

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

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Объектно-ориентированное программирование»
Тема: Полиморфизм

Студент гр. 3344

Пачев Д.К.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы.

Разработать систему полиморфных способностей в игре, используя интерфейсы и наследование для реализации различных типов способностей, а также обеспечить обработку исключительных ситуаций.

Задание.

- a. Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:
 - i. Двойной урон - следующая атак при попадании по кораблю нанесет сразу 2 урона (уничтожит сегмент).
 - ii. Сканер - позволяет проверить участок поля 2x2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.
 - iii. Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.
- b. Создать класс менеджер-способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.
- c. Реализовать функционал получения одной случайной способности при уничтожении вражеского корабля.
- d. Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):
 - i. Попытка применить способность, когда их нет
 - ii. Размещение корабля вплотную или на пересечении с другим кораблем
 - iii. Атака за границы поля

Примечания:

- Интерфейс события должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс
- Не должно быть явных проверок на тип данных

Выполнение работы.

- **Класс Ability**

Описание:

Представляет собой интерфейс для различных способностей, которые игрок может применять в игре. Предоставляет метод `apply`, который должен быть реализован в классах-наследниках для конкретизации логики применения способности к игровому полю.

Методы класса:

- **`virtual bool apply(GameField &field) = 0;`**

Чисто виртуальный метод, который должен быть переопределен в классах-наследниках. Принимает ссылку на объект класса `GameField` и применяет соответствующую способность, возвращая результат применения в виде логического значения.

- **`virtual ~Ability() = default;`**

Виртуальный деструктор по умолчанию, обеспечивающий корректное удаление объектов классов-наследников.

- **Класс Bombardment**

Описание:

Представляет собой способность "Обстрел", позволяющую игроку наносить 1 урон случайному сегменту случайного корабля на игровом поле. Реализует логику выбора атакуемого сегмента, нанесения урона и открытия клеток при уничтожении корабля.

Методы класса:

1. **`bool apply(GameField &field) override;`**

Реализует метод из интерфейса `Ability`. Применяет способность к игровому полю, случайным образом выбирая атакуемый сегмент

корабля. Возвращает true, если корабль был уничтожен, и false в противном случае.

- **Класс DoubleDamage**

Описание:

Представляет собой способность "Двойной урон", которая позволяет игроку нанести двойной урон следующей атаке по выбранной клетке игрового поля. Реализует логику применения способности и повторного нанесения урона по одной и той же клетке.

Поля класса:

1. **int x:**

Координата по оси X клетки, на которую будет применена способность.

2. **int y:**

Координата по оси Y клетки, на которую будет применена способность.

Методы класса:

1. **DoubleDamage(int x, int y):**

Конструктор, инициализирующий координаты клетки, на которую будет направлена способность.

2. **bool apply(GameField &field) override;**

Реализует метод из интерфейса Ability. Применяет способность к указанной клетке игрового поля, нанося урон дважды. Возвращает true, если корабль уничтожен.

- **Класс Scanner**

Описание:

Представляет собой способность "Сканер", которая позволяет игроку проверить участок игрового поля размером 2x2 клетки на наличие сегментов корабля. Не изменяет статус клеток, но выводит информацию о наличии или отсутствии кораблей в указанной области.

Поля класса:

1. **int x:**

Координата по оси X для начала проверки участка 2x2.

2. **int y:**

Координата по оси Y для начала проверки участка 2x2.

Методы класса:

1. **Scanner(int x, int y):**

Конструктор, инициализирующий координаты верхнего левого угла проверяемого участка.

2. **bool apply(GameField &field) override;**

Реализует метод из интерфейса Ability. Проверяет участок 2x2 на наличие сегментов корабля и выводит результаты проверки.

Возвращает true, если хотя бы один сегмент корабля был обнаружен и он не разрушен.

- **Класс AbilityBuilder**

Описание:

Представляет собой класс-интерфейс для создания различных способностей, которые игрок может применять в игре. Определяет основные методы для построения способности, получения её типа, установки координат и вывода информации о способности. Классы-

наследники должны реализовать конкретную логику создания способностей.

Методы класса:

1. **`std::unique_ptr<Ability> build() const = 0;`**

Чисто виртуальный метод, который должен быть реализован в классах-наследниках. Возвращает уникальный указатель на созданный объект способности.

2. **`AbilityType getType() const = 0;`**

Чисто виртуальный метод, который должен возвращать тип способности (например, DoubleDamage, Scanner или Bombardment).

3. **`void setCoords(int x, int y) = 0;`**

Чисто виртуальный метод, который устанавливает координаты (x, y) для способности. Реализация может различаться в зависимости от типа способности.

4. **`void printInfo() = 0;`**

Чисто виртуальный метод, который выводит информацию о способности. Должен быть реализован в каждом классе-наследнике для отображения специфической информации.

5. **`~AbilityBuilder() = default;`**

Виртуальный деструктор, позволяющий корректно освобождать ресурсы, если класс-наследник будет уничтожен через указатель на базовый класс.

- **Классы DoubleDamageBuilder, ScannerBuilder, BombardmentBuilder**

Описание

Каждый из этих классов отвечает за создание определенного типа способности, которую игрок может использовать в игре. Они предоставляют специализированные методы для настройки координат и построения

соответствующей способности. Каждый класс реализует уникальную логику, соответствующую своей способности, и выводит информацию, специфичную для данного типа способности.

- **Класс `BattleShipException`**

Описание

Является базовым классом для обработки исключений, связанных с игрой "Морской бой". Этот класс наследует стандартное исключение `std::exception` и предоставляет механизм для создания специализированных исключений с конкретными сообщениями об ошибках. Он служит для передачи информации о различных проблемах, возникающих в процессе выполнения игры, таких как ошибки размещения кораблей, атаки за пределами игрового поля и другие.

Поля класса:

1. **`std::string message`**

Строка, содержащая сообщение об ошибке, которое объясняет причину возникновения исключения.

Методы класса:

1. **`explicit BattleShipException(const std::string& msg);`**

Конструктор, который принимает сообщение об ошибке и инициализирует соответствующее поле.

2. **`const char what() const noexcept override;`**

Переопределяет метод `what` из класса `std::exception` для возврата сообщения об ошибке в виде строки. Этот метод информирует пользователя о причине исключения.

- Классы `OutOfBoundsAttackException`, `ShipPlacementException`, `InvalidShipIndexException` и `NoAbilitiesException`

Описание:

Каждый из этих классов наследуется от базового класса `BattleShipException` и предназначен для обработки конкретных ошибок, связанных с игровым процессом. Использование этих классов позволяет упростить отладку и улучшить читаемость кода, так как каждое исключение содержит информацию о типе ошибки, произошедшей во время выполнения игры.

Поля класса:

1. `std::string message`

Строка, содержащая сообщение об ошибке, объясняющее причину возникновения исключения (наследуется от `BattleShipException`).

Методы класса:

1. `explicit OutOfBoundsAttackException(const std::string& msg);`

Конструктор, инициализирующий сообщение об ошибке для случаев, когда атака производится за пределами игрового поля.

2. `explicit ShipPlacementException(const std::string& msg);`

Конструктор, инициализирующий сообщение об ошибке для ситуаций, когда размещение корабля является недопустимым (например, пересечение с другим кораблем или выход за пределы поля).

3. `explicit InvalidShipIndexException(const std::string& msg);`

Конструктор, инициализирующий сообщение об ошибке для случаев, когда используется неверный индекс корабля.

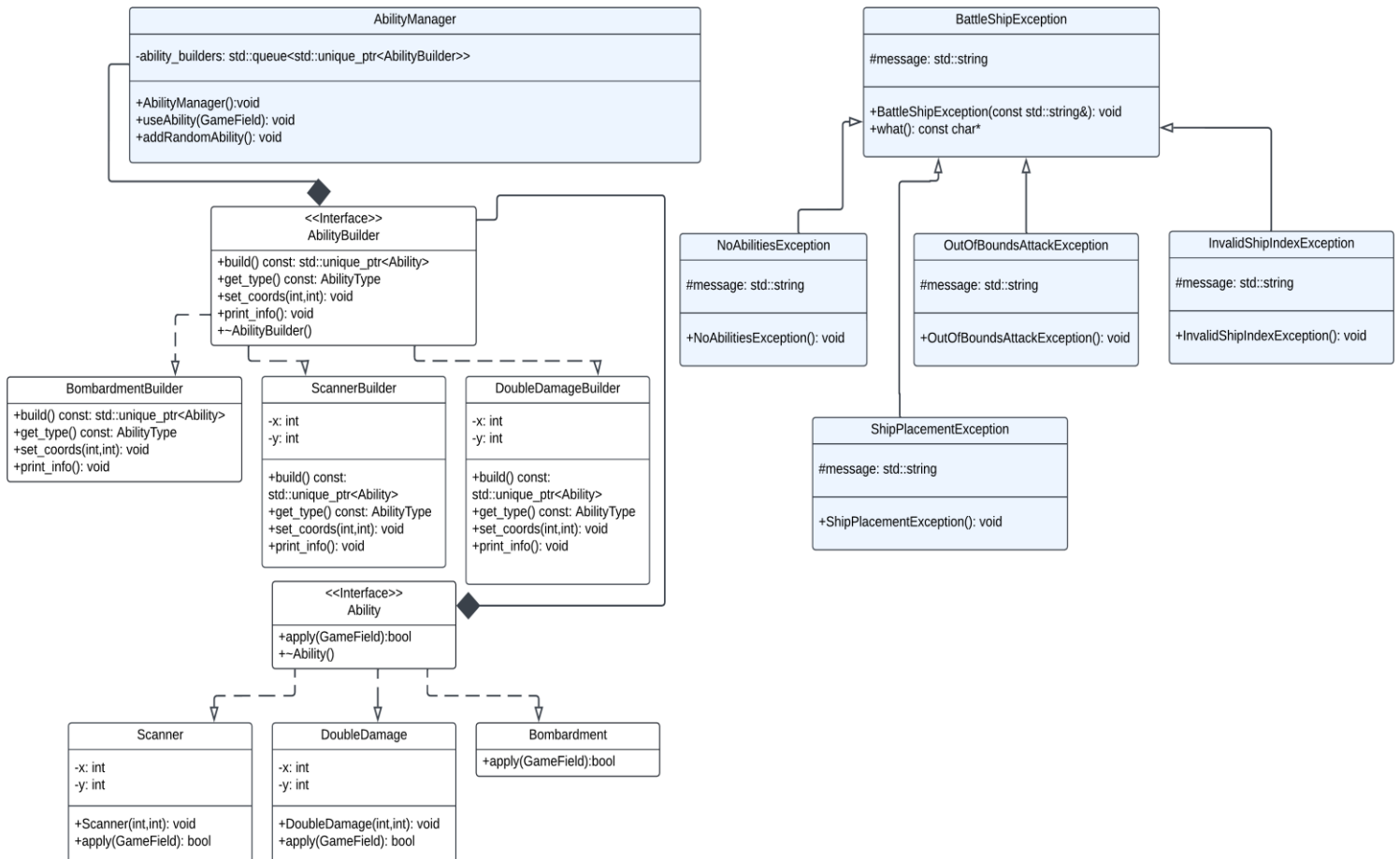
4. `explicit NoAbilitiesException(const std::string& msg);`

Конструктор, инициализирующий сообщение об ошибке для ситуаций, когда попытка применить способность происходит при отсутствии доступных способностей.

Методы what():

Каждый из этих классов переопределяет метод what(), предоставляя уникальное сообщение об ошибке, соответствующее типу исключения.

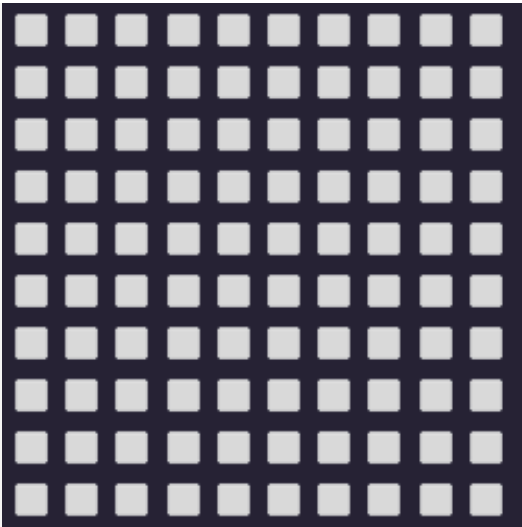
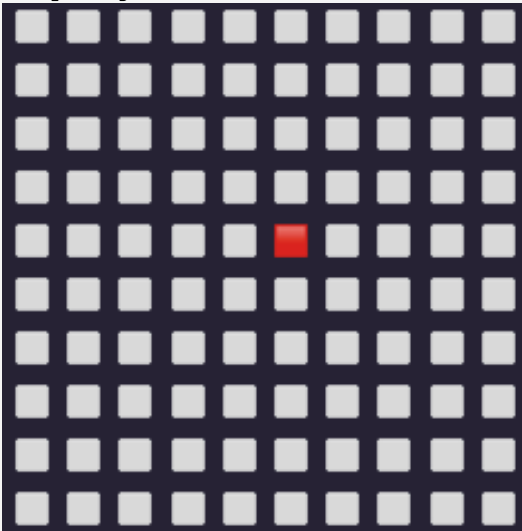
UML – диаграммы классов представлены на картинке ниже



Тестирование

Результаты тестирования представлены в Таблице 1

Таблица 1 - Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<pre> GameField field{ 10, 10}; auto ship_manager = ShipManager(3, {3, 1, 2}); int ships_coords[3][2] = {{5, 4},{1, 4},{9, 1}}; auto &ships = ship_manager.getShips() ; bool positions[] = {false, false, true}; for (int i = 0; i < 3; i++) { try{ field.placeShip(ships[i], ships_coords[i][0], ships_coords[i][1], positions[i]); } catch (ShipPlacementExceptio n& e){ std::cerr << "Error: " << e.what() << std::endl; } } field.show(); try{ field.attackCell(5,4); } catch (OutOfBoundsAttackEx ception& e){ </pre>	 <p>Scanner ability is applied Ship segment detected at: (1, 4)</p> 	Верно

	<pre> std::cerr << "Error: " << e.what() << std::endl; } auto ability_manager = AbilityManager(); try{ ability_manager.useAbili ty(field); } catch (NoAbilitiesException& e){ std::cerr << "Error: " << e.what() << std::endl; } field.show(); </pre>		
2.	<pre> GameField field{ 10, 10}; auto ship_manager = ShipManager(3, {3, 1, 2}); int ships_coords[3][2] = {{5, 4},{1, 4},{9, 1}}; auto &ships = ship_manager.getShips() ; bool positions[] = {false, false, true}; for (int i = 0; i < 3; i++) { try{ field.placeShip(ships[i], ships_coords[i][0], </pre>	<pre> Error: Attack coordinates are out of bounds </pre>	Верно

	<pre>ships_coords[i][1], positions[i]); } catch (ShipPlacementException& e){ std::cerr << "Error: " << e.what() << std::endl; } } field.show(); try{ field.attackCell(10,4); } catch (OutOfBoundsAttackException& e){ std::cerr << "Error: " << e.what() << std::endl; }</pre>		
3.	<pre>GameField field{ 10, 10}; auto ship_manager = ShipManager(3, {3, 1, 2}); int ships_coords[3][2] = {{5, 4},{1, 4},{10, 1}}; auto &ships = ship_manager.getShips() ; bool positions[] = {false, false, true}; for (int i = 0; i < 3; i++) { try{ field.placeShip(ships[i], ships_coords[i][0], ships_coords[i][1], positions[i]);</pre>	Error: Cannot place ship at the given coordinates.	верно

	<pre> } catch (ShipPlacementException& e){ std::cerr << "Error: " << e.what() << std::endl; } }ships_coords[i][1], positions[i]); } field.attackCell(1, 4); field.show(); field.attackCell(1, 4); field.show(); </pre>		
--	--	--	--

Исходный код программы см. в Приложении А.

Вывод:

В ходе выполнения лабораторной работы была реализована возможность применения различных типов способностей, с применением принципов полиморфизма и наследования, а также обеспечена эффективная обработка исключительных ситуаций.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ.

Ability.h:

```
#ifndef OOP_LAB2_ABILITY_H
#define OOP_LAB2_ABILITY_H
#include "GameField.h"

class Ability {
public:
    virtual bool apply(GameField &field) = 0;
    virtual ~Ability() = default;
};

#endif //OOP_LAB2_ABILITY_H
```


Bombardment.cpp:

```
#include "Bombardment.h"

bool Bombardment::apply(GameField &field) {
    std::vector<std::pair<int, int>> attackable_segments;
    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            auto &cell = field.getCell(x, y);
            auto *pointer_to_ship = cell.getPointerToShip();
            if (pointer_to_ship != nullptr &&
                pointer_to_ship-
>getSegmentState(cell.getIndexOfSegment()) != SegmentState::Destroyed)
            {
                attackable_segments.emplace_back(x, y);
            }
        }
    }
    if (attackable_segments.empty()) {
        return false;
    }
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(0, attackable_segments.size()
- 1);
    auto [target_x, target_y] = attackable_segments[dist(gen)];
    auto &target_cell = field.getCell(target_x, target_y);
    auto *target_ship = target_cell.getPointerToShip();
    auto index = target_cell.getIndexOfSegment();
    target_ship->takeDamage(index);
    if (target_ship->isShipDestroyed()) {
        if (target_ship->isVertical()) {
            for (int y = target_y-index; y < target_y-index+target_ship-
>getLength(); y++) {
                field.getCell(target_x, y).open();
            }
        }
        else {
            for (int x = target_x-index; x < target_x-index+target_ship-
>getLength(); x++) {
                field.getCell(x, target_y).open();
            }
        }
    }
    return target_ship->isShipDestroyed();
}
```

Bombardment.h:

```
#ifndef OOP_LAB2_BOMBARDMENT_H
#define OOP_LAB2_BOMBARDMENT_H

#include "Ability.h"
#include <vector>
#include <random>

class Bombardment : public Ability {
public:
    bool apply(GameField &field) override;
};

#endif //OOP_LAB2_BOMBARDMENT_H
```

DoubleDamage.cpp:

```
#include "DoubleDamage.h"
```

```
DoubleDamage::DoubleDamage(int x, int y) : x(x), y(y) {}
```

```
bool DoubleDamage::apply(GameField &field) {  
    field.attackCell(x, y);  
    return field.attackCell(x, y);  
}
```

DoubleDamage.h:

```
#ifndef OOP_LAB2_DOUBLEDAMAGE_H
#define OOP_LAB2_DOUBLEDAMAGE_H
#include "Ability.h"

class DoubleDamage : public Ability {
private:
    int x, y;
public:
    DoubleDamage(int x, int y);
    bool apply(GameField &field) override;
};

#endif //OOP_LAB2_DOUBLEDAMAGE_H
```

Scanner.cpp:

```
#include "Scanner.h"

Scanner::Scanner(int x, int y) : x(x), y(y) {}

bool Scanner::apply(GameField &field) {
    int width = field.getWidth();
    int height = field.getHeight();
    bool flag = false;
    if (x == width - 2) x -= 1;
    if (y == height - 2) y -= 1;

    for (int i = y; i < y + 2; ++i) {
        for (int j = x; j < x + 2; ++j) {
            auto& cell = field.getCell(j,i);
            if (cell.getStatus() == Status::Occupied) {
                int index = cell.getIndexOfSegment();
                if (cell.getPointerToShip()->getSegmentState(index) !=
SegmentState::Destroyed){
                    flag = true;
                    std::cout << "Ship segment detected at: (" << j <<
", " << i << ")\n";
                }
            }
        }
    }
    if (!flag) {
        std::cout << "Ship segment not detected in this area\n";
    }
    return flag;
}
```

Scanner.h:

```
#ifndef OOP_LAB2_SCANNER_H
#define OOP_LAB2_SCANNER_H

#include "Ability.h"
#include <iostream>

class Scanner : public Ability {
private:
    int x, y;
public:
    Scanner(int x, int y);
    bool apply(GameField &field) override;
};

#endif //OOP_LAB2_SCANNER_H
```

AbilityBuilder.h:

```
#ifndef OOP_LAB2_ABILITYBUILDER_H
#define OOP_LAB2_ABILITYBUILDER_H
#include "Ability.h"
enum class AbilityType {
    DoubleDamage,
    Scanner,
    Bombardment
};

class AbilityBuilder {
public:
    virtual std::unique_ptr<Ability> build() const = 0;

    virtual AbilityType getType() const = 0;

    virtual void setCoords(int x, int y) = 0;

    virtual void printInfo() = 0;

    virtual ~AbilityBuilder() = default;
};
#endif //OOP_LAB2_ABILITYBUILDER_H
```

BombardmentBuilder.cpp:

```
#include "BombardmentBuilder.h"

void BombardmentBuilder::setCoords(int, int) {}

std::unique_ptr<Ability> BombardmentBuilder::build() const {
    return std::make_unique<Bombardment>();
}

AbilityType BombardmentBuilder::getType() const {
    return AbilityType::Bombardment;
}

void BombardmentBuilder::printInfo() {
    std::cout<<"Bombardment ability is applied"<<"\n";
}
```

BombardmentBuilder.h:

```
#ifndef OOP_LAB2_BOMBARDMENTBUILDER_H
#define OOP_LAB2_BOMBARDMENTBUILDER_H

#include "AbilityBuilder.h"
#include "Bombardment.h"

class BombardmentBuilder : public AbilityBuilder {
public:
    void setCoords(int, int) override;

    std::unique_ptr<Ability> build() const override;

    void printInfo() override;

    AbilityType getType() const override;
};

#endif //OOP_LAB2_BOMBARDMENTBUILDER_H
```


DoubleDamageBuilder.cpp:

```
#include "DoubleDamageBuilder.h"

void DoubleDamageBuilder::setCoords(int x, int y) {
    this->x = x;
    this->y = y;
}

std::unique_ptr<Ability> DoubleDamageBuilder::build() const {
    return std::make_unique<DoubleDamage>(x, y);
}

AbilityType DoubleDamageBuilder::getType() const {
    return AbilityType::DoubleDamage;
}

void DoubleDamageBuilder::printInfo(){
    std::cout<<"Double damage ability is applied"<<'\\n';
}
```

DoubleDamageBuilder.h:

```
#ifndef OOP_LAB2_DOUBLEDAMAGEBUILDER_H
#define OOP_LAB2_DOUBLEDAMAGEBUILDER_H

#include "AbilityBuilder.h"
#include "DoubleDamage.h"

class DoubleDamageBuilder : public AbilityBuilder {
private:
    int x, y;
public:
    void setCoords(int x, int y) override;

    std::unique_ptr<Ability> build() const override;

    void printInfo() override;

    AbilityType getType() const override;
};

#endif //OOP_LAB2_DOUBLEDAMAGEBUILDER_H
```

ScannerBuilder.cpp:

```
#include "ScannerBuilder.h"

void ScannerBuilder::setCoords(int x, int y) {
    this->x = x;
    this->y = y;
}

std::unique_ptr<Ability> ScannerBuilder::build() const {
    return std::make_unique<Scanner>(x, y);
}

AbilityType ScannerBuilder::getType() const {
    return AbilityType::Scanner;
}

void ScannerBuilder::printInfo() {
    std::cout<<"Scanner ability is applied"<<"\n";
}
```

ScannerBuilder.h:

```
#ifndef OOP_LAB2_SCANNERBUILDER_H
#define OOP_LAB2_SCANNERBUILDER_H

#include "AbilityBuilder.h"
#include "Scanner.h"

class ScannerBuilder : public AbilityBuilder {
private:
    int x, y;
public:
    void setCoords(int x, int y) override;

    std::unique_ptr<Ability> build() const override;

    void printInfo() override;

    AbilityType getType() const override;
};

#endif //OOP_LAB2_SCANNERBUILDER_H
```

BattleShipException.h:

```
#ifndef BATTLESHIP_EXCEPTION_H
#define BATTLESHIP_EXCEPTION_H
#include <exception>
#include <string>

class BattleShipException : public std::exception {
protected:
    std::string message;

public:
    explicit BattleShipException(const std::string& msg) :
message(msg) {}
    const char* what() const noexcept override { return
message.c_str(); }
};

#endif // BATTLESHIP_EXCEPTION_H
```

InvalidShipIndexException.h:

```
#ifndef OOP_LAB2_INVALIDSHIPINDEXEXCEPTION_H
#define OOP_LAB2_INVALIDSHIPINDEXEXCEPTION_H
#include "BattleShipException.h"

class InvalidShipIndexException : public BattleShipException {
public:
    InvalidShipIndexException() : BattleShipException("Invalid ship
index") {}
};

#endif //OOP_LAB2_INVALIDSHIPINDEXEXCEPTION_H
```

NoAbilitiesException.h:

```
#ifndef OOP_LAB2_NOABILITIESEXCEPTION_H
#define OOP_LAB2_NOABILITIESEXCEPTION_H
#include "BattleShipException.h"

class NoAbilitiesException : public BattleShipException {
public:
    NoAbilitiesException() : BattleShipException("No abilities
available to use") {}
};

#endif
```

OutOfBoundsAttackException.h:

```
#ifndef OOP_LAB2_OUTOFBOUNDSATTACKEXCEPTION_H
#define OOP_LAB2_OUTOFBOUNDSATTACKEXCEPTION_H
#include "BattleShipException.h"

class OutOfBoundsAttackException : public BattleShipException {
public:
    OutOfBoundsAttackException()
        : BattleShipException("Attack coordinates are out of
bounds") {}
};

#endif
```


ShipPlacementException.h:

```
#ifndef OOP_LAB2_SHIPPLACEMENTEXCEPTION_H
#define OOP_LAB2_SHIPPLACEMENTEXCEPTION_H
#include "BattleShipException.h"

class ShipPlacementException : public BattleShipException {
public:
    ShipPlacementException()
        : BattleShipException("Cannot place ship at the given
coordinates.") {}
};
#endif
```