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

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Объектно-ориентированное программирование»
Тема: логирование, перегрузка операций

Студент гр. 0304 Крицын Д.Р.
Преподаватель Жангиров Т.Р.

Санкт-Петербург
2021

Цель работы.

Реализовать вывод сведений о течении процесса игровых событий проекта игры при помощи: перегрузки операторов ввода-вывода для игровых объектов, создания и использования классов для логирования\отслеживания объектов.

Задание.

Необходимо проводить логирование того, что происходит во время игры.

Требования:

- Реализован класс логгера, который будет получать объект, который необходимо отслеживать, и при изменении его состоянии записывать данную информацию.
- Должна быть возможность записывания логов в файл, в консоль или одновременно в файл и консоль.
- Должна быть возможность выбрать типа вывода логов
- Все объекты должны логироваться через перегруженный оператор вывода в поток.
- Должна соблюдаться идиома RAII

Выполнение работы.

1. Создание класса *global_logger*, который реализует глобальное логирование происходящих в игре событий, а именно действий, совершаемых игровыми сущностями. Данный класс имеет метод *static void init_loggers (const std::vector <std::reference_wrapper<ostream_wrapper>>& streams)*, принимающий на вход список ссылок на потоки ввода, и внутренне создающий несколько экземляров логгеров, которые выводят одну и ту же информацию одновременно в несколько

потоков вывода при вызове метода *static void message(const char*) / static void message(const std::string&).*

2. Создание класса *observable*, который пресдтавляет собой абстрактный класс-«интерфейс», который должны реализовывать объекты, чтобы выводить своё состояние при его изменении во время хода любой из взаимодействующих с ним сущностей. При добавлении экземпляра класса *observable* в объект класса *observer* вызывается метод *void setObserver(observer&)*, который задаёт ссылку на наблюдателя, который впоследствии уведомляется о событии при помощи метода *void notifyObserver()*.

3. Создание абстрактного класса *observer*, который хранит в себе обёртки одного или нескольких потоков вывода, и имеющий абстрактный метод *void notify()*, вызываемый наблюдаемым объектом, когда состояние последнего изменяется каким-либо образом, который должен осуществлять вывод нового состояния объекта одновременно во все хранимые потоки вывода.

4. Реализация класса *observer* классом *observer_actor*, работающего для объектов-«актёров», имеющего метод *void setActor(actor&)*, задающий отслеживаемого актёра, и переопределённый метод *void notify(),* выводящий данные об актёре при изменении некоторых его характеристик через перегруженный оператор вывода "<<".

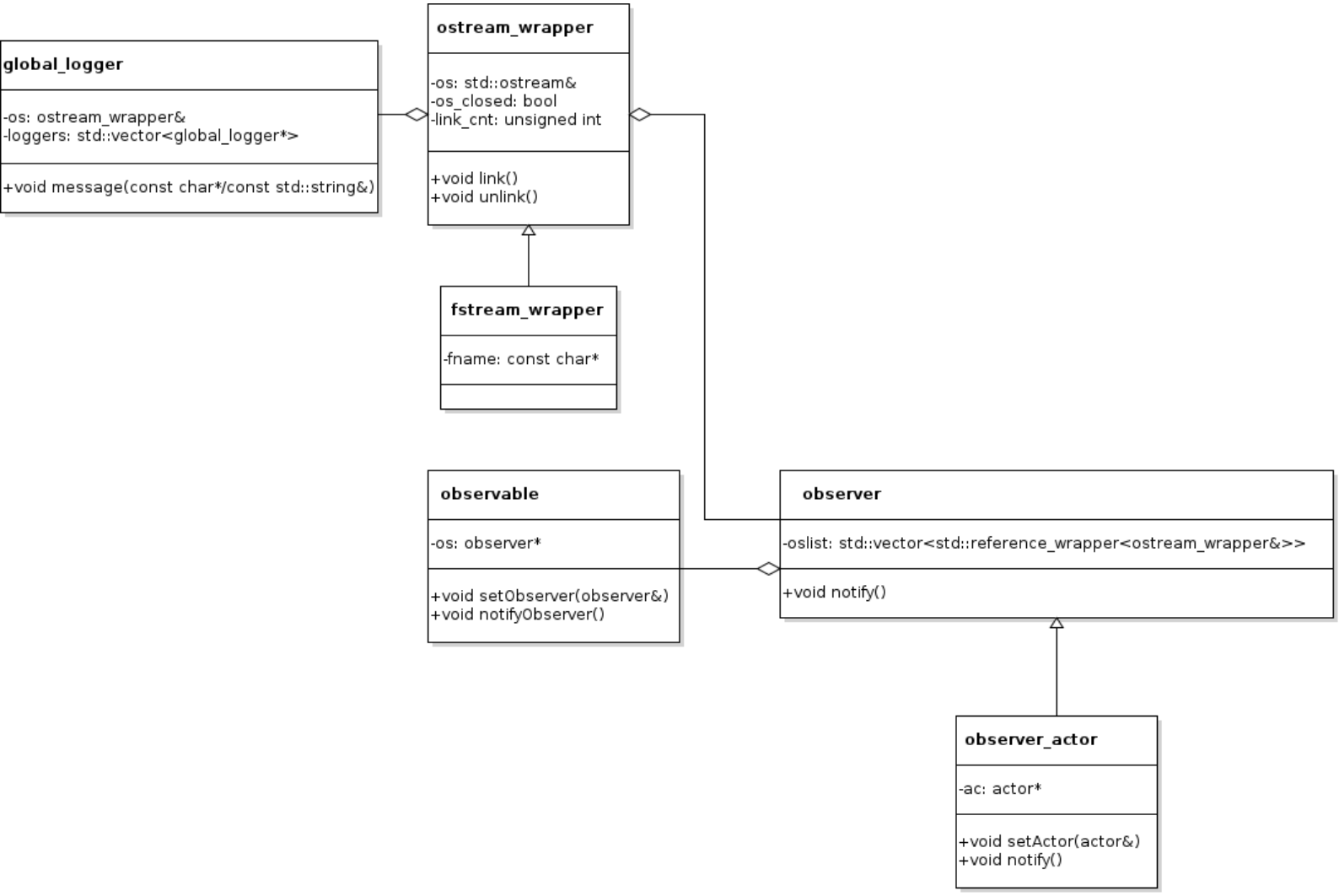
5. Создание класса *ostream_wrapper* для реализации идеологии RAII для классов логирования: он осуществляет подсчёт ссылок

при помощи методов *void link()* и *void unlink()*, которые вызываются объектами, использующими данную обёртку; закрывает\открывает файл по необходимости (удаление последней ссылки на поток вывода \ возникновение первой ссылки на поток). Для осуществления вывода используется геттер *std::ostream& get_stream()*. *fstream_wrapper* осуществляет те же функции, но для потока вывода в файл: данный класс переопределяет защищённые методы *ostream_wrapper void open()* и *void close()*, ответственные за открытие\закрытие файла (т.к. *std::basic_ostream* не имеет метода *close()*), при этом храня имя файла, передаваемое методу *open()* класса *std::ofstream*.

Выводы.

В ходе выполнения данной лабораторной работы был изучен паттерн проектирования «Наблюдатель», перегрузка операторов для организации вывода информации об объекте, использование статических полей для хранения инстанций класса и методов для инициализации и доступа к данном полю, на примере создания глобального логера.

ПРИЛОЖЕНИЕ А
UML-ДИАГРАММА КЛАССОВ



ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПРОГРАММЫ

global_logger.h

```
#ifndef GLOBAL_LOGGER_H
#define GLOBAL_LOGGER_H

#include <functional>
#include <iostream>
#include <vector>

class global_logger
{
 private:

 std::ostream& os;

 global_logger(std::ostream& os);

 static std::vector<global_logger*> loggers;


 public:

 static void init_loggers(const
    std::vector<std::reference_wrapper<std::ostream>>&
    streams);

 static void message(const char* msg);
 static void message(const std::string& msg);
};

#endif
```

global_logger.cpp

```
#include "global_logger.h"

std::vector<global_logger*> global_logger::loggers;

global_logger::global_logger(std::ostream& os) : os(os){}


void global_logger::init_loggers(const
    std::vector<std::reference_wrapper<std::ostream>>&
    streams)
{
 for(auto i = streams.begin(); i != streams.end(); ++i){
```

```
    std::ostream& os = (*i).get();
    loggers.push_back(new global_logger(os));
 }
}


void global_logger::message(const char* msg)
{
 for(auto i = loggers.begin(); i != loggers.end(); ++i)
    (*i)->os << msg << std::endl;
}
void global_logger::message(const std::string& msg)
{
 for(auto i = loggers.begin(); i != loggers.end(); ++i)
    (*i)->os << msg << std::endl;
}
```

observable.h

```
#ifndef OBSERVABLE_H
#define OBSERVABLE_H

#include "observer.h"

class observable
{
 private:

 observer* ob = nullptr;


 public:

 void setObserver(observer& ob);
 void notifyObserver();
};

#endif
```

observable.cpp

```
#include "observable.h"

void observable::setObserver(observer& ob){ this->ob =
    &ob; }
void observable::notifyObserver() { if(ob) ob->notify(); }
```

observer.h

```
#ifndef OBSERVER_H
#define OBSERVER_H
```

```
#include <functional>
#include <iostream>
#include <vector>

class observer
{
 protected:

 std::vector<std::reference_wrapper<std::ostream>> oslist;

 public:

 observer(std::ostream& os = std::cout);
 observer(const
    std::vector<std::reference_wrapper<std::ostream>>&
    streams);
 virtual void notify() = 0;
};

#endif
```

observer.cpp

```
#include "observer.h"

observer::observer(std::ostream& os)
{
 this->oslist.push_back(os);
}
observer::observer(const
    std::vector<std::reference_wrapper<std::ostream>>&
    streams)
{
 for(auto i = streams.begin(); i != streams.end(); ++i)
    oslist.push_back(*i);
}
```

observer_actor.h

```
#ifndef OBSERVER_ACTOR_H
#define OBSERVER_ACTOR_H

#include "observer.h"
#include "../actors/actor.h"

class observer_actor : public observer
{
 private:
```

```
 actor* ac;


 public:

 using observer::observer;

 virtual void setActor(actor& ac);
 virtual void notify();
};

#endif
```

observer_actor.cpp

```
#include "observer_actor.h"

void observer_actor::setActor(actor& ac)
{
 this->ac = &ac;
 ac.setObserver(*this);
}
void observer_actor::notify()
{
 actor& _ac = *ac;

 for(auto i = oslist.begin(); i != oslist.end(); ++i)
    (*i) << _ac << '\n';
}
```