скрипты, которые мы написали в предыдущих разделах, при запуске вы должны увидеть на экране одну кнопку под названием Spawn . Дрон, как показано на следующем скриншоте:

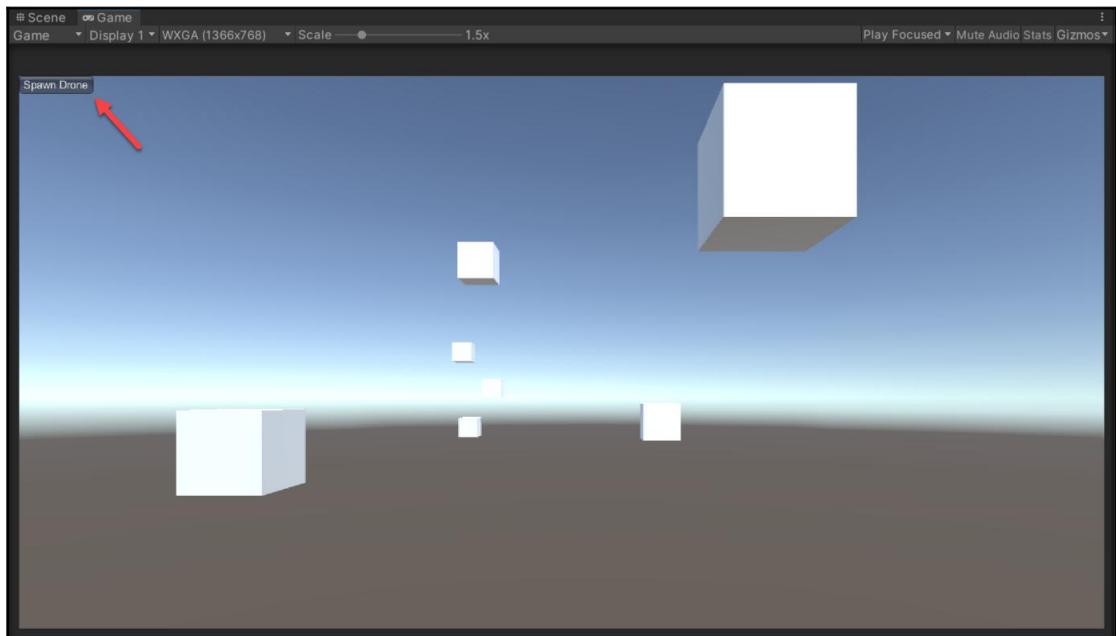


Рисунок 11.4 – Пример кода в действии внутри Unity

Если вы нажмете на ос новную кнопку с ц ены, новый куб, предс тавля ющий об ёект-дрон, должен появиться в случайной позиц ии во время выполнения с случайно выбранног о атакующег о маневра.

## Обзор реализации вражеских дронов

В предыдущем примере кода клиентский класс действует как генератор, который с случайным образом назначает с тратегию новому экземпляру дрона. Это может быть интересным подходом для реального использования. Но есть много других подходов, которые мы могли бы использовать, чтобы выбрать, какую стратегию назначить дрону. Он может быть основан на конкретных правилах и факторах, которые известны только во время выполнения. Следовательно, он не ограничивается случайностью но также может быть детерминированным и основанным на правилах.

## Расмотрение альтернативных решений

В примерах кода, представленных в этой главе, есть одна важная проблема. Мы инкапсулировали поведение атакующего маневра в отдельные классы стратегий, но каждый маневр представляет собой не что иное, как одну анимацию выполняющуюся в цикле. Итак, в реальном игровом проекте, созданном производственной командой, в которую одни аниматоры, чтобы не стал анимировать вражеских дронов в коде с помощью опорных или даже анимационного движка Tween. Вместо этого я бы попросил аниматора создать несколько подробных анимаций маневров атаки во внешнем инструменте разработки, а затем импортировать их в Unity в виде анимационных клипов. Затем я бы использовал встроенную функцию Unity и ее функцию OnceOnEnter автомата, чтобы динамически назначать дрону анимацию маневра атаки.

Используя этот подход, я бы получил качество анимации и гибкость плавного перехода от одного поведения атаки к другому, если бы решил, что дроны могут переключаться атаки при изменении внутреннего состояния. Поэтому я бы отошел от идеи инкапсулировать каждое поведение атаки в класс стратегии и вместо этого определил бы их как конечные состояния. Этот переход не будет кардинальным изменением в дизайне, поскольку концепции, лежащие в основе шаблонов FSM, State и Strategy, тесно связанны между собой.

Несмотря на то, что реализация шаблона «Стратегия» в примере, представленном в этой главе, является допустимой, разумно сначала рассмотреть, что можно достичь с помощью системы анимации Unity, когда вы управляете набором анимаций объекта. Но представьте себе другой вариант использования, в котором нам нужно реализовать варианты алгоритма обнаружения движения и назначить их дрону во время выполнения. В этом контексте шаблон «Стратегия» станет отличным выбором для построения такой системы.



Вы можете прочитать официальную документацию по теме анимации Unity по адресу <https://docs.unity3d.com/2021.2/Documentation/Manual/AnimationOverview.html>.

### Краткое содержание

В этой главе мы использовали шаблон «Стратегия» для реализации первого вражеского игрока противника нашей игры — летающего дрона, с треляющим огнем. Используя этот шаблон, мы инкапсулировали каждый вариант атакующих маневров дрона в отдельные классы. Такой подход облегчил поддержку нашего кода, поскольку позволил избежать раздущих классов, заполненных длинными условными операторами. Теперь мы можем просто написать новые варианты атакующего маневра или корректировать существующие. Таким образом, мы дали себе возможность проявлять творческий подход и использовать новые идеи, чтобы вляется жизненно важной частью разработки игр.

В следующей главе мы начнем работать над темой оружия и используем паттерн «Декоратор».

# 12

## Использование декоратора для внедрить систему вооружения

В этой главе мы собираемся создать настраиваемую систему оружия. На протяжении всей игры игрок может улучшать свое новое оружие с помощью мотоцикла, приобретая приспособления, которые улучшают определенные свойства, такие как дальность и силы. С новое оружие установлено в передней части мотоцикла и имеет два логотипа ширения для навесного оборудования, которое игрок может использовать для создания различных комбинаций. Для построения этой системы мы будем использовать паттерн Декоратор. Это не должно вызывать удивления, поскольку его название подразумевает его использование, как мы увидим далее в этой главе.

В этой главе будут рассмотрены следующие темы:

- Основные принципы паттерна Декоратор
- Реализация системы вооружения с навесным оборудованием

### Технические требования

Вам потребуется базовое понимание Unity и C#.

Мы будем использовать следующий движок Unity и концепции языка C#:

- Конструкторы
- ScriptableObjects

Если вы не знакомы с этими концепциями, просмотрите главу 3 «Краткое руководство по программированию Unity».

Файлы кода для этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/>

главная /Активы/Главы/Глава12.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/3r9rvJD>.



Мы часто используем ScriptableObjects в примерах кода в этой книге, поскольку хотим организовать для наших дизайнеров рабочий процесс, позволяющий создавать новые присоединения для оружия или настраивать существующие без изменения ни одной строки кода. Хорошей практикой является делать ваши системы, ингредиенты и механизмы простыми в настройке для непрограммистов.

## Понимание шаблона Декоратор

Короче говоря, Декоратор — это шаблон, который позволяет добавлять новые функции к существующему объекту, не изменяя его. Это стало возможным благодаря созданию класса декоратора, который обертывает исходный класс. Используя этот механизм, легкотранспортабельный, а также отсекая новое поведение от объекта.

Давайте рассмотрим следующую диаграмму, чтобы визуализировать структуру Decorator, прежде чем углубляться в предмет:

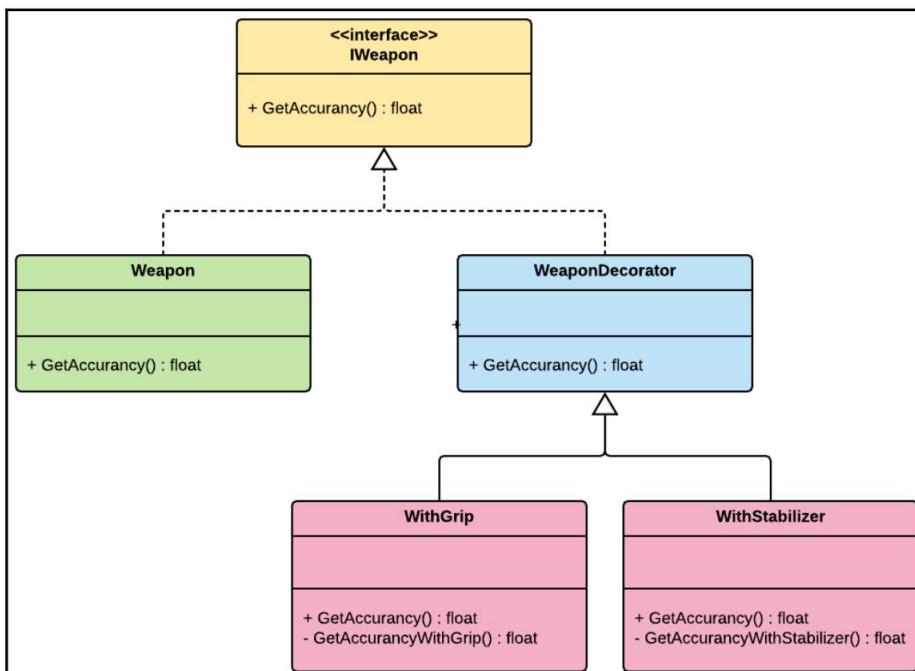


Рисунок 12.1 – UML-диаграмма шаблона Декоратор

Интерфейс IWeapon устанавливает контракт реализации, который будет поддерживать сформированную сигнатуру метода между декорированным объектом и его декораторами.

WeaponDecorator оборачивает целевой объект, а конкретные классы декораторов (WithGrip и WithStabilizer) укращают его, улучшая или определяя его поведение.

Сигнатуры методов и общая структура декорированного объекта не изменяются во время процесса, а только его поведение или значения свойств. Таким образом, мы можем легко коснуться укращения с объектами и вернуть ему первоначальный вид.

Большинство примеров этого шаблона из учебников сильно зависят от конструктора класса.

Однако базовые классы Unity API, такие как MonoBehaviour и ScriptableObject, не используют концепцию конструктора для инициализации экземпляра объекта. Вместо этого о, в случае MonoBehaviour, движок заботится об инициализации классов, привязанных к GameObjects. Ожидается, что любой код инициализации будет реализован в обратных вызовах Awake() или Start().

Поэтому перед нами стоит задача найти способ адаптировать шаблон Decorator так, чтобы он мог использовать новые функции Unity API, не теряя при этом своих новых преимуществ.

## Преимущества и недостатки шаблона «Декоратор»

Ниже приведены некоторые преимущества шаблона Декоратор:

- Альтернатива созданию подклассов: наследование — это статический процесс. В отличие от шаблона «Декоратор», он не позволяет расширять поведение существующего объекта во время выполнения. Вы можете заменить экземпляр только другим экземпляром того же родительского класса, который имеет желаемое поведение. Таким образом, шаблон «Декоратор» является более динамичной альтернативой с созданием подклассов и преодолевает ограничения наследования.
- Динамика во время выполнения: шаблон «Декоратор» позволяет нам добавлять функциональность объекту во время выполнения, присоединяя к нему декораторы. Но это также означает, что обратное возможно, и вы можете вернуть объекту его первоначальную форму, удалив его декораторы.

Ниже приведены некоторые потенциальные недостатки шаблона Декоратор:

- Сложность отношений. Отслеживание цепочки инициализации и отношений между декораторами может стать очень сложным, если вокруг объекта имеется несколько слоев декораторов.

- Сложность кода. В зависимости от того, как вы реализуете шаблон «Декоратор», он может усложнить вашу базу кода, поскольку вам может потребоваться поддерживать несколько небольших классов для декораторов. Но когда и как это становится реальным недостатком, зависит от контекста, а не от константы. В следующем примере кода, который мы рассматриваем, это не проблема, поскольку каждый декоратор представляет собой экземпляр `ScriptableObject`, который мы с огорчением как настраиваемый ресурс.



Шаблон «Декоратор» является частью семейства шаблонов с традиционным проектированием; некоторые из его членов включают шаблоны «Адаптер», «Мост», «Фасад» и «Милегес».

## Когда использовать шаблон «Декоратор»

В этой главе мы реализуем систему вооружения с надками. У нас есть несколько спецификаций, которые следует учитывать, например следующие:

- Нам нужно иметь возможность прикреплять к оружию несколько присоблений.
- Нам нужно иметь возможность добавлять и удалять их во время выполнения.

Паттерн «Декоратор» предлагает нам способ удовлетворить эти два основных требования. Таким образом, этот шаблон следует учитывать при реализации системы, в которой нам необходимо поддерживать возможность динамического добавления и удаления поведения отдельного объекта.

Например, если нам поручили работать над ССГ (коллекционной карточной игрой), нам, возможно, придется реализовать механизм, в котором игрок может увеличивать возможности базовой карты с помощью карт артефактов, сложенными друг на друга. Другой вариант использования — представить себе необязательно реализовать систему гаджетов, в которой игрок мог бы укашать с вооружениями для улучшения определенных характеристик.

В обоих случаях использование паттерна «Декоратор» может стать хорошей отправной точкой для создания подобных механизмов и систем. Теперь, когда у нас есть базовое представление о паттерне «Декоратор», перед написанием кода мы рассмотрим конструкции системы крепления оружия.

## Проектирование с системы вооружения

Пусть начнем с модели велосипеда в нашей игре с нацеленностью передней пушкой, стреляющей лазерными лучами. Игрок может использовать это оружие, чтобы уничтожать препятствия, которые могут оказаться на пути, а очень опытные игроки могут стрелять полетающим дронам, катаясь на заднем сиденье. Игровок также может приобрести различные приспособления, повышающие базовые характеристики оружия мотоцикла, как показано на следующем:

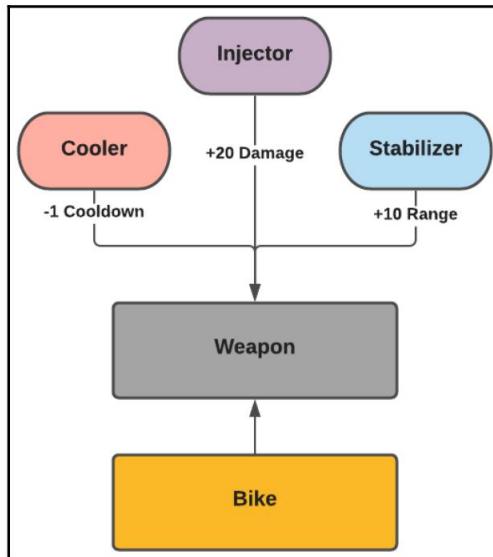


Рисунок 12.2 – Схема системы крепления оружия

Давайте рассмотрим краткий список потенциальных модулей, которые игрок может купить:

- Инжектор: плазменный инжектор, увеличивающий нанесенный оружием урон.
- Стабилизатор: приспособление, которое снижает вибрации, возникающие при движении велосипедом максимальной скорости. Он стабилизирует с трековый механизм оружия и расширяет дальность его действия.
- Охладитель: система охлаждения ног, предотвращающая перегрев оружия. Увеличивает скорость стрельбы и сокращает время восстановления.

Эти примеры представляют собой высокоразвитые концепции потенциальных вариантов привязанности. Каждая из этих модулей модифицируется или, другими словами, «украшает» определенные части оружия.

Еще одно требование, которое следует учитывать при внедрении нашей системы, заключается в том, что наше оружие должно иметь максимум двух ширин для навески ног о оборудования. Игровок должны иметь возможность динамически добавлять и удалять их настройки параметров с вогнутым автомобилем.

Теперь, когда у нас есть хороший обзор требований к проектированию нашей системы, пришло время перевести это в код, что мы и делаем в следующем разделе.

## Реализация с системы вооружения

В этом разделе мы собираемся реализовать новые классы нашей системы вооружения. Однако обратите внимание, что мы не собираемся описывать новые характеристики оружия посвящение вкратце. Мы хотим с подробностью о том, как работает шаблон «Декоратор». Но если вы хотите просмотреть более продвинутую версию этого примера кода, посмотрите папку FPP в репозитории этой книги на GitHub. Ссылку можно найти в разделе Технические требования.

## Реализация с системы вооружения

Но сначала давайте выполнить несколько шагов, поскольку нам нужно вместе просмотреть много кода:

1. Для начала мы реализуем класс BikeWeapon. Поэтому что это довольно долго, мы разделим его на три сегмента:

```
использование UnityEngine;
использование System.Collections;
```

```
после твоимен Chapter.Decorator {
```

```
    публичный класс BikeWeapon: MonoBehaviour {
        общедоступная конфигурация оружия
        WeaponConfig; общедоступное WeaponAttachment
        mainAttachment; общедоступное WeaponAttachment вторичныйAttachment;

        частный bool _isFiring; личное
        IWeapon _weapon; частный bool
        _isDecorated;

        недействительный старт()
        {
            _weapon = новое оружие (конфигурация оружия );
        }
    }
```

Первый сегмент — это код инициализации. Обратите внимание, что мы устанавливаем конфигурацию оружия в Start().

Для второго состояния класса в отладке нам помогут именные метки GUI:

```
недействительный OnGUI()
{
    GUI.цвет = Цвет.зеленый;

    GUI.Label(новый
        Rect(5, 50, 150, 100), "Диапазон: "+
        _weapon.Range);

    GUI.Label(новый
        Rect(5, 70, 150, 100), "Сила: "+
        _weapon.Strength);

    GUI.Label(новый
        Rect(5, 90, 150, 100), "Перезарядка: "+
        _weapon.Cooldown);

    GUI.Label(новый
        Rect(5, 110, 150, 100), «Скорость стрельбы:
        + _weapon.Rate);

    GUI.Label(
        новый прямой ольник (5, 130, 150, 100),
        "Стрельба из оружия : " + _isFiring);

    если (mainAttachment && _isDecorated)
        GUI.Label(
            новый Rect(5, 150, 150, 100), "Основное
            вложение: " + mainAttachment.name);

    если (вторичное присоединение && _isDecorated)
        GUI.Label(
            новый прямой ольник (5, 170, 200, 100),
            «Вторичное приложение:» + SecondaryAttachment.name);
}

Однако именно в последнем состоянии все становится интереснее:
```

```
общественная недействительность ToggleFire() {
    _isFiring = !_isFiring;

    если (_isFiring)
        StartCoroutine(FireWeapon());
}

IEnumerator FireWeapon() {
```

```

        float fireRate = 1.0f/_weapon.Rate;

        while (_isFiring) { выходит
            возвращает новый WaitForSeconds(firingRate); Debug.Log("огонь");

        }
    }

общественный
недействительный Reset () {_weapon = новое оружие
(weaponConfig); _isDecorated = !_isDecorated;
}

общественная недействительность Decorate() {
    if (mainAttachment && !secondAttachment)
        _оружие =
            новый WeaponDecorator(_weapon, mainAttachment);

    если (mainAttachment && secondAttachment)
        _weapon =
            новый WeaponDecorator(
                новый WeaponDecorator(
                    _weapon, mainAttachment),
                SecondaryAttachment);

        _isDecorated = !_isDecorated;
    }
}
}
}

```

Метод Reset() сбрасывает оружие к его первоначальным конфигурациям, инициализируя новое оружие с конфигурациими оружия по умолчанию. Это быстрый и простой способ удалить вложения, которые мы установили в методе Decorate(). Это не всегда лучший подход, но он работает в контексте данного примера.

Однако именно в Decorate() происходит волшебство и запускается механизм шаблона Decorator. Вы могли заметить, что мы можем накладывать вложения друг на друга, вызывая их в контексторе, как мы видим здесь:

```

если (mainAttachment && secondAttachment)
    _weapon =
        новый WeaponDecorator(
            новый WeaponDecorator(
                _weapon, mainAttachment),
                SecondaryAttachment);

```

Это небольшой трюк, который позволяет нам проделать Декоратор и конструкторы. Как это работает, станет более понятно, когда мы начнем реализовывать другие классы.

2. Следующий класс, который нужно реализовать, — это класс `Weapon`. Обратите внимание, что это не `MonoBehaviour`. Поэтому мы инициализируем его с помощью оконструктора:

```
последование имени Chapter.Decorator {

    Оружие публичного класса IWeapon {

        общедоступный диапазон с плавающей запятой
        {
            получить {вернуться _config.Range; }
        }

        публичная плавающая ставка
        {
            получить {вернуться _config.Rate; }
        }

        публичная плавающая сила{
            получить {вернуться _config.Strength; }
        }

        общедоступное плавающее время восстановления
        {
            получить {возврат _config.Cooldown; }
        }

        частный только для чтения WeaponConfig _config;

        публичное оружие (WeaponConfig WeaponConfig) {

            _config = конфигурация оружия;
        }
    }
}
```

В отличие от `BikeWeapon`, класс `Weapon` не реализует никакого поведения; это просто представление настраиваемых свойств оружия. Это объект, который мы будем украшать нашими. В его конструктор мы передаем экземпляр объекта `WeaponConfig`. Как мы увидим далее, это `ScriptableObject`.

3. Нам нужен общий интерфейс между классом декоратора и оружием, поэтому мы собираемся реализовать один с именем IWeapon:

```
просмотреть твоим Chapter.Decorator {

общедоступный интерфейс IWeapon
{
    Диапазон плавания {получить; }
    Float Duration {get; } Float Strength
    {получить; } Плавающая перезарядка
    {get; }
}
}
```

4. Теперь, когда у нас есть стандартный интерфейс, определяющий набор свойств, которые мы можем украсить, мы можем реализовать класс WeaponDecorator:

```
просмотреть твоим Chapter.Decorator {

публичный класс WeaponDecorator: IWeapon {

частный только для чтения IWeapon _decoratedWeapon;
частный только для чтения WeaponAttachment _attachment;

публичный оружейный декоратор
Оружие IWeapon, вложение WeaponAttachment) {

    _attachment = вложение;
    _decoratedWeapon = оружие;
}

общественная плавающая
ставка {get { return _decoratedWeapon.Rate +
_attachment.Rate; }
}

общедоступный диапазон с плавающей затяжкой {
    получить {возврат _decoratedWeapon.Range +
_attachment.Range; }
}

public float Strength { get { return
    _decoratedWeapon.Strength + _attachment.Strength; }

}

общедоступное плавающее время восстановления
{
```

```

        get { return _decoratedWeapon.Cooldown +
              _attachment.Cooldown; }
    }
}
}

```

Обратите внимание, как WeaponDecorator оборачивается вокруг экземпляра объекта, который имеет аналогичный интерфейс, в данном случае интерфейс IWeapon. Он никогда не изменяется непосредственно объект, который является его оболочкой; он лишь украшает с его общедоступные свойства с помощью WeaponAttachment.

Имейте в виду, что в нашем примере кода мы просто изменяли значения свойств оружия. Это соответствует нашей концепции, поскольку мы не изменяют новой механики оружия. Вместо этого они просто улучшают определенные свойства через слот для прикрепления.

Последняя деталь, о которой следует помнить, — это тот факт, что мы определяем поведение оружия внутри класса BikeWeapon, и его поведение определяется его свойствами, установленными в экземпляре объекта Weapon. Таким образом, украшая объект Weapon, мы изменяли его поведение в целом оружия.

- На следующем этапе мы начинаем отклонять традиционного подхода. реализация паттерна Декоратор. Вместо определения отдельных конкретных классов декораторов мы собираемся реализовать ScriptableObject с именем WeaponConfig. Такой подход позволит нам сдавать отдельные вложения и настраивать их свойства через инспектор. Затем мы будем использовать эти вложения для украшения оружия, как мы видим здесь:

использование UnityEngine;

последование Chapter.Decorator {

```

[CreateAssetMenu(fileName = «NewWeaponAttachment», MenuName =
«Weapon/Attachment», order = 1)] public class WeaponAttachment:
ScriptableObject, IWeapon {

    [Диапазон(0, 50)]
    [Tooltip("Увеличить скорость трельности в секунду")]
    [SerializeField] общедоступная плавающая ставка;

    [Диапазон(0, 50)]
    [Tooltip("Увеличить дальность оружия")]
    [SerializeField] диапазон плавающих значений;

    [Диапазон(0, 100)]
}

```

```
[Tooltip("Увеличить силу оружия")]
[SerializeField] общедоступная сила плавающего значения;

[Диапазон(0, -5)]
[Tooltip("Уменьшить время восстановления")]
[SerializeField] общедоступное плавающее время восстановления;

общедоступная строка AttachName;
общедоступное вложение GameObjectPrefab;
общедоступная строка AttachmentDescription;

публичная плавающая ставка
{get {ставка возврата; }
}

public float Range { get { return
    range; }
}

public float Strength { get { return
    Strength; }
}

общедоступное плавающее время восстановления {
    get { вернуть время восстановления; }
}
}

}
```

Важно отметить, что WeaponAttachment реализует интерфейс IWeapon. Это со временем даст возможность с классами WeaponDecorator и Weapon, поскольку все три из них имеют один и тот же интерфейс.

6. На последнем этапе мы собираемся реализовать класс ScriptableObject с именем WeaponConfig. Мы используем для этого создание различных конфигураций для нашего объекта «Оружие», которые затем можно настроить с помощью панели:

```
использование UnityEngine;

последовательность имен Chapter.Decorator {

    [CreateAssetMenu (fileName = «NewWeaponConfig», MenuName =
        «Weapon/Config», order = 1)] public class WeaponConfig:
        ScriptableObject, IWeapon {

        [Диапазон(0, 60)]
        [Tooltip("Скорость стрельбы в секунду")]
    }
}
```

```
[SerializeField] частная плавающая с тавка;
[Диапазон(0, 50)]
[Tooltip("Дальность оружия")]
[SerializeField] частный диапазон плавающих значений;

[Диапазон(0, 100)]
[Tooltip("Мощность оружия")]
[SerializeField] частная сила плавающей очисла;

[Диапазон(0, 5)]
[Tooltip("Продолжительность восстановления")]
[SerializeField] приватное плавающее время восстановления;

общедоступная строка WeaponName;
публичное оружие GameObjectPrefab; публичная
строка WeaponDescription;

публичная плавающая с тавка
{get {с тавка возврата; }
}

public float Range { get { return
    range; }
}

public float Strength { get { return
    Strength; }
}

общедоступное плавающее время восстановления {
    get { вернуть время восстановления; }
}
}

}
```

Теперь у нас есть все ключевые компоненты для нашей системы крепления оружия, и пришло время протестировать их на движке Unity.

## Тестирование системы вооружения

Если вы хотите протестировать только что расмотренный нами код на вашем собственном экземпляре Unity, вам необходимо выполнить следующие шаги:

1. Скопируйте все классы, которые мы только что расмотрели, в свой проект Unity.
2. Создайте пустуюцену Unity.

3. Добавьте на сцену новый GameObject.
4. Прикрепите следующий клиентский скрипт к новому GameObject:

использование UnityEngine;

```
последовательность Chapter.Decorator {
```

```
общественный класс ClientDecorator: MonoBehaviour {
```

```
личное BikeWeapon _bikeWeapon; частный
bool _isWeaponDecorated;
```

```
void Start()
```

```
{ _bikeWeapon =
    (ВелосипедОружие)
    FindObjectOfType<typeof(BikeWeapon)>;
```

```
}
```

```
недействительный OnGUI()
```

```
{
```

```
если (!_isWeaponDecorated)
```

```
    if (GUILayout.Button("Украсить оружие")) { _bikeWeapon.Decorate();}
```

```
        _isWeaponDecorated = !
```

```
        _isWeaponDecorated;
```

```
}
```

```
if (_isWeaponDecorated) if
```

```
(GUILayout.Button("Сбросить оружие")) {
```

```
    _bikeWeapon.Reset();
```

```
    _isWeaponDecorated = !_isWeaponDecorated;
```

```
}
```

```
if (GUILayout.Button("Переключить огонь"))
```

```
    _bikeWeapon.ToggleFire();
```

```
}
```

```
}
```

5. Добавьте в новый GameObject скрипт BikeWeapon, который мы реализовали в предыдущем разделе.

6. В разделе «Активы» | Создать | Пункт меню «Оружие», создайте новый конфиг и настройте его по своему усмотрению

7. В разделе «Активы» | Создать | Опция меню «Оружие», позволяющая создать несколько вариантов навесного оборудования с различными конфигурациями.

8. Теперь вы можете добавлять ресурсы WeaponConfig и WeaponAttachment в

Свойства компонента BikeWeapon в Инспекторе, как показано ниже.  
Скриншот:

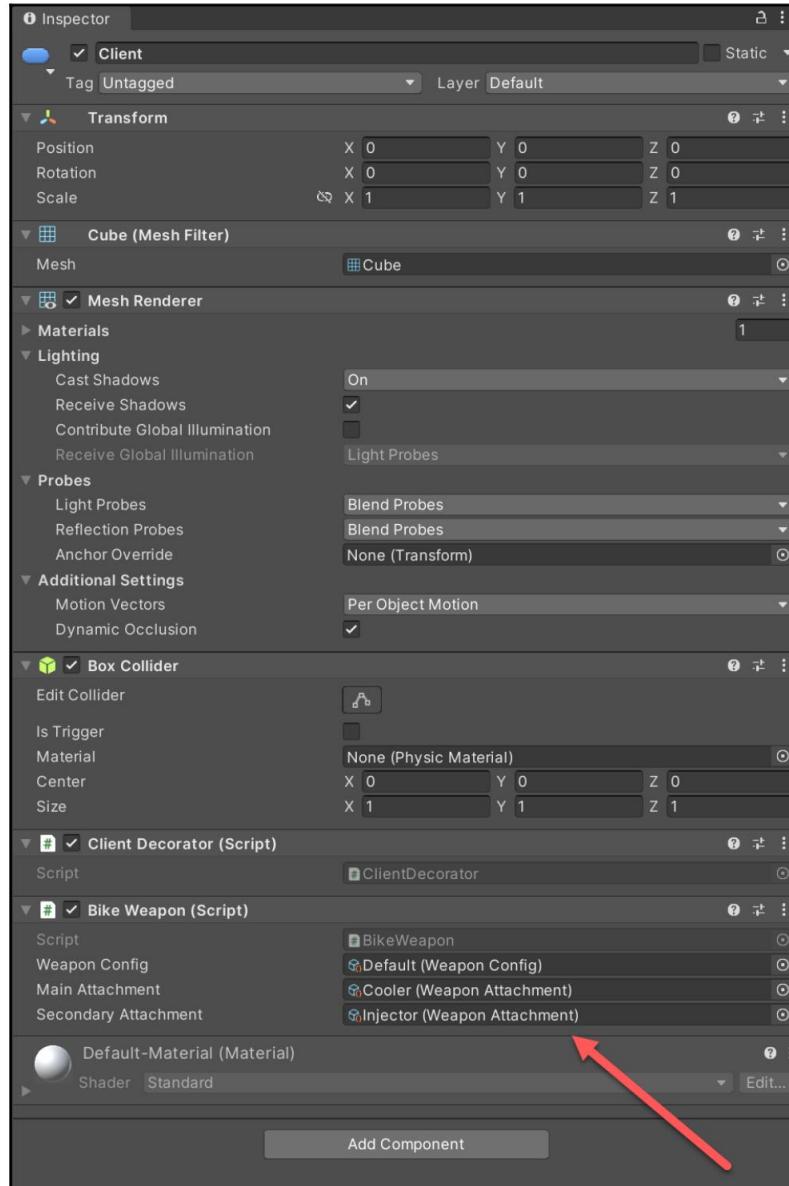


Рисунок 12.3 – Свойства компонента BikeWeapon

Когда вы запускаете сцену, вы должны увидеть следующие кнопки в верхнем левом углу экрана, как показано на следующем скриншете экрана:

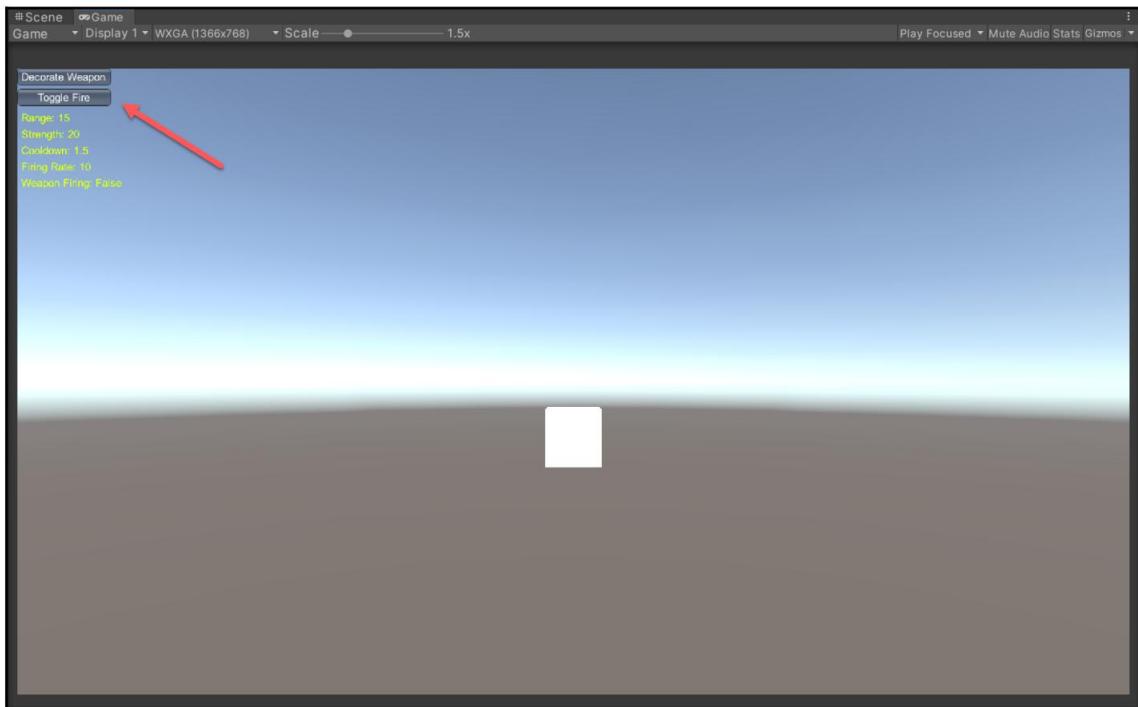


Рисунок 12.4 – Скриншот примера кода в действии внутри Unity

Теперь вы тестируете систему крепления оружия и настраиваете ее по своему желанию нажимая кнопки, и смотрите, как меняются характеристики оружия при изменении настроек декоратора.

## Обзор системы вооружения

Наша реализация Decorator не стандартна, поскольку мы смешиваем обобщенные функции языка C#, такие как конструкторы, и пытаемся использовать лучшие части Unity API. Преобразуя классы декораторов в настраиваемые ресурсы ScriptableObject, мы получаем возможность с легкостью вложить нашег оружия авторами и настраиваемыми для него раммиков. А под капотом наша система крепления построена на основе прочной конструкции.

Поэтому мы пытались найти баланс между удобством использования и ремонтопригодностью. Однако, как мы увидим в следующем разделе, есть еще способы альтернативные подходы.



Шаблоны проектирования — это рекомендации, а не заповеди. Поэтому можно экспериментировать с шаблонами и тестировать различные способы их реализации. Пример кода в этой главе является экспериментальным, поскольку не является традиционной реализацией шаблона Decorator. Но мы призываем вас, как читателя, продолжать экспериментировать с шаблонами, предложенными в этой книге, и находить новые способы их использования в Unity.

## Рассмотрение альтернативных решений

В контексте варианта использования, представленного в этой главе, мы могли бы реализовать систему оружия без шаблона Decorator и только с помощью ScriptableObjects. Мы могли бы перебрать список приобретенных модулей оружия и применить каждое из них к своим существенным классам оружия. Мы потеряли возможность ссыльяться на декораторы, но наш код станет более простым.

Основное преимущество использования шаблона «Декоратор» в нашем случае заключалось в том, что он дал нам структурированный и повторяющийся подход к реализации нашей системы. Тем не менее, в результате мы уложились в нашу кодовую базу.

### Краткое содержание

В этой главе мы реализовали систему крепления оружия, которую можно сдавать и настраивать. Несмотря на то что мы могли бы сдавать и настраивать новые положения, не написав ни единой строчки кода. Таким образом, мы можем сосредоточиться на создании системы, пока дизайнеры работают над их балансировкой. Шаблон «Декоратор» зарекомендовал себя как удобный шаблон для разработки игр, поэтому его следует держать в наборе инструментов нашего программиста.

В следующей главе мы рассмотрим шаблон пространственного разделения — предмет, который очень важно понимать при создании игр с большими картами.

## 13

## Реализация редактора уровней с пространственным разделением

В этой главе мы рассмотрим концепцию пространственного разделения. В отличие от предыдущих глав, основной предмет традиционно определяется не как шаблон проектирования программного обеспечения, а скорее как процесс и техника. Но поскольку он предлагает нам многое обратно используемый и структурированный подход к решению повторяющихся проблем проектирования игр, в контексте этой главы мы будем рассматривать его как шаблон проектирования.

Подход, о котором мы с обижаются использовать в этой главе, отличается от предыдущих глав по следующим конкретным причинам:

- Мы применяем подход неинвазивности; другими словами, мы не будем пытаться реализовать пример кода, а вместо этого рассмотрим некоторые сегменты кода.
- Мы не будем пытаться придерживаться какого-либо академического определения, а вместо этого будем использовать общую концепцию пространственного разделения для создания редактора уровней для нашей одиночной игры.

Простейшее определение пространственного разделения — это процесс, который предлагает эффективный способ размещения объектов путем расположения их в структуре данных, управляемой по их позициям. Редактор уровня, который мы реализуем в этой главе, будет построен на основе структуры данных стека, а тип объекта, который мы будем хранить в стеке в определенном порядке, — это сегмент генеральной трассы. Эти отдельные сегменты генеральной трассы будут сдаваться или удаляться в определенном порядке, в зависимости от их отношения к положению игрока на карте.

Все это может показаться очень абстрактным, но этот довольно легкий добиться с помощью интерфейса прикладного программирования (API) Unity, как мы видим в этой главе.



Система, которую мы реализуем в этой главе, слишком сложна, чтобы ее можно было свести к примеру с келетного кода. Таким образом, в отличие от предыдущих глав, предоставленный код не предназначен для воспроизведения или использования в качестве шаблона. Вместо этого мы рекомендуем просмотреть полный пример редактора уровней в папке /FPP проекта Git. Ссылку можно найти в разделе «Технические требования» этой главы.

В этой главе мы рассмотрим следующие темы:

- Понимание шаблона пространственного разделения
- Проектирование редактора уровней
- Реализация редактора уровней
- Рассмотрение альтернативных решений

## Технические требования

Мы также будем использовать следующие конкретные функции API движка Unity:

- Куча
- ScriptableObjects

Если вы не знакомы с этими концепциями, просмотрите главу 3 «Краткое руководство по программированию Unity».

Файлы кода этой главы можно найти на GitHub по следующей ссылке: [https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition /tree/main/Assets/Chapters/](https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/). Глава 13.



Стек — это линейная структура данных с двумя основными операциями: Push, которая добавляет элемент на вершину стека, и Pop, которая удаляет самый последний элемент с верху.

## Понимание шаблона пространственного разделения

Название шаблона «Пространственное разделение» происходит от процесса известного как разделение пространства, которое играет неотъемлемую роль в компьютерной графике и часто используется в реализации рендеринга с трассировкой лучей. Этот процесс используется для организации объектов в виртуальных сценах путем ранения их в структуре данных с пространственным разделением, такой как дерево разделения двоичного пространства(BSP); это укоряет выполнение геометрических запросов к большому набору трехмерных (3D) объектов. В этой главе мы будем использовать общую концепцию пространственного разделения, не придерживаясь того, как она обычно реализуется в компьютерной графике.

Очень высокий уровень и концептуальный способ визуализации пространственного разделения представлен следующей схемой:

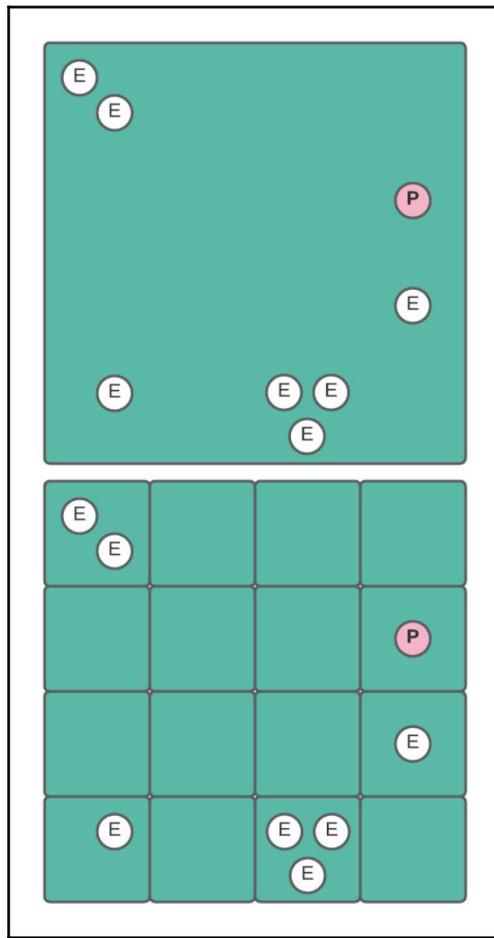


Рисунок 13.1 – Диаграмма, иллюстрирующая пространственное разделение на карте

Первый пример представляет положение врагов на карте без какого-либо разделения. Если мы хотим быстро определить расположение врагов относительно игрока, сделать это эффективно может оказаться непросто. Конечно, мы могли бы использовать приведение лучей для расчета расстояния между объектами, но это может стать неэффективным по мере роста количества врагов.

Следующая часть диаграммы показывает, что если мы разделим карту, мы теперь можем легким образом визуализировать скопления врагов относительно местоположения игрока. В коде мы теперь можем это посмотреть, какой враг находится ближе всего к игроку и где их наибольшее скопление, и поскольку нам не нужноточное положение каждого врага, а только общее географическое отношение к игроку, просто достаточно знать, в какой приблизительной ячейке сидят они нахождения.



BSP — это метод 3D-программирования, который рекурсивно разделяет пространство на выпуклые пары, используя серию иерархий костей. Метод реализован с использованием структуры данных двоичного дерева Джон Кармак, как известно, использовал BSP для разработки таких игр, как Doom и Quake.

## Когда использовать шаблон пространственного разделения

3D-программирование выходит за рамки этой книги, однако наиболее важным выводом из описания пространственного разделения является то, что оно предлагает решение для оптимальной организации большого набора объектов в системе. Поэтому, если вам нужен быстрый способ опроса обширной коллекции объектов в системе, лучше всего использовать пространственные отношения, помните о принципах пространственного разделения.

В следующем разделе мы рассмотрим общий дизайн редактора уровней и рассмотрим его отдельные требования.

## Проектирование редактора уровней

При работе в многопрофильной команде разработчиков уровня занятости игрового процесса не ограничивается реализацией крутых игровых механик и функций. Нам часто поручают создание конвейеров интеграции активов и инструментов редактирования. Самый распространенный инструмент, который нам может понадобиться в ранних этапах производственного цикла, — это пространственный редактор уровней для дизайнеров уровней в нашей команде.

Прежде чем писать одну строку кода, нам нужно помнить, что в нашей игре нет случайности, все спроектировано в ее основные игровые системы. Это игра наловка, в которой главной целью игроков является достижение вершины таблицы лидеров, запоминая тонкости каждой трассы и добираясь до финиша как можно быстрее.

Таким образом, с новываясь на этих новых принципах проектирования, мы не можем использовать такое решение, как карты процедурной генерации, которые справляются с лучайные проблемы на основе определенных правил и ограничений. Следовательно, дизайн уровня в нашей команде придется я вручную проектировать планировку каждой гоночной трассы.

И это подводит нас к самой важной задаче: в нашей игре мотоцикл движется по трехмерному миру по прямой линии на очень высоких скоростях. Если мы хотим, чтобы гонка длилась более десятка секунд, нам понадобится огромное количество ас сетов в памяти, и нашим дизайнерам придется заниматься редактированием огромных уровней в редакторе.

Этот подход неэффективен ни на этапе редактирования, ни во время выполнения. Таким образом, вместе с тем, чтобы управлять одной гонкой как единым объектом, мы разделим ее на элементы, и каждый элемент можно будет редактировать индивидуально, а затем собирая в определенном порядке, чтобы сформировать единую трассу.

Следующая диаграмма иллюстрирует эту концепцию вокруг уровня:

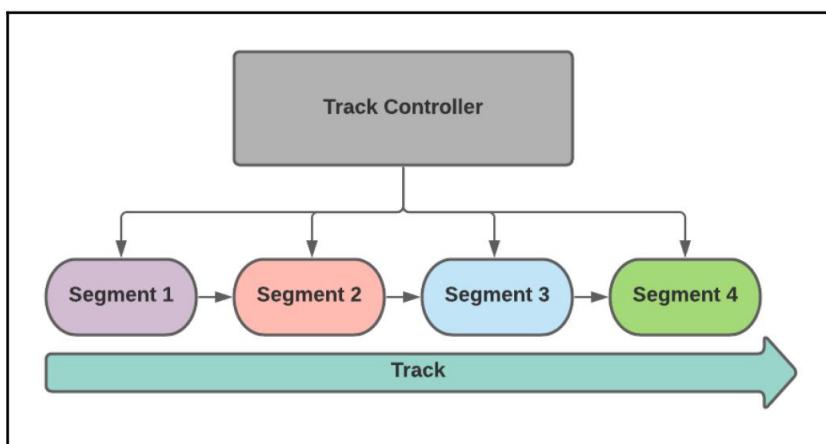


Рисунок 13.2 – Схема последовательности участков пути

Мы получаем два ключевых преимущества от этой системы:

1. Наши дизайнеры уровней могут создавать новые треки, создавая новые элементы и размещая их в различных макетах.
2. Нам не нужно загружать в память все содержимое гоночной трассы, достаточно лишь порождать нужные нам отрезки в нужный момент относительно текущего положения игрока.

Следующая диаграмма иллюстрирует, как контроллер трека использует структуру данных стека для управления тем, какой трек выгружать, а какой создавать в зависимости от текущей позиции игрока:

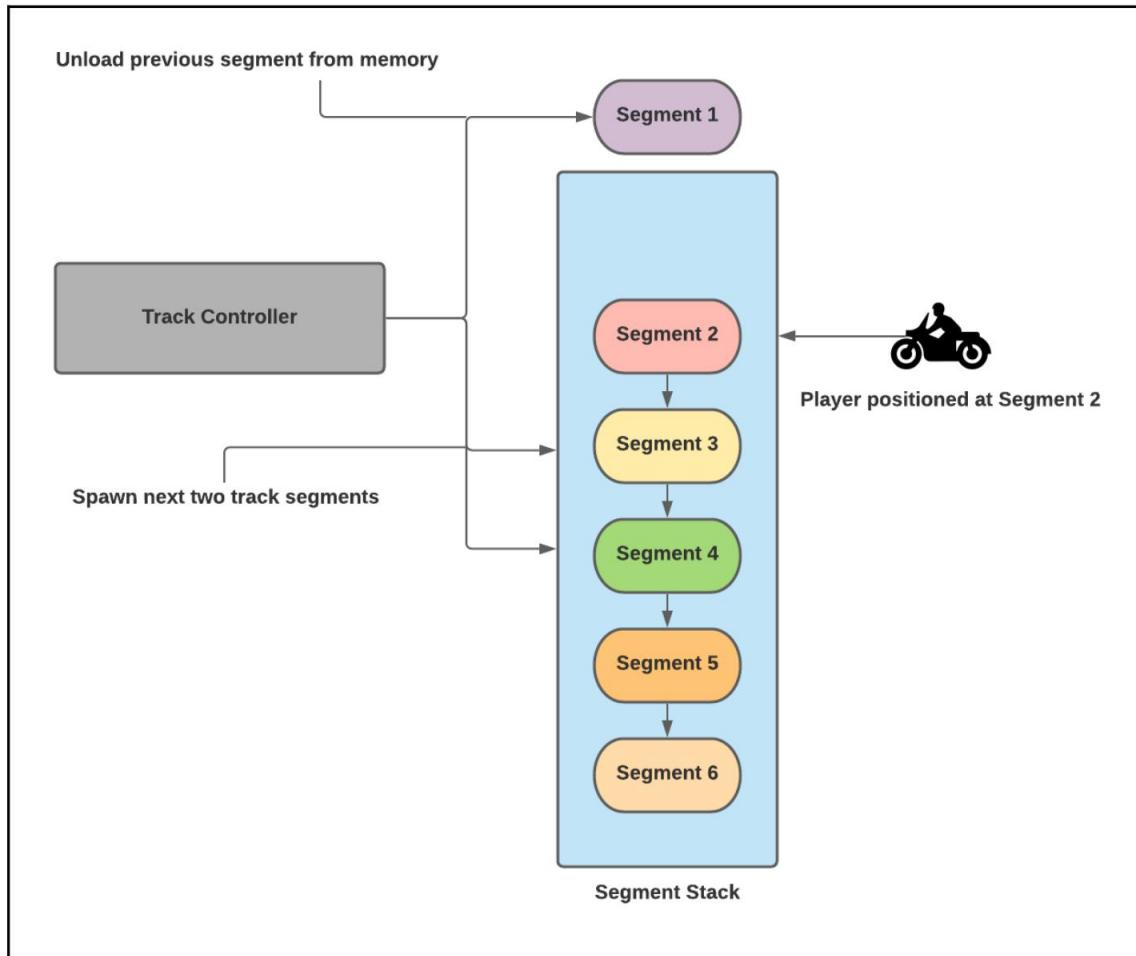


Рисунок 13.3 – Схема стека сегментов

Есть две отличительные характеристики нашей игры, которые мы должны учитывать при реализации этой системы:

1. Велосипедник не смещается с исходного положения. Это сегменты трека, которые движутся к игроку. Таким образом, ощущение скорости и движения моделируется и создает визуальную иллюзию

2. Игрок может видеть только вперед. Здесь нет окон заднего плана и камеры обзора. Это ограничение обзора камеры означает, что мы можем выгружать сегменты треков сразу после того, как они окажутся за полем зрения игрока.

Подводя итог, можно сказать, что с помощью единой системы мы решили две потенциальные ключевые проблемы нашего проекта. Прежде всего, мы создаем конвейер дизайна уровней, и, наконец, у нас есть механизм динамической загрузки наших уровней с оптимизацией, встроенной в дизайн.



При разработке головной игры, которую мы создаем в этой книге, меня вдохновили миниатюрные электрические игровые автоматы. Уникальность этой игры заключается в том, что вы можете собирать отдельные сегменты гусениц в различных конфигурациях. Иногда было веселее думать о новых и уникальных трассах, чем участвовать в гонках на игрушечных машинках.

## Реализация редактора уровней

В этом разделе мы рассмотрим код, реализующий основные компоненты нашего редактора уровней. В отличие от предыдущих глав, мы не будем пытаться сделать этот код работоспособным или тестируемым. Вместо этого мы рассмотрим реализации, чтобы понять, как мы используем общую идею пространственного разделения для создания редактора функциональных уровней для дизайнеров, одновременно оптимизируя способ загрузки уровней во время выполнения.



Код, представленный в следующем разделе, необязательно просматривать, но не компилировать, поскольку он не является полностью автономным примером.

## Шаг по реализации редактора уровней

1. Для начала мы напишем класс ScriptableObject с именем Track, как следует:

```
использование UnityEngine;
использование System.Collections.Generic;
```

```
пространство имён Chapter.SpatialPartition {
```

```
[CreateAssetMenu(fileName = «Новый трек», MenuName = «Track»)] public class Track:
ScriptableObject {
```

```
[Tooltip("Ожидаемая длина сегментов")] public float stageLength;

[Подсказка("Добавить сегменты в ожидаемом порядке загрузки")] public
List<GameObject> сегменты = новый List<GameObject>();
}

}
```

С помощью этого класса ScriptableObject наши дизайнеры уровня могут создавать новые варианты горочных трас, добавляя сегменты в список и затем упорядочивая их в определенном порядке. Каждый ресурс трека будет передан в класс TrackController, который автоматически предоставит каждый сегмент в том порядке, в котором их расположили дизайнеры.

Для игр этот процесс является плавным, поскольку он выполняется в фоновом режиме, а сегменты сдвигаются друг от друга, как они попадают в поле зрения камеры. Итак, с точки зрения игрока, это выглядит так, будто загружены все уровни.

2. Следующим идет класс TrackController. В нем мы собираемся реализовать механизм загрузки сегментов, но поскольку это обширный класс, мы собираемся разделить его на части смотреть по разделам следующим образом:

```
использование UnityEngine;
используя System.Linq;
использование System.Collections.Generic;

пространство имен Chapter.SpatialPartition {

    публичный класс TrackController: MonoBehaviour {

        частный плавающий _trackSpeed;
        частное преобразование _prevSeg;
        частный GameObject _trackParent; частное
        преобразование _segParent; частный
        список<GameObject> _segments; частный стек
        <GameObject> _segStack; частный Vector3
        _currentPosition = новый Vector3 (0, 0, 0);

        [Tooltip("Список горочных трас")]
        [SerializeField] частная
        дорожка отслеживания;

        [Tooltip("Начальное количество сегментов, загружаемых при запуске")]
        [SerializeField] частное
        int initSegAmount;

        [Tooltip("Количество дополнительных сегментов для загрузки при запуске")]
}
```

```
[SerializeField] частное
int incrSegAmount;

[Tooltip("Снизить скорость трассы")]
[Диапазон(0,0f, 100,0f)]
[SerializeField] частный
float SpeedDampener;

недействительный Пробуждение
{
    _сегменты =
        Enumerable.Reverse(track.segments).ToList();
}

недействительный Старт
{
    Инициализация();
}
```

Первый раздел предсказывает с обой прос то код инициализации и не требует поля с нений, но с ледующая часть кода становится более интересной:

```
недействительное Обновление
{
    _segParent.transform.Translate( Vector3.back *
        (_trackSpeed * Time.deltaTime));
}

частная пустота InitTrack() {
    Уничтожить(_trackParent);

    _trackParent =
        Создать экземпляр(
            Resources.Load("Track", typeof(GameObject))) как GameObject;

    если (_trackParent)
        _segParent =
            _trackParent.transform.Find("Сегменты");

    _prevSeg = ноль;

    _segStack = новый стек<GameObject>(_segments);

    LoadSegment(initSegAmount);
}
```

Как мы видим, в цикле Update() мы перемещаем родительский объект трека к игроку, чтобы имитировать движение. А в методе InitTrack() мы создаем экземпляр GameObject трека, который будет выступать в качестве контейнера для сегментов трека. Но в функции есть одна важная строка кода, которая является критически важным компонентом нашего механизма загрузки сегментов, и это проиллюстрировано здесь:

```
_segStack = новый стек<GameObject>(_segments);
```

В этой строке мы вставляем список сегментов в новый контейнер стека. Как упоминалось в начале главы, важной частью метода пространственного разделения является организация объектов в реды в структуре данных, чтобы к ним было легче запрашивать.

В следующем фрагменте кода мы увидим, как использовать структуру данных Stack для загрузки сегментов в правильном порядке:

```
частный недействительный LoadSegment (количество интервалов)
{
    for (int я = 0; я < сумма; я++) {

        если (_segStack.Count > 0) {

            Сегмент GameObject =
                Создать экземпляр (),
                _segStack.Pop(), _segParent.transform);

            если (!_prevSeg)
                _currentPosition.z = 0;

            если (_prevSeg)
                _currentPosition.z =
                    _prevSeg.position.z
                    +
                    track.segmentLength;

            сегмент.трансформ.позиция = _currentPosition;

            сегмент.AddComponent<Сегмент>();

            сегмент.GetComponent<Сегмент>().trackController
                = это;

            _prevSeg = сегмент.трансформ;
        }
    }
}
```

```
общественный недостоверный LoadNextSegment ()
{
    LoadSegment(incrSegAmount);
}
}
```

Приватный метод LoadSegment () лежит в основе системы. Он принимает в качестве параметра определенное количество сегментов. Это значение будет определять количество сегментов, которые будут загружены при вызове. Если в тексте осталось достаточно сегментов, он извлекает один с верху и инициализирует его после ранее загруженного сегмента. Он продолжает этот циклический процесс до тех пор, пока не загрузит ожидаемый количества.

Вы можете спросить себя: как нам уничтожить сегменты, прошедшие за一段时间?

Есть множество способов вычислить или определить, находится ли один объект позади другого, но для нашего контекста мы с обираем использовать двойное решение. Каждый префаб сегмента имеет объект, называемый маркером сегмента, загруженный на его край; он состоит из двух опор и невидимого сквозного крючка.

Как только велосипед проходит через тригер, маркер сегмента удаляется родительский GameObject, как мы видим здесь:

```
использование UnityEngine;

публичный класс SegmentMarker: MonoBehaviour {

    частный void OnTriggerExit (другой коллайдер) {

        если (other.GetComponent<BikeController>())
            Уничтожить (transform.parent.gameObject);
    }
}
```

Когда связь с компонентом BikeController выходит из триггера маркера сегмента, она запрашивает удаление с веб-родителя GameObject, который в данном случае будет соответствовать Segment.

Когда связь с компонентом BikeController выходит из триггера маркера сегмента, она запрашивает удаление с веб-родителя GameObject, который в этом контексте будет соответствием Segment.

Как видно из метода LoadSegment() класса TrackController, каждый раз, когда мы извлекаем новый сегмент из вершины стека, мы прикрепляем к нему скрипту в качестве компонента с именем Сегмент, как показано здесь:

```
сегмент.трансформ.позиция = _currentPosition;
сегмент.AddComponent<Сегмент>();
сегмент.GetComponent<Сегмент>().trackController = это;
```

Поскольку мы передаем текущий экземпляр класса TrackController в его параметр trackController, объект Segment может выполнить обратный вызов

TrackController и запросить загрузку следующей последовательности сегментов непосредственно перед ее уничтожением, как мы видим здесь:

использование UnityEngine;

```
Сегмент публичного класса MonoBehaviour {

    общедоступный TrackController trackController; частная
    пустота OnDestroy () {

        если (трекконтроллер)
            trackController.LoadNextSegments();
    }
}
```

Этот подход создает циклический механизм, который автоматически загружает и выгружает контролируемое количество сегментов через определенные промежутки времени. При таком подходе мы управляем количеством порожденных объектов на сцене в данный момент времени. Теоретически это приведет к более стабильной частоте кадров.

Еще одним преимуществом этого подхода, который больше связан с игровым процессом, является то, что маркеры сегментов могут выступать в качестве ориентиров для системы контрольных точек. Контрольные точки часто используются в режимах гоночных игр с ограничением по времени, в которых игрок должен достичь нескольких точек на трассе за определенный период времени.

Отличным примером гоночной игры с контрольными точками является Rad Racer 1987 года.



## Использование редактора уровней

Вы можете поговорить с редактором уровней, открыв папку /FPP в репозитории Git и выполнив следующие действия:

- В папке /Scenes/Gyms вы должны найти сцену с именем Segment. В этой сцене вы сможете редактировать и создавать новые префабы сегментов.
- В меню «Активы» -> «Создать» -> «Дорожка» у вас есть возможность создать новый трек. ресурсы.
- И, наконец, вы можете изменить и присоединить новые треки к классу TrackController, открыв сцену Track в папке Scenes/Main.

Не стесняйтесь улучшать код, чтобы более важно, получайте удовольствие!

## Обзор реализации редактора уровней

Реализации в этой главе представляют собой упрощенные версии кода более сложной системы, но если вы потратите время на просмотр расширенной версии редактора уровней в

/FPP проекта Git, мы увидим некоторые улучшения, например следующие:

- Сегменты. Для сегментов существует конвейер разработки, использующий ScriptableObjects.
- Объединение объектов: класс TrackController использует группу объектов для оптимизации времени загрузки отдельных сегментов.



Я не включил эти оптимизации в главу, чтобы сделать примеры кода короткими и простыми и в образовательных целях.

## Рассмотрение альтернативных решений

В реальном контексте производства если позволяет время, я бы построил редактор уровней нашей игры по-другому. Вместо этого бы разработал редактор треков сверху вниз, который позволил бы дизайнерам уровня рисовать рельсы и перетаскивать на них препятствия. Тогда дизайнеры смогут сюжетную работу в сериализованном формате.

Затем, используя принципы пространственного разделения, треки будут автоматически разделены на сегменты с помощью класса TrackController и помещены в пул объектов. Этот подход позволит автоматизировать процесс создания отдельных сегментов и одновременно оптимизировать процесс перестройки.

Следовательно, дизайнерам не придется создавать отдельные сегменты как префабы, и они смогут создавать новые треки, анализируя весь макет в редакторе.



Когда я создаю инструменты и настраиваю оконные рамы интеграции, моей конечной целью является автоматизация. Я всегда стараюсь автоматизировать всю работу, чтобы не тратить время на ручные задачи.

### Краткое содержание

В этой главе мы применили невмешательный подход с мотивами, как создать редактор базового уровня, используя общие идеи шаблона пространственного разделения. Нашей целью было быть верным стандартным определениям паттерна. Вместо этого мы используем его как отправную точку для построения нашей системы. Я советую вам потратить время на просмотр кода в папке /FPP и его рефакторинг, чтобы сделать его лучше.

В следующей главе мы рассмотрим некоторые альтернативные шаблоны, которые полезно знать, но некоторые имеют общие случаи использования. Таким образом, по сравнению с предыдущими главами, варианты использования будут иметь более широкую ферму применения без привязки к игровому механизму или системе. Первый шаблон, который мы рассмотрим, — это шаблон адаптера. Как следует из названия, мы будем использовать его для интеграции адаптера между двумя несвязанными системами.

# Раздел 3: Альтернативные шаблоны

В этом разделе книг и мы рассмотрим некоторые примечательные шаблоны, которые мы не использовали при программировании нашей игры, поскольку они предназначены для более общих случаев использования. Эти шаблоны полезно знать, и они могут помочь разработчикам Unity в решении различных архитектурных проблем.

Этот раздел состоит из следующих глав:

- Глава 14. Адаптация с системой с помощью адаптера
- Глава 15. Скрытие сложности с помощью сервисного фасада
- Глава 16. Управление зависимостями с помощью шаблона Service Locator

# 14

## Адаптаци я с ис тем с помошью

### Адаптер

В мире, полном различных типов кабелей и вилок, мы все привыкли к концепции адаптеров. Шаблон «Адаптер» будет одним из тех шаблонов, которые вам будет легко понять, поскольку он идеально коррелирует с нашим реальным опытом работы с технологиями. Название шаблона адаптера прекрасно раскрывает его основную идею; он предлагает нам возможность беспрепятственно использовать старый код с новым кодом, добавляя интерфейс между ними, который будет действовать как адаптер.

В этой главе будут рассмотрены следующие темы:

- Понимание шаблона адаптера
- Реализация шаблона адаптера



Примеры, представленные в этой главе, представляют собой скелетный код. Он упрощен для целей обучения, чтобы мы могли сосредоточиться на структуре шаблона. Возможно, он недостаточно оптимизирован или контекстualизирован для использования в вашем проекте в том виде, в котором он есть.

## Технические требования

Следующая глава имеет практический характер, поэтому вам необязательно иметь базовое представление о Unity и C#.

Файлы кода этой главы можно найти на GitHub: <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter14>.

Посмотрите следующее видео, чтобы увидеть Кодекс в действии:  
<https://bit.ly/3wBNqkX>

## Понимание шаблона адаптера

Как следует из названия, шаблон адаптера адаптирует два несовместимых интерфейса; как и адаптер, он не изменяет то, что настраивает, а соединяет один интерфейс с другим. Этот подход может быть полезен при работе с устаревшим кодом, который невозможно выполнить рефакторинг из-за его риска, или когда вам нужно добавить функции в стороннюю библиотеку, но вы не хотите ее изменять, чтобы избежать проблем при ее обновлении.

Вот краткий обзор двух основных подходов к реализации шаблона адаптера:

- Адаптер объекта: в этой версии шаблона используется композиция объекта, а адаптер действует как оболочка адаптированного объекта. Полезно, если у нас есть класс, у которого нет необязательных методов, но мы не можем изменить его напрямую. Адаптер объекта принимает методы исходного класса и адаптирует их к тому, что нам нужно.
- Адаптер класса: в этой версии шаблона адаптера используется наследование для адаптера и интерфейса существующего класса как интерфейс другого. Это полезно, когда нам нужно настроить класс, чтобы он мог работать с другими, но не мог изменять его напрямую.

В нашем примере кода мы будем использовать модифицированный адаптер класса, поскольку ее сложнее изучить. Если мы понимаем версию класса адаптера класса, мы можем экспериментировать с пониманием версии адаптера объекта.

Давайте посмотрим на диаграмму адаптеров объектов и классов; ядро различия незначительны, но с ходом очевидно:

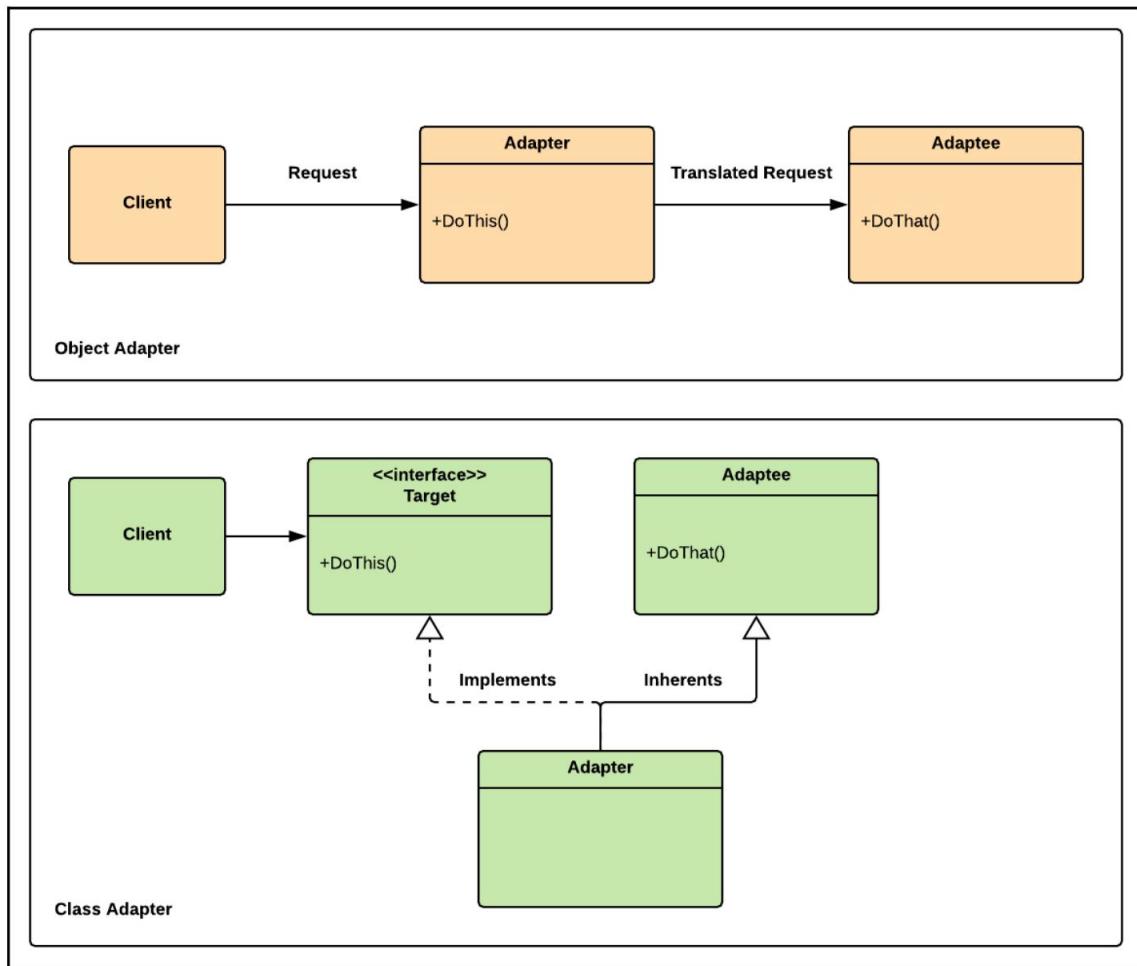


Рисунок 14.1 – Схемы системы повторов

Как видите, в обоих случаях класс Adapter располагается между Клиентом и адаптированной сущностью (Адаптируемый). Но адAPTER класса устанавливает связь с Адаптацией через наследование. Напротив, адAPTER объекта использует композицию для переноса объекта. Это пример Адаптируемого для адаптации.

В обоих случаях объект, который мы адаптируем, не изменяется. А клиент не в курсе, что мы адаптируем; он просто знает, что имеет сօгласованный интерфейс для связи с адаптированным объектом.

Композиция и наследование — это способы определения отношений между объектами; они описывают, как они относятся друг к другу. И эта разница в структуре отношений частично определяет различия между адаптерами объектов и классов.

Еще один момент, на который следует обратить внимание, заключается в том, что адаптер иногда можно спутать с шаблоном фасада. Мы должны понимать, что основное различие между ними заключается в том, что шаблон «Фасад» устанавливает упрощенный интерфейс для сложной системы. Но адаптер адаптирует несогласимые системы, сюда ранняя при этом сօгласованный интерфейс для клиента.

Оба шаблона связаны между собой, поскольку оба являются структурными шаблонами, но имеют совершенно разные цели.



Композиция — одна из основных концепций объектно-ориентированного программирования (ООП); это относится к концепции объединения простых типов для создания более сложных. Например, у мотоцикла есть колеса, двигатель и руль. Таким образом, композиция устанавливает отношение «имеет», а наследование устанавливает отношение «есть».

## Преимущества и недостатки шаблона адаптера

Ниже приведены некоторые преимущества шаблона адаптера:

- Адаптация без изменения. Основное преимущество шаблона адаптера заключается в том, что он предлагает стандартный подход к адаптации с тарифом или стороннего кода без его изменения.
- Возможность повторного использования и гибкость. Этот шаблон позволяет продолжать использовать устаревший код в новых системах с минимальными изменениями; это имеет немедленный возврат инвестиций.

Ниже приведены некоторые потенциальные недостатки шаблона адаптера:

- Сохранение устаревшего кода. Возможность использовать устаревший код с новыми системами экономически эффективна, но в долгосрочной перспективе это может стать проблемой, поскольку старый код может ограничить ваши возможности обновления, поскольку он устаревает и становится несогласимым с новыми версиями Unity или с сторонними библиотеками.
- Небольшие издержки: поскольку в некоторых случаях вы перенаправляете вызовы между объектами, это может привести к небольшому снижению производительности, обычно с слишком незначительным, чтобы стать проблемой.



Адаптер включает в себя частью свойства структурных шаблонов, включая фасад, Композитный, Присоединенный и Декоратор.

## Когда использовать шаблон адаптера

Потенциальный вариант использования адаптера в Unity — это когда у вас есть сторонняя библиотека, которую вы можете скачать из Unity Asset Store, и вам необходимо изменить некоторые из ее основных классов и интерфейсов для добавления новых функций. Но при изменении стороннего кода вы рискуете столкнуться с проблемами слияния каждый раз, когда получаете обновление от владельцев библиотеки.

Следовательно, вы попадаете в ситуацию, когда решаете подождать, пока владельцы сторонних библиотек не будут вносить изменения или изменять свой код и добавлять недостающие функции. Оба варианта имеют свои риски и выгоды. Но шаблон Адаптер дает нам решение этой дилеммы, позволяя разместить адаптер между существующими классами, чтобы они могли работать вместе, не изменяя их напрямую.

Давайте представим, что мы работаем над кодовой базой проекта, в котором используется пакет системы инвентаризации, загруженный из Unity Asset Store. Система превосходит одну; она со временем купленные или подаренные игроком предметы инвентаря в защищенной облачной серверной службе. Но есть одна проблема: он не поддерживает lokale ранение на локальном диске. Это ограничение стало проблемой в нашем проекте, поскольку в целях резервирования нам нужны как локальные, так и облачные lokale ранения.

В этом сценарии мы могли бы легко изменить код поставщика и добавить нужную нам функцию. Но когда они выпустят следующее обновление, нам придется объединить наши коды. Этот подход может быть подтвержден ошибкой. Вместо этого мы будем использовать шаблон адаптера и реализовать оболочку, которая будет поддерживать оба существующих интерфейса с системой инвентаризации, добавляя при этом поддержку локального storage. И при этом нам не придется модифицировать ни один из существующих классов. Таким образом, мы сможем избежать изменения кода поставщика и по-прежнему использовать вашу локальную систему storage для lokale ранения предметов инвентаря.

Мы реализуем пример этого варианта использования в следующем разделе. В заключение отметим, что шаблон «Адаптер» удобен в ситуациях, когда у вас есть несколько систем, которым необходимо взаимодействовать друг с другом. Тем не менее, вы должны избежать изменения существующего кода.

## Реализация шаблона адаптера

Пример кода будет простым; мы не будем пытаться написать всю систему с нуля на локальном диске, поскольку это не является целью данной главы. Вместо этого мы напишем скелет системы, чтобы сосредоточиться на использовании шаблона адаптера и не отвлекаться на несвязанные детали реализации.

## Реализация шаблона адаптера

Для начала мы реализуем класс-заполнитель, который будет имитировать стороннюю систему инвентаризации, как предстартено в цепочке предыдущего раздела:

- Класс с `InventorySystem`, предоставленный нашим вымышленным поставщиком, имеет три метода: `AddItem()`, `RemoveItem()` и `GetInventory()`. Все эти методы жестко запрограммированы на использование облачного хранилища, и мы не можем их изменить:
 

```
использование UnityEngine;
использование System.Collections.Generic;

пространство имен Chapter.Adapter {

общественный класс с InventorySystem {

    public void AddItem (пункт InventoryItem) {

        Debug.Log("Добавление элемента в облако");
    }

    public void RemoveItem (элемент InventoryItem) {

        Debug.Log("Удаление элемента из облака");
    }

    общественный список<InventoryItem> GetInventory() {

        Debug.Log("Возврат списка инвентаря, ранее лежащего в облаке");

        вернуть новый List<InventoryItem>();
    }
}
}
```

2. Следующий класс — это наш адаптер в этом сценарии. Он добавляет возможность синхронизировать локальный предметы инвентаря на облачный диск. Но он также предоставляет новую функциональность, которая позволяет объединять и синхронизировать локальный и облачный инвентарь и разные системы.

```

использование UnityEngine;
использование System.Collections.Generic;

последовательно имен Chapter.Adapter
{ публичный класс InventorySystemAdapter:
    InventorySystem, IInventorySystem {

        частный список<InventoryItem> _cloudInventory;

        общий статический недействительный
        SyncInventories () {var _cloudInventory = GetInventory ();

            Debug.Log("Синхронизация локального диска и облачных ресурсов");
        }

        общая статическая недействительность AddItem(
            Элемент InventoryItem, местоположение SaveLocation) {

            если (местоположение == SaveLocation.Cloud)
                ДобавитьItem (элемент);

            если (местоположение == SaveLocation.Local)
                Debug.Log("Добавление объекта на локальный диск");

            если (местоположение == SaveLocation.Both)

                Debug.Log("Добавление элемента на локальный диск и в облако");
        }

        общая статическая недействительность RemoveItem(
            Элемент InventoryItem, местоположение SaveLocation) {

            Debug.Log( "Удалить элемент из локального/облачного/обоих");
        }

        общий статический список<InventoryItem>
        GetInventory (местоположение SaveLocation) {

            Debug.Log("Получить инвентарь из локального/облачного/обоих");

            вернуть новый List<InventoryItem>();
}
}

```

```

    }
}
}
}
```

Обратите внимание: наследуя сторонний класс `InventorySystem`, мы получаем доступ к его функциям и методам. Таким образом, мы можем продолжать использовать его новые функции, добавляя при этом свои собственные. В процессе мы ничего не меняем, просто адаптируем.

3. Мы собираемся предоставить интерфейс нашей новой системе инвентаризации:

```

использование System.Collections.Generic;

протрансформировать имен Chapter.Adapter {

общедоступный интерфейс IInventorySystem {

недействительными SyncInventories();

недействительный AddItem(
    элемент InventoryItem, место положение SaveLocation);

недействительный RemoveItem(
    элемент InventoryItem, место положение SaveLocation);

List<InventoryItem> GetInventory (место положение SaveLocation);
}
}
```

Клиент, использующий этот интерфейс, не знает, что он общается с системой, адаптированной из другой. Адаптируемый также не знает, что мы его адаптируем. Как и адаптер зарядного устройства, телефон и кабель не знает, как они подключены, только тот проходит через систему и заряжает аккумулятор.

4. Чтобы завершить нашу реализацию нам нужно добавить перечисление, которое предоставляет сокращение локации:

```

протрансформировать имен Chapter.Adapter {

глобальное перечисление SaveLocation {

Диск,
Облачо,
Оба
}
}
```

5. И на последнем этапе мы реализуем класс -заполнитель InventoryItem :

```
использование UnityEngine;

последовательно имен Chapter.Adapter {

    [CreateAssetMenu(fileName
        = «Новый элемент», MenuName = «Инвентарь»)]
    публичный класс InventoryItem: ScriptableObject {

        // Класс с заполнителем
    }
}
```

## Тестирование реализации шаблона адаптера

Чтобы протестировать нашу реализацию вашего экземпляра Unity, скопируйте все классы, которые мы только что рассмотрели, в свой проект и прикрепите следующий клиентский класс к пустому GameObject в новом Сцене единства:

```
использование UnityEngine;

последовательно имен Chapter.Adapter {

    общеслужебный класс ClientAdapter: MonoBehaviour {

        общедоступный элемент InventoryItem;
        частная InventorySystem _inventorySystem; частный
        IInventorySystem _inventorySystemAdapter;

        недействительный старт()
        {
            _inventorySystem = новая InventorySystem(); _inventorySystemAdapter
            = новый InventorySystemAdapter();
        }

        недействительный OnGUI()
        {
            if (GUILayout.Button("Добавить элемент (без адаптера)"))
                _inventorySystem.AddItem(item);

            if (GUILayout.Button("Добавить элемент (с адаптером)"))
                _inventorySystemAdapter.
                    AddItem(пункт, SaveLocation.Both);
        }
    }
}
```

```
        }  
    }  
}
```

Теперь мы можем создать новую систему инвентаризации, используя функциональность старой системы, предствленной третьей стороной. Мы можем продолжать получать обновления библиотеки со стороны веб-сайта с уверенностью без необходимости беспокоиться о проблемах слияния. Наша система может расширяться в возможностях, пока мы продолжаем адаптировать их, если однажды мы захотим удалить их из нашей базы кода и просто использовать нашу собственную мы могли бы начать процесс устаревания, обновив класс адаптера.

## Краткое содержание

В этой главе мы добавили шаблон адаптера в наш набор инструментов. Это тот тип зора, который очень полезно иметь в заднем кармане. Одной из самых больших проблем для профессионального программиста является работа с несовместимыми системами, которые часто разрабатываются внешними поставщиками или другими командами внутри организации. Последовательный подход к адаптации существующих классов может быть только полезен, особенно когда время становится проблемой и разработчикам приходится повторно использовать старый код для новых целей.

В следующей главе мы рассмотрим близкого родственника адаптера — шаблон «Фасад», который мы будем использовать для управления распределеннойложностью кода.

# 15

## Скрытие с ложности с помощью Фасадный узор

Шаблон «Фасад» легко понять, поскольку его название подразумевает его назначение. Основная цель шаблона «Фасад» — предложить упрощенный интерфейс, который abstractирует сложную внутреннюю работу с ложной системы. Этот подход полезен для разработчиков, поскольку играет с состоянием из ложных взаимодействий между различными системами. В качестве варианта использования мы напишем код, который моделирует поведение и взаимодействие основных компонентов двигателя автомобиля, а затем предлагает простой интерфейс для взаимодействия со всей системой.

В этой главе будут рассмотрены следующие темы:

- Понимание шаблона Фасад
- Проектирование единственного двигателя
- Реализация единственного двигателя
- Базовая реализация двигателя автомобиля с паттерном Фасад



В этот раздел включена упрощенная версия реализации движка из сображений простоты и краткости. Полную реализацию этого примера кода можно найти в папке /FPP проекта GitHub — ссылка доступна в разделе «Технические требования».

### Технические требования

Это практическая глава, поэтому вам потребуется базовое понимание Unity и C#.

Файлы кода этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/>.

Главная /Активы/Главы/Глава15.

Посмотрите следующее видео, чтобы увидеть код в действии:

<https://bit.ly/36wJdxe>

## Понимание шаблона Фасад

Название шаблона «Фасад» аналогично фасаду здания — как с ледует из названия, это внешняя я поверхность, скрывающая внутреннюю структуру. В отличие от строительной архитектуры, при разработке программного обеспечения цель фасада — не украсить; вместе с этим оно упрощение. Как мы увидим на следующей диаграмме, реализация шаблона Фасад обычно ограничивается одним классом, который действует как упрощенный интерфейс для набора взаимодействующих подсистем:

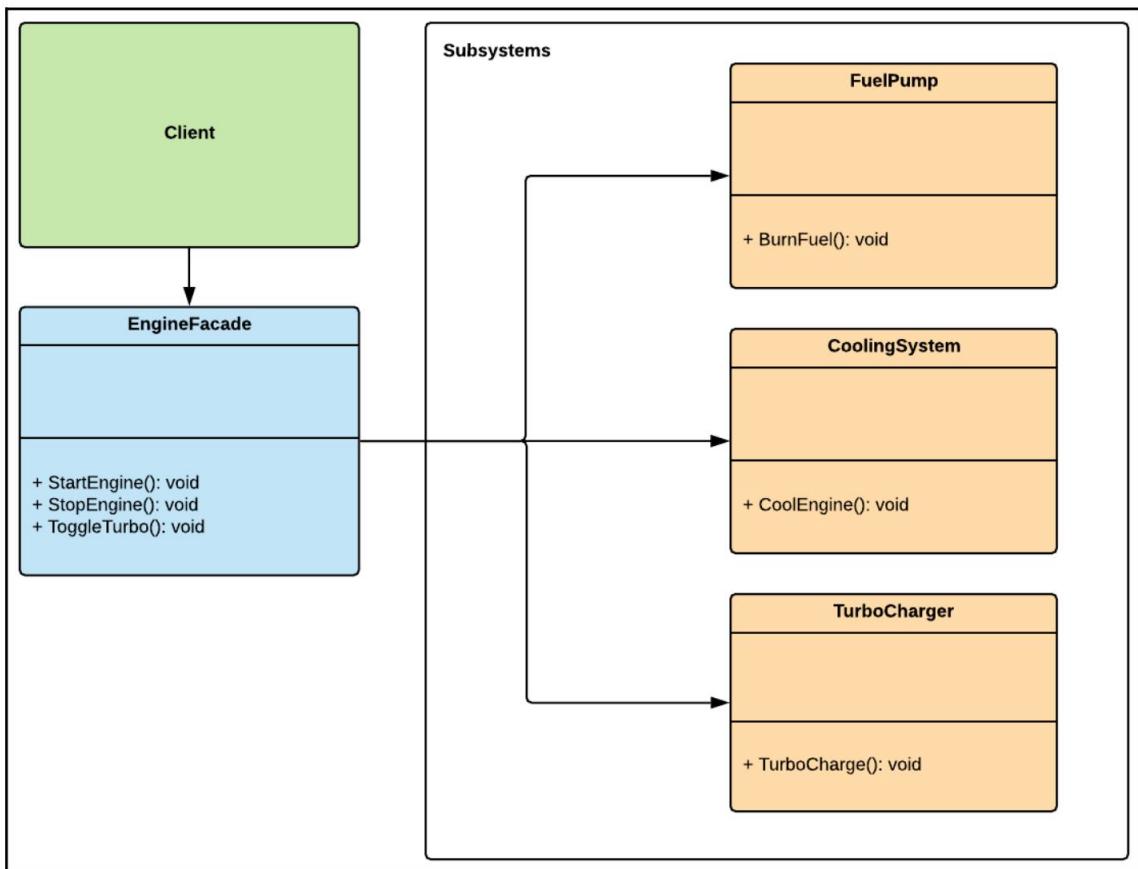


Рисунок 15.1 – Диаграмма унифицированного языка моделирования (UML) шаблона «Фасад»

Как мы видим на предыдущей диаграмме, EngineFacade действует как интерфейс для различных компонентов движка, поэтому клиент не знает, что происходит за кулисами при вызове StartEngine() в EngineFacade. Он не знает, из каких компонентов состоит двигатель и как до них добраться; он знает только то, что ему нужно знать. Это означает, что происходит, когда вы поворачиваете ключ зажигания в машине — вы не видите, что происходит под капотом, да это и не нужно; единственное, что вас беспокоит, это то, чтобы двигатель загулстался. Таким образом, шаблон «Фасад» предлагает тот же уровень абстракции в вашем коде, с ох раны детали системы под капотом.



Шаблон Фасад является частью категории структурных шаблонов. Шаблоны в этой категории инициируют взаимодействие с тем, как классы и объекты формируют более крупные структуры.

## Преимущества и недостатки

Вот некоторые преимущества шаблона Фасад:

- Упрощенный интерфейс для сложного кода. Надежный класс Facade скрывает сложность от клиента, обеспечивая при этом упрощенный интерфейс для взаимодействия с сложной системой.
- Простой рефакторинг: проще разогнать код, изолированный за фасадом, поскольку интерфейс с системой остается со связанным с клиентом, в то время как ее компоненты изменяются «закулисами».

Ниже приведены некоторые недостатки, на которые следует обратить внимание:

- Так легче скрывать беспорядок: использование шаблона «Фасад» для скрытия беспорядка за числом интерфейсов в долгосрочной перспективе ведет не к новым преимуществам шаблона, но к тому, что шаблон действительно предлагает с помощью замаскировать некоторые запахи кода, пока вы не поработаете над его рефакторингом. Однако расходуется время, чтобы что-то исправить, с момента вложения ловушкой, потому что у нас будет достаточно времени, чтобы что-то исправить, с момента вложения ловушкой, потому что у нас редко бывает достаточно времени для правильного рефакторинга.
- Слишком много фасадов. Среди разработчиков Unity популярны глобально доступные классы-менеджеры, которые действуют как фасады для основных систем; они часто реализуют их путем объединения шаблонов Singleton и Facade. К сожалению, слишком много классов менеджеров, функционирование каждого из которых зависит от другого. Как следствие, отладка, рефакторинг и модульное тестирование компонентов становятся очень трудными.



Шаблон «Фасад» устанавливает новый интерфейс, тог да как шаблон «Адаптер» адаптирует старый интерфейс. Поэтому при реализации шаблонов, которые могут упомянуться одновременно, важно помнить, что они не обязательно идентичны по своему назначению.

## Проектирование велосипедного двигателя

Мы не стремимся реализовать полную систему управления двигателем бензинового двигателя для велосипеда; это займет слишком много времени и потребует глубокого понимания физики и механики реального двигателя. Но мы будем стремиться в минимальной степени моделировать некоторые стандартные компоненты двигателя высокоскоростного транспортного средства. Итак, сначала давайте разберем ожидаемое поведение каждой части нашего движка следующим образом:

- Система лождения: Система лождения отвечает за предотвращение перегрева двигателя. При включении турбокомпрессора система лождения отключается во время турбонаддува. Такое поведение означает, что если игрок злоупотребляет турбокомпрессором, это может привести к перегреву двигателя, в результате чего двигатель остановится или взорвется, а моторикл перестанет двигаться.
- Топливный насос: этот компонент отвечает за контроль расхода топлива моторика. Он знает количество оставшегося бензина и устанавливает двигатель, если он заканчивается.
- Турбокомпрессор: если турбокомпрессор активирован, максимальная скорость автомобиля увеличивается, но система лождения временно отключается, чтобы с ходу двигатель мог передать мощность на трансмиссию.

В следующем разделе мы реализуем скелетный класс для каждого из этих компонентов и создадим класс фасада механизма, чтобы клиенты могли запускать и управлять механизмом.



Замысел конструкции, заключающейся в отключении системы лождения при включении турбокомпрессора, состоит в том, чтобы создать ощущение риска, а не вознаграждения. Игрок должен сбалансировать желание двигаться быстрее с потенциальными последствиями перегрева двигателя.

## Реализация велосипедного двигателя

Как мы видим, шаблон «Фасад» прост, поэтому следующий пример кода будет простым и понятным. Для начала мы напишем классы для каждого из основных компонентов, состояния которых двигатель мотоцикла, следующим образом:

- Начнем с бензонасоса; Целью этого компонента является имитация расхода топлива, чтобы он знал оставшееся количество и отключал двигатель, когда оно заканчивается. Вот код, который вам понадобится:

```
ис пользование UnityEngine;
ис пользование System.Collections;

пространство имен Chapter.Facade {

общественный класс FuelPump: MonoBehaviour {
    общедоступный движок BikeEngine;
    общедоступный IEnumerator burnFuel;

недействительный Старт
{
    сжечьтоТопливо = сжечьтоТопливо;
}

IEnumerator BurnFuel() {
    пока (правда)
    {
        дох односТЬ возврата новых WaitForSeconds(1);
        engine.fuelAmount -= engine.burnRate;

        если (engine.fuelAmount <= 0.0f) { engine.TurnOff();
            дох односТЬ, дох односТЬ
            0;
        }
    }
}

недействительный OnGUI()
{
    GUI цвет = Цвет.зеленый; GUI.Label(
        новый Rect(100, 40, 500, 20), "Fuel: " +
        engine.fuelAmount);
}
}
```

```

        }
    }
}
```

2. Далее идет система лождения, отвечающая за предвращение вращения двигателя. от перегрева, но отключается при включении турбокомпрессора. Код проиллюстрирован в следующем фрагменте:

```

ис пользование UnityEngine;
ис пользование System.Collections;

просмотр твоим именем Chapter.Facade
{общественный класс CoolingSystem: MonoBehaviour {общественный
двигатель BikeEngine; общественный
IEnumerator CoolEngine; частный bool
_isPaused;

недействительный
Start () { CoolEngine = CoolEngine ();
}

общественный недействительный
PauseCooling () { _isPaused = !_isPaused;
}

общественный недействительный
ResetTemperature () {engine.currentTemp = 0.0f;
}

IEnumerator CoolEngine() {
    в то время как (истина) {
        дох одноть возврата новых WaitForSeconds (1);

        if (_isPaused) { if
            (engine.currentTemp > engine.minTemp)
                engine.currentTemp -= engine.tempRate;
            если (engine.currentTemp < engine.minTemp)
                engine.currentTemp += engine.tempRate; } Ещё

        { engine.currentTemp += engine.tempRate;
        }

        если (engine.currentTemp > engine.maxTemp)
            двигатель.Выключить();
    }
}

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

```

        GUI.Label(
            новый Rect(100, 20, 500, 20),
            engine.currentTemp); "Температура: "
        }
    }
}

```

3. И, наконец, турбокомпрессор при активации увеличивает максимальную скорость мотоцикла, но для его работы необходимо временно отключить систему охлаждения. Вот код для этого:

```

использование UnityEngine;
использование System.Collections;

последовательность Chapter.Facade {

    публичный класс TurboCharger: MonoBehaviour {

        общедоступный движок BikeEngine;
        частный bool _isTurboOn; частная
        система охлаждения _coolingSystem;

        public void ToggleTurbo(CoolingSystem CoolingSystem) {

            _coolingSystem = система охлаждения ; если (!
            _isTurboOn)
                StartCoroutine(TurboCharge());
        }

        IEnumerator TurboCharge() {

            _isTurboOn = правда;
            _coolingSystem.PauseCooling();

            пока односТЬ return new WaitForSeconds(engine.turboDuration);

            _isTurboOn = ложь;
            _coolingSystem.PauseCooling();
        }

        недействительный OnGUI()
        {
            GUI цвет = Цвет.зеленый; GUI.Label(
                новый прямоугольник (100, 60, 500, 20),
                «Турбо активирован: » + _isTurboOn);
        }
    }
}

```

```

    }
}

```

4. Теперь, когда у нас готовы основные компоненты нашего двигателя, нам нужно реализовать класс, который позволит клиенту беспрепятственно взаимодействовать с ними. Итак, мы реализуем фасадный класс под названием BikeEngine, который будет предлагать клиенту интерфейс для запуска и остановки двигателя, а также переключения турбокомпрессора. Но поскольку следующий пример кода очень длинный, мы рассмотрим его в двух частях. Вот первая часть:

```

использование UnityEngine;

последовательно имен Chapter.Facade {

    публичный класс с BikeEngine: MonoBehaviour {

        общедоступное плавающее значение burnRate =
        1.0f; публичное плавающее топливоAmount = 100.0f;
        общедоступное плавающее значение tempRate =
        5.0f; общедоступное плавающее minTemp = 50.0f;
        общедоступное число плавающей запятой
        maxTemp = 65.0f; общедоступное плавающее
        значение currentTemp; общедоступное плавающее турбоDuration = 2.0f;

        частный bool _isEngineOn; частный
        FuelPump _fuelPump; частный TurboCharger
        _turboCharger; частная система охлаждения
        _coolingSystem;

        недействительный пробуждение () {
            _fuelPump =
                gameObject.AddComponent<FuelPump>(); _turboCharger =
                gameObject.AddComponent<TurboCharger>(); _coolingSystem =
                gameObject.AddComponent<CoolingSystem>();
        }

        недействительный Старт () {
            _fuelPump.engine = это;
            _turboCharger.engine = это; _coolingSystem.engine
            = это;
        }
    }
}

```

Первая часть этого класса представляет собой код инициализации и не требует поля с ней, но важной частью является следующий метод:

```
общественная недействительность TurnOn() {
    _isEngineOn = правда;
    StartCoroutine(_fuelPump.burnFuel);
    StartCoroutine(_coolingSystem.coolEngine);
}

общественная недействительность TurnOff() {
    _isEngineOn = ложь;
    _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), "Двигатель
работает: " + _isEngineOn);
}
}
```

Как мы видим, класс EngineFacade скрывает большую функциональность, которую предлагает двигатель ведомого, но в то же время скрывает взаимодействие между его компонентами. Если мы хотим запустить движок, нам просто нужно вызвать метод StartEngine(). Если бы у нас не было шаблона фасада, такого как тот, который мы только что реализовали, нам пришлось бы инициализировать каждый компонент двигателя индивидуально и знать каждый из их параметров, которые нужно установить, и методы, которые нужно вызывать. Шаблон «Фасад» позволяет нам с легкостью воспользоваться этим интерфейсом.

Но предположим, что мы хотим добавить еще один компонент двигателя, например нитрофорсунку; в этом случае нам нужно будет только изменить класс BikeFacade и предоставить новый общий доступный метод, который позволит нам запустить инжектор.

## Тестирование фасада двигателя

Мы можем быстрее протестировать только что реализованный код, добавив следующий клиентский скрипт в GameObject в пустой сцене Unity:

```
использование UnityEngine;
пространство имён Chapter.Facade {
    публичный класс ClientFacade: MonoBehaviour {
        частный BikeEngine _bikeEngine;

        недействительный Старт
        {
            _bikeEngine =
                gameObject.AddComponent<BikeEngine>();
        }

        недействительный OnGUI()
        {
            если (GUILayout.Button("Включить"))
                _bikeEngine.TurnOn();

            if (GUILayout.Button("Выключить"))
                _bikeEngine.TurnOff();

            if (GUILayout.Button("Переключить Турбо"))
                _bikeEngine.ToggleTurbo();
        }
    }
}
```

В клиентском классе мы видим, что он не знает о внутренней работе движка, и именно этого эффекта мы хотим достичь при использовании шаблона Фасада. Это единственное, что знает клиентский класс, это то, что он может запускать и останавливать двигатель, а также включать функцию турбонаддува, вызывая общедоступные методы, доступные классом BikeEngine. Другими словами, как и в реальной жизни, нам не нужно открывать капот, чтобы запустить двигатель; мы поворачиваем ключ зажигания, и компоненты начинают работать вместе, а мы не знаем, как они взаимодействуют друг с другом.

В следующем разделе мы рассмотрим альтернативные решения, которые следуют за рассмотреть, прежде чем принять решение об использовании шаблона Фасад.



В потенциально более продвинутой версии этого примера кода двигатель будет вычислять текущие обороты в минуту (об/мин), также известные как скорость двигателя, и мы могли бы подключить его к системе передач, регулируемой в одном шагом переключателя, с помощью оторога игрок мог бы контролировать скорость велосипеда.

Таким образом, мы могли легко повысить уровень реализма в любой момент.

## Рассмотрение альтернативных решений

Прежде чем рассмотреть шаблон «Фасад», следует иметь в виду не сколько альтернатив, в зависимости от того, что оно вам на самом деле пытаешься достичь. Они перечислены здесь:

- Шаблон «Абстрактная фабрика». Если вы хотите скрыть только способ обнаружения объектов подсистемы из клиента, вам следует рассмотреть возможность использования шаблона «Абстрактная фабрика» вместо шаблона «Фасад».
- Адаптер: если вы собираетесь написать «обертку» над существующими классами с намерением соединить два несовместимых интерфейса, вам следует рассмотреть возможность использования шаблона адаптера.

## Краткое содержание

Несмотря на то, что шаблон «Фасад» иногда используется для скрытия беспорядочного кода, если вы используете его оно может улучшить читаемость и удобство использования вашей кодовой базы, маскируя лежащие в ее основе сложные взаимодействия подсистем за единственным интерфейсом. Таким образом, этот шаблон может оказаться очень полезным для программирования игр, но если вы следите использовать с умом и с добрыми намерениями.

В следующей главе мы рассмотрим шаблон под названием Service Locator, который мы будем использовать для управления глобальными зависимостями и предоставления новых сервисов.

# 16

## Управление зависимостями с помощью шаблона локатора с сервисов

Эта глава будет краткой, поскольку шаблон Service Locator, который мы рассматриваем, прост и эффективен. Основная идея этого шаблона проста: он основан на наличии центрального реестра сущностей илиализированных зависимостей. Но если быть более точным, эти зависимости представляют собой компоненты, предлагающие определенные услуги, которые мы можем предоставить с помощью интерфейсов, которые мы называем «контрактами обслуживания». Следовательно, когда клиенту необходимо вызвать конкретную службу, ему не нужно знать, как ее локализовать и инициализировать; ему просто нужно запросить шаблон локатора службы, и он сделает всю работу по выполнению контракта с службы.

Как мы увидим в этой главе, это довольно простая конструкция, и ее легко реализовать.

В этой главе мы рассматриваем следующие темы:

- Понимание шаблона локатора с сервисов
- Реализация шаблона локатора с сервисов
- Рассмотрение альтернативных решений



Мы упростили пример кода в этой главе в целях обучения, чтобы представить основные концепции шаблона, не отвлекаясь на детали реализации. Таким образом, показанный код не оптимизирован и не контекстуализирован в достаточной степени, чтобы его можно было использовать в проекте как есть.

## Технические требования

Следующая глава имеет практический характер, поэтому вам необходимо иметь базовое представление о Unity и C#.

Мы будем использовать следующие специфичные для Unity движок и концепции языка C#:

- Статика
- Джениреки

Если вы не знакомы с этими концепциями, просмотрите главу 3 «Краткое руководство по программированию Unity».

Файлы кода этой главы можно найти на GitHub по адресу <https://github.com/PacktPublishing/Game-Development-Patterns-with-Unity-2021-Second-Edition/tree/main/Assets/Chapters/Chapter16>.

Посмотрите следующее видео, чтобы увидеть код в действии: <https://bit.ly/36AKWl>



Статический — это модификатор ключевого слова. Метод, объявленный в классе как статический, можно вызвать без создания экземпляра объекта.

## Понимание шаблона локатора сервисов

Привненious более традиционными шаблонами, шаблон Service Locator имеет меньше академической теории и очень pragmatic по своей общей конструкции. Как следует из названия, его цель — найти услугу для клиента. Это достигается за счет ведения центрального реестра объектов, предлагающих определенные услуги.

Давайте рассмотрим диаграмму типичной реализации Service Locator:

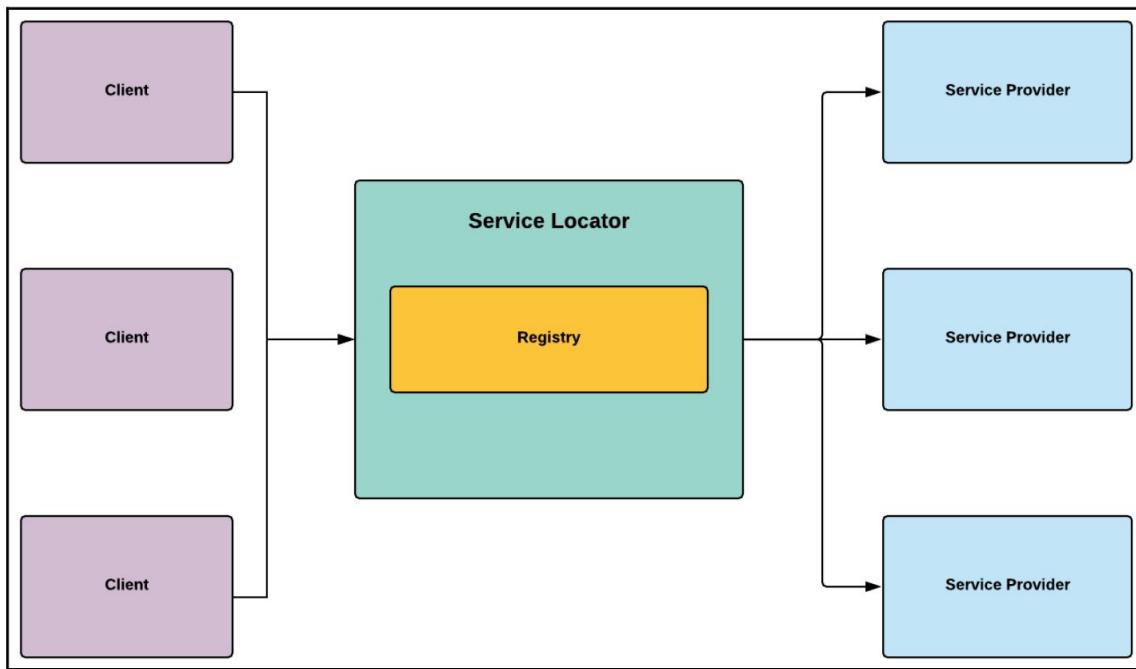


Рисунок 16.1 – Схема шаблона Service Locator

Как мы видим, мы могли бы легко сказать, что шаблон Service Locator действует как прокси между клиентами (запрашивающими) и поставщиками услуг, и этот подход в определенной степени отделяет их. Клиенту потребуется вызвать шаблон локатора с услугой только в том случае, если у него есть зависимость, которую он обладает разрешить, и ему необязательно дотягнуться до службы. Можно сказать, что шаблон «Локатор услуг» действует аналогично официальному ресторану: он принимает заказы от клиентов и выступает в качестве посредника между различными услугами, которые ресторан предлагает своим клиентам.



Шаблон Service Locator имеет плохую репутацию некоторых кругов; эксперты часто критикуют его за то, что он является антипаттерном. Основная причина этой критики заключается в том, что она нарушает несколько лучших практик кодирования, поскольку скрывает зависимости классов, а не раскрывает их. Как следствие, это может затруднить поддержку и тестирование вашего кода.

## Преимущества и недостатки шаблона Service Locator

Вот некоторые из потенциальных преимуществ использования шаблона Service Locator:

- Оптимизация во время выполнения. Шаблон Service Locator может оптимизировать приложение путем динамического обнаружения более оптимизированных библиотек или компонентов для выполнения определенного сервиса в зависимости от контекста времени выполнения.
- Простота: Service Locator — это один из наиболее простых реализаций шаблонов управления зависимостями, который не требует сложного обучения, как структура внедрения зависимостей (DI). Следовательно, вы можете быстро начать использовать его в проекте или научить коллег.

Вот некоторые недостатки использования шаблона Service Locator:

- Чёрный ящик: реестр шаблона Service Locator запутывает зависимости классов. Следовательно, некоторые проблемы могут возникнуть во время выполнения, а не во время компиляции, если зависимости отсутствуют или неправильно зарегистрированы.
- Глобальная зависимость. При чрезмерном использовании и с неправильными намерениями шаблон Service Locator сам по себе может стать глобальной зависимостью, которой будет суждено управлять. Ваш код станет чрезмерно зависеть от него, и в конечном итоге будет непросто отделить его от остальных новых компонентов.



Шаблон Service Locator был популярен среди разработчиков Java; частично оно было определено Мартином Фаулером в сообщении в блоге, опубликованном в 2004 году, которое вы можете прочитать здесь:

<https://martinfowler.com/articles/injection.html>

## Когда использовать шаблон Service Locator

Вопрос о том, когда использовать шаблон Service Locator, не требует пояснений, их ходя из его описания. Например, если у вас есть список сервисов, к которым вам необходимо добраться, но вы хотите инкапсулировать процесс их получения, то этот шаблон может предложить решение.

Но еще один аспект, который мы должны учитывать при рассмотрении использования шаблона Service Locator, — это когда его следует использовать. Поскольку шаблон локатора сервисов обычно доступен глобально, как следует из названия, он должен находиться с сервисами и обеспечивать доступ к ним. Затем мы должны использовать его только для предоставления услуг, имеющих глобальную область действия.

Например, нам нужен доступ к проекционному дисплею(HUD), чтобы обновить один из компонентов пользовательского интерфейса (UI). Должны ли мы расматривать HUD как глобальную службу, доступ к которой должен быть возможен через шаблон Service Locator? Ответ должен быть отрицательным, поскольку HUD поведется только в определенных частях игры и должен быть доступен только определенным компонентам в определенном контексте. Но если мы разработаем собственную систему журналирования, мы могли бы оправдать ее раскрытие с помощью шаблона локатора с службами, поскольку нам может потребоваться я регистрировать информацию из любого места нашего кода независимо от контекста и области действия.

Теперь, когда мы разобрались с теорией, давайте запачкаем руки и напишем шаблон Service Locator, обеспечивающий доступ к рекламатору, системе аналитики и провайдеру рекламной сети (рекламной сети).

## Реализация шаблона локатора с сервисов

Мы собираемся реализовать базовый шаблон локатора с службами для предоставления доступа к трем конкретным службам, а именно:

- Logger: с службой, которая действует как фасад централизованной системы журналирования.
- Аналитика: с службой, которая отправляет пользователюскую аналитическую информацию с первичной частью, чтобы получить представление о поведении игроков.
- Реклама: с сервисом, который извлекает видеорекламу (рекламу) из сети и отображает ее для монетизации контента игры в определенные моменты.

Мы добавляем эти службы в реестр шаблона локатора с службами из-за их следующих характеристик:

- Они предлагают конкретную услугу.
- Они должны быть доступны из любой точки базы кода.
- Их можно вызывать или удалить, не вызывая регрессии в коде игрового процесса.

Как мы увидим из следующего примера кода, реализация базового шаблона Service Locator — это простой процесс. Вот какие шаги мы предпримем:

1. Начнем с реализации самого важного ингредиента — класса ServiceLocator, например:

```
ис пользование
системы; ис пользование System.Collections.Generic;

просмотримоимен Chapter.ServiceLocator {
```

```

    публичный статический класс ServiceLocator {
        частный статический только для чтения
        IDictionary<Type, object> Services = new Dictionary<Type,
        Object>();

        public static void RegisterService<T>(T service) {
            if (!Services.ContainsKey(typeof(T))) {
                Службы[typeof(T)] = service;
            }
            еще
            {
                бросить новый
                Исключение приложения
                («Сервис уже зарегистрирован»);
            }
        }

        общественный статический T GetService<T>()
        {
            пытаюсь
            {
                return (T) Services[typeof(T)];
            }
            ловить
            {
                бросить новый
                ApplicationException
                («Запрошенная служба не найдена.»);
            }
        }
    }
}

```

Эта версия шаблона Service Locator имеет три основные особенности, изложенные следующим образом:

- Он управляет реестром услуг в форме словаря.
- Он предлагает статическую функцию именем RegisterService(), которая позволяет зарегистрировать объект в качестве службы.

- Он возвращает экземпляр службы определенного типа при запросе через функцию `getService()`.

Следует учитывать, что `RegisterService()`, и `GetService()` являются статическими функциями, поэтому они доступны напрямую без необходимости инициализации класса `ServiceLocator`.

Словарь с службами содержит список доступных служб, и мы отметили его как доступный только для чтения и частный; таким образом, мы защищаем его от переопределения или промодификации доступа. Вместо этого клиенту придется пройти через общедоступные методы, предоставляемые шаблоном `Service Locator`, чтобы добавить или получить сервис.

Теперь, когда у нас готов класс `Service Locator`, мы можем приступить к реализации некоторых сервисных контрактов в виде интерфейсов.

2. Наш первый интерфейс предназначен для службы `Logger`, как показано в следующем коде.

Фрагмент:

```
просмотр трансляции Chapter.ServiceLocator {
    общедоступный интерфейс ILoggerService {
        void Log (с строковое сообщение);
    }
}
```

3. Следующий интерфейс предназначен для службы `Analytics`, как мы видим здесь:

```
просмотр трансляции Chapter.ServiceLocator {
    общедоступный интерфейс IAnalyticsService {
        void SendEvent (с строка eventName);
    }
}
```

4. И, наконец, мы реализуем код интерфейса нашей службы рекламы, как следует:

```
просмотр трансляции Chapter.ServiceLocator {
    общедоступный интерфейс IAdvertisement {
        недействительный DisplayAd();
    }
}
```

5. А теперь мы собираемся реализовать конкретные классы сервисов, начиная с класса Logger. Код для выполнения этой задачи показан в следующем фрагменте:

```
использование UnityEngine;

последовательно имен Chapter.ServiceLocator {

    публичный класс Logger: ILogService {

        общес твенный недействительный журнал (с троекратное сообщение)
        {
            Debug.Log("Записано: " + сообщение);
        }
    }
}
```

6. Далее идет урок аналитики. Вот код, который вам понадобится для реализации этого о

```
использование UnityEngine;

последовательно имен Chapter.ServiceLocator {

    общес твенный публичный класс Analytics: IAnalyticsService {

        общес твенная недействительность SendEvent (с троекратное eventName)
        {
            Debug.Log("Отправлено: " + Имя события );
        }
    }
}
```

7. И, наконец, мы реализуем наш конкретный класс службы рекламы следующим образом:

```
использование UnityEngine;

последовательно имен Chapter.ServiceLocator {

    Реклама публичного класса IAdvertisement {

        общес твенный недействительный DisplayAd ()
        {
            Debug.Log("Показ видеорекламы");
        }
    }
}
```

Теперь у нас есть шаблон Service Locator с службами, которые мы можем зарегистрировать и получить к ним доступ из любого места.

## Тестирование шаблона Service Locator

Чтобы протестировать нашу реализацию шаблона Service Locator, давайте напишем клиентский класс, который мы прикрепим в качестве компонента к GameObject впустить сцене Unity, как показано ниже:

```
ис пользование UnityEngine;

последование твоим Chapter.ServiceLocator {

    публичный класс ClientServiceLocator: MonoBehaviour {

        недействительный Старт () {
            Регистрация Услуг ();
        }

        частная пустота RegisterServices() {
            ILoggerService logger = новый Logger();
            ServiceLocator.RegisterService(регистратор);

            Аналитика IAnalyticsService = новая Analytics();
            ServiceLocator.RegisterService(аналитика);

            Реклама IAdvertisement = новая реклама(); ServiceLocator.RegisterService(реклама);

        }

        недействительный OnGUI()
        {
            GUILayout.Label("Просмотрите вывод в консоли:");

            if (GUILayout.Button("Событие журнала")) { ILoggerService
                logger =
                    ServiceLocator.GetService<ILoggerService>();
                logger.Log("Привет, мир!");
            }

            if (GUILayout.Button("Отправить аналитику")) {
                Аналитика IAnalyticsService =
                    ServiceLocator.GetService<IAnalyticsService>(); аналитика.SendEvent("Привет,
                мир!");
            }

            if (GUILayout.Button("Показать рекламу")) { Реклама IAdvertisement =
                ServiceLocator.GetService<IAdvertisement>(); реклама.DisplayAd();

            }
        }
    }
}
```

```

        }
    }
}

```

Важно отметить, что при фактической реализации шаблона Service Locator в проекте Unity мы должны регистрировать наши сервисы как можно раньше в течение жизненного цикла нашей игры, чтобы они были доступны в любое время — например, эту задачу можно дать к объекту GameManager, который мы инициализировали в первой цене нашего проекта. Если мы знаем, что цена на объект Game Manager всегда будет загружаться, когда игра запускается игру, мы уверены, что реестр шаблона Service Locator будет обновлен до того, как клиенты начнут запрашивать доступ к сервисам.

Ключевое преимущество нашего подхода заключается в том, что мы регистрируем сервисы, связанные с нашими интерфейсами. Это означает, что в момент регистрации сервисов мы можем выбрать, какую конкретную реализацию использовать. Таким образом, мы могли бы легко запустить макетные версии каждого сервиса отладочной сборке. Кроме того, этот подход позволит избежать добавления шума в журналы и аналитику на этапе обеспечения качества (QA).

Следовательно, это одна из замечательных особенностей этого шаблона: вы можете динамически внедрять различные версии служб в зависимости от контекста времени выполнения.



Одним из основных преимуществ использования Unity в качестве движка является то, что он предлагает ряд динамированных сервисов, включая рекламные и аналитические услуги, поэтому в большинстве случаев вам не придется реализовывать их вручную. В проекте доступных сервисов Unity вы можете прочитать по следующей ссылке: <https://docs.unity3d.com/Manual/UnityServices.html>.

## Рассмотрение альтернативных решений

Если у вас возникли проблемы с управлением зависимостями в вашей базе кода, возможно, пришло время начать изучать использование DI-фреймворка. DI — это метод, при котором объект получает необходимые ему зависимости через «внедрение». Объект может получить свои зависимости несколькими способами — через конструктор, установщик или даже через интерфейс, предоставляющий метод внедрения.

Лучший способ начать использовать DI с тщательным образом — использовать структуру, поскольку она помогает вам управлять сложными отношениями между объектами, процессом инициализации и продолжительностью жизни зависимостей. В заключение, вам следует задуматься об использовании DI-фреймворка, когда вы видите тестовую связь между классами и когда их зависимости становятся узким местом для написания согласованного тестового кода.



Extenject — это бесплатная платформа DI для Unity, которую можно загрузить из Asset Store:

<https://assetstore.unity.com/packages/tools/utilities/extenject-dependent-injection-ioc-157735>

—

## Краткое содержание

В этой главе мы рассмотрели шаблон Service Locator. Этот шаблон представляет собой простое решение для решения повторяющейся проблемы управления зависимостями между объектами, зависящими от сервисов (функций), предлагаемых другими объектами. В своей простейшей форме шаблон Service Locator отделяет отношения между клиентом (запрашивающим) и поставщиком услуг.

Мы подошли к концу нашего путешествия, поскольку это последний день нашей книги. Мы надеемся, что вам понравилось содержание каждой главы. Помните, что концепции, представленные в этой книге, — это все олишь введение, а не последнее слово по теме.

О шаблонах проектирования, Unity и разработке игр можно узнать гораздо больше — настолько много, что мы не можем дать этому определение в одной книге. Таким образом, мы призываем вас продолжать обучение, использовать то, что мы рассмотрели вместе в каждой главе, и улучшать его, потому что всегда есть возможности для улучшения.



Packt.com

Подпишитесь на нашу цифровую онлайн-библиотеку, чтобы получить полный доступ к более чем 7000 книг и видеороликов, а также к ведущим в отрасли инструментам, которые помогут вам планировать свое личное развитие и продвигаться по карьерной лестнице. Для получения более подробной информации, пожалуйста, посетите наш веб-сайт.

## Зачем подписываться?

- Тратите меньше времени на обучение и больше времени на профраммирование с помощью практических электронных книг и видео от более чем 4000 профессионалов отрасли.
- Улучшите свое обучение с помощью планов навыков, созданных специально для вас.
- Получайте бесплатную электронную книгу или видео каждый месяц.
- Полная возможность поиска для легкого доступа к важной информации.
- Копируйте и вставляйте, распечатывайте и добавляйте в закладки с одеским.

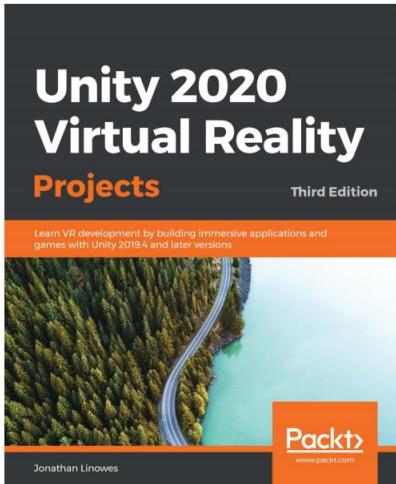
Знаете ли вы, что Packt предлагает электронные версии каждой опубликованной книги с доступными файлами PDF и ePub? Вы можете обновить версию электронной книги на сайте [www.packt.com](http://www.packt.com). И как покупатель печатной книги вы имеете право на скидку на электронную копию. Свяжитесь с нами по адресу [customerservice@packtpub.com](mailto:customerservice@packtpub.com) для получения более подробной информации.

На сайте Packt.com вы также можете прочитать коллекцию бесплатных технических статей, подписаться на ряд бесплатных информационных бюллетеней и получать эксклюзивные скидки и предложения на книги и электронные книги Packt.

Другие книги, которые могут вам понравиться я

## Другие книги, которые могут вам понравиться я

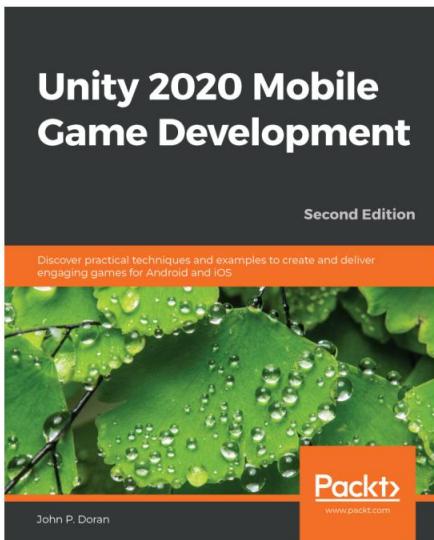
Если вам понравилась эта книга, возможно, вас заинтересует другая книга Packt:



Проекты виртуальной реальности Unity 2020 — третье издание  
Джонатан Линовес

ISBN: 978-1-83921-733-3

- Понять текущее состояние виртуальной реальности и потребительских продуктов VR.
- Начните работу с Unity, создав простуюцену диорамы с помощью редактора Unity и импортированных ресурсов.
- Настройте свои проекты Unity VR для работы на платформах виртуальной реальности, таких как Oculus, SteamVR иimmersivная MR для Windows.
- Спроектируйте и создайте анимации сказ в виртуальной реальности с соудрежением и временными рамками.
- Реализуйте звуковую игру с огненным шаром, используя игровую физику и системы частиц.
- Используйте различные шаблоны программного обеспечения для разработки сценарий Unity и интерактивных компонентов.
- Ознакомьтесь с лучшими практиками освещения, рендеринга и постобработки.



Разработка мобильных игр на Unity 2020 — второе издание  
Джон П. Доран

ISBN: 978-1-83898-733-6

- Создавайте адаптивные пользовательские интерфейсы для своих мобильных игр.
- Обнаруживайте столкновения, получайте данные от пользователя и создавайте движения игроков для ваших мобильных игр.
- Создавайте интересные элементы игрового процесса, используя данные с вашего мобильного устройства.
- Изучите пакет мобильных уведомлений в игровом движке Unity, чтобы поддерживать интерес игроков
- Создавайте интерактивный и визуально привлекательный контент для устройств Android и iOS.
- Монетизируйте свои игровые проекты с помощью Unity Ads и покупок в приложениях.

## Packt ищет таких авторов, как вы

Если вы хотите стать автором Packt, пожалуйста, посетите сайт [authors.packtpub.com](http://authors.packtpub.com) и подайте заявку сегодня. Мы работали с тысячами разработчиков и технических специалистов, таких как вы, чтобы помочь им поделиться своими знаниями с мировым сообществом. Вы можете подать общую заявку, подать заявку на конкретную актуальную тему, для которой мы набираем автора, или подать с общей темой.

## Поделитесь с вашими мыслями

Теперь, когда вы закончили «Шаблоны разработки игр с Unity 2021», мы будем рады услышать ваши мысли! Если вы купили книгу на Amazon, нажмите здесь, чтобы перейти прям на страницу обзора Amazon. об этой книге и поделитесь своим мнением или оставьте отзыв на сайте, на котором вы ее купили.

Ваш отзыв важен для нас и технического сообщества и поможет нам убедиться, что мы предоставляем контент отличного качества.

# Индекс

## А

Шаблон адаптера, адаптер класса  
подхода 183 адаптер  
объекта 183  
Схема адаптера около  
183, 184, 185 Преимущество  
185  
недостатки 185  
внедрение, тестирование 190, 191 внедрение  
187, 189, 190 необходиимо 186  
  
рекламная сеть (рекламная сеть) 207 интерфейс  
прикладного программирования (API) 10  
Искусственный интеллект (ИИ) 10, 21

## Б

Деревья поведения (BT) 61  
проектирование  
велосипедного  
двигателя 195 фасад  
тестирование 201, 202 реализация 196, 198,  
199, 200 разделение двоичного пространства (BSP) 168  
Маневр подрывания  
Blade Racer 14 139

## С

Расширенные функции языка C#,  
делегаты 31,  
события 30  
дженерики 32  
серIALIZАЦИЯ 32  
статический 29  
C#  
основные функции 28  
камера 18  
камера, управление, персонааж (3Cs) 18

состояния персонаажа  
определение 51, 52  
класс с адаптером 183  
коллекционная карточная игра (CCG) 153  
Шаблон команды  
около 78, 79, 80  
преимущества  
81 недостаток 81  
необходимость 81  
развязка  
основных компонентов  
контролера 21, с шаблоном Observer 109, 110  
основной игровой  
цикл,  
столпы 18, с опорой раммы 33

## Д

Шаблон декомпьютера  
151, 152  
альтернативных решений, обзор 166  
преимущества,  
152 недостатков 152, 153  
необходимости 153  
делегаты 31  
внедрение зависимостей (DI) 206 принцип  
инверсии зависимостей (DIP) 205 шаблонов  
проектирования  
около 10  
случаев использования с Unity 9

## Э

разработка  
вражеского дрона 139, 140, 141  
реализация, проверка 148 реализаций,  
испытания 146, 148 реализации 141, 144,  
146  
Шаблон шины событий

около 64, 65, 66 выг оды  
66

недостатки 66

Автобус для мероприятий

реализация , расмотрение 75 с  
использованием 67  
события 30

## Ф

Система фаза о 193,  
194 альтернативных  
решениях , обзор 202 преимущества 194

недостатки 194

запасной маневр 139

конечный автомат (автомат) 138

## Г

документы игрового дизайна (GDD) о 9, 14

камера

управление , персонаж (3Cs) 18 основной игровой  
цикл 17 игровая среда

игровые ингредиенты 21  
игровое меню 25 цели игры

16 правила игры 17

краткий обзор игры 15, 16

игровые системы

23 проектационный дисплей  
(HUD) 25 минимальные  
требования 15 обзор 14 уникальные  
преимущества 14 игровые  
ингредиенты

препятствия 23 пикапы 22

супербаки 22

вооружение 23

Геймменеджер

проектирование 39, 40

реализация 40

тестирование 45,

46 игровой проект  
обзор 11

игровые системы

около 23

железнодорожная

система 23 системаиков 23

Система Turbo Boost (TBS) 24 Система

модернизации транспортных средств

24 глобальных гоночных

события Управление 67, 68

## Н

Проектационный дисплей (HUD)

около 24, 25, 68, 207 Компоненты

интерфейса 25

## Л

редактор уровней

альтернативные решения , расмотрение 179, 180

разработка 170, 171, 172, 173 реализация ,

расмотрение 179 реализация 173, 176,

178 использование 179

## О

объектный адаптер 183

Шаблон пульта объектов

95, 96

альтернативных решений , обзор 105

преимущества 96

недостатки 97

внедрение , расмотрение 105 внедрение ,

тестирование 103 внедрение 98, 102,

103 потребность 97

объектно-ориентированное программирование (ООП) 9, 185

Модель наблюдателя :

107, 108

альтернативных решений , обзор 119

преимущества ,

108 недостатков , 108

реализация , тестирование 117,

внедрение 111, 113

необходимость 109

используется для разработки новых компонентов 109, 110

P

персонах, управля емый иг роком,  
около 19 лет  
описание персонажа 19 метрики  
персонажа 20  
состояния персонажа 20  
механика включения  
питания  
проектирование 124  
реализация 125 реализация систе мы, обзор 133  
реализация систе мы, тес тирование 131, 133 систе ма,  
реализация 125 практическое  
использование разработки игр 8, 9  
сборные  
конст рукции 32  
Вы кройка-прототип 105  
вопрос

этап обес печеня качества (QA) 212

P

Автобус для гоночных мероприятий  
альтернативные решения , расмотрение 76  
внедрение 68, 70  
тестирование 70,  
72 альтернативные  
модели и решения систе мы воспроизведения , расмотрение  
93 проектирование 82,  
83 внедрение, расмотрение 92  
внедрение 84, 86, 88 сецификац ии  
82 тестирование 88, 90,  
92  
каличество оборотов в минуту (RPM) 202  
ролевая игра (RPG) 11 ролевая игра  
(RPG), пример развлечения 11

производительность

11 личный 11

прос тога 11

C

Клас с ScriptableObjects 33  
Шаблон локатора с лужбы около  
204, 205

альтернативные решения , обзор 212 преимущество  
206

характеристики 207

недостатки 206

реализация 207, 209, 210 потребность  
в 206 услугах

207 тестирование

211

Шаблон Singleton около

37, 38,

преимущества 39

недостатки 39

Схема пространственного

разделения примерно 168,  
169, 170, нужно 170.

Укажите шаблон

о 49

преимуществах 59

альтернативных решений х , расмотрение 61

недостатков 61

реализации, тестировании 58, 59

реализации 52, 53, 57

ограничения х 60

структурой 50

статическое ключевое

слово 29 стратегии 137

Модель с стратегии, ключевые иг роки,

конкретная стратегия 137,

контекст 137

интерфейс стратегии 137

Схема стратегии:

136

альтернативных решений, обзор 148, 149 преимущества,  
137 недостатков

137, 138, необх одимость 138

T

трехмерное (3D) 10, 168

Система турбонаддува (TBS) 24

ты

Действия Единства 33

Движок Unity

около 10

ос обеннос ти 32, 33, 34

Мероприя тия Е динс тва 33

Ядро

Unity включает 28

шаблонов проектирования , ис пользование 9.

пользовательс кий интерфейс (UI) 207

## B

Шаблон пос етителей

около 121, 122, 123

преиму щес тва 123

недос татки 123

Разработка с и стемы

вооружения W 154

реализац ия 155, 156, 158, 160 рас смотрение

165, 166 ис пытания 162,

163, 164, 165

плетение маневра 139