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: Полиморфизм

Студент гр. 2383 _____ Борисов И.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2023

Цель работы.

Целью данной лабораторной работы является создание интерфейса событий, а также реализация нескольких событий, а также взаимодействие с ними. Также требуется создать генератор поля.

Задание.

а) Создать интерфейс игрового события. Интерфейс должен обеспечивать срабатывание события когда игрок наступает на клетку.

б) Реализовать интерфейс игрового события тремя конкретными событиями. Одно событие должно положительно влиять на характеристики игрока, второе должно негативно влиять на характеристики игрока, третье изменять координаты игрока на поле. При желании можно реализовать больше событий и/или события меняющие само поле (например, делать из непроходимой клетки проходимую).

в) В классе управления игроком добавить проверку на наличие события на клетке, если событие присутствует, то оно должно сработать. Срабатывание должно происходить через интерфейс события, и не должно быть никаких проверок на тип события (реализация через динамический полиморфизм)

г) Создать класс создающий поле. Предусмотреть возможность создания 2 разных уровней. По желанию можно сделать случайную генерацию уровней. Должно гарантироваться, что игрок может дойти от входа до выхода.

Примечания:

События должны быть такими, чтобы был сценарий проигрыша игрока.

В событиях и клетках не должно быть полей сообщающих информацию о типе события

Основные теоретические положения.

Паттерн Factory - это паттерн создания объектов в объектно-ориентированном программировании (ООП), который предоставляет возможность создавать объекты в суперклассе, но позволяет подклассам изменять тип создаваемых объектов. По сути, фабрика - это объект или метод, который возвращает объекты изменяющегося прототипа или класса. Это подпрограмма, возвращающая "новый" объект, и основа для ряда родственных паттернов проектирования программного обеспечения.

Алгоритм поиска A* (произносится "A-звезда") - алгоритм обхода графов и поиска путей, широко используемый в информатике благодаря своей полноте, оптимальности и оптимальной эффективности. A* является расширением алгоритма Дейкстры, который достигает лучшей производительности за счет использования эвристики для управления поиском. В отличие от алгоритма Дейкстры, который находит дерево кратчайших путей от заданного источника до всех возможных целей, A* находит только кратчайший путь от заданного источника до заданной цели. Это делает его особенно полезным для решения задач, в которых цель известна заранее.

Выполнение работы.

Был создан интерфейс ивента *EventInterface* у которого есть виртуальные методы сравнения, копирования, а также метод *virtual void trigger(EntityHandler *handler) const = 0* который отвечает за взаимодействие события с игроком.

Помимо интерфейса события были созданы такие события как: Potion (добавляет игроку 50 хп и увеличивает атаку на 10), ShieldKit (добавляет игроку 50 брони), Star (добавляет игроку 100 очков), Spikes (отнимает у игрока 50 % брони, если брони меньше 10, то отнимает 20 хп), RandomMine (отбрасывает игрока в случайном направлении на 1-3 клетки), Key (выдаёт игроку ключ для открытия двери), Door (блокирует данную клетку, т. е. не даёт туда подвинуться пока игрок не получит нужный ключ).

Создан интерфейс MapSubject. У него есть методы: *virtual bool can_move(EntityHandler *caller, const Position &position) const = 0* — позволяет узнать можно ли подвинуться на данную клетку или нет, *virtual Cell &get_cell(const Position &point) const = 0* - позволяет получить доступ к клетке по ссылке, *virtual std::vector<Position> find_route(EntityHandler *caller, const Position &begin, const Position &end) const = 0* — позволяет найти кратчайший путь (не учитываются ивенты на карте, т. е. потенциально путь может быть непроходимым).

Класс Map унаследован от MapSubject для удобного взаимодействия с игроком.

Класс PlayerHandler теперь наследуется от класса MapObserver, перегружены методы *void register_observer(MapSubject* observer) override, void remove_observer(MapSubject* observer) override.* Таким образом игрок получает возможность взаимодейтвися с картой без прямого доступа к реализации всей карты. Также добавлены методы добавления и получения ключей игрока. Взаимодействие с событием на клетке происходит при помощи метода *trigger* данного события без проверки типа (реализуется динамический полиморфизм).

В класс Vector2 добален метод *double distance(const Vector2 &other) const* который вычисляет расстояние между двумя точками.

Создан класс Timer, который позволяет замерять время выполнения блока кода (в конструкторе засекается время, а в деструкторе выводится разница).

Класс Random позволяет генерировать случайные значения. Метод *DIRECTION pick_direction() const* позволяют выбрать произвольное направление, метод *EventInterface *pick_event(EVENT_GROUP group) const* позволяет выбрать ивент соответствующий заданной группе (POSITIVE, NEGATIVE, NEUTRAL), шаблонный метод *Int pick_num(Int from, Int to) const* позволяет сгенерировать случайное число в диапозоне [from;to] включительно, метод *inline typename LegacyRandomAccessIterator::value_type pick_from_range(LegacyRandomAccessIterator begin, LegacyRandomAccessIterator end) const* выбирает случайный элемент из диапозона.

В классе Cell добавлены методы взаимодействия с ивентом. Метод *const EventInterface *get_active_event() const* позволяет получить ивент на клетке (возвращается nullptr если ивента не существует). Аналогично методы *void add_event(EventInterface *event)* и *void remove_event()* позволяют добавить и удалить ивент с клетки соответственно.

Класс Map переименован в GameField. Также он теперь наследуется от MapSubject и реализует интерфейс доступа к клетке (*Cell &get_cell(const Position &point) const override*), проверка проходимости клетки для данной вызывающей стороны (*bool can_move(EntityHandler *caller, const Position &point) const override*), а также метод *std::vector<Position> find_route(EntityHandler *caller, const Position &begin, const Position &end) const override* который использует алгоритм A* для поиска пути от стартовой точки до конечной точки (данный алгоритм не собирает ключи по пути, поэтому если у игрока нет ключа на момент вызова метода, но на пути есть дверь, а также и ключ, то вернётся пустой массив, т. е. путь не будет найден).

Создан класс Generator позволяющий генерировать карту по данным размерам, а также количеству ивентов. Конструктор принимает размерности карты, а также процент ивентов (т. е. процент от свободных клеток после генерации лабиринта).

Генерация карты начинаеся с создания новой карты, затем при помощи алгоритма Прима (постоение минимального остового дерева) генерируется лабиринт на карте. После этого случайным образом по заданному критерию выбираются точки старта и финиша, а также создаётся ключ. Строится путь от начала до финиша (проходящий через ключ), после чего предпоследняя клетка пути становится дверью. Далее вычисляются реальные количества ивентов каждого типа и выставляются на карту так, чтобы всегда существовал путь от старта до финиша, такой что игрок может дойти до конца уровня не подбирания ни одного бонусного ивента (т. е. со стартовым набором здоровья).

Создан класс DefaultLevelGenerator. Данный класс позволяет создавать несколько типов карт (SMALL, MEDIUM, BIG, HUGE), а также карты разной заполненности (т. е. сложности EASY, AVERAGE, HARD). Таким образом для создания уровней имеется возможность использовать любой из этих классов.

Оба класса возвращают указатель на полностью сгенерированную карту с событиями, но без врагов (в дальнейшем расстановкой ботов будет заниматься другой класс).

Для всех классов в программе была создана UML диаграмма (Рис. 1, Рис. 2, Рис. 3, Рис. 4, Рис. 5)

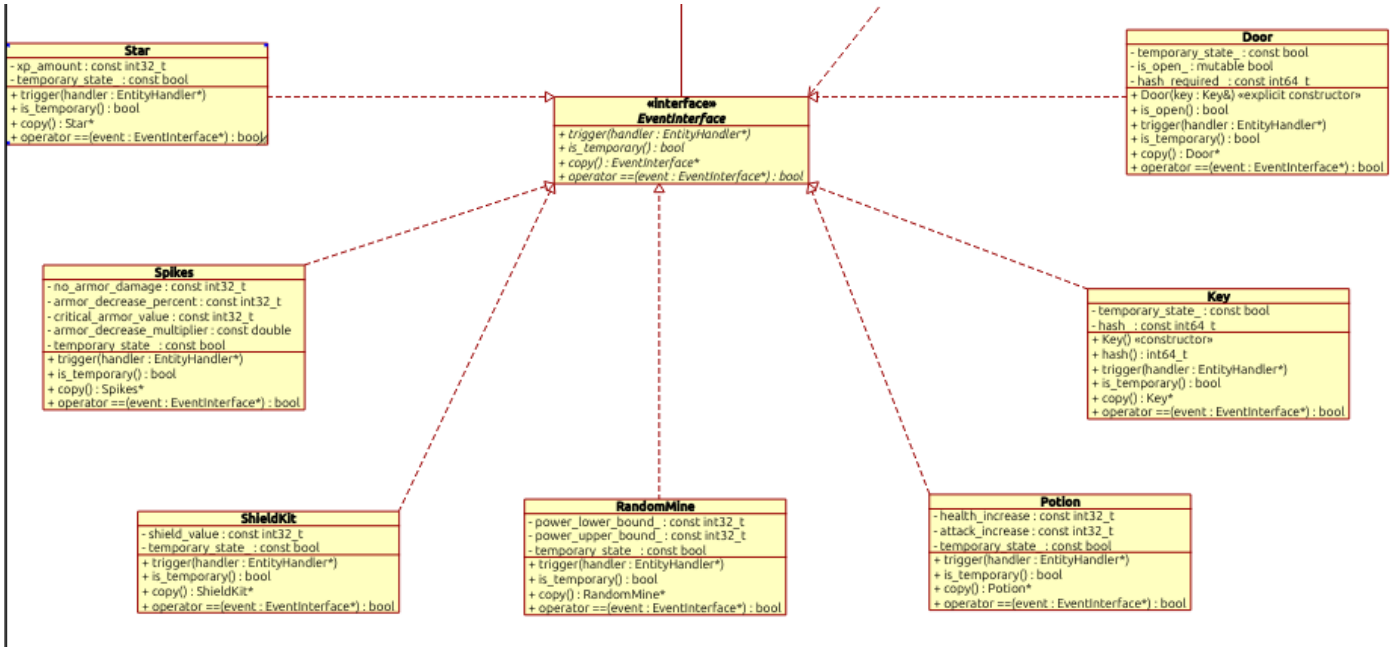

Рисунок 1 - Связь ивентов

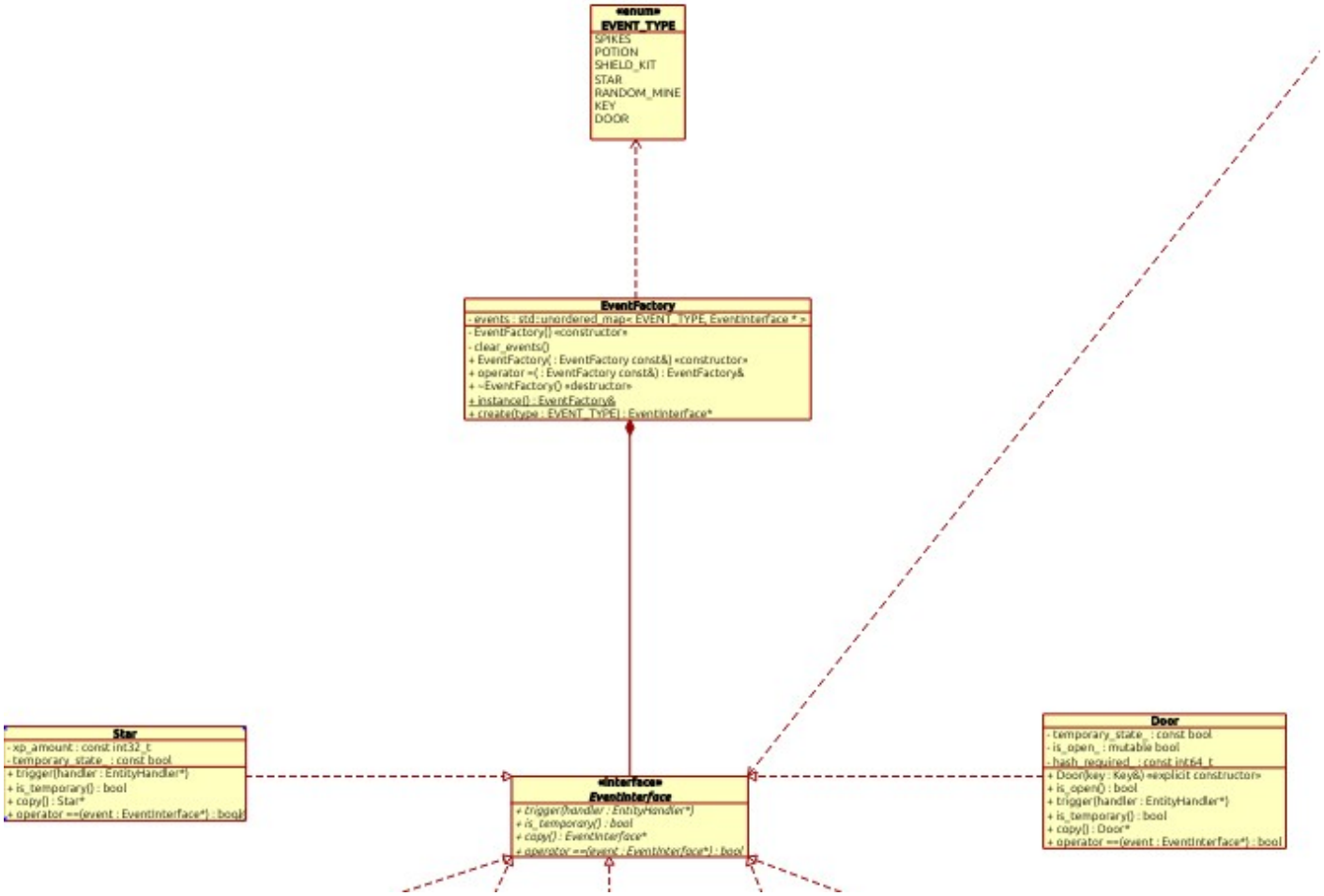

Рисунок 2 - Фабрика ивентов

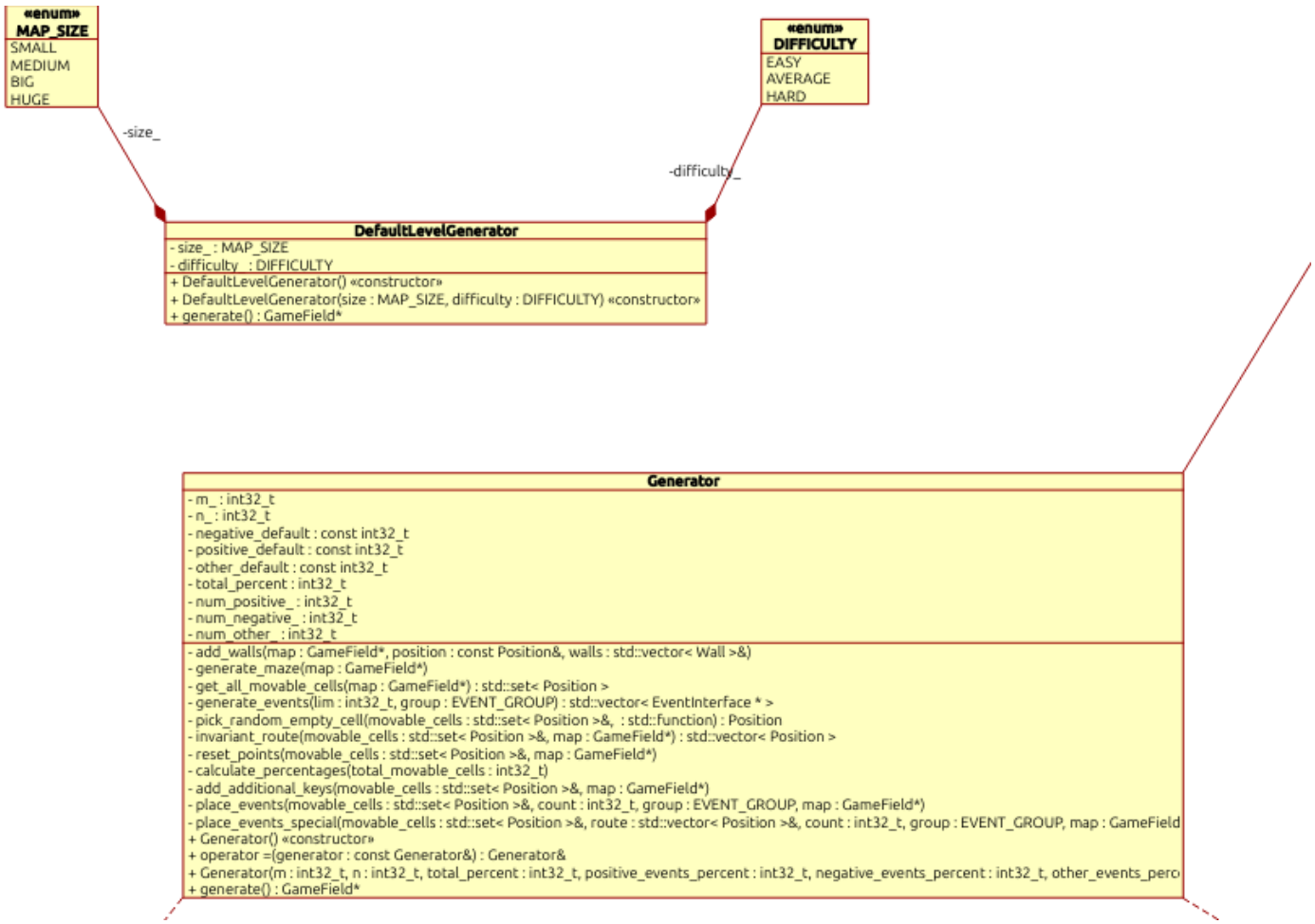

Рисунок 3 - Генерация уровней

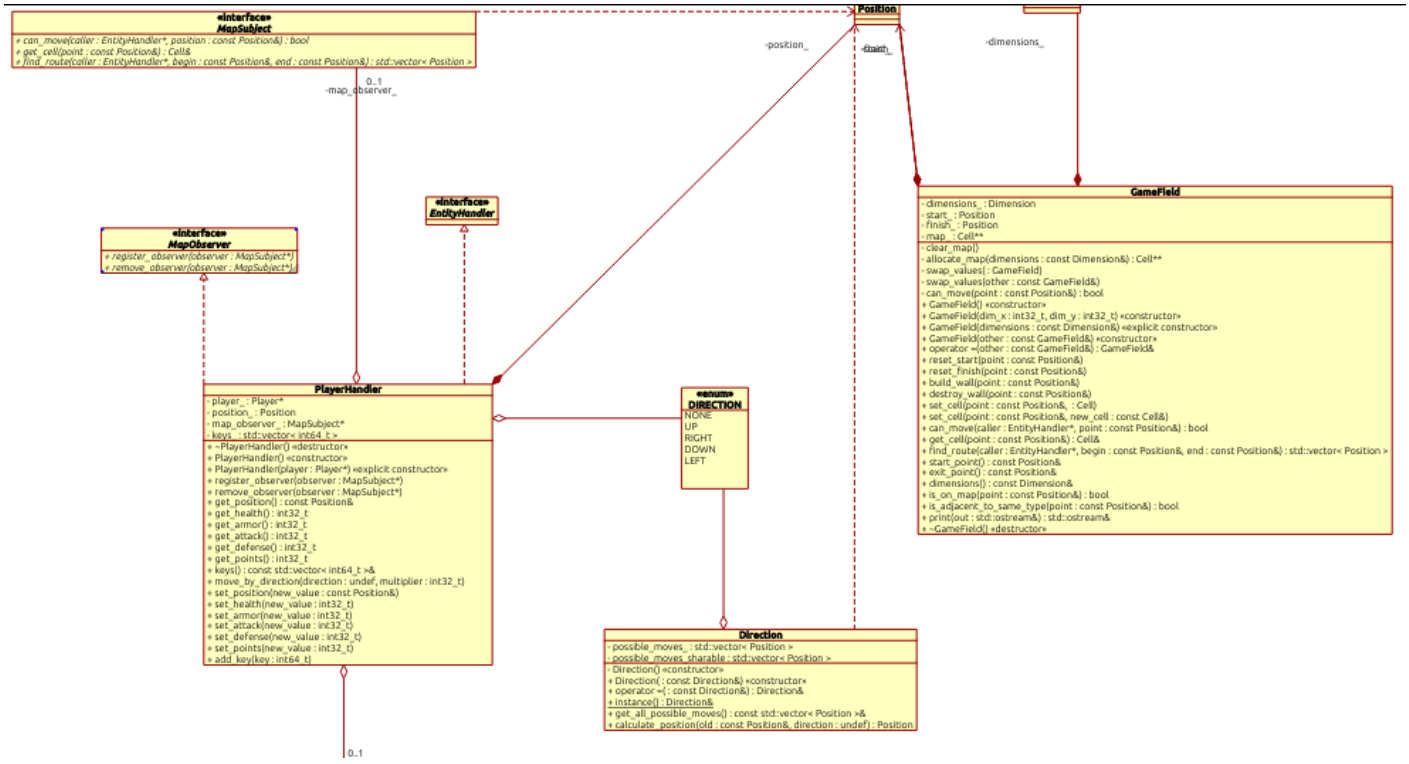

Рисунок 4 - Отношение игрока с картой

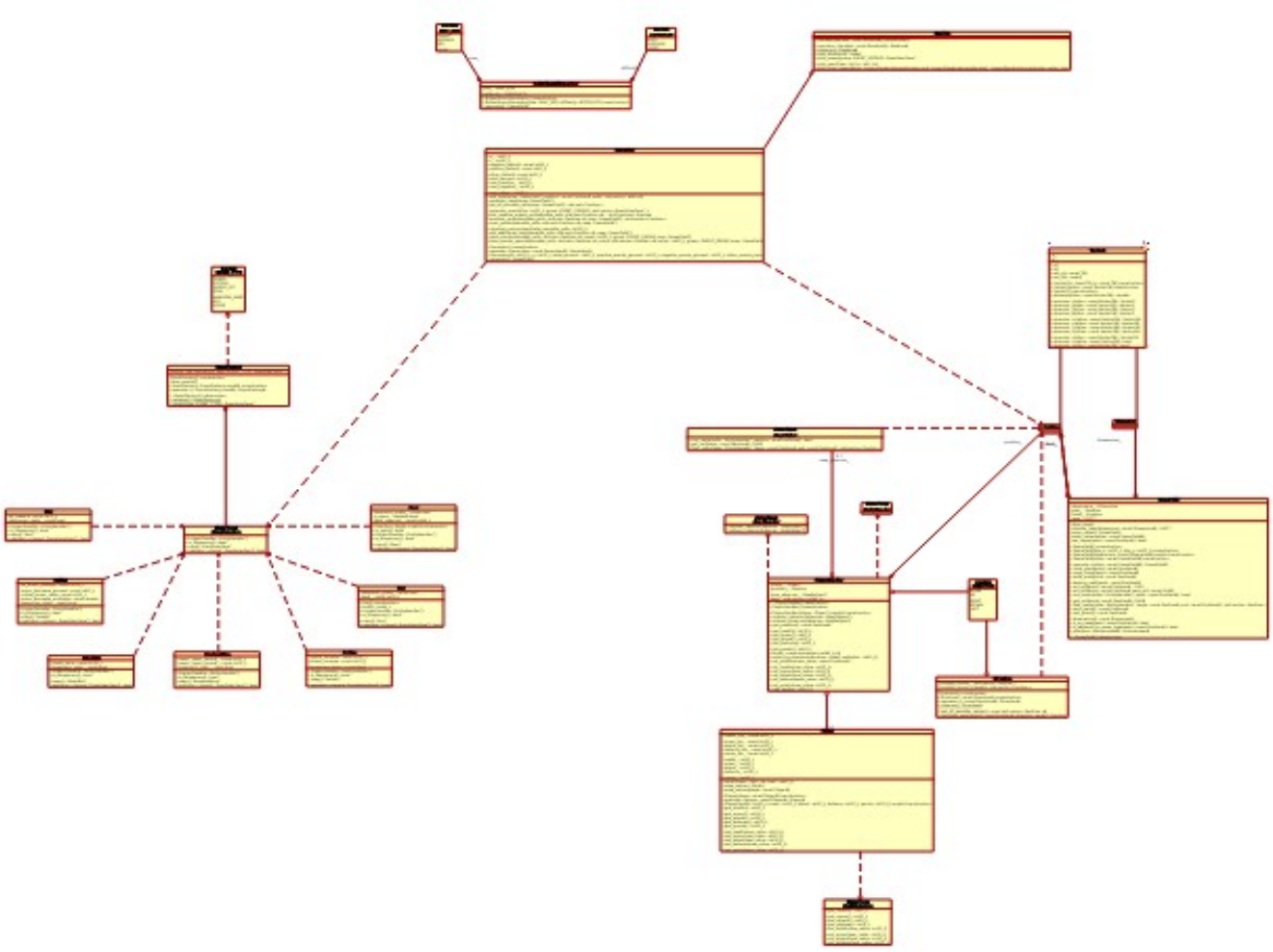

Рисунок 5 - Bird View

Для тестирования программы были использованы гугл тесты.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии

1.	PlayerHandler handler(new Player);
	auto *map = new GameField(10, 10);
	map->reset_start({0, 0});
		EXPECT_EQ(handler.keys().empty(), true);
		EXPECT_EQ(map->can_move(&handler, {1, 1}), false);
			Проверка корректной работы события Door

	map->reset_finish({7, 7});
	map->get_cell({1, 0}).add_event(EventFactory::instance().create(EVENT_TYPE::KEY));
	map->get_cell({1, 1}).add_event(EventFactory::instance().create(EVENT_TYPE::DOOR));
	handler.register_observer(map);
	handler.set_position({0, 0});
	map->print(std::cerr)
	handler.move_by_direction(DOWN, 1);
	delete map;
		EXPECT_EQ(map->can_move(nullptr, {1, 1}), true);
		EXPECT_EQ(handler.keys().size(), 1);
		EXPECT_EQ(map->can_move(&handler, {1, 1}), true);	

2.	DefaultLevelGenerator gen(MEDIUM, AVERAGE); auto new_map = gen.generate(); delete new_map;		Создание карты среднего размера

Выводы.

Разработан интерфейс событий, а также реализованы несколько базовых событий. Переработано взаимодействие игрока и карты, а такде добавлена проверка на наличие события на клетке. Создана случайная генерация карты, а также предусмотрено создание простейших уровней.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: src/Entities/Interface/EntityInterface.hpp

```
#pragma once
#include "../../Math/Vector2.hpp"
#include "../../Movement/Aliases.hpp"

class EntityInterface {
 public:
  virtual ~EntityInterface() = default;

  [[nodiscard]] virtual int32_t get_health() const = 0;
  [[nodiscard]] virtual int32_t get_armor() const = 0;
  [[nodiscard]] virtual int32_t get_attack() const = 0;
  [[nodiscard]] virtual int32_t get_defense() const = 0;

  virtual void set_health(int32_t new_value) = 0;
  virtual void set_armor(int32_t new_value) = 0;
  virtual void set_attack(int32_t new_value) = 0;
  virtual void set_defense(int32_t new_value) = 0;
};
```

Название файла: src/Entities/Player/Player.hpp

```
#pragma once
#include "../Interface/EntityInterface.hpp"

class Player : public EntityInterface {
 private:
  const int32_t health_lim_ = 500;
  const int32_t armor_lim_ = 100;
  const int32_t attack_lim_ = 50;
  const int32_t defence_lim_ = 50;
  const int32_t points_lim_ = 100000;

  int32_t health_;
  int32_t armor_;
  int32_t attack_;
  int32_t defence_;
  int32_t points_;

  void adjust(int32_t &value, int32_t limit);
  void swap_values(Player &&player);
  void swap_values(const Player &player);
 public:
  Player(const Player &player);
  Player(Player &&player) noexcept;
```

```
        Player &operator=(const Player &player);
        Player &operator=(Player &&player) noexcept;

        explicit Player(int32_t health = 100,
                        int32_t armor = 0,
                        int32_t attack = 10,
                        int32_t defence = 10,
                        int32_t points = 0);

        [[nodiscard]] int32_t get_health() const override;
        [[nodiscard]] int32_t get_armor() const override;
        [[nodiscard]] int32_t get_attack() const override;
        [[nodiscard]] int32_t get_defense() const override;
        [[nodiscard]] int32_t get_points() const;

        void set_health(int32_t new_value) override;
        void set_armor(int32_t new_value) override;
        void set_attack(int32_t new_value) override;
        void set_defense(int32_t new_value) override;
        void set_points(int32_t new_value);
    };
```

Название файла: src/Entities/Player/Player.cpp

```
    #include "Player.hpp"

    Player::Player(int32_t health, int32_t armor, int32_t attack,
int32_t defence, int32_t points) : health_(health),


armor_(armor),


attack_(attack),


defence_(defence),


points_(points)
    {
    }
    Player::Player(const Player &player) : Player()
    {
        swap_values(player);
    }
    Player::Player(Player &&player) noexcept: Player()
    {
        swap_values(std::move(player));
    }
    Player &Player::operator=(const Player &player)
    {
        if (this != &player)
        {
```

```
            health_ = player.health_;
            armor_ = player.armor_;
            attack_ = player.attack_;
            defence_ = player.defence_;
            points_ = player.points_;
      }

      return *this;
}
Player &Player::operator=(Player &&player) noexcept
{
      if (this != &player)
      {
            std::swap(health_, player.health_);
            std::swap(armor_, player.armor_);
            std::swap(attack_, player.attack_);
            std::swap(defence_, player.defence_);
            std::swap(points_, player.points_);
      }

      return *this;
}
int32_t Player::get_health() const
{
      return health_;
}
int32_t Player::get_armor() const
{
      return armor_;
}
int32_t Player::get_attack() const
{
      return attack_;
}
int32_t Player::get_defense() const
{
      return defence_;
}
int32_t Player::get_points() const
{
      return points_;
}
void Player::set_health(int32_t new_value)
{
      health_ = new_value;
      adjust(health_, health_lim_);
}
void Player::set_armor(int32_t new_value)
{
      armor_ = new_value;
      adjust(armor_, armor_lim_);
}
void Player::set_attack(int32_t new_value)
{
      attack_ = new_value;
```

```
	adjust(attack_, attack_lim_);
}
void Player::set_defense(int32_t new_value)
{
	defence_ = new_value;
	adjust(defence_, defence_lim_);
}
void Player::set_points(int32_t new_value)
{
	points_ = new_value;
	adjust(points_, points_lim_);
}
void Player::adjust(int32_t &value, int32_t limit)
{
	value = std::max(0, value);
	value = std::min(value, limit);
}
void Player::swap_values(const Player &player)
{
	if (this != &player)
	{
		health_ = player.health_;
		armor_ = player.armor_;
		attack_ = player.attack_;
		defence_ = player.defence_;
		points_ = player.points_;
	}
}
void Player::swap_values(Player &&player)
{
	if (this != &player)
	{
		std::swap(health_, player.health_);
		std::swap(armor_, player.armor_);
		std::swap(attack_, player.attack_);
		std::swap(defence_, player.defence_);
		std::swap(points_, player.points_);
	}
}
```

Название файла: src/Event/Factory/EventFactory.hpp

```
#pragma once

#include "../Interface/EventInterface.hpp"
#include "../../Handlers/PlayerHandler/PlayerHandler.hpp"
#include <unordered_map>

enum EVENT_TYPE {
	SPIKES, POTION, SHIELD_KIT, STAR, RANDOM_MINE, KEY, DOOR
};

class EventFactory {
```

```
 private:
  EventFactory();
  void clear_events();

  std::unordered_map<EVENT_TYPE, EventInterface *> events;
 public:
  EventFactory(EventFactory const &) = delete;
  EventFactory &operator=(EventFactory const &) = delete;
  ~EventFactory();

  static EventFactory &instance()
  {
       static EventFactory instance_;
       return instance_;
  }
  EventInterface *create(EVENT_TYPE type);
};
```

Название файла: src/Event/Factory/EventFactory.cpp

```
#include "EventFactory.hpp"
#include "../PositiveEvents/Potion.hpp"
#include "../PositiveEvents/ShieldKit.hpp"
#include "../PositiveEvents/Star.hpp"
#include "../NegativeEvents/Spikes.hpp"
#include "../MovementEvents/RandomMine.hpp"
#include "../MovementEvents/Door.hpp"

EventFactory::EventFactory()
{
     clear_events();

     events[RANDOM_MINE] = new RandomMine;
     events[SPIKES] = new Spikes;
     events[POTION] = new Potion;
     events[SHIELD_KIT] = new ShieldKit;
     events[STAR] = new Star;
     events[KEY] = new Key;
     events[DOOR] = new Door(*dynamic_cast<Key *>(events[KEY]));
}

EventFactory::~EventFactory()
{
     clear_events();
}

EventInterface *EventFactory::create(EVENT_TYPE type)
{
     EventInterface *event = nullptr;

     if (events.find(type) != events.end())
     {
```

```
            event = events[type]->copy();
        }
        return event;
    }

    void EventFactory::clear_events()
    {
        for (auto &[key, value] : events)
        {
            delete value;
        }
        events.clear();
    }
```

Название файла: src/Event/Interface/EventInterface.hpp

```
    #pragma once

    #include "Handlers/Interface/EntityHandler.hpp"

    class EventInterface {
     public:
      virtual ~EventInterface() = default;
      virtual void trigger(EntityHandler *handler) const = 0;
      [[nodiscard]] virtual bool is_temporary() const = 0;
      [[nodiscard]] virtual EventInterface *copy() const = 0;
       [[nodiscard]] virtual bool operator==(EventInterface *event)
const = 0;
    };
```

Название файла: src/Event/MovementEvents/Door.hpp

```
    #pragma once

    #include "Event/Interface/EventInterface.hpp"
    #include "Handlers/PlayerHandler/PlayerHandler.hpp"
    #include <cstdint>

    class Door : public EventInterface {
     private:
      const bool temporary_state_ = false;
      mutable bool is_open_ = false;

      const int64_t hash_required_;
     public:
      explicit Door(Key &key);
      bool is_open() const;
      void trigger(EntityHandler *handler) const override;
      [[nodiscard]] bool is_temporary() const override;
```

```
        [[nodiscard]] Door *copy() const override;
          [[nodiscard]]  bool  operator==(EventInterface  *event)  const
override;
      };
```

Название файла: src/Event/MovementEvents/Door.cpp

```
      #include "Door.hpp"

      Door::Door(Key &key) : hash_required_(key.hash())
      {
      }

      bool Door::is_open() const
      {
            return is_open_;
      }

      void Door::trigger(EntityHandler *handler) const
      {
            auto *handler_ = dynamic_cast<PlayerHandler *>(handler);

            if (handler_ == nullptr)
            {
                  return;
            }

            if (!is_open())
            {
                  bool has_needle_key = false;

                  for (const auto &key: handler_->keys())
                  {
                        if (key == hash_required_)
                        {
                              has_needle_key = true;
                              break;
                        }
                  }

                  if (has_needle_key)
                  {
                        is_open_ = true;
                  }
            }
      }

      bool Door::is_temporary() const
      {
            return temporary_state_;
      }
```

```
Door *Door::copy() const
{
     return new Door(*this);
}

bool Door::operator==(EventInterface *event) const
{
     return dynamic_cast<Door *>(event) != nullptr;
}
```

Название файла: src/Event/MovementEvents/Key.hpp

```
#pragma once

#include "Event/Interface/EventInterface.hpp"
#include <cstdint>

class PlayerHandler;

class Key : public EventInterface {
 private:
  const bool temporary_state_ = true;

  const int64_t hash_;
 public:
  Key();
  [[nodiscard]] int64_t hash() const;
  void trigger(EntityHandler *handler) const override;
  [[nodiscard]] bool is_temporary() const override;
  [[nodiscard]] Key *copy() const override;
    [[nodiscard]]  bool  operator==(EventInterface  *event)  const
override;
};
```

Название файла: src/Event/MovementEvents/Key.cpp

```
#include "Key.hpp"
#include "Random/Random.hpp"
#include "Handlers/PlayerHandler/PlayerHandler.hpp"

Key::Key()                                                          :
hash_(Random::instance().pick_num<int64_t>(INT32_MAX, INT64_MAX))
{
}

int64_t Key::hash() const
{
     return hash_;
}
```

```
void Key::trigger(EntityHandler *handler) const
{
     auto *handler_ = dynamic_cast<PlayerHandler *>(handler);

     if (handler_ == nullptr)
     {
          return;
     }

     handler_->add_key(hash());
}

bool Key::is_temporary() const
{
     return temporary_state_;
}
Key *Key::copy() const
{
     return new Key(*this);
}

bool Key::operator==(EventInterface *event) const
{
     return dynamic_cast<Key *>(event) != nullptr;
}
```

Название файла: src/Event/MovementEvents/RandomMine.hpp

```
#pragma once

#include "Event/Interface/EventInterface.hpp"
#include "Handlers/PlayerHandler/PlayerHandler.hpp"
#include <cstdint>

class RandomMine : public EventInterface {
 private:
  const int32_t power_lower_bound_ = 1;
  const int32_t power_upper_bound_ = 3;
  const bool temporary_state_ = false;
 public:
  void trigger(EntityHandler *handler) const override;
  [[nodiscard]] bool is_temporary() const override;
  [[nodiscard]] RandomMine *copy() const override;
    [[nodiscard]]  bool  operator==(EventInterface  *event)  const
override;
};
```

Название файла: src/Event/MovementEvents/RandomMine.cpp

```
#include "RandomMine.hpp"
#include "Random/Random.hpp"

void RandomMine::trigger(EntityHandler *handler) const
{
	auto *handler_ = dynamic_cast<PlayerHandler *>(handler);

	if (handler_ == nullptr)
	{
		return;
	}

	auto direction = Random::instance().pick_direction();
	auto power = Random::instance().pick_num(power_lower_bound_,
power_upper_bound_);

	handler_->move_by_direction(direction, power);
}

bool RandomMine::is_temporary() const
{
	return temporary_state_;
}

RandomMine *RandomMine::copy() const
{
	return new RandomMine(*this);
}

bool RandomMine::operator==(EventInterface *event) const
{
	return dynamic_cast<RandomMine *>(event) != nullptr;
}
```

Название файла: src/Handlers/Interface/EntityHandler.hpp

```
#pragma once

class EntityHandler {
 public:
  virtual ~EntityHandler() = default;
};
```

Название файла: src/Handlers/Interface/MapObserver.hpp

```
#pragma once

#include "MapSubject.hpp"

class MapObserver {
```

```
 public:
  virtual ~MapObserver() = default;
  virtual void register_observer(MapSubject* observer) = 0;
  virtual void remove_observer(MapSubject* observer) = 0;
};
```

Название файла: src/Handlers/Interface/MapSubject.hpp

```
#pragma once

#include "Movement/Aliases.hpp"
#include "Event/Interface/EventInterface.hpp"
#include "World/Cell.hpp"

class MapSubject {
 public:
  virtual ~MapSubject() = default;
    [[nodiscard]] virtual bool can_move(EntityHandler *caller,
const Position &position) const = 0;
    [[nodiscard]] virtual Cell &get_cell(const Position &point)
const = 0;
            [[nodiscard]] virtual std::vector<Position>
find_route(EntityHandler *caller, const Position &begin, const
Position &end) const = 0;
};
```

Название файла: src/Handlers/PlayerHandler/PlayerHandler.hpp

```
#pragma once
#include "../../Entities/Player/Player.hpp"
#include "../Interface/EntityHandler.hpp"
#include "../../Movement/Direction.hpp"
#include "../../World/GameField.hpp"
#include "Handlers/Interface/MapObserver.hpp"
#include "Event/MovementEvents/Key.hpp"

class PlayerHandler : public MapObserver, public EntityHandler {
 private:
  Player *player_;
  Position position_;
  MapSubject *map_observer_;
  std::vector<int64_t> keys_;
 public:
  ~PlayerHandler() override;
  PlayerHandler() = delete;
  explicit PlayerHandler(Player *player);

  void register_observer(MapSubject* observer) override;
  void remove_observer(MapSubject* observer) override;
```

```
        [[nodiscard]] const Position &get_position() const;
        [[nodiscard]] int32_t get_health() const;
        [[nodiscard]] int32_t get_armor() const;
        [[nodiscard]] int32_t get_attack() const;
        [[nodiscard]] int32_t get_defense() const;
        [[nodiscard]] int32_t get_points() const;
        [[nodiscard]] const std::vector<int64_t> &keys() const;

             void    move_by_direction(DIRECTION    direction,    int32_t
multiplier);
        void set_position(const Position &new_value);
        void set_health(int32_t new_value);
        void set_armor(int32_t new_value);
        void set_attack(int32_t new_value);
        void set_defense(int32_t new_value);
        void set_points(int32_t new_value);
        void add_key(int64_t key);
      };
```

Название файла: src/Handlers/PlayerHandler/PlayerHandler.cpp

```
      #include "PlayerHandler.hpp"

      PlayerHandler::~PlayerHandler()
      {
           delete player_;
      }
      const Position &PlayerHandler::get_position() const
      {
           return position_;
      }
      int32_t PlayerHandler::get_health() const
      {
           return player_->get_health();
      }
      int32_t PlayerHandler::get_armor() const
      {
           return player_->get_armor();
      }
      int32_t PlayerHandler::get_attack() const
      {
           return player_->get_attack();
      }
      int32_t PlayerHandler::get_defense() const
      {
           return player_->get_defense();
      }
      int32_t PlayerHandler::get_points() const
      {
           return player_->get_points();
      }
      void PlayerHandler::set_position(const Position &new_value)
      {
```

```
            position_ = new_value;
      }
      void PlayerHandler::set_health(int32_t new_value)
      {
            player_->set_health(new_value);
      }
      void PlayerHandler::set_armor(int32_t new_value)
      {
            player_->set_armor(new_value);
      }
      void PlayerHandler::set_attack(int32_t new_value)
      {
            player_->set_attack(new_value);
      }
      void PlayerHandler::set_defense(int32_t new_value)
      {
            player_->set_defense(new_value);
      }
      void PlayerHandler::set_points(int32_t new_value)
      {
            player_->set_points(new_value);
      }
      void    PlayerHandler::move_by_direction(DIRECTION    direction,
int32_t multiplier)
      {
            if (map_observer_ == nullptr)
            {
                  throw   std::invalid_argument("MapHandler   was   not
initialized");
            }
            for (int32_t i = 0; i < multiplier; ++i)
            {
                  auto                    new_position                   =
Direction::instance().calculate_position(position_, direction);

                  if (map_observer_->can_move(this, new_position))
                  {
                        set_position(new_position);

                        auto      active_event      =      map_observer_-
>get_cell(new_position).get_active_event();
                        if (active_event != nullptr)
                        {
                              active_event->trigger(this);

                              if (active_event->is_temporary())
                              {
                                    map_observer_-
>get_cell(new_position).remove_event();
                              }
                        }
                  }
            }

            // TODO notify subscribers about player move
```

```
}
PlayerHandler::PlayerHandler(Player  *player)  :  player_(player),
map_observer_(nullptr)
{
     if (player_ == nullptr)
     {
          throw    std::invalid_argument("Nullptr    passed    to
PlayerHandler");
     }
}
void PlayerHandler::register_observer(MapSubject *observer)
{
     map_observer_ = observer;
}
void PlayerHandler::remove_observer(MapSubject *observer)
{
     map_observer_ = nullptr;
}
const std::vector<int64_t> &PlayerHandler::keys() const
{
     return keys_;
}
void PlayerHandler::add_key(int64_t key)
{
     keys_.push_back(key);
}
```

Название файла: src/Math/Vector2.hpp

```
#pragma once

#include <algorithm>
#include <cmath>

template<typename T>
class Vector2 {
 private:
  T x_, y_;
 public:
  // getters/setters
  T y() const;
  T x() const;

  void set_y(const T &y);
  void set_x(T x);

  // constructors
  Vector2(const T &ix, const T &iy);
  Vector2(const Vector2 &other);
  Vector2();

  [[nodiscard]] double distance(const Vector2 &other) const;
```

```
        // operators
        Vector2 operator+(const Vector2 &other) const;
        Vector2 operator-(const Vector2 &other) const;
        Vector2 operator*(const Vector2 &other) const;
        Vector2 operator/(const Vector2 &other) const;
        Vector2 &operator+=(const Vector2 &other);
        Vector2 &operator-=(const Vector2 &other);
        Vector2 &operator*=(const Vector2 &other);
        Vector2 &operator/=(const Vector2 &other);
        Vector2 &operator=(const Vector2 &other);
        Vector2 &operator=(Vector2 &&other) noexcept;
        bool operator==(const Vector2 &other) const;
        bool operator<(const Vector2 &other) const;
    };
    template<typename T>
    double Vector2<T>::distance(const Vector2 &other) const
    {
        return std::sqrt(std::pow(x() - other.x(), 2) + std::pow(y()
- other.y(), 2));
    }

    template<typename T>
    Vector2<T>::Vector2(const T &ix, const T &iy): x_(ix), y_(iy)
    {
    }
    template<typename T>
    Vector2<T>::Vector2(): Vector2(0, 0)
    {
    }
    template<typename T>
    Vector2<T>::Vector2(const Vector2 &other) : Vector2()
    {
        x_ = other.x_;
        y_ = other.y_;
    }
    template<typename T>
    Vector2<T> Vector2<T>::operator+(const Vector2<T> &other) const
    {
        Vector2<T> temp(*this);
        temp += other;
        return temp;
    }
    template<typename T>
    Vector2<T> Vector2<T>::operator-(const Vector2<T> &other) const
    {
        Vector2<T> temp(*this);
        temp -= other;
        return temp;
    }
    template<typename T>
    Vector2<T> Vector2<T>::operator*(const Vector2<T> &other) const
    {
        Vector2<T> temp(*this);
        temp *= other;
```

```
	return temp;
}
template<typename T>
Vector2<T> Vector2<T>::operator/(const Vector2<T> &other) const
{
	Vector2<T> temp(*this);
	temp /= other;
	return temp;
}
template<typename T>
Vector2<T> &Vector2<T>::operator+=(const Vector2<T> &other)
{
	x_ += other.x_;
	y_ += other.y_;
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator-=(const Vector2<T> &other)
{
	x_ -= other.x_;
	y_ -= other.y_;
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator*=(const Vector2<T> &other)
{
	x_ *= other.x_;
	y_ *= other.y_;
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator/=(const Vector2<T> &other)
{
	if (other.x_ != 0 && other.y_ != 0)
	{
		x_ /= other.x_;
		y_ /= other.y_;
	}
	return *this;
}
template<typename T>
T Vector2<T>::y() const
{
	return y_;
}
template<typename T>
void Vector2<T>::set_y(const T &y)
{
	y_ = y;
}
template<typename T>
T Vector2<T>::x() const
{
	return x_;
}
```

```
template<typename T>
void Vector2<T>::set_x(T x)
{
	x_ = x;
}
template<typename T>
Vector2<T> &Vector2<T>::operator=(const Vector2 &other)
{
	if (this != &other)
	{
		x_ = other.x_;
		y_ = other.y_;
	}
	return *this;
}
template<typename T>
Vector2<T> &Vector2<T>::operator=(Vector2 &&other) noexcept
{
	if (this != &other)
	{
		x_ = std::move(other.x_);
		y_ = std::move(other.y_);
	}
	return *this;
}
template<typename T>
bool Vector2<T>::operator==(const Vector2 &other) const
{
	return x_ == other.x_ && y_ == other.y_;
}
template<typename T>
bool Vector2<T>::operator<(const Vector2 &other) const
{
	if (x_ == other.x_)
	{
		return y_ < other.y_;
	}
	return x_ < other.x_;
}
```

Название файла: src/Movement/Aliases.hpp

```
#pragma once
#include "../Math/Vector2.hpp"
#include <cstdint>

using Position = Vector2<int32_t>;
using Dimension = Vector2<int32_t>;
```

Название файла: src/Movement/Direction.hpp

```
#pragma once
#include "../Math/Vector2.hpp"
```

```
#include "../Entities/Interface/EntityInterface.hpp"
#include <vector>

enum DIRECTION {
     NONE, UP, RIGHT, DOWN, LEFT
};

class Direction {
 private:
  std::vector<Position> possible_moves_ = {{0, 0}};
  std::vector<Position> possible_moves_sharable = {{-1, 0},

{0,  1},

{1,  0},

{0,  -1}};
  Direction();
 public:
  Direction(const Direction &) = delete;
  Direction &operator=(const Direction &) = delete;

  static Direction &instance()
  {
       static Direction instance_;
       return instance_;
  }
               [[nodiscard]]      const      std::vector<Position>
&get_all_possible_moves() const;
    Position  calculate_position(const  Position  &old,  DIRECTION
direction);
};
```

Название файла: src/Movement/Direction.cpp

```
#include "Direction.hpp"

Position   Direction::calculate_position(const   Position   &old,
DIRECTION direction)
{
     return old + possible_moves_[direction];
}
Direction::Direction()
{
     std::copy(possible_moves_sharable.begin(),
possible_moves_sharable.end(), std::back_inserter(possible_moves_));
}
const  std::vector<Position>  &Direction::get_all_possible_moves()
const
{
     return possible_moves_sharable;
}
```

Название файла: src/Profiler/Timer.hpp

```
#pragma once

#include <ostream>
#include <chrono>
#include <string>

class Timer {
 private:
      std::chrono::time_point<std::chrono::high_resolution_clock>
start_time_;
  std::ostream& out_;
  std::string message_;
 public:
  Timer(std::ostream &out, std::string message);
  ~Timer();
};
```

Название файла: src/Profiler/Timer.cpp

```
#include "Timer.hpp"

Timer::Timer(std::ostream &out, std::string message) : out_(out),
message_(std::move(message))
{
      start_time_ = std::chrono::high_resolution_clock::now();
}
Timer::~Timer()
{
      auto end_time = std::chrono::high_resolution_clock::now();
      auto                        duration                        =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time       -
start_time_).count();
      auto                duration_microseconds                   =
std::chrono::duration_cast<std::chrono::microseconds>(end_time       -
start_time_).count();
      out_  <<  message_  <<  "  "  <<  duration  <<  "  ms  "  <<
duration_microseconds << " microseconds" <<  std::endl;
}
```

Название файла: src/Random/Random.hpp

```
#pragma once

#include "Movement/Direction.hpp"
#include "Event/Interface/EventInterface.hpp"
```

```
#include <random>

enum EVENT_GROUP
{
      NEGATIVE = -1,
      NEUTRAL,
      POSITIVE
};

class Random {
  Random() = default;
 public:
  Random(const Random &random) = delete;
  Random &operator=(const Random &random) = delete;

  static Random &instance()
  {
        static Random instance_;
        return instance_;
  }
  [[nodiscard]] DIRECTION pick_direction() const;
    [[nodiscard]]  EventInterface  *pick_event(EVENT_GROUP  group)
const;

  template <typename Int>
  [[nodiscard]] Int pick_num(Int from, Int to) const
  {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<Int> distribution(from, to);

        return distribution(gen);
  }

  template <class LegacyRandomAccessIterator>
                      [[nodiscard]]          inline          typename
LegacyRandomAccessIterator::value_type
pick_from_range(LegacyRandomAccessIterator                          begin,
LegacyRandomAccessIterator end) const
  {
                return    *std::next(begin,    pick_num<int64_t>(0,
std::distance(begin, end) - 1));
  }
};
```

Название файла: src/Random/Random.cpp

```
#include "Random.hpp"
#include "Event/Factory/EventFactory.hpp"

DIRECTION Random::pick_direction() const
{
```

```
          auto                        &directions                         =
Direction::instance().get_all_possible_moves();
          return
static_cast<DIRECTION>(pick_num(static_cast<size_t>(0),
directions.size() - 1));
     }
     EventInterface *Random::pick_event(EVENT_GROUP group) const
     {
          EventInterface *event = nullptr;
          if (group == NEUTRAL)
          {
               event                                                    =
EventFactory::instance().create(EVENT_TYPE::RANDOM_MINE);
          }
          else if (group == POSITIVE)
          {
               event                                                    =
EventFactory::instance().create(static_cast<EVENT_TYPE>(pick_num<int16
_t>(EVENT_TYPE::POTION, EVENT_TYPE::STAR)));
          }
          else if (group == NEGATIVE)
          {
               event                                                    =
EventFactory::instance().create(EVENT_TYPE::SPIKES);
          }
          return event;
     }
```

Название файла: src/World/Cell.hpp

```
     #pragma once

     #include "Event/Interface/EventInterface.hpp"
     #include <ostream>

     class Cell {
      public:
       enum TYPE {
            ENTRANCE, EXIT, MOVABLE, WALL, PATH_PART
       };
      public:
       Cell();
       ~Cell();
       explicit Cell(TYPE type);
       Cell(const Cell &other) noexcept;
       Cell(Cell &&other) noexcept;
       Cell &operator=(const Cell &other) noexcept;
       Cell &operator=(Cell &&other) noexcept;
       bool operator==(const Cell &other) const;

       void set_type(TYPE new_type);
```

```
        [[nodiscard]] bool is_entrance() const;
        [[nodiscard]] bool is_exit() const;
        [[nodiscard]] bool is_movable() const;
        [[nodiscard]] bool has_door() const;
        [[nodiscard]] TYPE type() const;

        [[nodiscard]] const EventInterface *get_active_event() const;
        void add_event(EventInterface *event);
        void remove_event();
         friend std::ostream &operator<<(std::ostream &out, const Cell
&cell);
       private:
        TYPE type_;
        EventInterface *event_;

        void swap_values(Cell &&other);
        void swap_values(const Cell &other);
      };
```

Название файла: src/World/Cell.cpp

```
      #include "Cell.hpp"
      #include "Event/PositiveEvents/Potion.hpp"
      #include "Event/PositiveEvents/ShieldKit.hpp"
      #include "Event/PositiveEvents/Star.hpp"
      #include "Event/NegativeEvents/Spikes.hpp"
      #include "Event/MovementEvents/RandomMine.hpp"
      #include "Event/MovementEvents/Door.hpp"

      Cell::Cell() : Cell(TYPE::MOVABLE)
      {
      }
      Cell::Cell(TYPE type) : type_(type), event_(nullptr)
      {
      }
      bool Cell::is_entrance() const
      {
           return type_ == TYPE::ENTRANCE;
      }
      bool Cell::is_exit() const
      {
           return type_ == TYPE::EXIT;
      }
      bool Cell::is_movable() const
      {
           return type_ == TYPE::MOVABLE || is_exit() || is_entrance();
      }
      Cell::Cell(const Cell &other) noexcept : Cell()
      {
           swap_values(other);
      }
      Cell &Cell::operator=(Cell &&other) noexcept
```

```
{
     if (this != &other)
     {
          swap_values(std::move(other));
     }
     return *this;
}
Cell::Cell(Cell &&other) noexcept: Cell()
{
     if (this != &other)
     {
          swap_values(std::move(other));
     }
}
Cell &Cell::operator=(const Cell &other) noexcept
{
     if (this != &other)
     {
          swap_values(other);
     }
     return *this;
}
void Cell::set_type(TYPE new_type)
{
     type_ = new_type;
}
Cell::~Cell()
{
     remove_event();
}
bool Cell::operator==(const Cell &other) const
{
     return type_ == other.type_/* && event_ == other.event_*/;
}
const EventInterface *Cell::get_active_event() const
{
     return event_;
}
void Cell::add_event(EventInterface *event)
{
     remove_event();
     event_ = event;
}
void Cell::remove_event()
{
     delete event_;
     event_ = nullptr;
}
Cell::TYPE Cell::type() const
{
     return type_;
}
void Cell::swap_values(Cell &&other)
{
     std::swap(type_, other.type_);
```

```
            remove_event();
            add_event(other.event_);
            other.event_ = nullptr;
      }
      void Cell::swap_values(const Cell &other)
      {
            type_ = other.type_;
            remove_event();
            add_event(other.event_ == nullptr ? nullptr : other.event_-
>copy());
      }
      std::ostream &operator<<(std::ostream &out, const Cell &cell)
      {
            if (cell.is_entrance())
            {
                  out << "[0]";
            }
            else if (cell.is_exit())
            {
                  out << "[≋]";
            }
            else if (cell.type_ == Cell::TYPE::WALL)
            {
                  out << "[■]";
            }
            else if (cell.type_ == Cell::TYPE::PATH_PART)
            {
                  out << "[△]";
            }
            else if (cell.has_door())
            {
                  out << "[⇹]";
            }
            else
            {
                  auto *active_event = cell.get_active_event();
                  if (active_event == nullptr)
                  {
                        out << "[ ]";
                  }
                  else if (const auto *p = dynamic_cast<const Spikes
*>(active_event); p != nullptr)
                  {
                        out << "[#]";
                  }
                  else if (const auto *k = dynamic_cast<const Potion
*>(active_event); k != nullptr)
                  {
                        out << "[+]";
                  }
                  else if (const auto *d = dynamic_cast<const RandomMine
*>(active_event); d != nullptr)
                  {
                        out << "[?]";
                  }
```

```
                else if (const auto *m = dynamic_cast<const ShieldKit
*>(active_event); m != nullptr)
                {
                     out << "[⛨]";
                }
                else if (const auto *n = dynamic_cast<const Star
*>(active_event); n != nullptr)
                {
                     out << "[⭐]";
                }
                else if (const auto *t = dynamic_cast<const Key
*>(active_event); t != nullptr)
                {
                     out << "[⚿]";
                }
           }
           return out;
     }
     bool Cell::has_door() const
     {
           return dynamic_cast<Door *>(event_) != nullptr;
     }
```

Название файла: src/World/GameField.hpp

```
     #pragma once

     #include <cstdint>
     #include "Handlers/Interface/MapSubject.hpp"
     #include "Movement/Aliases.hpp"
     #include "Cell.hpp"

     class GameField : public MapSubject {
      private:
     #define MAP_DIMENSION_UPPER_BOUND 1000
     #define MAP_DIMENSION_LOWER_BOUND 10

       Dimension dimensions_;
       Position start_;
       Position finish_;

       Cell **map_;

       void clear_map();
       Cell **allocate_map(const Dimension &dimensions);

       void swap_values(GameField &&other);
       void swap_values(const GameField &other);

       [[nodiscard]] bool can_move(const Position &point) const;
      public:
       GameField();
```

```
        GameField(int32_t dim_x, int32_t dim_y);
        explicit GameField(const Dimension &dimensions);

        GameField(const GameField &other);
        GameField(GameField &&other) noexcept;

        GameField &operator=(const GameField &other);
        GameField &operator=(GameField &&other) noexcept;

        void reset_start(const Position &point);
        void reset_finish(const Position &point);
        void build_wall(const Position &point);
        void destroy_wall(const Position &point);

        void set_cell(const Position &point, Cell &&new_cell);
        void set_cell(const Position &point, const Cell &new_cell);

          [[nodiscard]] bool can_move(EntityHandler *caller, const
Position &point) const override;
         [[nodiscard]] Cell &get_cell(const Position &point) const
override;
         [[nodiscard]] std::vector<Position> find_route(EntityHandler
*caller, const Position &begin, const Position &end) const override;

        [[nodiscard]] const Position &start_point() const;
        [[nodiscard]] const Position &exit_point() const;
        [[nodiscard]] const Dimension &dimensions() const;
        [[nodiscard]] bool is_on_map(const Position &point) const;
          [[nodiscard]] bool is_adjacent_to_same_type(const Position
&point) const;

        std::ostream &print(std::ostream &out) const;
        ~GameField() override;
      };
```

Название файла: src/World/GameField.cpp

```
      #include "GameField.hpp"
      #include "Movement/Direction.hpp"
      #include "Event/MovementEvents/Door.hpp"
      #include <ostream>
      #include <queue>
      #include <map>
      #include <vector>

      GameField::GameField(const Dimension &dimensions) :
dimensions_(dimensions),
```

```
      start_(-1, -1),

      finish_(-1, -1),

      map_(nullptr)
      {
            if (dimensions_.x() < MAP_DIMENSION_LOWER_BOUND ||
dimensions_.y() < MAP_DIMENSION_LOWER_BOUND
                  || dimensions_.x() > MAP_DIMENSION_UPPER_BOUND ||
dimensions_.y() > MAP_DIMENSION_UPPER_BOUND)
            {
                  throw std::logic_error("Unexpected dimensions for map
(too small or too big)");
            }
            map_ = allocate_map(dimensions_);
      }
      GameField::GameField(int32_t dim_x, int32_t dim_y) :
GameField(Position(dim_x, dim_y))
      {
      }
      GameField::~GameField()
      {
            clear_map();
      }
      GameField::GameField() : GameField(MAP_DIMENSION_LOWER_BOUND,
MAP_DIMENSION_LOWER_BOUND)
      {
      }
      GameField::GameField(const GameField &other) : GameField()
      {
            swap_values(other);
      }
      GameField::GameField(GameField &&other) noexcept: GameField()
      {
            swap_values(std::move(other));
      }
```

```
GameField &GameField::operator=(const GameField &other)
{
	if (&other != this)
	{
		swap_values(other);
	}
	return *this;
}
GameField &GameField::operator=(GameField &&other) noexcept
{
	if (&other != this)
	{
		swap_values(std::move(other));
	}
	return *this;
}
void GameField::clear_map()
{
	if (map_ == nullptr)
	{
		return;
	}

	for (int32_t i = 0; i < dimensions_.x(); ++i)
	{
		for (int32_t j = 0; j < dimensions_.y(); ++j)
		{
			map_[i][j].remove_event();
		}
		delete[] map_[i];
	}
	delete[] map_;
	map_ = nullptr;
}
Cell **GameField::allocate_map(const Dimension &dimensions)
{
	auto map = new Cell *[dimensions.x()];
```

```
            for (int32_t i = 0; i < dimensions.x(); ++i)
            {
                map[i] = new Cell[dimensions.y()];
            }
            return map;
        }
        Cell &GameField::get_cell(const Position &point) const
        {
            if (!is_on_map(point))
            {
                throw std::out_of_range("Point is out of map");
            }
            return map_[point.x()][point.y()];
        }
        void GameField::set_cell(const Position &point, Cell &&new_cell)
        {
            if (!is_on_map(point))
            {
                return;
            }
            map_[point.x()][point.y()] = std::move(new_cell);
        }
        void  GameField::set_cell(const  Position  &point,  const  Cell
&new_cell)
        {
            if (!is_on_map(point))
            {
                return;
            }
            map_[point.x()][point.y()] = new_cell;
        }
        const Dimension &GameField::dimensions() const
        {
            return dimensions_;
        }
        void GameField::reset_start(const Position &point)
```

```
	{
		if (can_move(point))
		{
			if (start_ != Position(-1, -1))
			{
				map_[start_.x()]
[start_.y()].set_type(Cell::TYPE::MOVABLE);
			}
			start_ = point;
			map_[start_.x()]
[start_.y()].set_type(Cell::TYPE::ENTRANCE);
		}
	}
	void GameField::reset_finish(const Vector2<int32_t> &point)
	{
		if (can_move(point))
		{
			if (finish_ != Position(-1, -1))
			{
				map_[finish_.x()]
[finish_.y()].set_type(Cell::TYPE::MOVABLE);
			}
			finish_ = point;
			map_[finish_.x()]
[finish_.y()].set_type(Cell::TYPE::EXIT);
		}
	}
	void GameField::build_wall(const Position &point)
	{
		if (is_on_map(point) && get_cell(point).is_movable())
		{
			map_[point.x()][point.y()].set_type(Cell::TYPE::WALL);
		}
	}
	void GameField::destroy_wall(const Position &point)
	{
		if (is_on_map(point) && !get_cell(point).is_movable())
```

```
            {
                  map_[point.x()]
[point.y()].set_type(Cell::TYPE::MOVABLE);
            }
      }
      bool GameField::is_on_map(const Position &point) const
      {
            return point.x() >= 0 && point.y() >= 0 && point.x() <
dimensions_.x() && point.y() < dimensions_.y();
      }
      const Position &GameField::start_point() const
      {
            return start_;
      }
      const Position &GameField::exit_point() const
      {
            return finish_;
      }
      std::ostream &GameField::print(std::ostream &out) const
      {
            for (int32_t i = 0; i < dimensions_.x(); i++)
            {
                  for (int32_t j = 0; j < dimensions_.y(); j++)
                  {
                        out << get_cell({i, j});
                        out << std::flush;
                  }
                  out << std::endl;
            }
            return out << std::endl;
      }
      bool GameField::is_adjacent_to_same_type(const Position &point)
const
      {
            return
std::ranges::any_of(Direction::instance().get_all_possible_moves().beg
```

```
in(), Direction::instance().get_all_possible_moves().end(), [&](const
Position &direction)
          {
               const auto &pos = point + direction;

               if (is_on_map(pos))
               {
                    auto *lhs = get_cell(pos).get_active_event();
                    auto *rhs = get_cell(point).get_active_event();

                    return lhs != nullptr && lhs == rhs;
               }
               return false;
          });
     }
     bool GameField::can_move(EntityHandler *caller, const Position
&point) const
     {
          if (!can_move(point))
          {
               return false;
          }

          auto &cell = get_cell(point);

          if (caller != nullptr && cell.has_door())
          {
               auto *handler = dynamic_cast<PlayerHandler *>(caller);

               if (handler != nullptr)
               {
                    cell.get_active_event()->trigger(caller);

                    return          dynamic_cast<const          Door
*>(cell.get_active_event())->is_open();
               }
          }
```

```
            return true;
      }
      void GameField::swap_values(GameField &&other)
      {
            clear_map();

            std::swap(dimensions_, other.dimensions_);
            std::swap(start_, other.start_);
            std::swap(finish_, other.finish_);
            std::swap(map_, other.map_);

            other.map_ = nullptr;
      }
      void GameField::swap_values(const GameField &other)
      {
            clear_map();
            map_ = allocate_map(other.dimensions_);

            dimensions_ = other.dimensions_;
            start_ = other.start_;
            finish_ = other.finish_;

            for (int32_t i = 0; i < dimensions_.x(); ++i)
            {
                  for (int32_t j = 0; j < dimensions_.y(); ++j)
                  {
                        map_[i][j] = other.map_[i][j];
                  }
            }
      }

      // A* search algorithm
      // https://en.wikipedia.org/wiki/A*_search_algorithm
      std::vector<Position>         GameField::find_route(EntityHandler
*caller, const Position &begin, const Position &goal) const
      {
```

```
            // Define the heuristic function
            auto heuristic = [](const Position &p1, const Position &p2)
-> int
            {
                  return abs(p1.x() - p2.x()) + abs(p1.y() - p2.y());
            };

            std::priority_queue<std::pair<int,                Position>,
std::vector<std::pair<int, Position>>, std::greater<>> frontier;
            std::map<Position, Position> came_from;
            std::map<Position, int> cost_so_far;
            bool found_path = false;

            // push start point
            frontier.push(std::move(std::make_pair(0, begin)));
            cost_so_far[begin] = 0;

            // While there is still places to go
            while (!frontier.empty())
            {
                  auto current = frontier.top().second;
                  frontier.pop();

                  // If we reached our goal we can stop
                  if (current == goal)
                  {
                        found_path = true;
                        break;
                  }

                  // Check all neighbours
                  for                     (auto                     &dir:
Direction::instance().get_all_possible_moves())
                  {
                        const auto &next = current + dir;
```

```
                    // Check if the neighbour is in the grid and is
not a wall
                    if (!can_move(caller, next))
                    {
                          continue;
                    }

                    auto new_cost = cost_so_far[current] + 1;

                    // If it's a new node or we found a better way to
get to this node
                    if  (cost_so_far.find(next)  ==  cost_so_far.end()
|| new_cost < cost_so_far[next])
                    {
                          cost_so_far[next] = new_cost;
                          auto  priority  =  new_cost  +  heuristic(next,
goal);

      frontier.push(std::move(std::make_pair(priority, next)));
                          came_from[next] = current;
                    }
               }
          }

          // Reconstruct the path
          std::vector<Position> path;

          if (found_path)
          {
               auto current = goal;
               while (current != begin)
               {
                    path.push_back(current);
                    current = came_from[current];
               }
               path.push_back(begin);
               std::reverse(path.begin(), path.end());
```

```
        }
        return path;
    }
    bool GameField::can_move(const Position &point) const
    {
        return is_on_map(point) && get_cell(point).is_movable();
    }
```

Название файла: src/MapGenerator/RandomLevelGen/Generator.hpp

```
    #pragma once

    #include "World/GameField.hpp"
    #include "Random/Random.hpp"
    #include <cstdint>
    #include <set>

    class Generator {
     private:
      struct Wall {
            Position pos;
            Position cell1;
            Position cell2;
      };

      int32_t m_;
      int32_t n_;

      const int32_t negative_default = 50;
      const int32_t positive_default = 30;
      const int32_t other_default = 20;

      int32_t total_percent;
      int32_t num_positive_;
      int32_t num_negative_;
      int32_t num_other_;

        void add_walls(GameField *map, const Position &position,
std::vector<Wall> &walls);
      void generate_maze(GameField *map);

                            [[nodiscard]]          std::set<Position>
get_all_movable_cells(GameField *map);
                [[nodiscard]]     std::vector<EventInterface     *>
generate_events(int32_t lim, EVENT_GROUP group) const;
                                    [[nodiscard]]             Position
pick_random_empty_cell(std::set<Position>              &movable_cells,
std::function<bool(const Position &)> criteria);
```

```
                          [[nodiscard]]          std::vector<Position>
invariant_route(std::set<Position> &movable_cells, GameField *map);

        void reset_points(std::set<Position> &movable_cells, GameField
*map);
       void calculate_percentages(int32_t total_movable_cells);

         void  add_additional_keys(std::set<Position>  &movable_cells,
GameField *map);
        void  place_events(std::set<Position>  &movable_cells,  int32_t
count, EVENT_GROUP group, GameField *map);
         void  place_events_special(std::set<Position>  &movable_cells,
std::vector<Position>  &route,  int32_t  count,  EVENT_GROUP  group,
GameField *map);
      public:
       Generator() = delete;
       Generator &operator=(const Generator &generator) = delete;
       Generator &operator=(Generator &&generator) = delete;
        Generator(int32_t m, int32_t n, int32_t total_percent, int32_t
positive_events_percent,  int32_t  negative_events_percent,  int32_t
other_events_percent);
       [[nodiscard]] GameField *generate();
     };
```

Название файла: src/MapGenerator/RandomLevelGen/Generator.cpp

```
     #include "Generator.hpp"
     #include "Random/Random.hpp"
     #include "Event/Factory/EventFactory.hpp"
     #include <set>
     #include <random>
     #include <stack>
     #include <iostream>
     #include <unordered_set>
     #include <queue>
     #include <utility>

     Generator::Generator(int32_t m,
                          int32_t n,
                          int32_t total_percent,
                          int32_t positive_events_percent,
                          int32_t negative_events_percent,
                          int32_t other_events_percent) :
                          m_(m),
                          n_(n),
                          total_percent(total_percent),

num_positive_(positive_events_percent),

num_negative_(negative_events_percent),
                          num_other_(other_events_percent)
     {
```

```
}

GameField *Generator::generate()
{
	// generate maze
	auto *map = new GameField(m_, n_);
	generate_maze(map);

	// calculate movable_cells
	std::set<Position>		movable_cells		=
get_all_movable_cells(map);

	// reset start and finish
	reset_points(movable_cells, map);

	// calculate event distribution

calculate_percentages(static_cast<int32_t>(movable_cells.size()));

	// find invariant route from start to finish
	auto route = invariant_route(movable_cells, map);

	// place events
	place_events(movable_cells,		num_positive_,
EVENT_GROUP::POSITIVE, map);
	place_events(movable_cells,		num_other_,
EVENT_GROUP::NEUTRAL, map);
	place_events_special(movable_cells,	route,	std::min(4,
num_negative_), EVENT_GROUP::NEGATIVE, map);

	for (auto &cell : route)
	{
		movable_cells.erase(cell);
	}

	place_events(movable_cells, std::max(0, num_negative_ - 4),
EVENT_GROUP::NEGATIVE, map);

	add_additional_keys(movable_cells, map);

	return map;
}

void	Generator::add_walls(GameField	*map,	const	Position
&position, std::vector<Wall> &walls)
{
	if (map == nullptr)
	{
		return;
	}
	for		(auto		&direction:
Direction::instance().get_all_possible_moves())
	{
		if (auto wall_pos = position + direction, neighbour =
wall_pos + direction;
```

```
                    map->is_on_map(wall_pos)        &&        !map-
>get_cell(wall_pos).is_movable() && map->is_on_map(neighbour)
                    && !map->get_cell(neighbour).is_movable())
                {
                    walls.push_back({wall_pos, position, neighbour});
                }
            }
        }

        // Iterative randomized Prim's algorithm
        //
https://en.wikipedia.org/wiki/Maze_generation_algorithm#Iterative_rand
omized_Prim's_algorithm_(without_stack,_without_sets)
        void Generator::generate_maze(GameField *map)
        {
            if (map == nullptr)
            {
                return;
            }

            auto engine = Random::instance();

            // fill map with walls first
            for (int32_t i = 0; i < map->dimensions().x(); ++i)
            {
                for (int32_t j = 0; j < map->dimensions().y(); ++j)
                {
                    map->build_wall({i, j});
                }
            }

            std::vector<Wall> walls;
            std::unordered_set<int32_t> visited;

            int i = engine.pick_num(0, map->dimensions().x() - 1);
            int j = engine.pick_num(0, map->dimensions().y() - 1);

            map->destroy_wall({i, j});

            // Add the walls of the cell to the wall list
            add_walls(map, {i, j}, walls);

            // While there are walls in the list
            while (!walls.empty())
            {
                // Pick a random wall from the list
                auto        index        =        engine.pick_num(0,
static_cast<int32_t>(walls.size() - 1));
                auto wall = walls[index];
                walls.erase(walls.begin() + index);

                // If only one of the cells that the wall divides is
visited
                if  (map->get_cell(wall.cell1).is_movable()  !=  map-
>get_cell(wall.cell2).is_movable())
```

```
                {
                        // Make the wall a passage and mark the unvisited
cell as part of the maze
                        map->destroy_wall(wall.pos);

                        if (!map->get_cell(wall.cell1).is_movable())
                        {
                                map->destroy_wall(wall.cell1);
                                add_walls(map, wall.cell1, walls);
                        }
                        else if (!map->get_cell(wall.cell2).is_movable())
                        {
                                map->destroy_wall(wall.cell2);
                                add_walls(map, wall.cell2, walls);
                        }
                }
        }
    }

    std::set<Position>     Generator::get_all_movable_cells(GameField
*map)
    {
        std::set<Position> movable_cells;

        if (map != nullptr)
        {
            for (int32_t i = 0; i < map->dimensions().x(); ++i)
            {
                for (int32_t j = 0; j < map->dimensions().y(); +
+j)
                {
                    Position cur_point(i, j);
                    if (!map->get_cell(cur_point).is_movable())
                    {
                        continue;
                    }
                    movable_cells.insert(std::move(cur_point));
                }
            }
        }

        return movable_cells;
    }

    std::vector<EventInterface  *>  Generator::generate_events(int32_t
lim, EVENT_GROUP group) const
    {
        std::vector<EventInterface *> events;
        events.reserve(lim);

        while (events.size() < lim)
        {

    events.push_back(Random::instance().pick_event(group));
        }
```

```
            return events;
      }

      Position      Generator::pick_random_empty_cell(std::set<Position>
&movable_cells, std::function<bool(const Position &)> criteria)
      {
            std::vector<Position> possible_cells;

            std::copy_if(movable_cells.begin(),      movable_cells.end(),
std::back_inserter(possible_cells), std::move(criteria));
            return
Random::instance().pick_from_range(possible_cells.begin(),
possible_cells.end());
      }

      std::vector<Position>
Generator::invariant_route(std::set<Position>           &movable_cells,
GameField *map)
      {
            auto  route  =  map->find_route(nullptr,  map->start_point(),
map->exit_point());
            auto                        key_point                        =
Random::instance().pick_from_range(route.begin()  +  1,  route.end()  -
3);

            map->get_cell(route[route.size()                              -
2]).add_event(EventFactory::instance().create(EVENT_TYPE::DOOR));
            movable_cells.erase(route[route.size() - 2]);

            map-
>get_cell(key_point).add_event(EventFactory::instance().create(EVENT_T
YPE::KEY));
            movable_cells.erase(key_point);

            route.erase(route.begin());
            route.pop_back();
            route.pop_back();
            route.erase(std::find(route.begin(),            route.end(),
key_point));

            return route;
      }

      void  Generator::reset_points(std::set<Position>  &movable_cells,
GameField *map)
      {
            auto start_point = pick_random_empty_cell(movable_cells, [&]
(const Position &position)
            {
                  return  position.x()  *  4  <  map->dimensions().x()  &&
position.y() * 4 < map->dimensions().y();
            });

            map->reset_start(start_point);
            movable_cells.erase(start_point);
```

```
          auto end_point = pick_random_empty_cell(movable_cells, [&]
(const Position &position)
          {
               return position.x() * 4 / 3 >= map->dimensions().x()
&& position.y() * 4 / 3 >= map->dimensions().y();
          });

          map->reset_finish(end_point);
          movable_cells.erase(end_point);
     }

     void                      Generator::calculate_percentages(int32_t
total_movable_cells)
     {
          auto                      cells_available                    =
static_cast<double>(total_movable_cells);
          auto total_events = static_cast<int32_t>(cells_available *
total_percent / 100);

          if (num_positive_ + num_negative_ + num_other_ > 100)
          {
               num_positive_ = positive_default;
               num_negative_ = negative_default;
               num_other_ = other_default;
          }

          num_positive_ = static_cast<int32_t>(total_events * 1.0 *
num_positive_ / 100.0);
          num_negative_ = static_cast<int32_t>(total_events * 1.0 *
num_negative_ / 100.0);
          num_other_ = static_cast<int32_t>(total_events * 1.0 *
num_other_ / 100.0);
     }

     void            Generator::add_additional_keys(std::set<Position>
&movable_cells, GameField *map)
     {
          if (!movable_cells.empty() && map->dimensions().x() * map-
>dimensions().y() > 200)
          {
               auto   new_keys   =   map->dimensions().x()   *   map-
>dimensions().y() / 150;

               std::vector<Position>    movable(movable_cells.begin(),
movable_cells.end());
               std::vector<EventInterface *> keys_;

               while (new_keys--)
               {


keys_.push_back(EventFactory::instance().create(EVENT_TYPE::KEY));
               }
```

```
                std::shuffle(movable.begin(),           movable.end(),
std::mt19937(std::random_device()()));

                for (size_t j = 0, i = 0; j < movable.size() && i <
keys_.size(); ++j)
                {
                     if (!map->is_adjacent_to_same_type(movable[j]))
                     {
                          map->get_cell(movable[j]).add_event(keys_[i+
+]);
                          movable_cells.erase(movable[j]);
                     }
                }
           }
     }

     void  Generator::place_events(std::set<Position>  &movable_cells,
int32_t count, EVENT_GROUP group, GameField *map)
     {
          std::vector<Position>  movable_cells_(movable_cells.begin(),
movable_cells.end());

          place_events_special(movable_cells,  movable_cells_,  count,
group, map);
     }

     void           Generator::place_events_special(std::set<Position>
&movable_cells,    std::vector<Position>    &route,    int32_t    count,
EVENT_GROUP group, GameField *map)
     {
          auto remained_events = generate_events(count, group);

          std::shuffle(route.begin(),                    route.end(),
std::mt19937(std::random_device()()));
          std::shuffle(remained_events.begin(), remained_events.end(),
std::mt19937(std::random_device()()));

          for  (size_t  j  =  0,  i  =  0;  j  <  route.size()  &&  i  <
remained_events.size(); ++j)
          {
                if (!map->is_adjacent_to_same_type(route[j]))
                {
                     map-
>get_cell(route[j]).add_event(remained_events[i++]);
                     movable_cells.erase(route[j]);
                }
          }
     }
```

Название файла: src/MapGenerator/DefaultLevels/DefaultLevel Generator.hpp

```
#pragma once

#include "../RandomLevelGen/Generator.hpp"

enum MAP_SIZE
{
	SMALL,
	MEDIUM,
	BIG,
	HUGE
};

enum DIFFICULTY
{
	EASY,
	AVERAGE,
	HARD
};

class DefaultLevelGenerator {
 private:
  MAP_SIZE size_;
  DIFFICULTY difficulty_;
 public:
  DefaultLevelGenerator() = delete;
  DefaultLevelGenerator(MAP_SIZE size, DIFFICULTY difficulty);
  [[nodiscard]] GameField *generate() const;
};
```

Название файла: src/MapGenerator/DefaultLevels/DefaultLevel Generator.cpp

```
#include "DefaultLevelGenerator.hpp"

DefaultLevelGenerator::DefaultLevelGenerator(MAP_SIZE size, DIFFICULTY difficulty) : size_(size), difficulty_(difficulty)
{
}

GameField *DefaultLevelGenerator::generate() const
{
	Dimension field_dimensions;
	int32_t events_percent;
	int32_t positive_events;
	int32_t negative_events;
	int32_t other_events;

	switch (size_)
	{
		case MEDIUM:
			field_dimensions.set_x(25);
```

```
                    field_dimensions.set_y(30);
                    break;
               case BIG:
                    field_dimensions.set_x(100);
                    field_dimensions.set_y(100);
                    break;
               case HUGE:
                    field_dimensions.set_x(300);
                    field_dimensions.set_y(300);
                    break;
               case SMALL:
               default:
                    field_dimensions.set_x(10);
                    field_dimensions.set_y(10);
                    break;
          }

          switch (difficulty_)
          {
               case AVERAGE:
                    events_percent = 40;
                    positive_events = 30;
                    negative_events = 30;
                    other_events = 30;
                    break;
               case HARD:
                    events_percent = 60;
                    positive_events = 10;
                    negative_events = 70;
                    other_events = 20;
                    break;
               case EASY:
               default:
                    events_percent = 30;
                    positive_events = 60;
                    negative_events = 20;
                    other_events = 10;
                    break;
          }
          Generator   gen(field_dimensions.x(),   field_dimensions.y(),
events_percent, positive_events, negative_events, other_events);
          return gen.generate();
     }
```