**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Шаблонные классы, генерация карты.**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 1304 |  | Чернякова В.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2022

## Цель работы.

Реализовать шаблонный класс генерирующий игровое поле. Данный класс должен параметризироваться правилами генерации (расстановка непроходимых клеток, как и в каком количестве размещаются события, расположение стартовой позиции игрока и выхода, условия победы, и.т.д.). Также реализовать набор шаблонных правил (например, событие встречи с врагом размещается случайно в заданном в шаблоне параметре, отвечающим за количество событий).

## Требования.

Реализован шаблонный класс генератор поля. Данный класс должен поддерживать любое количество правил, то есть должен быть variadic template.

Класс генератор создает поле, а не принимает его.

Класс генератор не должен принимать объекты классов правил в каком-либо методе, а должен сам создавать (в зависимости от реализации) объекты правил из шаблона.

Реализовано не менее 6 шаблонных классов правил.

Классы правила должны быть независимыми и не иметь общего класса-интерфейса.

При запуске программы есть возможность выбрать уровень (не менее 2) из заранее заготовленных шаблонов.

Классы правила не должны быть только “хранилищем” для данных.

Так как используются шаблонные классы, то в генераторе не должны быть dynamic\_cast.

## Примечания.

Для задания способа генерации можно использовать стратегию, компоновщик, прототип.

Не рекомендуется делать static методы в классах правилах.

## Описание архитектурных решений и классов.

Для реализации требований лабораторной работы был создан шаблонный класс генератор поля и шаблонные классы правил. Генератор поля отвечает за генерацию поля согласно выбранным правилам.

**Новые классы.**

*Класс FieldGen:* шаблонный класс генератор поля. Объявление шаблона происходит с переменным количеством параметров – *template <class ... Rules>.* Таким образом, данный класс будет поддерживать неограниченное количество переменных. В классе реализован единственный метод – *Filed\* execute(Player\* player),* который создает игровое поле. В методе создается новый объект класса поля *auto\* field = new Field.* Используется ключевое слово auto, которое даёт возможность автоматически определять тип данных на этапе компиляции программы. В зависимости от уровня, создается поле нужного размера, по умолчанию устанавливается игрок и клетка победы. На поле устанавливается игрок *field->set\_player(player).* Далее происходит распаковка шаблона *(Rules()(field), ...).* Метод возвращает указатель на сгенерированное поле.

Шаблонные классы правил реализованы как функторы или «функциональные объекты». То есть в них определён *operator().* Таким образом получается объект, который действует как функция, но может также хранить состояние. Это позволяет соблюдать одно из требований лабораторной работы.

Каждый класс правил, когда вызывается определённый в них оператор *()* принимает указатель на поле *\*Field*, при использовании же данного оператора поле изменяется.

В каждом классе используется новый метод поля Filed – check\_way(), которая проверяет, что игрок может гарантированной дойти до клетки победы.

Так как внутри методов шаблонных классов реализуются одни и те же проверки, для корректного создания поля, они были вынесены в отдельные файлы (заголовочный и реализация) и уже будут использоваться в самих классах правилах.

*Util\_Funcs*: файл, в котором реализуются функции проверок.

* Функция *bool check\_pos(int point, int length).* Осуществляет проверку выхода за границы *return (point < 0 || point > length).*
* Функция *void placer(Cell& cell, bool with\_force).* Делает клетку непроходимой, то есть устанавливает на нее стену *cell.set\_wall(true).* Но прежде проверяет, нет ли какого-то события на клетке с помощью значения переменной *with\_force.* Если событие уже установлено, то оно удаляется *cell.set\_event(nullptr).*
* Функция *int new\_count(int cur\_count, int free\_cells).* Возвращает количество клеток, которые можно заполнить. Чтобы все поле не было заполнено стенами или событиями необходима данная функция. Она принимает на вход количество клеток, которое хотят заполнить *int cur\_count*, и количество свободных клеток поля *int free\_cells.* Высчитывается процент желаемых клеток относительно свободных *double percent = (double)count / free\_cells*. Если данное число получилось больше *0.3*, то оно будет уменьшаться, пока не станет меньше, для корректного заполнения поля *count /= 1.2,* значение *percent* соответственно меняется *percent = count / free\_cells.* Функция возвращает измененное значение количества клеток *return (int)count.*

*Шаблонный класс-правил R\_Column\_Walls*: клетки в столбце изменяются на непроходимые. Через шаблон задается столбец *int column\_index*, где будут размещены непроходимые клетки, *int y\_start* и *int y\_end* – конкретное расположение столбца непроходимых клеток по оси y, изначальное наличие событий в столбце *with\_force.* Осуществляемые проверки с помощью описанных выше функций файла *Util\_Funcs:*

* Столбец находится в пределах поля *(check\_pos(column\_index, field->get\_height() - 1).*
* Начальная координата столбца находится в пределах поля *check\_pos(y\_start, field->get\_width() - 1)).*
* Конечная координата столбца находится в пределах поля *check\_pos(y\_end, field->get\_width() - 1)).*
* Проверка, что начальная координата столбца не больше конечной *(y\_start > y\_end).*

Если все проверки пройдены, то работает цикл, заполняющий столбец стенами. Внутри осуществляется проверка, не стоит ли игрок на данной клетке, в таком случае стена не будет установлена *if (std::make\_pair(column\_index, i) == field->get\_position()) continue*. С помощью функции placer файла *Util\_Funcs,* описанного выше, устанавливается стена *placer(field->get\_cell(column\_index, i), with\_force),* последним действием вызывается метод поля, отвечающий за количество свободных клеток на нем, оно на каждом шаге итерации уменьшается на единицу *field->set\_count(field->get\_count\_free() - 1).*

*Шаблонный класс-правил R\_Player\_Spawn:* появление игрока на любой клетке поля. Через шаблон задаются координаты игрока *int x* и *int y*, по умолчанию позиция игрока клетка *(0,0).* В выбранной координате с помощью метода класса *Field* значение клетки со стеной устанавливается как *false field->get\_cell(x, y).set\_wall(false).* Также по данной координате в клетке не может быть события *field->get\_cell(x, y).set\_event(nullptr).* Создается переменная *auto pair = std::make\_pair<int, int>(x, y)* с координатами игрока и передается в поле *field->set\_player\_pos(pair).* Также в этом правиле проверяется, что координаты игрока находятся в переделах поля.

*Шаблонный класс-правил R\_Rand\_Walls:* установка определённого количества стен на поле случайным образом. Через шаблон задается класс генератора событий *Gen\_Event,* он отвечает за то, какие события будут генерироваться – связанные с игроком или с поле. Также задается количество таких события на поле *int count*. С помощью класса *std::uniform\_int\_distribution<int>* объявляются переменные *dist\_height{ 0, field->get\_height() - 1 }, dist\_width{ 0, field->get\_width() - 1 }.* Так будут созданы рандомные числа на отрезках, указанных в фигурных скобках. С помощью функций файла, описанного выше *Util\_Funcs*:

* Проверка корректности введённого количества клеток, на которые хотят добавить события *real\_count = new\_count(count,field->get\_count\_free()).*

Далее работает цикл, до тех пор, пока определенное количество клеток не будет заполнено стенами. Координатам *x* и *y* клеток присваиваются значения *generator.get\_random\_value<int>(dist\_width)* и *generator.get\_random\_value<int>(dist\_height)* соответственно. *Generator* – является объектом класса *RNGenerator*, который был создан раннее в лабораторной работе. Он отвечает за рандомную генерацию. Метод *get\_random\_value* выбирает случайное число на выбранном отрезке. Далее осуществляются проверки: не является ли клетка уже стеной *field->get\_cell(x, y).is\_wall(),* установлено ли в клетке событие *field->get\_cell(x, y).get\_event() != nullptr,* не находится ли в клетке игрок *std::make\_pair(x, y) == field->get\_position().* В случае успешного прохождения проверок устанавливается событие *field->get\_cell(x,y).set\_event(gen.generate()).* *Gen* – объект класса *Gen\_Event*, который отвечает за рандомную генерацию какого-то события на поле. Последним действием вызывается метод поля, отвечающий за количество свободных клеток на нем, оно на каждом шаге итерации уменьшается на единицу *field->set\_count(field->get\_count\_free() - 1).*

*Шаблонный класс-правил R\_Rand\_Walls:* установка определённого количества стен на поле случайным образом. Через шаблон задается количество стен *int count*. С помощью класса *std::uniform\_int\_distribution<int>* объявляются переменные *dist\_height{ 0, field->get\_height() - 1 }, dist\_width{ 0, field->get\_width() - 1 }*. Так будут созданы рандомные числа на отрезках, указанных в фигурных скобках. С помощью функции файла, описанного выше *Util\_Funcs:*

* Проверка корректности введённого количества клеток, которые хотят сделать непроходимыми стенами *real\_count= new\_count(count, field->get\_count\_free()).*

Далее работает цикл, до тех пор, пока определенное количество клеток не будет заполнено стенами. Координатам *x* и *y* клеток присваиваются значения *gen.get\_random\_value<int>(dist\_width)* и *gen.get\_random\_value<int>(dist\_height)* соответственно. *Gen* – является объектом класса *RNGenerator*, который был создан раннее в лабораторной работе. Он отвечает за рандомную генерацию. Метод *get\_random\_value* выбирает случайное число на выбранном отрезке. Далее осуществляются проверки: не является лик летка уже стеной *field->get\_cell(x, y).is\_wall(),* установлено ли в клетке событие *field->get\_cell(x, y).get\_event() != nullptr*, не находится ли в клетке игрок *std::make\_pair(x, y) == field->get\_position().* В случае успешного прохождения проверок устанавливается стена *field->get\_cell(x, y).set\_wall(true).* Последним действием вызывается метод поля, отвечающий за количество свободных клеток на нем, оно на каждом шаге итерации уменьшается на единицу *field->set\_count(field->get\_count\_free() - 1).*

*Шаблонный класс-правил R\_Row\_Walls*: клетки в строке изменяются на непроходимые. Через шаблон задается строка *int row\_index*, где будут размещены непроходимые клетки, *int x\_start* и *int x\_end* – конкретное расположение строки непроходимых клеток по оси *x*, изначальное наличие событий в сроке *with\_force*. Осуществляемые проверки с помощью описанных выше функций файла *Util\_Funcs:*

* Строка находится в пределах поля *(check\_pos(row\_index, field->get\_height() - 1).*
* Начальная координата строки находится в пределах поля *check\_pos(x\_start, field->get\_width() - 1)).*
* Конечная координата строки находится в пределах поля *check\_pos(x\_end, field->get\_width() - 1)).*
* Проверка, что начальная координата строки не больше конечной (*x\_start > x\_end).*

Если все проверки пройдены, то работает цикл, заполняющий строку стенами. Внутри осуществляется проверка, не стоит ли игрок на данной клетке, в таком случае стена не будет установлена *if (std::make\_pair(row\_index, i) == field->get\_position()) continue.* С помощью функции *placer* файла *Util\_Funcs*, описанного выше, устанавливается стена *placer(field->get\_cell(row\_index, i), with\_force),* последним действием вызывается метод поля, отвечающий за количество свободных клеток на нем, оно на каждом шаге итерации уменьшается на единицу *field->set\_count(field->get\_count\_free() - 1).*

*Шаблонный класс-правил R\_Win\_Cell:* установки клетки победы. Через шаблон задаются координаты клетки-победы *int x* и *int y*. Осуществляемые проверки с помощью описанных выше функций файла *Util\_Funcs:*

* Проверка, что координата по *x* находится в пределах поля *check\_pos(x, field->get\_width() - 1).*
* Проверка, что координата по *y* находится в пределах поля *check\_pos(y, field->get\_height() - 1).*
* Проверка, что игрок не находится на данной клетке *std::make\_pair(x, y) == field->get\_position()).*

Если все проверки пройдены, то вызывается метод поля, отвечающий за установку в клетку стены, передается значение *false*, чтобы стены там не было *field->get\_cell(x, y).set\_wall(false).* Далее в клетку устанавливается события победы *field->get\_cell(x, y).set\_event(new TreasureEvent(field->get\_player())).* Последним действием вызывается метод поля, отвечающий за количество свободных клеток на нем, оно на каждом шаге итерации уменьшается на единицу *field->set\_count(field->get\_count\_free() - 1).*

**Изменения в классах.**

*Класс IOCommander*. Добавлен метод, считывающий номер уровня void *read\_level\_num().* Информация о прочтении уровня передается соответственно медиатору для дальнейших операций *mediator->notify(this, IMediator::LEVEL).*

Также добавлен метод, возвращающий значение номера выбранного уровня *int get\_level() const.*

*Класс Mediator*. В методе *void commander\_handler(IMediator::MEVENTS cmd)* в конструкции *switch-case* добавлено сравнение с константным значением *case IMediator::LEVEL*. В данном случае будет вызываться метод поля, предназначенный для инициализации поля в соответствии с выбранным уровнем *game->initialize\_field(commander->get\_level()).*

*Класс Game.* Добавлено новое поле класса *std::map<int, std::function<Field\* (Player\*)>> levels\_gens* – контейнер *map*, где ключом является номер уровня, а значением выступает лямбда выражение генерирующее поле.

Метод *void initialize\_field(int level\_num)* теперь принимает на вход номер уровня. Внутри метода по умолчанию переменная, отвечающая за уровень, который должен быть сгенерирован, равна первому *auto it = levels\_gens[1].* Переменная меняет значение, если при проверке *levels\_gens.count(level\_num) != 0,* то есть в контейнере такой уровень есть, тогда *it = levels\_gens[level\_num].*

В конструкторе класса *Game()* добавлена генерация двух уровней. Пример создания:

*levels\_gens[i] = [](Player\* pl) {*

*FieldGen<перечисление шаблонных классов правил> gen;*

*return gen.execute(pl); //создание поля согласно правилам*

*};*

**Вывод.**

Реализован шаблонный класс генератор поля. Также реализованы шаблонные классы-правил, влияющие на генерацию поля.

# ПРИЛОЖЕНИЕ.

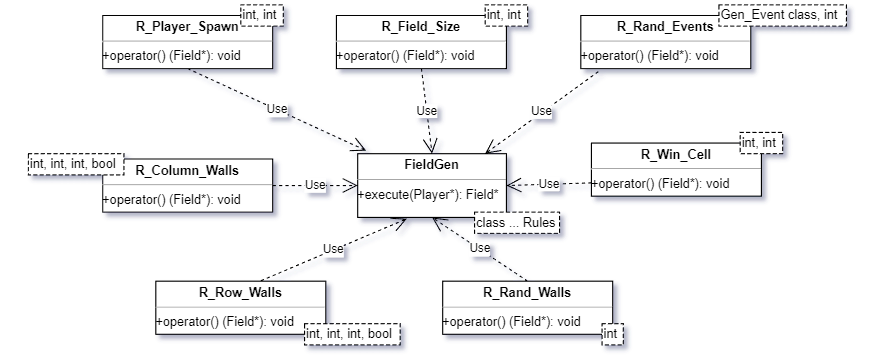
****

Рисунок 1 – UML-диаграмма межклассовых отношений.