**Постановка задачи**

Требуется написать игровое приложение, где игроки передвигают свои ракетки вертикально для защиты своих ворот. Со временем (с каждым ударом) скорость движения мячика постепенно увеличивается, и так игра усложняется.

**Цели исследования**

– графические *API*;

– принципы *SOLID* и ООП;

– шаблоны проектирования.

**Макет игрового приложения «*Ping*-*Pong*»**

На данном изображении можно увидеть макет игрового приложения *Ping*-*Pong*, который я взял за основу, так как он простой и удобный.

**Архитектура игрового приложения «*Ping*-*Pong*»**

После запуска приложения игрока встречает *UI*, то есть пользовательский интерфейс. Затем после нажатия на кнопку мы запускаем нашу бизнес логику, а она уже в свою очередь реализует *DirectX* и *GameObject*. А *GameObject* реализует *Entity*.

Бизнес логика – это класс *Game*, который связывает всю логику приложения.

**Иерархия классов *Entity***

В пространстве имён *Entity* находятся классы, реализующие логику объектов.

**Иерархия классов *GameObject***

В пространстве имён *GameObject* находятся классы, содержащие связку объектов, содержащих логику, и их спрайтов.

**Иерархия классов паттерна «Декоратор»**

Декоратор представляет структурный шаблон проектирования, который используется, когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта.

Декоратор позволяет модифицировать игрока, без порождения новых экземпляров классов. Он лишь переносит характеристики игрока из класса к классу, в зависимости от типа бонуса (обёртывает).

Достоинства:

– большая гибкость, чем у наследования;

– позволяет изменять характеристики на лету;

– можно изменять несколько характеристик сразу;

– позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Недостатки:

– \*трудно конфигурировать многократно обёрнутые объекты;

– много (обилие) крошечных классов.

**Иерархия классов фабричного метода**

Фабричный метод (*Factory* *Method*) – это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

Фабричный метод служит для переопределения создаваемых объектов.

Класс *BonusFactory* определяет, какой тип бонуса нам нужен и в своём методе *GetBonus* возвращает экземпляр одного из классов наследников генератора в зависимости от типа бонуса. А класс *Generator* генерирует (создаёт) нужный нам тип бонуса и возвращает его в своём методе *GetGenerator*.

Достоинства:

– избавляет класс от привязки к конкретным классам бонусов;

– выделяет код генерации бонусов в одно место;

– упрощает добавление новых бонусов в приложение;

– \*реализует принцип открытости/закрытости.

Недостатки:

– может привести к созданию больших параллельных иерархий классов, так как для каждого бонуса надо создать свой подкласс создателя;

**Классы, реализующие *DirectX***

Данное трио классов реализуют отображение графики. Класс *DX*2*D* описывает базовую функциональность для реализации графики. Класс *Drawer* представляет логику отображения игровых объектов на экране. Класс *InputManager* реализует подключение клавиатуры.

**Служебные классы**

Класс *Game* связывает компоненты графики и логики объектов. В классе *PhysicsEngine* реализуется логика физики игры. Класс *TimeHelper* предназначен для работы со временем.

**Спрайты проекта**

На данном слайде представлены спрайты, которые использовались для создания игрового приложения «*Ping*-*Pong*».

**Анализ результатов**

– реализован паттерн «Декоратор» для изменения характеристик игрока;

– реализован фабричный метод для различных видов бонусов.

**Вывод**

С разработанной архитектурой приложения и иерархией классов было написано игровое приложение, выполняющее поставленную задачу.