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

ОТЧЁТ

по лабораторной работе №2

по дисциплине «Объектно-ориентированное программирование»

Тема: Интерфейсы взаимодействие классов; перегрузка операций

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы

Научиться строить взаимодействие различных классов в программе.

Задание

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

Ход выполнения работы

- Создан механизм событий: класс события – `Event`, класс издателей событий – `EventEmitter`, класс подписчиков – `EventListener`. Также создан класс для объектов, которые „пробрасывают“ события, которые получают своим подписчикам – `EventForwarder`.

- Класс базы – `Base`. База содержит набор юнитов. У юнитов базы имеется числовой идентификатор. База управляет добавлением в неё юнитов, создаёт юниты по названию типа, а также генерирует свои события и пробрасывает события юнитов.

- База отслеживает уничтожение юнита, обрабатывая соответствующие типы событий.

- Базовый класс ландшафта – `Landscape`. Конкретные классы ландшафтов находятся в пространстве имён `landscapes`.

- Объекты ландшафта обрабатывают вход юнита на свою ячейку, а также выход юнита с этой ячейки. Они модифицируют возможности и параметры юнитов манипулируя их стратегиями. Например, ландшафт „Бо-

лото“ (`landscapes::Swamp`) запрещает атаку (временно заменяет стратегию атаки на экзмепляр класса `AttackForbidden`), а также ограничивает скорость перемещения до 1 ячейки за ход (декорирует текущую стратегию перемещения классом `ModifyingMovePolicy`).

- Базовый класс нейтрального объекта – `NeutralObject`. Подклассы нейтральных объектов расположены в пространстве имён `objects`.

- Нейтральные объекты обрабатывают их использование (и могут запретить юниту их использование), а также выход юнита с ячейки с объектом (на случай если объект при использовании изменяет характеристики юнита на время нахождения на его ячейке). Некоторые нейтральные объекты действуют так же как ландшафты – изменяют стратегии юнита.

- Класс `objects::WeaponSmiths` принимает стратегию, которая определяет, какие юниты могут его использовать, а какие нет. Таким образом, можно создать такой объект, работающий, например, только юнитами ближнего боя, или только с катапультами. Для этого придётся сделать несколько классов стратегий, но класс самих объектов будет одним и тем же.

Выводы

В ходе выполнения лабораторной работы была изучена организация взаимодействия классов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: event.hpp

```
#ifndef _H_EVENT_HPP
#define _H_EVENT_HPP

#include <set>

class Unit;
class EventListener;
class Event;

class EventEmitter {
    std::set<EventListener *> _listeners {};

public:
    void emit_shared(Event *e);
    void emit(Event *e);

    void subscribe(EventListener *l);
    void unsubscribe(EventListener *l);

    virtual ~EventEmitter() {}
};

class Event {
public:
    virtual ~Event() {}
};




class UnitEvent: public Event {
    Unit *_u;

public:
    UnitEvent(Unit *u) :_u{u} {}

    Unit *unit() const { return _u; }
};

class UnitDeathEvent: public UnitEvent {
public:
    using UnitEvent::UnitEvent;
};

class UnitAddedEvent: public UnitEvent {
```

```
public:
    using UnitEvent::UnitEvent;
};

class UnitLiveDeletedEvent: public UnitEvent {
public:
    using UnitEvent::UnitEvent;
};

class UnitTakesDamageEvent: public Event {
    Unit *_u;
    int _dmg;

public:
    UnitTakesDamageEvent(Unit *u, int dmg)
        :_u{u}, _dmg{dmg} {}

    Unit *unit() const { return _u; }
    int damage() const { return _dmg; }
};

class UnitGetsHealedEvent: public Event {
    Unit *_u;
    int _hp;

public:
    UnitGetsHealedEvent(Unit *u, int hp)
        :_u{u}, _hp{hp} {}

    Unit *unit() const { return _u; }
    int health() const { return _hp; }
};

class AttackEvent: public Event {
    Unit *_a, *_b;

public:
    AttackEvent(Unit *a, Unit *b) :_a{a}, _b{b} {}

    Unit *attacker() const { return _a; }
    Unit *target() const { return _b; }
};

class UnitWasAttackedEvent: public AttackEvent {
public:
    using AttackEvent::AttackEvent;
};

class UnitAttackedEvent: public AttackEvent {
public:
    using AttackEvent::AttackEvent;
};
```

```
class EventListener {
public:
    virtual void handle(Event *e) =0;

    virtual ~EventListener() {}
};

class EventForwarder: public EventEmitter,
                      public EventListener {
public:
    virtual void
    handle(Event *e) override
    {
        emit_shared(e);
    }
};

#endif
```

Название файла: event.cpp

```
#include "event.hpp"

void
EventEmitter::emit_shared(Event *e)
{
    for (auto iter = _listeners.begin(); iter != _listeners.end();) {
        auto *listener = *iter++;
        // note: the listener may safely unsubscribe when handling the
        // event.
        listener->handle(e);
    }
}

void
EventEmitter::emit(Event *e)
{
    emit_shared(e);
    delete e;
}

void
EventEmitter::subscribe(EventListener *l)
{
    _listeners.insert(l);
}

void
```

```
EventEmitter::unsubscribe(EventListener *l)
{
    _listeners.erase(l);
}
```

Название файла: base.hpp

```
#ifndef _H_BASE_HPP
#define _H_BASE_HPP

#include <map>
#include <string>
#include <vector>

#include "placeable.hpp"
#include "event.hpp"
#include "unit.hpp"


class UnitCreationStrategy {
public:
    virtual bool canCreate(const std::string &key) const =0;
    virtual std::vector<std::string> keys() const =0;
    virtual Unit *create(const std::string &key) =0;

    virtual ~UnitCreationStrategy() {}
};

class Base: public Placeable,
            public EventForwarder {

    std::map<int, Unit *> _units {};
    int _next_idx = 0;
    int _max_count = -1;

    UnitCreationStrategy *_cs;

public:
    Base(UnitCreationStrategy *cs)
        :_cs{cs} {}

    bool
    canCreateUnit(const std::string &key) const;
    Unit *
    createUnit(const std::string &key);

    int
    unitsCount() const { return (int)_units.size(); }
    bool
    setMaxUnitsCount(int m);
    int
    maxUnitsCount() const { return _max_count; }
```

```
    int
    addUnit(Unit *u);
    void
    removeUnit(Unit *u);
    Unit *
    getUnitById(int id) const;

    class unitsIter {
        using real_iter_t = std::map<int, Unit *>::const_iterator;
        real_iter_t _iter;

    public:
        unitsIter(real_iter_t it)
            :_iter{it} {}

        int id() const { return _iter->first; }
        Unit *unit() const { return _iter->second; }
        unitsIter &operator++() { ++_iter; return *this; }
        unitsIter operator++(int) { unitsIter x{_iter}; ++x; return x; }
        bool
        operator==(const unitsIter &o) const
        {
            return _iter == o._iter;
        }
        bool
        operator!=(const unitsIter &o) const
        {
            return !(*this == o);
        }
    };

    unitsIter
    unitsBegin() const { return unitsIter{_units.begin()}; }
    unitsIter
    unitsEnd() const { return unitsIter{_units.end()}; }

    virtual void
    handle(Event *e) override;

    virtual ~Base() override;
};

#endif
```

Название файла: base.cpp

```
#include <map>
#include <string>
#include <vector>

#include "unit.hpp"
```

```
#include "unit_factory.hpp"
#include "event.hpp"
#include "base.hpp"


bool
Base::canCreateUnit(const std::string &key) const
{
    return _cs->canCreate(key);
}

Unit *
Base::createUnit(const std::string &key)
{
    if (unitsCount() == maxUnitsCount()) {
        return nullptr;
    }

    if (!_cs->canCreate(key)) {
        return nullptr;
    }

    Unit *u = _cs->create(key);

    if (addUnit(u) < 0) {
        delete u;
        return nullptr;
    }

    return u;
}

bool
Base::setMaxUnitsCount(int m)
{
    if (m < unitsCount()) {
        return false;
    }
    _max_count = m;
    return true;
}

int
Base::addUnit(Unit *u)
{
    if (maxUnitsCount() >= 0
        && unitsCount() == maxUnitsCount()) {
        return -1;
    }

    _units[_next_idx] = u;
    u->subscribe(this);
```

```
    // u->emit(new UnitAddedEvent {u});

    return _next_idx++;
}

void
Base::removeUnit(Unit *u)
{
    u->unsubscribe(this);

    for (auto iter = _units.begin();
         iter != _units.end();
         ++iter) {
        if (iter->second == u) {
            _units.erase(iter);
            break;
        }
    }
}

Unit *
Base::getUnitById(int id) const
{
    auto iter = _units.find(id);
    return (iter != _units.end())
        ? iter->second
        : nullptr;
}

void
Base::handle(Event *e)
{
    EventForwarder::handle(e);

    if (auto *ee = dynamic_cast<UnitDeathEvent *>(e)) {
        removeUnit(ee->unit());
    } else if (auto *ee = dynamic_cast<UnitLiveDeletedEvent *>(e)) {
        removeUnit(ee->unit());
    }
}

Base::~Base()
{
    for (auto p: _units) {
        p.second->unsubscribe(this);
    }
    delete _cs;
}
```

Название файла: landscape.hpp

```
#ifndef _H_LANDSCAPE_HPP
```

```
#define _H_LANDSCAPE_HPP


class Unit;

class Landscape {
public:
    virtual void onEnter(Unit *u) =0;
    virtual void onLeave(Unit *u) =0;

    virtual ~Landscape() {}
};

namespace landscapes {

    class Normal: public Landscape {
    public:
        virtual void onEnter(Unit *) override {}
        virtual void onLeave(Unit *) override {}
    };

}

#endif
```

Название файла: landscape.cpp

Название файла: landscape_types.hpp

```
#ifndef _H_LANDSCAPE_TYPES_HPP
#define _H_LANDSCAPE_TYPES_HPP

#include "landscape.hpp"
#include "unit.hpp"
#include "map.hpp"
#include "common_policies.hpp"


namespace landscapes {

    // Swamp: max speed is 1; attacking is forbidden
    class Swamp: public Landscape {
        ModifyingMovePolicy *_p;
        AttackPolicy *_prev, *_cur;

    public:
        virtual void onEnter(Unit *u) override
        {
            _p = new ModifyingMovePolicy {u->movePolicy(), 1};
            u->setMovePolicy(_p);

            _prev = u->attackPolicy();
```

```
            _cur = new AttackForbidden {};
            u->setAttackPolicy(_cur);
        }

        virtual void onLeave(Unit *u) override
        {
            if (auto *mpc = u->findMoveContainerOf(_p)) {
                mpc->setMovePolicy(_p->movePolicy());
                _p->setMovePolicy(nullptr);
                delete _p;
                _p = nullptr;
            }

            // our policy might've been wrapped into something
            if (auto *apc = u->findAttackContainerOf(_cur)) {
                apc->setAttackPolicy(_prev);
                delete _cur;
                _cur = nullptr;
            }
        }
    };

    class Forest: public Landscape {
        DefensePolicy *_prev;
        MultiplierDefensePolicy *_cur;

    public:
        virtual void onEnter(Unit *u) override
        {
            _prev = u->defensePolicy();
            _cur = new MultiplierDefensePolicy {_prev, 2.0};
            u->setDefensePolicy(_cur);
        }

        virtual void onLeave(Unit *u) override
        {
            if (auto *dpc = u->findDefenseContainerOf(_cur)) {
                dpc->setDefensePolicy(_prev);
                _cur->setDefensePolicy(nullptr);
                delete _cur;
                _cur = nullptr;
            }
        }
    };
}

#endif
```

Название файла: neutral_object.hpp

```
#ifndef _H_NEUTRAL_OBJECT_HPP
#define _H_NEUTRAL_OBJECT_HPP
```

```
#include "placeable.hpp"
#include "unit.hpp"
#include "map.hpp"


class NeutralObject: public Placeable {
public:
    virtual bool canUse(const Unit *, MapIter) { return true; }
    virtual void onUse(Unit *u, MapIter at) =0;

    // It's the object's job to determine whether it was used by the
    // leaving unit.
    virtual void onLeave(Unit *) {};

    virtual ~NeutralObject() {};
};

#endif
```

Название файла: neutral_object.cpp

Название файла: neutral_object_types.hpp

```
#ifndef _H_NEUTRAL_OBJECT_TYPES_HPP
#define _H_NEUTRAL_OBJECT_TYPES_HPP

#include <random>

#include "neutral_object.hpp"
#include "map.hpp"
#include "unit.hpp"

#include "ranged_units.hpp"
#include "common_policies.hpp"


class ExtendedShootingRange: public NestedAttack {
    double _delta;

public:
    ExtendedShootingRange(AttackPolicy *p, double delta)
        :NestedAttack{p}, _delta{delta} {}

    virtual bool
    canAttackTo(const Unit *u, MapIter to) override
    {
        double dist = to.point().distance(u->position());
        auto *a = dynamic_cast<RangedAttack *>(attackPolicy());
        return dist >= a->minRange()
            && dist <= (a->maxRange() + _delta);
    }
```

```
    virtual MapIter
    actualPosition(const Unit *u, MapIter to) override
    {
        return attackPolicy()->actualPosition(u, to);
    }

    virtual std::pair<AttackKind, int>
    baseAttack(const Unit *u, MapIter to) override
    {
        return attackPolicy()->baseAttack(u, to);
    }
};




namespace objects {

    class HealingWell: public NeutralObject {
    public:
        virtual void
        onUse(Unit *u, MapIter) override
        {
            u->heal(25);
        }
    };

    class Tower: public NeutralObject {
        AttackPolicy *_prev;
        ExtendedShootingRange *_cur = nullptr;

    public:
        virtual bool
        canUse(const Unit *u, MapIter) override
        {
            return dynamic_cast<const BasicRangedUnit *>(u);
        }

        virtual void
        onUse(Unit *u, MapIter) override
        {
            _prev = u->attackPolicy();
            _cur = new ExtendedShootingRange {_prev, 5};
            u->setAttackPolicy(_cur);
        }

        virtual void
        onLeave(Unit *u) override
        {
            if (_cur == nullptr) {
                return;
```

```
        }
        if (auto *apc = u->findAttackContainerOf(_cur)) {
            apc->setAttackPolicy(_prev);
            _cur->setAttackPolicy(nullptr);
            delete _cur;
            _cur = nullptr;
        }
    }
};

class TunnelsEntrance: public NeutralObject {
public:
    virtual void
    onUse(Unit *, MapIter at) override
    {
        static const int w = 5;

        int max_n = 0;
        for (int j = -w; j <= w; ++j) {
            for (int i = -w; i <= w; ++i) {
                auto iter = at.shifted({i, j});
                if (iter.unit() == nullptr) {
                    ++max_n;
                }
            }
        }

        std::uniform_int_distribution<> distr {0, max_n-1};
        int n = distr(global_random);

        MapIter dest = MapIter::makeNull();
        for (int j = -w; j <= w; ++j) {
            for (int i = -w; i <= w; ++i) {
                auto iter = at.shifted({i, j});
                if (iter.unit() != nullptr) {
                    continue;
                }
                if (!--n) {
                    dest = iter;
                    break;
                }
            }
        }

        at.moveUnitTo(dest);
    }
};

class WeaponSmiths: public NeutralObject {
public:
    class UnitFilter {
    public:
```

```
            virtual bool
            applicable(const Unit *u) =0;
        };

        template<typename U>
        class SimpleUnitFilter: public UnitFilter {
        public:
            virtual bool
            applicable(const Unit *u) override
            {
                return dynamic_cast<const U *>(u);
            }
        };

    private:
        double _mul;
        UnitFilter *_filter;

    public:
        explicit WeaponSmiths(double mul, UnitFilter *filter=nullptr)
            :_mul{mul}, _filter{filter} {}

        virtual bool
        canUse(const Unit *u, MapIter) override
        {
            if (_filter
                && !_filter->applicable(u)) {
                return false;
            }

            for (const AttackPolicyContainer *apc = u; apc;
                 apc = dynamic_cast<AttackPolicyContainer *>(
                     apc->attackPolicy())) {
                if (dynamic_cast<const MultiplierAttackPolicy *>(apc)) {
                    return false;
                }
            }

            return true;
        }

        virtual void
        onUse(Unit *u, MapIter) override
        {
            auto *prev = u->attackPolicy();
            auto *new_p = new MultiplierAttackPolicy {prev, _mul};
            u->setAttackPolicy(new_p);
        }
    };

}
```

```
#endif
```