Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Вятский государственный университет»

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

Разработка приложение на основе ООП парадигмы

Отчет по лабораторной работе №6 дисциплины
«Технологии программирования»

Выполнил студент группы ИВТ-22_____/Крючков И. С/
Проверил_____/Долженкова М. Л./

Киров 2022

1. Задание

В выбранной предметной области создать иерархию классов состоящую минимум из одного родительского и двух дочерних классов. В каждом классе определить минимум два собственных член данных, две собственных, две унаследованных и две перекрытых член функции. Разработать приложение демонстрирующее принципы полиморфизма, наследования и инкапсуляции.

2. Иерархия классов

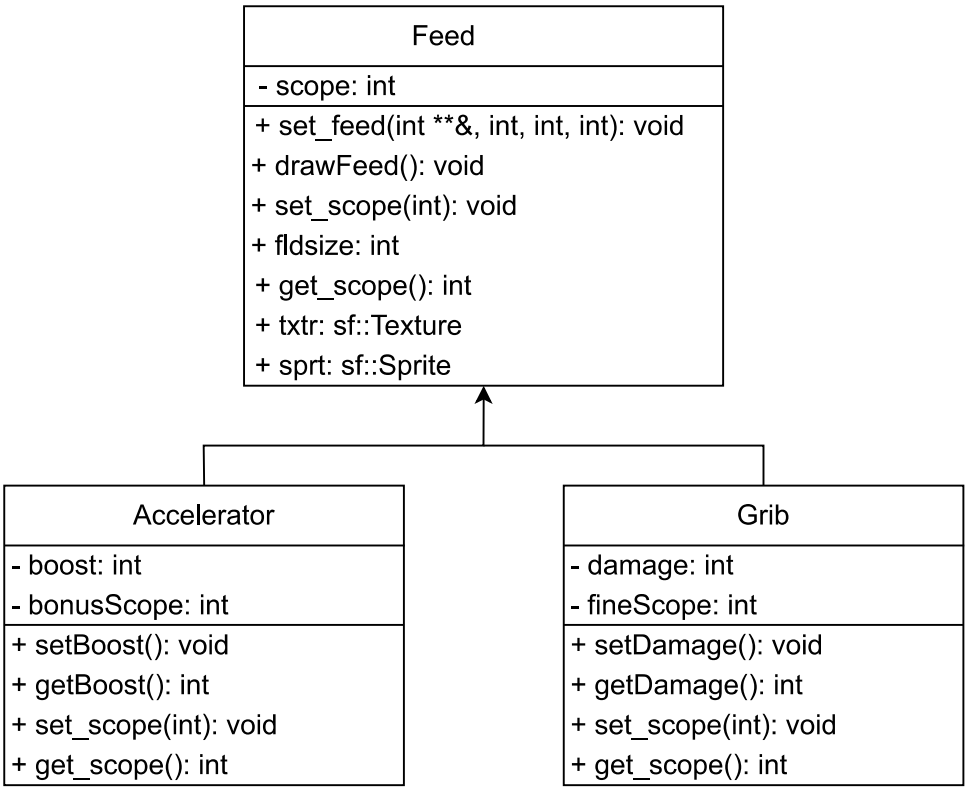

Рисунок 1 – Иерархия классов

3. Экранные формы

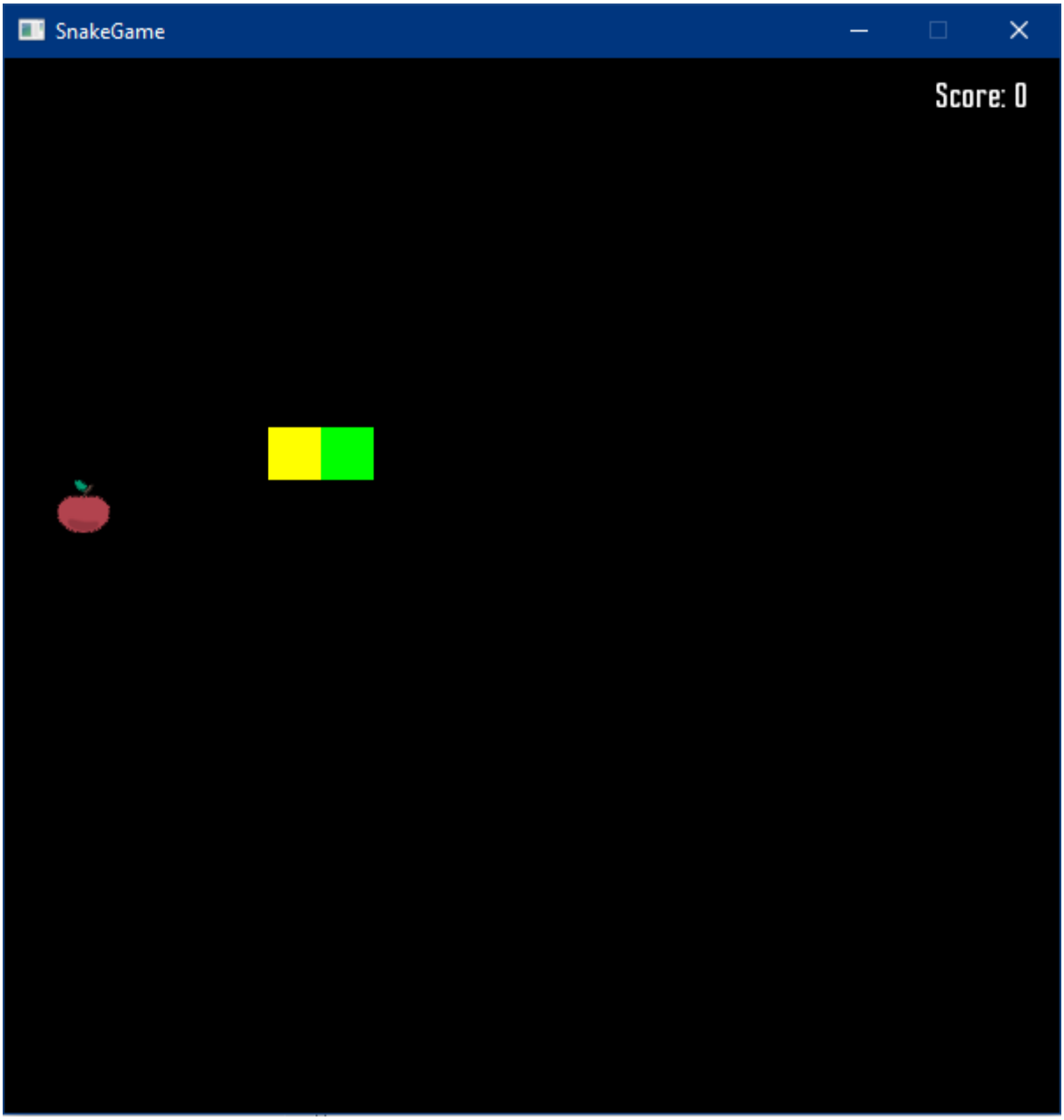

Рисунок 2 – Главное окно программы

Рисунок 2 – Проигрыш

4. Листинг кода

feed.h

```
#include <SFML/Graphics.hpp>

class Feed {
    public:
        Feed(sf::RenderWindow *, int);
        Feed() = default;
        void set_feed(int **&, int, int,
int);
        virtual void set_scope(int);
        virtual int get_scope();
        void drawFeed();
        sf::RenderWindow *window;
        int fldsize;
        sf::Texture txtr;
        sf::Sprite sprt;
    private:
        int scope;
};

class Accelerator : public Feed {
    public:
        Accelerator(sf::RenderWindow   *,
int);
        Accelerator() = default;

        // перекрытые
        void set_scope(int);
        int get_scope();

        //унаследованные
        // void drawFeed();
        //  void  set_feed(int  **&,  int,
int, int);

        // собственные
        int getBoost();
        void setBoost();
    private:
        int boost;
        int bonusScope;
};

class Grib : public Feed {
    public:
        Grib(sf::RenderWindow *, int);
        Grib() = default;

        // перекрытые
        void set_scope(int);
        int get_scope();

        // унаследованные
        // void drawFeed();
        //  void  set_feed(int  **&,  int,
int, int);

        // собственные
        void setDamage();
        int getDamage();

    private:
        int damage;
        int fineScope;
};
```

feed.cpp

```
#include "feed.h"
#include <iostream>

Feed::Feed(sf::RenderWindow    *w,    int
field_size){
    window = w;
    fldsize = field_size;
    scope = 50;


if(!txtr.loadFromFile("./img/apple.png")){
        std::cout << "img apple.png load
error" << std::endl;
    }
    sprt.setTexture(txtr);
}

void  Feed::set_feed(int  **&field,  int
cell_size, int field_size, int type_feed){
    struct position
    {
        int x;
        int y;
    };

    position rand_coords;
    std::vector<position> coords;

    for(int i = 0; i < field_size; i++){
        for(int  j  =  0;  j  <  field_size;
```

```
j++){
                    if(field[i][j] == 0){
                        coords.push_back({i, j});
                    }
                }
            }

            rand_coords  =  coords[std::rand()  %
(coords.size()-1)];
            field[rand_coords.x][rand_coords.y] =
type_feed;

sprt.setPosition(rand_coords.x*cell_size,
rand_coords.y*cell_size);
        }

        void Feed::drawFeed(){
            window->draw(sprt);
        }

        void Feed::set_scope(int sc){
            scope = sc;
        }

        int Feed::get_scope(){
            return scope;
        }

        Accelerator::Accelerator(sf::RenderWindow
*w, int field_size){
            window = w;
            fldsize = field_size;
            bonusScope = 200;


if(!txtr.loadFromFile("./img/accelerator.png")){
                std::cout << "img accelerator.png
load error" << std::endl;
            }
            sprt.setTexture(txtr);
        }

        void Accelerator::set_scope(int sc){
            bonusScope = sc;
        }
```

snake.h

```
        #include <SFML/Graphics.hpp>

        class Snake{
            public:
                Snake(sf::RenderWindow   *,   int,
int, int **&);
                int moveSnake(int, int **&);
                void drawSnake();
                void setTurnSnake(int);
                int getDirection();
```

```
        int Accelerator::get_scope(){
            return bonusScope;
        }

        void Accelerator::setBoost(){
            int b[3] = { 100, 150, 200 };
            boost = b[std::rand() % 3];
        }

        int Accelerator::getBoost(){
            return boost;
        }

        Grib::Grib(sf::RenderWindow    *w,    int
field_size){
            window = w;
            fldsize = field_size;
            fineScope = -200;


if(!txtr.loadFromFile("./img/grib.png")){
                std::cout  <<  "img  grib.png  load
error" << std::endl;
            }
            sprt.setTexture(txtr);
        }

        void Grib::set_scope(int sc){
            fineScope = sc;
        }

        int Grib::get_scope(){
            return fineScope;
        }

        void Grib::setDamage(){
            int d[3] = { 1, 3, 5 };
            damage = d[std::rand() % 3];
        }

        int Grib::getDamage(){
            return damage;
}
```

```
            void  dec_snake_length(int,  int,
int **&);
            void resetSnake(int, int **&);

        private:
            sf::RenderWindow *window;
            int      turn_direction;      //
направление поворота
            int direction;
            std::vector<sf::RectangleShape>
snake_body;

            struct position {
                int x;
                int y;
            };

            std::vector<position> snake_pos;
            sf::Color colorHead;
            sf::Color colorBody;
            int snake_length;
            int fldsize;
            position last_pos_tail;
            sf::RectangleShape rectangle;

            void  inc_snake_length(int,  int,
int, sf::Color, int **&);
            int  checkIntersections(int,  int,
int **&);
};
```

snake.cpp

```
#include "snake.h"
#include <iostream>
#include <algorithm>

Snake::Snake(sf::RenderWindow   *w,   int
cell_size, int field_size, int **&field){
    int   x_pos,   y_pos,   x_pos_window,
y_pos_window;
    fldsize = field_size;
    window = w;
    snake_length = 0;

    // направление движения при повороте
    // (0-nothing; 1-U; 2-R; 3-D; 4-L)
    turn_direction = 0;
    direction = 2; // текущее направление
движения
    colorHead = sf::Color::Yellow;
    colorBody = sf::Color::Green;

    rectangle   =   sf::RectangleShape   {
sf::Vector2f(cell_size, cell_size) };

    // голова
    x_pos = fldsize/2;
    y_pos = fldsize/2;
    inc_snake_length(x_pos,        y_pos,
cell_size, colorHead, field);

    //ячейка тела
    x_pos = fldsize/2 - 1;
    y_pos = fldsize/2;
    inc_snake_length(x_pos,        y_pos,
cell_size, colorBody, field);
    last_pos_tail = {x_pos-1, y_pos};
}

void Snake::drawSnake(){
    for(int i = 0; i < snake_length; i++){
        window->draw(snake_body[i]);
    }
}

void  Snake::inc_snake_length(int  x_pos,
int y_pos, int cell_size, sf::Color color, int
**&field){
    int x_pos_window, y_pos_window;
    x_pos_window = x_pos * cell_size;
    y_pos_window = y_pos * cell_size;
    rectangle.setPosition(x_pos_window,
y_pos_window);
    rectangle.setFillColor(color);
    snake_body.push_back(rectangle);
    snake_pos.push_back({x_pos, y_pos});
    field[x_pos][y_pos] = 1;
    snake_length++;
}

void  Snake::dec_snake_length(int  damage,
int cell_size, int **&field){
    int del_num, x_pos, y_pos, i_del_num;

    i_del_num = snake_length - 2;
    del_num     =     std::min(i_del_num,
damage);

    for(int i = 0; i < del_num; i++){
        snake_body.pop_back();
        x_pos = snake_pos.back().x;
```

```
            y_pos = snake_pos.back().y;
            field[x_pos][y_pos] = 0;
            snake_pos.pop_back();
            snake_length--;
        }
    }

    int Snake::checkIntersections(int dx, int
dy, int **&field){
        int spx, spy;
        // пересечения с границами поля
        spx = snake_pos[0].x + dx;
        spy = snake_pos[0].y + dy;

        if(spx >= fldsize || spx < 0){
            return 1;
        }else if(spy >= fldsize || spy < 0){
            return 1;
        }

        if(field[spx][spy] != 0){
            // пересечение с телом змеи
            if(field[spx][spy] == 1){
                return 1;
            // пересечение с яблоком
            }else if(field[spx][spy] == 2){
                field[spx][spy] = 0;
                return 2;
            // пересечение с молнией
            }else if(field[spx][spy] == 3){
                field[spx][spy] = 0;
                return 3;
            // пересечение с грибом
            }else{
                field[spx][spy] = 0;
                return 4;
            }
        }

        return 0;
    }

    int  Snake::moveSnake(int  cell_size,  int
**&field){
        int   x_pos,   y_pos,   x_pos_window,
y_pos_window;
        int dx, dy, tmp_x, tmp_y;
        int current_direction, intersected;
        int return_val = 0;

        direction = (turn_direction != 0) ?
turn_direction : direction;
        turn_direction = 0;
        switch (direction) {
        case 1: //up
            dx = 0;
            dy = -1;
            break;
        case 2: //right
            dx = 1;
            dy = 0;
            break;
        case 3: // down
            dx = 0;
            dy = 1;
            break;
        case 4: //left
            dx = -1;
            dy = 0;
            break;
        }

        intersected  =  checkIntersections(dx,
dy, field);

        //проверка на пересечение
        if(intersected != 0){
            // со стеной
            if (intersected == 1){
                return 1;

            // с грибом
            }else if(intersected == 4){
                return 4;

            // с молнией или яблоком
            }else{
                // увеличение змеи
                x_pos = last_pos_tail.x;
                y_pos = last_pos_tail.y;
                inc_snake_length(x_pos,
y_pos, cell_size, colorBody, field);
                return_val = intersected;
            }
        }

        // перемещение хвоста
        tmp_x = snake_pos[snake_length-1].x;
        tmp_y = snake_pos[snake_length-1].y;
        field[tmp_x][tmp_y] = 0;
        last_pos_tail = {tmp_x, tmp_y};

        //перемещение тела
        for(int i = snake_length-1; i > 0; i-
```

```
-){
                x_pos_window    =    snake_body[i-
1].getPosition().x;
                y_pos_window    =    snake_body[i-
1].getPosition().y;

snake_body[i].setPosition(x_pos_window,
y_pos_window);
                snake_pos[i] = snake_pos[i-1];
            }

            // перемещение головы
            x_pos_window                          =
snake_body[0].getPosition().x + dx*cell_size;
            y_pos_window                          =
snake_body[0].getPosition().y + dy*cell_size;

snake_body[0].setPosition(x_pos_window,
y_pos_window);
            snake_pos[0].x += dx;
            snake_pos[0].y += dy;
            field[snake_pos[0].x][snake_pos[0].y]
= 1;

            return return_val;
        }

        void Snake::setTurnSnake(int turn){
            turn_direction = turn;
        }

        int Snake::getDirection(){
            return direction;
        }

        void   Snake::resetSnake(int   cell_size,
int**& field){
            int x_pos, y_pos;
            direction = 2;
            turn_direction = 0;
            snake_body.clear();
            snake_pos.clear();
            snake_length = 0;

            // голова
            x_pos = fldsize/2;
            y_pos = fldsize/2;
            inc_snake_length(x_pos,        y_pos,
cell_size, colorHead, field);

            //ячейка тела
            x_pos = fldsize/2 - 1;
            y_pos = fldsize/2;
            inc_snake_length(x_pos,        y_pos,
cell_size, colorBody, field);
            last_pos_tail = {x_pos-1, y_pos};
}
```

main.cpp

```
        #include <SFML/Graphics.hpp>
        #include <iostream>
        #include "snake.h"
        #include "feed.h"
        #include <ctime>

        const int def_speed = 300; //ms
        const int field_size = 20;
        const int cell_size = 30;
        bool game_started = false;
        int speed = def_speed;
        int speed_times = 5;
        int **field;
        int scope = 0;
        int wnd_width = 600;
        int wnd_height = 600;
        bool game_over = false;
        bool start_menu = true;

        sf::Font font;
        sf::Text text_score;
        sf::Text text_gameover;
        sf::Text text_start;

        void draw_scope(sf::RenderWindow &window,
int sc){
            text_score.setString("Score:    "    +
std::to_string(sc));
            text_score.setPosition(wnd_width    -
text_score.getLocalBounds().width - 20, 10);

            window.draw(text_score);
        }

        int main()
        {
            std::srand(time(0));

            sf::RenderWindow
window(sf::VideoMode(wnd_width,      wnd_height),
"SnakeGame",          sf::Style::Close          |
sf::Style::Titlebar);
```

```
            window.setVerticalSyncEnabled(true);
            window.setFramerateLimit(60);

            font.loadFromFile("font/Ampero-
Regular.ttf");

            text_score.setFont(font);
            text_score.setCharacterSize(18);

text_score.setFillColor(sf::Color::White);

            text_gameover.setFont(font);
            text_gameover.setCharacterSize(32);

text_gameover.setFillColor(sf::Color::White);
            text_gameover.setString("Game Over (r
- restart)");
            text_gameover.setPosition((wnd_width
-   text_gameover.getLocalBounds().width)   /   2,
(wnd_height                                    -
text_gameover.getLocalBounds().height) / 2);

            text_start.setFont(font);
            text_start.setCharacterSize(32);

text_start.setFillColor(sf::Color::White);
            text_start.setString("Press   <Space>
to start");
            text_start.setPosition((wnd_width   -
text_start.getLocalBounds().width)      /      2,
(wnd_height - text_start.getLocalBounds().height)
/ 2);

            field = new int* [field_size];
            for(int i = 0; i < field_size; i++){
                field[i] = new int [field_size];
                for(int  j  =  0;  j  <  field_size;
j++){
                    field[i][j] = 0;
                }
            }

            Snake*  snake  =  new  Snake(&window,
cell_size, field_size, field);
            Accelerator*   accelerator   =   new
Accelerator(&window, field_size);
            Grib*   grib   =   new   Grib(&window,
field_size);

            Feed*   feed   =   new   Feed(&window,
field_size);

            Feed** mas = new Feed* [3];

            mas[0] = feed;
            mas[1] = accelerator;
            mas[2] = grib;

            std::vector<int>
snake_direction_queue;
            int             last_snake_direction,
move_snake_result;
            int dmg;
            int r = 0;

            feed->set_feed(field,      cell_size,
field_size, 2);

            while (window.isOpen())
            {
                sf::Event event;
                while (window.pollEvent(event))
                {
                    if       (event.type       ==
sf::Event::Closed)
                        window.close();

                    if       (event.type       ==
sf::Event::KeyPressed){
                        last_snake_direction    =
!snake_direction_queue.empty()                  ?
snake_direction_queue[0] : snake->getDirection();
                        switch (event.key.code) {
                            case
sf::Keyboard::Up:

if(last_snake_direction       !=       3       &&
last_snake_direction != 1){
                                    if
(snake_direction_queue.size() < 2){

snake_direction_queue.insert(snake_direction_queu
e.begin(), 1);
                                    }
                                }
                            break;
                            case
sf::Keyboard::Right:

if(last_snake_direction       !=       4       &&
last_snake_direction != 2){
                                    if
(snake_direction_queue.size() < 2){
```

```
snake_direction_queue.insert(snake_direction_queue.begin(), 2);
                                    }
                                }
                            break;
                            case sf::Keyboard::Down:
                                if(last_snake_direction != 1 && last_snake_direction != 3){
                                    if (snake_direction_queue.size() < 2){
                                        snake_direction_queue.insert(snake_direction_queue.begin(), 3);
                                    }
                                }
                            break;
                            case sf::Keyboard::Left:
                                if(last_snake_direction != 2 && last_snake_direction != 4){
                                    if (snake_direction_queue.size() < 2){
                                        snake_direction_queue.insert(snake_direction_queue.begin(), 4);
                                    }
                                }
                            break;
                            case sf::Keyboard::R:
                                if(game_over){
                                    scope = 0;
                                    speed = def_speed;

                                    for(int i = 0; i < field_size; i++){
                                        for(int j = 0; j < field_size; j++){
                                            field[i][j] = 0;
                                        }
                                    }

                                    snake->resetSnake(cell_size, field);
                                    feed->set_feed(field, cell_size, field_size, 2);

                                    r = 0;
                                    game_started = true;
                                    game_over = false;
                                }
                            break;
                            case sf::Keyboard::Space:
                                if(!game_started){
                                    game_started = true;
                                }
                            break;
                        }
                    }
                }

                if(!snake_direction_queue.empty()){
                    snake->setTurnSnake(snake_direction_queue.back());
                    snake_direction_queue.pop_back();
                }

                if(start_menu){
                    window.clear();
                    window.draw(text_start);
                    start_menu = false;
                }


                if (game_started){
                    window.clear();
                    draw_scope(window, scope);

                    move_snake_result = snake->moveSnake(cell_size, field);
                    if(move_snake_result != 0){
                        // столкновение
                        if(move_snake_result == 1){
                            game_started = false;
                            game_over = true;

                            window.draw(text_gameover);
                        // еда
                        }else{
                            // ускорение
                            if(move_snake_result == 3){
```

```
                                scope += mas[1]->get_scope();
                                Accelerator* rrr = (Accelerator*)mas[1];
                                speed = rrr->getBoost();
                                speed_times = 3;
                            // гриб
                            }else if (move_snake_result == 4){
                                scope = std::max(scope+grib->get_scope(), 0);
                                dmg = grib->getDamage();
                                snake->dec_snake_length(dmg, cell_size, field);
                            // яблоко
                            }else{
                                scope += feed->get_scope();
                                speed_times--;
                                if(speed_times == 0){
                                    speed = def_speed;
                                }
                            }
                            r = std::rand() % 3;
                            Accelerator* rrr = (Accelerator*)mas[1];
                            Grib* ttt = (Grib*)mas[2];
                            switch (r)
                            {
                            case 0:
                                feed->set_feed(field, cell_size, field_size, 2);
                                break;
                            case 1:
                                mas[1]->set_feed(field, cell_size, field_size, 3);
                                rrr->setBoost();
                                break;
                            case 2:
                                mas[2]->set_feed(field, cell_size, field_size, 4);
                                ttt->setDamage();
                                break;
                            }
                        }
                    }
                    snake->drawSnake();
                    switch (r)
                    {
                    case 0:
                        feed->drawFeed();
                        break;
                    case 1:
                        mas[1]->drawFeed();
                        break;
                    case 2:
                        mas[2]->drawFeed();
                        break;
                    }
                }
                window.display();
            sf::sleep(sf::milliseconds(speed));
        }
        return 0;
}
```

5. Вывод

В ходе выполнения лабораторной работы была написана игра «змейка». Для реализации необходимого функционала были разработаны три класса: один родительский и два дочерних. Каждый класс имеет минимум два собственных член данных, две собственных, две унаследованных и две перекрытых член функции. В структуре классов применяются принципы инкапсуляции, наследования и полиморфизма.