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. 1384 Прошичев А.В.

Преподаватель Жангиров Т.Р.

Санкт-Петербург
2022

Цель работы.

Изучить и освоить на практике логирование. Научиться перегружать операторы, а также грамотно расставлять логи для качественного отслеживания ошибок (дебага).

Задание.

Реализовать класс/набор классов отслеживающих изменения состояний в программе. Отслеживание должно быть 3-х уровней:

1. Изменения состояния игрока и поля, а также срабатывание событий
2. Состояние игры (игра начата, завершена, сохранена, и.т.д.)
3. Отслеживание критических состояний и ошибок (поле инициализировано с отрицательными размерами, игрок попытался перейти на непроходимую клетку, и.т.д.)

Реализованы классы для вывода информации разных уровней для в консоль и в файл с перегруженным оператором вывода в поток.

Требования:

- 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)

Примечания:

- Отслеживаемые сущности не должны ничего знать о сущностях, которые их логируют
- Уровни логирования должны быть заданными отдельными классами или перечислением
- Разные уровни в логах должны помечаться своим префиксом
- Рекомендуется сделать класс сообщения
- Для отслеживания изменений можно использовать наблюдателя
- Для вывода сообщений можно использовать адаптер, прокси и декоратор

Выполнение работы.

Для отслеживания изменений в сущностях используется паттерн «Наблюдатель». Для его реализации были созданы два класса: Observer и Subject.

Класс Observer – интерфейс, реализации которого получают возможность наблюдать за объектами, наследуемыми от класса Subject. Для уведомления Observer о изменениях используется функция Update().

Класс Subject – класс, который позволяет своим потомкам быть наблюдаемыми объектами. Такой класс передаёт методы: подпись на своё наблюдение Attach(), отписка от наблюдения за собой Detach() и уведомление своих наблюдателей о изменении Notify(), который вызывает метод Update() у всех подписанных наблюдателей.

Важной частью для передачи информации в логировании является класс Message. Данный класс передаётся с помощью Notify() и принимается благодаря Update(). Класс является реализацией стандартного сообщения (передающего только строку). Также класс позволяет устанавливать и проверять свой уровень логирования с помощью GetType() и SetType(). К тому же прибавляется метод вывода сообщения PrintMessage(). Так, если нужно передать строковый лог без каких-либо дополнительных данных, то нужно создать Message, передать в его конструктор строку лога и отправить с помощью Notify().

Для передачи не только строковых значений существуют классы IntMessage, IntIntMessage и CharMessage, которые наследуются от класса Message.

IntMessage принимает своим конструктором кроме строки ещё целочисленное значение и имеет переопределение функции PrintMessage() для вывода сообщения уже с новым целочисленным значением (сначала выводится строка, потом через пробел число)

CharMessage аналогично своим конструктором дополнительно со строкой принимает символ и также переопределяет функцию PrintMessage() (сначала выводится строка, потом через пробел символ)

IntIntMessage похожим образом своим конструктором забирает со строкой два целочисленных значения и переопределяет функцию PrintMessage() (сначала выводит строку, потом ставит двоеточие и выводит пару целочисленных значений через пробел)

Сами по себе сообщения нейтральные и не относятся к определённому уровню логирования, поэтому был добавлен паттерн «Декоратор». Его представителем является класс Decorator со своими потомками StatusDecorator, GameDecorator, ErrorDecorator.

Декоратор наследуется от класса Message и становится его «оболочкой», перенимая точно такое же поведение. Методы декораторы, переопределены так, что просто перекидывают на аналогичные методы Message.

Класс StatusDecorator наследуется от декоратора и точно также является оболочкой над ним, только он вносит свои изменения. Например, заносит в личный Message уровень логирования Log_Status. А также при попытке вывести сообщение через эту оболочку добавляет приписку [STATUS] в начале вывода, что указывает на уровень логирования.

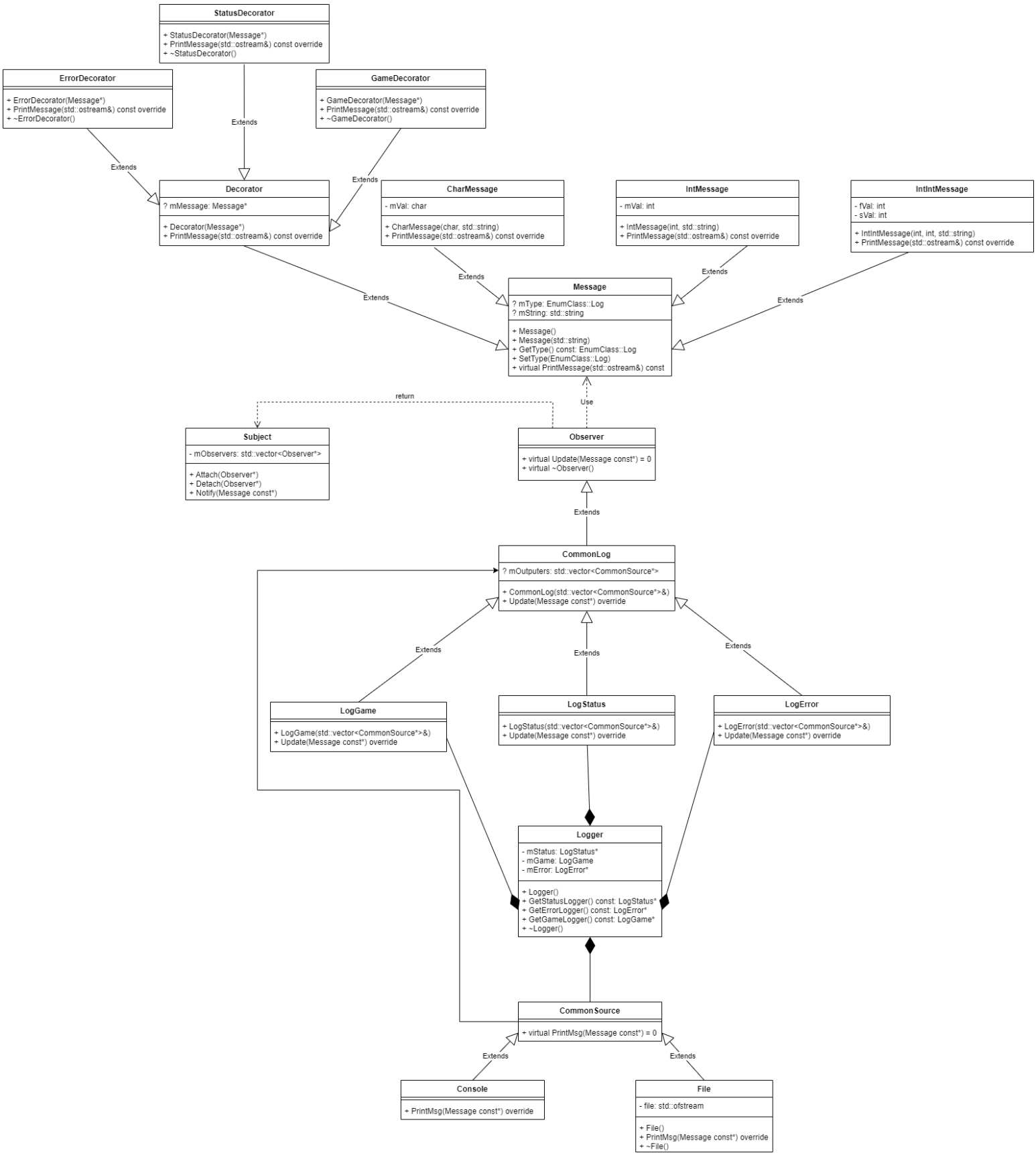
Аналогично с классами GameDecorator и ErrorDecorator, только они вносят свои приписку [GAME] и [ERROR] соответственно, а также устанавливают свой уровень логирования (Log_Game и Log_Error соотвественно).

Наследником Observer является класс CommonLog, который является промежуточным звеном между интерфейсом наблюдателя и его конечными реализациями. Данный класс нужен для ввода своим потомкам массива указателей на класс CommonSource, который является интерфейсом способа вывода лога.

Классы LogGame, LogStatus и LogError являются конечной реализацией наблюдателя и являются уровнями логирования. Эти классы хранят массив на возможные способы вывода лога, но допускают к нему только те сообщения, метка уровня логирования которых совпадает с ними.

CommonSource – интерфейс, который имеет только один метод вывода сообщения. Его реализациями являются классы File и Console. Данные классы переопределяют метод так, что вызывают метод вывода у сообщения и передаёт в него поток, за который они отвечают (File передаёт std::ofstream, а Console – std::cout). Также стоит упомянуть, что File имеет деструктор, в котором происходит закрытие файла, и коструктор, в котором происходит открытие файла.

Самым главным классом, создающим связь между всеми перечисленными частями, является класс Logger. Данный класс при создании производит считывания данных от пользователя о способах вывода логов и уровнях логирования. Данный класс и создаёт массив способов вывода и передаёт в уровни логирования, который точно также он создаёт и уничтожает. Данный класс создаётся в функции main() и подключает всех нужных сущностей к наблюдению от нужных классов уровней логирования.


StatusDecorator
+ StatusDecorator(Message*)
+ PrintMessage(std::ostream&) const override
+ ~StatusDecorator()
ErrorDecorator
+ ErrorDecorator(Message*)
+ PrintMessage(std::ostream&) const override
+ ~ErrorDecorator()
GameDecorator
+ GameDecorator(Message*)
+ PrintMessage(std::ostream&) const override
+ ~GameDecorator()
Extends
Decorator
? mMessage: Message*
+ Decorator(Message*)
+ PrintMessage(std::ostream&) const override
CharMessage
- mVal: char
+ CharMessage(char, std::string)
+ PrintMessage(std::ostream&) const override
IntMessage
- mVal: int
+ IntMessage(int, std::string)
+ PrintMessage(std::ostream&) const override
IntIntMessage
- fVal: int
- sVal: int
+ IntIntMessage(int, int, std::string)
+ PrintMessage(std::ostream&) const override
Message
? mType: EnumClass::Log
? mString: std::string
+ Message()
+ Message(std::string)
+ GetType() const: EnumClass::Log
+ SetType(EnumClass::Log)
+ virtual PrintMessage(std::ostream&) const
return
Use
Subject
- mObservers: std::vector<Observer*>
+ Attach(Observer*)
+ Detach(Observer*)
+ Notify(Message const*)
Observer
+ virtual Update(Message const*) = 0
+ virtual ~Observer()
CommonLog
? mOutputers: std::vector<CommonSource*>
+ CommonLog(std::vector<CommonSource*>&)
+ Update(Message const*) override
LogGame
+ LogGame(std::vector<CommonSource*>&)
+ Update(Message const*) override
LogStatus
+ LogStatus(std::vector<CommonSource*>&)
+ Update(Message const*) override
LogError
+ LogError(std::vector<CommonSource*>&)
+ Update(Message const*) override
Logger
- mStatus: LogStatus*
- mGame: LogGame
- mError: LogError*
+ Logger()
+ GetStatusLogger() const: LogStatus*
+ GetErrorLogger() const: LogError*
+ GetGameLogger() const: LogGame*
+ ~Logger()
CommonSource
+ virtual PrintMsg(Message const*) = 0
Console
+ PrintMsg(Message const*) override
File
- file: std::ofstream
+ File()
+ PrintMsg(Message const*) override
+ ~File()

Тестирование.

```
Please, choose one or several log types.
If program take an incorrect input from you, it will set default values.
The list of the types:
Enter 1 -- log type that controls Player, Field and events
Enter 2 -- log type that controls the status of the game
Enter 3 -- log type that controls critical mistakes from a user
if you'd like to use any output type, enter the corresponding number.
Could enter several numbers, spliting their with a space.
```

Изображение №1. Интерфейс при выборе уровней логов

```
Please, choose one or several output types.
The list of the types:
Enter 1 -- output to the console
Enter 2 -- output to the file
if you'd like to use any output type, enter the corresponding number.
Could enter several numbers, spliting their with a space.
```

Изображение №2. Интерфейс при выборе способов вывода логов

```
Incorrect command! Please, check the correct format of the enter!
[ERROR] Attemp to enter incorrect values: н
п
Incorrect command! Please, check the correct format of the enter!
[ERROR] Attemp to enter incorrect values: п
```

Изображение №3. Логи ошибок при неправильном вводе.

```
[STATUS] Created BearBush
[STATUS] Cell triggered from position that has X Coord: 1 Y coord: 10
[STATUS] Created SaintWaterCave
[STATUS] Cell triggered from position that has X Coord: 4 Y coord: 14
[STATUS] Thirst: 19

[LOG] The game did a step. Now 1 steps have been taken
```

Изображение №4. Логи от игрока, поля, событий и игрового шага

```
[LOG] Field got size
[STATUS] Player moved on X Coord: 7 Y coord: 6
[STATUS] Created WaterCave
[STATUS] Cell triggered from position that has X Coord: 11 Y coord: 4
[STATUS] Created WaterCave
[STATUS] Cell triggered from position that has X Coord: 14 Y coord: 6
[STATUS] Created RabbitBush
[STATUS] Cell triggered from position that has X Coord: 13 Y coord: 7
[STATUS] Created RabbitBush
[STATUS] Cell triggered from position that has X Coord: 1 Y coord: 8
[STATUS] Created WolfBush
[STATUS] Cell triggered from position that has X Coord: 4 Y coord: 10
[STATUS] Created WolfBush
[STATUS] Cell triggered from position that has X Coord: 6 Y coord: 10
[STATUS] Created WolfBush
[STATUS] Cell triggered from position that has X Coord: 4 Y coord: 11
[STATUS] Thirst: 19

[LOG] The game did a step. Now 1 steps have been taken
[STATUS] Player moved on X Coord: 6 Y coord: 6
[STATUS] Created WaterCave
[STATUS] Cell triggered from position that has X Coord: 11 Y coord: 3
[STATUS] Created BerryBush
[STATUS] Cell triggered from position that has X Coord: 2 Y coord: 10
[STATUS] Created RabbitBush
[STATUS] Cell triggered from position that has X Coord: 12 Y coord: 10
[STATUS] Created WaterCave
[STATUS] Cell triggered from position that has X Coord: 3 Y coord: 12
[STATUS] Created WaterCave
[STATUS] Cell triggered from position that has X Coord: 1 Y coord: 13
[STATUS] Created EarthQuakeCave
[STATUS] Cell triggered from position that has X Coord: 1 Y coord: 14
[STATUS] Thirst: 18
[STATUS] Hunger: 9
```

Изображение №5. Текстовые файл с выводом логов

Выводы.

Изучил и освоил на практике логирование. Научился перегружать операторы, а также грамотно расставлять логи для качественного отслеживания ошибок (дебага).