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

ОТЧЕТ

по лабораторной работе №1-7

по дисциплине «Объектно-ориентированное программирование»

Тема: Пошаговая стратегия

Студент гр. 8304 _____ Масалыкин Д.Р.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Разработать пошаговую игру средствами языка C++ в соответствие с принципами SOLID и поставленными задачами. Изучить паттерны программирование в ходе написания проекта.

Лабораторная работа №1

Разработать и реализовать набор классов:

- Класс игрового поля
- Набор классов юнитов

Игровое поле является контейнером для объектов представляющим прямоугольную сетку. Основные требования к классу игрового поля:

- Создание поля произвольного размера
- Контроль максимального количества объектов на поле
- Возможность добавления и удаления объектов на поле
- Возможность копирования поля (включая объекты на нем)
- Для хранения запрещается использовать контейнеры из stl

Юнит является объектов, размещаемым на поля боя. Один юнит представляет собой отряд. Основные требования к классам юнитов:

- Все юниты должны иметь как минимум один общий интерфейс
- Реализованы 3 типа юнитов (например, пехота, лучники, конница)
- Реализованы 2 вида юнитов для каждого типа(например, для пехоты могут быть созданы мечники и копейщики)
- Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.
- Юнит имеет возможность перемещаться по карте

Баллы за лаб. работу (* отмечает необязательные пункты)

Выполнены основные требования класса поле	3 балла	
Выполнены основные требования классов юнитов	4 балла	
Имеется 3+ демонстрационных примера	1 балл	
Все методы класса сохраняют инвариант этого класса	2 балл	
*Созданы конструкторы копирования и перемещения	2 балла	переход
*Все методы принимают параметры оптимальным образом (то есть, отсутствует лишнее копирование объектов)	1 балл	
*Для атрибутов юнитов созданы свои классы. Создавать их требуется, если это не противоречит логике.	2 балла	переход
*Для создания юнитов используются паттерны “Фабричный метод” / “Абстрактная фабрика”	3 баллов	переход
*Создан итератор для поля	2 балла	
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №2 (Интерфейсы классов; взаимодействие классов; перегрузка операций)

Разработать и реализовать набор классов:

- Класс базы
- Набор классов ландшафта карты
- Набор классов нейтральных объектов поля

Класс базы должен отвечать за создание юнитов, а также учитывать юнитов, относящихся к текущей базе. Основные требования к классу база:

- База должна размещаться на поле
- Методы для создания юнитов
- Учет юнитов, и реакция на их уничтожение и создание
- База должна обладать характеристиками такими, как здоровье, максимальное количество юнитов, которые могут быть одновременно созданы на базе, и.т.д.

Набор классов ландшафта определяют вид поля. Основные требования к классам ландшафта:

Должно быть создано минимум 3 типа ландшафта

- Все классы ландшафта должны иметь как минимум один интерфейс
- Ландшафт должен влиять на юнитов (например, возможно пройти по клетке с определенным ландшафтом или запрет для атаки определенного типа юнитов)
- На каждой клетке поля должен быть определенный тип ландшафта

Набор классов нейтральных объектов представляют объекты, располагаемые на поле и с которыми могут взаимодействие юнитов. Основные требования к классам нейтральных объектов поля:

- Создано не менее 4 типов нейтральных объектов
- Взаимодействие юнитов с нейтральными объектами, должно быть реализовано в виде перегрузки операций
- Классы нейтральных объектов должны иметь как минимум один общий интерфейс

Выполнены основные требования к классу база	2 балла	clion
Выполнены основные требования к набору классов ландшафта	2 балла	clion
Выполнены основные требования к набору классов нейтр. объектов	2 балла	clion
Добавлено взаимодействие юнитов	1 балла	clion
Имеется 3+ демонстрационных примера	1 балл	нет
Взаимодействие через перегрузку операторов	2 балла	переход
*Для хранения информации о юнитах в классе базы используется паттерн “Компоновщик”/ Использование “Легковеса” для хранения общих характеристик юнитов	2 балла	переход
*Для наблюдения над юнитами в классе база используется паттерн “Наблюдатель”	2 балла	переход

*Для взаимодействия ландшафта с юнитам используется паттерн “Прокси”	3 балла	переход
*Для взаимодействия одного типа нейтрального объекта с разными типами юнитов используется паттерн “Стратегия”	3 балла	Clion(не скопировалось)
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №3 (Логическое разделение классов)

Разработать и реализовать набора классов для взаимодействия пользователя с юнитами и базой. Основные требования:

- Должен быть реализован функционал управления юнитами
- Должен быть реализован функционал управления базой

Выполнены все основные требования к взаимодействию	6 баллов	clion
Добавлен функционал просмотра состояния базы	3 балла	clion
Имеется 3+ демонстрационных примера	1 балл	нет
*Реализован паттерн “Фасад” через который пользователь управляет программой	1 балл	переход
*Объекты между собой взаимодействуют через паттерн “Посредника”	3 балла	переход
*Для передачи команд используется паттерн “Команда”	3 балла	переход
*Для приема команд от пользователя используется паттерн “Цепочка обязанностей”	3 балла	переход
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №4 (Полиморфизм)

Реализовать набор классов, для ведения логирования действий и состояний программы. Основные требования:

- Логирование действий пользователя
- Логирование действий юнитов и базы

Выполнены основные требования к логированию	3 балла	clion
Реализована возможность записи логов в файл	3 балла	clion
Реализована возможность записи логов в терминал	3 балла	clion
Взаимодействие с файлами должны быть по идиоме RAII	1 балл	переход
*Для логирования состояний перегружен оператор вывода в поток	2 балла	переход
*Переключение между разным логированием (логирование в файл, в терминал, без логирования) реализуется при помощи паттерна “Прокси”	4 балла	См выше
*Реализован разный формат записи при помощи паттерна “Адаптер”	4 балла	переход
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №5 (Сериализация состояния программы)

Реализация сохранения и загрузки состояния программы. Основные требования:

- Возможность записать состояние программы в файл
- Возможность считать состояние программы из файла

Выполнены основные требования к сохранению и загрузке	4 баллов	См ниже
Загрузка и сохранение должно выполняться в любой момент программы	5 баллов	переход
Взаимодействие с файлами должны быть по идиоме RAII	1 балл	переход
*Сохранение и загрузка реализованы при помощи паттерна “Снимок”	5 баллов	переход
*Реализован контроль корректности файла с сохраненными данными	5 баллов	
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №6 (Шаблонные классы)

Разработка и реализация набора классов правил игры. Основные требования:

- Правила игры должны определять начальное состояние игры
- Правила игры должны определять условия выигрыша игроков
- Правила игры должны определять очередность ходов игрока
- Должна быть возможность начать новую игру

Выполнены основные требования	5 баллов	См GameSettings
Реализован шаблонный класс игры, в качестве параметра шаблона передаются конкретные правила	3 балла	выше
Должно быть реализовано минимум 2 правил игры	2 балла	выше
*Класс игры в шаблоне поддерживает кол-во игроков. И для определенного кол-ва должен быть специализирован отдельно	3 балла	выше
*Передача хода между игроками реализована при помощи паттерна “Состояние”	4 балла	переход
*Класс игры один единственный и создается паттерном “Синглтон”	3 балла	переход
Кол-во баллов за основные требования	**10 баллов**	
Максимальное кол-во баллов за лаб. работу	**20 баллов**	

Лабораторная работа №7 (Написание исключений)

Разработать и реализовать набор исключений. Основные требования к исключениям:

- Исключения покрывают как минимум все тривиальные случаи возникновения ошибки
- Все реализованные исключения обрабатываются в программе
- Исключения должны хранить подробную информацию об ошибке, а не только строку с сообщением об ошибке

Выполнены основные требования	5 баллов	StackExc(тривиальные)
*Проведено юнит-тестирование программы	5 баллов	нет
Кол-во баллов за основные требования	**5 баллов**	
Максимальное кол-во баллов за лаб. работу	**10 баллов**	

Дополнительные баллы

Далее приведены критерии, за которые можно получить дополнительные баллы. Данные пункты не являются обязательными к выполнению

Разработан GUI (GUI не должен содержать никакой бизнес-логики)	13 баллов
Все пользовательские классы логически объединены в пространства имен	3 балла
Логичное использование паттернов, помимо пунктов в лаб. работах	1 балл за паттерн (не более 5 баллов суммарно)
Построена UML-диаграмма проекта	3 балла
Соблюдены все принципы SOLID	10 баллов
Логичное использование шаблонов, помимо пунктов в лаб. работах	5 баллов
Программа запускается через исполняемый файл	2 балла
В программе используются умные указатели	5 баллов
Максимально количество доп. баллов	**50 баллов**

Приложение А.

```
#ifndef BATTLEFORHONOUR_ARCHER_H
#define BATTLEFORHONOUR_ARCHER_H


#include "../../../Weapon/Weapon.h"
#include "../../Unit.h"
#include "../../../Weapon/WeaponFlyweight.h"
#include "../../../Armor/ArmorFlyweight.h"

class Archer: public Unit{

public:
   Archer(Armor &armor, int health):
                                          Unit(UnitType::ARCHER,       armor,
*WeaponFlyweight::getFlyWeight<Bow>(), health){}
};


#endif //BATTLEFORHONOUR_ARCHER_H

#ifndef BATTLEFORHONOUR_BIGGAME_H
#define BATTLEFORHONOUR_BIGGAME_H


#include "GameRule.h"
#include "PlayerState.h"

class BigGame: public GameRule {

private:
   PlayerState* nowState;
public:

   BigGame():
      GameRule( 15, 15),
      nowState(new FirstPlayer){}
```

```
    bool isOver(GameState &gameState) override {
        int liveCount = gameState.getBases().size();
        for (auto b: gameState.getBases()){
            if (b && b->getHealth() <= 0){
                liveCount--;
            }
        }
        return liveCount <= 1;
    }

    int nextUser(GameState &gameState) override {
            int currUserPos = (gameState.getNowPlayerIndex() + nowState->getNextPlayerRecr()) % gameState.getBases().size();
        auto nextState = nowState->getNextPlayerState();
        delete nowState;
        nowState = nextState;
        if (nowState == nullptr)
            nowState = new FirstPlayer;
        return currUserPos;
    }

};


#endif //BATTLEFORHONOUR_BIGGAME_H

#ifndef BATTLEFORHONOUR_CMDLOGGER_H
#define BATTLEFORHONOUR_CMDLOGGER_H


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

class CmdLogger: public Logger {

public:
```

```
    void log(std::string &stream) override{
        std::cout << stream;
    }

};


#endif //BATTLEFORHONOUR_CMDLOGGER_H

#ifndef BATTLEFORHONOUR_ARMOR_H
#define BATTLEFORHONOUR_ARMOR_H

#include <ostream>
#include "ArmorType.h"


class Armor {

protected:

    ArmorType type;
    int absorbation{};

public:

    Armor(){}

    [[nodiscard]] int controlAbsorb() const {
        return this->absorbation;
    }
    ArmorType getArmorType(){
        return type;
    }

     friend std::ostream &operator<<(std::ostream &stream, const Armor
&armor){
```

```
        stream << "Armor = " << "Damage Absorb: " << armor.absorbation
<< " ;";
        return stream;
    }

    bool operator==(Armor &other){
                return this->type == other.type && this->absorbation ==
other.absorbation;
    }

    Armor& operator=(const Armor& tmp){
        if (this == &tmp)
            return *this;
        this->type = tmp.type;
        this->absorbation = tmp.absorbation;
        return *this;
    }
};


class LeatherArmor: public Armor {

public:
    LeatherArmor(){
        type = ArmorType::LIGHT;
        absorbation = 2;
    }
};

class PlateMail: public Armor{

public:
    PlateMail(){
        type = ArmorType::MEDIUM;
        absorbation = 5;
    }
```

```
};

class Robe: public Armor{

public:
   Robe(){
      type = ArmorType::MAGIC;
      absorbation = 1;
   }

};

class VladimirOffering: public Armor{

public:
   VladimirOffering(){
      type = ArmorType::HEAVY;
      absorbation = 10;
   }

};
#endif //BATTLEFORHONOUR_ARMOR_H

#ifndef BATTLEFORHONOUR_ARMORFLYWEIGHT_H
#define BATTLEFORHONOUR_ARMORFLYWEIGHT_H


#include <vector>
#include "Armor.h"

class ArmorFlyweight {

private:

   static ArmorFlyweight *self;
   std::vector<Armor*> armorArr;
```

```
public:
    template <typename Type>
    static Type* getFlyweight(){

        if (!self)
            self = new ArmorFlyweight();

        Type typeArmor;
        for (auto *armor: self->armorArr){
            if (typeArmor == *armor){
                return static_cast<Type*>(armor);
            }
        }

        Type *armorPtr = new Type();
        self->armorArr.push_back(armorPtr);
        return armorPtr;
    }
};

ArmorFlyweight *ArmorFlyweight::self = nullptr;


#endif //BATTLEFORHONOUR_ARMORFLYWEIGHT_H

#ifndef BATTLEFORHONOUR_ARMORTYPE_H
#define BATTLEFORHONOUR_ARMORTYPE_H

enum class ArmorType{
    LIGHT,
    MEDIUM,
    HEAVY,
    MAGIC
};

#endif //BATTLEFORHONOUR_ARMORTYPE_H
```

```
#ifndef BATTLEFORHONOUR_ATTACKCOMMAND_H
#define BATTLEFORHONOUR_ATTACKCOMMAND_H


#include "Command.h"
#include "AttackUnitCommand.h"

class AttackCommandHandler: public CommandHandler {

public:

   bool isHandle(std::vector<std::string> &terminal) override{
      return terminal.size() > 1 && terminal[0] == "attack";
   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

      if (isHandle(terminal)) {
         terminal.erase(terminal.begin());
         auto handleAttack = new AttackUnitCommandHandler();
         return handleAttack->handle(terminal);
      }

      if (next)
         return next->handle(terminal);

      return std::make_unique<Command>();
   }
};


#endif //BATTLEFORHONOUR_ATTACKCOMMAND_H

#ifndef BATTLEFORHONOUR_COMMAND_H
#define BATTLEFORHONOUR_COMMAND_H
```

```
#include <string>
#include <memory>
#include <sstream>
#include "../../Logs/Log.h"
#include "../../Game/GameState.h"
#include "CommandSnapshot.h"

class Command {

public:

   virtual void execute(GameState &gameInfo){}
   [[nodiscard]] virtual CommandSnapshot *getSnapshot() const {
      return new CommandSnapshot(">>wrong command\n");
   }
   virtual ~Command(){}

};


class CommandHandler{

protected:
   CommandHandler *next{};

public:

        virtual  std::unique_ptr<Command>  handle(std::vector<std::string>
&terminal)=0;
   virtual bool isHandle(std::vector<std::string> &terminal)=0;
   void setNext(CommandHandler *commandHandler){
      next = commandHandler;
   }
   virtual ~CommandHandler()= default;

};
```

```
int convertStr(const std::string& s) {

   try {
      return (int)std::stoull(s);
   } catch (std::invalid_argument) {
      Log::log << "Wrong format. No numbers." << Log::logend;
      return 0;
   } catch (std::out_of_range) {
      Log::log << "Wrong format. Range overflow." << Log::logend;
      return 0;
   } catch (...) {
      Log::log << "Wrong format. Anything goes wrong..." << Log::logend;
      return 0;
   }

}


#endif //BATTLEFORHONOUR_COMMAND_H

#ifndef BATTLEFORHONOUR_ATTACKUNITCOMMAND_H
#define BATTLEFORHONOUR_ATTACKUNITCOMMAND_H


#include "Command.h"

class AttackUnitCommand: public Command{

private:

   Point from;
   Point to;

public:

   AttackUnitCommand(Point from, Point to): from(from), to(to){}
```

```
    void execute(GameState &gameState) override{

        auto object1 = gameState.getField().getCell(from)->getObject();
        auto object2 = gameState.getField().getCell(to)->getObject();
         if (object1 && object1->getType() == ObjectType::UNIT && object1
&& object1->getType() == ObjectType::UNIT) {
            auto unit1 = dynamic_cast<Unit *>(object1);
            auto unit2 = dynamic_cast<Unit *>(object2);
            unit1->attack(*unit2);
            Log::log << "Gotten command attack" << Log::logend;
        } else
            Log::log << "Impossible attack" << Log::logend;

    }

    [[nodiscard]] CommandSnapshot * getSnapshot() const override{
        std::stringstream stream;
        stream << "attack unit " << from.x << " " << from.y << " " << to.x <<
" " << to.y << std::endl;
        return new CommandSnapshot(stream.str());
    }

};

class AttackUnitCommandHandler: public CommandHandler {

public:

    bool isHandle(std::vector<std::string> &terminal) override{
        return terminal.size() == 5 && terminal[0] == "unit";
    }

     std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
            int x1 = convertStr(terminal[1]);
```

```
            int y1 = convertStr(terminal[2]);
            int x2 = convertStr(terminal[3]);
            int y2 = convertStr(terminal[4]);
            Point from(x1,y1);
            Point to(x2,y2);
                                        return    std::unique_ptr<Command>(new
AttackUnitCommand(from, to));
        }

        if (next)
            return next->handle(terminal);
        return std::make_unique<Command>();

    }

};

#endif //BATTLEFORHONOUR_ATTACKUNITCOMMAND_H

#include "Base.h"

bool Base::addUnit(Unit *unit, Point position) {
    if (units.size() < limit){

        units.push_back(unit);
        for (auto bo: baseObservers){
            bo->onBaseNewUnit(unit, position);
        }
        Log::log << "[#Base] Unit added";
        return true;

    } else{

        Log::log << "[#Base] Can't add unit" << Log::logend;
        return false;

    }
```

```
}

void Base::onUnitAttack(Unit *unit, Unit *other) {
   Log::log << "[#Base]" << *unit << " attack\n";
}

void Base::onUnitMove(Unit *unit, Point p) {

   Log::log << "[#Base] " << *unit << " moving" << Log::logend;

}

void Base::onUnitDestroy(Unit *unit) {

   auto pos = std::find(units.begin(), units.end(), unit);
   if (pos != units.end()) {
      units.erase(pos);
      Log::log << "[#Base] " << *unit << " killed" << Log::logend;
   } else{
        Log::log << "Called observer of base for unit don't belong to it" <<
Log::logend;
   }

}

void Base::onUnitDamaged(Unit *unit) {

   Log::log << "[#Base] " << *unit << " damaged" << Log::logend;

}

void Base::onUnitHeal(Unit *unit) {

   Log::log << "[#Base] " << *unit << " healed" << Log::logend;

}
```

```
void Base::print(std::ostream &stream) const {

    stream << "B";

}

void Base::addObserver(BaseObserver *baseObserver) {

    baseObservers.push_back(baseObserver);
    Log::log << "[#Base] added observer" << Log::logend;

}
#ifndef BATTLEFORHONOUR_BASE_H
#define BATTLEFORHONOUR_BASE_H

#include "../Armor/Armor.h"
#include "GameObject.h"
#include "Unit.h"
#include "../Observers/Observers.h"
#include <vector>
#include <iostream>
#include <algorithm>


class Base: public GameObject, public UnitObserver {

private:

    std::vector<BaseObserver*> baseObservers;

protected:

    void print(std::ostream &stream) const override;

public:

    Base(int health, Armor &armor):
```

```
        GameObject(ObjectType::BASE),
        health(health),
        armor(armor) {}

    bool addUnit(Unit *unit, Point position);
    void addObserver(BaseObserver *baseObserver);

    [[nodiscard]] int getHealth() const{
        return health;
    }
    Armor& getArmor(){
        return armor;
    }
    [[nodiscard]] int getMaxObjectsCount() const{
        return limit;
    }

    template <typename Type>
    Type *createUnit(Point position);
    void onUnitAttack(Unit *unit, Unit *other) override;
    void onUnitMove(Unit *unit, Point p) override;
    void onUnitDestroy(Unit *unit) override;
    void onUnitDamaged(Unit *unit) override;
    void onUnitHeal(Unit *unit) override;

private:
    std::vector<Unit*> units;
    int health;
    const int limit = 10;
    Armor &armor;
};

template<typename Type>
Type *Base::createUnit(Point position) {
    if (units.size() < limit) {
        Type *unit = new Type();
        units.push_back(unit);
```

```
        unit->addObserver(this);

        for (auto elem:baseObservers)
            elem->onBaseNewUnit(unit, position);

        Log::log << "[#Base] Unit created\n";

        return unit;
    } else{
        Log::log << "[#Base] Cannot create unit. Limit is exceeded.\n";
        return nullptr;
    }
}

#endif //BATTLEFORHONOUR_BASE_H

#ifndef BATTLEFORHONOUR_COMMANDINTERPRETER_H
#define BATTLEFORHONOUR_COMMANDINTERPRETER_H


#include "Commands/Command.h"
#include "Commands/AttackCommand.h"
#include "Commands/CreateCommand.h"
#include "Commands/MoveCommand.h"
#include "Commands/ShowCommand.h"
#include "Commands/ExitCommand.h"
#include "Commands/SaveCommand.h"
#include "Commands/LoadCommand.h"
#include "Commands/NewCommand.h"
#include "Commands/SkipCommand.h"

class CommandInterpreter {

private:

    AttackCommandHandler *attackHandler;
    CreateCommandHandler *createHandler;
```

```
    MoveCommandHandler *moveHandler;
    ShowCommandHandler *showHandler;
    ExitCommandHandler *exitHandler;
    SaveCommandHandler *saveHandler;
    LoadCommandHandler *loadHandler;
    NewCommandHandler *newHandler;
    SkipCommandHandler *skipHandler;

public:

    CommandInterpreter(){
        newHandler = new NewCommandHandler();
        attackHandler = new AttackCommandHandler();
        createHandler = new CreateCommandHandler();
        moveHandler = new MoveCommandHandler();
        showHandler = new ShowCommandHandler();
        exitHandler = new ExitCommandHandler();
        saveHandler = new SaveCommandHandler();
        loadHandler = new LoadCommandHandler();
        skipHandler = new SkipCommandHandler();

        attackHandler->setNext(createHandler);
        createHandler->setNext(moveHandler);
        moveHandler->setNext(showHandler);
        showHandler->setNext(exitHandler);
        exitHandler->setNext(saveHandler);
        saveHandler->setNext(loadHandler);
        loadHandler->setNext(newHandler);
        newHandler->setNext(skipHandler);
    }

    std::unique_ptr<Command> handle(const std::string& commandString)
{

        std::vector <std::string> commandSplitted;
        std::stringstream stream(commandString);
        std::string commandWord;
```

```
        while (stream >> commandWord)
            commandSplitted.push_back(commandWord);

        return attackHandler->handle(commandSplitted);

    }

    ~CommandInterpreter(){

        delete attackHandler;
        delete createHandler;
        delete moveHandler;
        delete showHandler;
        delete exitHandler;
        delete saveHandler;
        delete skipHandler;

    }

};

#endif //BATTLEFORHONOUR_COMMANDINTERPRETER_H

#ifndef BATTLEFORHONOUR_COMMANDSNAPSHOT_H
#define BATTLEFORHONOUR_COMMANDSNAPSHOT_H


#include <string>
#include <fstream>
#include <utility>

class CommandSnapshot{

private:

    std::string commandLine;
```

```
public:

        explicit CommandSnapshot(std::string commandLine): commandLine(std::move(commandLine)){}

    void saveToFile(std::ofstream &fs) const{

        fs << commandLine;

    }

    unsigned long int getHash(std::hash<std::string> &toHash){
        return toHash(commandLine);
    }

};


#endif //BATTLEFORHONOUR_COMMANDSNAPSHOT_H

#ifndef BATTLEFORHONOUR_CREATEBASECOMMAND_H
#define BATTLEFORHONOUR_CREATEBASECOMMAND_H


#include "Command.h"
#include "../../Armor/ArmorFlyweight.h"
#include "../../Armor/Armor.h"

class CreateBaseCommand: public Command {

private:

    Point basePos;

public:

    explicit CreateBaseCommand(Point pos): basePos(pos){}
```

```
    void execute(GameState &gameState) override{

                                        auto *base = new Base(100,
*ArmorFlyweight::getFlyweight<LeatherArmor>());
        if (gameState.setNowPlayerBase(base)) {
            gameState.getField().addBase(base, basePos);
            Log::log << "Command to create base" << Log::logend;
        } else
            Log::log << "This player already has base" << Log::logend;
    }

    [[nodiscard]] CommandSnapshot * getSnapshot() const override{

        std::stringstream stream;
            stream << "create base " << basePos.x << " " << basePos.y <<
std::endl;
        return new CommandSnapshot(stream.str());

    }

};

class CreateBaseCommandHandler: public CommandHandler{

public:

    bool isHandle(std::vector<std::string> &terminal) override{

        return terminal.size() == 3 && terminal[0] == "base";

    }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override {

        if (isHandle(terminal)){
            int x = convertStr(terminal[1]);
```

```
            int y = convertStr(terminal[2]);
            Point basePosition(x, y);
                                        return    std::unique_ptr<Command>(new
CreateBaseCommand(basePosition));
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }
};



#endif //BATTLEFORHONOUR_CREATEBASECOMMAND_H

#ifndef BATTLEFORHONOUR_CREATECOMMAND_H
#define BATTLEFORHONOUR_CREATECOMMAND_H


#include "Command.h"
#include "CreateUnitCommand.h"
#include "CreateBaseCommand.h"

class CreateCommandHandler: public CommandHandler {

public:

    bool isHandle(std::vector<std::string> &terminal) override{
        return terminal.size() > 1 && terminal[0] == "create";
    }

     std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
```

```
            terminal.erase(terminal.begin());

            auto handleUnit = new CreateUnitCommandHandler();
            auto handleBase = new CreateBaseCommandHandler();
            handleUnit->setNext(handleBase);

            return handleUnit->handle(terminal);
        }
        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }

};

#endif //BATTLEFORHONOUR_CREATECOMMAND_H

#ifndef BATTLEFORHONOUR_CREATEUNITCOMMAND_H
#define BATTLEFORHONOUR_CREATEUNITCOMMAND_H


#include "Command.h"
#include "../../Objects/Units/Infantry/SwordMan.h"
#include "../../Objects/Units/Archer/CrossBowMan.h"
#include "../../Objects/Units/Druid/Hermit.h"
#include "../../Objects/Units/Archer/LongBowMan.h"

class CreateUnitCommand: public Command {

private:

    Point unitPos;
    UnitType unitType;

public:
```

```
        explicit CreateUnitCommand(Point position, UnitType unitType): unitPos(position), unitType(unitType){}
    void execute(GameState &gameState) override{

        if (!gameState.getNowPlayerBase()){
            Log::log << "Can't create a unit without a base" << Log::logend;
            return;
        }

        switch (unitType) {

            case UnitType::ARCHER:
                                                    gameState.getNowPlayerBase()->createUnit<LongBowMan>(unitPos);
                break;
            case UnitType::INFANTRY:
                                                    gameState.getNowPlayerBase()->createUnit<SwordMan>(unitPos);
                break;
            case UnitType::DRUID:
                gameState.getNowPlayerBase()->createUnit<Hermit>(unitPos);
                break;

        }
        Log::log << "Command to create a unit " << Log::logend;

    }

    [[nodiscard]] CommandSnapshot * getSnapshot() const override{

        std::stringstream stream;
         stream << "create unit " << unitPos.x << " " << unitPos.y << " " << static_cast<int>(unitType) << std::endl;
        return new CommandSnapshot(stream.str());

    }
```

```
};

class CreateUnitCommandHandler: public CommandHandler {

public:

   bool isHandle(std::vector<std::string> &terminal) override{

      return terminal.size() == 4 && terminal[0] == "unit";

   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override {

      if (isHandle(terminal)){
         int x = convertStr(terminal[1]);
         int y = convertStr(terminal[2]);
         auto type = static_cast<UnitType>(std::stoi(terminal[3]));
         Point basePos(x, y);
                                    return    std::unique_ptr<Command>(new
CreateUnitCommand(basePos, type));
      }

      if (next)
         return next->handle(terminal);
      return std::make_unique<Command>();

   }

};


#endif //BATTLEFORHONOUR_CREATEUNITCOMMAND_H

#ifndef BATTLEFORHONOUR_CROSSBOWMAN_H
#define BATTLEFORHONOUR_CROSSBOWMAN_H
```

```
#include "Archer.h"
#include "../../../Armor/Armor.h"

class CrossBowMan: public Archer{

public:
   CrossBowMan():
      Archer(*ArmorFlyweight::getFlyweight<LeatherArmor>(), 100){}
};



#endif //BATTLEFORHONOUR_CROSSBOWMAN_H

#ifndef BATTLEFORHONOUR_DRUID_H
#define BATTLEFORHONOUR_DRUID_H


#include "../../Unit.h"
#include "../../../Armor/Armor.h"
#include "../../../Weapon/Weapon.h"
#include "../../../Weapon/WeaponFlyweight.h"
#include "../../../Armor/ArmorFlyweight.h"

class Druid: public Unit{

public:
   Druid(Weapon &weapon, int health):
         Unit(UnitType::DRUID,  *ArmorFlyweight::getFlyweight<Robe>(),
weapon, health){}
};


#endif //BATTLEFORHONOUR_DRUID_H
```

```
#ifndef BATTLEFORHONOUR_EXITCOMMAND_H
#define BATTLEFORHONOUR_EXITCOMMAND_H

#include "Command.h"

class ExitCommand: public Command{

   void execute(GameState &gameState) override{
      exit(0);
   }

};

class ExitCommandHandler: public CommandHandler{

public:

   bool isHandle(std::vector<std::string> &terminal) override{
      return terminal.size() == 1 && terminal[0] == "exit";
   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

      if (isHandle(terminal)){
         terminal.erase(terminal.begin());
         return std::unique_ptr<Command>(new ExitCommand());
      }

      if (next)
         return next->handle(terminal);

      return std::make_unique<Command>();

   }

};
```

```
#endif //BATTLEFORHONOUR_EXITCOMMAND_H

#include "FieldCell.h"

FieldCell::FieldCell(Terrain *terrain): FieldCell() {
    this->terrain = terrain;
}

bool FieldCell::setObject(GameObject *object) {
    if (isEmpty()){
        this->object = object;
        return true;
    } else
        return false;
}

bool FieldCell::setTerrain(Terrain *terrain) {
    if (!this->terrain){
        this->terrain = terrain;
        return true;
    } else
        return false;
}

void FieldCell::eraseObject() {
    this->object = nullptr;
}

std::ostream &operator<<(std::ostream &stream, const FieldCell &cell) {

    stream << " ";

    if (cell.terrain){

        if (cell.object)
```

```
            cell.terrain->print(stream, *cell.object);
        else
            stream << *(cell.terrain);

    } else{

        if (cell.object)
            stream << *(cell.object);
        else
            stream << "#";

    }
    return stream;
}

FieldCell::FieldCell(FieldCell &&other):
        object(other.object),
        terrain(other.terrain)
        {
    other.object = nullptr;
}

FieldCell &FieldCell::operator=(FieldCell &&other) {
    if (&other == this)
        return *this;

    object = other.object;
    other.object = nullptr;

    return *this;

}

FieldCell::FieldCell(const FieldCell &cell):
        object(cell.object),
        terrain(cell.terrain){}
```

```
FieldCell &FieldCell::operator=(const FieldCell &cell) {
    object = cell.object;
    terrain = cell.terrain;

    return *this;

}
#ifndef BATTLEFORHONOUR_FIELDCELL_H
#define BATTLEFORHONOUR_FIELDCELL_H


#include <ostream>
#include "../Terrains/Terrain.h"
#include "../Objects/GameObject.h"

class FieldCell {

private:

    GameObject *object;
    Terrain *terrain;
    bool ismovable;

public:

    FieldCell():
        object(nullptr),
        terrain(nullptr),
        ismovable(true){}

    FieldCell(const FieldCell& cell);
    FieldCell(FieldCell &&other);
    explicit FieldCell(Terrain *terrain);

    bool isEmpty() {
        return object == nullptr;
    }
```

```
    [[nodiscard]] GameObject *getObject() const {
        return object;
    }
    [[nodiscard]] Terrain *getTerrain() const {
        return terrain;
    }
    bool isMovable() const{
        return this->ismovable;
    }
    bool setObject(GameObject *object);
    bool setTerrain(Terrain *terrain);
    void eraseObject();

    void changeMovable(bool flag){
        this->ismovable = flag;
    }

     friend std::ostream& operator<< (std::ostream &stream, const FieldCell
&cell);
    FieldCell& operator=(FieldCell &&other);
    FieldCell& operator=(const FieldCell &cell);

};

#endif //BATTLEFORHONOUR_FIELDCELL_H

#ifndef BATTLEFORHONOUR_FILELOGGER_H
#define BATTLEFORHONOUR_FILELOGGER_H


#include <fstream>
#include "Logger.h"

class FileLogger: public Logger {

private:
```

```
    std::ofstream fileStream;

public:

    explicit FileLogger(const std::string& filePath): fileStream(filePath){}

    ~FileLogger() override {
        fileStream.close();
    }

    void log(std::string &fs) override{
        fileStream << fs;
    }

    void log(Log::LogEnd &l) override{
        fileStream.flush();
    }

};


#endif //BATTLEFORHONOUR_FILELOGGER_H

#ifndef BATTLEFORHONOUR_GAMEFACADE_H
#define BATTLEFORHONOUR_GAMEFACADE_H


#include <sstream>
#include "GameState.h"
#include "../User/CommandInterpreter.h"


template<typename Rule, int playersCount>
class GameFacade: public GameState {

private:
```

```
    CommandInterpreter actCommand;
    Rule rule;
    GameFacade(int fieldWidth, int fieldHeight):
        GameState(playersCount, fieldWidth, fieldWidth, new Rule){}

public:

    static GameFacade& single(){
        Rule rule;
        static GameFacade subSystem(rule.fieldWidth, rule.fieldHeight);
        return subSystem;
    }

        friend std::ostream &operator<<(std::ostream &stream, const
GameFacade &game){
        stream << "Current user: " << game.currentUser << std::endl;
        stream << game.gameField << std::endl;
        return stream;
    }

    bool isOver(){
        return rule.isOver(*this);
    }

    void nextTurn(){

        std::string commandString;
        std::getline(std::cin, commandString);

        std::cout << "----------------------------------------" << std::endl;

                                    std::unique_ptr<Command>    command    =
actCommand.handle(commandString);
        try {
            command->execute(*this);
        } catch(DoubleBasePlacingExc &exception) {
```

```
            std::cout << "User " << exception.playerIndex << " trying to place
second base." << std::endl;
        } catch (DoublePlacingExc &exception){
            std::cout << "This cell is full by another object." << std::endl;
        } catch (OutOfRangeExc &exception){
             std::cout << "Out of range. Cell point " << exception.x << " " <<
exception.y << " is not exist." << std::endl;
        } catch (ImpossibleMoveExc &exception){
            std::cout << "Can't move to this cell. They busy by other object." <<
std::endl;
        } catch (InvalidFileLoadExc &exception){
            std::cout << "Wrong file." << std::endl;
        } catch (...){
            std::cout << "Unknown error." << std::endl;
        }
        gameActions.push_back(command->getSnapshot());
        nextUser();
    }

};


#endif //BATTLEFORHONOUR_GAMEFACADE_H

#include "GameField.h"
#include "../Exceptions/StackExceptions.h"


GameField::GameField():
        fieldHeight(0),
        fieldWidth(0),
        field(nullptr)
{}

void GameField::setBorders(){
    int i;
    int j;
```

```
    for(i = 0, j = 0; i < fieldHeight; i++) {
        Border *border;
        field[i][j].setTerrain(border);
        field[i][j].changeMovable(false);
    }
    for(i = 0, j = 0; j < fieldWidth; j++) {
        Border *border;
        field[i][j].setTerrain(border);
        field[i][j].changeMovable(false);
    }

    for(i = 0, j = fieldWidth - 1; j >= 0; j--) {
        Border *border;
        field[i][j].setTerrain(border);
        field[i][j].changeMovable(false);
    }
    for(i = fieldHeight - 1, j = 0; j < fieldWidth; j++) {
        Border *border;
        field[i][j].setTerrain(border);
        field[i][j].changeMovable(false);
    }
}

GameField::GameField(int fieldHeight, int fieldWidth):
        fieldHeight(fieldHeight),
        fieldWidth(fieldWidth)
{
    field = new FieldCell* [fieldHeight];
    for (int i = 0; i < fieldHeight; i++){
        field[i] = new FieldCell [fieldWidth];
    }
}

void GameField::deleteObject(int x, int y) {

    if (y < 0 || y > fieldHeight || x < 0 || x > fieldWidth){
        throw OutOfRangeExc(x, y);
```

```
    }
    field[y][x].eraseObject();

}

bool GameField::addObject(GameObject *object, int x, int y) {

    if (object->isOnField){
        Log::log << "[#GameField] Impossible addition of " << *object << "
on field" << Log::logend;
        throw DoublePlacingExc();
    }

    bool isInBorder = x < fieldWidth && y < fieldHeight && x >= 0 && y >=
0;

    if (isInBorder && field[y][x].isEmpty()){

        field[y][x].setObject(object);
        object->pos = Point(x, y);
        object->isOnField = true;

    } else{

        Log::log << "[#GameField] Impossible addition of " << *object << "
on field" << Log::logend;
        throw OutOfRangeExc(x, y);

    }

    return true;
}

void GameField::deleteObject(GameObject *object) {
    deleteObject(object->pos.x, object->pos.y);
}
```

```
void GameField::moveObject(const Point &p1, const Point &p2) {

        if (checkBorder(p1) && checkBorder(p2) && !field[p1.y]
[p1.x].isEmpty() && field[p2.y][p2.x].isEmpty()){

    field[p2.y][p2.x] = std::move(field[p1.y][p1.x]);
    field[p2.y][p2.x].getObject()->pos = p2;
    field[p1.y][p1.x].eraseObject();

  } else{

     Log::log << "[#GameField] Impossible to move object from " << p1.x
<< " " << p1.y << " to " << p2.x << " " << p2.y << Log::logend;
    throw ImpossibleMoveExc();

  }

}


void GameField::deleteObject(const Point &point) {
  deleteObject(point.x, point.y);
}


FieldCell *GameField::getCell(const Point &p) const{

  if (p.x < fieldWidth && p.y < fieldHeight)
    return &field[p.y][p.x];
  throw OutOfRangeExc(p.x, p.y);
}

FieldCell *GameField::getCell(const int x, const int y) {
  if (x < fieldWidth && y < fieldHeight)
    return &field[y][x];
  throw OutOfRangeExc(x, y);
}
```

```
GameField::~GameField() {

   for (int i=0; i<fieldHeight; i++){
      delete []field[i];
   }

   delete []field;

}

void GameField::reset() {

   for (int i=0; i<fieldHeight; i++){
      delete []field[i];
   }

   delete []field;

   field = new FieldCell* [fieldHeight];
   for (int i=0; i<fieldHeight; i++){
      field[i] = new FieldCell [fieldWidth];
   }
}

std::ostream &operator<<(std::ostream &stream, const GameField &field)
{

   for (int y = 0; y < field.fieldHeight; y++){
      for (int x = 0; x < field.fieldWidth; x++){
         stream << field.field[y][x];
      }
      stream << std::endl;
   }
   return stream;
}
```

```
void GameField::onUnitMove(Unit *unit, Point p) {

   FieldCell *cell = getCell(p);

            if (!cell->isEmpty() && cell->getObject()->getType() ==
ObjectType::NEUTRAL_OBJECT){

               auto *neutralObject = dynamic_cast<NeutralObject*>(cell-
>getObject());

      switch (unit->getUnitType()){
         case UnitType::INFANTRY:
            neutralObject->setStrategy(new InfantryStrategy());
            break;
         case UnitType::ARCHER:
            neutralObject->setStrategy(new ArcherStrategy());
            break;
         case UnitType::DRUID:
            neutralObject->setStrategy(new DruidStrategy());
            break;
      }
      (*unit) << neutralObject;
      cell->eraseObject();
   }
   moveObject(unit->getPosition(), p);

}

void GameField::onUnitDestroy(Unit *unit) {
   deleteObject(unit->getPosition());
}

bool GameField::addObject(GameObject *object, Point position) {
   return addObject(object, position.x, position.y);
}

void GameField::onBaseNewUnit(Unit *unit, Point pos) {
```

```
    bool isPossibleAdd = addObject(unit, pos);
    if (isPossibleAdd)
        unit->addObserver(this);

}

bool GameField::addBase(Base *base, Point pos) {

    return addBase(base, pos.x, pos.y);

}

bool GameField::addBase(Base *base, int x, int y) {

    bool isPossibleAdd = addObject(base, x, y);
    if (isPossibleAdd)
        base->addObserver(this);
    return isPossibleAdd;

}

void GameField::onUnitAttack(Unit *unit, Unit *enemy) {

    Terrain *terrain = getCell(unit->getPosition())->getTerrain();
    TerrainProxy terrainProxy(terrain);

                int damage = unit->getWeapon().getDamage() +
terrainProxy.getDamageMultiply(unit->getWeapon().getType());
                int def = enemy->getArmor().controlAbsorb() +
terrainProxy.getAbsorbMultiply(enemy->getArmor().getArmorType());

    int resDamage = damage - def;

    if (resDamage < 0)
        resDamage = 0;
```

```
    enemy->damage(resDamage);
}

bool GameField::checkBorder(const Point &p) const {
    return p.x >= 0 && p.y >= 0 && p.x < fieldWidth && p.y < fieldHeight;
}


#ifndef BATTLEFORHONOUR_GAMEFIELD_H
#define BATTLEFORHONOUR_GAMEFIELD_H


#include "Point.h"
#include "GameFieldIterator.h"
#include "FieldCell.h"
#include "../Objects/Neutrals/InfantryStrategy.h"
#include "../Objects/Neutrals/ArcherStrategy.h"
#include "../Objects/Neutrals/DruidStrategy.h"
#include "../Objects/Base.h"
#include <iostream>

class GameField: public UnitObserver, public BaseObserver {

private:

    FieldCell **field;

    int fieldHeight;
    int fieldWidth;

public:

    GameField();
    GameField(int fieldHeight, int fieldWidth);
    ~GameField();
    void reset();
```

```
    void deleteObject(int x, int y);
    void deleteObject(const Point &point);
    void deleteObject(GameObject *object);

    bool addObject(GameObject *object, int x, int y);
    bool addObject(GameObject *object, Point p);

    void moveObject(const Point &p1, const Point &p2);
    void setBorders();

    [[nodiscard]] FieldCell *getCell(const Point &p) const;
    FieldCell *getCell(const int x, const int y);

        friend std::ostream& operator<< (std::ostream &stream, const
GameField &field);

     GameFieldIterator begin(){ return GameFieldIterator(Point(0, 0), field,
fieldHeight, fieldWidth); }
          GameFieldIterator end(){ return GameFieldIterator(Point(0,
fieldHeight), field, fieldHeight, fieldWidth); }

    void onUnitAttack(Unit *unit, Unit *enemy) override;
    void onUnitMove(Unit *unit, Point p) override;
    void onUnitDestroy(Unit *unit) override;
    void onUnitDamaged(Unit *unit) override {}
    void onUnitHeal(Unit *unit) override {}

    void onBaseNewUnit(Unit *unit, Point pos) override;
    bool addBase(Base *base, Point pos);
    bool addBase(Base *base, int x, int y);

    [[nodiscard]] bool checkBorder(const Point &p) const;

};

#endif //BATTLEFORHONOUR_GAMEFIELD_H
```

```
#ifndef BATTLEFORHONOUR_GAMEFIELDITERATOR_H
#define BATTLEFORHONOUR_GAMEFIELDITERATOR_H


#include <iterator>
#include "Point.h"
#include "FieldCell.h"

class GameFieldIterator: public std::iterator<std::input_iterator_tag, FieldCell>{

   friend class GameField;

private:

   Point point;
   const Point cpoint;
   FieldCell **field;
   const int fieldHeight;
   const int fieldWidth;

    GameFieldIterator(const Point p, FieldCell **field, const int fieldHeight, const int fieldWidth):
          point(p),
          cpoint(p),
          field(field),
          fieldWidth(fieldWidth),
          fieldHeight(fieldHeight){};

public:

   GameFieldIterator(const GameFieldIterator &it):
          point(it.point),
          field(it.field),
          fieldWidth(it.fieldWidth),
          fieldHeight(it.fieldHeight){};
```

```
    bool operator!=(const GameFieldIterator &sub) {
        return cpoint != sub.point;
    };
    bool operator==(const GameFieldIterator &sub) {
        return cpoint == sub.point;
    };
    typename GameFieldIterator::reference operator*() {
        return  field[point.y][point.x];
    };

    GameFieldIterator&  operator++() {

        Point next = point;
        next.x++;

        if (next.x < fieldWidth) {
            point = next;

            return *this;
        } else{
            next.x = 0;
            next.y++;
            point = next;

            return *this;
        }

    };


};


#endif //BATTLEFORHONOUR_GAMEFIELDITERATOR_H

#include "GameObject.h"
```

```
std::ostream &operator<<(std::ostream &stream, const GameObject &object){
   object.print(stream);
   return stream;
}
#ifndef BATTLEFORHONOUR_GAMEOBJECT_H
#define BATTLEFORHONOUR_GAMEOBJECT_H


#include <ostream>
#include "../GameField/Point.h"
#include "../Logs/Log.h"
#include "ObjectType.h"


class GameObject {

   friend class GameField;

protected:

   ObjectType type;
   Point pos;
   bool isOnField = false;

   virtual void print(std::ostream &stream) const = 0;

public:

   explicit GameObject(ObjectType type): type(type){}
   Point getPosition() {
      return pos;
   }
   ObjectType getType() {
      return type;
   }
```

```
        friend std::ostream &operator<<(std::ostream &stream, const GameObject &object);
   friend LogProxy& operator<<(LogProxy &logger, GameObject &object)
{

      logger << "Object = x: " << object.pos.x << " y: " << object.pos.y;
      return logger;

   }

};


#endif //BATTLEFORHONOUR_GAMEOBJECT_H

#ifndef BATTLEFORHONOUR_GAMERULE_H
#define BATTLEFORHONOUR_GAMERULE_H


class GameState;

class GameRule {

public:
   int fieldWidth;
   int fieldHeight;
   virtual bool isOver(GameState &gameState)=0;
   virtual int nextUser(GameState &gameState)=0;

   GameRule(int fieldWidth, int fieldHeight):
         fieldWidth(fieldWidth),
         fieldHeight(fieldHeight){}

};

#endif //BATTLEFORHONOUR_GAMERULE_H
```

```
#ifndef BATTLEFORHONOUR_GAMESTATE_H
#define BATTLEFORHONOUR_GAMESTATE_H


#include "../User/Commands/CommandSnapshot.h"
#include "../GameField/GameField.h"
#include "../Exceptions/StackExceptions.h"
#include "../GameSettings/GameRule.h"

class GameState {

protected:

   GameField gameField;
   std::vector<Base*> userBases;
   std::vector<CommandSnapshot*> gameActions;
   int currentUser;
   GameRule *rule;

public:

    GameState(int playersCount, int fieldWidth, int fieldHeight, GameRule
*rule):
          gameField(fieldHeight, fieldWidth),
          userBases(playersCount, nullptr),
          currentUser(0),
          rule(rule)
   {}

   Base *getNowPlayerBase(){
      return userBases[currentUser];
   }
   bool setNowPlayerBase(Base *base){
      if (userBases[currentUser]){
         throw DoubleBasePlacingExc(currentUser);
      } else{
         userBases[currentUser] = base;
```

```
            return true;
        }
    }

    [[nodiscard]] int getNowPlayerIndex() const{
        return currentUser;
    }

    void newGame(){
        int playersCount = userBases.size();
        gameField.reset();
        userBases.clear();
        gameActions.clear();
        userBases.resize(playersCount, nullptr);
    }

    void addAction(CommandSnapshot *snapshot){
        gameActions.push_back(snapshot);
    }

    void nextUser(){
        currentUser = rule->nextUser(*this);
    }

    std::vector<CommandSnapshot*> getActions(){
        return gameActions;
    }

    GameField &getField(){
        return gameField;
    }

    const std::vector<Base*> &getBases(){
        return userBases;
    }

};
```

```
#endif //BATTLEFORHONOUR_GAMESTATE_H

#ifndef BATTLEFORHONOUR_HERMIT_H
#define BATTLEFORHONOUR_HERMIT_H


#include "Druid.h"
#include "../../../Weapon/Weapon.h"

class Hermit: public Druid{

public:
   Hermit():
      Druid(*WeaponFlyweight::getFlyWeight<AbolishMagic>(), 100){}
};

#endif //BATTLEFORHONOUR_HERMIT_H

#ifndef BATTLEFORHONOUR_HILLKING_H
#define BATTLEFORHONOUR_HILLKING_H


#include "PlayerState.h"
#include "GameRule.h"
#include "../Game/GameState.h"

class HillKing: public GameRule {

private:
   PlayerState* nowState;
public:

   HillKing():
         GameRule( 15, 15),
         nowState(new FirstPlayer){}
```

```
   bool isOver(GameState &gameState) override {
                        if(!gameState.getField().getCell(7,    7)->isEmpty()    &&
gameState.getField().getCell(7,            7)->getObject()->getType()          !=
ObjectType::BASE){
          Log::log << "Game over" << Log::logend;
               std::cout << "User " << gameState.getNowPlayerIndex() << "
won!";
          return true;
       }
       else
          return false;
   }

   int nextUser(GameState &gameState) override {
            int currUserPos = (gameState.getNowPlayerIndex() + nowState-
>getNextPlayerRecr()) % gameState.getBases().size();
       auto nextState = nowState->getNextPlayerState();
       delete nowState;
       nowState = nextState;
       if (nowState == nullptr)
          nowState = new FirstPlayer;
       return currUserPos;
   }

};

#endif //BATTLEFORHONOUR_HILLKING_H
#ifndef BATTLEFORHONOUR_INFANTRY_H
#define BATTLEFORHONOUR_INFANTRY_H


#include "../../Unit.h"
#include "../../../Armor/Armor.h"
#include "../../../Weapon/WeaponFlyweight.h"
#include "../../../Armor/ArmorFlyweight.h"

class Infantry: public Unit {
```

```
public:
   Infantry(Weapon &weapon, int health):
                                                    Unit(UnitType::INFANTRY,
*ArmorFlyweight::getFlyweight<PlateMail>(), weapon, health) {}
};


#endif //BATTLEFORHONOUR_INFANTRY_H

#ifndef BATTLEFORHONOUR_LOADCI_H
#define BATTLEFORHONOUR_LOADCI_H


#include "Commands/Command.h"
#include "Commands/AttackCommand.h"
#include "Commands/CreateCommand.h"
#include "Commands/MoveCommand.h"
#include "Commands/ShowCommand.h"
#include "Commands/ExitCommand.h"
#include "Commands/NewCommand.h"
#include "Commands/SkipCommand.h"

class LoadCI {

private:

   AttackCommandHandler *attackHandler;
   CreateCommandHandler *createHandler;
   MoveCommandHandler *moveHandler;
   ShowCommandHandler *showHandler;
   ExitCommandHandler *exitHandler;
   NewCommandHandler *newHandler;
   SkipCommandHandler *skipHandler;


public:
```

```
LoadCI(){
   newHandler = new NewCommandHandler();
   attackHandler = new AttackCommandHandler();
   createHandler = new CreateCommandHandler();
   moveHandler = new MoveCommandHandler();
   showHandler = new ShowCommandHandler();
   exitHandler = new ExitCommandHandler();
   skipHandler = new SkipCommandHandler();

   attackHandler->setNext(createHandler);
   createHandler->setNext(moveHandler);
   moveHandler->setNext(showHandler);
   showHandler->setNext(exitHandler);
   exitHandler->setNext(newHandler);
   newHandler->setNext(skipHandler);
}

std::unique_ptr<Command> handle(std::string commandString){

   std::vector <std::string> splitCommands;
   std::stringstream stream(commandString);
   std::string commandWord;
   while (stream >> commandWord)
      splitCommands.push_back(commandWord);

   return attackHandler->handle(splitCommands);

}

~LoadCI(){
   delete attackHandler;
   delete createHandler;
   delete moveHandler;
   delete showHandler;
   delete exitHandler;
   delete skipHandler;
```

```
    }

};

#endif //BATTLEFORHONOUR_LOADCI_H

#ifndef BATTLEFORHONOUR_LOADCOMMAND_H
#define BATTLEFORHONOUR_LOADCOMMAND_H


#include "../LoadCI.h"

class LoadCommand: public Command {

private:

    std::ifstream fs;
    LoadCI inter;

public:

    explicit LoadCommand(std::string &filename): fs(filename){}
    void execute(GameState &gameState) override{

        gameState.newGame();
        std::string terminal;
        std::hash<std::string> toHash;
        unsigned long int calculatedHash = 0;
        unsigned long int fileHash = 0;

        std::string fileHashStr;
        std::getline(fs, fileHashStr);

        fileHash = convertStr(fileHashStr);
```

```
    while (std::getline(fs, terminal)){

        std::unique_ptr<Command> command = inter.handle(terminal);
        try {
            command->execute(gameState);
        } catch(DoubleBasePlacingExc &exception) {
            Log::log << "[#FileLoader]" << "User " << exception.playerIndex
<< " trying to place second base." << Log::logend;
        } catch (DoublePlacingExc &exception){
                Log::log << "[#FileLoader]" << "This cell is busy by other
object." << Log::logend;
        } catch (OutOfRangeExc &exception){
                 Log::log << "[#FileLoader]" << "Out of range. Cell " <<
exception.x << " " << exception.y << " is not exist." << Log::logend;
        } catch (ImpossibleMoveExc &exception){
              Log::log << "[#FileLoader]" << "Can't move to this cell. They
busy by other object." << Log::logend;
        } catch (...){
                    Log::log << "[#FileLoader]" << "Unknown error." <<
Log::logend;
        }
        auto snapshot = command->getSnapshot();
        gameState.addAction(snapshot);
        calculatedHash += snapshot->getHash(toHash);
        gameState.nextUser();

    }

    Log::log << "String hash: " << fileHashStr << Log::logend;
    Log::log << "Integer hash: " << fileHash << Log::logend;
    Log::log << "Calculated hash: " << calculatedHash << Log::logend;
    Log::log << "Commands were read: " << gameState.getActions().size()
<< Log::logend;

    if (fileHash != calculatedHash){
            Log::log << "Wrong file format. File may be incorrect." <<
Log::logend;
```

```
            throw InvalidFileLoadExc();
        }

    }

    ~LoadCommand() override{
        fs.close();
    }

};

class LoadCommandHandler: public CommandHandler{

public:

    bool isHandle(std::vector<std::string> &terminal) override{
        return terminal.size() == 2 && terminal[0] == "load";
    }

     std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
                                        return    std::unique_ptr<Command>(new
LoadCommand(terminal[1]));
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();

    }

};
```

```
#endif //BATTLEFORHONOUR_LOADCOMMAND_H

#include "Log.h"

LogProxy Log::log = LogProxy();

#ifndef BATTLEFORHONOUR_LOG_H
#define BATTLEFORHONOUR_LOG_H


#include "LogProxy.h"
#include "LogEnd.h"

namespace Log{

   extern LogProxy log;
   const LogEnd logend;

}

#endif //BATTLEFORHONOUR_LOG_H

#ifndef BATTLEFORHONOUR_LOGEND_H
#define BATTLEFORHONOUR_LOGEND_H


namespace Log {
   class LogEnd {};
}

#endif //BATTLEFORHONOUR_LOGEND_H

#ifndef BATTLEFORHONOUR_LOGGER_H
#define BATTLEFORHONOUR_LOGGER_H

#include <string>
```

```
#include "LogEnd.h"

class Logger {

public:
   virtual void log(std::string &str)=0;
   virtual void log(Log::LogEnd &l){}
   virtual ~Logger(){}
};

#endif //BATTLEFORHONOUR_LOGGER_H

#ifndef BATTLEFORHONOUR_LOGPROXY_H
#define BATTLEFORHONOUR_LOGPROXY_H

#include "NoLogger.h"
#include "LogString.h"
#include <string>
#include <iostream>

class LogProxy {

private:

   Logger *logger;
   LogString *logString;
   bool firstLine = true;

   void log(std::string s){
      if (firstLine) {
         std::string toLog = logString->getString(s);
         logger->log(toLog);
         firstLine = false;
      } else{
         logger->log(s);
      }
   }
```

```
public:

   LogProxy():
         logger(new NoLogger()){}

   ~LogProxy(){
      delete logger;
      delete logString;
   }

   friend LogProxy& operator<< (LogProxy &logger, const std::string &s){
      logger.log(s);
      return logger;
   }

   friend LogProxy& operator<< (LogProxy &logger, const int i){
      logger.log(std::to_string(i));
      return logger;
   }

    friend LogProxy& operator<< (LogProxy &logger, const Log::LogEnd
&l){
      logger.log("\n");
      logger.firstLine = true;
      return logger;
   }

   void setLogFormat(Logger *tmp){

      delete logger;
      logger = tmp;
   }

   void setLogStrOutput(LogString *tmp){
      delete logString;
      logString = tmp;
```

```
    }

};


#endif //BATTLEFORHONOUR_LOGPROXY_H

#ifndef BATTLEFORHONOUR_LOGSTRING_H
#define BATTLEFORHONOUR_LOGSTRING_H


#include <string>

class LogString {
public:
    std::string getString(std::string &str){
        return str;
    }
};

#endif //BATTLEFORHONOUR_LOGSTRING_H

#ifndef BATTLEFORHONOUR_LONGBOWMAN_H
#define BATTLEFORHONOUR_LONGBOWMAN_H


#include "Archer.h"
#include "../../../Armor/Armor.h"

class LongBowMan: public Archer{

public:
    LongBowMan():
        Archer(*ArmorFlyweight::getFlyweight<PlateMail>(), 50){}
};
```

```
#endif //BATTLEFORHONOUR_LONGBOWMAN_H

#include <iostream>
#include "Game/GameFacade.h"
#include "Logs/FileLogger.h"
#include "Logs/CmdLogger.h"
#include "GameSettings/BigGame.h"
#include "GameSettings/MidGame.h"
#include "GameSettings/SmallGame.h"
#include "GameSettings/HillKing.h"

int main() {

    Log::log.setLogFormat(new CmdLogger());
    Log::log.setLogStrOutput(new LogString());

    auto game = GameFacade<SmallGame, 2>::single();

    while (!game.isOver()){
        std::cout << game;
        game.nextTurn();
    }

    return 0;
}
#ifndef BATTLEFORHONOUR_MIDGAME_H
#define BATTLEFORHONOUR_MIDGAME_H


#include "GameRule.h"
#include "PlayerState.h"

class MidGame: public GameRule {

private:

    PlayerState* nowState;
```

```
public:

    MidGame():
        GameRule(10, 10),
        nowState(new FirstPlayer){}

    bool isOver(GameState &gameState) override {
        int liveCount = gameState.getBases().size();
        for (auto b: gameState.getBases()){
            if (b && b->getHealth() <= 0){
                liveCount--;
            }
        }

        return liveCount <= 1;
    }

    int nextUser(GameState &gameState) override {
            int nowPlayerIndex = (gameState.getNowPlayerIndex() + nowState-
>getNextPlayerRecr()) % gameState.getBases().size();
        auto nextState = nowState->getNextPlayerState();
        delete nowState;
        nowState = nextState;
        if (nowState == nullptr)
            nowState = new FirstPlayer;
        return nowPlayerIndex;
    }

};


#endif //BATTLEFORHONOUR_MIDGAME_H

#ifndef BATTLEFORHONOUR_MOVECOMMAND_H
#define BATTLEFORHONOUR_MOVECOMMAND_H
```

```
#include "Command.h"
#include "MoveUnitCommand.h"

class MoveCommandHandler: public CommandHandler{

public:

   bool isHandle(std::vector<std::string> &terminal) override{
      return terminal.size() > 1 && terminal[0] == "move";
   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

      if (isHandle(terminal)){

         terminal.erase(terminal.begin());
         auto handleTemp = new MoveUnitCommandHandler();
         return handleTemp->handle(terminal);

      }
      if (next)
         return next->handle(terminal);

      return std::make_unique<Command>();
   }

};


#endif //BATTLEFORHONOUR_MOVECOMMAND_H

#ifndef BATTLEFORHONOUR_MOVEUNITCOMMAND_H
#define BATTLEFORHONOUR_MOVEUNITCOMMAND_H
```

```
#include "Command.h"

class MoveUnitCommand: public Command {

private:

   Point from;
   Point to;

public:

   MoveUnitCommand(Point from, Point to):
      from(from),
      to(to){}
   void execute(GameState &gameInfo) override{

      auto object = gameInfo.getField().getCell(from)->getObject();
      if (object && object->getType() == ObjectType::UNIT){
         auto unit1 = dynamic_cast<Unit *>(object);
         unit1->move(to);
         Log::log << "Command to unit moved" << Log::logend;
      } else
         Log::log << "No unit on this cell" << Log::logend;
   }

   [[nodiscard]] CommandSnapshot * getSnapshot() const override{

      std::stringstream stream;
       stream << "move unit " << from.x << " " << from.y << " " << to.x <<
" " << to.y << std::endl;
      return new CommandSnapshot(stream.str());

   }

};

class MoveUnitCommandHandler: public CommandHandler {
```

```
public:

    bool isHandle(std::vector<std::string> &terminal) override{

        return terminal.size() == 5 && terminal[0] == "unit";

    }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){

            int x1 = convertStr(terminal[1]);
            int y1 = convertStr(terminal[2]);
            int x2 = convertStr(terminal[3]);
            int y2 = convertStr(terminal[4]);
            Point from(x1, y1);
            Point to(x2, y2);
             return std::unique_ptr<Command>(new MoveUnitCommand(from,
to));
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();

    }

};

#endif //BATTLEFORHONOUR_MOVEUNITCOMMAND_H

#ifndef BATTLEFORHONOUR_NEWCOMMAND_H
#define BATTLEFORHONOUR_NEWCOMMAND_H
```

```
#include "Command.h"
#include "NewGameCommand.h"

#include <memory>

class NewCommandHandler: public CommandHandler{

public:

   bool isHandle(std::vector<std::string> &terminal) override{
      return terminal.size() > 1 && terminal[0] == "new";
   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

      if (isHandle(terminal)){

         terminal.erase(terminal.begin());
         auto handlerTemp = new NewGameCommandHandler;

         return handlerTemp->handle(terminal);
      }

      if (next)
         return next->handle(terminal);

      return std::make_unique<Command>();

   }

};
```

```
#endif //BATTLEFORHONOUR_NEWCOMMAND_H

#ifndef BATTLEFORHONOUR_NEWGAMECOMMAND_H
#define BATTLEFORHONOUR_NEWGAMECOMMAND_H


#include "Command.h"

class NewGameCommand: public Command {

public:

   explicit NewGameCommand(){}
   void execute(GameState &gameState) override{
      gameState.newGame();
   }

};

class NewGameCommandHandler: public CommandHandler {

public:

   bool isHandle(std::vector<std::string> &terminal) override{
      return terminal.size() == 1 && terminal[0] == "game";
   }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

      if (isHandle(terminal)){
         return std::unique_ptr<Command>(new NewGameCommand());
      }

      if (next)
         return next->handle(terminal);
```

```
        return std::make_unique<Command>();
    }

};


#endif //BATTLEFORHONOUR_NEWGAMECOMMAND_H

#ifndef BATTLEFORHONOUR_NOLOGGER_H
#define BATTLEFORHONOUR_NOLOGGER_H


#include "Logger.h"

class NoLogger: public Logger {

public:
    void log(std::string &s) override{
        return;
    }
};


#endif //BATTLEFORHONOUR_NOLOGGER_H

#ifndef BATTLEFORHONOUR_OBJECTTYPE_H
#define BATTLEFORHONOUR_OBJECTTYPE_H
enum class ObjectType{
    UNIT,
    BASE,
    NEUTRAL_OBJECT
};
#endif //BATTLEFORHONOUR_OBJECTTYPE_H

#ifndef BATTLEFORHONOUR_OBSERVERS_H
#define BATTLEFORHONOUR_OBSERVERS_H
```

```
#include "../Objects/Unit.h"
#include "../GameField/Point.h"

class Unit;

class UnitObserver {

public:

   virtual void onUnitAttack(Unit *unit, Unit *other) = 0;
   virtual void onUnitMove(Unit *unit, Point p) = 0;
   virtual void onUnitDestroy(Unit *unit) = 0;
   virtual void onUnitHeal(Unit *unit) = 0;
   virtual void onUnitDamaged(Unit *unit) = 0;

};

class BaseObserver {

public:
   virtual void onBaseNewUnit(Unit *unit, Point position) = 0;
};


#endif //BATTLEFORHONOUR_OBSERVERS_H

#ifndef BATTLEFORHONOUR_PLAYERSTATE_H
#define BATTLEFORHONOUR_PLAYERSTATE_H


class PlayerState {

public:
   virtual int getNextPlayerRecr()=0;
   virtual PlayerState* getNextPlayerState()=0;
};
```

```
class SecondPlayer: public PlayerState {

   int getNextPlayerRecr() override{
      return 2;
   }

   PlayerState* getNextPlayerState() override{
      return nullptr;
   }
};

class SpecPlayer: public PlayerState {
   int getNextPlayerRecr() override{
      return -1;
   }
   PlayerState* getNextPlayerState() override{
      return new SecondPlayer;
   }
};

class FirstPlayer: public PlayerState {

public:
   int getNextPlayerRecr() override{
      return 2;
   }
   PlayerState* getNextPlayerState() override{
      return new SpecPlayer;
   }
};


#endif //BATTLEFORHONOUR_PLAYERSTATE_H

#ifndef BATTLEFORHONOUR_POINT_H
#define BATTLEFORHONOUR_POINT_H
```

```
class Point {

public:
   int x, y;

   Point(int x, int y) {
      this->x = x;
      this->y = y;
   }

   Point() :
         x(0),
         y(0){}

   bool operator!=(Point &other) const {
      return !(x == other.x && y == other.y);
   }

   bool operator!=(Point other) const {
      return !(x == other.x && y == other.y);
   }

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

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

};

#endif //BATTLEFORHONOUR_POINT_H

#ifndef BATTLEFORHONOUR_PRIESTESS_H
#define BATTLEFORHONOUR_PRIESTESS_H
```

```
#include "Druid.h"
#include "../../../Weapon/Weapon.h"

class Priestess: public Druid{

public:
   Priestess():
      Druid(*WeaponFlyweight::getFlyWeight<StarFall>(), 20){}
};

#endif //BATTLEFORHONOUR_PRIESTESS_H

#ifndef BATTLEFORHONOUR_SAVECOMMAND_H
#define BATTLEFORHONOUR_SAVECOMMAND_H


#include "Command.h"

class SaveCommand: public Command {

private:

   std::ofstream fs;

public:

   explicit SaveCommand(std::string &filename){
      fs = std::ofstream(filename);
      Log::log << "File opened" << Log::logend;
      Log::log << "File is opened: " << fs.is_open() << Log::logend;

   }
   void execute(GameState &gameState) override{

      std::hash<std::string> toHash;
```

```
        unsigned long int fileHash = 0;

        Log::log << "Saving..." << Log::logend;

        auto actions = gameState.getActions();

        for (auto elem: actions){
            fileHash += elem->getHash(toHash);
        }

        fs << fileHash << std::endl;

        for (auto elem: actions){
            elem->saveToFile(fs);
        }

                        Log::log << "Saved commands count: " <<
gameState.getActions().size() << Log::logend;

    }

    ~SaveCommand() override{
        Log::log << "File closed" << Log::logend;
        fs.close();
        Log::log << "File is opened: " << fs.is_open() << Log::logend;
    }

};

class SaveCommandHandler: public CommandHandler{

    bool isHandle(std::vector<std::string> &terminal) override{
        return terminal.size() == 2 && terminal[0] == "save";

    }
```

```
    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
                                        return    std::unique_ptr<Command>(new
SaveCommand(terminal[1]));
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }

};


#endif //BATTLEFORHONOUR_SAVECOMMAND_H

#ifndef BATTLEFORHONOUR_SHOWBASECOMMAND_H
#define BATTLEFORHONOUR_SHOWBASECOMMAND_H


#include "Command.h"
#include "../../GameField/Point.h"

class ShowBaseCommand: public Command {

private:

    Point basePosition;

public:

    explicit ShowBaseCommand(Point p): basePosition(p){}
    void execute(GameState &gameInfo) override{
```

```
        auto object = gameInfo.getField().getCell(basePosition)->getObject();
        if (object && object->getType() == ObjectType::BASE){

            auto base = dynamic_cast<Base*>(object);
            std::cout << "Base: " << std::endl
                    << "HP: " << base->getHealth() << std::endl
                    << "Armor: " << base->getArmor() << std::endl
                      << "Max Objects Count: " << base->getMaxObjectsCount()
<< std::endl;
            Log::log << "Show base command" << Log::logend;

        } else{
            Log::log << "Empty cell" << Log::logend;
        }

    }

};

class ShowBaseCommandHandler: public CommandHandler {

public:

    bool isHandle(std::vector<std::string> &terminal) override{

        return terminal.size() == 3 && terminal[0] == "base";

    }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
            int x = convertStr(terminal[1]);
            int y = convertStr(terminal[2]);
            Point basePosition(x, y);
```

```
                                    return    std::unique_ptr<Command>(new
ShowBaseCommand(basePosition));
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }

};


#endif //BATTLEFORHONOUR_SHOWBASECOMMAND_H

#ifndef BATTLEFORHONOUR_SHOWCOMMAND_H
#define BATTLEFORHONOUR_SHOWCOMMAND_H


#include "Command.h"

#include <memory>
#include "ShowBaseCommand.h"
#include "ShowUnitCommand.h"

class ShowCommandHandler: public CommandHandler{

public:

    bool isHandle(std::vector<std::string> &terminal) override{

        return terminal.size() > 1 && terminal[0] == "show";

    }

    std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{
```

```
        if (isHandle(terminal)){

            terminal.erase(terminal.begin());

            auto handlerUnit = new ShowUnitCommandHandler;
            auto handlerBase = new ShowBaseCommandHandler;

            handlerUnit->setNext(handlerBase);

            return handlerUnit->handle(terminal);
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }

};


#endif //BATTLEFORHONOUR_SHOWCOMMAND_H

#ifndef BATTLEFORHONOUR_SHOWUNITCOMMAND_H
#define BATTLEFORHONOUR_SHOWUNITCOMMAND_H


#include "Command.h"

class ShowUnitCommand: public Command {

private:

    Point unitPosition;

public:
```

```
    explicit ShowUnitCommand(Point p): unitPosition(p){}
    void execute(GameState &gameInfo) override{

        auto object = gameInfo.getField().getCell(unitPosition)->getObject();
        if (object && object->getType() == ObjectType::UNIT){

            auto unit = dynamic_cast<Unit*>(object);
            std::cout << "Unit: " << std::endl
                      << "HP: " << unit->getHealth() << std::endl
                      << "Armor: " << unit->getArmor() << std::endl
                      << "Weapon: " << unit->getWeapon() << std::endl
                      << "Unit class: ";
            switch(unit->getUnitType()){
                case UnitType::ARCHER:
                    std::cout << "Archer" << std::endl;
                    break;
                case UnitType::DRUID:
                    std::cout << "Druid" << std::endl;
                    break;
                case UnitType::INFANTRY:
                    std::cout << "Infantry" << std::endl;
                    break;
            }

            Log::log << "Command show unit" << Log::logend;

        } else{
            Log::log << "Empty cell" << Log::logend;
        }

    }

};

class ShowUnitCommandHandler: public CommandHandler{
```

```
public:

   bool isHandle(std::vector<std::string> &terminal) override{

      return terminal.size() == 3 && terminal[0] == "unit";

   }

       virtual  std::unique_ptr<Command>  handle(std::vector<std::string>
&terminal){

      if (isHandle(terminal)){
         int x = convertStr(terminal[1]);
         int y = convertStr(terminal[2]);
         Point unitPosition(x, y);
                                    return    std::unique_ptr<Command>(new
ShowUnitCommand(unitPosition));
      }

      if (next)
         return next->handle(terminal);

      return std::make_unique<Command>();
   }

};



#endif //BATTLEFORHONOUR_SHOWUNITCOMMAND_H

#ifndef BATTLEFORHONOUR_SKIPCOMMAND_H
#define BATTLEFORHONOUR_SKIPCOMMAND_H

#include "Command.h"

class SkipCommand: public Command{
```

```
    void execute(GameState &gameState) override{
            Log::log << "[#User] " << gameState.getNowPlayerIndex() << "
skiped turn" << Log::logend;
    }

};

class SkipCommandHandler: public CommandHandler {

public:

    bool isHandle(std::vector<std::string> &terminal) override{
        return terminal.size() == 1 && terminal[0] == "skip";
    }

     std::unique_ptr<Command> handle(std::vector<std::string> &terminal)
override{

        if (isHandle(terminal)){
            terminal.erase(terminal.begin());
            return std::unique_ptr<Command>(new SkipCommand());
        }

        if (next)
            return next->handle(terminal);

        return std::make_unique<Command>();
    }

};

#endif //BATTLEFORHONOUR_SKIPCOMMAND_H

#ifndef BATTLEFORHONOUR_SMALLGAME_H
#define BATTLEFORHONOUR_SMALLGAME_H
```

```
#include "GameRule.h"
#include "PlayerState.h"

class SmallGame: public GameRule {

private:

   PlayerState* nowState;

public:

   SmallGame():
      GameRule( 7, 7),
      nowState(new FirstPlayer){}

   bool isOver(GameState &gameState) override {
      int liveCount = gameState.getBases().size();
      for (auto b: gameState.getBases()){
         if (b && b->getHealth() <= 0){
            liveCount--;
         }
      }

      return liveCount <= 1;
   }

   int nextUser(GameState &gameState) override {

         int nowPlayerIndex = (gameState.getNowPlayerIndex() + nowState-
>getNextPlayerRecr()) % gameState.getBases().size();
      auto nextState = nowState->getNextPlayerState();
      delete nowState;
      nowState = nextState;
      if (nowState == nullptr)
         nowState = new FirstPlayer;
      return nowPlayerIndex;
```

```
    }
};

#endif //BATTLEFORHONOUR_SMALLGAME_H

#ifndef BATTLEFORHONOUR_STACKEXCEPTIONS_H
#define BATTLEFORHONOUR_STACKEXCEPTIONS_H


#include <exception>


class DoubleBasePlacingExc: std::exception {

public:

    int playerIndex;
                        explicit DoubleBasePlacingExc(int playerIndex):
playerIndex(playerIndex){}

};

class OutOfRangeExc: std::exception {

public:
    int x;
    int y;
    OutOfRangeExc(int x, int y): x(x), y(y){}
};

class DoublePlacingExc: std::exception {

};

class ImpossibleMoveExc: std::exception {

};
```

```
class InvalidFileLoadExc: std::exception {

};




#endif //BATTLEFORHONOUR_STACKEXCEPTIONS_H

#ifndef BATTLEFORHONOUR_SPEARMAN_H
#define BATTLEFORHONOUR_SPEARMAN_H


#include "Infantry.h"
#include "../../../Weapon/Weapon.h"

class SpearMan: public Infantry{

public:
   SpearMan():
      Infantry(*WeaponFlyweight::getFlyWeight<Spear>(), 50){}
};


#endif //BATTLEFORHONOUR_SPEARMAN_H

#ifndef BATTLEFORHONOUR_SWORDMAN_H
#define BATTLEFORHONOUR_SWORDMAN_H


#include "Infantry.h"
#include "../../../Weapon/Weapon.h"

class SwordMan: public Infantry{

public:
```

```
    SwordMan():
        Infantry(*WeaponFlyweight::getFlyWeight<Sword>(), 100){}
};


#endif //BATTLEFORHONOUR_SWORDMAN_H

#ifndef BATTLEFORHONOUR_TERRAIN_H
#define BATTLEFORHONOUR_TERRAIN_H


#include "../Weapon/Weapon.h"
#include "../Armor/Armor.h"
#include "../Objects/GameObject.h"

class Terrain {

public:

     virtual void print(std::ostream &stream, GameObject &object) const =
0;
    virtual void print(std::ostream &stream) const = 0;

    virtual int getDamageMultiply(WeaponType type) = 0;
    virtual int getAbsorbMultiply(ArmorType type) = 0;

     friend std::ostream& operator<<(std::ostream &stream, const Terrain
&terrain){
        terrain.print(stream);
        return stream;
    }

};

class Wasteland: public Terrain {

public:
```

```
void print(std::ostream &stream, GameObject &object) const override{

   stream << "[" << object << "]";

}

void print(std::ostream &stream) const override{

   stream << "[" << "#" << "]";

}

int getDamageMultiply(WeaponType type) override {
   switch (type){

      case WeaponType::PHYSIC:
         return 0;
      case WeaponType::DISTANCE:
         return 1;
      case WeaponType::MAGIC:
         return 100;
   }
}

int getAbsorbMultiply(ArmorType type) override {
   switch (type){
      case ArmorType::MAGIC:
         return 100;
      case ArmorType::HEAVY:
         return 0;
      case ArmorType::LIGHT:
         return 0;
      case ArmorType::MEDIUM:
         return 1;
   }
}
```

```
};

class Swamp: public Terrain {

public:

   void print(std::ostream &stream, GameObject &object) const override{

      stream << "<" << object << ">";

   }

   void print(std::ostream &stream) const override{

      stream << "<" << "#" << ">";

   }

   int getDamageMultiply(WeaponType type) override {
      switch (type){

         case WeaponType::PHYSIC :
            return 1;
         case WeaponType::MAGIC:
            return 2;
         case WeaponType::DISTANCE:
            return 3;
      }
   }

   int getAbsorbMultiply(ArmorType type) override {
      switch (type){

         case ArmorType::MAGIC:
            return 10;
         case ArmorType::HEAVY:
```

```
            return 1;
        case ArmorType::LIGHT:
            return 5;
        case ArmorType::MEDIUM:
            return 2;
    }
  }

};

class Desert: public Terrain {

public:

  void print(std::ostream &stream, GameObject &object) const override{

    stream << "{" << object << "}";

  }

  void print(std::ostream &stream) const override{

    stream << "{" << "#" << "}";

  }

  int getDamageMultiply(WeaponType type) override {

    switch (type){
        case WeaponType::PHYSIC :
            return 1;
        case WeaponType::MAGIC:
            return 0;
        case WeaponType::DISTANCE:
            return 3;
    }
```

```
    }

    int getAbsorbMultiply(ArmorType type) override {
        switch (type){

            case ArmorType::MAGIC:
                return 1;
            case ArmorType::HEAVY:
                return 2;
            case ArmorType::LIGHT:
                return 3;
            case ArmorType::MEDIUM:
                return 4;
        }
    }

};

class Border: public Terrain {

public:
    void print(std::ostream &stream) const override {
        stream << "+";
    }
};

#endif //BATTLEFORHONOUR_TERRAIN_H

#include "TerrainProxy.h"

TerrainProxy::TerrainProxy(Terrain *terrain):
    terrain(terrain) {}

int TerrainProxy::getAbsorbMultiply(ArmorType type) {
    if (terrain != nullptr) {
        return terrain->getAbsorbMultiply(type);
    } else{
```

```
        return 1;
    }
}

int TerrainProxy::getDamageMultiply(WeaponType type) {
    if (terrain != nullptr) {
        return terrain->getDamageMultiply(type);
    } else{
        return 1;
    }
}


#ifndef BATTLEFORHONOUR_TERRAINPROXY_H
#define BATTLEFORHONOUR_TERRAINPROXY_H


#include "Terrain.h"
#include "../GameField/Point.h"

class TerrainProxy {

private:
    Terrain *terrain;
public:
    explicit TerrainProxy(Terrain *terrain);
    int getDamageMultiply(WeaponType type);
    int getAbsorbMultiply(ArmorType type);
};


#endif //BATTLEFORHONOUR_TERRAINPROXY_H

#include "Unit.h"

Unit::Unit(const Unit &other):
        GameObject(ObjectType::UNIT),
```

```
    armor(other.armor),
    weapon(other.weapon),
    health(other.health) {}

void Unit::addObserver(UnitObserver *observer) {

  Log::log << "[#Unit] observer added" << Log::logend;
  observers.push_back(observer);

}

void Unit::move(Point point) {

  for (auto elem: observers){
    elem->onUnitMove(this, point);
  }
  Log::log << "[#Unit] moves" << Log::logend;

}

void Unit::attack(Unit &other) {

  for (auto elem: observers){
    elem->onUnitAttack(this, &other);
  }
  Log::log << "[#Unit] attacks" << Log::logend;

}

void Unit::damage(int damage) {

  for (auto elem: observers) {
    elem->onUnitDamaged(this);
  }

  if (damage < 0)
    damage = 0;
```

```
    health -= damage;

    if (health <= 0){
        for (auto elem: observers) {
            elem->onUnitDestroy(this);
        }
    }
        Log::log << "[#Unit] damaged by " << damage << " points" <<
Log::logend;
}

void Unit::heal(int hp) {
    for (auto elem: observers) {
        elem->onUnitHeal(this);
    }
    health += hp;
    Log::log << "[#Unit] healed by " << hp << " points \n" << Log::logend;
}

Unit &Unit::operator=(const Unit &unit) {

    armor = unit.armor;
    weapon = unit.weapon;
    health = unit.health;
    return *this;
}

Unit &Unit::operator<<(NeutralObject *neutralObject) {
    neutralObject->toEffect(*this);
    return *this;
}

Unit::Unit(UnitType unitType, Armor &armor, Weapon &weapon, int
health):
        GameObject(ObjectType::UNIT),
        unitType(unitType),
        armor(armor),
```

```
        weapon(weapon),
        health(health)
{}

int Unit::getHealth() const {
    return health;
}

void Unit::print(std::ostream &stream) const {
    switch(unitType){
        case UnitType::DRUID:
            stream << "D";
            break;
        case UnitType::ARCHER:
            stream << "A";
            break;
        case UnitType::INFANTRY:
            stream << "I";
            break;
    }
}
#ifndef BATTLEFORHONOUR_UNIT_H
#define BATTLEFORHONOUR_UNIT_H


#include <vector>
#include <ostream>
#include "../Armor/Armor.h"
#include "../Weapon/Weapon.h"
#include "../Observers/Observers.h"
#include "../GameField/Point.h"
#include "GameObject.h"
#include "../Terrains/TerrainProxy.h"
#include "Neutrals/NeutralObject.h"
#include "../Logs/Log.h"
#include "UnitType.h"
```

```
class Unit: public GameObject {

protected:

    UnitType unitType;
    int health;
    Armor &armor;
    Weapon &weapon;
    std::vector<UnitObserver*> observers;

    void print(std::ostream &stream) const override;

public:

    Unit(const Unit &other);
    Unit(UnitType unitType, Armor &armor, Weapon &weapon, int health);

    Weapon &getWeapon(){
        return weapon;
    }
    Armor &getArmor(){
        return armor;
    }

    void move(Point position);
    void attack(Unit &other);
    void heal(int hp);
    void damage(int damage);

    void addObserver(UnitObserver *observer);
    Unit& operator=(const Unit &unit);
    Unit& operator<<(NeutralObject *neutralObject);

    friend LogProxy& operator<<(LogProxy &logger, Unit &unit){
```

```
        logger << "Unit = x: " << unit.pos.x << " y: " << unit.pos.y << "
health: " << unit.health;
    return logger;

  }

  UnitType getUnitType(){
    return unitType;
  }

  int getHealth() const;

};


#endif //BATTLEFORHONOUR_UNIT_H

#ifndef BATTLEFORHONOUR_UNITTYPE_H
#define BATTLEFORHONOUR_UNITTYPE_H
enum class UnitType{
  INFANTRY,
  DRUID,
  ARCHER
};
#endif //BATTLEFORHONOUR_UNITTYPE_H

#ifndef BATTLEFORHONOUR_WEAPON_H
#define BATTLEFORHONOUR_WEAPON_H


#include <ostream>
#include "WeaponType.h"


class Weapon {

protected:
```

```
    WeaponType type;
    int damage;

public:

    int getDamage() const {
        return damage;
    }
    WeaponType getType() const {
        return type;
    }

    bool operator==(const Weapon &other){
        return type == other.type && damage == other.damage;
    }

    Weapon& operator=(const Weapon& other){
        if (this == &other) return *this;
        type = other.type;
        damage = other.damage;
        return *this;
    }

     friend std::ostream &operator<<(std::ostream &stream, const Weapon
&weapon){
        stream << "Weapon = " << "Damage: " << weapon.damage;
        return stream;
    }

};

class Sword: public Weapon{
public:
    Sword(){
        damage = 10;
        type = WeaponType::PHYSIC;
```

```
    }
};

class StarFall: public Weapon{
public:
    StarFall(){
        damage = 50;
        type = WeaponType::MAGIC;
    }
};

class Spear: public Weapon{
public:
    Spear(){
        damage = 20;
        type = WeaponType::PHYSIC;
    }
};

class Bow: public Weapon{
public:
    Bow(){
        damage = 10;
        type = WeaponType::DISTANCE;
    }
};

class AbolishMagic: public Weapon{

public:
    AbolishMagic(){
        damage = 20;
        type = WeaponType::MAGIC;
    }
};

#endif //BATTLEFORHONOUR_WEAPON_H
```

```
#ifndef BATTLEFORHONOUR_WEAPONFLYWEIGHT_H
#define BATTLEFORHONOUR_WEAPONFLYWEIGHT_H


#include <vector>
#include "Weapon.h"

class WeaponFlyweight {

private:

   static WeaponFlyweight *self;
   std::vector<Weapon*> weapons;

public:

   template <typename Type>
   static Type* getFlyWeight(){

      if (!self)
         self = new WeaponFlyweight();

      Type setWeapon;
      for (auto *weapon: self->weapons){

         if (setWeapon == *weapon){
            return static_cast<Type*>(weapon);
         }
      }

      Type *tmp = new Type();
      self->weapons.push_back(tmp);
      return tmp;
   }
};
```

```
WeaponFlyweight *WeaponFlyweight::self = nullptr;


#endif //BATTLEFORHONOUR_WEAPONFLYWEIGHT_H

#ifndef BATTLEFORHONOUR_WEAPONTYPE_H
#define BATTLEFORHONOUR_WEAPONTYPE_H

enum class WeaponType{
   MAGIC,
   PHYSIC,
   DISTANCE
};

#endif //BATTLEFORHONOUR_WEAPONTYPE_H
```