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: Добавления игрока и элементов для поля

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2020

Цель работы.

Изучить парадигму объектно-ориентрированного программирования; реализовать класс игрока и классы элементов игрового поля; изучить и реализовать паттерны проектирования *FactoryMethod* и *Strategy*.

Задание.

Создан класс игрока, которым управляет пользователь. Объект класса игрока может перемещаться по полю, а также взаимодействовать с элементами поля. Для элементов поля должен быть создан общий интерфейс и должны быть реализованы 3 разных класса элементов, которые по разному взаимодействуют с игроком. Для взаимодействия игрока с элементом должен использоваться перегруженный оператор *(Например, оператор +)*. Элементы поля могут добавлять очки игроку/замедлять передвижения/и.т.д.

Обязательные требования:

- Реализован класс игрока;
- Реализованы три класса элементов поля;
- Объект класса игрока появляется на клетке со входом;
- Уровень считается пройденным, когда объект класса игрока оказывается на клетке с выходом (и при определенных условиях: например, набрано необходимое кол-во очков);
- Взаимодействие с элементами происходит через общий интерфейс;
- Взаимодействие игрока с элементами происходит через перегруженный оператор.

Дополнительные требования:

- Для создания элементов используется паттерн *Фабричный метод/Абстрактная фабрика;*

- Реализовано динамическое изменение взаимодействия игрока с элементами через паттерн *Стратегия*. Например, при взаимодействии с определенным количеством элементов, игрок не может больше с ними взаимодействовать.

Выполнение работы.

Для начала были реализованы абстрактные классы *Creature* и *Object*. Класс *Creature* представляет собой базовый класс для живых сущностей (например, для классов игрока или врагов). Класс *Object* представляет собой базовый класс для элементов игрового поля (например, для класса медикаментов или брони). Далее были реализованы классы, производные от *Creature* и *Object*: *Medicines*, *Armor*, *Weapon*, *LevelPassObject* — классы элементов поля и *Player* — класс игрока. Для управления игрой был реализован класс *GameController*, через интерфейс которого происходит управление персонажем и самой игрой.Для создания экземпляров классов *Medicines*, *Armor*, *Weapon*, *LevelPassObject* был использован паттерн проектирования *FactoryMethod*: реализованы классы *ObjectFactory*, *MedicinesFactory*, *ArmorFactory*, *WeaponFactory* и *LevelPassObjectFactory*. Также реализовано динамическое изменение взаимодействия игрока с элементами поля при помощи паттерна проектирования *Strategy*: написаны классы *InteractionStrategy*, *InteractionStrategyUse* и *InteractionStrategyNone*.

В программе используются умные указатели, поэтому очистка памяти для них не требуется. Для реализации GUI-интерфейса программы был использован фреймворк *Qt*.

Подробное описание классов, структур и перечислений приведено ниже (см. разделы *Описание перечислений* и *Описание классов и структур*).

Разработанный программный код см. в приложении А.

Описание перечислений.

Перечисление Direction.

Хранит направление в двумерной плоскости. Используется для хранения поворота персонажа, а также направления движения. Существуют 4 направления:

- *kDirectionTop* — верх.
- *kDirectionLeft* — лево.
- *kDirectionRight* — право.
- *kDirectionBottom* — низ.

Описание классов и структур.

Класс Object.

Абстрактный класс. Используется в качестве общего интерфейса для элементов игрового поля. Является базовым для классов *Medicines*, *Armor*, *Weapon*, *LevelPassObject.* Объекты данного класса могут находится на клетках игрового поля, а также принимать взаимодействия от класса *Creature* и его наследников, тем самым изменяя их параметры. Позволяет получить такие характеристики объекта, как: класс, текстура и его многоразовое использование (если значение равно false, то после взаимодействия игрока объект удаляется с поля).

Методы класса *Object*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*getCopy() const = 0*
public	*const std::type_info&*	*getClass() const = 0*
public	*Texture*	*getTexture() const = 0*
public	*void*	*executeInteraction(Creature& creature) = 0*

public	*bool*	*getReusable() const = 0*
public	-	*~Object() = default*

Класс Creature.

Абстрактный класс. Используется в качестве базового для классов живых сущностей игрового поля (например, *Player*). Хранит информацию о параметрах живого объекта: здоровье, атака, защита, позиция на поле и т.д.

Поля класса *Creature*:

Модификатор доступа	Название и тип поля	Предназначение	Значение по умолчанию
private	*int health_*	Хранит количество очков здоровью у существа.	-
private	*int maxHealth_*	Хранит максимальное количество очков здоровья у существа.	-
private	*int attackDamage_*	Хранит количество очков урона существа.	-
private	*int protection_*	Хранит количество очков защиты у существа	-
private	*Position2D position_*	Хранит позицию существа на игровом поле	-
private	*Rotation rotation_*	Хранит ориентацию существа на игровом поле.	*kDirectionBottom*

Методы класса *Creature*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*void*	*interact(pObject& object) = 0*
public	*Texture*	*getTexture() const = 0*
public	-	*~Creature() = default*

public	*Rotation*	*getRotation() const*
public	*Position2D*	*getPosition() const*
public	*int*	*getHealth() const*
public	*int*	*getMaxHealth() const*
public	*int*	*getAttackDamage() const*
public	*int*	*getProtection() const*
public	*void*	*setRotation(Rotation rotation)*
public	*void*	*setPosition(Position2D position)*
public	*void*	*setHealth(int health)*
public	*void*	*setMaxHealth(int maxHealth)*
public	*void*	*setAttackDamage(int damage)*
public	*void*	*setProtection(int protection)*

Класс Armor.

Представляет собой класс элемента поля — броня. Наследуется от класса *Object*. При взаимодействии игрока с объектами данного типа происходит увеличение очков защиты у игрока до определенного уровня.

Поля класса *Armor*:

Модификатор доступа	Название и тип поля	Предназначение	Значение по умолчанию
private	*int protectionValue_*	Хранит значение очков защиты данной брони.	-

Методы класса *Armor*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	-	*Armor(int protectionValue)*
public	*pObject*	*getCopy() const = 0*
public	*const std::type_info&*	*getClass() const = 0*
public	*Texture*	*getTexture() const = 0*

public	*void*	*executeInteraction(Creature& creature) = 0*
public	*bool*	*getReusable() const = 0*

Класс Medicines.

Представляет собой класс элемента поля — медикаменты. Наследуется от класса *Object*. При взаимодействии игрока с объектами данного типа происходит увеличение очков здоровья у игрока на определенный уровень.

Поля класса *Medicines*:

Модификатор доступа	Название и тип поля	Предназначение	Значение по умолчанию
private	*int healthRecovery_*	Хранит количество регенерируемых очков здоровья данных медикаментов.	-

Методы класса *Medicines*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	-	*Medicines(int healthRecovery)*
public	*pObject*	*getCopy() const = 0*
public	*const std::type_info&*	*getClass() const = 0*
public	*Texture*	*getTexture() const = 0*
public	*void*	*executeInteraction(Creature& creature) = 0*
public	*bool*	*getReusable() const = 0*

Класс Weapon.

Представляет собой класс элемента поля — оружие. Наследуется от класса *Object*. При взаимодействии игрока с объектами данного типа происходит увеличение очков урона у игрока до определенного уровня.

Поля класса *Weapon*:

Модификатор доступа	Название и тип поля	Предназначение	Значение по умолчанию
private	*int damage_*	Хранит значение очков урона у данного оружия.	-

Методы класса *Weapon*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	-	Weapon(int damage)
public	*pObject*	*getCopy() const = 0*
public	*const std::type_info&*	*getClass() const = 0*
public	*Texture*	*getTexture() const = 0*
public	*void*	*executeInteraction(Creature& creature) = 0*
public	*bool*	*getReusable() const = 0*

Класс LevelPassObject.

Представляет собой класс элемента поля — пропуск для выхода с уровня. Наследуется от класса *Object*. Для того, чтобы игрок смог завершить уровень на клекте выхода, ему необходимо провзаимодействовать с данным объектом (подобрать его для использовании на клетке выхода), при этом у объекта класса *Player* установится определенный флаг, означающий, что игрок подобрал пропуск.

Методы класса *LevelPassObject*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*getCopy() const = 0*

public	*const std::type_info&*	*getClass() const = 0*
public	*Texture*	*getTexture() const = 0*
public	*void*	*executeInteraction(Creature& creature) = 0*
public	*bool*	*getReusable() const = 0*

Класс Player.

Используется для представления класса игрока, которым управляет пользователь. Может перемещаться по полю и взаимодействовать с объектами игрового поля.

Поля класса *Player*:

Модификатор доступа	Название и тип поля	Предназначение	Значение по умолчанию
private	*bool passFounded_*	Хранит информацию о том, был ли найден пропуск на выход с уровня.	*false*
private	*pInteractionStrategy objectInteractionStrategy_*	Хранит текущую стратегию взаимодействия с объектами поля.	-

Методы класса *Player*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	-	*Player(Position2D position)*
public	*void*	*interact(pObject& object)*
public	*Texture*	*getTexture() const*
public	*void*	*operator<=(pObject& object)*
public	*bool*	*getPassFounded() const*
public	*void*	*setPassFounded(bool value)*

Класс InteractionStrategy.

Используется для реализации паттерна Стратегия. Определяет стратегия взаимодействия игрока с объектами поля. Используется в качестве базового интерфейса для классов *InteractionUse* и *InteractionNone*.

Методы класса *InteractionStrategy*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*void*	*interact(Creature& creature, pObject& object)*
public	-	*~InteractionStrategy() = default*

Класс InteractionUse.

Используется для реализации паттерна Стратегия. Определяет для игрока взаимодействие «использование» с объектами игрового поля.

Методы класса *InteractionUse*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*void*	*interact(Creature& creature, pObject& object)*

Класс InteractionNone.

Используется для реализации паттерна *Стратегия*. Определяет для игрока пустое взаимодействие с объектами игрового поля (при данной стратегии взаимодействия игрок, по-сути, никак не взаимодействует с объектом).

Методы класса *InteractionNone*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*void*	*interact(Creature& creature, pObject& object)*

Класс ObjectFactory.

Используется для реализации паттерна *Фабричный метод* в качестве базового интерфейса для классов *ArmorFactory, MedicinesFactory, WeaponFactory* и *LevelPassObjectFactory*.

Методы класса *ObjectFactory*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*createObject() = 0*
public	-	*~ObjectFactory() = default*

Класс ArmorFactory.

Используется для реализации паттерна *Фабричный метод*. Позволяет создавать объекты класса *Armor*.

Методы класса *ArmorFactory*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*createObject() = 0*
public	*pObject*	*createArmor(int protectionValue)*

Класс MedicinesFactory.

Используется для реализации паттерна *Фабричный метод*. Позволяет создавать объекты класса *Medicines*.

Методы класса *MedicinesFactory*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*createObject() = 0*
public	*pObject*	*createMedicines(int healthRecovery)*

Класс WeaponFactory.

Используется для реализации паттерна *Фабричный метод*. Позволяет создавать объекты класса *Weapon*.

Методы класса *WeaponFactory*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*createObject() = 0*
public	*pObject*	*createWeapon(int damage)*

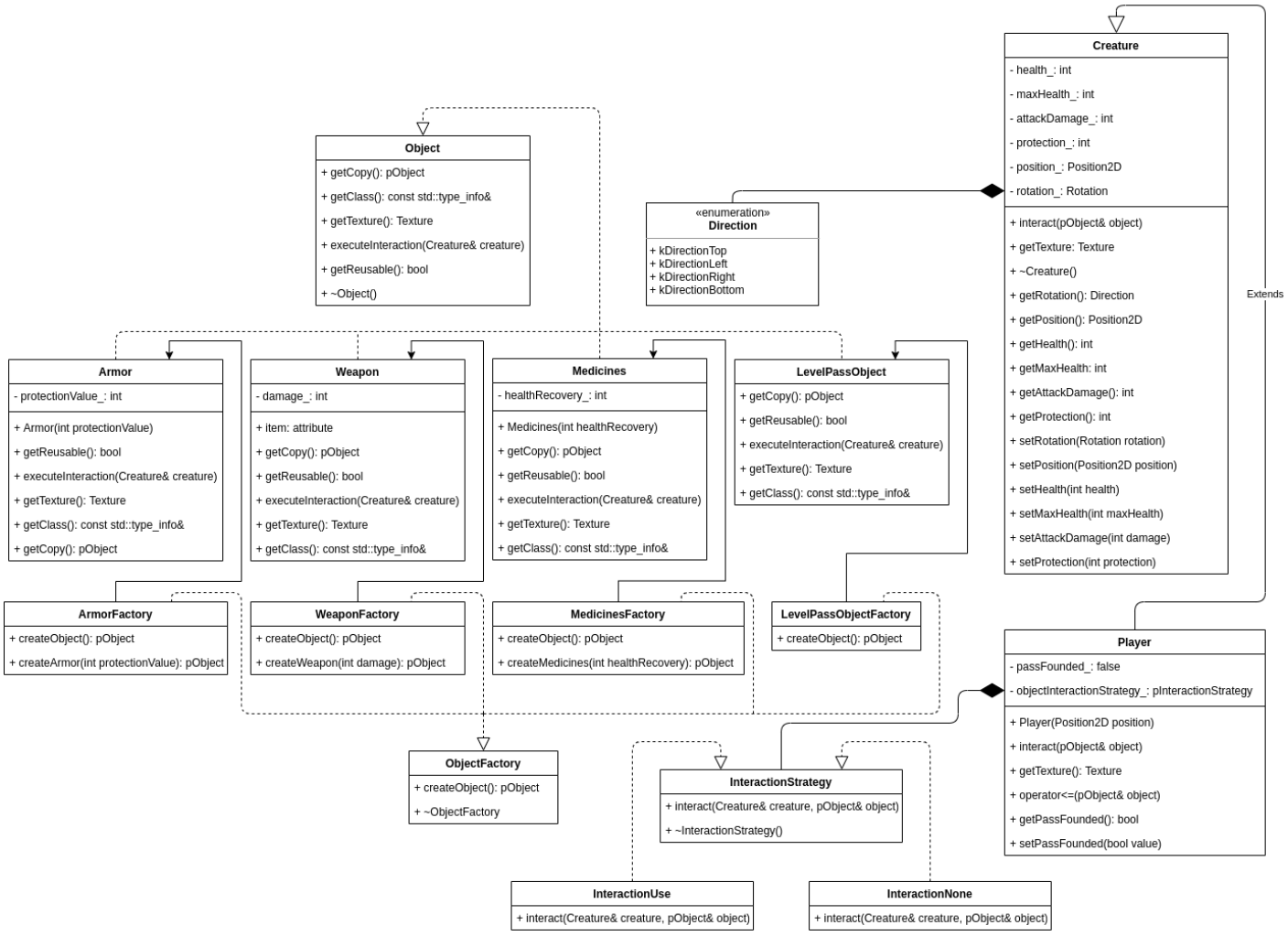
Класс LevelPassObjectFactory.

Используется для реализации паттерна *Фабричный метод*. Позволяет создавать объекты класса *LevelPassObject*.

Методы класса *LevelPassObjectFactory*:

Модификатор доступа	Возвращаемое значение	Название метода и принимаемые аргументы
public	*pObject*	*createObject() = 0*

UML-диаграмма.



Тестирование.

Результаты тестирования представлены на рис. 1, 2, 3, 4.

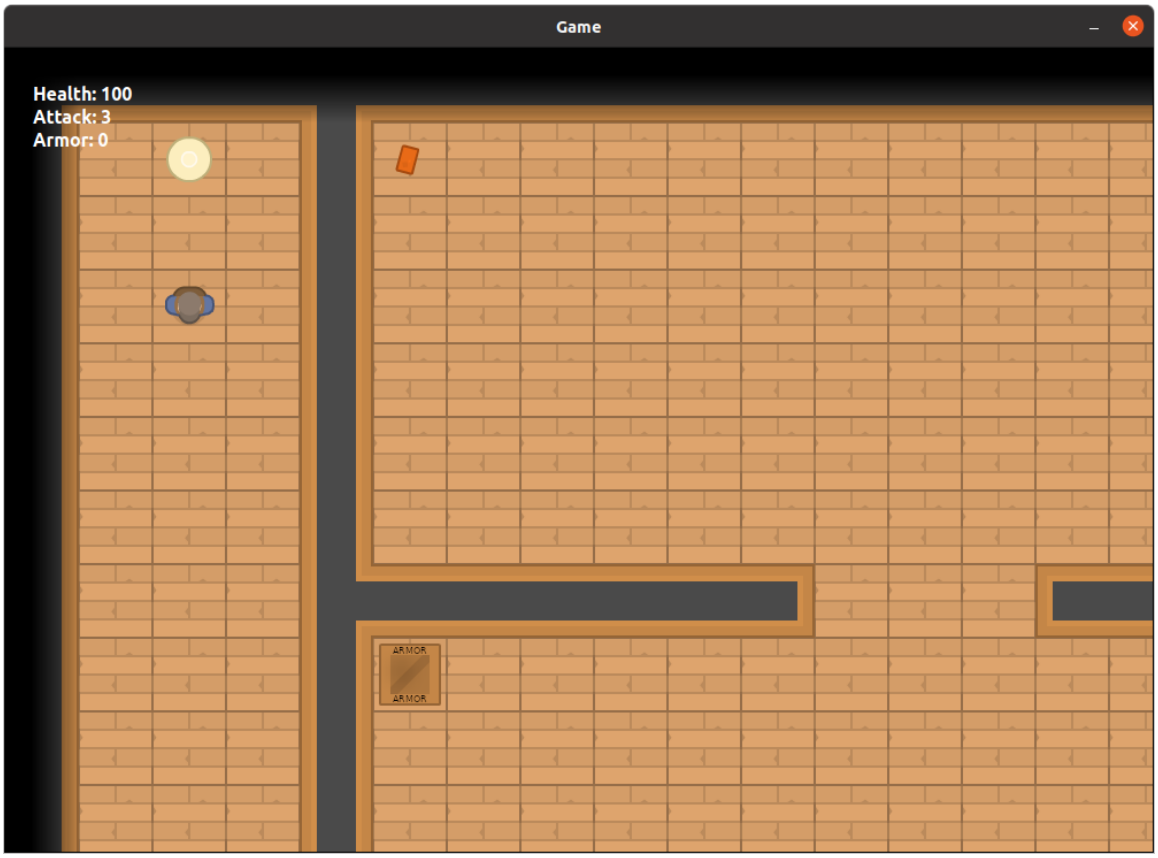


Рисунок 1 — Игровое поле с игроком и элементами поля

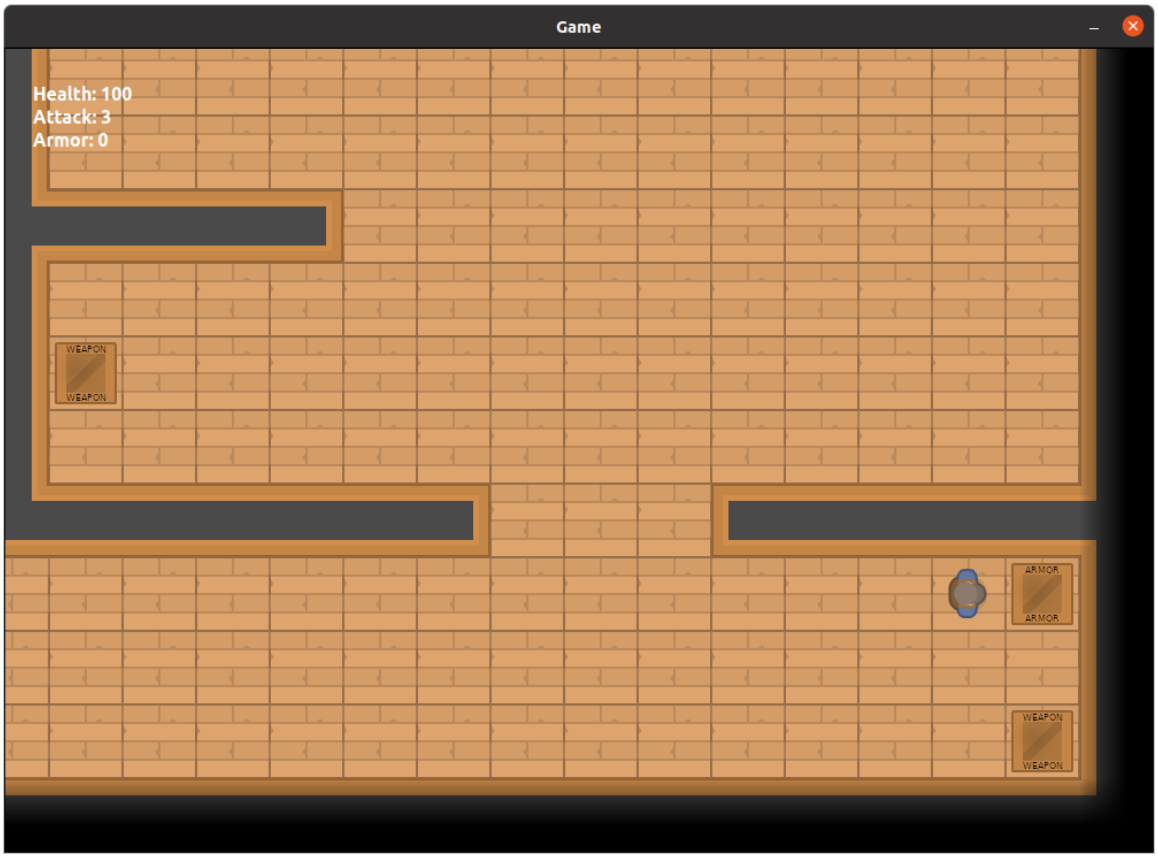


Рисунок 2 — До взаимодействи с элементом

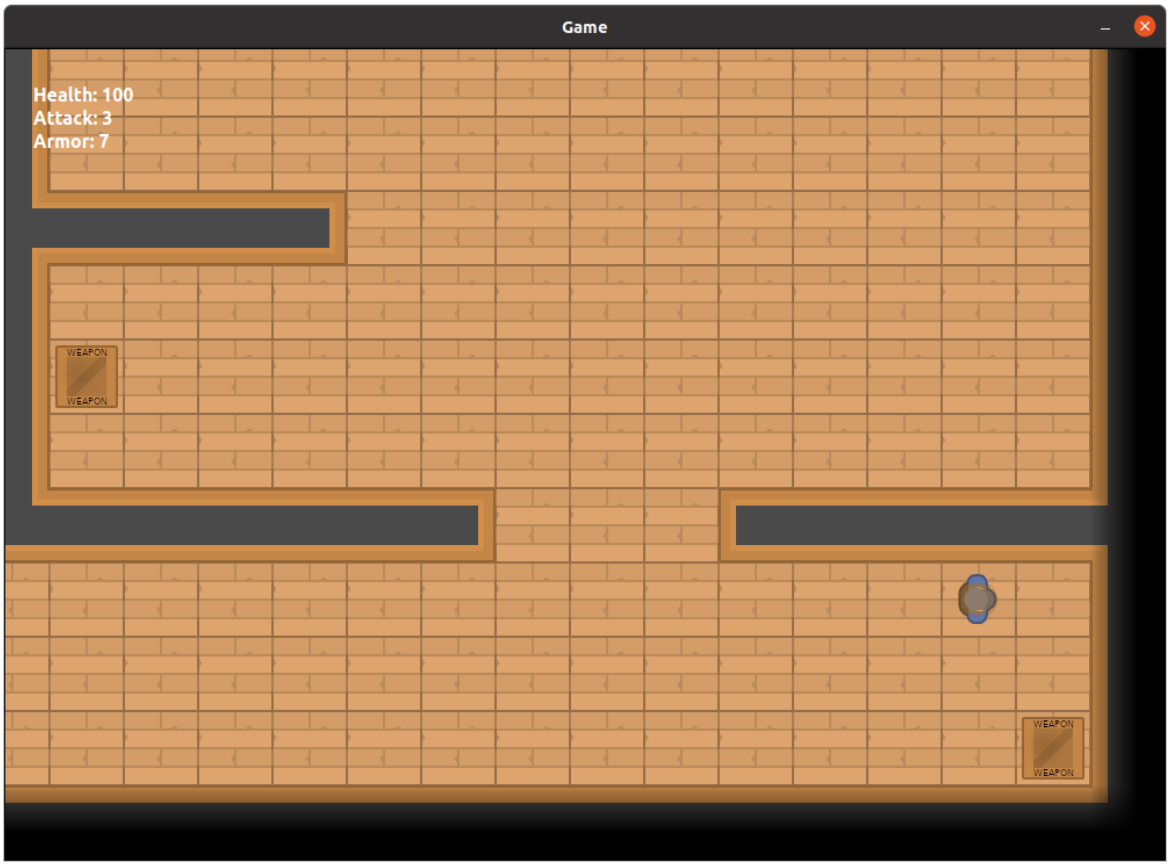


Рисунок 3 — После взаимодействия с элементом

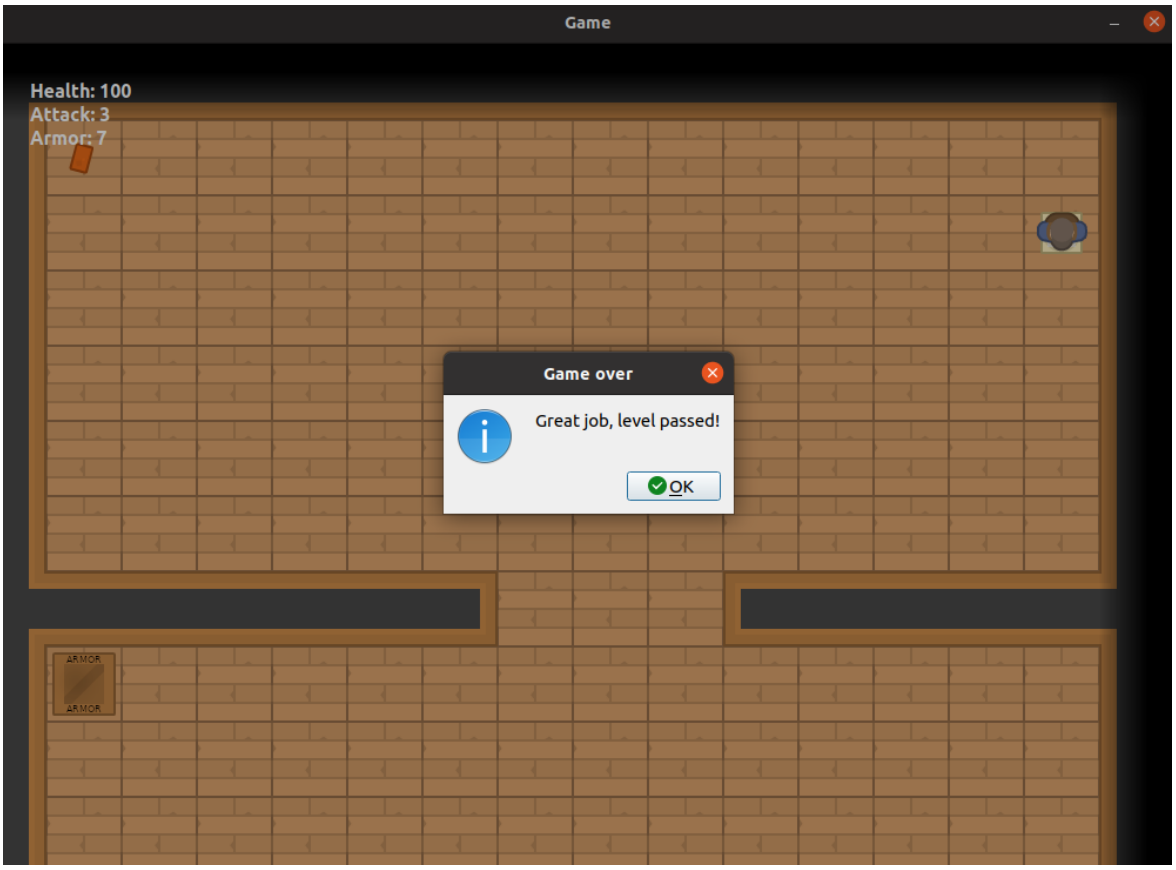


Рисунок 4 — Игрок дошел до выхода, выполнив необходимые условия

Выводы.

Была изучена парадигма объектно-ориентрированного программирования. Были реализованны класс игрока и классы элементов игрового поля. Для классов элементов игрового поля был создан общий интерфейс. Для класса игрока был перегружен оператор <= для взаимодействия с элементами игрового поля. Были изучены и реализованы паттерны проектирования *FactoryMethod* и *Strategy*. По мимо этого, был реализован GUI-интерфейс игры при помощи фреймворка Qt.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <QApplication>
#include "classes/mainwindow.h"

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    MainWindow window;
    window.show();
    return app.exec();
}
```

Название файла: armor.h

```
#ifndef ARMOR_H
#define ARMOR_H

#include "memory"
#include "object.h"

typedef std::shared_ptr<class Armor> pArmor;


class Armor: public Object {
private:
    int protectionValue_;

public:
    explicit Armor(int protectionValue);
    pObject getCopy() const;
    void executeInteraction(Creature& creature);
    const std::type_info& getClass() const;
    Texture getTexture() const;
    bool getReusable() const;
};


#endif // ARMOR_H
```

Название файла: armor.cpp

```
#include "armor.h"
#include "armorfactory.h"

Armor::Armor(int protectionValue): protectionValue_(protectionValue) {}

pObject Armor::getCopy() const {
    pArmorFactory factory(new ArmorFactory);
    return pObject(factory->createArmor(protectionValue_));
}

void Armor::executeInteraction(Creature& creature) {
    if (creature.getProtection() < protectionValue_) {
        creature.setProtection(protectionValue_);
```

```
    }
}

const std::type_info &Armor::getClass() const {
    return typeid(Armor);
}

Texture Armor::getTexture() const {
    return kTextureObjectArmor;
}

bool Armor::getReusable() const {
    return false;
}
```

Название файла: armorfactory.h

```
#ifndef ARMOR_FACTORY_H
#define ARMOR_FACTORY_H

#include "objectfactory.h"
#include "armor.h"

typedef std::shared_ptr<class ArmorFactory> pArmorFactory;


class ArmorFactory: public ObjectFactory {
public:
    virtual pObject createObject();
    virtual pObject createArmor(int protectionValue);
};


#endif // ARMOR_FACTORY_H
```

Название файла: armorfactory.cpp

```
#include "armorfactory.h"

pObject ArmorFactory::createObject() {
    return pObject(new Armor(5));
}

pObject ArmorFactory::createArmor(int protectionValue) {
    return pObject(new Armor(protectionValue));
}
```

Название файла: cell.h

```
#ifndef CELL_H
#define CELL_H

#include <memory>
#include "point2d.h"
#include "celltype.h"
#include "texture.h"
#include "object.h"
```

```
typedef std::shared_ptr<class Cell> pCell;
typedef std::shared_ptr<std::shared_ptr<class Cell>> ppCell;


class Cell {
private:
    bool passable_ = false;
    CellType type_ = kCellTypeNone;
    Texture texture_ = kTextureVoid;
    Position2D position_;
    pObject object_;

public:
    Cell() = default;
    explicit Cell(Position2D position, Texture texture = kTextureVoid,
CellType type = kCellTypeNone, pObject object = nullptr);
    Cell(const Cell& other);
    Cell(Cell&& other);
    ~Cell() = default;

    Cell& operator=(const Cell& other);
    Cell& operator=(Cell&& other);

    bool isPassable() const;
    bool getPassible() const;
    pConstObject getObject() const;
    Texture getTexture() const;
    CellType getType() const;
    Position2D getPosition() const;
    pObject& getObject();
    void setObject(const pObject& object);
    void setTexture(Texture texture);
    void changeType(CellType type);
};


#endif // CELL_H
```

Название файла: cell.cpp

```
#include "cell.h"

Cell::Cell(Position2D coords, Texture texture, CellType type, pObject
object) {
    position_ = coords;
    texture_ = texture;
    object_ = object;
    changeType(type);
}

Cell::Cell(const Cell& other) {
    operator=(other);
}

Cell::Cell(Cell&& other) {
    position_ = other.position_;
    texture_ = other.texture_;
    type_ = other.type_;
    passable_ = other.passable_;
    object_ = other.object_;
```

```
}

Cell& Cell::operator=(const Cell& other) {
    if (this != &other) {
        position_ = other.position_;
        texture_ = other.texture_;
        type_ = other.type_;
        passable_ = other.passable_;

        if (other.object_ != nullptr) {
            object_ = other.object_->getCopy();
        }
    }

    return *this;
}

Cell& Cell::operator=(Cell&& other) {
    if (this != &other) {
        std::swap(position_, other.position_);
        std::swap(texture_, other.texture_);
        std::swap(type_, other.type_);
        std::swap(passable_, other.passable_);
        std::swap(object_, other.object_);
    }

    return *this;
}

bool Cell::isPassable() const {
    return passable_ && object_ == nullptr;
}

bool Cell::getPassible() const {
    return passable_;
}

pConstObject Cell::getObject() const {
    return object_;
}

Texture Cell::getTexture() const {
    return texture_;
}

CellType Cell::getType() const {
    return type_;
}

Position2D Cell::getPosition() const {
    return position_;
}

pObject& Cell::getObject() {
    return object_;
}

void Cell::setObject(const pObject& object) {
    object_ = object;
}
```

```
void Cell::setTexture(Texture texture) {
    texture_ = texture;
}

void Cell::changeType(CellType type) {
    type_ = type;

    switch (type) {
    case kCellTypeEmpty:
    case kCellTypeEntry:
    case kCellTypeExit:
        passable_ = true;
        return;

    case kCellTypeNone:
    case kCellTypeWall:
    default:
        passable_ = false;
        return;
    }
}
```

Название файла: celltype.h

```
#ifndef CELL_TYPE_H
#define CELL_TYPE_H

enum CellType {
    kCellTypeNone,
    kCellTypeEmpty,
    kCellTypeWall,
    kCellTypeEntry,
    kCellTypeExit
};

#endif // CELL_TYPE_H
```

Название файла: creature.h

```
#ifndef CREATURE_H
#define CREATURE_H

#include <memory>
#include "point2d.h"
#include "direction.h"
#include "texture.h"
#include "interactionstrategy.h"

typedef std::shared_ptr<class Creature> pCreature;
typedef std::shared_ptr<class Object> pObject;


class Creature {
private:
    int health_;
    int maxHealth_;
    int attackDamage_;
    int protection_;
    Position2D position_;
```

```
    Rotation rotation_ = kDirectionBottom;

public:
    virtual void interact(pObject& object) = 0;
    virtual Texture getTexture() const = 0;
    virtual ~Creature() =default;

    Rotation getRotation() const;
    Position2D getPosition() const;
    int getHealth() const;
    int getMaxHealth() const;
    int getAttackDamage() const;
    int getProtection() const;
    void setRotation(Rotation rotation);
    void setPosition(Position2D position);
    void setHealth(int health);
    void setMaxHealth(int maxHealth);
    void setAttackDamage(int damage);
    void setProtection(int protection);
};


#endif // CREATURE_H
```

Название файла: creature.cpp

```
#include "creature.h"

Rotation Creature::getRotation() const {
    return rotation_;
}

Position2D Creature::getPosition() const {
    return position_;
}

int Creature::getHealth() const {
    return health_;
}

int Creature::getMaxHealth() const {
    return maxHealth_;
}

int Creature::getAttackDamage() const {
    return attackDamage_;
}

int Creature::getProtection() const {
    return protection_;
}

void Creature::setRotation(Rotation rotation) {
    rotation_ = rotation;
}

void Creature::setPosition(Position2D position) {
    position_ = position;
}
```

```
void Creature::setHealth(int health) {
    if (health > maxHealth_) {
        health_ = maxHealth_;
    } else {
        health_ = health;
    }
}

void Creature::setMaxHealth(int maxHealth) {
    maxHealth_ = maxHealth;
}

void Creature::setAttackDamage(int damage) {
    attackDamage_ = damage;
}

void Creature::setProtection(int protection) {
    protection_ = protection;
}
```

Название файла: direction.h

```
#ifndef DIRECTION_H
#define DIRECTION_H

enum Direction {
    kDirectionTop,
    kDirectionLeft,
    kDirectionRight,
    kDirectionBottom
};

typedef Direction Rotation;

#endif // DIRECTION_H
```

Название файла: exception.h

```
#ifndef EXCEPTION_H
#define EXCEPTION_H

#include <string>


class Exception {
private:
    std::string error_;

public:
    Exception(const std::string& error);
    const std::string& getError() const;
};


#endif // EXCEPTION_H
```

Название файла: exception.cpp

```
#include "exception.h"

Exception::Exception(const std::string& error): error_(error) {}

const std::string& Exception::getError() const {
    return error_;
}
```

Название файла: field.h

```
#ifndef FIELD_H
#define FIELD_H

#include <memory>
#include "cell.h"
#include "point2d.h"

typedef std::unique_ptr<class Field> pField;


class Field {
private:
    static pField instance_;

    Size2D size_ = Size2D(0, 0);
    ppCell cells_ = nullptr;

    Field(const Size2D& size);
    Field(const Field& other);
    Field(Field&& other);

    Field& operator=(const Field& other);
    Field& operator=(Field&& other);

    class FieldIterator;
    class ConstFieldIterator;

public:
    static Field& initInstance(const Size2D& size);
    static Field& getInstance();
    static void deleteInstance();
    static bool isInstanceCreated();
    Cell& getCell(const Position2D& position);
    const Cell& getCell(const Position2D& position) const;
    Size2D getSize() const;
    FieldIterator begin();
    FieldIterator end();
    const ConstFieldIterator begin() const;
    const ConstFieldIterator end() const;
};


class Field::FieldIterator {
    Position2D position_;

public:
    explicit FieldIterator(const Position2D& position);

    bool operator==(const FieldIterator& other) const;
    bool operator!=(const FieldIterator& other) const;
```

```
        FieldIterator& operator++();
        FieldIterator operator++(int);
        Cell& operator*();
    };

    class Field::ConstFieldIterator {
        Position2D position_;

    public:
        explicit ConstFieldIterator(const Position2D& position);

        bool operator==(const ConstFieldIterator& other) const;
        bool operator!=(const ConstFieldIterator& other) const;
        ConstFieldIterator& operator++();
        ConstFieldIterator operator++(int);
        const Cell& operator*() const;
    };


    #endif // FIELD_H
```

Название файла: field.cpp

```
    #include "field.h"
    #include "exception.h"
    #include <iostream>

    pField Field::instance_ = nullptr;

    Field::Field(const Size2D& size): size_(size) {
        cells_ = ppCell(new pCell[size.y], std::default_delete<pCell[]>());

        for (size_t y = 0; y < size.y; y++) {
            cells_.get()[y] = pCell(new Cell[size.x],
std::default_delete<Cell[]>());

            for (size_t x = 0; x < size.x; x++) {
                cells_.get()[y].get()[x] = Cell(Position2D(x, y));
            }
        }
    }

    Field::Field(const Field& other) {
        size_ = other.size_;

        if (other.cells_ != nullptr) {
            cells_ = ppCell(new pCell[size_.y],
std::default_delete<pCell[]>());

            for (size_t y = 0; y < size_.y; y++) {
                cells_.get()[y] = pCell(new Cell[size_.x],
std::default_delete<Cell[]>());

                for (size_t x = 0; x < size_.x; x++) {
                    cells_.get()[y].get()[x] = other.cells_.get()[y].get()
[x];
                }
            }
        }
    }
```

```
Field::Field(Field&& other) {
    size_ = other.size_;
    cells_ = other.cells_;
}

Field& Field::operator=(const Field& other) {
    if (this != &other) {
        size_ = other.size_;

        if (other.cells_ != nullptr) {
            cells_ = ppCell(new pCell[size_.y],
std::default_delete<pCell[]>());

            for (size_t y = 0; y < size_.y; y++) {
                cells_.get()[y] = pCell(new Cell[size_.x],
std::default_delete<Cell[]>());

                for (size_t x = 0; x < size_.x; x++) {
                    cells_.get()[y].get()[x] = other.cells_.get()
[y].get()[x];
                }
            }
        }
    }

    return *this;
}

Field& Field::operator=(Field&& other) {
    if (this != &other) {
        std::swap(size_, other.size_);
        std::swap(cells_, other.cells_);
    }

    return *this;
}

Field& Field::initInstance(const Size2D& size) {
    if (!isInstanceCreated()) {
        instance_ = pField(new Field(size));
    }
    return *instance_;
}

Field& Field::getInstance() {
    if (!isInstanceCreated()) {
        instance_ = pField(new Field(Size2D(10, 10)));
    }
    return *instance_;
}

void Field::deleteInstance() {
    Field::instance_ = nullptr;
}

bool Field::isInstanceCreated() {
    return Field::instance_ != nullptr;
}

Cell& Field::getCell(const Position2D& position) {
```

```
        if (position.x >= size_.x || position.y >= size_.y) {
            throw Exception("Method Field::getCell. Out of range.");
        }
        return cells_.get()[position.y].get()[position.x];
    }

    const Cell& Field::getCell(const Position2D& position) const {
        if (position.x >= size_.x || position.y >= size_.y) {
            throw Exception("Method Field::getCell. Out of range.");
        }
        return cells_.get()[position.y].get()[position.x];
    }

    Size2D Field::getSize() const {
        return size_;
    }

    Field::FieldIterator Field::begin() {
        return FieldIterator(Position2D(0, 0));
    }

    Field::FieldIterator Field::end() {
        return FieldIterator(Position2D(0, getSize().y));
    }

    const Field::ConstFieldIterator Field::begin() const {
        return ConstFieldIterator(Position2D(0, 0));
    }

    const Field::ConstFieldIterator Field::end() const {
        return ConstFieldIterator(Position2D(0, getSize().y));
    }

    Field::FieldIterator::FieldIterator(const Position2D& position):
position_(position) {}

    bool Field::FieldIterator::operator==(const FieldIterator& other) const
{
        return position_ == other.position_;
    }

    bool Field::FieldIterator::operator!=(const FieldIterator& other) const
{
        return !operator==(other);
    }

    Field::FieldIterator& Field::FieldIterator::operator++() {
        Field& field = Field::getInstance();

        if (position_.x + 1 >= field.getSize().x) {
            position_.y++;
            position_.x = 0;
        } else {
            position_.x++;
        }

        return *this;
    }

    Field::FieldIterator Field::FieldIterator::operator++(int) {
        FieldIterator iterator(*this);
```

```
        operator++();
        return iterator;
    }

    Cell& Field::FieldIterator::operator*() {
        Field& field = Field::getInstance();
        return field.getCell(position_);
    }

    Field::ConstFieldIterator::ConstFieldIterator(const Position2D&
position): position_(position) {}

    bool Field::ConstFieldIterator::operator==(const
Field::ConstFieldIterator& other) const {
        return position_ == other.position_;
    }

    bool Field::ConstFieldIterator::operator!=(const
Field::ConstFieldIterator& other) const {
        return !operator==(other);
    }

    Field::ConstFieldIterator& Field::ConstFieldIterator::operator++() {
        const Field& field = Field::getInstance();

        if (position_.x + 1 >= field.getSize().x) {
            position_.y++;
            position_.x = 0;
        } else {
            position_.x++;
        }

        return *this;
    }

    Field::ConstFieldIterator Field::ConstFieldIterator::operator++(int) {
        ConstFieldIterator iterator(*this);
        operator++();
        return iterator;
    }

    const Cell& Field::ConstFieldIterator::operator*() const {
        const Field& field = Field::getInstance();
        return field.getCell(position_);
    }
```

Название файла: gamecontroller.h

```
    #ifndef GAMECONTROLLER_H
    #define GAMECONTROLLER_H

    #include "player.h"
    #include "field.h"

    typedef std::shared_ptr<const Player> pConstPlayer;


    class GameController {
    private:
        pPlayer player_;
```

```
    bool gameOver_ = false;

public:
    GameController();
    void createFieldMap();
    const Field& getField() const;
    pConstPlayer getPlayer() const;
    bool isPlayerReachedExit() const;
    void movePlayer(Direction direction);
    void executePlayerInteraction();
    bool isGameOver();
};


#endif // GAMECONTROLLER_H
```

Название файла: gamecontroller.cpp

```
#include "gamecontroller.h"
#include "medicinesfactory.h"
#include "weaponfactory.h"
#include "armorfactory.h"
#include "levelpassobjectfactory.h"
#include <iostream>

GameController::GameController() {
    createFieldMap();
}

void GameController::createFieldMap() {
    Field& field = Field::initInstance(Size2D(20, 20));
    pMedicinesFactory medicinesFactory(new MedicinesFactory);
    pArmorFactory armorFactory(new ArmorFactory);
    pWeaponFactory weaponFactory(new WeaponFactory);
    pLevelPassObjectFactory levelPassFactory(new
LevelPassObjectFactory);

    int textureMap[20][20] = {
        {6, 2, 2, 2, 8, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 7},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 12, 2, 2, 2, 2, 2, 9, 1, 1, 1, 13, 2, 2, 2, 2, 14},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 12, 2, 2, 2, 9, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 4, 2, 2, 2, 2, 2, 9, 1, 1, 1, 13, 2, 2, 2, 2, 14},
        {3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3},
        {4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 5}
    };
```

```
            int cellTypeMap[20][20] = {
                {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
                {2, 1, 3, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 2, 2, 2, 2, 2, 2},
                {2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
                {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
            };

            Position2D entryPoint;

            for (Cell& cell : field) {
                Position2D coords = cell.getPosition();
                cell.changeType(static_cast<CellType>(cellTypeMap[coords.y]
[coords.x]));
                cell.setTexture(static_cast<Texture>(textureMap[coords.y]
[coords.x]));

                if (cell.getType() == kCellTypeEntry) {
                    entryPoint = coords;
                }
            }

            player_ = pPlayer(new Player(entryPoint));
            player_->setMaxHealth(100);
            player_->setHealth(100);
            player_->setAttackDamage(3);
            player_->setProtection(0);

            field.getCell(Position2D(2, 17)).setObject(medicinesFactory-
>createMedicines(25));
            field.getCell(Position2D(5, 1)).setObject(medicinesFactory-
>createMedicines(25));
            field.getCell(Position2D(18, 16)).setObject(armorFactory-
>createArmor(7));
            field.getCell(Position2D(18, 18)).setObject(weaponFactory-
>createWeapon(8));
            field.getCell(Position2D(5, 8)).setObject(armorFactory-
>createArmor(10));
            field.getCell(Position2D(5, 13)).setObject(weaponFactory-
>createWeapon(10));
            field.getCell(Position2D(18, 8)).setObject(levelPassFactory-
>createObject());
        }

        const Field& GameController::getField() const {
            return Field::getInstance();
```

```
}

pConstPlayer GameController::getPlayer() const {
    return pConstPlayer(player_);
}

bool GameController::isPlayerReachedExit() const {
    const Field& field = Field::getInstance();
    return field.getCell(player_->getPosition()).getType() ==
kCellTypeExit;
}

void GameController::movePlayer(Direction direction) {
    Field& field = Field::getInstance();
    Position2D newPosition = Position2D(player_->getPosition().x,
player_->getPosition().y);

    newPosition.shift(direction);
    player_->setRotation(direction);

    if (field.getCell(newPosition).isPassable()) {
        player_->setPosition(newPosition);
    }

    if (isPlayerReachedExit() && player_->getPassFounded()) {
        gameOver_ = true;
    }
}

void GameController::executePlayerInteraction() {
    Field& field = Field::getInstance();
    Position2D interactionPosition = player_->getPosition();

    interactionPosition.shift(player_->getRotation());

    Cell& cell = field.getCell(interactionPosition);
    pObject& object = cell.getObject();

    *player_ <= object; // Взаимодействие через оператор <=
}

bool GameController::isGameOver() {
    return gameOver_;
}
```

Название файла: interactionnone.h

```
#ifndef INTERACTION_NONE_H
#define INTERACTION_NONE_H

#include "interactionstrategy.h"

typedef std::shared_ptr<class InteractionNone> pInteractionNone;


class InteractionNone: public InteractionStrategy {
public:
    void interact(Creature& creature, pObject& object);
};
```

```
#endif // INTERACTION_NONE_H
```

Название файла: interactionnone.cpp

```
#include "interactionnone.h"

void InteractionNone::interact(Creature&, pObject&) {}
```

Название файла: interactionstrategy.h

```
#ifndef INTERACTION_STRATEGY_H
#define INTERACTION_STRATEGY_H

#include <memory>
#include "object.h"
#include "creature.h"

typedef std::shared_ptr<class InteractionStrategy> pInteractionStrategy;
typedef std::shared_ptr<class Creature> pCreature;
typedef std::shared_ptr<class Object> pObject;


class InteractionStrategy {
public:
    virtual void interact(Creature& creature, pObject& object) = 0;
    virtual ~InteractionStrategy() = default;
};


#endif // INTERACTION_STRATEGY_H
```

Название файла: interactionuse.h

```
#ifndef INTERACTION_USE_H
#define INTERACTION_USE_H

#include "interactionstrategy.h"

typedef std::shared_ptr<class InteractionUse> pInteractionUse;


class InteractionUse: public InteractionStrategy {
public:
    void interact(Creature& creature, pObject& object);
};


#endif // INTERACTION_USE_H
```

Название файла: interactionuse.cpp

```
#include "field.h"
#include "interactionuse.h"

void InteractionUse::interact(Creature& creature, pObject& object) {
    if (object != nullptr) {
```

```
        object->executeInteraction(creature);

        if (!object->getReusable()) {
            object = nullptr;
        }
    }
}
```

Название файла: levelpassobject.h

```
#ifndef LEVEL_PASS_OBJECT_H
#define LEVEL_PASS_OBJECT_H

#include "object.h"


class LevelPassObject: public Object {
public:
    pObject getCopy() const;
    const std::type_info& getClass() const;
    Texture getTexture() const;
    void executeInteraction(Creature& creature);
    bool getReusable() const;
};


#endif // LEVEL_PASS_OBJECT_H
```

Название файла: levelpassobject.cpp

```
#include "levelpassobject.h"
#include "levelpassobjectfactory.h"
#include "player.h"

pObject LevelPassObject::getCopy() const {
    pLevelPassObjectFactory factory(new LevelPassObjectFactory);
    return pObject(factory->createObject());
}

const std::type_info &LevelPassObject::getClass() const {
    return typeid(LevelPassObject);
}

Texture LevelPassObject::getTexture() const {
    return kTextureObjectLevelPass;
}

void LevelPassObject::executeInteraction(Creature& creature) {
    try {
        Player& player = dynamic_cast<Player&>(creature);
        player.setPassFounded(true);
    } catch (std::bad_cast) {
        return;
    }
}

bool LevelPassObject::getReusable() const {
    return false;
}
```

Название файла: levelpassobjectfactory.h

```
#ifndef LEVEL_PASS_OBJECT_FACTORY_H
#define LEVEL_PASS_OBJECT_FACTORY_H

#include "objectfactory.h"
#include "levelpassobject.h"

typedef std::shared_ptr<class LevelPassObjectFactory>
pLevelPassObjectFactory;


class LevelPassObjectFactory: public ObjectFactory {
public:
    virtual pObject createObject();
};


#endif // LEVEL_PASS_OBJECT_FACTORY_H
```

Название файла: levelpassobjectfactory.cpp

```
#include "levelpassobjectfactory.h"

pObject LevelPassObjectFactory::createObject() {
    return pObject(new LevelPassObject);
}
```

Название файла: mainwindow.h

```
#ifndef MAIN_WINDOW_H
#define MAIN_WINDOW_H

#include <QMainWindow>
#include <QGraphicsView>
#include <QGraphicsScene>
#include <QImage>
#include <QLabel>
#include <QMap>
#include "gamecontroller.h"
#include "texture.h"

QT_BEGIN_NAMESPACE
namespace Ui {
    class MainWindow;
}
QT_END_NAMESPACE

typedef std::shared_ptr<Ui::MainWindow> pMainWindowUi;
typedef std::shared_ptr<QGraphicsView> pQGraphicsView;
typedef std::shared_ptr<QGraphicsScene> pQGraphicsScene;
typedef std::shared_ptr<QPixmap> pQPixmap;
typedef std::shared_ptr<QLabel> pQLabel;


class MainWindow: public QMainWindow {
    Q_OBJECT

private:
```

```
        pMainWindowUi ui;
        pQGraphicsView view;
        pQGraphicsScene scene;
        pQPixmap fieldPixelMap;
        pQLabel healthLabel;
        pQLabel attackLabel;
        pQLabel armorLabel;
        GameController controller;
        QMap<Texture, QImage> textures;
        bool screenPinning = false;
        bool isPressed = false;

    public:
        MainWindow(QWidget* parent = nullptr);
        void updateScene();
        void keyPressEvent(QKeyEvent* event);
        void keyReleaseEvent(QKeyEvent* event);
    };


    #endif // MAIN_WINDOW_H
```

Название файла: mainwindow.cpp

```
    #include <QGraphicsScene>
    #include <QGraphicsView>
    #include <QMap>
    #include <QKeyEvent>
    #include <QMessageBox>
    #include <iostream>
    #include "mainwindow.h"
    #include "ui_mainwindow.h"
    #include "field.h"

    MainWindow::MainWindow(QWidget *parent): QMainWindow(parent), ui(new
Ui::MainWindow) {
        ui->setupUi(this);

        view = pQGraphicsView(new QGraphicsView(this));
        scene = pQGraphicsScene(new QGraphicsScene(this));
        healthLabel = pQLabel(new QLabel(this));
        attackLabel = pQLabel(new QLabel(this));
        armorLabel = pQLabel(new QLabel(this));

        textures[kTextureVoid] = QImage(":/textures/tiles/tile_00.png");
        textures[kTextureWoodFloor1] =
QImage(":/textures/tiles/tile_100.png");
        textures[kTextureWoodWall1] =
QImage(":/textures/tiles/tile_120.png");
        textures[kTextureWoodWall2] =
QImage(":/textures/tiles/tile_147.png");
        textures[kTextureWoodWall3] =
QImage(":/textures/tiles/tile_145.png");
        textures[kTextureWoodWall4] =
QImage(":/textures/tiles/tile_146.png");
        textures[kTextureWoodWall5] =
QImage(":/textures/tiles/tile_118.png");
        textures[kTextureWoodWall6] =
QImage(":/textures/tiles/tile_119.png");
```

```
          textures[kTextureWoodWall7] =
QImage(":/textures/tiles/tile_121.png");
          textures[kTextureWoodWall8] =
QImage(":/textures/tiles/tile_123.png");
          textures[kTextureWoodWall9] =
QImage(":/textures/tiles/tile_124.png");
          textures[kTextureWoodWall10] =
QImage(":/textures/tiles/tile_122.png");
          textures[kTextureWoodWall11] =
QImage(":/textures/tiles/tile_148.png");
          textures[kTextureWoodWall12] =
QImage(":/textures/tiles/tile_151.png");
          textures[kTextureWoodWall13] =
QImage(":/textures/tiles/tile_149.png");
          textures[kTextureEntry] = QImage(":/textures/tiles/tile_132.png");
          textures[kTextureExit] = QImage(":/textures/tiles/tile_133.png");
          textures[kTextureShadow1] =
QImage(":/textures/tiles/shadow_01.png");
          textures[kTextureShadow2] =
QImage(":/textures/tiles/shadow_02.png");
          textures[kTextureShadow3] =
QImage(":/textures/tiles/shadow_03.png");
          textures[kTextureShadow4] =
QImage(":/textures/tiles/shadow_04.png");
          textures[kTextureCell] = QImage(":/textures/tiles/cell.png");
          textures[kTexturePlayerStandT] =
QImage(":/textures/player/player_stand_t.png");
          textures[kTexturePlayerStandB] =
QImage(":/textures/player/player_stand_d.png");
          textures[kTexturePlayerStandR] =
QImage(":/textures/player/player_stand_r.png");
          textures[kTexturePlayerStandL] =
QImage(":/textures/player/player_stand_l.png");
          textures[kTextureObjectMedicines] =
QImage(":/textures/tiles/tile_290.png");
          textures[kTextureObjectArmor] =
QImage(":/textures/tiles/tile_129.png");
          textures[kTextureObjectWeapon] =
QImage(":/textures/tiles/tile_129_2.png");
          textures[kTextureObjectLevelPass] =
QImage(":/textures/tiles/tile_key.png");

          view-
>setHorizontalScrollBarPolicy(Qt::ScrollBarPolicy::ScrollBarAlwaysOff);
          view-
>setVerticalScrollBarPolicy(Qt::ScrollBarPolicy::ScrollBarAlwaysOff);
          view->setStyleSheet("background-color: black;");
          view->setScene(scene.get());

          healthLabel->move(25, 25);
          attackLabel->move(25, 45);
          armorLabel->move(25, 65);
          healthLabel->setStyleSheet("QLabel { font-weight: bold; font-size:
16px; color: white; }");
          attackLabel->setStyleSheet("QLabel { font-weight: bold; font-size:
16px; color: white; }");
          armorLabel->setStyleSheet("QLabel { font-weight: bold; font-size:
16px; color: white; }");

          if (!screenPinning) {
              view->setDragMode(QGraphicsView::ScrollHandDrag);
```

```
        }

        updateScene();
        setCentralWidget(view.get());
    }

    void MainWindow::updateScene() {
        const Field& field = controller.getField();
        pConstPlayer player = controller.getPlayer();
        Size2D fieldSize = field.getSize();

        if (fieldPixelMap == nullptr ||
            static_cast<size_t>(fieldPixelMap->width()) != fieldSize.x * 64
||
            static_cast<size_t>(fieldPixelMap->height()) != fieldSize.y *
64)
        {
            fieldPixelMap = pQPixmap(new QPixmap(fieldSize.x * 64,
fieldSize.y * 64));
        }

        QPainter painter(fieldPixelMap.get());

        for (const Cell& cell : field) {
            Position2D coords = cell.getPosition();

            painter.drawImage(coords.x * 64, coords.y * 64,
textures[cell.getTexture()]);

            if (cell.getType() == kCellTypeEntry) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureEntry]);
            } else if (cell.getType() == kCellTypeExit) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureExit]);
            }

            if (cell.getPassible()) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureCell]);
            }

            if (cell.getObject() != nullptr) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[cell.getObject()->getTexture()]);
            }

            if (coords.y == 0) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureShadow2]);
            }

            if (coords.x == 0) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureShadow1]);
            }

            if (coords.y == fieldSize.y - 1) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureShadow4]);
            }
```

```
            if (coords.x == fieldSize.x - 1) {
                painter.drawImage(coords.x * 64, coords.y * 64,
textures[kTextureShadow3]);
            }
        }

        painter.drawImage(player->getPosition().x * 64, player-
>getPosition().y * 64, textures[player->getTexture()]);

        healthLabel->setText("Health: " + QString::number(player-
>getHealth()));
        attackLabel->setText("Attack: " + QString::number(player-
>getAttackDamage()));
        armorLabel->setText("Armor: " + QString::number(player-
>getProtection()));

        scene->clear();
        scene->addPixmap(*fieldPixelMap);

        if (screenPinning) {
            view->centerOn(player->getPosition().x * 64 + 32, player-
>getPosition().y * 64 + 32);
        }
    }

    void MainWindow::keyPressEvent(QKeyEvent* event) {
        if (!isPressed) {
            isPressed = true;

            if (event->key() == Qt::Key_W) {
                controller.movePlayer(kDirectionTop);
            } else if (event->key() == Qt::Key_S) {
                controller.movePlayer(kDirectionBottom);
            } else if (event->key() == Qt::Key_A) {
                controller.movePlayer(kDirectionLeft);
            } else if (event->key() == Qt::Key_D) {
                controller.movePlayer(kDirectionRight);
            } else if (event->key() == Qt::Key_E) {
                controller.executePlayerInteraction();
            }

            updateScene();

            if (controller.isGameOver()) {
                QMessageBox::information(this, "Game over", "Great job,
level passed!");
                QApplication::quit();
            }
        }
    }

    void MainWindow::keyReleaseEvent(QKeyEvent* event) {
        if (!event->isAutoRepeat()) {
            isPressed = false;
        }
    }
```

Название файла: medicines.h

```
#ifndef MEDICINES_H
#define MEDICINES_H

#include <memory>
#include "object.h"
#include "creature.h"

typedef std::shared_ptr<class Medicines> pMedicines;


class Medicines: public Object {
private:
    int healthRecovery_;

public:
    explicit Medicines(int healthRecovery);
    pObject getCopy() const;
    void executeInteraction(Creature& creature);
    const std::type_info& getClass() const;
    Texture getTexture() const;
    bool getReusable() const;
};


#endif // MEDICINES_H
```

Название файла: medicines.cpp

```
#include "medicines.h"
#include "medicinesfactory.h"

Medicines::Medicines(int healthRecovery):
healthRecovery_(healthRecovery) {}

pObject Medicines::getCopy() const {
    pMedicinesFactory factory(new MedicinesFactory);
    return pObject(factory->createMedicines(healthRecovery_));
}

void Medicines::executeInteraction(Creature& creature) {
    creature.setHealth(creature.getHealth() + healthRecovery_);
}

const std::type_info& Medicines::getClass() const {
    return typeid(Medicines);
}

Texture Medicines::getTexture() const {
    return kTextureObjectMedicines;
}

bool Medicines::getReusable() const {
    return false;
}
```

Название файла: medicinesfactory.h

```
#ifndef MEDICINES_FACTORY_H
#define MEDICINES_FACTORY_H

#include "objectfactory.h"
#include "medicines.h"

typedef std::shared_ptr<class MedicinesFactory> pMedicinesFactory;


class MedicinesFactory: public ObjectFactory {
public:
    virtual pObject createObject();
    virtual pObject createMedicines(int healthRecovery);
};


#endif // MEDICINES_FACTORY_H
```

Название файла: medicinesfactory.cpp

```
#include "medicinesfactory.h"

pObject MedicinesFactory::createObject() {
    return pObject(new Medicines(20));
}

pObject MedicinesFactory::createMedicines(int healthRecovery) {
    return pObject(new Medicines(healthRecovery));
}
```

Название файла: object.h

```
#ifndef OBJECT_H
#define OBJECT_H

#include <typeinfo>
#include <memory>
#include "texture.h"
#include "creature.h"

typedef std::shared_ptr<class Object> pObject;
typedef std::shared_ptr<const class Object> pConstObject;
typedef std::shared_ptr<class Creature> pCreature;


class Object {
public:
    virtual pObject getCopy() const = 0;
    virtual const std::type_info& getClass() const = 0;
    virtual Texture getTexture() const = 0;
    virtual void executeInteraction(Creature& creature) = 0;
    virtual bool getReusable() const = 0;
    virtual ~Object() = default;
};


#endif // OBJECT_H
```

Название файла: objectfactory.h

```
#ifndef OBJECT_FACTORY_H
#define OBJECT_FACTORY_H

#include "object.h"

typedef std::shared_ptr<class ObjectFactory> pObjectFactory;


class ObjectFactory {
public:
    virtual pObject createObject() = 0;
    virtual ~ObjectFactory() = default;
};


#endif // OBJECT_FACTORY_H
```

Название файла: player.h

```
#ifndef PLAYER_H
#define PLAYER_H

#include <memory>
#include "creature.h"

typedef std::shared_ptr<class Player> pPlayer;


class Player: public Creature {
private:
    bool passFounded_ = false;
    pInteractionStrategy objectInteractionStrategy_;

public:
    Player(Position2D position);
    void interact(pObject& object);
    Texture getTexture() const;
    void operator<=(pObject& object);
    bool getPassFounded() const;
    void setPassFounded(bool value);
};


#endif // PLAYER_H
```

Название файла: player.cpp

```
#include "player.h"
#include "interactionuse.h"
#include "interactionnone.h"
#include <iostream>

Player::Player(Position2D position) {
    objectInteractionStrategy_ = pInteractionStrategy(new
InteractionUse);
    setPosition(position);
}
```

```
void Player::interact(pObject& object) {
    //static int count = 0;

    /* if (count == 4) {
        objectInteractionStrategy_ = pInteractionStrategy(new
InteractionNone);
    }*/

    if (objectInteractionStrategy_ != nullptr) {
        objectInteractionStrategy_->interact(*this, object);
    }

    /* if (object != nullptr) {
        count++;
    }*/
}

Texture Player::getTexture() const {
    switch (getRotation()) {
    case kDirectionTop:
        return kTexturePlayerStandT;
    case kDirectionBottom:
        return kTexturePlayerStandB;
    case kDirectionLeft:
        return kTexturePlayerStandL;
    case kDirectionRight:
        return kTexturePlayerStandR;
    }
    return kTextureVoid;
}

void Player::operator<=(pObject& object) {
    interact(object);
}

bool Player::getPassFounded() const {
    return passFounded_;
}

void Player::setPassFounded(bool value) {
    passFounded_ = value;
}
```

Название файла: point2d.h

```
#ifndef POINT_2D_H
#define POINT_2D_H

#include <cstddef>
#include "direction.h"

typedef struct Point2D Size2D;
typedef struct Point2D Position2D;


struct Point2D {
public:
    size_t x = 0;
    size_t y = 0;
```

```
    Point2D() = default;
    Point2D(size_t x, size_t y);
    bool operator==(const Point2D& other) const;
    bool operator!=(const Point2D& other) const;
    void shift(Direction direction);
};


#endif // POINT_2D_H
```

Название файла: point2d.cpp

```
#include "point2d.h"

Point2D::Point2D(size_t x, size_t y): x(x), y(y) {}

bool Point2D::operator==(const Point2D& other) const {
    return x == other.x && y ==other.y;
}

bool Point2D::operator!=(const Point2D& other) const {
    return !operator==(other);
}

void Point2D::shift(Direction direction) {
    switch (direction) {
    case kDirectionTop:
        y--;
        return;
    case kDirectionBottom:
        y++;
        return;
    case kDirectionLeft:
        x--;
        return;
    case kDirectionRight:
        x++;
        return;
    }
}
```

Название файла: texture.h

```
#ifndef TEXTURE_H
#define TEXTURE_H

enum Texture {
    kTextureVoid,
    kTextureWoodFloor1,
    kTextureWoodWall1,
    kTextureWoodWall2,
    kTextureWoodWall3,
    kTextureWoodWall4,
    kTextureWoodWall5,
    kTextureWoodWall6,
    kTextureWoodWall7,
    kTextureWoodWall8,
    kTextureWoodWall9,
```

```
    kTextureWoodWall10,
    kTextureWoodWall11,
    kTextureWoodWall12,
    kTextureWoodWall13,
    kTextureEntry,
    kTextureExit,
    kTextureShadow1,
    kTextureShadow2,
    kTextureShadow3,
    kTextureShadow4,
    kTextureCell,
    kTexturePlayerStandT,
    kTexturePlayerStandB,
    kTexturePlayerStandR,
    kTexturePlayerStandL,
    kTextureObjectMedicines,
    kTextureObjectArmor,
    kTextureObjectWeapon,
    kTextureObjectLevelPass
};

#endif // TEXTURE_H
```

Название файла: weapon.h

```
#ifndef WEAPON_H
#define WEAPON_H

#include "memory"
#include "object.h"

typedef std::shared_ptr<class Armor> pArmor;


class Weapon: public Object {
private:
    int damage_;

public:
    explicit Weapon(int damage);
    pObject getCopy() const;
    void executeInteraction(Creature& creature);
    const std::type_info& getClass() const;
    Texture getTexture() const;
    bool getReusable() const;
};


#endif // WEAPON_H
```

Название файла: weapon.cpp

```
#include "weapon.h"
#include "weaponfactory.h"

Weapon::Weapon(int damage): damage_(damage) {}

pObject Weapon::getCopy() const {
    pWeaponFactory factory(new WeaponFactory);
```

```
    return pObject(factory->createWeapon(damage_));
}

void Weapon::executeInteraction(Creature& creature) {
    if (creature.getAttackDamage() < damage_) {
        creature.setAttackDamage(damage_);
    }
}

const std::type_info &Weapon::getClass() const {
    return typeid(Weapon);
}

Texture Weapon::getTexture() const {
    return kTextureObjectWeapon;
}

bool Weapon::getReusable() const {
    return false;
}
```

Название файла: weaponfactory.h

```
#ifndef WEAPON_FACTORY_H
#define WEAPON_FACTORY_H

#include "objectfactory.h"
#include "weapon.h"

typedef std::shared_ptr<class WeaponFactory> pWeaponFactory;


class WeaponFactory: public ObjectFactory {
public:
    virtual pObject createObject();
    virtual pObject createWeapon(int damage);
};


#endif // WEAPON_FACTORY_H
```

Название файла: weaponfactory.cpp

```
#include "weaponfactory.h"

pObject WeaponFactory::createObject() {
    return pObject(new Weapon(5));
}

pObject WeaponFactory::createWeapon(int damage) {
    return pObject(new Weapon(damage));
}
```