МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Объектно-ориентированное программирование»

Тема: Шаблонные классы

Студентка гр. 3383 _____ Земерова С.Н.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Оглавление

1. Цель работы

Цель: разработать обобщённые (шаблонные) классы **управления игрой** и **отрисовки**, позволяющие подменять реализацию ввода/рендера без изменения кода ядра игры; обеспечить консольную и GUI-конфигурации..

2. Задание

a) Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.

b) Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.

c) Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.

d) Реализовать класс, отвечающий за отрисовку поля.

Примечание:

- Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания
- После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду.

- Для представления команды можно разработать системы классов или использовать перечисление enum.
- Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”
- При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две команды, что на одну команду не назначено две клавиши.

3. Архитектурные решения

3.1. Пакетная структура

```
src/
├── core/     (ЛР №1)          Основная логика игры
├── additional/           Вспомогательные компоненты
│   ├── Other.h/cpp           Вспомогательные функции
│   └── ShipCoordinateExceptions.h/cpp   Исключения
├── abilities/      Система способностей (ЛР №2)
└── controlGame/  (ЛР №3)    Классы для процесса игры
    ├── Game.h/cpp       Игровой процесс
    ├── GameState.h/cpp       Состояние игра
    ├── Result.h        Структуры для результатов атак и способностей
    ├── FileHandler.h       Для безопасной работы с файлами
    ├── GameSettings.h/cpp        Настройки игры
    └── controllerGame/   (ЛР №4)  Шаблонные обёртки и конфигурации
        ├── Command.h                 Игровая команда
        ├── GameController.h/cpp          Основной контроллер игры
        ├── InputHandler.h                Интерфейс обработки ввода
        ├── IRendererStrategy.h           Интерфейс отображения
        ├── Renderer.h                    Адаптер для рендереров
        ├── SoundManager.h/cpp            Управление звуком
        ├── console/   Случай игры для консоли
        │   ├── TerminalInputHandler.h/cpp    Консольный ввод
        │   └── ConsoleRenderer.h/cpp          Консольное отображение
        └── GUI/    Случай игры для графического представления GUI
            ├── GUIInputHandler.h/cpp                Графический ввод
            └── GUIRenderer.h/cpp             Графическое отображение
```

3.2. Принципы

- **Инверсия зависимостей:** Классы GameController и Renderer зависят от абстракций (InputHandler, IRenderStrategy), а не от конкретных реализаций. Ядро игры (Game, GameState) ничего не знает о существовании контроллеров и рендереров.

- **Принцип подстановки:** TerminalInputHandler, GUIInputHandler, ConsoleRenderer, GUIRenderer могут быть использованы везде, где ожидаются их базовые интерфейсы, не нарушая работу программы.

- **Разделение интерфейсов:** InputHandler отвечает только за ввод, IRenderStrategy — только за отрисовку. Интерфейсы не перегружены.

- **Шаблонный метод (Template Method) и Стратегия (Strategy):** Шаблонные классы GameController и Renderer используют *статический полиморфизм* (на этапе компиляции) для подмены стратегий ввода и вывода, что эффективнее динамического.

4. Классы и интерфейсы

4.1. Интерфейсы

- InputHandler

  Методы: bool initialize(); std::unique_ptr<Command> getCommand(); неблокирующий getCommand() возвращает команду или nullptr.
- IRenderStrategy

  Методы: initialize(), render(const Game&), сервисные коллбеки onAttackResult(...), onAbilityResult(...), showMessage(...).

4.2. Шаблонные обёртки

- Renderer<StrategyT>

  Хранит StrategyT strategy_. Делегирует все вызовы методу одноимённой стратегии. Позволяет компоновкой выбрать консоль/GUI.
- GameController<InputHandlerT, RendererT>

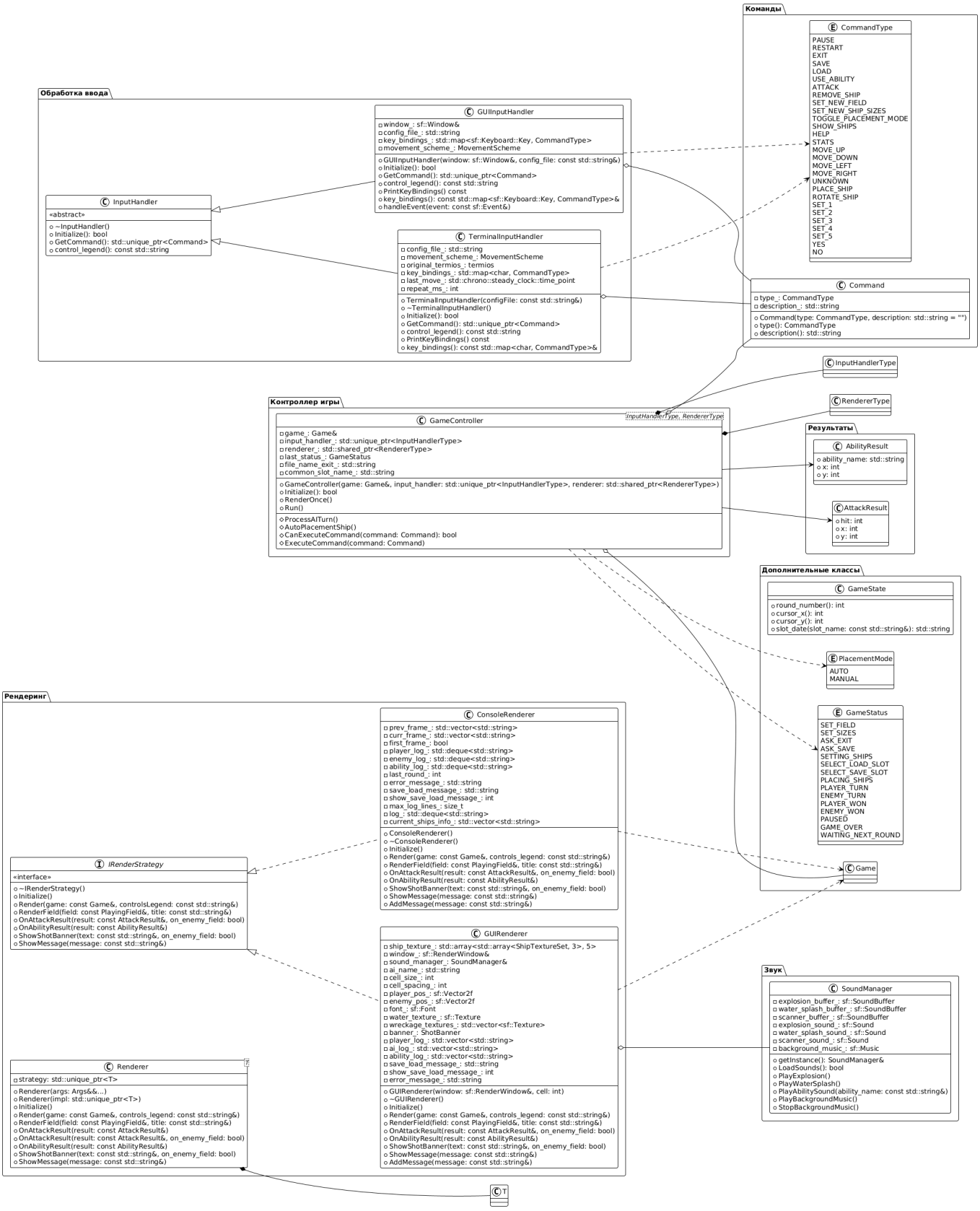
  Хранит Game& game_, unique_ptr<InputHandlerT> input_, unique_ptr<RendererT> renderer_.

  Цикл: отрисовать → взять команду → выполнить (с проверкой статуса) → спец-ветка для ENEMY_TURN.

4.3. Консольная конфигурация

1. TerminalInputHandler
   - файл keybinds.cfg: пары ключ=команда; ключи SPACE/ESC/UP/DOWN/LEFT/RIGHT и одиночные символы;
   - дефолт: WASD + SPACE + ESC и пр.;
   - валидация: конфликты авторазруливаются;
2. ConsoleRenderer
   - двойная буферизация списка строк: строим curr_frame_, выводим только различающиеся строки (перемещение курсора ANSI);
   - подсветка курсора на поле врага; лог внизу; легенда управления.

5. Диаграмма классов



Обоснование типов связей:

Композиция (Composition):

GameController → InputHandlerType: Контроллер полностью владеет обработчиком ввода через unique_ptr, время жизни обработчика зависит от контроллера

GameController → RendererType: Контроллер полностью владеет рендерером через shared_ptr, рендерер уничтожается с контроллером

Renderer → T: Рендерер полностью владеет стратегией рендеринга через unique_ptr, стратегия не существует без рендерера

Агрегация (Aggregation):

GameController → Game: Контроллер использует игру через ссылку, но не владеет ею, игра может существовать отдельно

GameController → Command: Контроллер временно использует команды через unique_ptr, команды создаются и уничтожаются в процессе работы

GUIRenderer → SoundManager: Рендерер использует менеджер звуков через ссылку на синглтон, звуковой менеджер существует независимо

InputHandler → Command: Обработчики ввода временно создают команды для передачи контроллеру

Ассоциации (Associations):

GameController → AttackResult/AbilityResult: Контроллер временно создает объекты результатов для передачи данных между компонентами

Зависимости (Dependencies):

GameController → GameStatus/PlacementMode: Контроллер зависит от перечислений для управления логикой игры

Renderers → Game: Рендереры зависят от игры для получения данных для отрисовки

InputHandlers → CommandType: Обработчики ввода зависят от типов команд для парсинга и создания команд

6. Реализация

6.1. Вспомогательные структуры

6.1.1. Класс InputHandler

Назначение:

InputHandler — это абстрактный базовый класс, определяющий интерфейс для обработки пользовательского ввода в игре. Он задаёт контракт для всех конкретных обработчиков ввода (например, для консольного или графического интерфейса), обеспечивая единообразный способ получения команд от пользователя. Основная задача — принимать ввод (например, нажатия клавиш или события SFML) и преобразовывать его в команды Command, которые затем обрабатываются игровой логикой.

Основные особенности:

- Абстрактный интерфейс: Определяет три чисто виртуальных метода (Initialize и GetCommand), которые должны быть реализованы в производных классах, таких как GUIInputHandler.
- Неблокирующий контракт: Метод GetCommand возвращает не более одной команды за вызов, что позволяет интегрировать обработку ввода в игровой цикл без блокировки.

Методы:

virtual ~InputHandler() = default

- *Описание:* Виртуальный деструктор, обеспечивающий корректное освобождение ресурсов в производных классах.
- *Функциональность:*
- Не выполняет дополнительных действий, но позволяет производным классам (например, GUIInputHandler) безопасно освобождать свои ресурсы.
- Гарантирует полиморфное удаление объектов через указатель на базовый класс.

virtual bool Initialize() = 0

- *Описание:* Чисто виртуальный метод для инициализации обработчика ввода.

- *Функциональность:*

  - Должен быть реализован в производных классах для настройки обработчика (например, загрузки конфигурации клавиш или инициализации SFML-окна).

  - Возвращает true при успешной инициализации, false в случае ошибки (например, если не удалось загрузить конфигурацию).

virtual std::unique_ptr<Command> GetCommand() = 0

- *Описание:* Чисто виртуальный метод для получения следующей команды на основе пользовательского ввода.

- *Функциональность:*

  - Реализует неблокирующий контракт, возвращая не более одной команды (std::unique_ptr<Command>) за вызов.

  - В производных классах (например, GUIInputHandler) обрабатывает события ввода (клавиши, мышь, закрытие окна) и преобразует их в команды Command, соответствующие текущей конфигурации.

- Возвращает nullptr, если в текущий момент нет доступных команд.

virtual const std::string control_legend() = 0

- *Описание:* Чисто виртуальный метод, возвращающий актуальную легенду управления в виде строки.

- Функциональность:

– Используется UI для отображения подсказки игроку (внизу экрана, в меню помощи).

– Возвращает человекочитаемый текст вида: "WASD — перемещение, Space — выстрел, E — способность, H — помощь"

– Учитывает текущий ремаппинг клавиш (если он реализован в наследнике).

6.1.2. Класс Command

Назначение:

Класс Command представляет собой контейнер для игровой команды, содержащий её тип (CommandType) и необязательное описание. Он используется для передачи пользовательских действий (например, атака, перемещение курсора, пауза) от обработчика ввода (InputHandler) к игровой логике. Перечисление CommandType определяет все возможные команды, включая специфические для графического интерфейса (например, MOVE_UP, ATTACK).

Основные особенности:

- Перечисление CommandType: Содержит полный набор игровых команд, таких как управление курсором (MOVE_UP, MOVE_DOWN, и т.д.), атака (ATTACK, SHOOT), использование способностей (USE_ABILITY), сохранение/загрузка (SAVE, LOAD) и другие действия.
- Гибкость: Поддерживает добавление описания для команд, что полезно для отладки или пользовательских интерфейсов.
- Простота: Лёгкая структура с минимальной функциональностью, обеспечивающая передачу типа команды и её контекста.

Перечисление CommandType:

Группа	Команда	Описание
Управление игрой	PAUSE	Поставить игру на паузу / открыть меню паузы
	RESTART	Перезапустить текущий раунд
	EXIT	Полный выход из игры
Сохранение и загрузка	SAVE	Открыть меню сохранения
	LOAD	Открыть меню загрузки сохранённой игры
Расстановка и редактирование флота	SET_NEW_FIELD	Изменение размера игрового поля
	SET_NEW_SHIP_SIZES	Изменение состава флота
	TOGGLE_PLACEMENT_MODE	Переключить режим расстановки: ручная / авто
	SHOW_SHIPS	Показать/скрыть список размещенных кораблей
	REMOVE_SHIP	Удалить выбранный корабль (ручная расстановка)
	PLACE_SHIP	Подтвердить размещение текущего корабля
	ROTATE_SHIP	Повернуть корабль (горизонталь / вертикаль)
Выбор размера корабля	SET_1	Выбор_1
	SET_2	Выбор_2
	SET_3	Выбор_3
	SET_4	Выбор_4
	SET_5	Выбор_5
Действия в бою	ATTACK	Выполнить выстрел по

Группа	Команда	Описание
		позиции курсора
	USE_ABILITY	Активировать выбранную способность
Управление курсором	MOVE_UP	Курсор вверх
	MOVE_DOWN	Курсор вниз
	MOVE_LEFT	Курсор влево
	MOVE_RIGHT	Курсор вправо
Меню подтверждения	YES	Подтвердить действие
	NO	Отменить действие
Вспомогательные	HELP	Показать справку
	STATS	Показать статистику игрока и ИИ
Служебные	UNKNOWN	Команда не распознана (обработка ошибок ввода)

Методы:

Command(CommandType type, const std::string& description = "")

- *Описание:* Конструктор класса, создающий объект команды с указанным типом и необязательным описанием.

- *Параметры:*

  - type: Тип команды из перечисления CommandType.

  - description: Строковое описание команды (по умолчанию пустое).

- *Функциональность:*

  - Инициализирует поля type_ и description_ переданными значениями.

- Позволяет создавать команды, такие как Command(CommandType::ATTACK) для атаки или Command(CommandType::EXIT, "Выход из игры") с описанием для отладки.

CommandType type() const

- *Описание:* Возвращает тип команды.

- *Функциональность:*

  - Возвращает значение type_, позволяя игровой логике определить, какое действие необходимо выполнить (например, MOVE_UP для перемещения курсора вверх).

std::string description() const

- *Описание:* Возвращает описание команды.

- *Функциональность:*

  - Возвращает строку description_, которая может использоваться для вывода в интерфейсе или логах (например, для отображения сообщения «Игрок выбрал выход»).

6.1.3. Класс IRenderStrategy

Назначение:

IRenderStrategy — это абстрактный базовый класс, определяющий интерфейс для стратегий рендеринга игрового состояния. Он задаёт контракт для визуализации игры, включая отображение игровых полей, обработку результатов атак и способностей, а также отображение сообщений и баннеров. Конкретные реализации (например, GUIRenderer) используют этот интерфейс для рендеринга через SFML или другие средства.

Основные особенности:

Абстрактный интерфейс: определяет методы для инициализации, рендеринга игры, полей, обработки событий (атаки, способности) и отображения сообщений.

Гибкость отображения: поддерживает рендеринг полей игрока и противника, визуальных эффектов (например, взрывы, баннеры) и текстовых сообщений.

Обработка событий: обрабатывает результаты атак (AttackResult) и способностей (AbilityResult), позволяя визуализировать их на поле игрока или противника.

Короткие баннеры: поддерживает отображение временных сообщений (баннеров) над полями, указывающих, кто выполняет действие (игрок или противник).

Методы:

virtual ~IRenderStrategy() = default

Описание: Виртуальный деструктор, обеспечивающий корректное освобождение ресурсов в производных классах.

– Позволяет безопасно удалять объекты через указатель на базовый класс.

virtual void Initialize() = 0

Описание: Чисто виртуальный метод для инициализации ресурсов рендеринга.

– Должен быть реализован в производных классах для загрузки шрифтов, текстур, звуков и других ресурсов.

virtual void Render(const Game& game, const std::string& controlsLegend) = 0

Описание: Чисто виртуальный метод для рендеринга текущего состояния игры.

– Отрисовывает все элементы игры: поля, интерфейс, статус, эффекты. – Принимает объект Game и строку с легендой управления (controlsLegend) для отображения подсказки.

virtual void RenderField(const PlayingField& field, const std::string& title) = 0

Описание: Чисто виртуальный метод для отрисовки игрового поля.

– Отрисовывает поле (PlayingField) с указанным заголовком (например, «Игрок», «Противник»).

virtual void OnAttackResult(const AttackResult& result)

```
{
    OnAttackResult(result, true);
}
```

Описание: Метод по умолчанию, упрощающий вызов.

– Перенаправляет вызов к перегруженной версии, считая атаку на поле противника (on_enemy_field = true).

virtual void OnAttackResult(const AttackResult& result, bool on_enemy_field) = 0

Описание: Чисто виртуальный метод для обработки результата атаки с указанием стороны.

– Визуализирует эффект атаки на нужном поле (true — противник, false — игрок).

virtual void OnAbilityResult(const AbilityResult& result) = 0

Описание: Чисто виртуальный метод для обработки результата использования способности.

– Принимает AbilityResult и отображает соответствующий эффект (анимация, звук).

virtual void ShowShotBanner(const std::string& text, bool on_enemy_field) = 0

Описание: Чисто виртуальный метод для отображения короткого баннера с текстом над полем.

– Показывает временное сообщение (например, «Игрок стреляет») над нужным полем.

virtual void ShowMessage(const std::string& message) = 0

Описание: Чисто виртуальный метод для отображения общих текстовых сообщений.

6.2. Шаблонный рендерер Renderer<T>

Назначение

Renderer<T> — тонкая компилируемая обёртка (compile-time полиморфизм) над конкретной стратегией визуализации T, удовлетворяющей контракту IRenderStrategy. Он сам наследует IRenderStrategy, чтобы его можно было использовать «как рендерер» при статической подстановке конкретной стратегии (консольной или GUI).

Ключевые идеи

Инверсия зависимостей: ядро игры (Game) не знает про конкретный способ отрисовки.

Zero-overhead: вместо виртуальных вызовов — инлайнинг вызовов в T.

Безопасность владения: реализация хранится в std::unique_ptr<T>.

Конструкторы

template <typename... Args>

explicit Renderer(Args&&... args)

: strategy(std::make_unique<T>(std::forward<Args>(args)...)) {}

Действие: Создаёт объект стратегии типа T с передачей любых аргументов.

explicit Renderer(std::unique_ptr<T> impl)

: strategy(std::move(impl)) {}

Действие: Принимает готовый объект стратегии через std::move.

Методы рендеринга

void Initialize() override

Действие: Инициализирует систему рендеринга.

Вызывает: strategy->Initialize().

void Render(const Game& game, const std::string& controls_legend) override

Действие: Отрисовывает полное игровое состояние.

Вызывает: strategy->Render(game, controls_legend).

void RenderField(const PlayingField& field, const std::string& title) override

Действие: Отображает конкретное игровое поле.

Вызывает: strategy->RenderField(field, title).

Методы обработки событий

void OnAttackResult(const AttackResult& result) override

Действие: Обрабатывает результат атаки (по умолчанию — на поле противника).

Вызывает: strategy->OnAttackResult(result) → внутри стратегии вызывается версия с on_enemy_field = true.

void OnAttackResult(const AttackResult& result, bool on_enemy_field) override

Действие: Обрабатывает результат атаки с указанием поля.

Вызывает: strategy->OnAttackResult(result, on_enemy_field).

void OnAbilityResult(const AbilityResult& result) override

Действие: Обрабатывает результат использования способности.

Вызывает: strategy->OnAbilityResult(result).

Методы интерфейса

void ShowShotBanner(const std::string& text, bool on_enemy_field) override

Действие: Показывает информационный баннер о выстреле.

Вызывает: strategy->ShowShotBanner(text, on_enemy_field).

void ShowMessage(const std::string& message) override

Действие: Отображает системное сообщение.

Вызывает: strategy->ShowMessage(message).

Внутреннее состояние (приватное поле)

std::unique_ptr<T> strategy - хранит конкретную реализацию стратегии рендеринга.

Примечание: Renderer<T> ничего не решает сам, а только строго перенаправляет вызовы выбранной стратегии T.

Инварианты

- strategy != nullptr после конструктора.
- T обязан реализовывать все методы IRenderStrategy.

6.3. Шаблонный контроллер GameController<InputHandlerType, RendererType>

Назначение Шаблонный класс GameController<InputHandlerType, RendererType> представляет собой **основной контроллер игры**, который координирует взаимодействие между обработчиком ввода, системой рендеринга и игровой логикой. Класс запрашивает у InputHandlerType команды, проверяет их допустимость по текущему GameStatus, вызывает соответствующие методы Game и сообщает RendererType о результатах (баннеры, эффекты, перерисовка).

Поля класса (приватные)

Game& game_

std::unique_ptr<InputHandlerType> input_handler_

std::shared_ptr<RendererType> renderer_

GameStatus last_status_ = GameStatus::GAME_OVER — для возврата из меню

std::string file_name_exit_ = "exit_save"

std::string common_slot_name_ = "slot

Конструктор

GameController(Game& game, std::unique_ptr<InputHandlerType> input_handler, std::shared_ptr<RendererType> renderer)

Действие: создаёт контроллер игры, принимая ссылку на игровой объект, уникальный указатель на обработчик ввода и общий указатель на рендерер.

Использует семантику перемещения (std::move) для эффективной передачи владения ресурсами.

Основные методы

bool Initialize()

Действие: выполняет первоначальную настройку всех компонентов системы.
– Проверяет наличие инициализированных обработчика ввода и рендерера.

– Последовательно вызывает input_handler_->Initialize() и renderer_→Initialize().

– Возвращает false при обнаружении критических ошибок инициализации.

void RenderOnce()

Выполняет однократную отрисовку текущего состояния.

Вызывает: renderer_->Render(game_, input_handler_->control_legend()).

void Run()

Действие: запускает главный игровой цикл, который продолжает выполняться, пока game_.IsRunning().

– Сначала выполняет первичную отрисовку (RenderOnce()).

– Отслеживает изменения состояния игры и позиции курсора.

– Если статус — ENEMY_TURN → вызывает ProcessAITurn().

– Иначе — неблокирующе опрашивает input_handler_→GetCommand().

– Перерисовывает только при наличии изменений.

– Использует std::this_thread::sleep_for(16ms).

Вспомогательные методы

void ProcessAITurn()

Действие: обрабатывает ход компьютерного противника.

– Показывает баннер ("AI стреляет...", on_enemy_field = false).

– Выполняет game_.MakeAIMove() и передаёт результат в renderer_->OnAttackResult(result, false).

– Включает задержки для визуальной плавности.

bool CanExecuteCommand(const Command& command) const

Действие: проверяет возможность выполнения команды в текущем состоянии игры.

Учитывает все статусы игры, включая ASK_SAVE, SET_SIZES, SELECT_LOAD_SLOT и т.д.

Выступает в роли главного предохранителя от логических ошибок.

v***oid ExecuteCommand(const Command& command)***

Действие: выполняет команду, предварительно проверяя её допустимость.

Особенности:

– Полная обработка всех CommandType (включая SET_1–SET_5, YES/NO).

– Правильная реакция на атаку: game_.AttackShip() → баннер → OnAttackResult → если ход перешёл к ИИ → ProcessAITurn().

– Корректное перемещение курсора с учётом границ поля противника.

– Обработка исключений (try/catch) — все ошибки домена отображаются через ShowMessage,.

"

6.4. Консольный ввод TerminalInputHandler

Назначение

TerminalInputHandler — обработчик пользовательского ввода для консольной версии игры «Морской бой».

Он отвечает за:

- настройку терминала под неблокирующий режим (termios, POSIX);
- загрузку пользовательского конфига keybinds.cfg и автоматическое исправление ошибок в нём;
- автоматическую конфигурацию движения по выбранной схеме WASD или ARROWS;
- подавление «дребезга» перемещения (одно срабатывание не чаще чем раз в N мс);
- корректную обработку ESC-последовательностей (стрелки, функциональные клавиши).

Что делает класс

Настройка терминала

- Использует POSIX termios (режимы ICANON, ECHO, VMIN=0, VTIME=0).
- RAII — при уничтожении объекта настройки терминала всегда восстанавливаются.

Загрузка и обработка конфига

Файл keybinds.cfg:

- допускает любые пробелы и комментарии ;

- понимает специальные названия клавиш: SPACE, ESC, ENTER, TAB, BACKSPACE, F1–F12;
- запрещает вручную назначать команды движения (MOVE_UP… MOVE_RIGHT);
- допускает схему движения movement_scheme = WASD или ARROWS;
- при конфликте клавиши с WASD-движением автоматически переназначает команду на ближайшую безопасную клавишу;
- при невозможности использовать клавишу — ищет свободную приоритетно:
    - для SET_1…SET_5: сперва цифры, затем буквы;
    - для всех остальных — сперва буквы, затем цифры.

Если конфиг содержит ошибки — они исправляются, исправленный вариант автоматически сохраняется.

Логика валидации

Метод ValidateOrFixBindings():

- проверяет все обязательные команды;
- если команда отсутствует — подбирает свободную клавишу;
- возвращает true, если все клавиши корректны и подправлять не пришлось;
- если фиксации были — выводит информацию и пересохраняет конфиг.

Схема движения

Схема выбирается в конфиге:

- WASD — клавиши движения w/a/s/d зарезервированы и недоступны для других команд;
- ARROWS — движение выполняется стрелками (ESC [A/B/C/D), w/a/s/d свободны.

Метод SetMovementBindings():

- очищает все старые биндинги движения;
- при WASD назначает: w=UP, s=DOWN, a=LEFT, d=RIGHT;
- при ARROWS движения идут только через ESC-последовательности, в карту биндингов не добавляются.

Конструктор и деструктор

Конструктор

explicit TerminalInputHandler(const std::string& configFile = "keybinds.cfg");

- загружает имя конфига;
- по умолчанию схема движения — WASD.

Деструктор

~TerminalInputHandler() override

- автоматически восстанавливает исходный режим терминала (RAII).

Основные методы

bool Initialize()

Выполняет полную инициализацию:

1. настраивает терминал (SetUpTerminal());
2. пытается загрузить конфиг (loadFromFile());
3. при неудаче — создаёт безопасный дефолт (SetDefaultBindings());
4. проверяет и исправляет конфигурацию (ValidateOrFixBindings());
5. после исправления, но *до движения* — сохраняет конфиг при необходимости;
6. назначает схему движения (SetMovementBindings());
7. выводит список команд (PrintKeyBindings()).

std::unique_ptr<Command> GetCommand()

Поддержка:

- неблокирующего POSIX-чтения через read();
- ESC-последовательностей:
    - стрелки: ESC [A/B/C/D;
    - функциональные клавиши F1–F4 через ESC OP/OQ/OR/OS;
- WASD-движения при выбранной схеме;
- авто-дребезг (repeat_ms_).

При движении возвращает команду лишь если прошло достаточно времени.

Методы работы с конфигурацией

bool loadFromFile()

- читает конфиг;
- парсит movement_scheme;

- запрещает назначение команд движения вручную;
- автоматически перекидывает команды, попавшие на WASD при выбранной схеме движения;
- сохраняет исправленный файл, если были конфликты.

void SetDefaultBindings()

- заполняет карту безопасными назначениями;
- включает SPACE=ATTACK, ESC=PAUSE и т.п.;
- не назначает движение — это делает SetMovementBindings().

bool ValidateOrFixBindings()

- проверяет наличие критических команд;
- подбирает свободные клавиши без конфликтов с движением;
- логика различает SET_1–SET_5 и остальные команды.

bool SaveDefaultConfig() const

- пересохраняет набор команд (включая movement_scheme);
- форматирует вывод через KeyToDisplayString().

Вспомогательные методы

ParseCommand() / CommandToString()

- регистронезависимое сопоставление enum ↔ строка.
- поддерживает все команды, включая SET_, YES/NO и др.

ParseTerminalKey()

- распознаёт:
    - одиночные символы;
    - SPACE, ESC, ENTER, TAB, BACKSPACE;
    - F1–F12.

KeyToDisplayString()

- возвращает корректное название клавиши для вывода и записи в конфиг.

CreateMoveCommand()

- применяет анти-дребезг, ограничивая частоту движения.

Приоритет обработки ввода

1. ESC → проверка на стрелки → проверка на F1–F4 → одиночный ESC = PAUSE.
2. Если_active_ WASD — сначала проверка движения.
3. Затем поиск клавиши в key_bindings_.
4. Если ничего не распознано — возврат nullptr.

6.5. Консольный рендер ConsoleRenderer

Назначение

Класс ConsoleRenderer — реализация стратегии рендеринга для консольной версии игры «Морской бой». Обеспечивает быстрый, без мерцания вывод в терминал с помощью дифференциального обновления экрана, поддержкой ANSI-цветов, скрытием курсора, логами событий и красивой легендой управления.

Конструктор и деструктор

ConsoleRenderer()

- Инициализирует внутренние буферы кадров (prev_frame_, curr_frame_), флаги и логи.
- first_frame_ = true, ansi_ready_ = true.

~ConsoleRenderer() override

- RAII: показывает курсор обратно и сбрасывает все цветовые атрибуты (\x1b[0m).

Основные методы

void Initialize() override

Алгоритм:

1. Полная очистка экрана (\x1b[2J\x1b[H)
2. Скрытие системного курсора (\x1b[?25l) — для «чистого» отображения

void Render(const Game& game, const std::string& controls_legend) override

Главный метод отрисовки:

1. Проверяет текущий GameStatus и при необходимости перехватывает отрисовку:
   - SET_FIELD → RenderFieldSizeSelection()
   - SET_SIZES → RenderShipSizeSelection()

- ASK_EXIT, ASK_SAVE, SETTING_SHIPS → RenderQuestionDialog()
- SELECT_SAVE/LOAD_SLOT → RenderDialogFiles()
- HELP → RenderHelp()

2. При смене раунда — очищает логи (ClearLog())

3. Обновляет информацию о кораблях (RenderShipsInfo()), статистику (RenderStats())

4. Формирует новый кадр через BuildFrame(game, controls_legend)

5. Выводит только изменившиеся строки через PrintDiff()

6. Сохраняет текущий кадр как предыдущий

Методы обработки игровых событий

void OnAttackResult(const AttackResult& result) override

void OnAttackResult(const AttackResult& result, bool on_enemy_field) override

Команда: визуализация результата выстрела

Что делает: добавляет цветное сообщение в лог (свой/чужой поле)

Коды результата (result.hit):

- < 0 → «Недействительный выстрел» (жёлтый)
- 0 → «Промах» (красный)
- 1 → «Попадание» (голубой)
- 2 → «Корабль потоплен» (зелёный)

void OnAbilityResult(const AbilityResult& result) override

Команда: визуализация результата способности

Что делает: добавляет в лог строку вида

Способность: Радар @(5,3) или Способность: Двойной выстрел

void ShowShotBanner(const std::string& text, bool on_enemy_field) override

Команда: отображение баннера выстрела

Пример: [Поле противника] Вы стреляете...

void ShowMessage(const std::string& message) override

Команда: системное сообщение (ошибки, сохранение, загрузка)

→ попадает в общий лог или в специальное поле ошибок

void AddMessage(const std::string& message)

Внутренний метод — распределяет сообщения по логам:

- Сообщения с «Ваше» → player_log_ (действия противника по вам)
- Ошибки → error_message_ (красным)
- Сохранение/загрузка → временное сообщение (фиолетовое, 2 кадра)
- Остальное → enemy_log_ (ваши действия)

Внутренние методы отрисовки

std::vector<std::string> BuildFrame(const Game& game, const std::string& controls_legend)

Главный строитель кадра — собирает всё в один вектор строк:

- Заголовок игры, раунд, счёт
- Координаты курсора (A01, x=0, y=0)
- Легенда символов (. ~ O d S)
- Два поля (своё + противника) через RenderBaseFields()
- Логи действий (слева — действия противника, справа — ваши)
- Текущая способность
- Статус игры (цветной: Ваш ход, Ход противника и т.д.)
- Легенда управления (от TerminalInputHandler)
- Статистика (если включена)
- Ошибки и уведомления о сохранении

char CellGlyph(const PlayingField& field, int x, int y, bool reveal_ships) const

Определяет символ и цвет клетки

O - Целый корабль (своё поле)

d - Повреждённый сегмент

 - Уничтоженный корабль

~ - Промах (вода)

S - Корабль при сканировании

. - Неизвестная клетка

→ Своё поле: reveal_ships = true → всё видно

→ Поле противника: только то, что открыто выстрелами или способностью

void PrintDiff(const std::vector<std::string>& next)

Дифференциальный вывод — в текущей реализации упрощён:

→ ClearScreenFull() + полный вывод (но всё равно без мерцания благодаря двойной буферизации и flush())

Управление терминалом

- ***ClearScreenFull()*** → \x1b[2J\x1b[H

- ***HideCursor(bool)*** → \x1b[?25l/h

- ***MoveCursor(row, col)*** → позиционирование

Особые режимы отрисовки:

SET_FIELD - Выбор размера поля (10×10 … 14×14)

SET_SIZES - Создание собственного флота

ASK_EXIT, ASK_SAVE - Диалог Да/Нет

SELECT_SAVE/LOAD_SLOT - Список слотов с датами

HELP - Полный текст помощи (блокирует ввод)

Особенности реализации

- Полная поддержка всех статусов игры

- Два отдельных лога: действия противника и игрока

- Автоочистка логов при смене раунда

- Временные уведомления (сохранено/загружено)
- Подсветка курсора
- Статус игры
- Легенда управления
- Статистика с цветами

6.6. GUI-рендер GUIRenderer (SFML)

Назначение:

Класс GUIRenderer реализует интерфейс IRendererStrategy и отвечает за визуализацию игры с использованием библиотеки SFML. Он обеспечивает адаптивную вёрстку, отображение игровых полей (игрока и противника), визуальные эффекты и звуковое сопровождение через SoundManager.

Основные особенности:

- Адаптивная вёрстка: динамически подстраивает масштаб отображения под размер окна.
- Два поля: отображает поле игрока и противника с рамками, заголовками и подсветкой курсора.
- Эффекты: отображение кораблей, маркеров попаданий и промахов, баннеры с сообщениями.
- Звуки: разные игровые события сопровождаются соответствующими звуками через SoundManager.
- Дополнительные панели: статистика, лог действий, информация о способностях, помощь.

Методы

Конструктор

GUIRenderer(sf::RenderWindow& window, int cell)

- Описание: Конструктор класса, инициализирующий объект GUIRenderer.
- Параметры:
  - window: Ссылка на окно SFML, используемое для отрисовки.
  - cell: Размер клетки игрового поля (30, 40 или 50 пикселей).
- Функциональность:
  - Сохраняет ссылку на окно SFML (window_) для последующей отрисовки.
  - Инициализирует SoundManager через вызов SoundManager::getInstance().

- Устанавливает размер клетки (cell_size_) и вычисляет расстояние между клетками (cell_spacing_ = cell + 5).

- Устанавливает начальное имя противника (ai_name_ = "AI").

Деструктор

~GUIRenderer()

- Описание: Деструктор класса, определён как default.

- Функциональность: Автоматически освобождает ресурсы, управляемые объектами SFML.

void Initialize()

- Описание: Инициализирует ресурсы, необходимые для отрисовки (шрифты, текстуры).

- Функциональность:

- Вызывает метод LoadResources() для загрузки шрифтов, текстур воды, кораблей и обломков.

- Если загрузка ресурсов завершается неудачно, выводит сообщение об ошибке в консоль.

void Render(const Game& game, const std::string& controls_legend)

- Описание: Основной метод рендеринга игрового кадра.

- Функциональность:

- Очищает окно тёмно-синим цветом (sf::Color(30, 30, 60)).

- Рассчитывает позиции и размеры элементов интерфейса.

- Настраивает адаптивное масштабирование через sf::View.

- Вызывает методы рендеринга различных компонентов:

- RenderGameTitle() - заголовок игры с номером раунда

- RenderScore() - счёт игры

- RenderField() - поля игрока и противника

- RenderCursor() - курсор

- RenderGameStatus() - статус игры
- RenderShipsInfo() - информация о кораблях
- RenderInput() - панель ввода
- RenderAbilitiesInfo() - информация о способностях
- RenderBanners() - баннеры
- RenderLog() - лог действий
- RenderSaveLoadMessage() - сообщения о сохранении/загрузке
- RenderControlsLegend() - легенда управления
- RenderStats() - статистика
- И другие вспомогательные методы для диалоговых окон

void RenderField(const PlayingField& field, const std::string& title)

- Описание: Отрисовывает игровое поле (игрока или противника).
- Функциональность:
  - Определяет, является ли поле полем противника, сравнивая title с ai_name_.
  - Вызывает RenderFieldFrame для отрисовки рамки и заголовка поля.
  - Перебирает все клетки поля, для каждой:
    - Рисует текстуру воды
    - Для поля противника отображает маркеры сканирования, попаданий и промахов
    - Для поля игрока отображает корабли и маркеры

void RenderCell(const PlayingField& field, CellState state, int ship_size, const std::tuple<int, SegmentState, Orientation>& segmentInfo, const sf::Vector2f& position, bool is_enemy_field)

- Описание: Отрисовывает отдельную клетку игрового поля.
- Функциональность:
  - Рисует текстуру воды в указанной позиции.

- Если состояние клетки - корабль, вызывает RenderShipPart для отображения части корабля.

void RenderShipPart(const sf::Vector2f& position, int ship_size, const std::tuple<int, SegmentState, Orientation>& segmentInfo, bool fullyDestroyed)

- Описание: Отрисовывает сегмент корабля или обломки.
- Функциональность:
  - Извлекает из segmentInfo индекс сегмента, состояние и ориентацию.
  - Если корабль полностью уничтожен и доступны текстуры обломков, использует их.
  - Иначе выбирает текстуру (целую или повреждённую) из ship_texture_.
  - Устанавливает текстуру спрайта и позицию.
  - Если ориентация вертикальная, поворачивает спрайт на 90 градусов.

void RenderCursor(const Game& game)

- Описание: Отрисовывает курсор на поле.
- Функциональность:
  - Получает координаты курсора из объекта game.
  - Создаёт полупрозрачный жёлтый прямоугольник с обводкой.
  - Устанавливает позицию прямоугольника на соответствующем поле.

void RenderGameStatus(const Game& game)

- Описание: Отображает текущий статус игры.
- Функциональность:
  - На основе значения status выбирает текст сообщения и цвет текста для различных состояний игры.

bool LoadResources()

- Описание: Загружает все ресурсы (шрифты, текстуры).
- Функциональность:

- Пытается загрузить шрифт из списка путей.
- Вызывает LoadShipTextures() для загрузки текстур кораблей.
- Загружает текстуру воды в зависимости от размера клетки.
- Вызывает LoadWreckageTextures() для загрузки текстур обломков.

bool LoadShipTextures()

- Описание: Загружает текстуры кораблей для разных размеров (1-4) и клеток (30/40/50 px).
- Функциональность:
  - Очищает массив ship_texture_.
  - Для каждого размера корабля и размера клетки загружает текстуры целых и повреждённых сегментов.

bool LoadWreckageTextures()

- Описание: Загружает текстуры обломков кораблей.
- Функциональность:
  - Очищает вектор wreckage_textures_.
  - Загружает текстуры обломков для кораблей размером 1-4 и клеток 30/40/50 px.

void OnAttackResult(const AttackResult& result)

- Описание: Обрабатывает результат атаки, перенаправляя вызов к перегруженной версии.

void OnAttackResult(const AttackResult& result, bool on_enemy_field)

- Описание: Обрабатывает результат атаки с указанием поля.
- Функциональность:
  - Проигрывает соответствующий звук (взрыв или всплеск воды).
  - Добавляет сообщение в лог.

void OnAbilityResult(const AbilityResult& result)

- Описание: Обрабатывает результат использования способности.
- Функциональность:
  - Проигрывает звук способности.
  - Добавляет сообщение в лог.

void ShowShotBanner(const std::string& text, bool on_enemy_field)

- Описание: Показывает временный баннер с текстом над полем.
- Функциональность:
  - Устанавливает текст баннера, флаг поля, активирует баннер и перезапускает таймер.

void ShowMessage(const std::string& message)

- Описание: Отображает текстовое сообщение (добавляет в лог).

void AddMessage(const std::string& message)

- Описание: Добавляет сообщение в соответствующий лог (игрока, AI или способностей, ошибки).

void RenderBanners()

- Описание: Отрисовывает активные баннеры.
- Функциональность:
  - Проверяет время отображения баннера и деактивирует его при превышении.

void DrawHitMark(const sf::Vector2f& center)

- Описание: Рисует маркер попадания (красный крест).

void DrawMissMark(const sf::Vector2f& center)

- Описание: Рисует маркер промаха (белая точка).

void DrawScanMark(const sf::Vector2f& center, bool positive)

- Описание: Рисует маркер результата сканирования (ромб для найденного сегмента, круг для пустой клетки).

void RenderGameTitle(int round)

- Описание: Отрисовывает заголовок игры с номером раунда.

void RenderScore(const Game& game)

- Описание: Отображает счёт игры.

void RenderFieldFrame(const PlayingField& field, const sf::Vector2f& position, const std::string& title)

- Описание: Рисует рамку и заголовок игрового поля.

int CellIdx() const

- Описание: Возвращает индекс размера клетки (0 для 30px, 1 для 40px, 2 для 50px).

std::vector<std::string> wrapText(const std::string& text, float maxWidth, unsigned int characterSize)

- Описание: Разбивает текст на строки по заданной ширине.

Дополнительные методы рендеринга:

- RenderLog() - отображает лог действий игрока и AI
- RenderInput() - панель ввода при расстановке кораблей
- RenderAbilitiesInfo() - информация о следующей способности
- RenderShipsInfo() - информация о размещённых кораблях
- RenderHelp() - окно помощи
- RenderStats() - статистика игры

- RenderDialogFiles() - диалог выбора слота сохранения/загрузки
- RenderAskWindow() - универсальное диалоговое окно подтверждения
- И другие методы для различных состояний игры

static sf::String utf8(const std::string& s)

- *Описание:* Преобразует строку UTF-8 в sf::String.
- *Функциональность:*
  - Конвертирует входную строку std::string в sf::String с помощью sf::String::fromUtf8(s.begin(), s.end()) для корректного отображения текста.

static sf::String utf8(const char s)

- *Описание:* Преобразует C-строку UTF-8 в sf::String.
- *Функциональность:*
  - Вычисляет длину C-строки (std::strlen(s)) и конвертирует её в sf::String с помощью sf::String::fromUtf8(s, s + std::strlen(s)).

6.7. GUI-ввод GUIInputHandler

Назначение:

GUIInputHandler — наследник InputHandler, отвечающий за обработку пользовательского ввода в графическом GUI-интерфейсе игры, реализованном через SFML.
В отличие от консольного обработчика, GUI-версия работает с системой событий SFML (window.pollEvent) и преобразует их в игровые команды Command.

Класс поддерживает конфигурацию управления из файла keybinds_gui.cfg, гибкую автоматическую коррекцию конфликтов, выбор схемы движения (ARROWS или WASD), сохранение исправленных настроек и строгую валидацию всех обязательных команд.

Класс гарантирует неблокирующую обработку ввода: за один вызов GetCommand() возвращается не более одной команды.

Основные особенности

1. Интеграция с системой событий SFML

GUIInputHandler принимает события из окна:

- закрытие окна → команда EXIT
- нажатие клавиши → поиск привязки
- отдельная обработка движения (зависит от схемы WASD/ARROWS)
- поддержка неблокирующей обработки через внутренний буфер last_command

2. Конфигурация из файла keybinds_gui.cfg

Поддерживается читаемый формат:

Space = ATTACK

L = LOAD

movement_scheme = WASD

Функции парсера:

- устойчивость к пробелам, табам, пустым строкам
- поддержка комментариев (и ;)
- подробные сообщения об ошибках
- автоматическое исправление конфликтов

3. Две схемы движения

Управление движением не берётся из файла, а определяется параметром:

- movement_scheme = ARROWS (по умолчанию)
- movement_scheme = WASD

При загрузке файла:

- команды MOVE_UP/LEFT/DOWN/RIGHT в конфиге игнорируются
- клавиши W/A/S/D или стрелки считаются "зарезервированными"

После загрузки конфигурации вызывается SetMovementBindings() — движение всегда безопасно перезаписывает любые привязки.

4. Автоматическая коррекция конфигурации

Метод ValidateOrFixBindings():

- проверяет наличие всех обязательных команд
- если команда отсутствует — назначает безопасную свободную клавишу
- если клавиша конфликтует со схемой движения — переприязывает

- гарантирует сохранение рабочей конфигурации даже при полностью испорченном файле

Если конфигурация была исправлена, файл автоматически пересохраняется.

5. Дефолтные привязки

Если файл отсутствует или повреждён:

- загружается минимальный безопасный набор команд
- затем применяются исправления
- затем добавляется движение (WASD или стрелки)

6. Логирование

Класс сообщает:

- об ошибках конфигурации
- об игнорированных строках
- о переназначенных клавишах
- о автоматически исправленных командных привязках
- о итоговом списке действующих клавиш

Методы GUIInputHandler

GUIInputHandler(sf::Window& window, const std::string& config_file = "keybinds_gui.cfg")

Конструктор инициализирует:

- ссылку на окно SFML
- имя конфигурационного файла
- схему движения (ARROWS по умолчанию)

- пустую таблицу привязок

bool Initialize()

Полная процедура настройки управления:

1. LoadFromFile()
    - попытка загрузить конфиг
    - если не удалось → загрузить дефолт
2. ValidateOrFixBindings()
    - добавляет отсутствующие команды
    - заменяет конфликтующие клавиши
    - выбирает безопасные клавиши (буквы/цифры)
    - при исправлениях → сохранить файл
3. SetMovementBindings()
    - всегда перезаписывает клавиши движения согласно схеме
4. Выводит все итоговые привязки и возвращает true

Метод всегда завершает инициализацию успешно, так как конфигурация автоматически исправляется.

std::unique_ptr GetCommand()

Обрабатывает очередь событий SFML:

- sf::Event::Closed → возвращает EXIT
- sf::Event::KeyPressed
    - проверяет клавиши движения в зависимости от схемы

- затем ищет клавишу в key_bindings_

- использует временный буфер last_command, чтобы обеспечить правило:
  "не более одной команды за вызов"

Если команд нет — возвращает nullptr.

void handleEvent(const sf::Event& event)

Альтернативный вспомогательный метод:

- используется для тестирования и отладки
- выводит распознанную команду в консоль

Конфигурационные методы

bool LoadFromFile()

- читает файл построчно
- игнорирует пустые строки и комментарии
- парсит пары <клавиша> = <команда>
- блокирует команды движения
- проверяет конфликт клавиши со схемой движения
- при конфликте автоматически ищет безопасную свободную клавишу
- может частично восстановить файл
- не прекращает загрузку при ошибках в строках

Возвращает true, даже если часть строк была некорректна.

void SetDefaultBindings()

Заполняет таблицу привязок стандартными командами:

- Space = ATTACK
- P = PLACE_SHIP
- R = ROTATE_SHIP
- Q = EXIT
- U = USE_ABILITY
- H, T, L, F2, F5, Escape ...
- Num1–Num5 = SET_1 … SET_5
- YES/NO: Y, N
- команды размещения зависят от схемы движения

bool SaveDefaultConfig() const

Сохраняет текущие (=исправленные) привязки в читаемом виде:

- комментарии
- movement_scheme
- пары <Key> = <Command>

bool ValidateOrFixBindings()

Гарантирует, что каждая важная команда присутствует:

- ATTACK
- PLACE/ROTATE_SHIP
- EXIT

- HELP
- USE_ABILITY
- LOAD, SAVE, RESTART
- SHOW/REMOVE_SHIPS
- SET_1…SET_5
- YES/NO
- *(движение не проверяется — оно задаётся схемой)*

Если команда отсутствует — назначается безопасная клавиша: буква → если занято → цифра → если занято → следующая.

Если привязки менялись, метод возвращает false → вызывается сохранение конфига.

Парсеры

CommandType ParseCommand(const std::string&)

Преобразует строку в enum. Нераспознанное → UNKNOWN.

sf::Keyboard::Key ParseKey(const std::string&)

Поддерживает:

- A–Z, 0–9
- F1–F15
- стрелки
- спец. клавиши (Space, Enter, Escape, Delete…)

Обратные преобразования

- KeyToString(sf::Keyboard::Key)

- CommandToString(CommandType)

Гарантируют корректную запись в файл.

Вспомогательные функции

static void Trim(std::string&)

Удаляет пробелы, табы и переносы по краям строки.

static bool IsComment(const std::string&)

Строки, начинающиеся с или ;.

6.8. Класс SoundManager

Назначение:

Класс SoundManager реализует паттерн синглтон и отвечает за управление всеми звуковыми эффектами и фоновой музыкой в игре. Он обеспечивает централизованное воспроизведение звуков через библиотеку SFML Audio.

Основные особенности:

- Синглтон: гарантирует единственный экземпляр класса в приложении

- Управление звуками: загрузка и воспроизведение звуковых эффектов (взрывы, всплески воды, сканирование)

- Фоновая музыка: поддержка фоновой музыки с возможностью циклического воспроизведения

- Обработка ошибок: корректная работа даже при отсутствии некоторых звуковых файлов

Методы

Статический метод static SoundManager& getInstance()

- Описание: Возвращает единственный экземпляр класса SoundManager.

- Функциональность:

  - Реализует паттерн синглтон через статическую локальную переменную

  - Гарантирует создание только одного экземпляра класса

  - Возвращает ссылку на существующий экземпляр

bool LoadSounds()

- Описание: Загружает все звуковые ресурсы из файлов.

- Функциональность:

  - Пытается загрузить звук взрыва из "resources/sounds/explosion.wav"

- При неудаче выводит предупреждение в консоль

- При успехе устанавливает буфер для звука взрыва

- Пытается загрузить звук всплеска воды из "resources/sounds/water_splash.wav"

- При неудаче выводит предупреждение в консоль

- При успехе устанавливает буфер для звука всплеска

- Пытается загрузить звук сканера из "resources/sounds/scanner.wav"

- При неудаче выводит предупреждение в консоль

- При успехе устанавливает буфер для звука сканера

- Пытается загрузить фоновую музыку из "resources/sounds/background_music.ogg"

- При неудаче выводит ошибку в консоль

- Возвращает true если фоновая музыка загружена успешно, иначе false

void PlayExplosion()

- Описание: Воспроизводит звук взрыва.

- Функциональность:

- Запускает воспроизведение звука взрыва через explosion_sound_.play()

- Используется при попадании в корабль

void PlayWaterSplash()

- Описание: Воспроизводит звук всплеска воды.

- Функциональность:

- Запускает воспроизведение звука всплеска воды через water_splash_sound_.play()

- Используется при промахе

void PlayAbilitySound(const std::string& ability_name)

- Описание: Воспроизводит звук, соответствующий способности.

- Функциональность:

  - Если ability_name равно "Scanner", воспроизводит звук сканера

  - Для всех других способностей воспроизводит звук взрыва

  - Параметры:

    - ability_name: название способности для выбора соответствующего звука

void PlayBackgroundMusic()

- Описание: Запускает фоновую музыку.

- Функциональность:

  - Устанавливает цикличное воспроизведение (setLoop(true))

  - Запускает воспроизведение фоновой музыки

void StopBackgroundMusic()

- Описание: Останавливает фоновую музыку.

- Функциональность:

  - Останавливает воспроизведение фоновой музыки

  - Используется при завершении игры

Приватные поля

Конструктор

SoundManager() = default

- Описание: Приватный конструктор по умолчанию.

- Функциональность: Предотвращает создание экземпляров класса извне

Звуковые буферы

sf::SoundBuffer explosion_buffer_

- Назначение: Буфер для хранения данных звука взрыва

sf::SoundBuffer water_splash_buffer_

- Назначение: Буфер для хранения данных звука всплеска воды

sf::SoundBuffer scanner_buffer_

- Назначение: Буфер для хранения данных звука сканера

Звуковые объекты

sf::Sound explosion_sound_

- Назначение: Объект для воспроизведения звука взрыва

sf::Sound water_splash_sound_

- Назначение: Объект для воспроизведения звука всплеска воды

sf::Sound scanner_sound_

- Назначение: Объект для воспроизведения звука сканера

Фоновая музыка

sf::Music background_music_

- Назначение: Объект для воспроизведения фоновой музыки

- Особенности: Поддерживает потоковое воспроизведение из файла

6.9. Главная функция main

Назначение:

Инициализирует необходимые объекты и начинает работу программы.

1. Инициализация звуковой системы

- Загружаются звуковые эффекты через синглтон SoundManager

- Если загрузка не удалась, выводится предупреждение

2. Настройка игры

- Создается объект настроек игры GameSettings

- Показывается диалог настроек (пользователь выбирает параметры)

- Создается основной объект игры Game с выбранными настройками

3. Выбор типа интерфейса

Консольный интерфейс

- Создается обработчик ввода для терминала

- Создается стратегия рендеринга для консоли

- Создается контроллер для консольной версии

Графический интерфейс (GUI)

- Определяется разрешение рабочего стола

- Создается окно SFML с размерами чуть меньше экрана

- Включается вертикальная синхронизация

- Создается стратегия рендеринга для GUI с указанным размером клетки

- Создается обработчик ввода для GUI с конфигурацией клавиш

- Создается контроллер для GUI версии

4. Запуск игры

- Инициализируется контроллер

- Если звук активен, включается фоновая музыка

- Запускается главный игровой цикл

5. Завершение работы

- Останавливается фоновая музыка при завершении игры

6. Обработка ошибок

- Перехватываются и логируются все исключения

- При ошибке возвращается код 1

Выполнение:

1. Инициализация звуков → 2. Настройки игры → 3. Создание интерфейса → 4. Запуск игрового цикла → 5. Корректное завершение

6.10. Решения и их обоснования

6.10.1. Основные архитектурные решения

1. Strategy Pattern для рендеринга и ввода

Решение: Введены интерфейсы IRenderStrategy и InputHandler.

Причина: Позволяет заменять реализации (Консоль/GUI) без изменения кода ядра игры (Game) или контроллера (GameController).

Обоснование: Чистое разделение ответственности. Консольная и GUI-версии разрабатываются независимо, следуя единому контракту.

2. Шаблонный класс-адаптер Renderer<T>

Решение: Renderer<T> — это тонкая обёртка, реализующая IRenderStrategy и делегирующая вызовы конкретной стратегии T.

Причина: Компиляция гарантирует, что тип T реализует весь интерфейс IRenderStrategy. Нулевые накладные расходы по сравнению с виртуальными вызовами.

Обоснование: Renderer<ConsoleRenderer> и Renderer<GUIRenderer> — это разные типы, обеспечивая type-safety и высокую производительность.

3. Шаблонный контроллер GameController<InputHandlerT, RendererT>

Решение: Класс GameController параметризуется типами обработчика ввода и рендерера.

Причина: Позволяет создавать связки конкретных реализаций на этапе компиляции. Невозможно создать контроллер с несовместимыми компонентами.

Обоснование: Прямая подстановка типов (TerminalInputHandler, Renderer<ConsoleRenderer>) исключает необходимость в dynamic_cast и проверках типов во время выполнения.

4. Объекты-команды (Command Pattern)

Решение: Класс Command инкапсулирует все пользовательские действия.

Причина: Унифицированный протокол взаимодействия между InputHandler и GameController. InputHandler создаёт команды, GameController их исполняет.

Обоснование: Позволяет легко добавлять новые команды, логировать действия и реализовывать систему "повтора" (undo/redo) в будущем.

5. Централизованная валидация команд

Решение: Метод GameController::CanExecuteCommand проверяет, допустима ли команда в текущем GameStatus.

Причина: Запрещает нелогичные действия (например, атаку во время хода ИИ или в меню паузы).

Обоснование: Инвариант игрового процесса контролируется в одном месте, а не распределен по всем обработчикам ввода.

6. Разделение конфигураций ввода

Решение: TerminalInputHandler использует keybinds.cfg, GUIInputHandler — keybinds_gui.cfg.

Причина: У консоли и GUI принципиально разная природа ввода (символы/ESC-последовательности vs. клавиши SFML).

Обоснование: Каждый обработчик может иметь собственную, оптимальную для него, схему конфигурации без компромиссов.

7. Отказоустойчивые конфигурации ввода

Решение: Оба обработчика ввода (TerminalInputHandler, GUIInputHandler) автоматически проверяют и исправляют конфигурационные файлы, создавая рабочие настройки по умолчанию в случае ошибок.

Причина: Игра должна запускаться и быть управляемой даже при повреждённом или отсутствующем конфиге.
Обоснование: Обеспечивает "нулевую настройку" (zero-configuration) для пользователя и устойчивость к ошибкам.

6.10.2. Сквозные инварианты и контракты взаимодействия

1. Инвариант неблокирующего ввода

Контракт: Метод InputHandler::GetCommand() должен быть неблокирующим и возвращать не более одной команды за вызов.

Обоснование: Гарантирует отзывчивость игрового цикла. Цикл продолжает работать, обрабатывать рендеринг и логику ИИ, даже если пользователь не нажимает клавиши.

2. Инвариант согласованности состояния

Контракт: RendererT обязан отображать состояние, получаемое исключительно через публичный интерфейс Game и GameState.

Обоснование: Рендерер — это только рисует. Все изменения состояния происходят внутри Game, что обеспечивает согласованность данных.

3. Контракт обработки событий рендерера

Контракт: Методы OnAttackResult и OnAbilityResult интерфейса IRenderStrategy вызываются *после* того, как игра обработала действие и обновила своё состояние.

Обоснование: Рендерер получает уже валидные результаты для отображения визуальных эффектов, синхронизированных с игровой логикой.

6.10.3. Обработка ошибок

1. Глобальный перехват в контроллере

Решение: GameController::ExecuteCommand обёрнут в try/catch. Все исключения, возникающие при выполнении команды, перехватываются и отображаются пользователю через Renderer::ShowMessage.

Обоснование: Игровой цикл никогда не "падает" из-за исключений в игровой логике.

6.10.4. Производительность, отзывчивость, UX

1. Дифференциальный рендеринг в ConsoleRenderer

Решение: ConsoleRenderer строит кадр в буфере и выводит только изменившиеся строки, используя ANSI-коды для позиционирования курсора.

Обоснование: Полностью устраняет мерцание консоли и резко снижает нагрузку на подсистему ввода-вывода.

2. Тактические паузы

Решение: GameController добавляет небольшие задержки (std::sleep_for) при ходе ИИ и отображении баннеров.

Обоснование: Делает процесс визуально понятным для игрока, даёт время на восприятие событий, улучшает пользовательский опыт.

3. Неблокирующий игровой цикл

Решение: Цикл в GameController::Run() всегда выполняет рендеринг, а затем *быстро* проверяет ввод. Это гарантирует плавность анимаций в GUI и отзывчивость интерфейса.

Обоснование: Интерфейс не "зависает" даже при активной работе игры.

7. Выводы

В ходе выполнения лабораторной работы были разработаны и реализованы шаблонные классы для управления игрой и отрисовки игрового состояния. Основные результаты:

1. Была достигнута цель — создание обобщённых (шаблонных) классов управления игрой (GameController) и отрисовки (Renderer), которые позволяют подменять реализации ввода и рендеринга без изменения кода ядра игры. Это обеспечило поддержку консольной и графической конфигурации.

2. Архитектурная гибкость: Применение шаблонного подхода позволило достичь высокой степени модульности и расширяемости системы. Компоненты ввода и отрисовки были изолированы от игровой логики.

3. Производительность и отзывчивость: Использование полиморфизма в шаблонных классах позволило избежать накладных расходов, связанных с виртуальными вызовами, таких как игровой цикл и рендеринг. Дифференциальный рендеринг в консольной версии и паузы в GUI обеспечили «чистый» пользовательский интерфейс.

4. Пользовательский опыт: Интуитивно понятное управление, цветовое кодирование, визуальные эффекты и звуковое сопровождение значительно. Настройка управления через конфигурационные файлы предоставляет пользователям возможность кастомизации.