УНИВЕРСИТЕТ ИТМО

International Culture & Technology Studies
online multimedia journal

ITMO UNIVERSITY

Культура и технологии
электронный мультимедийный журнал

Journal Homepage: http://cat.ifmo.ru
ISSN 2587-800X

Адрес статьи / To link this article: http://cat.ifmo.ru/ru/2020/v5-i1/225

Реализация поведенческих шаблонов проектирования на примере игрового приложения

В.В. Эпп, И.С. Паршин

Пензенский государственный университет, Россия

vitalinae@mail.ru, frest58rus@yandex.ru

Аннотация. В век активного развития информационных технологий большую роль играет оптимизация и ускорение разработки приложений без потери качества. Программирование — недетерминированный процесс и для одной задачи существует огромное количество решений, из-за чего у программиста возникает сложности с выбором того или иного решения. Особенно в такой сфере, как разработка игровых приложений, где надо создать высокопроизводительное приложение в короткие сроки. В рамках статьи будет рассмотрена разработка игры в среде Unity с использованием поведенческих шаблонов (паттернов) и без них. Шаблоны проектирования — это лишь общее решение задачи, которое может быть дополнено и подведено под необходимые требования. Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, вычленяет участвующие классы и экземпляры, их роль, а так же отношения и функции — что, в конце концов, позволяет применять его для создания повторно используемого дизайна.

Ключевые слова: паттерны проектирования, паттерн «Стратегия», паттерн «Состояние», разработка игр в Unity

В настоящее время разработка игровых приложений сводится к достижению наибольших продаж при минимизации финансовых и временных затрат. В целом, это достигается за счет различных инструментов, в число которых входят паттерны проектирования. Паттерны предоставляют возможность одинакового понимания дизайна решения задачи у всех членов команды, как проектировщиков, так и у самих программистов-разработчиков. Использования паттернов в процессе создания кода сокращает время, которое затрачивается на обсуждение и принятие того или иного решения.

Первый паттерн, с помощью которого разрабатывалась игра — это паттерн «Состояние» [1] — шаблон, позволяющий объекту изменять своё поведение в зависимости от внутреннего состояния. UML диаграмма классов шаблона представлена на рисунке 1.

Участниками паттерна являются:

– Context — контекст, определяющий интерфейс, который представляет интерес для клиентов. Так же хранит экземпляр подкласса ConcreteState, которым определяется текущее состояние.

- State — состояние, которое определяет интерфейс для инкапсуляции поведения, ассоциированного с конкретным состоянием контекста Context.
- Подклассы ConcreteState — конкретное состояние, каждый класс которого реализует поведение, ассоциированное с некоторым состоянием контекста Context.

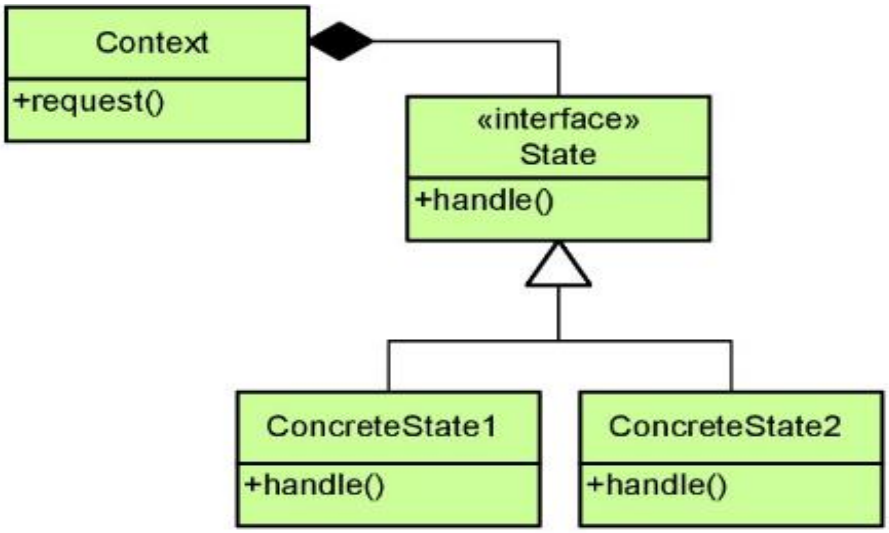

Рис. 1. Диаграмма классов паттерна «Состояние»

Шаблон рекомендован для использования в условиях, когда:

- Поведение объекта зависит от его состояния и должно изменяться во время выполнения.
- В коде операций встречаются состоящие из многих ветвей условные операторы, в которых выбор ветви зависит от состояния.

Второй паттерн, с помощью которого разрабатывалась игра — это паттерн «Стратегия» [1] — шаблон, который позволяет выбирать алгоритм путём определения соответствующего класса. Паттерн позволяет менять выбранный алгоритм независимо от объектов-клиентов, которые его используют. Задача шаблона — по типу клиента (или по типу обрабатываемых данных) выбрать подходящий алгоритм, который следует применить. UML-диаграмма классов шаблона представлена на рисунке 2.

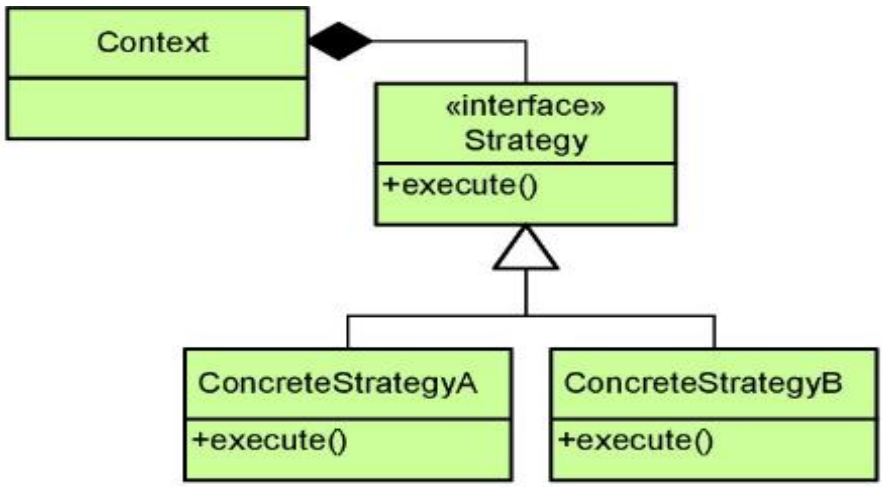

Рис. 2. Диаграмма классов паттерна «Стратегия»

Участниками паттерна являются:

- Strategy — стратегия, объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс Context пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе ConcreteStrategy.
- ConcreteStrategy — конкретная стратегия, реализует алгоритм, использующий интерфейс, объявленный в классе Strategy.
- Context — контекст, который конфигурируется объектом класса ConcreteStrategy и хранит ссылку на объект класса Strategy. Так же может определять интерфейс, который позволяет объекту Strategy получить доступ к данным контекста.

Достоинствами паттерна является то, что он выступает альтернативой порождения подклассов, с его помощью можно избавиться от условных операторов и выделить семейство родственных алгоритмов. Так же предоставляет клиенту возможность выбора реализации в зависимости от своих требований к быстродействию и памяти. В качестве минусов можно отметить то, что клиенты должны знать о различных стратегиях и понимать чем они отличаются, и то, что при использовании данного шаблона увеличивается количество объектов в приложении.

Паттерн рекомендован для использования в условиях, когда:

- Имеется множество родственных классов, которые отличаются только поведением.
- Необходимо иметь несколько разных вариантов алгоритма. Шаблон разрешается применять, когда варианты алгоритмов реализованы в виде иерархии классов.
- Необходимо скрыть от клиента сложные и специфичные для алгоритма структуры данных.
- В классе определено большое количество поведений, что представлено разветвленными условными операторами. В таком случае будет целесообразно перенести код из ветвей в отдельные классы стратегий.

Игровое приложение, о котором рассказывается в этой статье, реализовывает 2 поведенческих паттерна проектирования: «Состояние» и «Стратегия», а так же пример без использования шаблонов (рис. 3). Для возможности быстрого переключения и оценки каждого примера, каждая часть представлена в виде отдельного игрового уровня в рамках единого игрового приложения. На каждом уровне разработаны идентичные игровые объекты с одинаковым поведением, визуально ничем не отличающиеся, но имеющие различия в структуре кода.

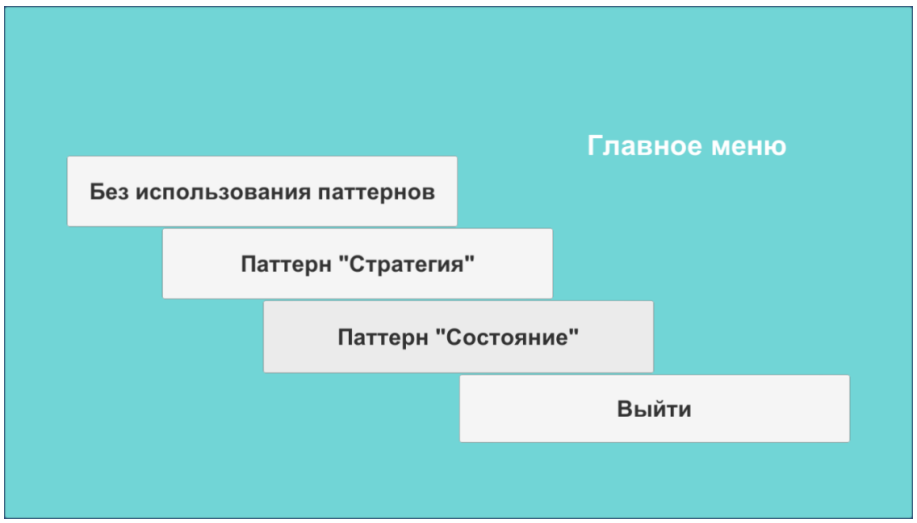

Рис. 3. Игровое меню

Для наглядности в приложении разработаны персонажи, у которых изменяется состояние (движение объекта). Выделено 8 видов поведения: атака, отступление, движение вверх, движение вниз, движение влево, движение вправо, стойка на месте и отдельное поведение для игрового объекта, управляемого пользователем.

Диаграмма последовательности игрового приложения, представлена на рисунке 4.

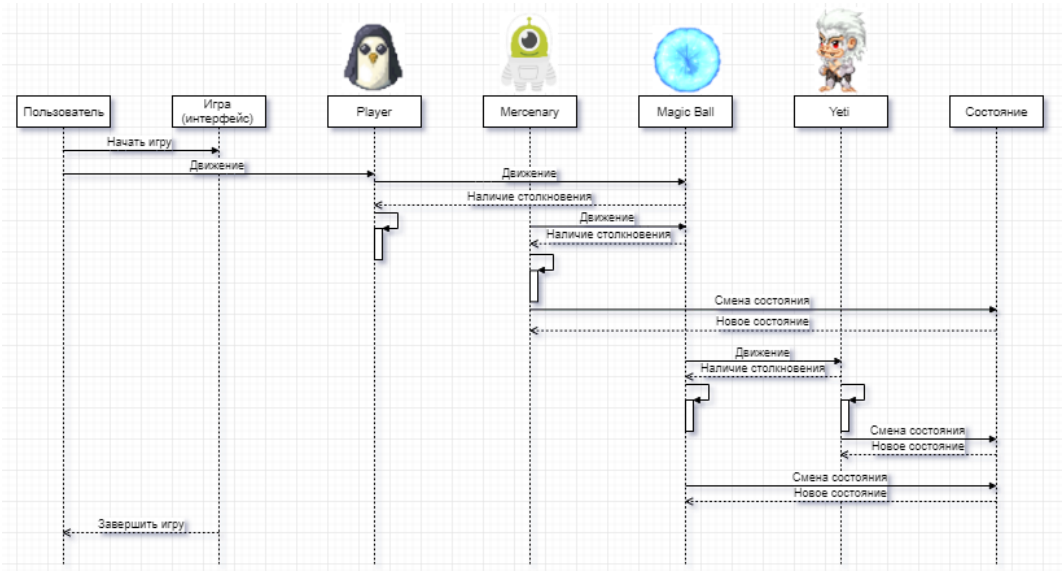

Рис. 4. Диаграмма последовательности сценария приложения

Реализация паттерна «Стратегия» представлена в виде абстрактного класса, являющегося родительским классом всех создаваемых игровых объектов уровня. Класс содержит экземпляр интерфейса, который содержит в себе функцию движения, а так же возможность переключения стратегии. Само изменение поведения выполняется с помощью функции, которая в качестве параметра принимает класс-поведение. Для каждого конкретного поведения игрового объекта разработан класс, реализующий конкретную стратегию. Все такие классы унаследованы от единого интерфейса, содержащего функцию движения. На рисунке 5 представлена диаграмма классов для игрового уровня, реализующего паттерн «Стратегия».

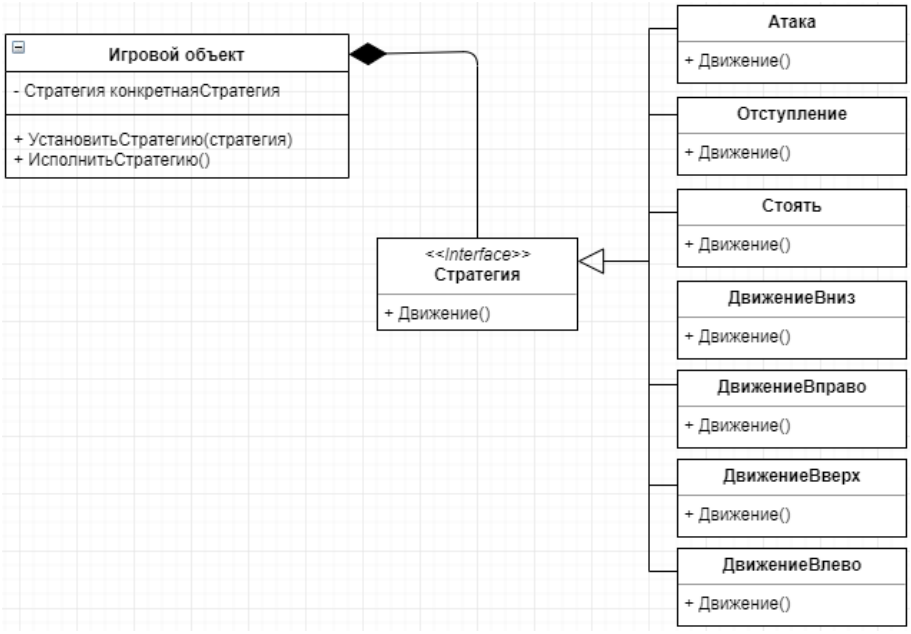

Рис. 5. Диаграмма классов паттерна «Стратегия»

Реализация паттерна «Состояние» представлена в виде абстрактного класса. Этот класс является родительским для всех создаваемых игровых объектов в рамках текущего игрового уровня. Класс содержит в себе экземпляры классов-состояний, функцию движения и возможность переключения состояния. Все классы состояний унаследованы от единого интерфейса и реализуют конкретное поведение. На рисунке 6 представлена диаграмма классов для игрового уровня, реализующего паттерн «Состояние».

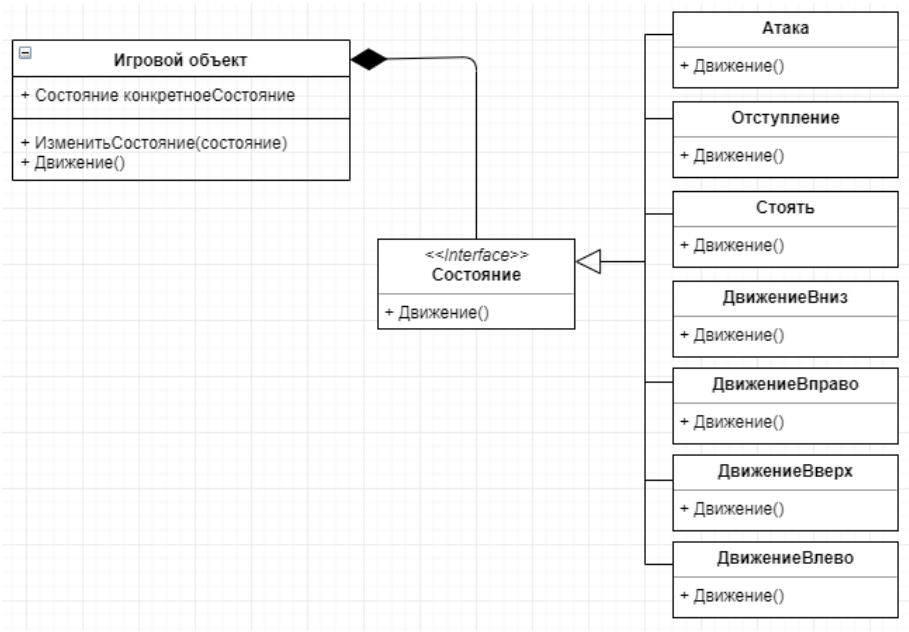

Рис. 6. Диаграмма классов паттерна «Состояние»

Реализация уровня без применения шаблонов проектирования подразумевает самое обычное программирование. В рамках разработки этого уровня реализован единый абстрактный класс, который содержит в себе реализацию движения для каждого конкретного состояния. Набор состояний представлен в виде перечисления, который содержит в себе весь вышеописанный список состояний. Таким образом, класс должен содержать в себе экземпляр состояния, а так же возможность изменения состояния, от которого зависит тип движения. Переключение состояния реализовано в виде машины состояний, которая представлена в виде конструкции switch, сравнивающей состояния. В зависимости от исполняемого блока, выполняется то или иное движение, соответствующее состоянию.

В редакторе Unity [2] смоделированы следующие игровые объекты: Magic Ball (рис. 7), Mercenary (рис. 8), Player (рис. 9), Yety (рис. 10).

Все они имеют свои компоненты и представлены в качестве префабов, для их дальнейшего использования. Причём для каждого уровня-реализации — свои шаблоны, внешне идентичные, но имеющие различия только в управляющем скрипте, который относится к тому или иному варианту, определенному на этапе проектирования.

Из основных настроек объекта использовались тэги — такие маркеры, которые используются для идентификации объектов. Существуют как стандартные маркеры, так пользовательские, которые можно добавлять с помощью менеджера тегов слоев (рис. 11). В приложении были использованы следующие тэги: Player (стандартный тэг), Wall, Enemy, NPC и Mercenary.

Обратим внимание на такой компонент как Circle Collider 2D [3] — коллайдер, используемый с 2D-физикой, являющийся кругом с заданной позицией и радиусом в координатах локального пространства объекта. Как можно заметить, у каждого объекта есть 2 таких коллайдера. Один из них является простым коллайдером, который при столкновении с коллайдером другого объекта использует физику 2D, другой — является триггером, который не

работает с физикой (устанавливается при активации поля «Is Trigger»). В данном случае он является границей области видимости конкретного объекта. Примеры представлены на рисунках 12, 13.

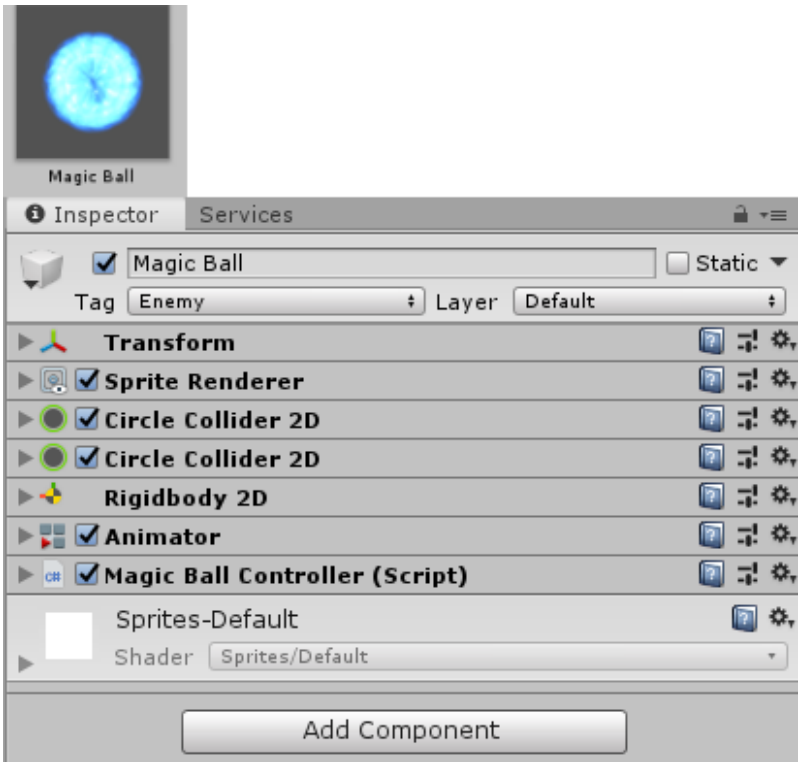

Рис. 7. Объект Magic Ball и его компоненты

Рис. 8. Объект Mercenary и его компоненты

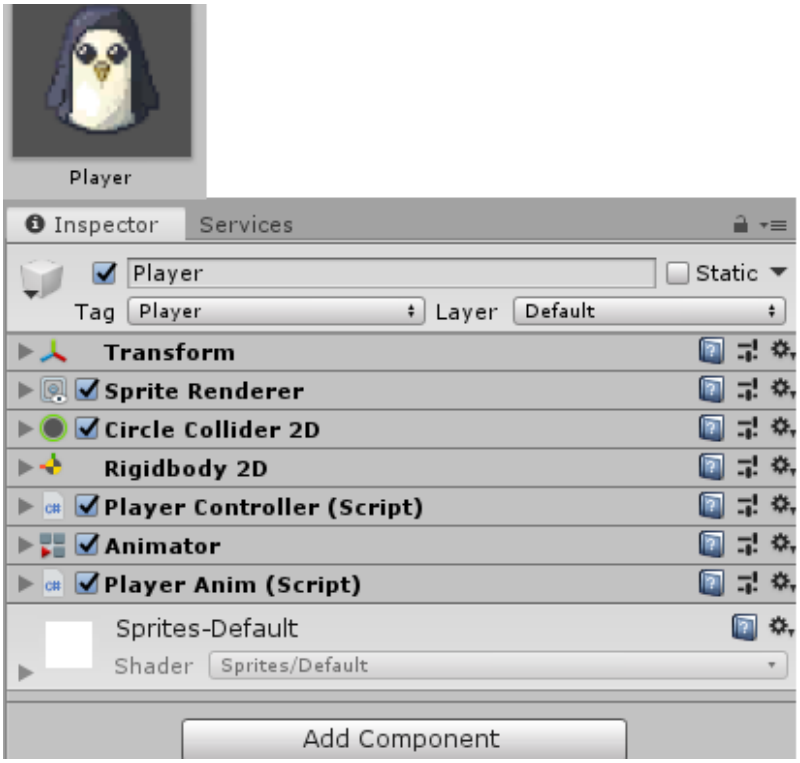

Рис. 9. Объект Player и его компоненты

Рис. 10. Объект Yeti и его компоненты

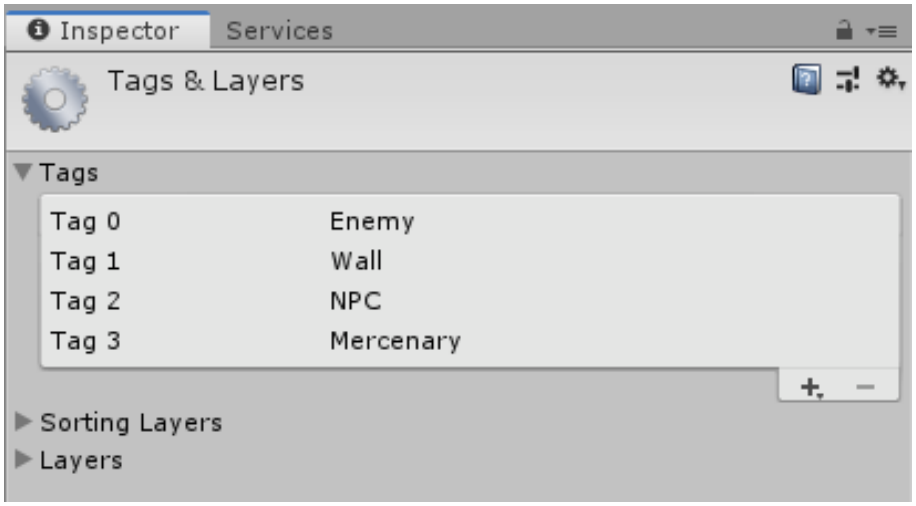

Рис. 11. Менеджер тэгов и слоев

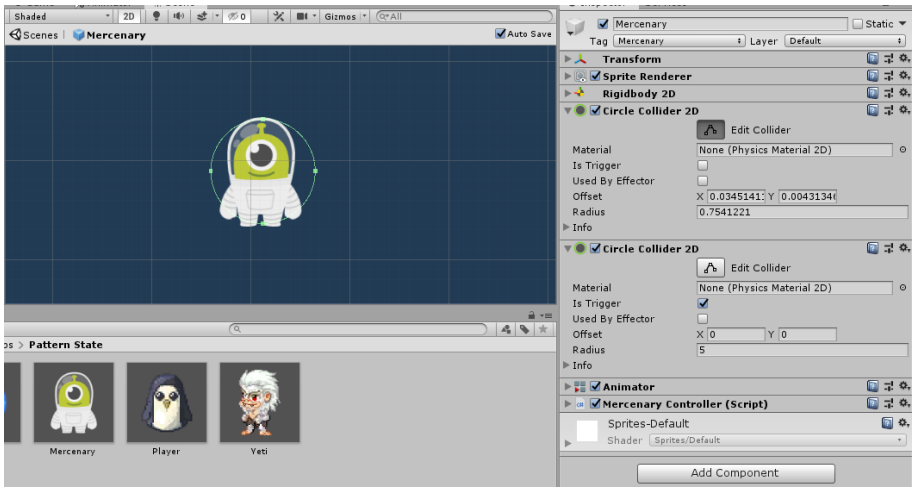

Рис. 12. Редактирование коллайдера

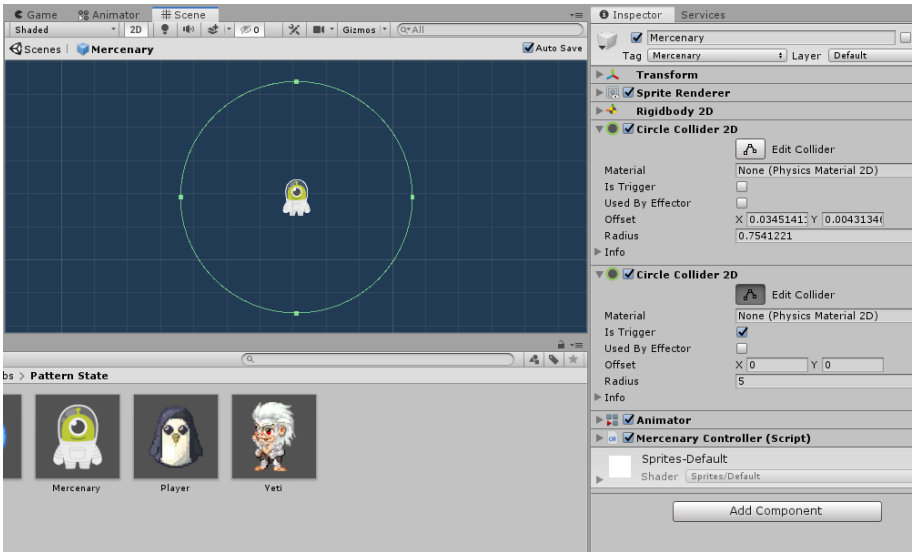

Рис. 13. Редактирование коллайдера-триггера

На основе триггеров строится система изменений поведения игровых объектов в разработанном приложении. При попадании в эту область того или иного инородного объекта, происходит смена состояния объекта, использующего этот коллайдер. На рисунке 14 представлена схема смена состояний, где:

- Если в поле видимости Mercenary попадает Magic Ball, то объект переходит в состояние атаки.
- Если в поле видимости Magic Ball попадает Mercenary, то объект переходит в состояние побега.
- Если в поле видимости Magic Ball попадает Player или Yeti, то объект переходит в состояние атаки.
- Если в поле видимости Yeti попадает Magic Ball, то объект переходит в состояние побега.
- По умолчанию у каждого объекта при старте устанавливается свое состояние движения. При потере из поля видимости объектов, на которые может произойти реакция, объект переходит в обычное состояние движения, которое устанавливается специальной функцией.

Игровое приложение, которое было разработано для исследования применения паттернов при разработке игр имеет 36 классов, из которых три класса является абстрактными, и два класса — классами интерфейса. Из-за специфики Unity, все скрипты по умолчанию наследуются от базового класса MonoBehaviour, который содержит в себе набор переменных, открытых функций и функций событий.

В ходе разработки игрового приложения было принято решение локализовать и вынести в отдельный класс общие поля и методы, причиной этому стало обилие разрабатываемых классов. Таким образом, у всех игровых объектов есть один общий родительский абстрактный класс BaseUnit.

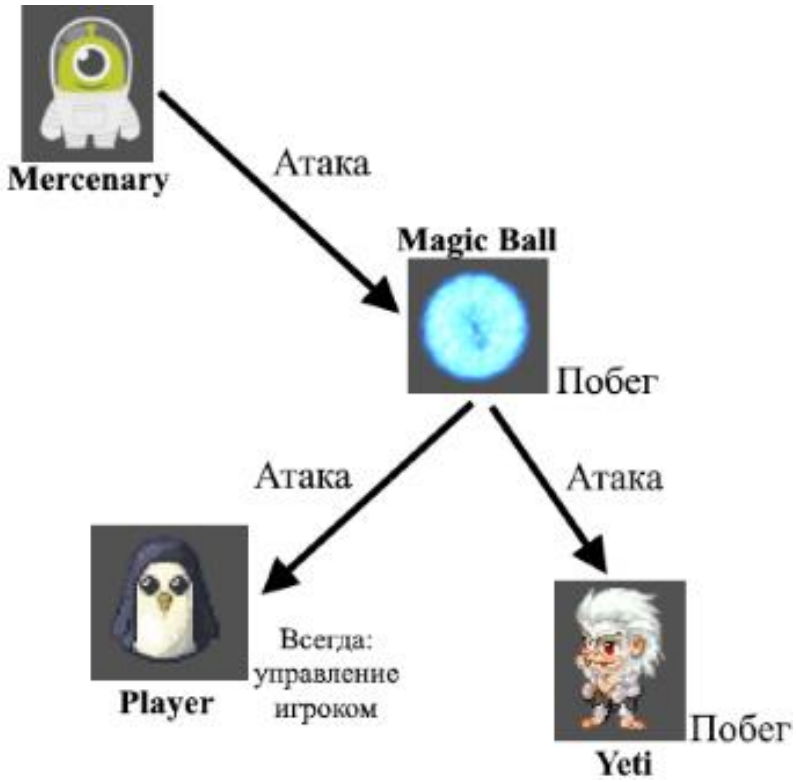

Рис. 14. Схема смены состояний

Для реализации паттерна «Состояние» была реализована иерархия классов, представленная на рисунке 15. Абстрактный класс UnitState, который наследуется от ранее рассмотренного BaseUnit, является родителем конкретных классов игровых объектов: MagicBallController, MercenaryController, PlayerController и YetiController.

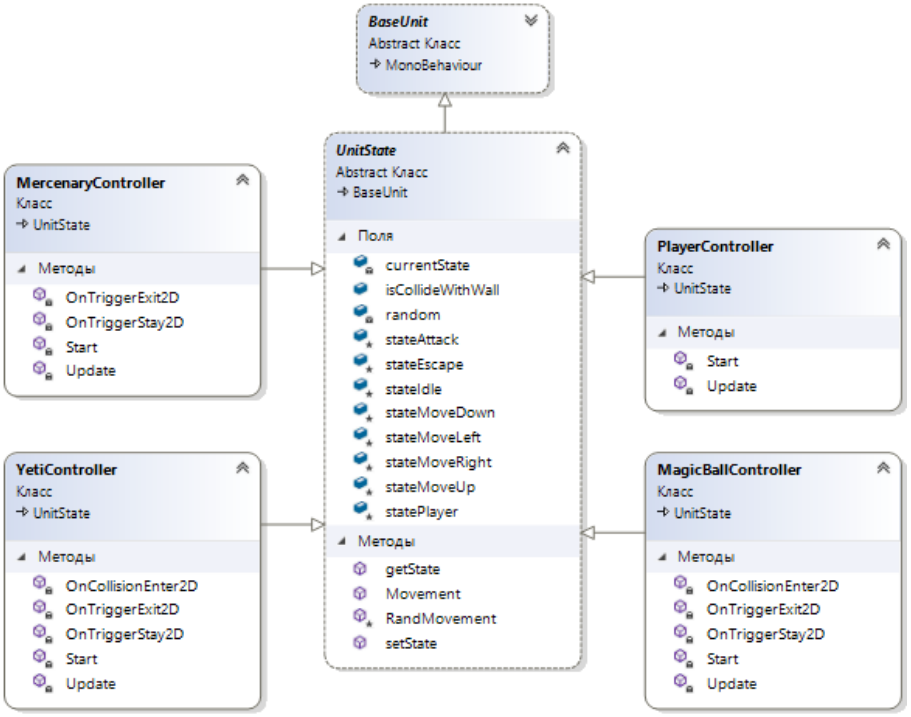

Рис. 15. Иерархия классов при использовании паттерна «Состояние»

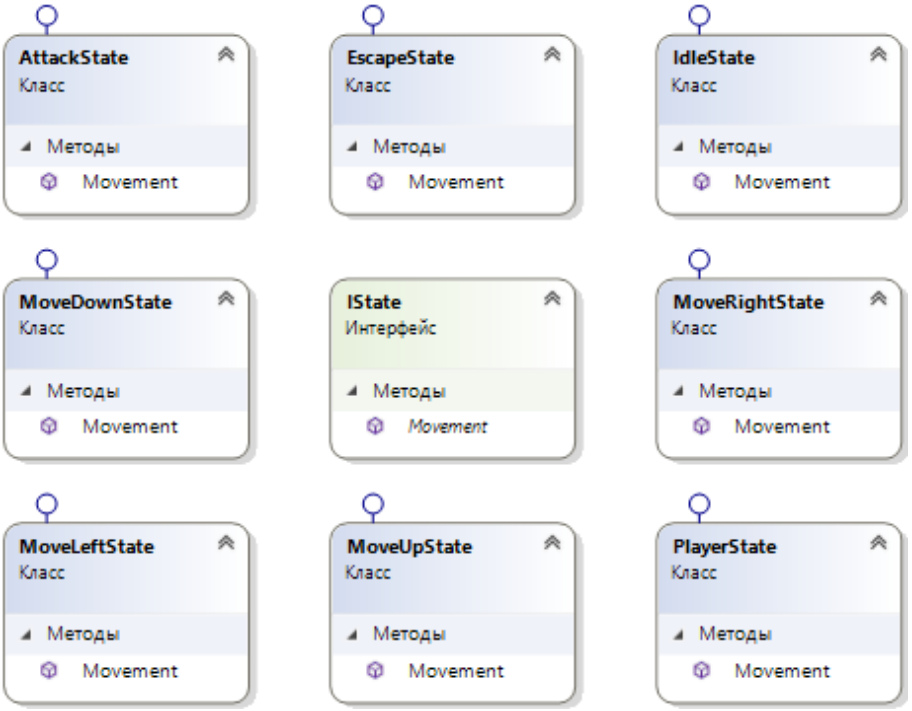

Рис. 16. Интерфейс и наследуемые классы

Класс-родитель содержит в себе следующие поля: экземпляр интерфейса IState и экземпляры всех возможных классов-состояний, определенных на этапе проектирования, которые наследуются от этого интерфейса (рис. 16). Кроме того, он содержит параметр, который является «внутренним» переключателем состояний, присущий данному паттерну. Из реализованных методов стоит отметить функцию Movement(), которая выполняет движение в зависимости от текущего состояния игрового объекта. С помощью методов setState(IState state) и getState() можно установить и получить текущее состояние объекта.Каждый класс-наследник содержит в себе набор функции событий:

- Start() — вызывается при старте работы скрипта.
- Update() — вызывается каждый раз перед отрисовкой кадра и перед расчётом анимации.
- OnCollisionEnter2D(Collision2D collision) — вызывается при контакте входящего коллайдера с коллайдером текущего объекта.
- OnTriggerStay2D(Collider2D collision) — вызывается, когда объект находится в пределах триггерного коллайдера, прикрепленного к текущему объекту.
- OnTriggerExit2D(Collider2D collision) — вызывается при выходе объекта из триггерной зоны коллайдера, прикрепленного к текущему объекту.

Рассмотрим функцию в объекте Magic Ball, на примере которой можно проследить изменение состояния:

```
private void OnTriggerStay2D(Collider2D collision)
{
     if (collision.gameObject.tag == "Player" ||
     collision.gameObject.tag == "NPC")
     {
          targetObject = collision.gameObject;
          currentSpeed = baseSpeed * 1.5f;
          setState(stateAttack);
     }
     if (collision.gameObject.tag == "Mercenary")
     {
          targetObject = collision.gameObject;
          currentSpeed = baseSpeed * 1.1f;
          setState(stateEscape);
     }
}
```

Если в пределах триггера объекта Magic Ball находится объект с тэгом «Player» или «NPC», то устанавливается состояние stateAttack с помощью метода setState(IState state), который в качестве параметра принимает экземпляр класса поведения. Если же находится объект с тэгом «Mercenary» — устанавливается состояние stateEscape с помощью того же метода setState(IState state).

Следующий фрагмент кода демонстрирует смену поведения на основе своего внутреннего параметра:

```
private void OnCollisionEnter2D(Collision2D collision)
{
     if (collision.gameObject.tag == "Wall")
     {
```

```
            isCollideWithWall = true;
        }
        if (collision.gameObject.tag == "Player" ||
        collision.gameObject.tag == "Mercenary")
        {
            Destroy(gameObject);
        }
    }
```

Если Magic Ball соприкасается с объектом, имеющим тэг «Wall», то происходит присвоение переменной булевого значения, на основе которой произойдет смена состояния. Предположим, что Magic Ball имел состояние MoveRightState, то есть двигался в правую сторону, где столкнулся со стеной, имеющей тэг «Wall» и у него изменилось поле isCollideWithWall. При очередном выполнении метода своего поведения, произойдёт проверка этого самого поля, который в данном случае выступает в качестве внутреннего фактора, влияющего на смену состояния. В результате произойдёт смена состояния на MoveLeftState прямо изнутри выполняющегося метода текущего поведения. Пример представлен ниже:

```
public class MoveRightState : IState
{
    public Vector2 Movement(UnitState unit, float speed,
    GameObject currentObject, GameObject targetObject)
    {
        if (unit.isCollideWithWall)
        {
            unit.setState(new MoveLeftState());
            unit.isCollideWithWall = false;
        }
    Vector2 direction = new Vector2(1f, 0);
    return direction.normalized * speed * Time.deltaTime;
    }
}
```

Реализации паттерна «Состояние» представлена на рисунке 17. Абстрактный класс UnitState, который наследуется от ранее рассмотренного BaseUnit, является родителем конкретных классов игровых объектов: MagicBallController, MercenaryController, PlayerController и YetiController. Класс-родитель содержит в себе следующие поля: экземпляр интерфейса IStrategy.

Все классы-состояния представлены в виде наследников интерфейса IStrategy и исполняют конкретные поведения (рис. 18). Из реализованных методов стоит отметить функцию PerformMove(), которая выполняет движение в зависимости от текущего состояния игрового объекта. С помощью метода setStrategy(IStrategy strategy), принимающего в качестве параметра новый класс поведения, устанавливается новое состояние объекта.

Классы, наследуемые от UnitStrategy, так же как и рассмотренные ранее классы-наследники UnitState, имеют функции событий.

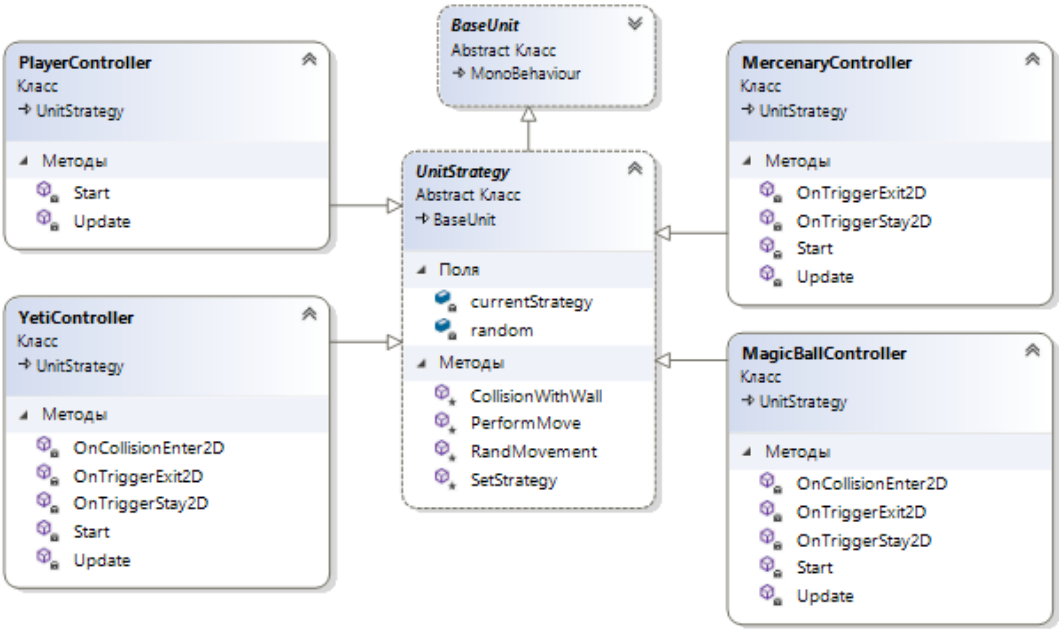

Рис. 17. Диаграмма классов при использовании паттерна «Стратегия»

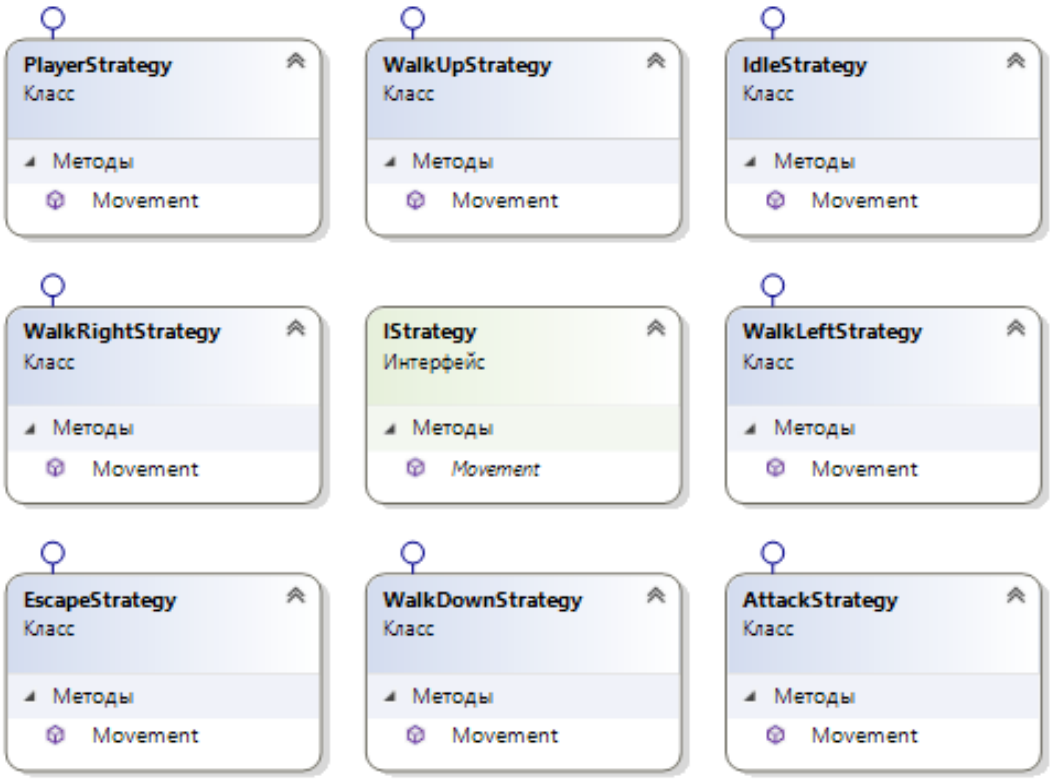

Рис. 18. Интерфейс и наследуемые классы

Переключения состояний происходят по тем же условиям, что и в предыдущей подсистеме. Так же рассмотрим фрагмент кода из класса MagicBallController, который демонстрирует смену поведения:

```
private void OnTriggerStay2D(Collider2D collision)
{
      if (collision.gameObject.tag == "Player" ||
      collision.gameObject.tag == "NPC")
      {
            targetObject = collision.gameObject;
            currentSpeed = baseSpeed * 1.5f;
            SetStrategy(new AttackStrategy());
      }
      if (collision.gameObject.tag == "Mercenary")
      {
            targetObject = collision.gameObject;
            currentSpeed = baseSpeed * 1.1f;
            SetStrategy(new EscapeStrategy());
      }
}
```

Если в пределах триггера объекта Magic Ball находится объект с тэгом «Player» или «NPC», то устанавливается новое состояние new AttackStrategy() с помощью метода setStrategy(IStrategy strategy), который в качестве параметра принимает новый экземпляр поведения определенного класса. Если же находится объект с тэгом «Mercenary» — устанавливается состояние new EscapeStrategy() с помощью того же метода setStrategy(IStrategy strategy).

Диаграмма классов игрового уровня без использования поведенческих паттернов проектирования представлена на рисунке 19.

Абстрактный класс NoPatternUnit, наследуемый от вышеописанного класса BaseUnit, является родителем всех классов персонажей, используемых на этом уровне. Все состояния описаны в виде перечисления:

```
enum StateEnum { Idle, WalkLeft, WalkRight, WalkUp,
WalkDown, Attack, Escape, Player }
```

Класс NoPatternUnit содержит в себе экземпляр перечисления, и методы setState(StateEnum state) и getState(), позволяющие устанавливать и получать текущее поведение объекта. Метод Movement() содержит в себе реализацию движения всех поведений, выбор которых осуществляется на основе конструкции switch, которая выполняет блок, совпадающий с текущим состоянием.

Абстрактный класс NoPatternUnit, наследуемый от вышеописанного класса BaseUnit, является родителем всех классов персонажей, используемых на этом уровне. Все состояния описаны в виде перечисления:

```
enum StateEnum { Idle, WalkLeft, WalkRight, WalkUp,
WalkDown, Attack, Escape, Player }
```

Класс NoPatternUnit содержит в себе экземпляр перечисления, и методы setState(StateEnum state) и getState(), позволяющие устанавливать и получать текущее поведение объекта. Метод Movement() содержит в себе реализацию движения всех поведений, выбор которых осуществляется на основе конструкции switch, которая выполняет блок, совпадающий с текущим состоянием.

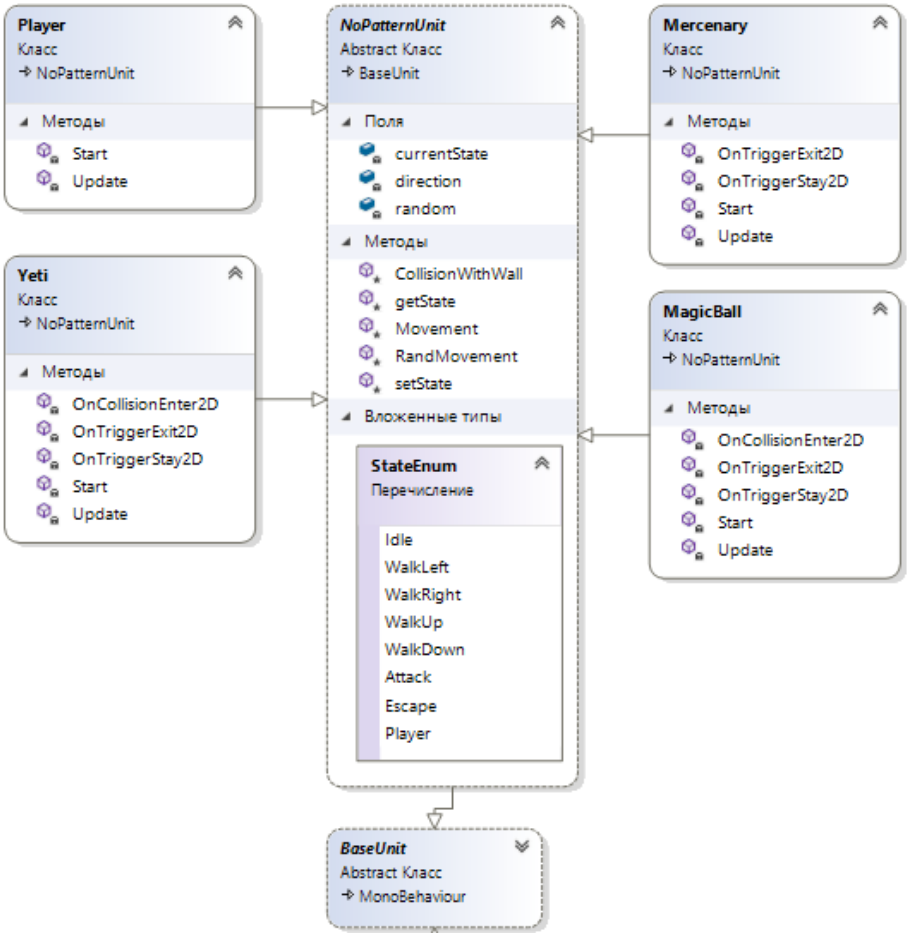

Рис. 19. Диаграмма классов без использования шаблонов

Для определения наиболее эффективного шаблона для создания игровых приложений, проанализирована реализация выбранных в этой работе паттернов. В целях сокращения объема занимаемого кодом, из фрагментов был удален код самой реализации, при этом сохранена основная структура методов. Фактически, в примере, где не используются паттерны, в методе Movement() реализованы сразу все поведения, что значительно увеличивает размер класса и объем кода и снижает его понимание и читабельность. При необходимости добавления новых поведений, возникнут сложности с пониманием и читабельностью кода, что влечет за собой проблемы с корректным написанием требуемого кода. В примерах реализаций с помощью поведенческих паттернов такая проблема отсутствует, так как каждое конкретное поведение — отдельный класс. Если потребуется добавить дополнительные поведения, то достаточно будет просто создать новый класс с его описанием этого самого поведения.

В качестве параметров оценки выбраны следующие показатели:

- Общее количество строк исполняемого кода (меньшее значение предпочтительнее).
- Глубина дерева наследования — наибольший путь по иерархии классов к данному классу (меньшее значение предпочтительнее).
- Общее количество реализованных классов.
- Взаимозависимость классов — определяет число классов, на которые есть ссылки. В расчет берутся уникальные классы из параметров, локальных переменных, возвращаемого типа, базового класса и атрибутов (меньшее значение предпочтительнее).

- Сложность организации циклов — определяет число ветвей (меньшее значение предпочтительнее).
- Индекс удобства поддержки — оценивает простоту обслуживания кода (большее значение предпочтительнее).

Метрика индекс удобства поддержки разработана специалистами из Carnegie Mellon Software Engineering Institute и рассчитывается по следующей формуле:

$$MI = \max\ (0, (171 - 5.2 * \ln HV - 0.23 * \mathrm{CC} - \frac{16.2 * \ln LoC * 100}{171})$$, где:

- HV — Halstead Volume, вычислительная сложность. Чем больше операторов, тем больше значение этой метрики;
- CC — Cyclomatic Complexity — сложность организации циклов, описана выше;
- LoC — количество строк кода.

Эта метрика может принимать значения от 0 до 100 и показывает относительную сложность поддержки кода. Чем больше значение этой метрики, тем легче поддерживать код.

Данные показателей 1, 2, 4–6 были определены автоматически с помощью встроенного в Visual Studio инструмента анализа — вычисление метрики кода. Все результаты представлены в таблице 1.

Таблица 1. Сравнительная таблица показателей

№	Критерий	Паттерн «Стратегия»	Паттерн «Состояние»	Без паттерна	Вес
1	Общее количество строк исполняемого кода	118	130	116	1
2	Глубина дерева наследования	7	7	7	3
3	Количество классов	14	14	6	2
4	Взаимозависимость классов	23	23	15	2
5	Сложность организации циклов	68	68	62	2
6	Индекс удобства поддержки	82	80	80	1
Результат:		431	441	383	

В сравнительной таблице, кроме ранее рассмотренных параметров оценки, так же есть веса, определяющие приоритет того или иного параметра. Интеграционная оценка высчитывается по следующей формуле:

$$\sum_{i=1}^{6} (\text{Значение критерия}_i * \text{вес}_i)$$

Из сравнительной таблицы можно сделать вывод, что использование поведенческих паттернов проектирования в текущем приложении не имеет никакой пользы. Хотя на самом деле это не так. Сравнивая параметры по отдельности видно, что разница несущественна, а где-то и вовсе зеркальная. Так же стоит взять во внимание ранее рассмотренные фрагменты кода, демонстрирующие преимущества использования поведенческих шаблонов. Рассматриваемые шаблоны позволяют максимально просто удалять и добавлять новые поведения игровым объектам. Примеры реализации поведенческих шаблонов представлены в видеоматериале.

Шаблон «Состояние» отличается от «Стратегии» тем, что в первом случае состояние объекта меняется за счёт какого-то заданного внутреннего показателя, во втором — от внешних условий. В работе в качестве условия перехода были триггеры персонажей, которые реагировали на нахождение или отсутствие внутри себя других объектов, поэтому в рамках работы наиболее эффективным оказался паттерн «Стратегия». Шаблон «Состояние» может подойти для более

узкоспециализированных игровых приложений, в которых состояния объектов могут изменяться, например, в зависимости от времени.

Литература

[1] Э. Фримен, Э. Робсон, К. Сьерра, Б. Бейтс. «Head First. Паттерны проектирования»; Издательский Дом ПИТЕР, 2018. 656 с.
[2] К. Дикинсон. «Оптимизация игр в Unity 5. Советы и методы оптимизации игровых приложений»; ДМК-Пресс, 2017. 306 с.
[3] А. Торн. «Искусство создания сценариев в Unity»; ДМК-Пресс, 2016. 360 с.

Implementation of Behavioral Design Patterns on The Example of a Game Application

V.V. Epp, I.S. Parshin

Penza State University, Russia

Abstract. In the age of active development of information technologies, optimization and acceleration of application development without loss of quality plays an important role. Programming is a non-deterministic process and there are a huge number of solutions for one task, which makes it difficult for the programmer to choose a particular solution. Especially in the field of game application development, where you need to create a high-performance application in a short time. This article will cover the development of the game in the Unity environment using behavioral templates (patterns) and without them. Design templates are only a General solution to a problem that can be supplemented and adjusted to the necessary requirements. The design pattern names, abstracts, and identifies key aspects of the overall solution structure, identifies participating classes and instances, their roles, and relationships and functions – which ultimately allows you to use it to create a reusable design.

Keywords: design patterns, Strategy pattern, State pattern, game development in Unity

References

[1] Je. Frimen, Je. Robson, K. S'erra, B. Bejts. «Head First. Patterny proektirovanija»; Izdatel'skij Dom PITER, 2018. 656 s.
[2] K. Dikinson. «Optimizacija igr v Unity 5. Sovety i metody optimizacii igrovyh prilozhenij»; DMK-Press, 2017. 306 s.
[3] A. Torn. «Iskusstvo sozdanija scenariev v Unity»; DMK-Press, 2016. 360 s.