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

ОТЧЁТ

по лабораторной работе №6

по дисциплине «Объектно-ориентированное программирование»

Тема: Шаблонные классы

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы

Научиться работать с шаблонными классами в C++.

Задание

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

Ход выполнения работы

- Был добавлен класс `GameDriver`, который в качестве шаблонного параметра принимает класс объекта правил игры.
- Класс `GameDriver` использует правила, чтобы создавать карту, настраивать новую игру (объект `Game`), а также проверять условие конца игры и определять победителя.
- `GameDriver` позволяет сбрасывать игру в процессе работы, а также может при этом принимать новую игру вместо создания её с помощью правил (это используется для загрузки сохранённой игры из файла).
- Созданы два класса правил игры – `DefaultRules`, и `FancyRules`. В классе `FancyRules` создаётся более крупная карта с лесом в середине, а также 4 игрока вместо двух.

Выводы

В ходе выполнения лабораторной работы были изучены шаблоны языка C++.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: game_driver.hpp

```
#ifndef _H_GAME_DRIVER_HPP
#define _H_GAME_DRIVER_HPP

#include "map.hpp"
#include "game.hpp"


class GameRules {
public:
    virtual Map *makeMap() =0;
    virtual void setup(Game *g, Map *m) =0;
    virtual bool gameEnded(Game *g) =0;
    virtual int winner(Game *g) =0;
};

template<typename Rules>
class GameDriver: public ResetHandler {

    Game *_g = nullptr;
    Game *_g_reset = nullptr;   // required to reset game while running
    Rules *_r {new Rules {}};

    std::vector<EventPrinter *> _loggers {};
    EventPrinter *_printer = nullptr;

    Game *init()
    {
        Map *m = _r->makeMap();
        Game *g = new Game {m};
        _r->setup(g, m);
        return g;
    }

    void setBasePrefixes(EventPrinter *p)
    {
        int n = _g->basesCount();
        for (int i = 0; i < n; ++i) {
            std::ostringstream oss {};
            oss << "Base " << (i+1);
            p->setPrefix(_g->baseByIdx(i), oss.str());
        }
    }

public:
    void
```

```
addLogger(EventPrinter *l)
{
    _loggers.push_back(l);
    if (_g) {
        _g->logSink()->subscribe(l);
        setBasePrefixes(l);
    }
}

void
setPrinter(EventPrinter *pr)
{
    if (_printer) {
        if (_g) {
            _g->unsubscribe(_printer);
        }
        delete _printer;
    }

    _printer = pr;
    if (_g && _printer) {
        _g->subscribe(_printer);
        setBasePrefixes(_printer);
    }
}

virtual void reset() override
{
    resetFrom(init());
}

void resetFrom(Game *g)
{
    _g_reset = g;
}

void run()
{
    for (;;) {
        if (_g_reset) {
            delete _g;
            _g = _g_reset;
            _g_reset = nullptr;
            _g->setResetHandler(this);

            if (_printer) {
                _g->subscribe(_printer);
                setBasePrefixes(_printer);
            }
            for (auto *l: _loggers) {
                _g->logSink()->subscribe(l);
                setBasePrefixes(l);
```

```
                }
            }
            if (_r->gameEnded(_g)) {
                break;
            }
            _g->spin();
        }
    }

    int winner()
    {
        return _r->winner(_g);
    }

    virtual ~GameDriver() override
    {
        delete _g;
        delete _g_reset;
        delete _printer;
        for (auto *l: _loggers) {
            delete l;
        }
    }
};

#endif
```

Название файла: game_rules.hpp

```
#ifndef _H_GAME_RULES_HPP
#define _H_GAME_RULES_HPP

#include <iostream>
#include <utility>

#include "map.hpp"
#include "game.hpp"
#include "iostream_player.hpp"
#include "game_driver.hpp"
#include "landscape_types.hpp"


class BaseWithSpawnCountdown: public Base {
    int _cd_max;
    int _cd = 0;

public:
    BaseWithSpawnCountdown(int cd_max=0)
        :_cd_max{cd_max} {}

    virtual bool
    canCreateUnit(const std::string &key) const override
```

```
    {
        return _cd == 0
            && Base::canCreateUnit(key);
    }

    virtual int
    createUnit(const std::string &key, Mediator *m) override
    {
        int id = Base::createUnit(key, m);
        if (id < 0) {
            return id;
        }

        _cd = _cd_max;
        return id;
    }

    virtual void
    store(std::ostream &os) const override
    {
        os << "base_w_countdown " << maxUnitsCount()
           << " " << destroyed() << " " << _cd_max << " "
           << _cd << "\n";
    }

    virtual bool
    restore(std::istream &is, RestorerTable *tab) override
    {
        if (!Base::restore(is, tab)) {
            return false;
        }

        is >> _cd_max >> _cd;
        return !is.fail();
    }

    virtual void
    spin() override
    {
        if (_cd > 0) {
            --_cd;
        }
    }
};

class DefaultRules: public GameRules {
protected:
    static void
    addBaseAndPlayer(Game *g, Map *m,
                     std::string name, int x, int y)
    {
        Base *b = new BaseWithSpawnCountdown {5};
```

```
        m->addBase(b, {x, y});
        int id = g->addBase(b);

        auto *p = new IostreamPlayer {std::move(name)};
        p->setOstream(std::cout);
        p->setIstream(std::cin);
        g->setPlayer(id, p);
    }

public:
    virtual Map *makeMap() override
    {
        return new Map {10, 10};
    }

    virtual void setup(Game *g, Map *m) override
    {
        addBaseAndPlayer(g, m, "Player 1", 1, 1);
        addBaseAndPlayer(g, m, "Player 2", 8, 8);
    }

    virtual bool gameEnded(Game *g) override
    {
        return g->playersCount() < 2;
    }

    virtual int winner(Game *g) override
    {
        int n = g->basesCount();
        int only = -1;

        for (int i = 0; i < n; ++i) {
            if (!g->baseByIdx(i)->destroyed()) {
                if (only < 0) {
                    only = i;
                } else {
                    return -1;
                }
            }
        }

        return only;
    }
};

class FancyRules: public DefaultRules {
    virtual Map *makeMap() override
    {
        Map *m = new Map {15, 15};

        for (int y = 0; y < 5; ++y) {
            for (int x = 0; x < 5; ++x) {
```

```
                m->setLandscape(new landscapes::Forest {},
                                {5+x, 5+y});
            }
        }

        return m;
    }

    virtual void setup(Game *g, Map *m) override
    {
        addBaseAndPlayer(g, m, "Player 1", 3, 3);
        addBaseAndPlayer(g, m, "Player 2", 3, 11);
        addBaseAndPlayer(g, m, "Player 3", 11, 11);
        addBaseAndPlayer(g, m, "Player 4", 11, 3);
    }
};

#endif
```

Название файла: main.cpp

```
#include <string.h>

#include <iostream>
#include <fstream>

#include "demo.hpp"

#include "event_printer.hpp"
#include "game_driver.hpp"
#include "game_rules.hpp"

void
run_demos(void)
{
    std::cout << "Demo 1\n";
    demo1();

    std::cout << "\nDemo 2\n";
    demo2();

    std::cout << "\nDemo 3\n";
    demo3();

    std::cout << "\nDemo 4\n";
    demo4();

    std::cout << "\nDemo 5\n";
    demo5();

    std::cout << "\nDemo 6\n";
    demo6();
```

```
    std::cout << "\nDemo 7\n";
    demo7();

    std::cout << "\nDemo 8\n";
    demo8();

    std::cout << "\nDemo 9\n";
    demo9();
}

int
run_game(int argc, char **argv)
{
    std::vector<EventPrinter *> loggers {};
    bool have_stdout = false;

    const char *load_fn = nullptr;

    for (int i = 1; i < argc; ++i) {
        if (!strcmp(argv[i], "-log")) {
            char *fn = argv[++i];
            if (!strcmp(fn, "-")) {
                loggers.push_back(new LoggingEventPrinter {std::cout});
                have_stdout = true;
            } else {
                auto *of = new std::ofstream {fn};
                if (!*of) {
                    std::cerr << "Failed to open file: " << fn << "\n";
                    return 1;
                }
                loggers.push_back(new LoggingEventPrinter {of});
            }
        } else if (!strcmp(argv[i], "-load")) {
            load_fn = argv[++i];
        } else {
            std::cerr << "Unknown option: " << argv[i] << "\n";
            return 1;
        }
    }

    GameDriver<DefaultRules> drv {};

    for (auto *logger: loggers) {
        drv.addLogger(logger);
    }

    if (!have_stdout) {
        drv.setPrinter(new EventPrinter {std::cout});
    }

    if (load_fn) {
```

```
        std::ifstream f {load_fn};
        if (!f) {
            std::cerr << "Failed to open save file: "
                      << load_fn << "\n";
            return 1;
        }
        auto *tab = RestorerTable::defaultTable();
        Storable *s = tab->restore(f);
        delete tab;

        if (auto *lg = dynamic_cast<Game *>(s)) {

            drv.resetFrom(lg);

        } else {
            std::cerr << "Invalid save file contents\n";
            delete s;
            return 1;
        }
    } else {
        drv.reset();
    }

    drv.run();

    return 0;
}

int
main(int argc, char **argv)
{
    if (argc == 2
        && !strcmp(argv[1], "-demo")) {
        run_demos();
        return 0;
    }

    return run_game(argc, argv);
}
```