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.

Студентка гр. 1304 _____ Чернякова В.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2022

Цель работы.

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.

Описание архитектурных решений и классов.

Новые классы.

Класс *enum class Level:* класс уровней логирования. Для того, чтобы было удобно представлять возможные уровни логирования: *TRACE, INFO, ERROR_INFO* – создан данный класс перечислений.

Класс *Message*: класс сообщения. В данном классе определенно два поля: *std::string message* для текста сообщения и *Level level* для отслеживания какого уровня создаваемое сообщение. Реализованы конструктор по умолчанию и конструктор *Message(std::string message, Level level)* от конкретных аргументов. Метод *std::string& getMessage()* возвращает текст сообщения. *Level getLevel() const* – метод, возвращающий значение уровня логирования нашего сообщения, метод константный, так как внутри него никаких изменений не происходит.

Интерфейс *ILogger*: интерфейс логирования. Представлен чисто виртуальным методом *virtual ILogger& operator<<(Message message)*, его наследники будут перегружать данный оператор << вывода в зависимости консоль это или файл. Реализован чисто виртуальный деструктор.

Класс *ConsoleLogger:* класс логирования в консоль. Класс-наследник интерфейса *ILogger,* в нем перегружается оператор вывода в поток <<с спецификатором *final*, чтобы дальше данная функция не перегружалась. Внутри метода происходит вывод сообщения с помощью *getMessage().*

Класс *FileLogger:* класс логирования в файл. Класс-наследник интерфейса *ILogger*, где переопределён метод вывода оператора в поток <<. Определено приватное поле *std::ofstream file*, в кортом хранится файла, куда будет производиться вывод. Также есть конструктор по умолчанию – создание

файла *FileLogger("FileLogger.txt"),* конструктор от одного аргумента *FileLogger(const std::string& filename) : file(std::ofstream(filename))* и деструктор, закрывающий файл *FileLogger::~FileLogger() {file.close();}*, что позволяет соблюдать идиому *RAII*. Перегружение происходит за счет *file << message.getMessage() << '\n'*.

Класс *Log*: основной класс логирования. Приватные поля: вектор *loggers*, хранящий <*ILogger**>, то есть ссылки на объекты, с помощью которых вывод производится либо в консоль или в файл. Вектор *levels*, хранящий <*Level*>, то есть уровни логирования, выбираемые пользователем. Поле *is_console* необходим для проверки, куда выводить сообщения. Одни из методов – конструктор *Log()* и деструктор *~Log()*. Для добавления в вектора значений с помощью функции *push_back,* то есть уровней и самих логов соответственно, реализованы методы *addLevel(Level level)* и *addLogger(ILogger* logger)*. Конструктор от нескольких аргументов *Log(bool level_trace, bool level_info, bool level_error, bool file_logger, bool console_logger)*, который проверяет, наличие каких уровней есть и какие логи присутствуют, и вызывает соответственно *addLogger* и *addLevel*. Для вывода сообщения по выбору пользователя в консоль или в файл реализован метод *viewMessage*: проход по значениям векторов, если происходит совпадение, то есть уровень сообщения и такой уровень выбран пользователем, то после проверки о необходимости вывода в консоль или в файл происходит соответствующий вывод.

Наблюдатели. Отслеживание изменений.

Для отслеживания изменений игры, поля, событий и игрока были написаны интерфейсы-наблюдатели и соотнесённо сами наблюдатели.

Каждый виртуальный метод внутри интерфейсов принимает в качестве аргументов *Log* параметр, чтобы осуществлять создание и вывод сообщения соответственно.

Интерфейс *IEventObserver*: интерфейс наблюдателя событий. Виртуальные методы срабатывания событий.

Интерфейс *IFieldObserver*: интерфейс наблюдатель за полем. Виртуальные методы изменений, происходящих на поле.

Интерфейс *IGameObserver*: интерфейс наблюдатель за процессом игры. Виртуальные методы изменений, происходящих с самой игрой.

Интерфейс *IPlayerObserve*r: интерфейс наблюдатель за игроком. Виртуальные методы изменений, происходящих с игроком, его характеристиками.

Класс *EventObserver*: класс наследник от соответствующего интерфейса. В переопределенных методах создается сообщение, соответствующее изменениям, у аргумента *Log* вызывается метод *viewMessage* для вывода.

Класс *FieldObserver*: класс наследник от соответствующего интерфейса. В переопределенных методах создается сообщение, соответствующее изменениям, у аргумента *Log* вызывается метод *viewMessage* для вывода. Конструктор создания наблюдателя для его помещения на поле *FieldObserver::FieldObserver(Log& log_) {log = log_;}.*

Класс *GameObserver*: класс наследник от соответствующего интерфейса. В переопределенных методах создается сообщение, соответствующее изменениям, у аргумента *Log* вызывается метод *viewMessage* для вывода.

Класс *PlayerObserver*: класс наследник от соответствующего интерфейса. В переопределенных методах создается сообщение, соответствующее изменениям, у аргумента *Log* вызывается метод *viewMessage* для вывода соответствующего изменения. Конструктор создания наблюдателя для его помещения к игроку *PlayerObserver::PlayerObserver(Log& log_) {log = log_;}.*

Изменения в классах.

Класс *Player*: добавлено поле *IPlayerObserver* player_observer* и метод *void setObserver(Log& log),* в котором создается *new PlayerObserver(log)*. Также внутри методов, соответствующих изменению игрока, у поля *player_observer* вызываются соответственные методы.

Класс *Field*: добавлено поле *IFieldObserver* field_observer* и метод *void setObserver(Log& log),* в котором создается *new FieldObserver(log).* Также внутри методов, соответствующих изменению поля, у *field_observer* вызываются соответствующие методы. Для получения наблюдателя создан геттер *IFieldObserver*& getFieldObserver().*

Интерфейс *IEvent*: добавлен новый аргумент функции *virtual void reaction(Player& player,Log& log)*. То есть во всех конкретных событиях теперь при переопределении данного метода в конструкторе классов устанавливается наблюдатель за событиями, а в методе реакции у данного метода вызывается соответствующий метод наблюдателя.

Класс *CommandReader*: добавлено считывание уровня логирования *getLevel().* Считывание: вывод будет происходить в консоль или в файл *getIsFileLogger()* и *getIsConsoleLogger()*. Методы bool-типа для определения-получения, какие уровни логирования присутствуют: *getIsTraceLevel(), getIsInfoLevel(), getIsErrorLevel().*

Класс *Controller*: проверка уровней логирования и места вывода логов, считанных через *CommandReader*, для конструктора логов соответственно: *bool isTraceLevel(CommandReader& reader), bool isInfoLevel(CommandReader& reader), bool isErrorLevel(CommandReader& reader), bool isFileLogger(CommandReader& reader), bool isConsoleLogger(CommandReader& reader).* В методах, отвечающих за изменения, передается новый аргумент *Log& log* и у соответствующих наблюдателей вызываются методы.

Класс *Application*: Добавлено создание логера *Log log = Log(controller.isTraceLevel(reader), controller.isInfoLevel(reader), controller.isErrorLevel(reader), controller.isFileLogger(reader), controller.isConsoleLogger(reader))*. Новое приватное поле *IGameObserver* game_observer*, который будет отслеживать изменение игровых процессов. Установка наблюдателей на поле и у игрока. Конструктор

Application::Application() : game(true) {game_observer = new GameObserver;} – осуществление наблюдения за игрой.

ПРИЛОЖЕНИЕ.

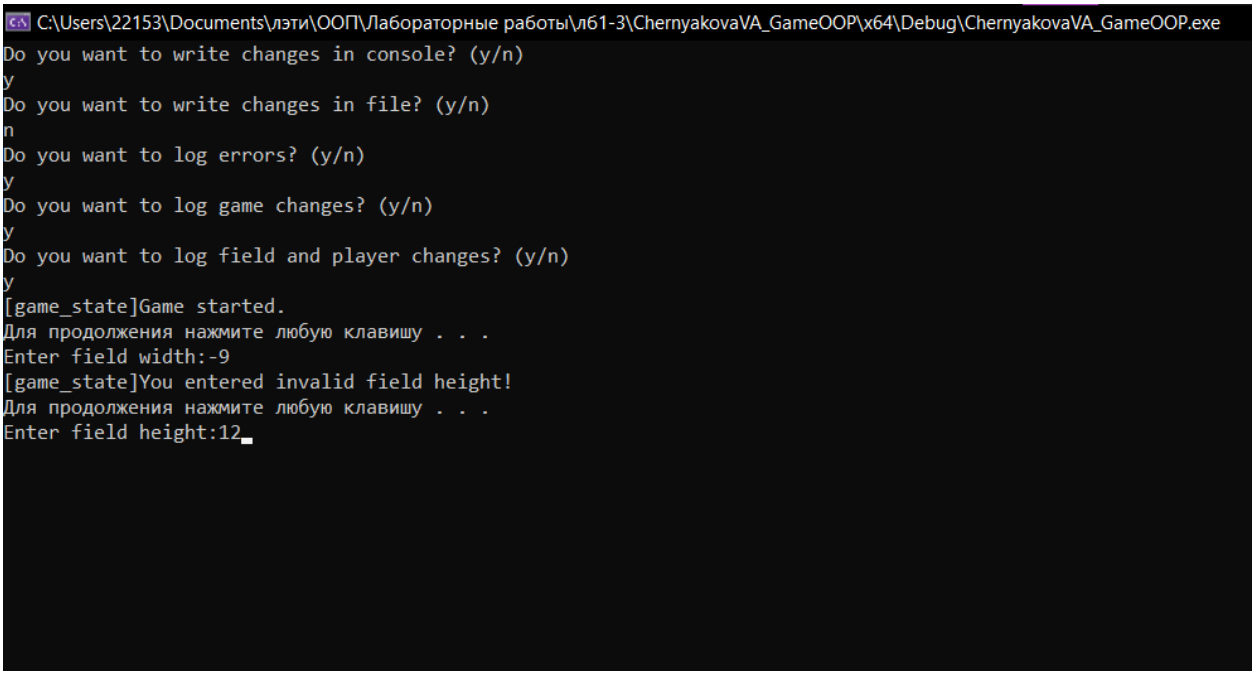

Рисунок 1 – Проверка работоспособности программы. Выбор уровня логирования и способа записи сообщений.

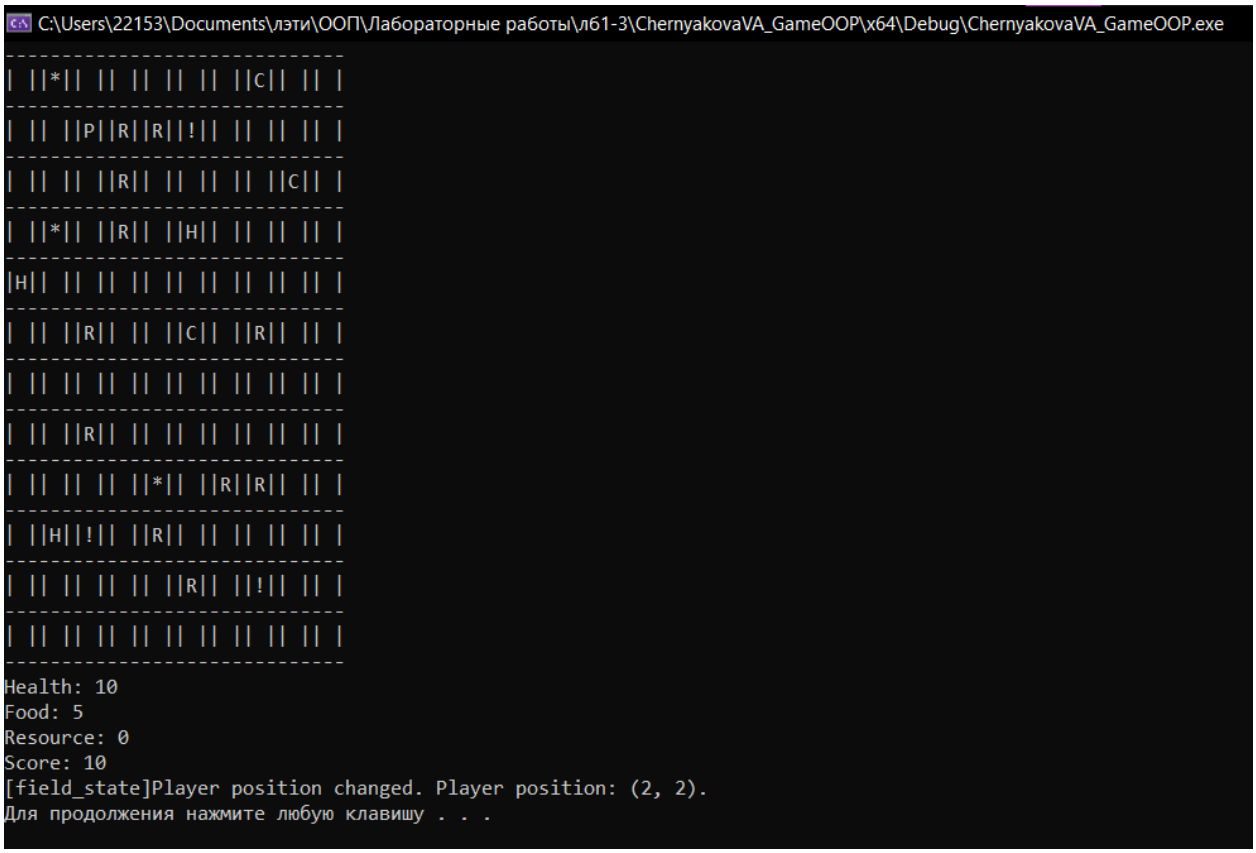

Рисунок 2 – Проверка работоспособности программы. Переход на клетку, которая находится снизу от игрока. Вывод изменений.

Содержимое файла FileLogger.txt после прохождения игры:

```
[game_state]Game started.
[field_state]Player position changed. Player position: (1, 0).
[field_state]Player position changed. Player position: (2, 0).
[event_state]Resource event is active.
[player_state]Player's resource changed. Now there are 3 health.
[player_state]Player's score changed. Now there are 13 score.
[field_state]Player position changed. Player position: (3, 0).
[event_state]Clan event is active.
[player_state]Player's health changed. Now there are 5 health.
[player_state]Player's food changed. Now there are 2 food.
[player_state]Player's score changed. Now there are 9 score.
[player_state]Player lost!
[field_state]Player position changed. Player position: (3, 1).
[field_state]Player position changed. Player position: (3, 2).
[field_state]Player position changed. Player position: (3, 3).
[field_state]Player position changed. Player position: (4, 3).
[field_state]Player position changed. Player position: (5, 3).
[field_state]Player position changed. Player position: (6, 3).
[field_state]Player position changed. Player position: (6, 4).
[field_state]Player position changed. Player position: (7, 4).
[field_state]Player position changed. Player position: (8, 4).
[event_state]Resource event is active.
[player_state]Player's resource changed. Now there are 6 health.
[player_state]Player's score changed. Now there are 12 score.
[field_state]Exit is open now.
[field_state]Player position changed. Player position: (7, 4).
[field_state]Player position changed. Player position: (6, 4).
[field_state]Player position changed. Player position: (6, 5).
[field_state]Player position changed. Player position: (6, 6).
[field_state]Player position changed. Player position: (5, 6).
[field_state]Player position changed. Player position: (5, 7).
[field_state]Player position changed. Player position: (4, 7).
[event_state]Clan event is active.
[player_state]Player's health changed. Now there are 0 health.
[player_state]Player's food changed. Now there are -1 food.
[player_state]Player's score changed. Now there are 8 score.
[player_state]Player lost!
```

```
[game_state]Game ended.
```

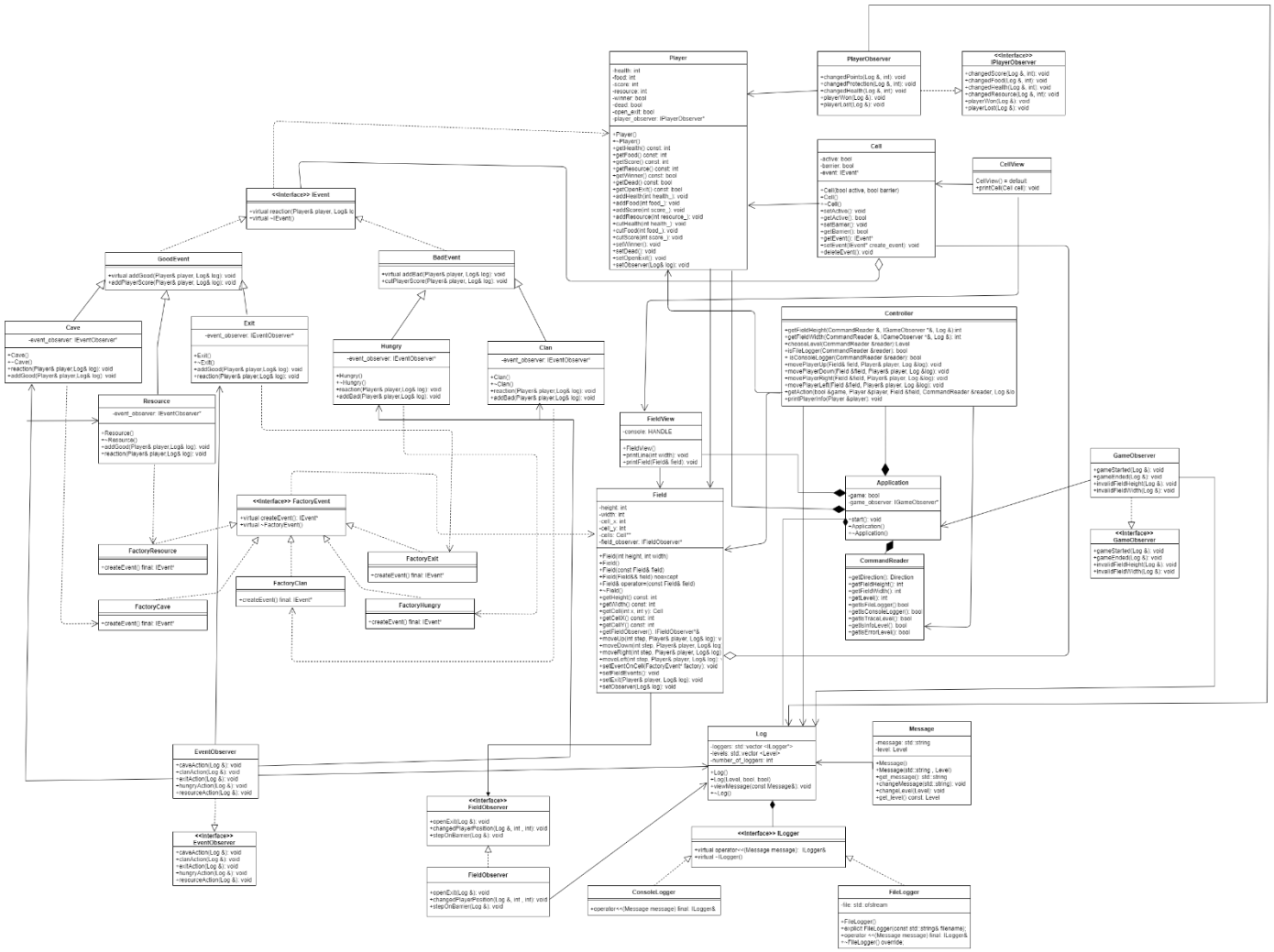

Рисунок 3 – UML-диаграмма межклассовых отношений.