

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(ка) гр. 1381

Денисова О.К.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2022

Цель работы

Изучить способы логирования, реализовать логирование в лабораторной работе.

Общая формулировка задачи

- Разработан класс/набор классов отслеживающий изменения разных уровней
- Разработаны классы для вывода в консоль и файл с соблюдением идиомы RAII и перегруженным оператором вывода в поток.
- Разработанные классы спроектированы таким образом, чтобы можно было добавить новый формат вывода без изменения старого кода (например, добавить возможность отправки логов по сети)
- Выбор отслеживаемых уровней логирования должен происходить в runtime
- В runtime должен выбираться способ вывода логов (нет логирования, в консоль, в файл, в консоль и файл)

Выполнение работы

В ходе выполнения работы был реализован паттерн Наблюдатель. В рамках паттерна написаны два интерфейса – `IObservable` и `IObserver`.

Интерфейс `IObservable` имеет метод `notify`, который сообщает каждому конкретному `Observer`'у (наследникам `IObserver`) об изменениях, а также метод `addObserver`. Игровое поле наследуется от интерфейса `IObservable`, так как оно является наблюдаемым объектом.

Интерфейс `IObserver` имеет метод `update`, отвечающий за обновление информации, в нашем случае – логирование.

Класс `LoggerPool` является реализацией интерфейса `IObserver`. В классе в частных полях находятся два вектора – `std::vector<LogLevels*>` и `std::vector<Loggers*>`. Таким образом, в зависимости от добавленных в векторы конкретных уровней и конкретных логгеров, определяется, что будет логироваться и куда. Для добавления в векторы уровней и логгеров реализованы соответственно методы `addLevel` и `addLogger`. Метод `update`, наследованный от интерфейса `IObserver`, проходит двумя вложенными циклами по каждому уровню и каждому логгеру, добавленным в векторы, таким образом реализуя логирование всех запрошенных уровней во все запрошенные логгеры. Также в классе реализован метод `config`, в который в runtime подается запрос от пользователя для настройки уровней и логгеров.

Для сообщений реализован интерфейс `Message`, который реализуют три класса – `GameMessage`, `EventMessage` и `ErrorMessage`, каждый из которых отвечает за создание сообщений с префиксом, отвечающим этому виду сообщения. Сообщения создаются в различные игровые моменты при условии выполнения определенных условий, после создания сообщения сразу же вызывается метод `notify`, оповещающий наблюдателей (в нашем случае конкретным наблюдателем является `LoggerPool`) о появившихся изменениях. Для каждой реализации `Message`'а реализован перегруженный оператор вывода в поток, помеченный как `friend`. В интерфейсе `Message` реализован чисто виртуальный деструктор, в наследниках – конкретные деструкторы.

Интерфейс `LogLevel` отвечает за контроль уровня логирования, в нем объявлен чисто виртуальный метод `send`, который переопределяется в каждом из реализаций `LogLevel`'а – `GameLog`, `EventLog` и `ErrorLog`. В каждом из реализаций метод `send` получает `Message *` и пытается откатовать сообщение к соответствующему ему типу сообщения, если ему это удастся – сообщение проходит дальше и отправляется на вывод в логи. Таким образом, работу вектора `std::vector<LogLevel*>` и метода `send`

можно рассматривать как фильтр сообщений нужного уровня. В интерфейсе LogLevel реализован чисто виртуальный деструктор, в наследниках – конкретные деструкторы.

Интерфейс Logger отвечает за запись, имеет чисто виртуальный метод print, переопределенный в реализациях – ConsoleLogger и FileLogger. Метод print в ConsoleLogger выводит в std::cout, в FileLogger – в файл с заданным именем (конструктор FileLogger'а принимает имя выходного файла и открывает/создает файл). Далее метод print делает вывод в заданный файл, и деструктор FileLogger'а закрывает файл, таким образом соблюдается идиома RAII.

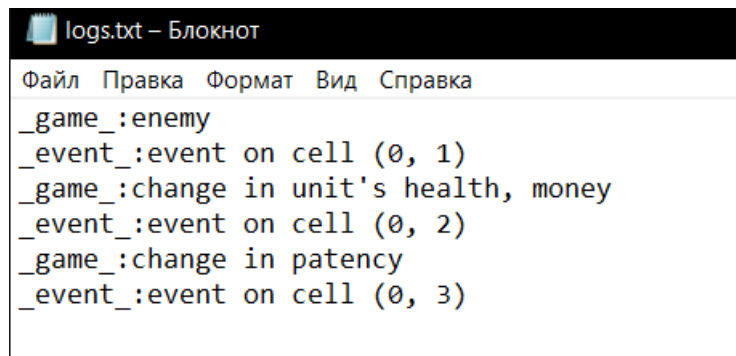
Очищение памяти от всех созданных сообщений происходит в методе update: вне зависимости от того, вывелось сообщение в логи или нет, оно удаляется сразу после данной обработки.

Очищение памяти от созданных логгеров и уровней осуществляется в деструкторе LoggerPool'а.

Примеры работы программы

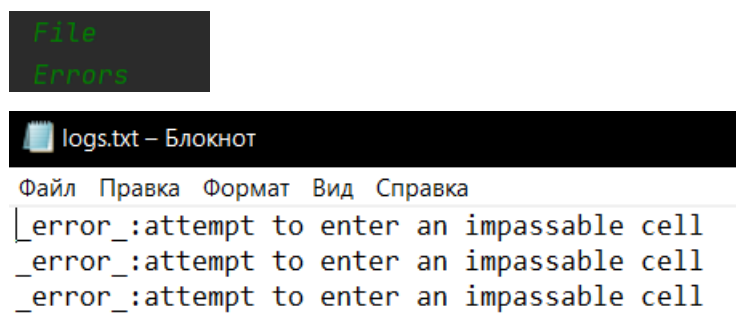
1) На данном примере выбран вывод логов игры и событий, вывод и в консоль, и в файл.

```
File Console
Events Game
_game_:enemy
_event_:event on cell (0, 1)
_game_:change in unit's health, money
_event_:event on cell (0, 2)
_game_:change in patency
_event_:event on cell (0, 3)
```



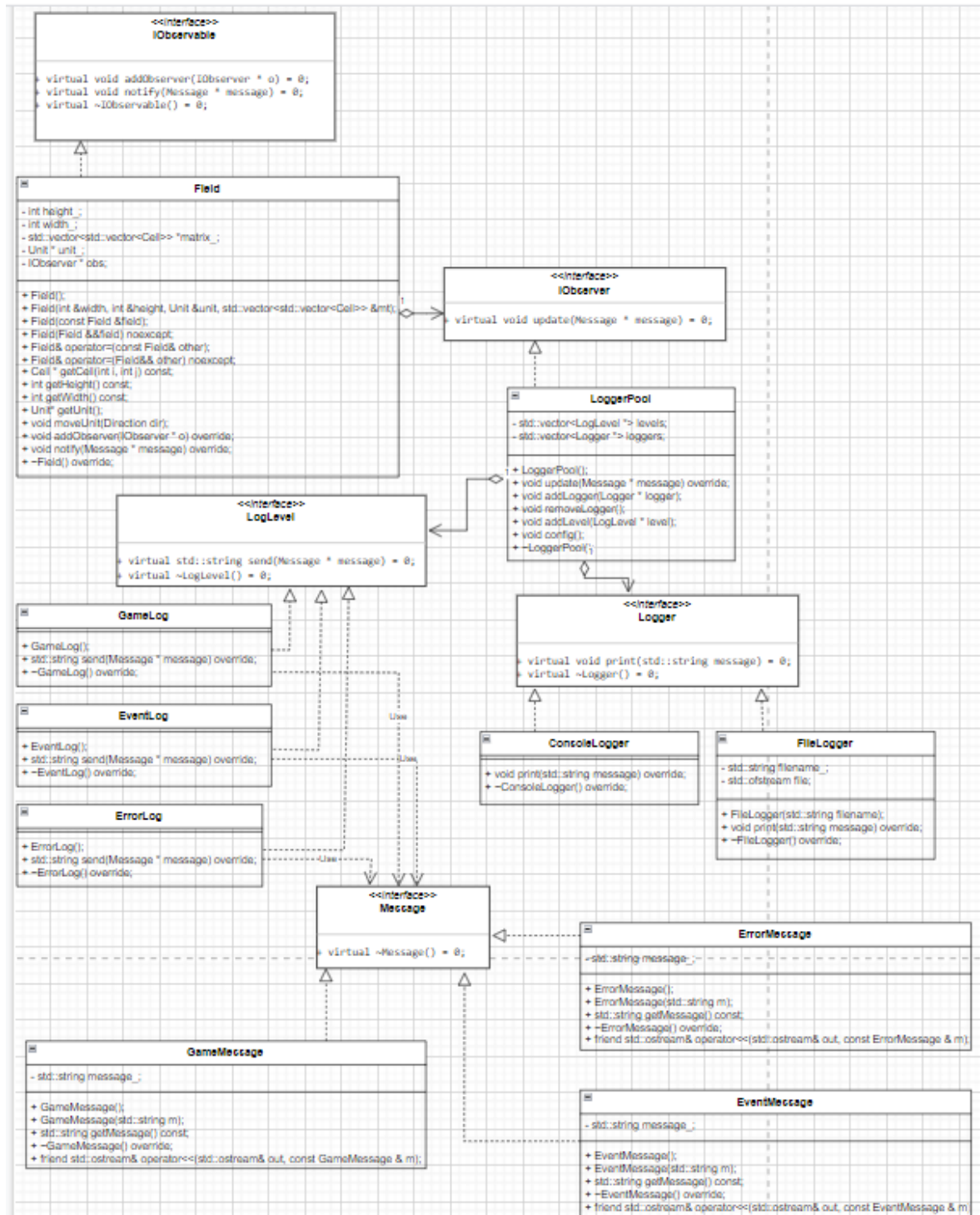
```
logs.txt - Блокнот
Файл  Правка  Формат  Вид  Справка
_game_:enemy
_event_:event on cell (0, 1)
_game_:change in unit's health, money
_event_:event on cell (0, 2)
_game_:change in patency
_event_:event on cell (0, 3)
```

2) На данном примере выбран вывод логов ошибок, вывод только в файл.



```
File
Errors
logs.txt - Блокнот
Файл  Правка  Формат  Вид  Справка
_error_:attempt to enter an impassable cell
_error_:attempt to enter an impassable cell
_error_:attempt to enter an impassable cell
```

UML-диаграмма



Выводы

В ходе выполнения лабораторной работы были приобретены знания о перегрузке операторов и функций, логировании. Был приобретен опыт написания логирования в игре.