Отчет лабораторной работе Поиск выхода из лабиринта (объектно-ориентированная реализация с паттернами)

Задача

Разработать программу для загрузки лабиринта из файла, поиска пути от старта до выхода с возможностью выбора алгоритма, визуализации процесса и экспериментального сравнения алгоритмов. При этом для обеспечения гибкости и расширяемости программу реализовать в парадигме ООП с использованием паттернов проектирования из списка GoF.

Выбранные паттерны

1) Builder для загрузки лабиринта из файла.

Он позволяет вынести конструирование объекта(лабиринта) за пределы его собственного класса, поручив это дело отдельным объектам. Кроме того этот паттерн позволяет создать задел на будущее, можно будет дописать другого строителя, который будет работать с другими типами файлов. В программе роль "семейства" строителей играет интерфейс **MazeBuilder**.

2) Strategy для создания семейства алгоритмов поиска пути выхода.

Определяет семейство взаимозаменяемых алгоритмов, определяемых во время выполнения. Таких алгоритмов может быть сколько угодно, чтобы добавить новый, не нужно переписывать весь код, а лишь дописать новый алгоритм и подключить его в интерфейс к другим алгоритмам. В программе семейство алгоритмов реализуется через интерфейс **PathFindingStrategy**.

3) Класс-оркестратор для сбора статистики.

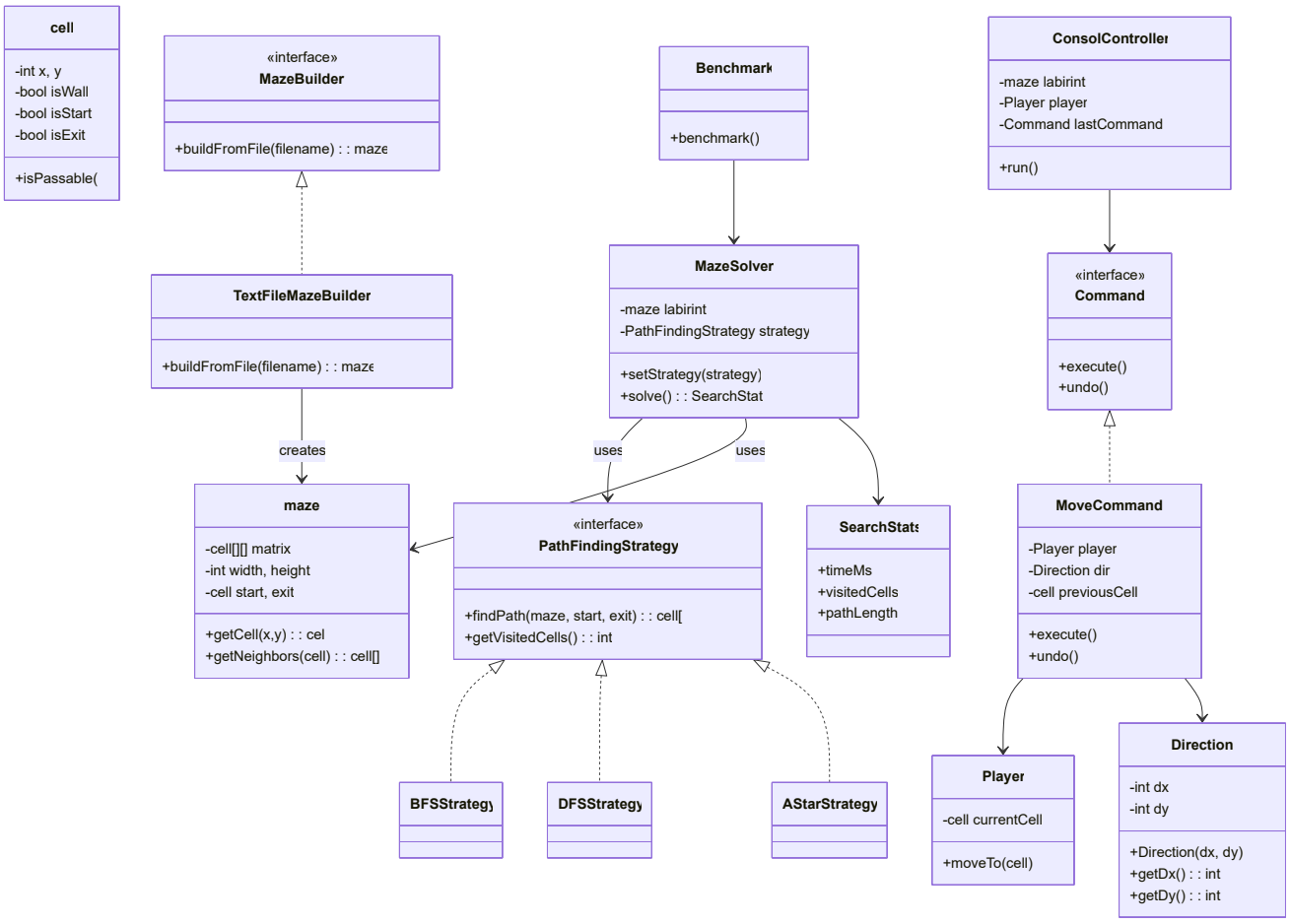
В соответствии с этим паттерном, такой класс управляет каким-либо процессом в общем, но какие-то конкретные задачи делегирует другим классам. В программе таких класса 3:
1) **MazeSolver** управляет процессом решения для одного лабиринта и одной стратегии
2) **Benchmark** управляет не одним поиском, а всей серией запусков.
3) **ConsoleController** координирует взаимодействие пользователя с программой(перемещение игрока по лабиринту)

4) Command для пошагового перемещения игрока по лабиринту.

Смысл этого паттерна представить действие как отдельный объект. По аналогии со строителем или стратегией, можно создать семейство команд. В нашем случае,

команда может отменять последнее действие(себя). В программе это интерфейс **Command**

Общая схема программы:



Листинги ключевых классов

Стоит упомянуть, что в программе во всех классах работа идет не с копиями объектов, а непосредственно с ними. Поэтому везде в программе поля классов зачастую либо указатели, либо указатели на указатели. Поэтому схема выше не буквально отображает то, как устроен класс, а схематически.

Класс Лабиринта

```
class maze{
    private:
        int width;
        int height;
        cell** matrix;

        cell* start;
        cell* exit;

    public:
        maze(int width, int height)
        {
```

```
            this->width = width;
            this->height = height;
            this->start = nullptr;
            this->exit = nullptr;

            matrix = new cell*[width];

            for (int x = 0; x < width; ++x) {
                matrix[x] = new cell[height];
                for (int y = 0; y < height; ++y)
                    matrix[x][y] = cell(x, y, false, false, false);
            }
        }
        maze(int width, int height, int startX, int startY, int exitX, int
exitY)
        {
            this->width = width;
            this->height = height;
            this->start = nullptr;
            this->exit = nullptr;

            matrix = new cell*[width];
            for (int x = 0; x < width; ++x) {
                matrix[x] = new cell[height];
                for (int y = 0; y < height; ++y) {
                    matrix[x][y] = cell(x, y, false, false, false);
                }
            }


            matrix[startX][startY].setStart(true);
            matrix[exitX][exitY].setExit(true);

            start = &matrix[startX][startY];
            exit = &matrix[exitX][exitY];
        }

        ~maze()
        {
        for (int i = 0; i < width; ++i)
            delete[] matrix[i];
        delete[] matrix;
        }

        cell* getCell(int x, int y) {
            if (x < 0 || x >= width || y < 0 || y >= height)
                return nullptr;

            return &matrix[x][y];
        }
```

```
void setCell(int x, int y, cell newCell) {
    if (x < 0 || x >= width || y < 0 || y >= height)
        return;

    matrix[x][y] = newCell;

    if (matrix[x][y].getIsStart())
        start = &matrix[x][y];

    if (matrix[x][y].getIsExit())
        exit = &matrix[x][y];
}

cell** getNeighbors(cell* current) {
    cell** neighbors = new cell*[5];
    int count = 0;

    int x = current->getX();
    int y = current->getY();

    cell* up = getCell(x, y - 1);
    cell* down = getCell(x, y + 1);
    cell* left = getCell(x - 1, y);
    cell* right = getCell(x + 1, y);

    if (up != nullptr && up->isPassable()) {
        neighbors[count] = up;
        count++;}

    if (down != nullptr && down->isPassable()) {
        neighbors[count] = down;
        count++;}

    if (left != nullptr && left->isPassable()) {
        neighbors[count] = left;
        count++;}

    if (right != nullptr && right->isPassable()) {
        neighbors[count] = right;
        count++;}

    neighbors[count] = nullptr;

    return neighbors;
}

cell* getStart() {return start;}
cell* getExit() {return exit;}
int getWidth() {return width;}
```

```
        int getHeight() {return height;}
};
```

Класс строитель для загрузки лабиринта из файла

```
class TextFileMazeBuilder : public MazeBuilder {
    public:
        maze* buildFromFile(const std::string& filename) override
        {

            std::ifstream file(filename);
            if (!file.is_open())
                throw std::runtime_error("Ошибка: Не удалось открыть
файл!");

            std::string line;
            int width = 0;
            int height = 0;

            while (std::getline(file, line)) {
                if (height == 0) {
                    width = line.length();
                }
                else {
                    if (line.length() != width)
                        throw std::runtime_error("Ошибка: строки лабиринта
разной длины!");
                }
                height++;
            }

            if (width == 0 || height == 0)
                throw std::runtime_error("Ошибка: файл пустой!");

            file.clear();
            file.seekg(0);

            maze* labirint = new maze(width, height);
            bool hasStart = false;
            bool hasExit = false;
            int y = 0;


            while (std::getline(file, line)) {

                for (int x = 0; x < width; x++) {
                    char ch = line[x];

                    bool isWall = false;
```

```
                    bool isStart = false;
                    bool isExit = false;

                    switch(ch){
                        case '#':
                            isWall = true;
                            break;
                        case ' ':
                            isWall = false;
                            break;
                        case 'S':
                            isStart = true;

                            if (hasStart)
                                throw std::runtime_error("Ошибка: в
лабиринте больше одного старта!");

                            hasStart = true;
                            break;
                        case 'E':
                            isExit = true;

                            if (hasExit)
                                throw std::runtime_error("Ошибка: в
лабиринте больше одного выхода!");
                            hasExit = true;
                            break;
                        default:
                            throw std::runtime_error("Ошибка: неизвестный
символ в файле!");
                            break;
                    }

                    cell current(x, y, isWall, isExit, isStart);
                    labirint->setCell(x, y, current);
                }

                y++;
            }
            file.close();
            if (!hasStart)
                throw std::runtime_error("Ошибка: в лабиринте нет старта!");
            if (!hasExit)
                throw std::runtime_error("Ошибка: в лабиринте нет выхода!");
            return labirint;
        }
};
```

Класс для BFF стратегии + класс для восстановления и записи пути.

```
class PathBuilder {
    public:
        static cell** buildPath(cell* start, cell* exit, cell*** parent) {
            int length = 0;
            cell* current = exit;

            while (current != nullptr) {
                length++;
                if (current == start)
                    break;
                current = parent[current->getX()][current->getY()];
            }

            cell** path = new cell*[length + 1];
            current = exit;

            for (int i = length - 1; i >= 0; i--) {
                path[i] = current;
                if (current == start)
                    break;
                current = parent[current->getX()][current->getY()];
            }
            path[length] = nullptr;
            return path;
        }
};

class BFSStrategy : public PathFindingStrategy {
    private:
        int visitedCells;
    public:

        BFSStrategy() {visitedCells = 0;}

        int getVisitedCells() override {return visitedCells;}

        cell** findPath(maze* m, cell* start, cell* exit) override {
            visitedCells = 0;
            int width = m->getWidth();
            int height = m->getHeight();
            bool** visited = new bool*[width];
            cell*** parent = new cell**[width];

            for (int x = 0; x < width; x++) {
                visited[x] = new bool[height];
                parent[x] = new cell*[height];

                for (int y = 0; y < height; y++) {
                    visited[x][y] = false;
```

```
            parent[x][y] = nullptr;
        }
    }

    cell** deque = new cell*[width * height];
    int head = 0;
    int tail = 0;

    deque[tail] = start;
    tail++;
    visited[start->getX()][start->getY()] = true;
    bool found = false;

    while (head < tail) {
        cell* current = deque[head];
        head++;
        visitedCells++;

        if (current == exit) { //сравниваются указатели
            found = true;
            break;
        }

        cell** neighbors = m->getNeighbors(current);

        for (int i = 0; neighbors[i] != nullptr; i++) {
            cell* next = neighbors[i];

            int nx = next->getX();
            int ny = next->getY();

            if (!visited[nx][ny]) {
                visited[nx][ny] = true;
                parent[nx][ny] = current;

                deque[tail] = next;
                tail++;
            }
        }
        delete[] neighbors;
    }

    cell** path;

    if (found) {
        path = PathBuilder::buildPath(start, exit, parent);
    }
    else {
        path = new cell*[1];
        path[0] = nullptr;
```

```
            }

            delete[] deque;

            for (int x = 0; x < width; x++) {
                delete[] visited[x];
                delete[] parent[x];
            }
            delete[] visited;
            delete[] parent;
            return path;
        }
};
```

Один из классов оркестраторов (MazeSolver)

```
class MazeSolver{
    private:
        maze* labirint;
        PathFindingStrategy* strategy;
    public:
        MazeSolver(maze* labirint) {this->labirint = labirint; this->strategy = nullptr;}

        void setStrategy(PathFindingStrategy* strategy){
            this->strategy = strategy;
        }

        SearchStats solve(){
            auto start = std::chrono::high_resolution_clock::now();
            cell** path = strategy->findPath(labirint,labirint->getStart(),labirint->getExit());
            auto end = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double, std::milli> duration = end - start;

            int pathLength = 0;

            if (path[0] != nullptr)
                while (path[pathLength] != nullptr) {pathLength++;}

            int visitedCells = 0;
            visitedCells = strategy->getVisitedCells();
            delete[] path;

            SearchStats stats(duration.count(), visitedCells, pathLength);
            return stats;
```

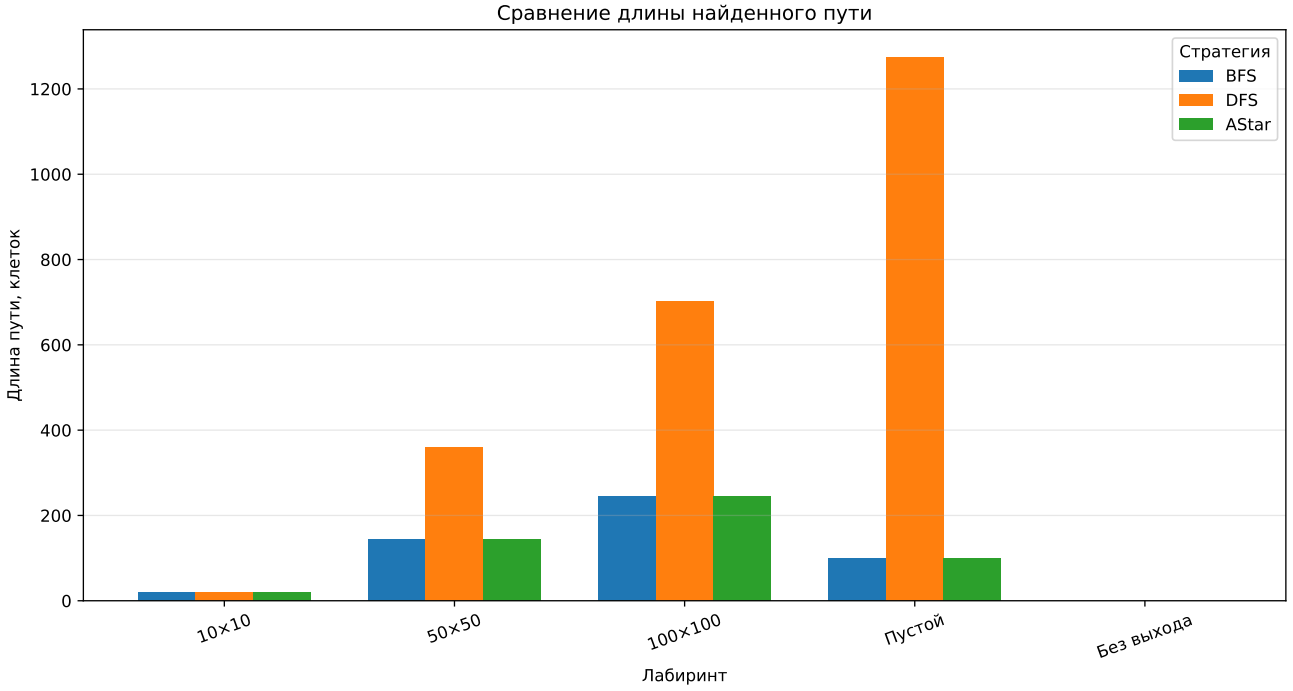
```
        }
};
```

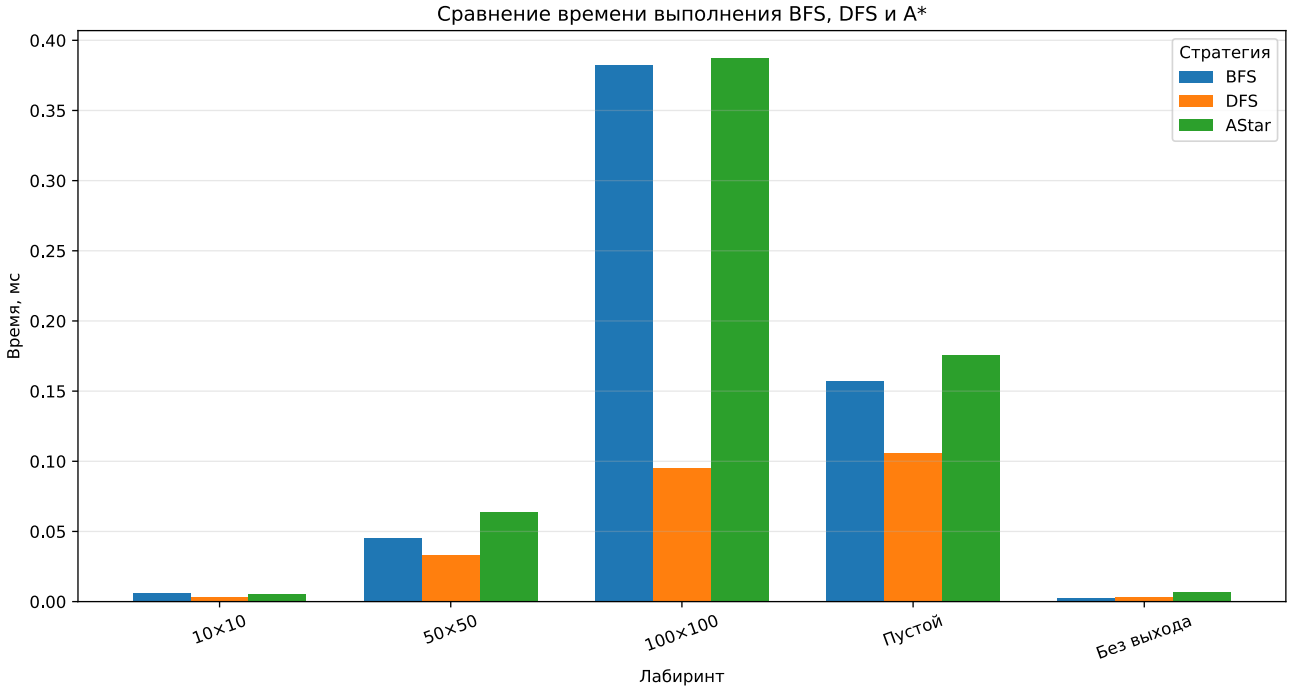
Класс Команды для движения

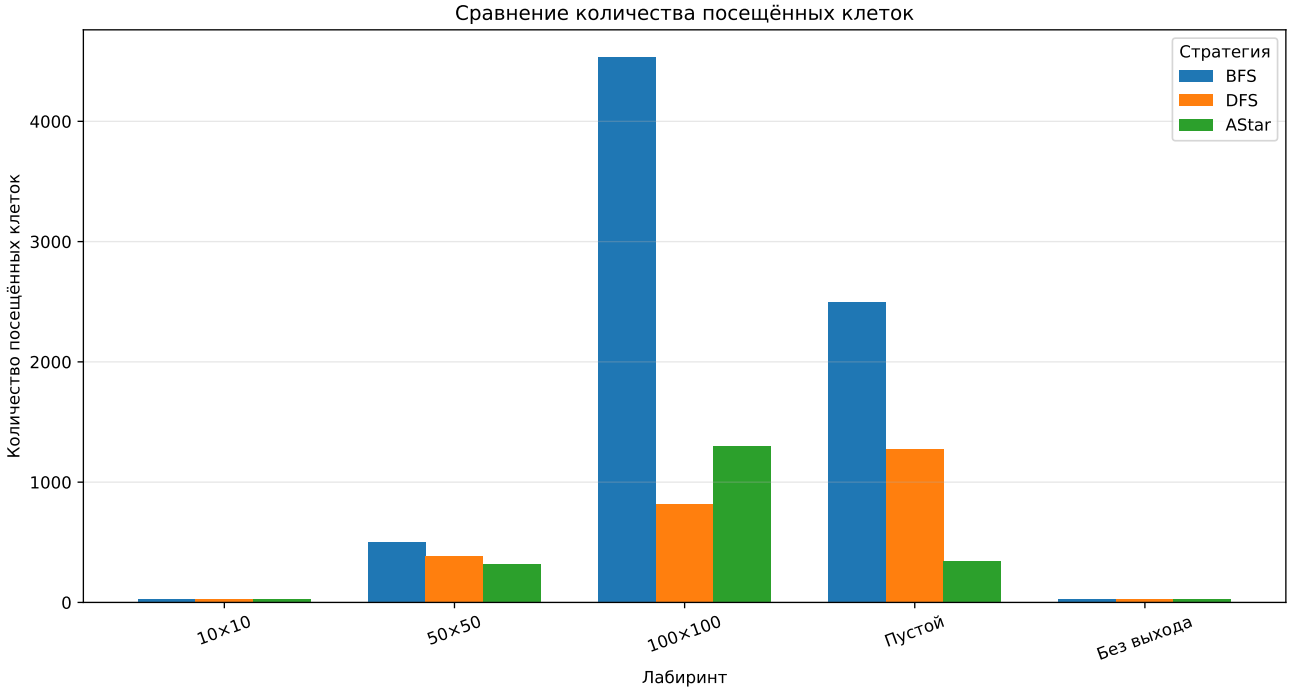
```
class MoveCommand : public Command{
    private:
        Player* player;
        Direction dir;
        cell* previousCell;
        maze* labirint;
    public:
        MoveCommand(Player* player, maze* labirint, Direction dir) :
dir(dir) {
            this->player = player;
            this->labirint = labirint;
            this->previousCell = nullptr;
        }

        void execute() override {
            cell* currentCell = player->getCurrent();

            int newX = currentCell->getX() + dir.getDx();
            int newY = currentCell->getY() + dir.getDy();

            cell* nextCell = labirint->getCell(newX, newY);

            if (nextCell != nullptr && nextCell->isPassable()) {
                previousCell = currentCell;
                player->moveTo(nextCell);
            }
            else {
                std::cout << "Нельзя сделать ход!" << std::endl;
            }
        }

        void undo() override {
            if (previousCell != nullptr) {
                player->moveTo(previousCell);
                previousCell = nullptr;
            }
        }
};
```

Результаты экспериментов:

Лабиринт	Стратегия	Время, мс	Посещено клеток	Длина пути
Маленький 10×10	BFS	0.00724	31	21
Маленький 10×10	DFS	0.00360	31	21
Маленький 10×10	AStar	0.00519	24	21
Средний 50×50	BFS	0.04465	505	145
Средний 50×50	DFS	0.03666	385	361
Средний 50×50	AStar	0.05370	319	145
Большой 100×100	BFS	0.44010	4534	245
Большой 100×100	DFS	0.09760	816	703
Большой 100×100	AStar	0.37331	1298	245
Пустой 50×50	BFS	0.15303	2500	99
Пустой 50×50	DFS	0.09335	1275	1275
Пустой 50×50	AStar	0.17047	341	99
Без выхода	BFS	0.00259	25	0
Без выхода	DFS	0.00244	25	0
Без выхода	AStar	0.00494	25	0







Анализ эффективности алгоритмов

В ходе эксперимента сравнивались три алгоритма поиска пути: BFS, DFS и `A*`. По результатам видно, что BFS и `A*` находят кратчайший путь, поэтому длина маршрута у них обычно совпадает. DFS не гарантирует кратчайший путь, поэтому на большом и пустом лабиринтах он построил значительно более длинный маршрут.

Недостатком BFS является большое количество посещенных клеток, потому что алгоритм проверяет лабиринт (граф) "слоями", проходясь по свободным клеткам(иначе говоря, на слое 1 все клетки находятся на расстоянии 1 от старта, на слое 2 - на расстоянии 2 и так далее).

DFS по результатам измерений посещает меньше клеток. Это связано с тем, что он не проверяет лабиринт (граф) не слоями, а ветками. То есть он быстро углубляется в лабиринт. И если ему "повезет"(будет подходящий лабиринт), то он быстро найдет путь до выхода. При этом путь будет "какой-то", то есть не обязательно самый короткий. Это также зависит от того, какая соседняя клетка была добавлена последней в список соседей текущей клетки(верх, низ, лево, право), ведь DFS берет именно ее и ищет уже ее соседей. Особенно хорошо это видно на примере пустого лабиринта.

Алгоритм `A*` выбирает следующую клетку с помощью критерия f=g+h, где
g - уже пройденный путь
h - Эвристика; `h = abs(x1 - x2) + abs(y1 - y2)`, где x1,y1 - текущая клетка, x2y2 - клетка выхода. Алгоритм `A*` выбирает ту клетку, у которой наименьший параметр f. Такой алгоритм балансирует между DFS и BFS по количеству пройденных клеток и длине пути.

На практике результаты также зависят от самих лабиринтов. Например, в пустом лабиринте `A*` посетил меньше всего клеток, в лабиринте 100x100 оказался в середине между BFS и DFS. Во всех трех случаях, где путь существовал, `A*` дал самый короткий путь. Однако, если оценка f будет "переоценена"(например, за счет неидеальной эвристики), `A*` может потерять гарантию нахождения кратчайшего пути.

Большое количество времени поиск пути занимает у алгоритма BFS, поскольку он проходит самое большое количество клеток. DFS тратит меньше времени по вышеуказанным причинам.

Алгоритм `A*` в теории также должен занимать относительно мало времени, но на практике время сопоставимо с BFS. Это происходит потому, что в программе, чтобы реализовать очередь с приоритетом, на которой основан `A*` (выбор клетки с наименьшим f), перед каждым выбором следующей клетки происходит поиск наименьшего f у всех клеток, которые пока что не были посещены, но имеют к этому возможность. Эта часть:

```
            while (openCount > 0) {
                int bestIndex = 0;
                for (int i = 1; i < openCount; i++) {
                    int ix = open[i]->getX();
                    int iy = open[i]->getY();
                    int bx = open[bestIndex]->getX();
                    int by = open[bestIndex]->getY();

                    if (fScore[ix][iy] < fScore[bx][by]) {
                        bestIndex = i; /*(fSchore наименьший в
[bestIndex])*/
                    }
                }
```

замедляет работу, в худшем случае нам нужно пройти весь список потенциальных к посещению клеток. И делать это нужно перед каждым выбором следующей клетки. То есть, из-за реализации очереди с приоритетом "руками", время работы увеличивается.

Анализ эффективности паттернов

Паттерн Builder позволил отделить процесс создания лабиринта от остальных частей программы. Класс **TextFileMazeBuilder** отвечает за чтение файла, проверку размеров строк, поиск старта и выхода, обработку символов и создание объекта **maze**. Благодаря этому остальные классы не зависят от формата входного файла и работают уже с готовым лабиринтом.

Паттерн Strategy позволил вынести алгоритмы поиска пути в отдельные классы с общим интерфейсом. **MazeSolver** не зависит от конкретного алгоритма и работает через указатель на **PathFindingStrategy**. Благодаря этому BFS, DFS и `A*` можно подключать одинаковым способом через **setStrategy()**.

Паттерн **Command** позволил представить перемещение игрока как отдельный объект команды. **MoveCommand** содержит логику выполнения хода и его отмены. Благодаря этому **ConsolController** не выполняет перемещение напрямую, а создаёт команду и вызывает у неё **execute()** либо **undo()**.

Выбранные паттерны позволили сделать код программы "чище" (за счет классов-оркестраторов), а также предрасположенным к расширениям(за счет паттернов Builder и Strategy). Также эти два паттерна позволяют стандартизировать код, за счет чего с ним легче работать.

Вывод

Использование ООП позволило разделить программу на отдельные классы, каждый из которых отвечает за свою часть логики

Паттерн *Builder* помог вынести создание лабиринта в отдельный класс. Благодаря этому код чтения файла, проверки символов, поиска старта и выхода не смешивается с логикой поиска пути. Без этого паттерна при добавлении нового формата лабиринта пришлось бы менять основной код программы.

Паттерн *Strategy* сделал алгоритмы поиска взаимозаменяемыми, за счет общего интерфейса.

Паттерн *Command* оказался полезен для консольного управления игроком. Действие перемещения вынесено в отдельный класс **MoveCommand**, который умеет выполнить ход и отменить его. Без этого паттерна логика перемещения, проверки возможности хода и отмены действия была бы смешана внутри **ConsoleController**, и добавлять новые действия игрока было бы сложнее.

Таким образом, ООП и паттерны сделали программу более гибкой и расширяемой. Отдельные части программы можно изменять независимо: можно добавить новый алгоритм поиска, способ загрузки лабиринта или новую команду игрока без переработки всей архитектуры.