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

ОТЧЁТ

по лабораторной работе №4

по дисциплине «Объектно-ориентированное программирование»

Тема: Полиморфизм

Студент гр. 8303 _____ Парфентьев Л.М.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы

Изучить полиморфизм и его применение.

Задание

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

- Логирование действий пользователя
- Логирование действий юнитов и базы

Ход выполнения работы

- Ранее был написан класс `EventPrinter`. Он является подписчиком, который выводит получаемые события на переданный ему поток вывода (`stsd::ostream`).
- Поток вывода может быть, например, стандартным потоком вывода (который по-умолчанию подсоединён к терминалу), или потоком вывода в файл, или `std::ostringstream` – потоком вывода в строку.
- Был добавлен класс событий, обозначающие действия пользователя – `events::UserActionEvent`.
- Класс логгера – `LoggingEventPrinter`. Он особенным образом обрабатывает действия пользователя, печатая его имя перед сообщением.
- Класс `EventPrinter` принимает поток вывода по ссылке или по указателю. Если ему был передан указатель, считается, что поток принадлежит ему, и он уничтожает его в своём деструкторе (при помощи `delete`).

Выводы

В ходе выполнения лабораторной работы был изучен полиморфизм на примере класса вывода логов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: event_printer.hpp

```
#ifndef _H_EVENT_PRINTER_HPP
#define _H_EVENT_PRINTER_HPP

#include <sstream>
#include <iostream>
#include <string>
#include <map>

#include "event.hpp"
#include "unit.hpp"
#include "base.hpp"
#include "player.hpp"
#include "object_print.hpp"


class EventPrinter: public EventListener {
    std::ostream *_os;
    bool _free_os;

    std::map<EventForwarder *, std::string> _prefix_map {};
    int _base_idx = 0;

    std::string
    makeName(const char *base, int idx)
    {
        std::ostringstream oss {};
        oss << base << " " << idx;
        return oss.str();
    }

public:
    EventPrinter(std::ostream &os)
        :_os{&os}, _free_os{false} {}

    EventPrinter(std::ostream *os)
        :_os{os}, _free_os{true} {}

    std::ostream &
    ostream() const { return *_os; }

    void
    setPrefix(EventForwarder *f, const std::string &s)
    {
        _prefix_map[f] = s;
    }
```

```
virtual void
handle(Event *e) override
{
    if (auto *ee = dynamic_cast<events::Forwarded *>(e)) {
        auto iter = _prefix_map.find(ee->forwarder());
        if (iter != _prefix_map.end()) {
            (*_os) << iter->second << ": ";
        }

        return handle(ee->event());

    } else if (auto *ee =
                dynamic_cast<events::UnitAdded *>(e)) {
        (*_os) << "Unit added: " << ee->unit() << "\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitDeath *>(e)) {
        (*_os) << "Unit died: " << ee->unit() << "\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitTakesDamage *>(e)) {
        (*_os) << "Unit " << ee->unit() << " takes "
                << ee->damage() << " health points of damage\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitGetsHealed *>(e)) {
        (*_os) << "Unit " << ee->unit() << " gets healed by "
                << ee->health() << " health points\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitAttacked *>(e)) {
        (*_os) << "Unit " << ee->attacker()
                << " attacked another unit " << ee->target() << "\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitWasAttacked *>(e)) {
        (*_os) << "Unit " << ee->target()
                << " was attacked by another unit "
                << ee->attacker() << "\n";
    } else if (auto *ee =
                dynamic_cast<events::UnitMoved *>(e)) {
        (*_os) << "Unit " << ee->unit()
                << " moved from " << ee->sourcePos() << "\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitUsedObject *>(e)) {
        (*_os) << "Unit " << ee->unit()
                << " used object " << ee->neutralObject() << "\n";

    } else if (auto *ee =
                dynamic_cast<events::UnitLiveDeleted *>(e)) {
```

```
            (*_os) << "(Live unit " << ((void *)ee->unit())
                   << " deleted)\n";

        } else if (auto *ee =
                   dynamic_cast<events::BaseAdded *>(e)) {
            auto name = makeName("Base", ++_base_idx);
            setPrefix(ee->base(), name);
            (*_os) << "New base: " << name << "\n";

        } else if (auto *ee =
                   dynamic_cast<events::TurnStarted *>(e)) {
            (*_os) << "Turn of player "
                   << ee->player()->name() << "\n";

        } else if (auto *ee =
                   dynamic_cast<events::TurnOver *>(e)) {
            (*_os) << "Turn of player "
                   << ee->player()->name() << " over\n";

        } else {
            (*_os) << "Unknown event\n";
        }
    }

    virtual ~EventPrinter() override
    {
        if (_free_os) {
            _os->flush();
            delete _os;
        }
    }
};

#endif
```

Название файла: logging.hpp

```
#ifndef _H_LOGGING_HPP
#define _H_LOGGING_HPP

#include <string>
#include <utility>

#include "event.hpp"
#include "event_printer.hpp"


namespace events {

    class UserActionEvent: public Event {
        Player *_p;
        std::string _s;
```

```
    public:
        UserActionEvent(Player *p, std::string s)
            :_p{p}, _s{std::move(s)} {}

        const std::string &
        message() const { return _s; }

        Player *player() const { return _p;}
    };

}

class LoggingEventPrinter: public EventPrinter {
public:
    using EventPrinter::EventPrinter;

    virtual void
    handle(Event *e) override
    {
        if (auto *ee = dynamic_cast<events::UserActionEvent *>(e)) {
            ostream() << ee->player()->name()
                      << ": " << ee->message() << "\n";
            return;
        }

        EventPrinter::handle(e);
    }
};

#endif
```

Название файла: main.cpp

```
#include <string.h>

#include <iostream>
#include <fstream>

#include "demo.hpp"

#include "player.hpp"
#include "iostream_player.hpp"
#include "event_printer.hpp"
#include "base.hpp"
#include "map.hpp"
#include "factory_table.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<LoggingEventPrinter *> loggers {};
    bool have_stdout = false;

    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 {
            std::cerr << "Unknown option: " << argv[i] << "\n";
            return 1;
        }
    }
```

```
    Map *map = new Map {10, 10};

    Game g {map};

    for (auto *logger: loggers) {
        g.logSink()->subscribe(logger);
    }

    EventPrinter *pr = nullptr;
    if (!have_stdout) {
        pr = new EventPrinter {std::cout};
        g.subscribe(pr);
    }

    Base *b1 = new Base {};
    map->addBase(b1, {1, 1});
    g.addBase(b1);

    Base *b2 = new Base {};
    map->addBase(b2, {8, 8});
    g.addBase(b2);

    auto *p1 = new IostreamPlayer {"Player 1"};
    p1->setOstream(std::cout);
    p1->setIstream(std::cin);
    g.setPlayer(0, p1);

    auto *p2 = new IostreamPlayer {"Player 2"};
    p2->setOstream(std::cout);
    p2->setIstream(std::cin);
    g.setPlayer(1, p2);

    while (g.playersCount())
        g.spin();

    for (auto *logger: loggers) {
        g.logSink()->unsubscribe(logger);
        delete logger;
    }

    if (pr) {
        g.unsubscribe(pr);
        delete pr;
    }

    return 0;
}

int
main(int argc, char **argv)
{
```

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

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