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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Объектно-ориентированное программирование»

Тема: Создание классов

Студентка гр. 3383 _____ Земерова С.Н.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Оглавление

1. Цель работы

Разработать основные классы для игры «Морской бой» в соответствии с техническим заданием.

2. Задание

a) Создать класс корабля, который будет размещаться на игровом поле. Корабль может иметь длину от 1 до 4, а также может быть расположен вертикально или горизонтально. Каждый сегмент корабля может иметь три различных состояния: целый, поврежден, уничтожен. Изначально у корабля все сегменты целые. При нанесении 1 урона по сегменту, он становится поврежденным, а при нанесении 2 урона по сегменту, уничтоженным. Также добавить методы для взаимодействия с кораблем.

b) Создать класс менеджера кораблей, хранящий информацию о кораблях. Данный класс в конструкторе принимает количество кораблей и их размеры, которые нужно расставить на поле.

c) Создать класс игрового поля, которое в конструкторе принимает размеры. У поля должен быть метод, принимающий корабль, координаты, на которые нужно поставить, и его ориентацию на поле. Корабли на поле не могут соприкасаться или пересекаться. Для игрового поля добавить методы для указания того, какая клетка атакуется. При попадании в сегмент корабля изменения должны отображаться в менеджере кораблей.

Каждая клетка игрового поля имеет три статуса:

i. неизвестно (изначально вражеское поле полностью неизвестно),
ii. пустая (если на клетке ничего нет)
iii. корабль (если в клетке находится один из сегментов корабля).

Для класса игрового поля также необходимо реализовать конструкторы копирования и перемещения, а также соответствующие им операторы присваивания.

3. Модель предметной области и инварианты

3.1. Сущности

- Клетка (Cell): состояние UNKNOWN | EMPTY | SHIP.
- Сегмент корабля: состояние INTACT | DAMAGED | DESTROYED.
- Корабль (Ship): длина 1-4, ориентация HORIZONTAL | VERTICAL, массив сегментов.
- Игровое поле (PlayingField): прямоугольная сетка NxM; размещение кораблей без пересечений/касаний.
- Менеджер кораблей (ShipManager): учёт расставляемых кораблей.

3.2. Инварианты

- Размер корабля — от 1 до 4 клеток.
- Координаты кораблей и выстрелов всегда в пределах поля.
- Размещение корабля: никаких касаний (включая по диагонали) и пересечений.
- Повреждения сегментов: INTACT → DAMAGED → DESTROYED строго по урону; итог «корабль уничтожен», если все сегменты DESTROYED.
- После выстрела клетка не остаётся в UNKNOWN. Попадание/промах отражается в поле.

4. Детальное описание классов

4.1. Корабль Ship

Общее назначение

Класс Ship представляет корабль с механикой повреждений. Представлена трехуровневая система повреждений сегментов, где каждый сегмент должен быть поражен дважды для полного уничтожения.

Структура данных

Приватные поля:

- start_position_ - структура Position, содержащая координаты x и y начальной точки корабля на игровом поле

- ship_size_ - целочисленное значение размера корабля (от 1 до 4 сегментов)

- ship_orientation_ - ориентация корабля (VERTICAL или HORIZONTAL)

- segments_ - вектор состояний SegmentState каждого сегмента корабля

- ship_number_ - уникальный номер корабля для идентификации

- destroyed_segments_ - счетчик сегментов в состояниях DAMAGED или DESTROYED

- hit_count_ - общее количество попаданий по кораблю

Описание методов

Конструкторы и операторы:

Ship(Position start, Orientation orient, int size, int number)

- Создает новый корабль с указанными параметрами

- Инициализирует все сегменты состоянием INTACT

- Устанавливает счетчики destroyed_segments_ и hit_count_ в 0

- Выбрасывает исключение при невалидном размере (меньше 1 или больше 4)

Ship(const Ship& other) - конструктор копирования

- Создает полную глубокую копию корабля

- Копирует все поля включая состояния сегментов и счетчики

Ship(Ship&& other) noexcept - конструктор перемещения

- Эффективно перемещает ресурсы из временного объекта

- Использует семантику перемещения для вектора segments

operator=(const Ship& other) - оператор присваивания копированием

- Выполняет проверку на самоприсваивание

- Копирует все поля из другого корабля

operator=(Ship&& other) noexcept - оператор присваивания перемещением

- Выполняет проверку на самоприсваивание

- Перемещает ресурсы из временного объекта

~Ship() - деструктор

- Освобождает ресурсы, занимаемые объектом

Методы доступа к свойствам:

ship_size const

- Возвращает размер корабля в сегментах

- Всегда возвращает значение от 1 до 4

orientation() const

- Возвращает ориентацию корабля (VERTICAL или HORIZONTAL)

set_orientation(Orientation orientation)

- Изменяет ориентацию корабля. Принимает значение типа Orientation (VERTICAL или HORIZONTAL) и присваивает его полю ship_orientation_.

start_position() const

- Возвращает структуру Position с координатами начальной точки

set_start_position(Position pos)

- Устанавливает новые координаты начальной точки корабля на игровом поле. Принимает структуру Position и полностью заменяет текущее значение поля start_position_.

ship_number() const

- Возвращает уникальный номер корабля**set_ship_number(int number)** Задаёт новый уникальный идентификатор корабля. Принимает целое число и сохраняет его в поле ship_number_.

set_ship_number(int number)

- Задаёт новый уникальный идентификатор корабля. Принимает целое число и сохраняет его в поле ship_number_.

destroyed_segments() const

- Возвращает количество пораженных сегментов (DAMAGED или DESTROYED)

set_destroyed_segments(int count)

- Принудительно задаёт текущее значение счётчика «поражённых» сегментов (сегментов, находящихся в состоянии DAMAGED или DESTROYED). Выбрасывает std::invalid_argument, если передано отрицательное значение или значение больше размера корабля. Полезен при сериализации/десериализации игры или при откате состояний.

hit_count() const

- Возвращает общее количество попаданий по кораблю

set_hit_count(int count)

- Принудительно устанавливает общее количество попаданий по кораблю за всё время. Выбрасывает std::invalid_argument при попытке передать отрицательное значение. Используется вместе с set_destroyed_segments при загрузке сохранённой игры или при тестировании.

segment_state(int index) const

- Возвращает состояние сегмента по индексу

- Выбрасывает исключение std::out_of_range при невалидном индексе

set_segment_state(int index, SegmentState state)

- Принудительно изменяет состояние конкретного сегмента корабля по его индексу (от 0 до ship_size_−1). Выбрасывает исключение std::out_of_range, если переданный индекс выходит за допустимые границы. Используется в

редких случаях (например, при применении специальных игровых способностей или при восстановлении состояния из сохранения).

segment_index(Position current) const

- Определяет индекс сегмента по координатам на поле

- Для горизонтальных кораблей: проверяет совпадение Y-координаты, вычисляет смещение по X

- Для вертикальных кораблей: проверяет совпадение X-координаты, вычисляет смещение по Y

- Возвращает -1, если координаты не принадлежат кораблю

segment_position(int index) const

- Определяет координаты сегмента по его индексу

- Для горизонтальных кораблей: start_position_.x + index, start_position_.y

- Для вертикальных кораблей: start_position_.x, start_position_.y + index

- Возвращает Position(-1, -1) для невалидного индекса

const std::vector<SegmentState>& segments() const

- Возвращает константную ссылку на вектор состояний сегментов

- Позволяет получить полную информацию о состоянии всех сегментов

segment_state(int x, int y) const

- Возвращает кортеж с полной информацией о сегменте по координатам

- Формат: (индекс_сегмента, состояние_сегмента, ориентация_корабля)

- Возвращает (-1, SegmentState::NONE, Orientation::HORIZONTAL) если сегмент не найден

Методы обработки повреждений:

DamageShip(Position position_for_damage, int damage = 1)

- Основной метод для нанесения урона кораблю

- *Параметры:*

  - position_for_damage: координаты цели

  - damage: уровень урона (1 - обычный, 2 - двойной)

- *Возвращаемые значения:*

  - -1: промах или попадание в уже уничтоженный сегмент

  - 1: попадание, приведшее к уничтожению корабля

- 2: попадание без уничтожения корабля

Hit(int index, int damage)

- Внутренний метод обработки попадания в конкретный сегмент

- *Логика обработки:*

  - Для INTACT сегмента: переводит в DAMAGED (при damage=1) или сразу в DESTROYED (при damage=2)

  - Для DAMAGED сегмента: всегда переводит в DESTROYED

  - Увеличивает destroyed_segments_ при первом попадании в сегмент

  - Всегда увеличивает hit_count_ при любом попадании

Методы проверки состояния:

IsDestroyed() const

- Проверяет, полностью ли уничтожен корабль

- Возвращает true только если ВСЕ сегменты находятся в состоянии DESTROYED

- Возвращает false если хотя бы один сегмент в состоянии INTACT или DAMAGED

Особенности реализации

Механика повреждений

Трехуровневая система состояний:

1. INTACT → DAMAGED - первое попадание (обычный урон)

2. INTACT → DESTROYED - первое попадание (двойной урон)

3. DAMAGED → DESTROYED - второе попадание (любой урон)

Особенности системы:

- Сегменты в состоянии DESTROYED игнорируют последующие попадания

- Корабль считается уничтоженным только когда все сегменты в состоянии DESTROYED

- Система поддерживает различные уровни урона для реализации специальных способностей

4.2. Менеджер кораблей ShipManager

Общее назначение

Класс ShipManager является менеджером кораблей, который управляет метаинформацией о флоте кораблей в игре "Морской бой". Он отвечает за хранение и предоставление информации о количестве и размерах кораблей, без управления их конкретным расположением или состоянием.

Структура данных

Приватные поля:

- ship_count_ - целочисленное значение, хранящее общее количество кораблей во флоте

- ship_sizes_ - вектор целых чисел, содержащий размеры каждого корабля в сегментах

Описание методов

Конструкторы

ShipManager() — конструктор по умолчанию, создаёт пустой флот (ship_count_ = 0, ship_sizes_ пустой)

ShipManager(int count, const std::vector<int>& sizes) — создаёт менеджер с заданными параметрами:

- Инициализирует ship_count_ значением count

- Копирует переданный вектор в ship_sizes_

ShipManager(const ShipManager& other) — конструктор копирования

operator=(const ShipManager& other) — оператор присваивания

Сериализация

friend std::ostream& operator<<(std::ostream&, const ShipManager&) — вывод состояния в поток

friend std::istream& operator>>(std::istream&, ShipManager&) — чтение состояния из потока

Методы доступа к данным:

ship_count() const

- Возвращает общее количество кораблей во флоте

- Возвращает значение поля ship_count_

ship_size(int index) const

- Возвращает размер корабля по указанному индексу

- Параметры:

  - index: целочисленный индекс корабля (от 0 до ship_count_-1)

- Выбрасывает исключение std::out_of_range если индекс превышает или равен ship_count_

- Возвращает значение из вектора ship_sizes_ по указанному индексу

- Важно: индекс должен быть строго меньше ship_count_

Методы управления состоянием:

clear()

- Полностью очищает состояние менеджера кораблей

- Устанавливает ship_count_ в 0

- Очищает вектор ship_sizes_, удаляя все элементы

Особенности реализации

Простота интерфейса:

- Класс предоставляет минимальный необходимый интерфейс для работы с данными кораблей

- Не управляет расположением, ориентацией или состоянием повреждений кораблей

Использование исключений:

- Метод ship_size использует исключения для обработки ошибок выхода за границы

4.3. Клетка поля Cell

Общее назначение

Класс Cell представляет отдельную клетку игрового поля в игре "Морской бой". Он инкапсулирует состояние клетки, информацию о корабле (если он присутствует) и предоставляет методы для управления этим состоянием.

Структура данных

Приватные поля:

- state_ - перечисление CellState, определяющее текущее состояние клетки

- ship_index_ - целое число, хранящее номер корабля, который занимает эту клетку (если применимо)

- segment_index_ - целое число, хранящее индекс сегмента корабля в этой клетке (если применимо)

Перечисление состояний клетки

CellState:

- UNKNOWN - состояние неизвестности (клетка еще не была открыта/обстреляна)

- EMPTY - клетка пустая (вода, по которой уже стреляли)

- SHIP - в клетке находится корабль или его часть

Описание методов

Конструкторы и присваивание

Cell(CellState state) — создаёт клетку с заданным начальным состоянием, ship_index_ = -1, segment_index_ = -1

Cell(const Cell& other) — конструктор копирования

operator=(const Cell& other) — оператор присваивания с защитой от самоприсваивания

Методы проверки состояния

bool IsUnknown() const — возвращает true, если state_ == CellState::UNKNOWN

bool IsEmpty() const — возвращает true, если state_ == CellState::EMPTY

bool IsShip() const — возвращает true, если state_ == CellState::SHIP

Методы изменения состояния

void set_unknown() — устанавливает state_ = UNKNOWN, ship_index_ = -2, segment_index_ = -2 (используется для полного «затемнения» клетки, например при скрытии поля противника)

void set_empty() — устанавливает state_ = EMPTY, ship_index_ = -1, segment_index_ = -1 (отметка промаха)

void set_ship(int ind, int number) — устанавливает state_ = SHIP, segment_index_ = ind, ship_index_ = number (размещение части корабля в клетке)

Методы-сеттеры (добавленные в текущей реализации)

void set_ship_index(int index) — напрямую задаёт ship_index_

void set_segment_index(int index) — напрямую задаёт segment_index_

void set_state(CellState new_state) — напрямую изменяет состояние клетки

Методы доступа к данным

CellState segment_state() const — возвращает текущее значение state_

int ship_index() const — возвращает номер корабля (-1 или -2 если корабля нет)

int segment_index() const — возвращает индекс сегмента (-1 или -2 если нет)

Специальные методы

bool CanBeShot() const — возвращает true, если по клетке ещё можно стрелять (state_ == UNKNOWN || state_ == SHIP)

void reset() — приводит клетку к «чистому» состоянию: state_ = EMPTY, ship_index_ = -1, segment_index_ = -1

Особенности текущей реализации

Значения -1 и -2 используются для различения «нет корабля» и «информация скрыта». -1 — обычное отсутствие корабля (например, после промаха или при пустой клетке). -2 — используется в set_unknown() для полного скрытия любой привязки к кораблю.

4.4. Игровое поле PlayingField

Общее назначение

Класс PlayingField — основная структура игрового поля в «Морском бое». Управляет размещением кораблей, обработкой выстрелов, двойным отображением (реальное и видимое), механикой сканирования, режимом замены удалённых кораблей и полной сериализацией.

Структура данных

Приватные поля:

- real_grid_ — vector<vector<Cell>> — истинное расположение кораблей

- visible_grid_ — vector<vector<Cell>> — то, что видит игрок

- scanned_overlay_ — vector<vector<bool>> — клетки, подсмотренные сканером

- ships_ — vector<Ship> — активные корабли на поле

- removed_ships_ — vector<Ship> — временно удалённые корабли (для замены)

- x_size_, y_size_ — размеры поля

- count_ — количество активных кораблей

- is_in_replacement_mode_ — флаг режима замены

- current_orientation_ — текущая ориентация корабля при ручном размещении

Конструктор, копирование, перемещение

PlayingField(int x = 10, int y = 10)

1. Сохраняет размеры в x_size_, y_size_

2. Инициализирует real_grid_ размером y × x, все клетки — Cell(CellState::EMPTY)

3. Инициализирует visible_grid_ размером y × x, все клетки — Cell(CellState::UNKNOWN)

4. Инициализирует scanned_overlay_ размером y × x, все значения — false

5. Устанавливает count_ = 0, is_in_replacement_mode_ = false

PlayingField(const PlayingField& other) — глубокое копирование всех контейнеров и полей. Создаёт полностью независимую копию поля.

PlayingField& operator=(const PlayingField& other) — выполняет глубокое копирование всех данных.

PlayingField(PlayingField&& other) noexcept и operator=(PlayingField&&) noexcept — эффективно перемещают владение всеми векторами, обнуляя поля источника. Гарантируют отсутствие исключений.

~PlayingField() = default — автоматическое освобождение ресурсов (RAII).

Размещение кораблей

void RotateOrientation()

Метод переключает значение current_orientation_ между HORIZONTAL и VERTICAL.

Orientation current_orientation() const

Метод возвращает текущее значение ориентации корабля.

bool SetRandomShips(const ShipManager& manager, size_t max_attempts = 10000)

Метод выполняет автоматическое размещение всех кораблей из ShipManager игрока в случайных позициях:

1. Получает менеджер кораблей игрока.

2. Для каждого корабля из флота:

- Запрашивает его размер.

- В цикле до max_attempts попыток:

- Генерирует случайные координаты x, y и ориентацию.

- Пытается разместить корабль через PlaceShip().

- При успехе переходит к следующему кораблю.

- При исключении (пересечение, выход за границы) продолжает попытки.

- Если после всех попыток корабль не размещён — прерывает процесс и возвращает false.

3. При успешном размещении всех кораблей возвращает true.

bool PlaceShip(int x, int y, int size, Orientation orientation)

1. Если is_in_replacement_mode_ == true и removed_ships_ не пуст → вызывает PlaceRemovedShip()

2. Иначе → вызывает PlaceNewShip()

3. Возвращает результат вызова

void MoveShip(int x, int y, int size, Orientation orientation) (внутренний валидатор)

1. Проходит по всем сегментам корабля (по ориентации)

2. Проверяет:

   • Координаты в пределах поля → иначе ShipOutOfBoundsException

   • Клетка уже занята кораблём → ShipsOverlapException

3. Проверяет ореол 3×3 вокруг каждого сегмента:

   • Если соседняя клетка содержит корабль → ShipsTooCloseException

bool PlaceNewShip(int x, int y, int size, Orientation orientation)

1. Вызывает MoveShip() (валидация)

2. Создаёт новый Ship с номером count_

3. Вызывает PlaceShipOnGrid(ship)

4. Возвращает true

bool PlaceRemovedShip(int x, int y, int size, Orientation orientation)

1. Если removed_ships_ пуст → false

2. Берёт последний удалённый корабль

3. Вызывает MoveShip() (валидация новой позиции)

4. Обновляет у корабля позицию, ориентацию и номер (count_)

5. Вызывает PlaceShipOnGrid()

6. Удаляет корабль из removed_ships_

7. Если removed_ships_ пуст → is_in_replacement_mode_ = false

8. Возвращает true

void PlaceShipOnGrid(const Ship& ship)

1. Проходит по всем сегментам корабля

2. В real_grid_ по координатам вызывает set_ship(segment_index, ship_number)

3. Добавляет корабль в ships_

4. Увеличивает count_

Обработка выстрелов

int Damage(int x, int y, int damage = 1)

1. Проверка границ → иначе исключение

2. Если клетка не корабль:

   • Если уже не UNKNOWN → -1 (повторный выстрел)

   • Иначе: visible_grid_ → set_empty(), вернуть 0 (промах)

3. Если корабль:

   • Получаем ship_index и segment_index из real_grid_

   • Если сегмент уже DESTROYED → -1

   • Вызываем ships_[ship_index].DamageShip(Position(x,y), damage)

   • Обновляем visible_grid_ текущей клетки как SHIP

   • Если корабль уничтожен (IsDestroyed()):

     ◦ Раскрываем все сегменты корабля в visible_grid_

     ◦ Проходим по ореолу вокруг каждого сегмента → set_empty() для неизвестных клеток

     ◦ Вызываем MarkFullyDestroyed()

     ◦ Возвращаем 2 (потоплен)

• Иначе возвращаем 1 (попадание)

Доступ к данным

const Cell& visible_cell(int x, int y) const

1. Проверка границ → иначе ShipOutOfBoundsException

2. Возвращает ссылку на visible_grid_[y][x]

const Ship& ship(int index) const

1. Проверка индекса → иначе std::out_of_range

2. Возвращает ships_[index]

bool ship_info_at(int x, int y, int& ship_index, int& segment_index, int& ship_size, SegmentState& segment_state, Orientation& orientation) const

1. Перебирает все корабли в ships_

2. Для каждого вызывает segment_index(Position(x,y))

3. Если ≥0 — найден:

• Заполняет все выходные параметры

• Возвращает true

4. Если ничего не найдено → false

Сканирование

void set_cell_visible(int x, int y)

1. Проверка границ

2. Если валидно → scanned_overlay_[y][x] = true

bool IsScanned(int x, int y) const

1. Проверка границ → если нет → false

2. Возвращает scanned_overlay_[y][x]

Управление состоянием

void ClearField()

1. Все клетки real_grid_ → set_empty()

2. Все клетки visible_grid_ → set_unknown()

3. scanned_overlay_ → все false

4. ships_.clear(), count_ = 0

void ClearShip(int x, int y)

1. Находит корабль по клетке

2. Сохраняет его в removed_ships_

3. Очищает все его клетки в real_grid_ (reset())

4. Удаляет из ships_

5. Вызывает UpdateShipNumbersAfterRemoval()

6. Включает режим замены, уменьшает count_

void UpdateShipNumbersAfterRemoval(int removed_index)

1. Проходит по всему real_grid_: если ship_index > removed_index → уменьшает на 1

2. Проходит по оставшимся кораблям: уменьшает их ship_number_ на 1 (начиная с removed_index)

void ReturnStartState()

1. Все visible_grid_ → set_unknown()

2. Все scanned_overlay_ → false

3. Для всех кораблей: все сегменты → set_segment_state(..., INTACT)

Проверка конца игры

bool IsAllShipsDestroyed() const

Перебирает ships_, возвращает false, если хотя бы один корабль не уничтожен

Сериализация

void save(std::ostream&) const / void load(std::istream&)

Полная запись:

- Размеры поля

- count_, флаг режима замены

- Полные данные real_grid_, visible_grid_, scanned_overlay_

- Все активные и удалённые корабли (с полным состоянием сегментов)

Геттеры

- count(), x_size(), y_size() — прямой возврат полей

- removed_ship_size() — возвращает размер последнего в removed_ships_

Особенности реализации

Двойная система сеток:

- real_grid_ хранит истинное состояние для игровой логики

- visible_grid_ хранит то, что видит игрок (открытые клетки)

Механика сканирования:

- scanned_overlay_ работает независимо от visible_grid_

- Позволяет реализовать способности "сканирования" без совершения выстрелов

Обработка потопления:

- При уничтожении корабля автоматически раскрываются все его сегменты

- Также раскрывается вода вокруг корабля (правила морского боя)

4.5. Класс игрока Player

Общее назначение Класс Player представляет участника игры (человек или ИИ) и объединяет его игровое поле, флот и статистику стрельбы.

Структура данных

Приватные поля

std::string name_ — имя игрока

PlayerType type_ — тип игрока (HUMAN / AI)

std::unique_ptr<PlayingField> field_ — собственное игровое поле (полное владение)

std::shared_ptr<ShipManager> ship_manager_ — информация о количестве и размерах кораблей

int destroyed_ships_ — сколько кораблей противника уже потоплено

int hit_count_ — общее количество попаданий по противнику

int all_shots_ — общее количество сделанных выстрелов

int current_hit_ — зарезервировано под результат последнего выстрела (по умолчанию -1)

Описание методов

Player(const std::string& player_name, PlayerType player_type, std::shared_ptr<ShipManager> ship_manager, int field_width, int field_height)

- Сохраняет имя и тип
- Создаёт PlayingField(field_width, field_height) через make_unique
- Сохраняет переданные shared_ptr менеджеров
- Обнуляет все счётчики статистики

bool IsAllShipsDestroyed() const

- Возвращает remaining_ships() == 0

bool IsAllShipsPlaced() const

- Сравнивает ship_manager_->ship_count() и field_->count()

int MakeMove(std::unique_ptr<Player>& opponent, int x, int y)

- Проверяет координаты на валидность → иначе ShipOutOfBoundsException
- Увеличивает all_shots_
- Вызывает opponent->field_for_modification().Damage(x, y) → получает результат
- Если результат > 0 → увеличивает hit_count_
- Если результат == 2 → увеличивает destroyed_ships_
- Возвращает полученный результат

void RotateCurrentShip()

Переключает ориентацию следующего корабля, который будет размещён на поле вручную (с горизонтальной на вертикальную и наоборот).

- Делегирует вызов методу RotateOrientation() игрового поля.
- Ориентация хранится непосредственно в PlayingField, так как именно поле отвечает за логику размещения кораблей и знает текущую ориентацию для ручной расстановки.

Orientation current_orientation()

Возвращает текущую ориентацию, которая будет использована при размещении следующего корабля вручную.

- Делегирует вызов методу current_orientation() игрового поля. Позволяет узнать, в каком положении сейчас будет поставлен корабль при подтверждении координат.

bool PlaceShipsRandomly()

Выполняет автоматическую случайную расстановку всех кораблей из ShipManager на поле игрока.

- Проверяет наличие действительного ship_manager_. Если менеджер отсутствует — возвращает false.

- Делегирует выполнение методу SetRandomShips() игрового поля, передавая по ссылке текущий ShipManager.

- Возвращает true, если все корабли успешно размещены; false — если хотя бы один корабль не удалось разместить (например, из-за слишком плотного флота).

float accuracy() const

- Если all_shots_ == 0 → возвращает 0.0f
- Иначе возвращает (hit_count_ / all_shots_) * 100.0f

int remaining_ships() const

- Если ship_manager_ == nullptr → возвращает 0
- Иначе возвращает ship_manager_->ship_count() - destroyed_ships_

const PlayingField& field() const &

- Возвращает *field_. Метод возвращает константную ссылку. Используется в тех случаев, когда нужно будет только чтение поля (и изменять его нельзя)

PlayingField& field_for_modification() &

- Возвращает *field_. Метод возвращает обычную ссылку. Используется для тех случаев, когда нужно изменение состояние поля.

Геттеры и сеттеры

std::string name() const — возвращает имя игрока

PlayerType type() const — возвращает тип игрока (HUMAN или AI)

int destroyed_ships() const — возвращает количество уничтоженных кораблей противника

void set_destroyed_ships(int count) — принудительно задаёт количество уничтоженных кораблей противника

int hit_count() const — возвращает общее количество попаданий по противнику

void set_hit_count(int count) — принудительно задаёт количество попаданий

int all_shots() const — возвращает общее количество произведённых выстрелов

void set_all_shots(int count) — принудительно задаёт общее количество выстрелов

float accuracy() const — вычисляет и возвращает точность стрельбы в процентах (0.0f, если выстрелов не было)

int remaining_ships() const — возвращает количество ещё живых кораблей игрока

std::shared_ptr<ShipManager> ship_manager() const — возвращает shared-указатель на менеджер кораблей

void set_ship_manager(std::shared_ptr<ShipManager> ship_manager) — заменяет менеджер кораблей

const PlayingField& field() const & — возвращает константную ссылку на игровое поле (только чтение)

PlayingField& field_for_modification() & — возвращает неконстантную ссылку на игровое поле (для изменения состояния поля)

Особенности реализации

Разделение ответственности:

- Только игровой процесс и статистика, размещение кораблей делегировано.

Поддержка человека и ИИ:

- Используется перечисление PlayerType, способности только у человека.

Позднее связывание:

- set_ship_manager() для менеджера кораблей.

Система исключений:

- EmptyQueueException - отсутствие допустимых способностей при попытке их использования, ShipOutOfBoundsException - выстрел вне поля.

Безопасный доступ к полю:

- field() const в случае только чтения поля и field_for_modification() в случае изменения состояния поля

4.7. Исключения для координат кораблей ShipCoordinateExceptions

Общее назначение

Специализированная систему исключений для обработки ошибок, связанных с координатами и размещением кораблей. Основное назначение - детализированная валидация игрового поля и предоставление информативных сообщений об ошибках.

Структура иерархии

```
std::runtime_error
│
└── ShipPlacementException (общая ошибка размещения)
        ├── ShipOutOfBoundsException (координаты вне поля)
        ├── ShipsOverlapException (пересечение кораблей)
        ├── ShipsTooCloseException (корабли слишком близко)
        └── ImpossibleFleetException (невозможность размещения кораблей)
```

Описание классов

ShipPlacementException

Назначение:

- Универсальное исключение для общих ошибок размещения кораблей

Конструктор:

ShipPlacementException(const std::string& msg)

- Принимает готовое сообщение об ошибке

- Является базовым классом для остальных исключений размещения кораблей

ShipOutOfBoundsException

Назначение:

- Исключение для случаев, когда координаты выходят за пределы игрового поля

Конструктор:

ShipOutOfBoundsException(int x, int y, int xSize, int ySize)

- Принимает проблемные координаты и размеры поля

- Формирует информативное сообщение с указанием допустимого диапазона

ShipsOverlapException

Назначение:

- Исключение для случаев пересечения кораблей при размещении

Конструктор:

ShipsOverlapException(int x, int y)

- Принимает координаты клетки пересечения кораблей

- Формирует информативное сообщение с указанием координат клетки

ShipsTooCloseException

Назначение:

- Исключение для случаев, когда корабли расположены слишком близко (с учетом диагоналей)

Конструктор:

ShipsTooCloseException(int x1, int y1, int x2, int y2)

- Принимает координаты двух близко расположенных клеток

- Сообщает о нарушении правила размещения кораблей без касаний

ImpossibleFleetException

Назначение:

- Исключение используется в случае, когда автоматическая расстановка (или расстановка противника) заданного набора кораблей на игровом поле оказывается невозможной по геометрическим причинам (слишком много кораблей, слишком маленькое поле, неудачные размеры и т.п.).

Конструктор

ImpossibleFleetException()

- Конструктор передаёт в базовый класс понятное сообщение, которое информирует пользователя, что текущая конфигурация флота несовместима с размером поля и процесс расстановки будет перезапущен.

Совместимость с другими сущностями

С классом Player:

- Используется в методе makeMove() для валидации координат атаки

- Обеспечивает защиту от некорректных входных данных

С системой размещения кораблей:

- Интегрируется с логикой проверки валидности расстановки

- Предотвращает создание некорректных игровых состояний

4.8. Вспомогательные функции Other

Общее назначение

Заголовочный файл Other.h предоставляет набор вспомогательных функций для валидации данных (далее будет расширяться). Назначение — обеспечение проверки.

Функция isValid

Назначение:

- Быстрая проверка принадлежности координат игровому полю

bool isValid(int x, int y, int x_size, int y_size)

Параметры:

- x, y - проверяемые координаты

- x_size, y_size - размеры игрового поля

Логика проверки:

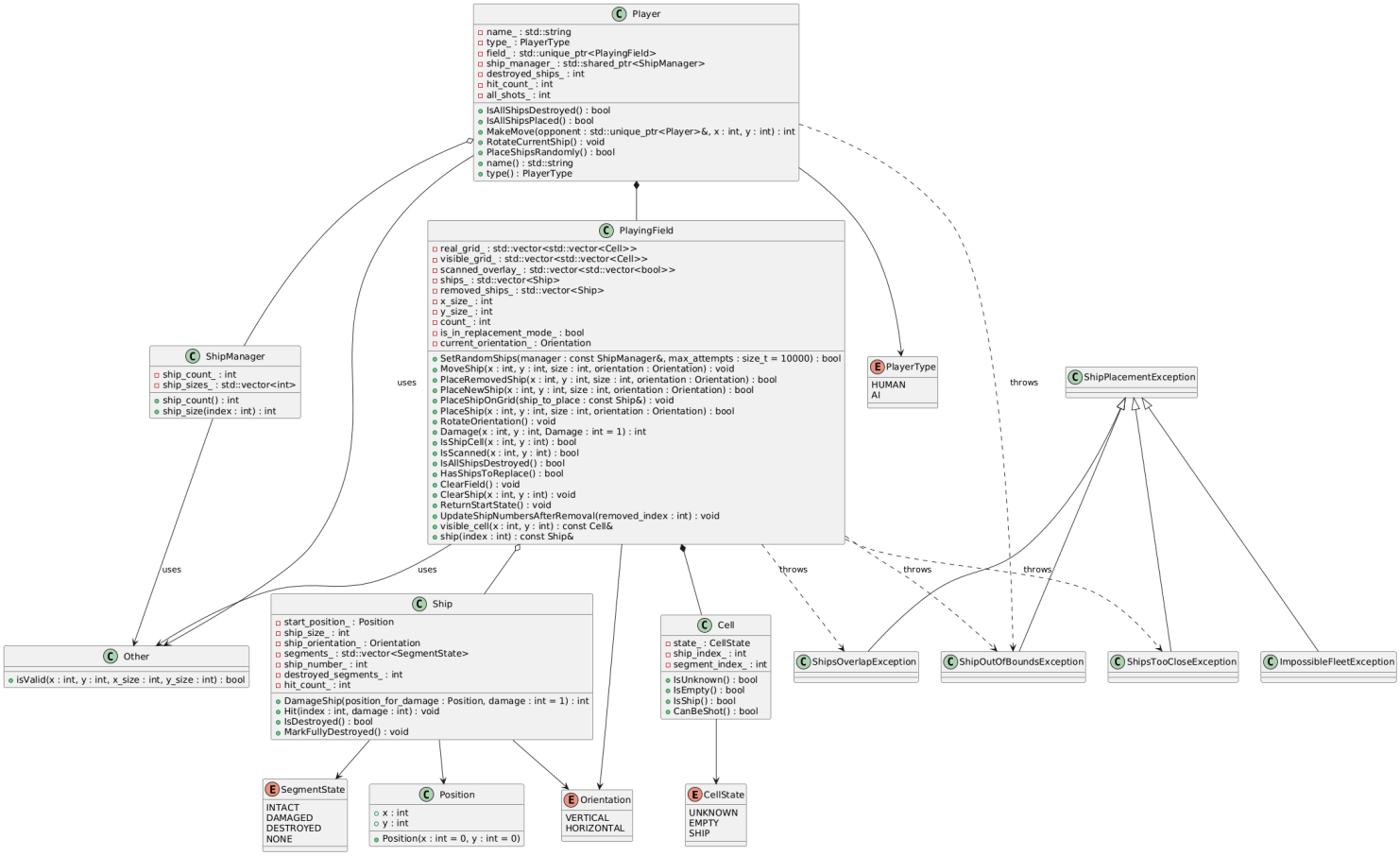
- Координата X должна быть в диапазоне [0, xSize-1]

- Координата Y должна быть в диапазоне [0, ySize-1]

- Возвращает true, только если обе координаты валидны

ДАЛЬШЕ НУЖНО UML ПЕРЕДЕЛАТЬ.

5. Диаграмма классов



Обоснование типов связей:

Композиция (Composition):

Player → PlayingField: Игрок владеет полем через unique_ptr<PlayingField>, время жизни поля зависит от игрока

PlayingField → Cell: Поле содержит ячейки как неотъемлемую часть, ячейки не существуют без поля

Агрегация (Aggregation):

Player → ShipManager: Игрок использует менеджер кораблей через shared_ptr<ShipManager>, менеджер может использоваться несколькими игроками

PlayingField → Ship: Поле содержит корабли, но корабли могут существовать независимо (например, в removed_ships_)

Зависимость (Dependency):

Используется для связей с перечислениями и структурами, которые являются типами полей

Все классы зависят от перечислений, которые определяют их состояние

Использование (Usage):

Классы используют вспомогательные функции из Other для валидации данных

Генерация исключений:

Классы бросают исключения при нарушении бизнес-правил

Связь показывает, какие исключения могут быть сгенерированы методами

Наследование:

Иерархия исключений для организации доменно-специфичных ошибок

6. Архитектурные решения

6.1. Причины выбора структуры классов и их взаимосвязей

1. Класс Ship - модель данных с поведением

- Почему отдельный класс: Корабль - центральная сущность игры с сложной логикой состояний сегментов

- Реализация с вектором SegmentState: Позволяет эффективно управлять трехуровневой системой повреждений (INTACT → DAMAGED → DESTROYED)

- Связь с Position и Orientation: Корабль должен "знать" свое положение и ориентацию для корректного взаимодействия с полем

- Методы segment_index/segment_position: Необходимы для преобразования между координатами поля и индексами сегментов

2. Класс Cell - универсальный блок для игрового поля

- Почему отдельный класс: Каждая клетка имеет собственное состояние и может содержать часть корабля

- Три состояния (UNKNOWN, EMPTY, SHIP): Отражают требования задания и логику отображения для игрока

- Хранение ship_number и segment_index: Позволяет связать клетку с конкретным сегментом корабля для обработки повреждений

- Метод CanBeShot(): Инкапсулирует логику валидации выстрелов

3. Класс PlayingField - менеджер игрового пространства

- Двойная система сеток (real_grid_/visible_grid_): Разделение реального состояния и видимой информации соответствует требованиям игры

- Композиция с Cell: Ячейки - неотъемлемая часть поля, их жизненный цикл зависит от поля

- Агрегация с Ship: Корабли могут существовать независимо, поле лишь содержит ссылки на них

- Метод PlaceShip с проверками: Централизованная валидация правил размещения кораблей

- Сканирующий слой ScannedOverlay: Поддержка будущих расширений (способность сканирования)

4. Класс ShipManager - конфигуратор флота

- Почему отдельный класс: Разделение ответственности - управление мета-информацией о кораблях отделено от их игрового представления

- Простой интерфейс (ship_count/ship_sizes): Обеспечивает гибкость в настройке флота без усложнения логики игры

- Вектор ship_sizes: Позволяет настраивать произвольные комбинации кораблей

5. Класс Player - центральный управляющий компонент

- Почему отдельный класс: Player представляет игрока как самостоятельную сущность, инкапсулируя все связанные с ним данные и поведение - игровое поле, статистику, тип игрока и логику выполнения ходов

- Композиция с PlayingField: Каждый игрок владеет эксклюзивным правом на свое игровое поле через unique_ptr, что обеспечивает строгое владение и автоматическое управление временем жизни поля

- Ассоциация с ShipManager: Использование shared_ptr для менеджера кораблей позволяет гибко управлять конфигурацией флота с разделяемым владением между двумя игроками

- Поддержка различных типов игроков: Перечисление PlayerType (HUMAN/AI) обеспечивает полиморфное поведение в зависимости от типа игрока без использования наследования

- Метод MakeMove(): Централизует всю логику выполнения хода - валидацию координат, взаимодействие с полем противника, обработку результатов выстрела и обновление статистики

- Система статистики: Встроенные счетчики попаданий, выстрелов и уничтоженных кораблей с методами для расчета точности и отслеживания прогресса игры

6.2. Пакетная структура

```
src/
├── core/                # Основная логика игры
│   ├── Ship.h/cpp           # Класс корабля
│   ├── ShipManager.h/cpp    # Менеджер кораблей
│   ├── PlayingField.h/cpp   # Игровое поле
│   ├── Cell.h/cpp           # Клетка игрового поля
│   └── Player.h/cpp         # Класс игрока
│
│
└── additional/          # Вспомогательные компоненты
    ├── Other.h/cpp          # Вспомогательные функции
    └── ShipCoordinateExceptions.h  # Исключения
```

6.3. Принципы

Принцип единственной ответственности (SRP) **- Каждый класс решает строго определённую задачу:**

- Ship: управление состоянием корабля и сегментов

- Cell: представление состояния одной клетки поля

- PlayingField: управление всем игровым пространством и взаимодействиями

- ShipManager: хранение конфигурации флота

- Player: управление игровым процессом конкретного участника, включая поле, статистику и выполнение ходов

Принцип инкапсуляции:

- Все поля классов приватные с доступом через методы

- Логика изменения состояний инкапсулирована (например, Hit() в Ship)

- Валидация данных централизована в соответствующих методах

7. Выводы

В ходе лабораторной работы были реализованы основные классы для игры «Морской бой» - Корабль, Менеджер кораблей, Клетка поля, Игровое поле и Интерфейс для взаимодействия с кораблями и полем.

Достигнутые результаты:

1. Соответствие техническому заданию

- Реализована многоуровневая система повреждений кораблей (INTACT → DAMAGED → DESTROYED)
- Обеспечено корректное размещение кораблей с проверкой пересечений и касаний
- Реализована система хранения флота через специальный менеджер
- Реализована двойная система отображения поля (real_grid_ и visible_grid_)

2. Соблюдение принципов ООП

- Инкапсуляция: все данные классов защищены, доступ через методы
- Абстракция: сложная логика игры скрыта за простыми интерфейсами
- Композиция: объекты строятся из более простых компонентов (Position, SegmentState и др.)