1 Подготовка системы

Наша работа как программистов заключается в том, чтобы проектировать, создавать и интегрировать программные функции в целостные продукты, которые будут полезны и, как мы надеемся, принесут радость нашим пользователям. И где-то среди всего этого программное обеспечение (ПО) должно работать. Мы можем заниматься чем угодно — от создания функции входа в систему или сетевого уровня до страницы подписки и оплаты. В играх мы думаем о том, как переместить игрового персонажа, создать адаптивный искусственный интеллект (ИИ) врага или добавить генератор случайных уровней. Главное, что вы должны понять, — эти элементы не существуют в вакууме.

Когда вы думаете о коде, который управляет вашими приложениями и играми, вы думаете о наборе систем, построенных для выполнения определенных функций или механик, но, что еще важнее, для решения определенных задач, чтобы заставить эти функции работать так, как ожидается. Если бы мне удалось это сделать, я бы заставил слово «системы» спрыгнуть со страницы и станцевать, чтобы привлечь ваше внимание, потому что вся книга именно об этом: *игры (и приложения) состоят из систем — чем больше эти системы растут и взаимодействуют друг с другом, тем сложнее они становятся*.

Возможно, системы не являются для вас чем-то новым, и уж точно они не новы для индустрии программирования, но есть нюансы, которые нужно учитывать и которые напрямую связаны с работой и взаимодействием систем. И именно это я хочу донести до вас, прежде чем мы начнем: паттерны проектирования обеспечивают *повторно используемые* решения для устранения проблем и предлагают общий язык для понимания и обсуждения лучших практик.

Теперь начинается настоящая работа — как определить, к каким частям нашего кода стоит применить паттерн проектирования? Как все это связано с архитектурой ПО? Как адаптировать эти паттерны для разработки игр в Unity?

Это очень важные вопросы, поэтому первую главу посвящу следующим темам:

- архитектура ПО в сравнении с проектированием ПО;
- паттерны проектирования и их категории (порождающие, структурные и поведенческие);
- зачем нужны паттерны проектирования;
- когда следует использовать паттерны проектирования;
- скрытые проблемы;
- стартовые проекты.

К концу главы у вас будет необходимая основа для погружения в реализацию паттернов проектирования, а также общее представление о гибкости, возможности повторного использования и структуре, которые они могут привнести в ваши проекты. Но сначала нужно убедиться, что вы знаете, чего ожидать от книги, и что у вас есть необходимые предварительные знания для получения максимальной пользы от этого опыта.

Все примеры из глав и файлы проектов были созданы с помощью Unity 2023.1.5f1. Unity — это визуальный инструмент, который мы используем, чтобы сделать процесс обучения более привлекательным. Основные различия между Unity, C# и паттернами проектирования заключаются в том, как создаются объекты (новые классы против скриптовых компонентов MonoBehaviour) и собираются (наследование против объектов GameObject и компонентов Component). Не волнуйтесь, мы рассмотрим эти вариации в каждом паттерне. Изучение принципов работы каждого паттерна проектирования не зависит от конкретной платформы или языка — это все навыки решения проблем!

Несколько слов об архитектуре программного обеспечения

Архитектура ПО — это обширная тема, и ее не следует путать с проектированием ПО, хотя оба понятия связаны с «паттернами». Когда мы говорим об архитектурных паттернах, мы имеем в виду задачи, которые обычно сосредоточены на решении проблем, связанных с совместной работой нескольких компонентов приложения, а также соглашения, лучшие практики и принципы, которые направляют общий процесс разработки. А когда мы говорим о паттернах проектирования ПО, мы подразумеваем проблемы, возникающие при создании этих внутренних компонентов. На рис. 1.1 представлен упрощенный пример,

показывающий эту разницу. *Процесс A* состоит из множества внутренних компонентов — это часть, связанная с проектированием ПО, в то время как управление и организация того, как эти внутренние компоненты сочетаются друг с другом, относятся к архитектурной части.

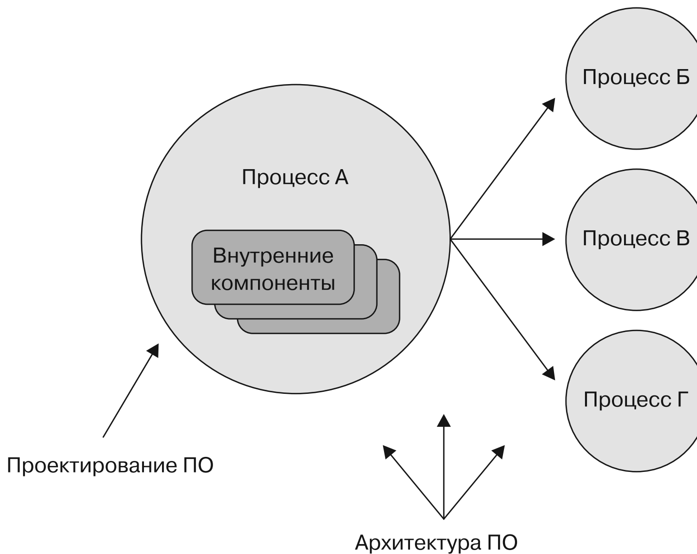

Рис. 1.1. Разница между архитектурой и проектированием ПО

Сложно? Хорошо, рассмотрим более понятный пример: если мы думаем о строительстве дома, то архитектурные паттерны сосредоточены на том, как все его элементы собираются воедино и как составные части (электрика, водопровод, изоляция и т. д.) объединяются в функционирующую структуру, в которой можно жить. Вспомните LEGO (мне это всегда помогает): архитектурные паттерны решают проблемы, влияющие на общую структуру, которую вы строите, в то время как паттерны проектирования сосредоточены на отдельных блоках LEGO, из которых состоит конечная структура. В идеале вам нужны оба типа решений для создания отличного приложения или игры, но они служат для устранения принципиально разных проблем, вот почему я заостряю на этом внимание в самом начале. Мы сосредоточимся на проектировании ПО, но вы можете (и должны) продолжить свое путешествие в мир архитектуры после (или параллельно) прочтения книги.

В повседневной жизни, когда вы открываете любое приложение, вы участвуете в так называемом архитектурном паттерне *«клиент — сервер»*: клиент (приложе-

ние на вашем телефоне) получает информацию с сервера, расположенного где-то в другом месте, и отображает ее в понятном для вас виде. Но его не волнует, как приложение запоминает ваши учетные данные для входа или использует распознавание лиц для авторизации. Это вопросы проектирования ПО, потому что они сосредоточены на внутренних функциях приложения.

Прежде чем мы перейдем к вопросам, почему, когда и как использовать паттерны проектирования, нам необходимо прийти к соглашению о том, что такое *хорошее* проектирование ПО, чтобы увидеть преимущества паттернов.

Мне нравится думать о проектировании ПО как об инструменте, который может как усложнить жизнь в краткосрочной перспективе, так и облегчить ее в долгосрочной, как, например, посещение спортзала. Каждый день ходить в тренажерный зал — это мучение, но здоровый образ жизни имеет далекоидущие положительные последствия. То же самое можно сказать и о подходе к проекту с точки зрения разработки программного обеспечения. Учитывая это, давайте сформулируем для себя простое определение, которое станет своего рода ориентиром того, что мы подразумеваем под хорошим проектированием ПО.

Гибкая, поддерживаемая и многократно используемая кодовая база — это результат хорошего проектирования программного обеспечения.

Это легче сказать, чем сделать, независимо от того, насколько добросовестно вы или ваша команда начинаете разработку игры. Разделение кода может ускорить процесс разработки, но это увеличивает накладные расходы и время на обслуживание. С другой стороны, взаимозависимый код может быть дорогостоящим, когда вы захотите внести в него какие-либо изменения. Быстро, хорошо или дешево — у вас может быть только два варианта.

Определившись с тем, что такое *хорошее* проектирование ПО, можем перейти к обсуждению паттернов, их классификации и места в проектах Unity.

Понятие паттернов проектирования

Если я недостаточно акцентировал внимание на этом моменте, я хочу, чтобы вы снова и снова повторяли про себя: «Паттерны проектирования — это системы». Еще раз: паттерны проектирования — это системы, а системы предназначены для решения конкретных задач. Неважно, идет ли речь о тормозной системе вашего автомобиля, о биологических системах, которые управляют нашим телом, или о банковской системе, как показано на рис. 1.2. Все это системы, и все они служат для решения какой-то проблемы или поддержания своей системы в организованном и сбалансированном состоянии (а иногда и то и другое).

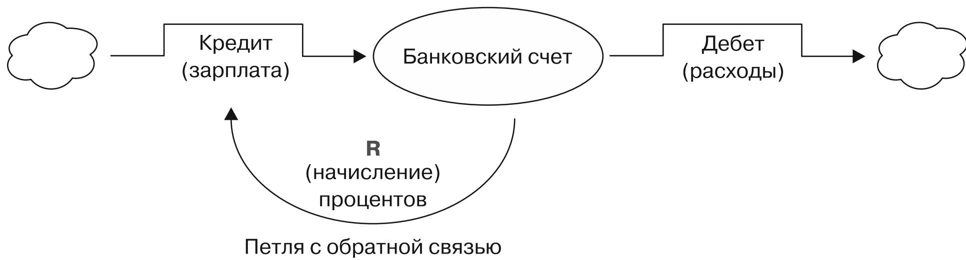

Рис. 1.2. Схема работы банковских счетов с использованием кредита и дебета

Прежде всего паттерны проектирования направлены на то, чтобы сделать код более гибким и пригодным для повторного использования, — два постулата, которые мы будем транслировать на протяжении всей книги. Просто повторяйте нашу мантру каждый раз, когда мы изучаем новый паттерн: паттерны проектирования — это системы!

«Банда четырех»

В 1994 году четыре смелых инженера объединились, чтобы лучше разобраться с практикой ООП и повторяющимися проблемами, которые постоянно возникали в их программах. Книга Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса «Паттерны объектно-ориентированного проектирования»[1] стала результатом этого сотрудничества. В ней рассматриваются 23 паттерна проектирования, разделенных на три категории в соответствии с их основными функциями, с примерами, написанными на C++ и Smalltalk. Хотя книга была и остается важным источником информации, на сегодняшний день эти паттерны были адаптированы и расширены, а часть из них стали более (а некоторые менее) актуальными в зависимости от языка программирования и среды.

Вы наверняка слышали фразу «Банда четырех», произносимую с тихим благоговением (или громко и с гневным размахиванием кулаками, в зависимости от того, с кем вы разговариваете). Но я пришел к выводу, что лучше сначала освоить паттерны проектирования как навык, прежде чем делать выводы. Многие споры теряются в теоретических или педагогических тонкостях (и личностях), но базовый навык работы с паттернами проектирования всегда был полезным инструментом в программировании.

---

[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2022.

Есть и те, кто отмечает, что многие из этих паттернов являются заменой недостающих функций в языке C++. Однако этот аргумент никогда не находил отклика во мне при просмотре своего кода на других языках (будь то C# или что-то вроде Swift), поскольку, надевая «очки системного мышления», я всегда начинаю мыслить более критически и писать код более осознанно.

Категории паттернов

Существует три категории, к которым относятся все оригинальные паттерны проектирования, — *порождающие* (creational), *поведенческие* (behavioral) и *структурные* (structural).

Прежде чем мы углубимся в детали категорий, я считаю важным обсудить проблему, которая возникает в самом начале изучения паттернов проектирования. Как найти подходящий паттерн для решения моей задачи? Нужно ли читать и запоминать каждый паттерн, или есть более эффективный способ ориентироваться в этой теме?

Ответ может вас удивить, но нет, вы не получите дополнительные баллы за знание каждого паттерна проектирования. Системное мышление — это приобретаемый навык, а не тест на память. Во-первых, *очень* важно понимать, какие проблемы решает каждая категория паттернов, потому что это сужает область поиска. Во-вторых, прочитав первые несколько страниц каждой главы в соответствующей категории, вы быстро поймете, находитесь ли вы на правильном пути. Чем чаще вы будете использовать паттерны проектирования, тем лучше вы будете понимать проблемы и принимать эффективные решения в реальном проекте. Как вы увидите, паттерны проектирования предлагают решения хорошо задокументированных проблем, но они не являются незыблемыми. Вам предстоит адаптировать их под свои задачи.

Теперь, когда мы знаем основы, углубимся в тонкости каждой категории паттернов проектирования и в то, какие конкретные проблемы они призваны решить.

Порождающие паттерны

Порождающие паттерны связаны с созданием объектов. Этот тип паттернов касается того, как скрыть логику создания объектов и классов, чтобы вызывающий экземпляр не был перегружен деталями. По мере того как ваши потребности в создании объектов и классов будут расти, эти паттерны помогут вам отказаться от жестко фиксированных моделей поведения и перейти к написанию небольших наборов моделей поведения, которые можно использовать для создания более сложных функций (вспомните LEGO). Хороший порождающий паттерн закрывает логику создания и просто передает утилиту для управления тем, кто (как и когда) создает объект или класс.

Порождающие паттерны, которые мы рассмотрим, перечислены в табл. 1.1.

Таблица 1.1. Список порождающих паттернов проектирования с описанием

Паттерн	Описание
Одиночка (Singleton), с. 41	Гарантирует, что класс имеет только один экземпляр, и обеспечивает глобальную точку доступа к нему. Обычно используется для таких функций, как ведение журнала или подключение к базе данных, которые должны быть скоординированы и доступны во всем приложении
Прототип (Prototype), с. 71	Определяет типы объектов для создания с помощью прототипического экземпляра и создает новые объекты на основе «скелета» существующего объекта
Фабричный метод (Factory Method), с. 101	Определяет интерфейс для создания одного объекта, но делегирует логику создания экземпляра подклассам, которые сами решают, какой класс инстанцировать
Абстрактная фабрика (Abstract Factory), с. 146	Определяет интерфейс для создания семейств связанных или зависимых объектов, но позволяет подклассам решать, какой класс инстанцировать
Строитель (Builder), с. 172	Позволяет создавать сложные объекты шаг за шагом, разделяя процесс построения объекта от его представления. Обычно используется при создании различных версий объекта
Пул объектов (Object Pool), с. 199	Помогает избежать дорогостоящего создания и высвобождения ресурсов путем повторного использования объектов, которые больше не задействуются. Обычно применяется, когда ресурсы дороги, в избытке или и то и другое

Поведенческие паттерны

Поведенческие паттерны связаны с тем, как классы и объекты взаимодействуют друг с другом. Точнее, эти паттерны концентрируются на различных обязанностях и связях между объектами, когда они работают вместе. Как и структурные паттерны, поведенческие используют наследование для разделения поведений между классами, что позволяет вам освободиться от жесткого контроля над потоком управления и фокусироваться на том, как объекты могут взаимодействовать.

Поведенческие паттерны, которые мы рассмотрим, перечислены в табл. 1.2.

Таблица 1.2. Список поведенческих паттернов проектирования с описанием

Паттерн	Описание
Команда (Command), с. 231	Инкапсулирует запрос в виде объекта, что позволяет параметризировать клиентов с различными запросами, а также ставить запросы в очередь или регистрировать их

Паттерн	Описание
Наблюдатель (Observer), с. 269	Определяет отношение «один ко многим» между объектами, при котором изменение состояния одного объекта приводит к автоматическому уведомлению и обновлению всех его зависимых объектов
Состояние (State), с. 316	Позволяет объекту менять свое поведение при изменении его внутреннего состояния. Объект будет выглядеть меняющим свой класс. Обычно используется, когда поведение объекта кардинально меняется в зависимости от его внутреннего состояния
Посетитель (Visitor), с. 353	Определяет новую операцию класса без изменения самого объекта
Стратегия (Strategy), с. 378	Определяет семейство взаимозаменяемых моделей поведения и откладывает выбор конкретного поведения до момента выполнения
Объект как тип (Type Object), с. 403	Позволяет гибко создавать новые «классы» на основе одного класса, каждый экземпляр которого будет представлять объект другого типа
Хранитель (Memento), с. 429	Захватывает и передает вовне внутреннее состояние объекта, чтобы его можно было восстановить или вернуть к этому состоянию позже — без нарушения инкапсуляции

Структурные паттерны

Структурные паттерны сосредоточены на композиции, то есть на том, как классы и объекты объединяются в более крупные и сложные структуры. В структурных паттернах много абстракции, что упрощает управление и настройку отношений между объектами. Паттерны этой категории задействуют наследование, чтобы вы могли комбинировать структуры классов, а также создавать объекты с новой функциональностью.

Структурные паттерны, которые мы рассмотрим, перечислены в табл. 1.3.

Таблица 1.3. Список структурных паттернов проектирования с описанием

Узор	Описание
Декоратор (Decorator), с. 453	Динамически добавляет объекту новые обязанности, сохраняя при этом тот же интерфейс
Адаптер (Adapter), с. 478	Преобразует интерфейс класса в другой интерфейс, ожидаемый клиентами. Адаптер позволяет работать вместе классам, которые иначе не могли бы работать из-за несовместимости интерфейсов

Окончание ☞

Таблица 1.3 *(окончание)*

Узор	Описание
Фасад (Façade), с. 496	Предоставляет единый интерфейс для набора интерфейсов в подсистеме. Фасад определяет высокоуровневый интерфейс, который упрощает использование подсистемы
Приспособленец (Flyweight), с. 511	Обменивается общими данными между похожими объектами, чтобы ограничить использование памяти и повысить производительность
Локатор служб (Service Locator), с. 538	Предоставляет глобальную точку доступа к службам, не привязывая клиентский код к классам конкретных служб

Каждый из этих паттернов представляет собой согласованное, многократно применяемое решение.

Зачем нужны паттерны проектирования

Коротко: для многократного использования, гибкости и удобства сопровождения. Эти понятия часто встречаются в темах, связанных с архитектурой ПО и сложностью кода, но они являются основными постулатами паттернов проектирования. Например, если вам нужно создать вариации одного и того же объекта в вашей игре, то логичнее будет создать систему сборки вместо того, чтобы прописывать каждый объект вручную. На рис. 1.3 показана сборочная линия, на которой производятся маленькие роботы, причем на каждом этапе выполняется определенная функция в процессе создания объекта. Это упрощенная, но эффективная ментальная модель для паттерна Строитель.

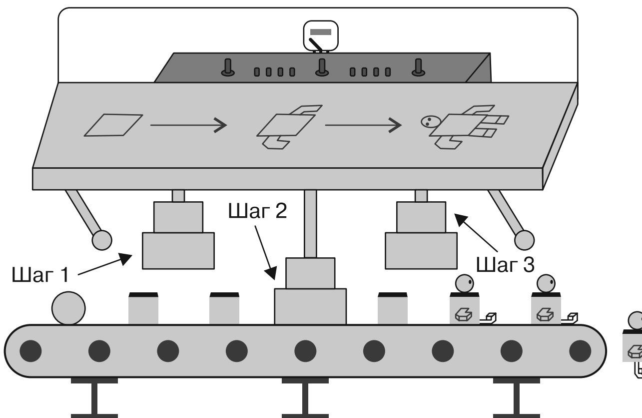

Рис. 1.3. Шаблон многоразового конструктора, создающего объекты

Начнем с *многократного использования*, которое уже рассматривается в таких принципах ООП, как инкапсуляция, абстракция и наследование, и является краеугольным камнем многих паттернов проектирования в данной книге. Нет ничего лучше, чем, сев за реализацию новой функции, осознать, что у вас уже есть все необходимые строительные блоки. Все, что от вас требуется, — сложить их в форму, которая заставит функцию работать, и протестировать ее. С помощью паттернов проектирования вы можете применить наследование классов, делегирование и композицию объектов для создания многократно используемых компонентов вашего игрового кода. У каждого из них есть свои плюсы и минусы, о которых мы поговорим позже, но они также могут повысить эффективность при написании нового кода.

Что касается *гибкости*, то паттерны проектирования помогут вам структурировать код с учетом будущих изменений. Когда мы говорим об изменениях и возможности повторного использования, мы имеем в виду код, который может справиться с меняющимися требованиями *и* масштабами. Ни одна программа, которую вы когда-либо напишете, не будет статичной, потому что ни один код не бывает идеально сбалансирован с первой попытки. Если вы не предусмотрите гибкость, первый же «сильный ветер» разорвет ваш проект на части, а это означает перепроектирование, внедрение новых функций и новые раунды тестирования.

Прелесть правильного использования паттернов проектирования заключается в том, что вам не нужно выбирать только один — вы можете комбинировать их, чтобы получить желаемый результат. Например, вашей игре всегда будут нужны элементы управления игровым персонажем (рис. 1.4). Но что, если вы захотите поменять местами отдельные команды или назначения кнопок? Ваш код будет более гибким, если вы сможете отделить ввод элементов управления от их реализации, и с этим как раз отлично справляется паттерн Команда.

Рис. 1.4. Схема гибкого управления, использующего паттерн Команда

Наконец, мы переходим к *удобству сопровождения*. Идеальный вариант — иметь рабочий код с *правильным* уровнем архитектуры и сложности. Это не только делает наш код читаемым, но и упрощает отладку, если каждый компонент или класс имеет определенную роль, которую можно изолировать. Я акцентирую внимание на слове «правильный», потому что это непростая задача. С одной стороны, вы хотите, чтобы в вашем проекте было достаточно структуры, чтобы обновление кода было максимально простым и эффективным. С другой — излишняя сложность может загромождать проект, оказывая прямо противоположный эффект. Корректное определение паттернов проектирования, которые помогут вам достичь первого, а не второго, — это и есть нахождение баланса при сопровождении кода (рис. 1.5).

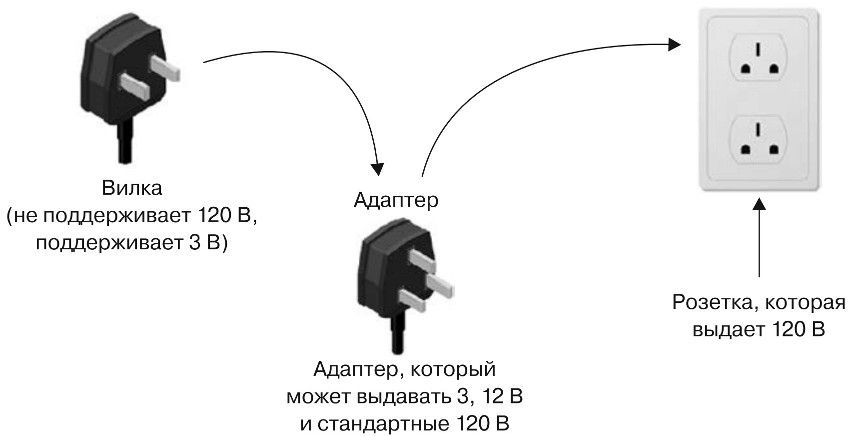

Рис. 1.5. Паттерн Адаптер как пример проектирования для возможности внесения изменений и простоты сопровождения

Когда следует использовать паттерны проектирования

Вы знаете, что делаете что-то интересное, когда ответ на вопрос «почему» или «когда» одновременно безумно прост и сложен. Самый распространенный ответ, который вы можете услышать, звучит так: их следует использовать только тогда, когда они необходимы. Полезно, правда?

Простой способ понять, стоит ли применять тот или иной паттерн проектирования, — это выявить конкретную проблемную область в коде. Симптомами проблемы могут быть «запахи кода», такие как тесно связанные классы, монолитные иерархии подклассов, зависимости от конкретных операций, представлений объектов и оборудования. Аппаратные и программные зависимости особенно остро ощущаются при разработке игр, когда необходимо переносить код на различные устройства или системы.