МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №6

по дисциплине «ООП»

Тема: Сохранение и загрузка/написание исключений.

Студент гр. 9383	___________________	Камзолов Н.А.
Преподаватель	___________________	Жангиров Т.Р.

Санкт-Петербург

2020

Цель работы.

Применить на практике знания о создании собственных исключений. Реализовать процесс сохранения и загрузки состояния игры в проекте.

Задание.

Создать классы, которые позволяют сохранить игру, а потом загрузить ее. Также, написать набор исключений, которые как минимум позволяют контролировать процесс сохранения и загрузки

Обязательные условия:

- Игру можно сохранить в файл.
- Игру можно загрузить из файла.
- Взаимодействие с файлами по идиоме RAII.
- Добавлена проверка файлов на корректность.
- Написаны исключения, которые обеспечивают транзакционность.

Дополнительные требования:

- Для получения состояния программы используется паттерн **Снимок.**

Выполнение работы.

Был написан код для сохранения информации о текущем состоянии игры. Для этого были созданы следующие классы:

- GameInfo – класс, содержащий поля для сохранения информации об игре, такой как положение героев, положение врагов, текущее состояние поля и т.д.
- Memento и ConcreteMemento – класс-интерфейс и конкретная реализация этого класса. Этот класс является частью реализации паттерна Снимок и

нужен для хранения поля GameInfo. Он не раскрывает состояние класса GameManager, а лишь получает в конструкторе информацию о нем.

- CareTaker – класс, который ответственен за корректность сохранения и загрузки файла. При сохранении путем несложных преобразований он строит строку для последующей записи в файл. При загрузке сохранения он вытаскивает из строки, хранящейся в файле значения для загрузки, а также проверяет корректность введенных данных(если данные некорректны срабатывает исключение, которое было написано специально для этого события). Для того чтобы получить или обновить информацию о классе GameManager внутри методов класса CareTaker вызываются методы класса GameManager.

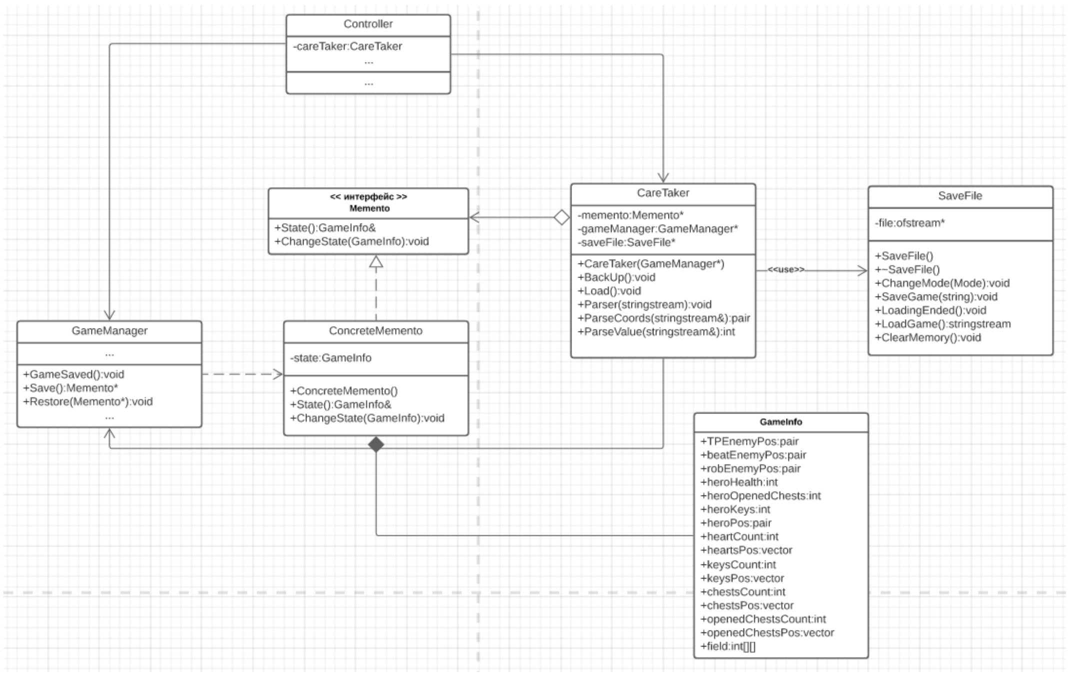
- SaveFile – класс, реализующий идиому RAII для записи в файл при сохранении и чтения из файла при загрузке.

Сохранение реализовано с использованием паттерна Снимок, суть которого в сохранении и восстановлении прошлых состояний объекта, не раскрывая подробности реализации.

То есть класс Memento не знает о том, каким образом создается сохранение, а лишь содержит поле, которое инициализируется в конструкторе с уже заполненным содержанием. Сохранение делает сам GameManager, он же и восстанавливает данные при загрузке сохранения.

Были написаны исключения для обработки сохранения и загрузки(SaveException и LoadSaveException).

UML-диаграмма.



Выводы.

Применены на практике знания о создании собственных исключений. Реализован процесс сохранения и загрузки состояния игры в проекте.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

GameManager.cpp:

```
#include "GameManager.h"


void GameManager::StartGame()
{
    setlocale(0, "");
    gameField = Field::GetInstance();
    gameField->DefineField();

    itemUseObserver = new ItemUseObserver();
    itemTakeObserver = new ItemTakeObserver();
    itemSpawnObserver = new ItemSpawnObserver();
    heroDigObserver = new HeroDigObserver();
    heroGetItemObserver = new HeroGetItemObserver();
    heroMovingObserver = new HeroMovingObserver();
    enemyMovingObserver = new EnemyMovedObserver();
    enemySpawnObserver = new EnemySpawnedObserver();
    enemyHeroMeetObserver = new EnemyHeroMeetObserver();
    loadSaveObserver = new LoadSaveObserver();
    saveObserver = new SaveObserver();


    eventManager.Attach(itemSpawnObserver, Event::ItemSpawn);
    eventManager.Attach(itemUseObserver, Event::ItemUse);
    eventManager.Attach(itemTakeObserver, Event::ItemTake);
    eventManager.Attach(heroDigObserver, Event::HeroDig);
    eventManager.Attach(heroGetItemObserver, Event::HeroGetItem);
    eventManager.Attach(heroMovingObserver, Event::HeroMoving);
    eventManager.Attach(enemyMovingObserver, Event::EnemyMove);
    eventManager.Attach(enemySpawnObserver, Event::EnemySpawn);
    eventManager.Attach(enemyHeroMeetObserver,
Event::EnemyHeroMeet);
```

```
    gameState = new HeroState(this);
    //gameState->SetContext(this);

    PlaceItems();

    displayView = new Display(gameField);
    PlaceEnemies();
}

void GameManager::GameSaved()
{
    saveObserver->Update("");
}

void GameManager::SetState(GameState* newState)
{
    if (gameState)
    {
        delete gameState;
    }
    gameState = newState;
}

void GameManager::EnemyTurn()
{
    gameState->Move();
}

void GameManager::HeroMoving(int moveX, int moveY)
{
    if  (hero.GetX()  +  moveX  <  0  ||  hero.GetX()  +  moveX  ==
FIELD_WIDTH || hero.GetY() + moveY < 0 || hero.GetY() + moveY ==
FIELD_HEIGHT) return;
```

```
    if (gameField->GetField()[hero.GetY() + moveY][hero.GetX() +
moveX].GetType() == CellType::WALL) return;
    hero.ChangePosition(moveX, moveY);
    gameField->GetField()[hero.GetY() - moveY][hero.GetX() -
moveX].isHeroOnCell = false;
    gameField->GetField()[hero.GetY()][hero.GetX()].isHeroOnCell =
true;
    eventManager.Notify(Event::HeroMoving, "Y: " +
std::to_string(hero.GetY()) + ", X: " +
std::to_string(hero.GetX()));
    if (hero.GetX() == robEnemy->getX() && hero.GetY() ==
robEnemy->getY())
    {
        *robEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(RobEnemy)");
        LogHeroInfo();
    }

    if (hero.GetX() == beatEnemy->getX() && hero.GetY() ==
beatEnemy->getY())
    {
        *beatEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(BeatEnemy)");
        LogHeroInfo();
    }


    if (hero.GetX() == TPEnemy->getX() && hero.GetY() == TPEnemy-
>getY())
    {
        *TPEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(TPEnemy)");
        LogHeroInfo();
    }
    SetState(new EnemyState(this));
    EnemyTurn();
}

void GameManager::EnemyMoving()
{
    gameField->GetField()[TPEnemy->getY()][TPEnemy-
>getX()].DeleteEnemy();
    TPEnemy->Move();
    gameField->GetField()[TPEnemy->getY()][TPEnemy-
>getX()].PlaceEnemy(TPEnemy);
    eventManager.Notify(Event::EnemyMove,       "X:       "       +
std::to_string(TPEnemy->getX())      +      ",      Y:      "      +
std::to_string(TPEnemy->getY()) + "(TPEnemy)");
    if (hero.GetX() == TPEnemy->getX() && hero.GetY() == TPEnemy-
>getY())
    {
        *TPEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(TPEnemy)");
        LogHeroInfo();
    }
    gameField->GetField()[beatEnemy->getY()][beatEnemy-
>getX()].DeleteEnemy();
    beatEnemy->Move();
    gameField->GetField()[beatEnemy->getY()][beatEnemy-
>getX()].PlaceEnemy(beatEnemy);
    eventManager.Notify(Event::EnemyMove,       "X:       "       +
std::to_string(beatEnemy->getX())      +      ",      Y:      "      +
std::to_string(beatEnemy->getY()) + "(BeatEnemy)");
    if  (hero.GetX()  ==  beatEnemy->getX()  &&  hero.GetY()  ==
beatEnemy->getY())
    {
        *beatEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(BeatEnemy)");
        LogHeroInfo();
    }
    gameField->GetField()[robEnemy->getY()][robEnemy-
>getX()].DeleteEnemy();
    robEnemy->Move();
    gameField->GetField()[robEnemy->getY()][robEnemy-
>getX()].PlaceEnemy(robEnemy);
    eventManager.Notify(Event::EnemyMove,       "X:       "       +
std::to_string(robEnemy->getX())      +      ",      Y:      "      +
std::to_string(robEnemy->getY()) + "(RobEnemy)");
    if   (hero.GetX()   ==   robEnemy->getX()   &&   hero.GetY()   ==
robEnemy->getY())
    {
        *robEnemy - hero;
        eventManager.Notify(Event::EnemyHeroMeet, "(RobEnemy)");
        LogHeroInfo();
    }



}

void GameManager::DigGrass()
{
    gameField-
>GetField()[hero.GetY()][hero.GetX()].SetType(CellType::DIGGEDTRAI
L);
    eventManager.Notify(Event::HeroDig,       "Y:       "       +
std::to_string(hero.GetY())       +       "       X:       "       +
std::to_string(hero.GetX()));
    SetState(new EnemyState(this));
    EnemyTurn();
}

void GameManager::UseItem()
{
    Item*              item              =              gameField-
>GetField()[hero.GetY()][hero.GetX()].GetItem();
```

```
    if (item)
    {
        if   (item->GetItemName()   ==   "Chest"   &&   gameField-
>GetField()[hero.GetY()][hero.GetX()].GetType()                ==
CellType::DIGGEDTRAIL)
        {
            itemStrategy.SetStrategy(item);
            itemStrategy.UseItem(hero);
            eventManager.Notify(Event::ItemUse,     "X:     "     +
std::to_string(hero.GetX())       +       ",       Y:       "       +
std::to_string(hero.GetY()) + "(" + ")" + item->GetItemName());
            LogHeroInfo();
        }
        else   if   (item->GetItemName()   ==   "Key"   ||   item-
>GetItemName() == "Heart")
        {
            itemStrategy.SetStrategy(item);
            itemStrategy.UseItem(hero);
            eventManager.Notify(Event::ItemTake,"X:       "       +
std::to_string(hero.GetX())       +       ",       Y:       "       +
std::to_string(hero.GetY()) + "(" + ")" + item->GetItemName());
            eventManager.Notify(Event::HeroGetItem,          item-
>GetItemName());
            gameField-
>GetField()[hero.GetY()][hero.GetX()].DeleteItem();
            LogHeroInfo();
        }

    }
    SetState(new EnemyState(this));
    EnemyTurn();
    item = nullptr;
}

void GameManager::LogHeroInfo()
```

```
{
    std::stringstream buffer;
    buffer << hero;
    LogFile* logFile = new LogFile();
    logFile->PrintLog(buffer.str());
    delete logFile;
}

void GameManager::PlaceEnemies()
{
    TPBehaviour tpBehaviour;
    BeatBehaviour beatBehaviour;
    RobBehaviour robBehaviour;
    beatEnemy = new Enemy<BeatBehaviour>(6, 0);
    gameField->GetField()[6][0].PlaceEnemy(beatEnemy);
    eventManager.Notify(Event::EnemySpawn, "X:0, Y:6(BeatEnemy)");
    robEnemy = new Enemy<RobBehaviour>(8, 4);
    gameField->GetField()[8][4].PlaceEnemy(robEnemy);
    eventManager.Notify(Event::EnemySpawn, "X:4, Y:8(RobEnemy)");
    TPEnemy = new Enemy<TPBehaviour>(2, 10);
    gameField->GetField()[2][10].PlaceEnemy(TPEnemy);
    eventManager.Notify(Event::EnemySpawn, "X:10, Y:2(TPEnemy)");

}



void GameManager::PlaceItems()
{
    ItemChestFactory chest;
    ItemHeartFactory heart;
    ItemKeyFactory key;

    gameField->GetField()[10][1].PlaceItem(key.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 1, Y: 10(KeyItem)");
    gameField->GetField()[10][5].PlaceItem(key.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 5, Y: 10(KeyItem)");
    gameField->GetField()[0][10].PlaceItem(key.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 10, Y: 0(KeyItem)");

    gameField->GetField()[7][1].PlaceItem(heart.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 1, Y:
7(HeartItem)");
    gameField->GetField()[0][3].PlaceItem(heart.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 3, Y:
0(HeartItem)");
    gameField->GetField()[3][8].PlaceItem(heart.createItem());
    eventManager.Notify(Event::ItemSpawn, "X: 8, Y:
3(HeartItem)");

    std::srand(time(NULL));
    int counter = 0;
    int x, y;
    while (counter != 4)
    {
        x = std::rand() % 9 + 1;
        y = std::rand() % 9 + 1;
        if (gameField->GetField()[y][x].GetType() ==
CellType::WALL || gameField->GetField()[y][x].GetItem()) continue;
        gameField->GetField()[y][x].PlaceItem(chest.createItem());
        eventManager.Notify(Event::ItemSpawn, "X: " +
std::to_string(x) + ", Y: " + std::to_string(y) + "(ChestItem)");
        counter++;
    }
}

void GameManager::EndGame()
{
    closeGame = true;
}
```

```
GameState* GameManager::getState()
{
    return gameState;
}

GameManager::~GameManager()
{
    if (displayView)
        delete displayView;
    if (gameState)
        delete gameState;
}

bool GameManager::CheckPath(int moveX, int moveY)
{
    if (hero.GetX() + moveX < 0 || hero.GetX() + moveX ==
FIELD_WIDTH || hero.GetY() + moveY < 0 || hero.GetY() + moveY ==
FIELD_HEIGHT) return false;
    if (gameField->GetField()[hero.GetY() + moveY][hero.GetX() +
moveX].GetType() == CellType::WALL) return false;
    return true;
}

bool GameManager::CheckGrass()
{
    if ((hero.GetX() == 0 && hero.GetY() == 0) || (hero.GetX() ==
FIELD_WIDTH - 1 && hero.GetY() == FIELD_HEIGHT - 1)) return false;
    if (gameField->GetField()[hero.GetY()][hero.GetX()].GetType()
== CellType::DIGGEDTRAIL) return false;
    return true;
}

bool GameManager::CheckItem()
{
    Item*               item                =                 gameField-
>GetField()[hero.GetY()][hero.GetX()].GetItem();
    return item;
}

Memento* GameManager::Save()
{
    GameInfo gameInfo;
    gameInfo.heartsPos.clear();
    gameInfo.chestsPos.clear();
    gameInfo.keysPos.clear();
    gameInfo.openedChestsPos.clear();

    int keysCount = 0;
    int heartsCount = 0;
    int chestsCount = 0;
    int openedChestsCount = 0;

    gameInfo.beatEnemyPos = std::pair<int, int>(beatEnemy->getX(),
beatEnemy->getY());
    gameInfo.robEnemyPos  =  std::pair<int,  int>(robEnemy->getX(),
robEnemy->getY());
    gameInfo.TPEnemyPos   =   std::pair<int,   int>(TPEnemy->getX(),
TPEnemy->getY());
    gameInfo.heroPos     =     std::pair<int,     int>(hero.GetX(),
hero.GetY());
    gameInfo.heroHealth = hero.GetHealthPoints();
    gameInfo.heroKeys = hero.GetKeyCounter();
    gameInfo.heroOpenedChests = hero.GetOpenedChestCounter();
    for (int y = 0; y < FIELD_HEIGHT; y++)
    {
        for (int x = 0; x < FIELD_WIDTH; x++)
        {
            gameInfo.field[y][x]          =          (int)gameField-
>GetField()[y][x].GetType();
```

```
            if      (gameField->GetField()[y][x].GetItem()      &&
gameField->GetField()[y][x].GetItem()->GetItemName() == "Heart")
            {
                gameInfo.heartsPos.push_back(std::pair<int,
int>(x, y));
                heartsCount++;
            }

            if      (gameField->GetField()[y][x].GetItem()      &&
gameField->GetField()[y][x].GetItem()->GetItemName() == "Key")
            {
                gameInfo.keysPos.push_back(std::pair<int,   int>(x,
y));
                keysCount++;
            }

            if      (gameField->GetField()[y][x].GetItem()      &&
gameField->GetField()[y][x].GetItem()->GetItemName() == "Chest")
            {
                gameInfo.chestsPos.push_back(std::pair<int,
int>(x, y));
                chestsCount++;
            }

            if      (gameField->GetField()[y][x].GetItem()      &&
gameField->GetField()[y][x].GetItem()->GetItemName()            ==
"OpenedChest")
            {
                gameInfo.openedChestsPos.push_back(std::pair<int,
int>(x, y));
                openedChestsCount++;
            }

        }
    }
    gameInfo.heartCount = heartsCount;
    gameInfo.keysCount = keysCount;
    gameInfo.chestsCount = chestsCount;
    gameInfo.openedChestsCount = openedChestsCount;

    return new ConcreteMemento(gameInfo);
}

void GameManager::Restore(Memento* memento)
{
    ItemChestFactory chest;
    ItemHeartFactory heart;
    ItemKeyFactory key;

    for (int i = 0; i < FIELD_WIDTH; i++)
    {
        for (int j = 0; j < FIELD_HEIGHT; j++)
        {
            switch (memento->State().field[i][j])
            {
            case          0:                    gameField-
>GetField()[i][j].SetType(CellType::BEGIN); break;
            case          1:                    gameField-
>GetField()[i][j].SetType(CellType::END); break;
            case          2:                    gameField-
>GetField()[i][j].SetType(CellType::WALL); break;
            case          3:                    gameField-
>GetField()[i][j].SetType(CellType::TRAIL); break;
            case          4:                    gameField-
>GetField()[i][j].SetType(CellType::DIGGEDTRAIL); break;
            }
            gameField->GetField()[i][j].DeleteItem();
            gameField->GetField()[i][j].DeleteEnemy();
        }
    }
    TPEnemy->SetPos(memento->State().TPEnemyPos.first,     memento->State().TPEnemyPos.second);
    beatEnemy->SetPos(memento->State().beatEnemyPos.first, memento->State().beatEnemyPos.second);
    robEnemy->SetPos(memento->State().robEnemyPos.first,   memento->State().robEnemyPos.second);
    hero.SetPos(memento->State().heroPos.first,            memento->State().heroPos.second);
    hero.SetHealth(memento->State().heroHealth);
    hero.SetKeyCounter(memento->State().heroKeys);
    hero.SetOpenedChestCounter(0);


    for (int i = 0; i < memento->State().heartCount; i++)
    {
        gameField->GetField()[memento->State().heartsPos[i].second][memento->State().heartsPos[i].first].PlaceItem(heart.createItem());
        eventManager.Notify(Event::ItemSpawn,      "X:      "      + std::to_string(memento->State().heartsPos[i].first)  +  ",  Y:  "  + std::to_string(memento->State().heartsPos[i].second)              + "(HeartItem)");
    }
    for (int i = 0; i < memento->State().keysCount; i++)
    {
        gameField->GetField()[memento->State().keysPos[i].second][memento->State().keysPos[i].first].PlaceItem(key.createItem());
        eventManager.Notify(Event::ItemSpawn,      "X:      "      + std::to_string(memento->State().keysPos[i].first)   +",   Y:   "   + std::to_string(memento->State().keysPos[i].second)+ "(KeyItem)");
    }
    for (int i = 0; i < memento->State().chestsCount; i++)
    {
        gameField->GetField()[memento-
>State().chestsPos[i].second][memento-
>State().chestsPos[i].first].PlaceItem(chest.createItem());
        eventManager.Notify(Event::ItemSpawn,      "X:      "      +
std::to_string(memento->State().chestsPos[i].first)  +  ",  Y:  "  +
std::to_string(memento->State().chestsPos[i].second)               +
"(ChestItem)");
    }

    for (int i = 0; i < memento->State().openedChestsCount; i++)
    {
        gameField->GetField()[memento-
>State().openedChestsPos[i].second][memento-
>State().openedChestsPos[i].first].PlaceItem(chest.createItem());
        eventManager.Notify(Event::ItemSpawn,      "X:      "      +
std::to_string(memento->State().openedChestsPos[i].first)  +  ",  Y:
"  +  std::to_string(memento->State().openedChestsPos[i].second)  +
"(OpenedChestItem)");
        itemStrategy.SetStrategy(gameField->GetField()[memento-
>State().openedChestsPos[i].second][memento-
>State().openedChestsPos[i].first].GetItem());
        itemStrategy.UseItem(hero);
    }
    LogHeroInfo();

    gameField->GetField()[memento-
>State().beatEnemyPos.second][memento-
>State().beatEnemyPos.first].PlaceEnemy(beatEnemy);
    eventManager.Notify(Event::EnemySpawn,       "X:       "       +
std::to_string(memento->State().beatEnemyPos.first)  +  ",  Y:  "  +
std::to_string(memento->State().beatEnemyPos.second)               +
"(BeatEnemy)");
    gameField->GetField()[memento-
>State().TPEnemyPos.second][memento-
>State().TPEnemyPos.first].PlaceEnemy(TPEnemy);
```

```
    eventManager.Notify(Event::EnemySpawn,       "X:       "       +
std::to_string(memento->State().TPEnemyPos.first)  +  ",  Y:  "  +
std::to_string(memento->State().TPEnemyPos.second) + "(TPEnemy)");
    gameField->GetField()[memento-
>State().robEnemyPos.second][memento-
>State().robEnemyPos.first].PlaceEnemy(robEnemy);
    eventManager.Notify(Event::EnemySpawn,       "X:       "       +
std::to_string(memento->State().robEnemyPos.first)  +  ",  Y:  "  +
std::to_string(memento->State().robEnemyPos.second)               +
"(RobыEnemy)");
    loadSaveObserver->Update("");
}
```

GameManager.h:

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include <SFML/Graphics.hpp>
#include "../GameField/Field.h"
#include "../Display/Display.h"
#include "iostream"
#include "../Hero/Hero.h"
#include "../Items/ItemStrategy.h"
#include "../Log/LogFile.h"
#include "../Log/EventManager.h"
#include "../Enemy/Enemy.h"
#include "../Enemy/BeatBehaviour.h"
#include "../Enemy/RobBehaviour.h"
#include "../Enemy/TPBehaviour.h"
#include "GameState.h"
#include "HeroState.h"
#include "../Save/ConcreteMemento.h"

#include <sstream>
#include <vector>
```

```
class GameManager
{
private:
	friend class Controller;

	ItemStrategy itemStrategy;

	GameState* gameState = nullptr;

	EventManager eventManager;
	ItemUseObserver* itemUseObserver = nullptr;
	ItemTakeObserver* itemTakeObserver = nullptr;
	ItemSpawnObserver* itemSpawnObserver = nullptr;
	HeroDigObserver* heroDigObserver = nullptr;
	HeroGetItemObserver* heroGetItemObserver = nullptr;
	HeroMovingObserver* heroMovingObserver = nullptr;
	EnemyMovedObserver* enemyMovingObserver = nullptr;
	EnemySpawnedObserver* enemySpawnObserver = nullptr;
	EnemyHeroMeetObserver* enemyHeroMeetObserver = nullptr;
	SaveObserver* saveObserver = nullptr;
	LoadSaveObserver* loadSaveObserver = nullptr;

	Enemy<BeatBehaviour>* beatEnemy = nullptr;
	Enemy<RobBehaviour>* robEnemy = nullptr;
	Enemy<TPBehaviour>* TPEnemy = nullptr;
	Display* displayView = nullptr;

	Hero hero;

	Field* gameField = nullptr;
	bool closeGame = false;

	void LogHeroInfo();
	void PlaceItems();
	void PlaceEnemies();
public:
	GameManager() = default;
	void GameSaved();
	void SetState(GameState* newState);
	void HeroMoving(int moveX, int moveY);
	void EnemyMoving();
	void DigGrass();
	void UseItem();
	void StartGame();
	void EndGame();
	void EnemyTurn();
	bool CheckPath(int x, int y);
	bool CheckGrass();
	bool CheckItem();
	Memento* Save();
	void Restore(Memento* memento);
	GameState* getState();
	~GameManager();
};

#endif
```

Controller.cpp

```
#include "Controller.h"

Controller::Controller()
{
    sf::RenderWindow            window(sf::VideoMode(WINDOW_WIDTH,
WINDOW_HEIGHT), "Gismos");

    gameManager = new GameManager();

    closeGameCommand = new CloseGameCommand(gameManager);
    digGrassCommand = new DigGrassCommand(gameManager);
    moveDownCommand = new MoveDownCommand(gameManager);
    moveLeftCommand = new MoveLeftCommand(gameManager);
    moveRightCommand = new MoveRightCommand(gameManager);
    moveUpCommand = new MoveUpCommand(gameManager);
    startGameCommand = new StartGameCommand(gameManager);
    takeItemCommand = new TakeItemCommand(gameManager);
    careTaker = new CareTaker(gameManager);
    while (window.isOpen())
    {

        sf::Event event;
        while (window.pollEvent(event))
        {
            if  (event.type  ==  sf::Event::Closed  ||  gameManager-
>closeGame)
                window.close();

            if (!isGameWin && isGameStarted && !isGameLost)
            {
                if (event.type == sf::Event::KeyPressed)
                {
                    switch (event.key.code)
                    {
                    case   sf::Keyboard::S:   careTaker->BackUp();
break;
                    case   sf::Keyboard::Left:   moveLeftCommand-
>execute(); break;
                    case  sf::Keyboard::Right:  moveRightCommand-
>execute(); break;
                    case     sf::Keyboard::Up:     moveUpCommand-
>execute(); break;
                    case   sf::Keyboard::Down:   moveDownCommand-
>execute(); break;
                    case   sf::Keyboard::Space:   digGrassCommand-
>execute(); break;
```

```
                    case     sf::Keyboard::E:     takeItemCommand-
>execute();  break;
                    case    sf::Keyboard::L:    careTaker->Load();
break;
                    default: break;
                    }
                }
            }
            if    (event.type    ==    sf::Event::KeyPressed    &&
event.key.code == sf::Keyboard::Enter && !isGameStarted)
            {
                startGameCommand->execute();
                isGameStarted = !isGameStarted;

            }
            if    (event.type    ==    sf::Event::KeyPressed    &&
event.key.code == sf::Keyboard::Escape)
            {
                closeGameCommand->execute();
            }

        }

        if (!isGameStarted)
        {
            gameManager->displayView->DisplayStartWindow(window);
        }
        else if (isGameWin)
        {
            gameManager->displayView->DisplayWinWindow(window);

        }
        else if (isGameLost)
        {
            gameManager->displayView->DisplayLostWindow(window);
        }
        else
        {
            IsGameEnd();
            gameManager->displayView-
>DisplayHeroInformation(window, gameManager->hero);
            gameManager->displayView->DisplayField(window);
            gameManager->displayView-
>DisplayItemsAndEnemies(window);
            gameManager->displayView->DisplayHero(window,
gameManager->hero);
        }
        window.display();
        window.clear();

    }
     std::cout << "\n";
}
```

```
void Controller::IsGameEnd()
{
    if (gameManager->hero.GetHealthPoints() == 0)
    {
        isGameLost = true;
    }
    if    (gameManager->hero.GetOpenedChestCounter()    ==    4    &&
gameManager->hero.GetX() == 10 && gameManager->hero.GetY() == 10)
    {
        isGameWin = true;
    }
    if (gameManager->hero.GetX() == 10 && gameManager->hero.GetY()
== 10 && gameManager->hero.GetKeyCounter() == 3)
    {
        isGameWin = true;
    }
}


Controller::~Controller()
{
    if (gameManager)
        delete gameManager;
    if (closeGameCommand)
        delete closeGameCommand;
    if (digGrassCommand)
        delete digGrassCommand;
    if (moveDownCommand)
        delete moveDownCommand;
    if (moveLeftCommand)
        delete moveLeftCommand;
    if (moveRightCommand)
        delete moveRightCommand;
    if (moveUpCommand)
        delete moveUpCommand;
    if (startGameCommand)
        delete startGameCommand;
    if (takeItemCommand)
        delete takeItemCommand;
}
```

Controller.h

```
#ifndef STARTGAME_H
#define STARTGAME_H
```

```
#include "GameManager.h"
#include "../Commands/CloseGameCommand.h"
#include "../Commands/DigGrassCommand.h"
#include "../Commands/MoveDownCommand.h"
#include "../Commands/MoveLeftCommand.h"
#include "../Commands/MoveRightCommand.h"
#include "../Commands/MoveUpCommand.h"
#include "../Commands/StartGameCommand.h"
#include "../Commands/TakeItemCommand.h"
#include "../Save/CareTaker.h"


class Controller
{
public:
	Controller();
	~Controller();

private:

	GameManager* gameManager = nullptr;
	CareTaker* careTaker = nullptr;
	bool isGameWin = false;
	bool isGameLost = false;
	bool isGameStarted = false;
	void IsGameEnd();

	CloseGameCommand* closeGameCommand = nullptr;
	DigGrassCommand* digGrassCommand = nullptr;
	MoveDownCommand* moveDownCommand = nullptr;
	MoveLeftCommand* moveLeftCommand = nullptr;
	MoveRightCommand* moveRightCommand = nullptr;
	MoveUpCommand* moveUpCommand = nullptr;
	StartGameCommand* startGameCommand = nullptr;
```

```
     TakeItemCommand* takeItemCommand = nullptr;
};


#endif
```

Cell.cpp:

```
#include "Cell.h"
#include <iostream>
#include "../Enemy/Enemy.h"

Item* Cell::GetItem()
{
     return itemOnCell;
}

void Cell::SetPosition(int x, int y)
{
     this->x = x;
     this->y = y;
}

void Cell::SetType(CellType cellType)
{
     this->cellType = cellType;
}

void Cell::DeleteItem()
{
     delete itemOnCell;
     itemOnCell = nullptr;
}

void Cell::DeleteEnemy()
{
     if(enemyRobOnCell)
```

```
		enemyRobOnCell = nullptr;
	if(enemyTPOnCell)
		enemyTPOnCell = nullptr;
	if(enemyBeatOnCell)
		enemyBeatOnCell = nullptr;
}

CellType Cell::GetType()
{
	return cellType;
}



Cell::~Cell()
{
	if(itemOnCell)
		delete itemOnCell;
	if (enemyRobOnCell)
		delete enemyRobOnCell;
	if (enemyTPOnCell)
		delete enemyTPOnCell;
	if (enemyBeatOnCell)
		delete enemyBeatOnCell;
}

void Cell::PlaceItem(Item* tempItem)
{
	this->itemOnCell = tempItem;
}

void Cell::PlaceEnemy(Enemy<BeatBehaviour>* enemy)
{
	this->enemyBeatOnCell = enemy;
}
```

```
void Cell::PlaceEnemy(Enemy<TPBehaviour>* enemy)
{
     this->enemyTPOnCell = enemy;
}

void Cell::PlaceEnemy(Enemy<RobBehaviour>* enemy)
{
     this->enemyRobOnCell = enemy;
}
```

Cell.h:

```
#ifndef CELL_H
#define CELL_H



#include "../Items/Item.h"
#include "../Items/Key/ItemKey.h"
#include "../Items/Chest/ItemChest.h"
#include "../Items/Heart/ItemHeart.h"
#include "../Items/Key/ItemKeyFactory.h"
#include "../Items/Chest/ItemChestFactory.h"
#include "../Items/Heart/ItemHeartFactory.h"
#include "../Enemy/BeatBehaviour.h"
#include "../Enemy/TPBehaviour.h"
#include "../Enemy/RobBehaviour.h"


template<class T>class Enemy;

enum class CellType
{
     BEGIN = 0,
     END = 1,
     WALL = 2,
     TRAIL = 3,
     DIGGEDTRAIL = 4
};

class Cell
{

private:

     //Координаты относительно массива клеток
     int x, y;
```

```
	//Поля класса, хранящие информацию о клетке и том, что на ней находится
	CellType cellType;
	Item* itemOnCell = nullptr;
	Enemy<RobBehaviour>* enemyRobOnCell = nullptr;
	Enemy<TPBehaviour>* enemyTPOnCell = nullptr;
	Enemy<BeatBehaviour>* enemyBeatOnCell = nullptr;




public:

	bool isHeroOnCell = false;

	Cell() : cellType(CellType::TRAIL){}
	~Cell();

	void PlaceItem(Item* tempItem);
	void PlaceEnemy(Enemy<RobBehaviour>* enemy);
	void PlaceEnemy(Enemy<TPBehaviour>* enemy);
	void PlaceEnemy(Enemy<BeatBehaviour>* enemy);

	void DeleteEnemy();
	Enemy<RobBehaviour>* GetRobEnemy() { return enemyRobOnCell; }
	Enemy<TPBehaviour>* GetTPEnemy() { return enemyTPOnCell; }
	Enemy<BeatBehaviour>* GetBeatEnemy() { return enemyBeatOnCell; }

	void SetPosition(int x, int y);

	//Функция установки информации о клетке
	void SetType(CellType cellType);

	void DeleteItem();

	//Функции получения информации о клетке
	Item* GetItem();
	CellType GetType();
};

#endif
```

Enemy.cpp:

```
#include "Enemy.h"
```

Enemy.h:

```
#ifndef ENEMY_H

#define ENEMY_H
```

```
#include "../Hero/Hero.h"
#include "../GameField/Field.h"


template<class T>
class Enemy
{
private:
	T behaviour;
	int x = 0, y = 0;
public:
	bool goUp = false;
	Enemy(int y, int x) : x(x), y(y) {}
	int getX() { return x; }
	int getY() { return y; }

	void SetPos(int x, int y)
	{
		this->x = x;
		this->y = y;
	}

	void Move()
	{
		int dy;
		if (goUp) dy = -1;
		else dy = 1;
		Field* gameField = Field::GetInstance();
		if (y+dy < 0 || y+dy >= 11 || gameField->GetField()[y +
dy][x].GetType() == CellType::WALL)
		{
			y -= dy;
			goUp = !goUp;
			return;
		}
		y += dy;
	}
	void operator-(Hero& hero)
	{
		behaviour.Action(hero);
	}

};
#endif
```

CareTaker.cpp

```
#include "CareTaker.h"

CareTaker::CareTaker(GameManager* gameManager)
{
	this->gameManager = gameManager;
	saveFile = new SaveFile();
}

void CareTaker::BackUp()
{
	std::string resStr = "";
	if (this->memento)
		delete this->memento;
	try
	{
		this->memento = this->gameManager->Save();
		if (this->memento == nullptr)
			throw SaveException();
	}
	catch (SaveException e)
	{
		std::cerr << e.what();
		exit(0);
	}
	catch (std::bad_alloc e)
	{
		std::cerr << "Out of memory!\n";
		exit(0);
	}


	resStr += std::to_string(memento->State().beatEnemyPos.first)
+ " " + std::to_string(memento->State().beatEnemyPos.second) +
"\n"
		+ std::to_string(memento->State().TPEnemyPos.first) + " "
+ std::to_string(memento->State().TPEnemyPos.second) + "\n"
		+ std::to_string(memento->State().robEnemyPos.first) + "
" + std::to_string(memento->State().robEnemyPos.second) + "\n"
		+ std::to_string(memento->State().heroPos.first) + " " +
std::to_string(memento->State().heroPos.second) + "\n"
		+ std::to_string(memento->State().heroHealth) + "\n"
		+ std::to_string(memento->State().heroKeys) + "\n"
		+ std::to_string(memento->State().heroOpenedChests) +
"\n";
	resStr += std::to_string(memento->State().heartCount) + "\n";
	for (int i = 0; i < memento->State().heartsPos.size(); i++)
	{
		resStr += std::to_string(memento-
>State().heartsPos[i].first) + " " + std::to_string(memento-
>State().heartsPos[i].second) + "\n";
	}
	resStr += std::to_string(memento->State().chestsCount) +
"\n";
	for (int i = 0; i < memento->State().chestsPos.size(); i++)
	{
		resStr += std::to_string(memento-
>State().chestsPos[i].first) + " " + std::to_string(memento-
>State().chestsPos[i].second) + "\n";
	}
```

```
     resStr += std::to_string(memento->State().keysCount) + "\n";
     for (int i = 0; i < memento->State().keysPos.size(); i++)
     {
          resStr              +=              std::to_string(memento-
>State().keysPos[i].first)   +   "   "   +   std::to_string(memento-
>State().keysPos[i].second) + "\n";
     }

     resStr  +=  std::to_string(memento->State().openedChestsCount)
+ "\n";
     for  (int  i  =  0;  i  <  memento->State().openedChestsPos.size();
i++)
     {
          resStr              +=              std::to_string(memento-
>State().openedChestsPos[i].first)  +  "  "  +  std::to_string(memento-
>State().openedChestsPos[i].second) + "\n";
     }
     for (int i = 0; i < FIELD_HEIGHT; i++)
     {
          for (int j = 0; j < FIELD_WIDTH; j++)
          {
               resStr           +=           std::to_string(memento-
>State().field[i][j]) + " ";
          }
          resStr += "\n";
     }
     saveFile->ChangeMode(Mode::SAVE);
     saveFile->SaveGame(resStr);
     gameManager->GameSaved();
}

void CareTaker::Load()
{
     saveFile->ChangeMode(Mode::LOAD);
     Parser(saveFile->LoadGame());
```

```
	this->gameManager->Restore(this->memento);

}

void CareTaker::Parser(std::stringstream str)
{
	try
	{

		if (this->memento == nullptr)
		{
			GameInfo state;
			this->memento = new ConcreteMemento(state);
		}

		memento->State().beatEnemyPos = ParseCoords(str);
		memento->State().TPEnemyPos = ParseCoords(str);
		memento->State().robEnemyPos = ParseCoords(str);
		memento->State().heroPos = ParseCoords(str);
		memento->State().heroHealth = ParseValue(str);
		memento->State().keysCount = ParseValue(str);
		memento->State().openedChestsCount = ParseValue(str);
		memento->State().heartCount = ParseValue(str);
		memento->State().heartsPos.empty();
		for (int i = 0; i < memento->State().heartCount; i++)
			memento-
>State().heartsPos.push_back(ParseCoords(str));
		memento->State().chestsCount = ParseValue(str);
		memento->State().chestsPos.empty();
		for (int i = 0; i < memento->State().chestsCount; i++)
			memento-
>State().chestsPos.push_back(ParseCoords(str));
		memento->State().keysCount = ParseValue(str);
		memento->State().keysPos.empty();
		for (int i = 0; i < memento->State().keysCount; i++)
```

```
                memento-
>State().keysPos.push_back(ParseCoords(str));
            memento->State().openedChestsCount = ParseValue(str);
            memento->State().openedChestsPos.empty();
            for (int i = 0; i < memento->State().openedChestsCount;
i++)
                memento-
>State().openedChestsPos.push_back(ParseCoords(str));

            std::string temp;
            std::string temp2;
            for (int i = 0; i < FIELD_HEIGHT; i++)
            {
                temp2 = "";
                std::getline(str, temp);
                std::stringstream ss(temp);
                for (int j = 0; j < FIELD_WIDTH; j++)
                {
                    ss >> memento->State().field[i][j];
                    temp2         +=         std::to_string(memento-
>State().field[i][j]) + " ";
                    if   (memento->State().field[i][j]   >   4   ||
memento->State().field[i][j] < 0)
                    {
                        throw LoadSaveException();
                    }
                }
                if (temp != temp2)
                {
                    throw LoadSaveException();
                }
            }
            //memento->State().print();
        }
        catch (LoadSaveException e)
```

```
	{
		std::cerr << e.what();
		exit(0);
	}

}

std::pair<int,int> CareTaker::ParseCoords(std::stringstream& str)
{
	std::string tempString;
	std::getline(str, tempString);

	std::pair<int, int> coords;
	std::stringstream ss(tempString);
	int coord;

	ss >> coord;
	coords.first = coord;

	ss >> coord;
	coords.second = coord;

	if      (std::to_string(coords.first)      +      "      "      +
std::to_string(coords.second) != tempString)
	{
		throw LoadSaveException();
	}

	if (coords.first < 0 || coords.first > 10 || coords.second <
0 || coords.second > 10)
	{
		throw LoadSaveException();
	}
	return coords;

}

int CareTaker::ParseValue(std::stringstream& str)
{
	std::string tempString;
	std::getline(str, tempString);
	int value = std::stoi(tempString);
	if (std::to_string(value) != tempString)
	{
		throw LoadSaveException();
	}
	return value;
}
```

CareTaker.h

```
#ifndef CARETAKER_H
#define CARETAKER_H

#include "Memento.h"
#include "../GameManager/GameManager.h"
#include "SaveFile.h"
#include "../Exceptions/LoadSaveException.h"
#include "../Exceptions/SaveException.h"

class CareTaker
{
private:
	Memento* memento = nullptr;
	GameManager* gameManager;
	SaveFile* saveFile;
public:
	CareTaker(GameManager* gameManager);
	void BackUp();
	void Load();
	void Parser(std::stringstream str);
	void FirstLoading();
	std::pair<int, int> ParseCoords(std::stringstream& str);
	int ParseValue(std::stringstream& str);
};

#endif
```

ConcreteMemento.cpp

```
#include "ConcreteMemento.h"

ConcreteMemento::ConcreteMemento(GameInfo state)
{
	this->state = state;
}

void ConcreteMemento::ChangeState(GameInfo state)
{
	this->state = state;
}

GameInfo& ConcreteMemento::State()
{
	return state;
}
```

ConcreteMemento.h

```
#ifndef CONCRETEMEMENTO_H
#define CONCRETEMEMENTO_H

#include "Memento.h"

class ConcreteMemento:public Memento
{
private:
	GameInfo state;
public:
     ConcreteMemento(GameInfo state);
     GameInfo& State();
     void ChangeState(GameInfo state);
};

#endif
```

GameInfo.cpp

```
#include "GameInfo.h"

void GameInfo::print()
{
     std::cout << "TPEnemyPos: X: " << TPEnemyPos.first << ", Y: "
<< TPEnemyPos.second << '\n';
     std::cout << "robEnemyPos: X: " << robEnemyPos.first << ", Y:
" << robEnemyPos.second << '\n';
     std::cout << "beatEnemyPos: X: " << beatEnemyPos.first << ",
Y: " << beatEnemyPos.second << '\n';
     std::cout << "heroPos: X: " << heroPos.first << ", Y: " <<
heroPos.second << '\n';
     std::cout << "heroHealth: " << heroHealth << '\n';
     std::cout << "heroKeys: " << heroKeys << '\n';
     std::cout << "heroOpenedChests: " << heroOpenedChests <<
'\n';
     std::cout << "heartCount: " << heartCount << '\n';
     std::cout << "heartsPos: " << '\n';
     for (int i = 0; i < heartCount; i++)
          std::cout << "\tX: " << heartsPos[i].first << "Y: " <<
heartsPos[i].second << '\n';



     std::cout << "keysCount: " << keysCount << '\n';
     std::cout << "keysPos: " << '\n';
     for (int i = 0; i < keysCount; i++)
          std::cout << "\tX: " << keysPos[i].first << "Y: " <<
keysPos[i].second << '\n';
```

```
	std::cout << "heartCount: " << chestsCount << '\n';
	std::cout << "chestsPos: " << '\n';
	for (int i = 0; i < chestsCount; i++)
		std::cout << "\tX: " << chestsPos[i].first << "Y: " <<
chestsPos[i].second << '\n';

	std::cout << "heartCount: " << openedChestsCount << '\n';
	std::cout << "openedChestsPos: " << '\n';
	for (int i = 0; i < openedChestsCount; i++)
		std::cout << "\tX: " << openedChestsPos[i].first << "Y: "
<< openedChestsPos[i].second << '\n';
	for (int i = 0; i < 11; i++)
	{
		for (int j = 0; j < 11; j++)
		{
			std::cout << field[i][j] << ' ';
		}
		std::cout << '\n';
	}
}
```

GameInfo.h

```
#ifndef GAMEINFO_H
#define GAMEINFO_H

#include <map>
#include <vector>
#include <iostream>

class GameInfo
{
public:
	std::pair<int, int> TPEnemyPos;
	std::pair<int, int> robEnemyPos;
	std::pair<int, int> beatEnemyPos;
```

```
     int heroHealth;
     int heroOpenedChests;
     int heroKeys;
     std::pair<int, int> heroPos;
     int heartCount;
     std::vector<std::pair<int, int>> heartsPos;
     int keysCount;
     std::vector<std::pair<int, int>> keysPos;
     int chestsCount;
     std::vector<std::pair<int, int>> chestsPos;
     int openedChestsCount;
     std::vector<std::pair<int, int>> openedChestsPos;
     int field[11][11];
     void print();
};

#endif
```

Memento.cpp

```
#include "Memento.h"
```

Memento.h

```
#ifndef MEMENTO_H
#define MEMENTO_H

#include "GameInfo.h"

class Memento
{
public:
     virtual GameInfo& State() = 0;
     virtual void ChangeState(GameInfo state) = 0;
};

#endif
```

SaveFile.cpp

```
#include "SaveFile.h"
```

```
SaveFile::SaveFile()
{
	try
	{
		file = new std::ofstream;
		file-
>open("C:/Users/nikei/source/repos/OOPLab/OOPLab/src/Save/saver.tx
t", std::ios::in);
		if (!file->is_open())
			throw SaveException();
	}
	catch (std::bad_alloc e)
	{
		std::cerr << "Out of memory!\n";
		ClearMemory();
		exit(0);
	}
	catch (SaveException)
	{
		std::cerr << "Can't open file.\n";
		ClearMemory();
		exit(0);
	}

}

SaveFile::~SaveFile()
{
	ClearMemory();
}

void SaveFile::ChangeMode(Mode mode)
{
	this->mode = mode;
	try
	{
		if (this->mode == Mode::SAVE)
		{
			file->close();
			file-
>open("C:/Users/nikei/source/repos/OOPLab/OOPLab/src/Save/saver.tx
t", std::ios::out);
		}
		else if (this->mode == Mode::LOAD)
		{
			file->close();
			file-
>open("C:/Users/nikei/source/repos/OOPLab/OOPLab/src/Save/saver.tx
t", std::ios::in);
		}
		if (!file->is_open())
			throw SaveException();
	}
	catch (SaveException)
	{
		std::cerr << "Can't open file.\n";
		ClearMemory();
		exit(0);
	}

}

std::stringstream SaveFile::LoadGame()
{
	std::stringstream sstr;
	sstr << file->rdbuf();
	return sstr;
}
```

```
void SaveFile::SaveGame(std::string saveMessage)
{
	*file << saveMessage;
}

void SaveFile::LoadingEnded()
{
	try
	{
		file->close();
		file-
>open("C:/Users/nikei/source/repos/OOPLab/OOPLab/src/Save/saver.tx
t", std::ios::out || std::ios::trunc);
		if (!file->is_open())
			throw SaveException();
	}
	catch (SaveException)
	{
		std::cerr << "Can't open file.\n";
		ClearMemory();
		exit(0);
	}
}

void SaveFile::ClearMemory()
{
	if (file->is_open())
		file->close();
	if (file)
		delete file;
}
```

SaveFile.h

```
#ifndef SAVEFILE_H
#define SAVEFILE_H
```

```
#include <fstream>
#include <iostream>
#include <string>
#include <sstream>
#include "../Exceptions/SaveException.h"
#include "../Exceptions/LoadSaveException.h"

enum class Mode
{
	SAVE,
	LOAD
};

class SaveFile
{
private:
	std::ofstream* file = nullptr;
	Mode mode = Mode::SAVE;
public:
	SaveFile();
	~SaveFile();
	void ChangeMode(Mode mode);
	void SaveGame(std::string saveMessage);
	void LoadingEnded();
	std::stringstream LoadGame();
	void ClearMemory();
};

#endif
```

LoadSaveException.cpp

```
#include "LoadSaveException.h"
```

LoadSaveException.h

```
#ifndef LOADSAVEEXCEPTION_H
#define LOADSAVEEXCEPTION_H
```

```
class LoadSaveException
{
public:
	const char* what() const throw()
	{
		return "An exception occurred while loading the save.
File is corrupted.";
	}
};

#endif
```

SaveException.cpp

```
#include "SaveException.h"
```

SaveException.h

```
#ifndef SAVEEXCEPTION_H
#define SAVEEXCEPTION_H

class SaveException
{
public:
	const char* what() const throw()
	{
		return "An exception occurred while saving.\n";
	}
};

#endif
```

LoadSaveObserver.cpp

```
#include "LoadSaveObserver.h"

void LoadSaveObserver::Update(std::string log)
{
	LogFile* logger = new LogFile();
	logger->PrintLog("Last Save Loaded.");
	std::cout << "Last Save Loaded.\n";
```

```
     delete logger;
}
```

LoadSaveObserver.h

```
#ifndef LOADSAVEOBSERVER_H
#define LOADSAVEOBSERVER_H

#include "Observer.h"

class LoadSaveObserver:public Observer
{
public:
     void Update(std::string log);
};

#endif
```

SaveObserver.cpp

```
#include "SaveObserver.h"

void SaveObserver::Update(std::string log)
{
     LogFile* logger = new LogFile();
     logger->PrintLog("Game Saved.");
     std::cout << "Game Saved.\n";
     delete logger;
}
```

SaveObserver.h

```
#ifndef SAVEOBSERVER_H
#define SAVEOBSERVER_H

#include "Observer.h"

class SaveObserver:public Observer
{
public:
	void Update(std::string log);
};

#endif
```

Hero.cpp

```
#include "Hero.h"

Hero::Hero()
{
	this->xPos = 0;
	this->yPos = 0;
	this->healthPoints = 5;
}

void Hero::Damage()
{
	if(healthPoints > 0)
		this->healthPoints--;
}

void Hero::Rob()
{
	if (keyCounter > 0)
		keyCounter--;
}

void Hero::ChangePosition(int x, int y)
{
	this->xPos += x;
	this->yPos += y;
}

void Hero::AddKey()
{
	keyCounter++;
}

void Hero::AddOpenedChest()
{
	openedChestCounter++;
}

void Hero::AddHealtPoints()
{
	healthPoints++;
}

int Hero::GetHealthPoints()
{
	return healthPoints;
}

int Hero::GetX()
{
	return xPos;
}

int Hero::GetY()
{
	return yPos;

}

void Hero::SetPos(int x, int y)
{
	this->xPos = x;
	this->yPos = y;
}

void Hero::SetHealth(int health)
```

```
{
	this->healthPoints = health;
}

void Hero::SetOpenedChestCounter(int chest)
{
	this->openedChestCounter = chest;
}

void Hero::SetKeyCounter(int key)
{
	this->keyCounter = key;
}

std::ostream& operator<<(std::ostream& out, const Hero &hero)
{
	out << "Info about hero: "
		<< hero.openedChestCounter << " OpenedChests, "
		<< hero.keyCounter << " Keys, "
		<< hero.healthPoints << " Health."
		<< "Hero position: "
		<< "X: " << hero.xPos
		<< ", Y: " << hero.yPos;
	return out;
```

Hero.h

```
#ifndef HERO_H
#define HERO_H

#include <fstream>
#include <iostream>

class Hero
{
private:
	int healthPoints;
```

```
     int xPos, yPos;
     int openedChestCounter = 0;
     int keyCounter = 0;
public:
     Hero();
     int GetHealthPoints();
     int GetX();
     int GetY();

     void SetPos(int x, int y);
     void SetHealth(int health);
     void SetOpenedChestCounter(int chest);
     void SetKeyCounter(int key);

     void Damage();
     void Rob();
     void ChangePosition(int x, int y);
     void AddKey();
     void AddOpenedChest();
     int GetKeyCounter() { return keyCounter; }
     void AddHealtPoints();
     int GetOpenedChestCounter() { return openedChestCounter; }
     friend std::ostream& operator<<(std::ostream& out, const Hero
&hero);
};

#endif
```