

**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.0382

Ильин Д.А.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2021

## **Цель работы.**

Изучить применение логгеров, изучение перегрузки оператора вывода в поток; написание логгеров нескольких типов, для отслеживания изменений в состоянии объекта.

## **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 состояния 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

*Потенциальные паттерны проектирования, которые можно использовать:*

- *Адаптер (Adapter) - преобразование данных к нужному формату логирования*
- *Декоратор (Decorator) - форматирование текста для логирования*
- *Мост (Bridge) - переключение между логированием в файл/консоль*
- *Наблюдатель (Observer) - отслеживание объектов, которые необходимо логировать*
- *Синглтон (Singleton) - гарантия логирования в одно место через одну сущность*
- *Заместитель (Proxy) - подстановка и выбор необходимого логирования*

### **Выполнение работы.**

В ходе работы были использованы паттерны: Наблюдатель, Одиночка, Мост и Декоратор.

Паттерн Наблюдатель был реализован, чтобы создать механизм, благодаря которому класс логгера сможет следить и реагировать на события в классах объектов. Его суть проста — создаются Наблюдатели и Субъекты. Наблюдатели следят за субъектами. Соответственно, был создан интерфейс Observer (т.е. интерфейс Наблюдателей), который имеет всего один виртуальный метод Update() (переопределяется у наследников, нужен для обновления информации об объекте). Для Субъектов был создан класс Subject. В этом классе мы создали поле Observer\* observer, для хранения указателя на Наблюдателя, что будет следить за данным субъектом. Также были написаны два метода: SetObs(Observer\* obs) и Notify(). Первый нужен для установки указателя на наблюдателя obs в соответствующее поле observer. Второй же вызывает метод Update() у наблюдателя, если таковой установлен. Т.е. вызывает у наблюдателя метод, для обновления информации о себе. Также мы перегружаем оператор вывода в поток. Теперь при попытке вывести в поток объект этого класса (ну или классов-наследников) будут выведены данные о нем. Для этого же мы написали в этом классе виртуальную функцию GetLog() (переопределена у классов-наследников), что и вернет эти данные. Кстати говоря, поскольку Наблюдатель следит за Субъектами, а они, в нашем случае, все объекты, то наследуем класс Elem от класса Subject.

Паттерн Одиночка был применен, чтобы гарантировать наличие единственного экземпляра класса и предоставлять глобальную точку доступа. Именно этот паттерн и был применен вместе с паттерном Наблюдатель для создания класса Logger, который в дальнейшем и будет управлять логгированием игровых объектов. Данный класс наследуется от интерфейса Observer (поскольку он у нас и является наблюдателем). Соответственно, класс имеет поле Subject\* subject, где и будет храниться субъект, за которым

происходит слежка. Также имеет поле `LoggerImplication* log`, где будет храниться определенно собранный логгер (смысл этой фразы будет ясен, когда будет момент о паттерне Декоратор). Переопределяем тут метод `Update()` интерфейса `Observer`. При вызове данного метода вызывается вывод информации о субъекте `subject` (что хранится в наблюдателе) слежки через метод `Out(subject)` объекта `log`. Теперь конкретно о реализации паттерна Одиночка: было добавлено поле `Logger* logger`. Именно в нем и будет храниться единственный экземпляр класса. Как раз поэтому мы лишаем пользователя доступа к конструктору данного класса. Все, что он может — это вызвать специальный метод `GetInstance(Subject* sub, LoggerImplication* log)`, аргументы которого — это объект, за которым будет слежка и как раз специально собранный логгер. Если изначально в поле `logger` нет экземпляра класса `Logger`, то мы создаем новый с применением конструктора; если же экземпляр уже имеется, то просто помещаем указатели на субъект `sub` и логгер `log` в соответствующие поля экземпляра `subject` и `log`. Не забываем установить в субъект данный экземпляр как наблюдателя методом `SetObs(logger)`.

Для реализации разных типов логгеров был написан интерфейс `LoggerImplication`, который уже упоминался ранее. Он имеет виртуальную функцию `Out(Subject* sub)` (переопределяется у классов-наследников), которая как раз нужна для вывода информации об объекте. Именно от этого интерфейса наследуется класс `Decorator`. Поскольку мы имеем несколько типов логгеров, то можем собирать различные их комбинации. Именно для этого мы применим паттерн Декоратор. Он позволит нам «оборачивать» разные логгеры друг в друга, тем самым позволяя нам составлять различные их комбинации. Как это реализовано? Довольно просто. В данном классе мы имеем поле `LoggerImplication* logger`, в котором можем хранить указатель на определенный тип логгера. Соответственно, когда будет вызван переопределенный метод `Out(Subject* sub)`, этот же метод будет вызван у логгера, что хранится в `logger`; соответственно, если таковой тоже хранит логгер, то для него также будет

вызван метод `Out()`... Вместе с этим каждый логгер выводит в поток переданный субъект `sub` (поскольку для `Subject` перегружен вывод в поток — это возможно). Так что же имелось ввиду под «собранный логгер» в одном из предыдущих абзацев. А понимать под этим следует логгер, который собран из разных типов «оборачиваниями».

Согласно условиям были созданы два класса, соответствующие двум типам логгеров: `FileLogger` и `ConsoleLogger`. Каждый из них наследуется от декоратора, т. е. от класса `Decorator`. Когда создаются их экземпляры (`Decorator(LoggerImplication* log)`), то в поле `LoggerImplication* logger` помещается указатель на переданный объект `log`. Это как раз и обеспечивает «оборачивание». В переопределенном методе `Out(Subject* sub)` происходит две вещи, о которых уже говорилось. Во-первых, вызывается метод `Out(sub)` у хранящегося объекта `logger`, а также происходит вывод в поток субъекта `sub`. В случае `ConsoleLogger` вывод в консоль; в случае с `FileLogger` вывод в файл `log.txt`.

Кстати говоря, был же еще применен паттерн Мост, что позволяет нам переключать разные собранные логгеры. По сути, мы создали Реализацию и Абстракцию. Под реализацией здесь следует понимать класс `Logger`, под абстракцией - интерфейс `LoggerInteraction` (т. е. при помощи наблюдателя мы управляем логгерами).

Далее чуть подробнее о методах созданных классов:

1. Был создан класс-интерфейс `Observer`, имеющий метод:
  - виртуальный метод `Update()` для вывода информации об изменениях состояния объекта (метод реализуется в классе-наследнике)
2. Был создан класс субъекта `Subject` (субъекты и будут отслеживаться логгером). Имеет поле `Observer* observer` (т.е. субъект может хранить указатель на наблюдателя). Объявлены следующие методы:

- метод `SetObs(Observer* obs)` для установки указателя наблюдателя в поле субъекта `observer`
- метод `Notify()` вызывает метод `Update()` наблюдателя для обновления информации о его состоянии, если у субъекта установлен указатель на наблюдателя, иначе в консоль выведется сообщение, что наблюдатель у объекта не установлен
- виртуальный метод `GetLog()` для получения строки с информацией о состоянии субъекта (метод реализуется в классах-наследниках)
- дружественная функция-перегрузка оператора вывода в поток `operator<<(std::ostream &out, Subject* sub);` теперь при выводе в поток субъекта будет вызван вывод в поток строки с состоянием субъекта, т.е. результат метода `GetLog()`

3. Чтобы логгер мог работать с субъектами, они должны быть. В нашей игре мы будем отслеживать состояния игрока, противников и вещей. Все они являются наследниками интерфейса `Elem`. Делаем так, чтобы класс `Elem` наследовался от класса субъектов `Subject`. В соответствующих классах был добавлен метод:

- переопределенный метод `GetLog()`, который возвращает строку с информацией о состоянии объекта

4. Был создан класс-интерфейс `LoggerImplication`, имеющий метод:

- виртуальный метод `Out(Subject* sub)` для вывода объекта в поток (метод реализуется в классах-наследниках)

5. Был создан класс `FileLogger`, наследуемый от `LoggerImplication`. Имеет поле `std::ofstream output` для хранения потока для файлового вывода. Имеет метод:

- переопределенный метод `Out(Subject* sub)` для вывода в поток `output` объект (поскольку оператор вывода в поток перегружен, то будет выведена информация об объекте)

6. Был создан класс ConsoleLogger, наследуемый от LoggerImplication.

Имеет метод:

- переопределенный метод Out(Subject\* sub) для вывода в поток стандартный поток (т.е. в консоль) объект (поскольку оператор вывода в поток перегружен, то будет выведена информация об объекте)

7. Был создан класс-одиночка Logger, наследуемый от Observer. Имеет поля static Logger\* logger, Subject\* subject, LoggerImplication\* log.

Имеет следующие методы:

- переопределенный метод Update() для вывода информации об изменениях состояния объекта; для этого вызывается метод Out(subject) у объекта log (т.е. происходит вывод информации через определенный тип логгера)
- метод GetInstance(Subject\* sub, LoggerImplication\* log), который возвращает единственный экземпляр своего класса. Это нужно, поскольку конструктор класса-одиночки скрыт от пользователя, а экземпляр нужен. Собственно, проверяем, если logger имеет нулевой указатель, то передаем в поле logger новый объект Logger(sub, log), а затем устанавливаем субъекту наблюдателя с помощью метода SetObs(logger)

UML-диаграмма классов представлена на рис. 1.

## Тестирование:

```
type: Player
Id:0
now health: 302
max health: 302
damage: 304
coord: (4, 9)
//-----
```

```
type: Player
Id:0
now health: 311
max health: 311
damage: 319
coord: (5, 9)
//-----
```

```
type: Player
Id:0
```

```
type: Tank
Id:1
Live: Live
now health: 66
damage: 22
coord: (5, 3)
//-----
```

```
type: Tank
Id:1
Live: Live
now health: 1072
damage: 24
coord: (6, 3)
//-----
```

```
type: Tank
Id:1
```

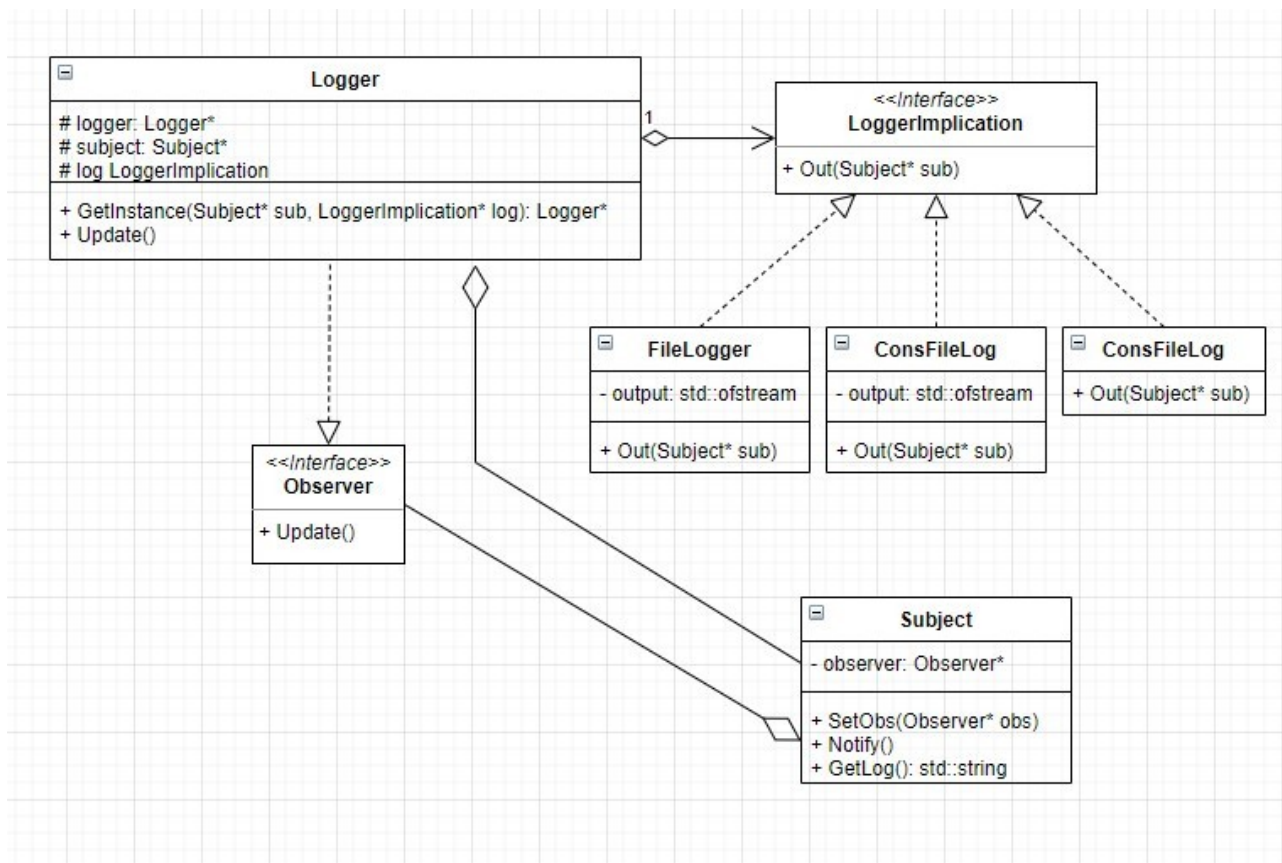
```
type: MP
Live: Live
coord: (1, 7)
//-----
```

```
type: MP
Live: Dead
coord: (1, 7)
//-----
```

```
type: MP
Live: Dead
coord: (1, 7)
//-----
```



## UML-диаграмма:



## **Выводы.**

В ходе работы было изучено применение логгеров и изучение перегрузки оператора вывода в поток; был написан логгер с возможностью вывода в консоль/в файл/в консоль и файл для отслеживания изменений в состоянии объекта.