**ПЗ 03.01 Работа с поведенческими паттернами**

1. **Описание предметной области**

Задача заключается в разработке генератора отчетов о продажах, который выполняет три основных шага:

1. **Проверка данных** – обеспечение корректности данных перед обработкой.
2. **Форматирование** – подготовка данных для представления в нужном формате.
3. **Сохранение отчета** – сохранение отчета в нужных форматах (например, PDF и CSV).

Для реализации этих шагов используется паттерн **Шаблонный метод (Template Method)**, который позволяет определить основной алгоритм выполнения операции и делегировать конкретные шаги подклассам. В данной задаче основной шаги обработки данных, такие как проверка, форматирование и сохранение, реализуются через абстрактные методы, которые переопределяются в дочернем классе.

Классы и их назначение:

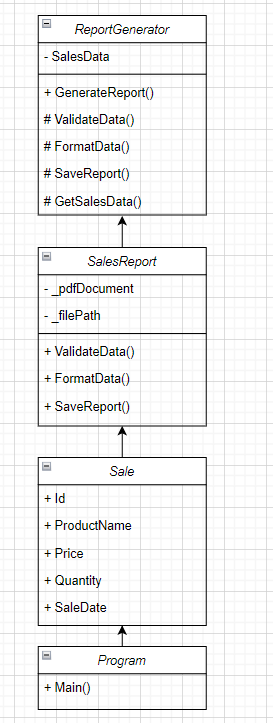
ReportGenerator — абстрактный класс, который задает общий алгоритм генерации отчета с использованием шаблонного метода.

SalesReport — конкретная реализация алгоритма генерации отчета, в которой реализуются конкретные шаги, такие как проверка данных, форматирование (с использованием библиотеки iTextSharp для создания PDF) и сохранение отчета в форматах CSV и PDF.

Sale - класс, представляющий данные о продаже, включая такие поля как ID товара, название, цену, количество и дату продажи.

Program - Класс, который служит для выполнения программы (точка входа). Он вызывает метод GenerateReport() из SalesReport.

1. **UML диаграмма классов реализованного проекта**



1. **Краткий вывод по использованию паттерна в текущей задаче**

В данной задаче паттерн «Шаблонный метод» решает проблему унифицированного процесса генерации отчёта при возможности изменять отдельные стадии этого процесса в наследниках. Базовый класс ReportGenerator определяет общий алгоритм (шаги в методе GenerateReport()), а конкретный класс SalesReport переопределяет детали проверки, форматирования и сохранения отчёта. Благодаря этому подходу легко поддерживать разные форматы вывода и дополнительные проверки, при этом не ломая и не дублируя общую логику формирования отчёта. Использование этого паттерна повышает гибкость и расширяемость проекта, так как добавление нового типа отчета или изменение существующих шагов потребует только переопределения нужных методов в дочерних классах.

1. **Листинг кода**

Класс, представляющий данные о продаже

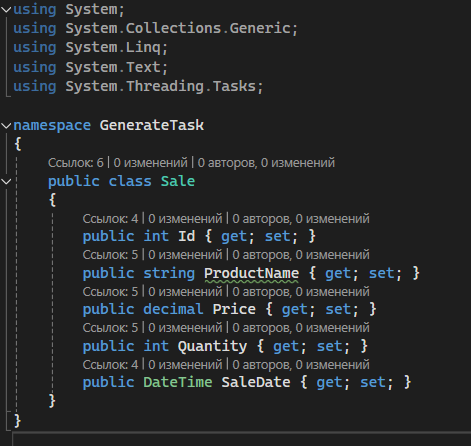


Рисунок 1 - Класс Sale

Абстрактный класс ReportGenerator



Рисунок 2 - КлассReportGenerator

Конкретный класс SalesReport



Рисунок 3 - Класс SalesReport

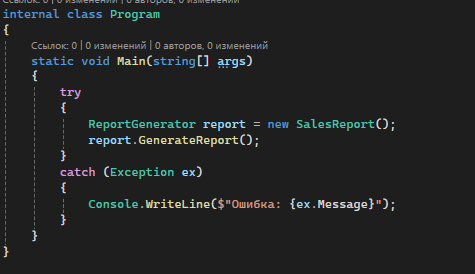


Рисунок 4 - Главный класс программы

1. **Описание предметной области**

Проект представляет собой простую консольную игру, в которой игрок управляет персонажем, двигает его по экрану, атакует врагов и собирает предметы. Основные элементы игры включают:

Персонаж игрока: управляется стрелками на клавиатуре.

Атака: реализована с помощью клавиши пробела.

Враги: каждый враг имеет здоровье и наносит урон при столкновении с игроком.

Предметы: игрок может собирать предметы, повышающие его счет.

Цель игры: победить 3 врагов или проиграть, если здоровье игрока достигает нуля.

Игра реализована в виде консольного интерфейса с использованием консольных команд и взаимодействий.

Классы и их назначение:

Game — основной класс игры, который управляет логикой и состоянием игрового процесса. В нем реализованы методы для перемещения игрока и атаки, а также для отображения игрового поля и проверки состояния игры. Класс отслеживает здоровье игрока и врага, счет и количество побед, генерирует врагов и предметы, а также контролирует завершение игры.

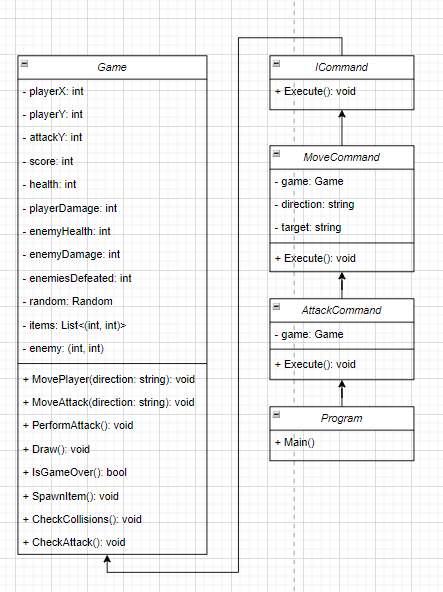
ICommand — интерфейс, который определяет общий контакт для всех команд.

MoveCommand — класс, реализующий команду перемещения персонажа или атаки. В зависимости от переданных параметров, команда выполняет движение игрока или перемещение зоны атаки по игровому полю. Каждый объект MoveCommand привязан к определенному направлению (вверх, вниз, влево, вправо) и цели (персонаж или атака).

AttackCommand — класс, реализующий команду атаки. Когда эта команда выполняется, она инициирует атаку игрока, которая наносит урон врагу. Команда также проверяет, попала ли атака в врага, и, если здоровье врага достигает нуля, враг побежден.

Program — класс, являющийся точкой входа в программу. Он запускает игру, отображает инструкции для пользователя, создает объекты команд для управления игрой (например, перемещение и атака), а также управляет циклом игры, который продолжается до тех пор, пока игрок не проиграет или не победит.

1. **UML диаграмма классов реализованного проекта**



1. **Краткий вывод по использованию паттерна в текущей задаче**

В данной задаче используется паттерн Command, который позволяет инкапсулировать запросы (команды) в отдельные объекты. Это дает возможность параметризовать объекты с различными действиями, которые могут быть выполнены на основе ввода пользователя. В этой игре команды, такие как движение персонажа, атака или перемещение объекта атаки, реализованы как отдельные классы, реализующие интерфейс ICommand. Это позволяет легко добавлять новые команды, не изменяя код главного цикла игры.

Основные преимущества использования паттерна Command в данном контексте:

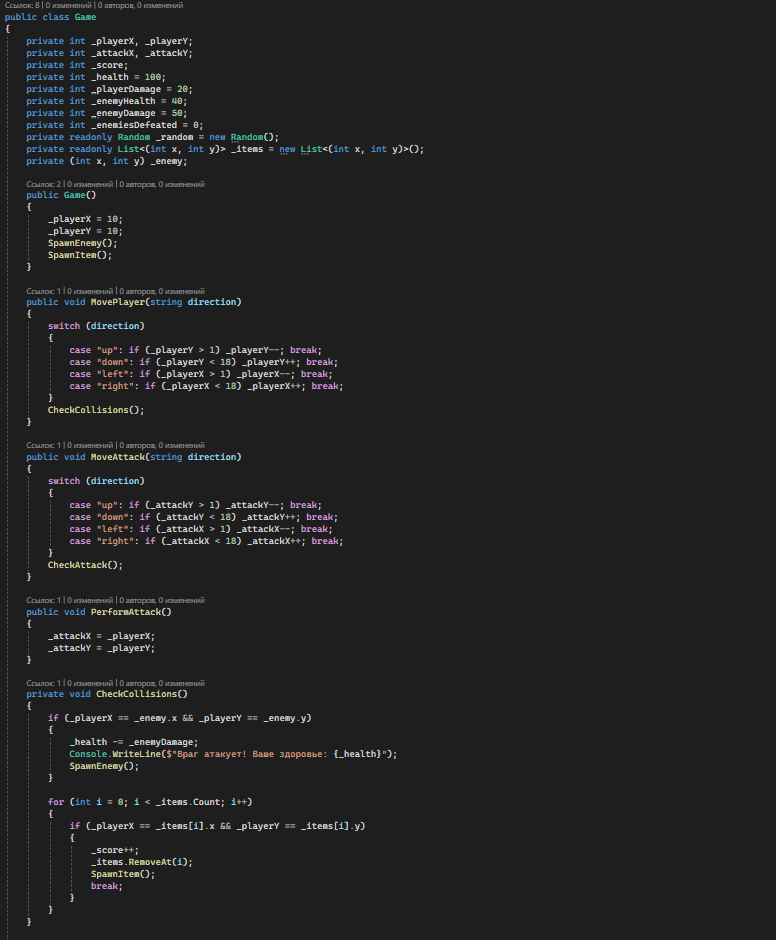
- Отделение логики команд от обработки ввода: Логика выполнения команд изолирована в отдельных классах, что упрощает тестирование и расширение функционала.

- Легкость добавления новых команд: Чтобы добавить новую команду (например, новую атаку или действие), достаточно создать новый класс, реализующий интерфейс ICommand, и добавить его в словарь команд.

- Гибкость: Код игры остаётся гибким, поскольку обработка нажатий клавиш сводится к вызову метода Execute() соответствующего объекта команды.

1. **Листинг кода**

Класс Game



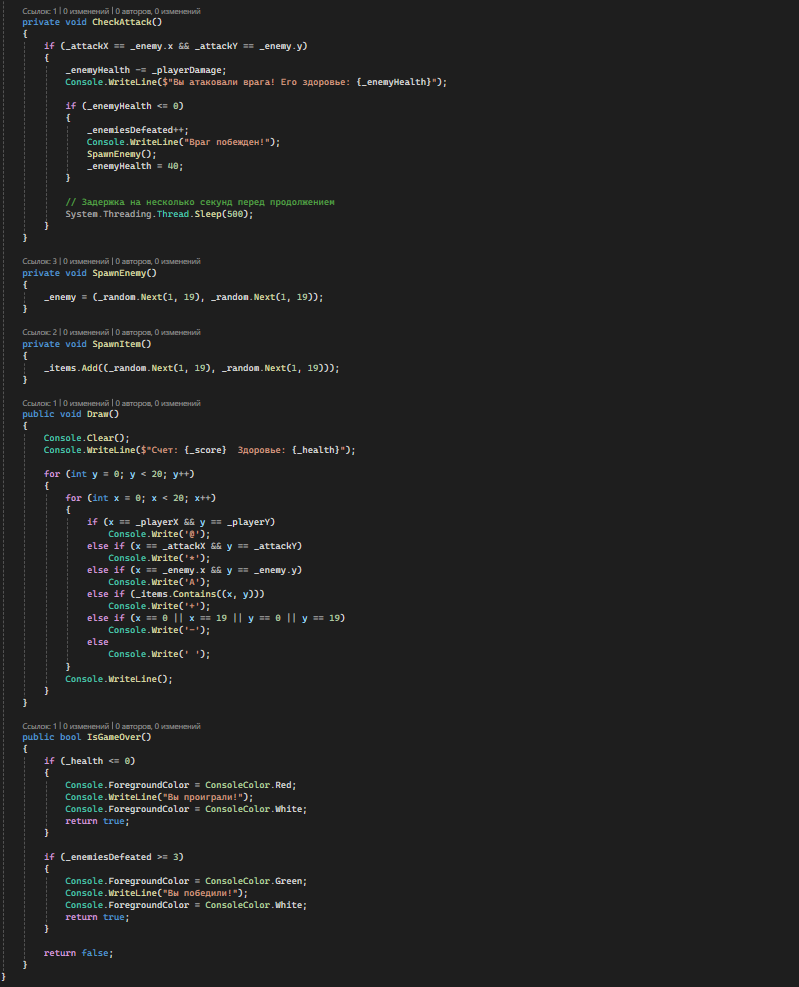


Рисунок 5 - Класс Game

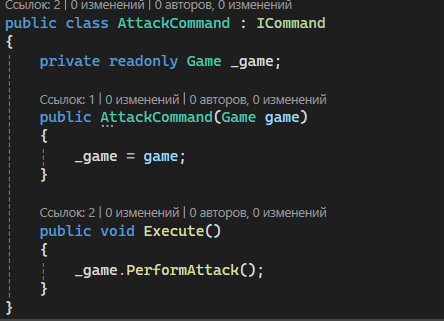


Рисунок 6 - Класс AttackCommand

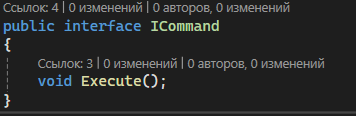


Рисунок 7 - Класс ICommand

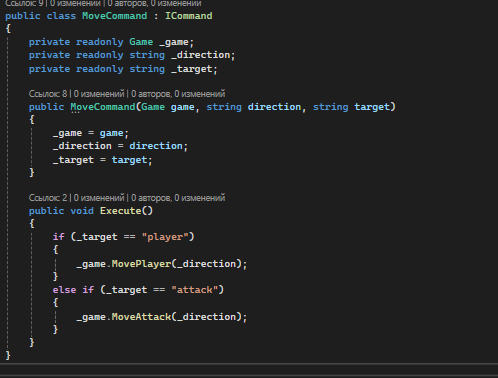


Рисунок 8 - Класс MoveCommand

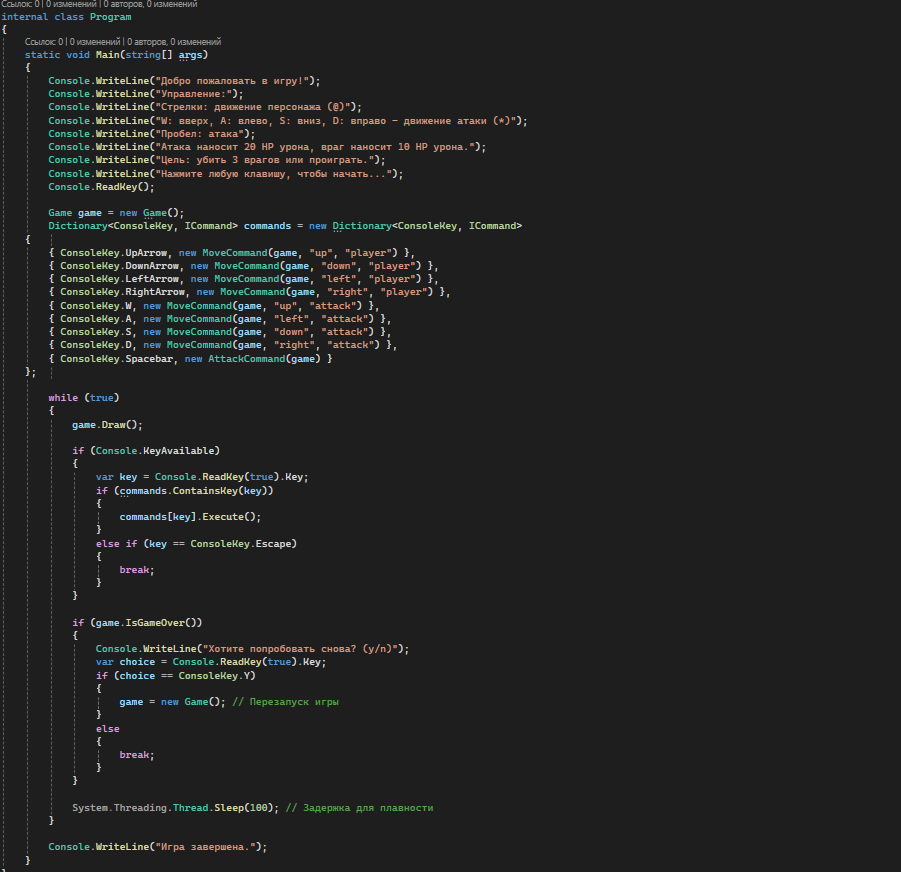


Рисунок 9 - Класс Program

#### Описание предметной области

Предметная область данного проекта — это календарь для добавления, редактирования и просмотра событий. В рамках задачи частично реализован календарь, который использует паттерн **Состояние** (State Pattern), позволяющий менять доступные действия в зависимости от текущего состояния (добавление события, редактирование события, просмотр событий).

Календарь состоит из различных состояний, каждое из которых разрешает или запрещает определенные действия пользователя:

**- Добавление события** — позволяет добавлять новые события в календарь.

- **Редактирование события** — в перспективе должен позволять изменять уже добавленные события.

**- Просмотр событий** — позволяет просматривать все добавленные события.

Паттерн "Состояние" помогает эффективно управлять состоянием объекта, упрощая логику изменения доступных действий.

#### 3. Краткий вывод по использованию паттерна в текущей задаче

В данном проекте паттерн **Состояние** помогает управлять различными возможностями взаимодействия пользователя с календарем. Каждый класс состояния реализует определенные действия, разрешенные в конкретном режиме.

В данном примере приложение «Календарь» может находиться в одном из трёх состояний:

- AddEventState (добавление),

- EditEventState (редактирование),

- ViewEventState (просмотр).

Каждое состояние реализует общий интерфейс ICalendarState, но имеет своё поведение для операций добавления, редактирования, удаления и просмотра событий. Таким образом, если код переходит в состояние «добавления», то только метод AddEvent работает, а остальные методы выбрасывают исключения, сигнализируя о недопустимости этих операций для данного режима. Аналогично, в состоянии «просмотра» разрешено удалять и просматривать, но добавление или редактирование недоступны.

Благодаря этому:

\* Логика доступных действий не «захардкожена» во множестве if/else, а разделена по разным классам состояний.

\* Добавление новых состояний или изменение логики в одном из состояний не влияет на остальные, что упрощает поддержку и расширение кода.

\* Переключение состояния в коде (calendar.SetState(new EditEventState());) даёт явный контроль над тем, какие операции сейчас доступны пользователю.

Паттерн **Состояние** идеально подходит для задач, где поведение объекта зависит от его состояния, и каждое состояние требует своей логики для выполнения операций.