Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Рыбинский государственный авиационный технический университет

имени П. А. Соловьева»

Факультет радиоэлектроники и информатики

Кафедра вычислительных систем

**КУРСОВАЯ РАБОТА**

Разработка игры «Морской бой»

на языке Java

по дисциплине «Объектно-ориентированное программирование»

**Пояснительная записка**

Cтудент группы ИВБ-17 Павлов И. М.

Руководитель канд. техн. наук, профессор Паламарь И. Н.

Нормоконтролер канд. техн. наук, профессор Паламарь И. Н.

Рыбинск 2019

Содержание

[Задание 3](#_Toc25408718)

[Введение 3](#_Toc25408719)

[1 Анализ задания 3](#_Toc25408720)

[2 Проектирование программы 7](#_Toc25408722)

[2.1 Анализ реализуемых требований к курсовой работе 7](#_Toc25408723)

[2.2 Описание классов 7](#_Toc25408724)

[2.3 Объектная декомпозиция 13](#_Toc25408724)

[3 Разработка алгоритмов 14](#_Toc25408725)

[4 Тестирование программы 16](#_Toc25408726)

[5 Руководство пользователя 17](#_Toc25408727)

[Заключение 17](#_Toc25408728)

[Список используемых источников 18](#_Toc25408729)

[Приложение А UML-диаграммы 1](#_Toc25408730)9

[Приложение Б Текст основной программы 2](#_Toc25408730)4

# Задание

Написать программу для игры «Морской бой».

# Введение

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования. Идеологически ООП — подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов. Задачей курсовой работы является написание программы для игры «Морской бой» на основе знаний, полученных за курс объектно-ориентированного программирования.

# 1 Анализ задания

В основе программы лежат два поля на основе двумерных массивов – поле игрока и поле искусственного интеллекта. Эти поля формируются на основе заданного по умолчанию размера – 10х10 клеток. Каждая клетка поля имеет 1 из 6 состояний: клетка воды, клетка вокруг корабля, клетка здорового корабля, клетка раненого корабля, клетка убитого корабля, клетка промаха. Клетки формируются на этапе подготовки программы, когда расставляется заданное по умолчанию количество кораблей - 10: 1 четырехпалубный корабль, 2 трехпалубных корабля, 3 двухпалубных корабля и 4 однопалубных корабля, которые занимают 4, 3, 2 и 1 клетку соответственно в случайно заданном направлении. Впоследствии клетки также могут менять состояние в зависимости от производимых действий игроком или искусственным интеллектом в ходе выполнения программы. Действия искусственного интеллекта в процессе игры выполняются в 3 фазы: обстрел любой доступной клетки со всего поля, обстрел любой клетки из доступных направлений вокруг раненого корабля, обстрел любой из двух или одного оставшихся направлений после повторного ранения корабля. Разберем этот процесс поэтапно. Первый этап: для обстрела доступны все непомеченные клетки вражеского поля (рисунок 1.1).

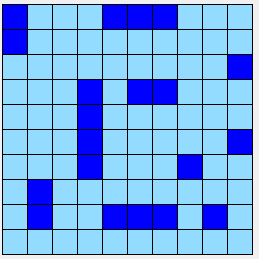


Рисунок 1.1 – первый этап

Как только фиксируется попадание по кораблю, происходит переход ко второму этапу. Если это не однопалубный корабль, то область обстрела сокращается до клеток в доступных направлениях вокруг раненой клетки, иначе корабль помечается как убитый, клетки вокруг него помечаются промахом и происходит возвращение к первому этапу (рисунок 1.2).

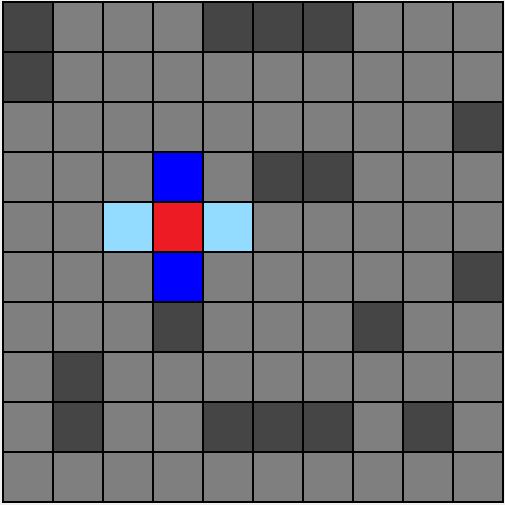


Рисунок 1.2 – второй этап

Далее, как только фиксируется еще одно попадание по этому кораблю, происходит переход к третьему этапу, когда остаются доступными только два направления и по достижению убийства корабля, клетки вокруг него помечаются промахом и происходит переход к первому этапу (рисунок 1.3).

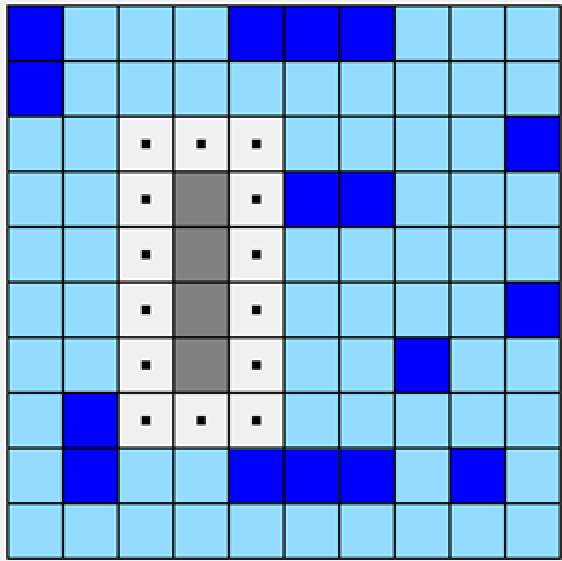
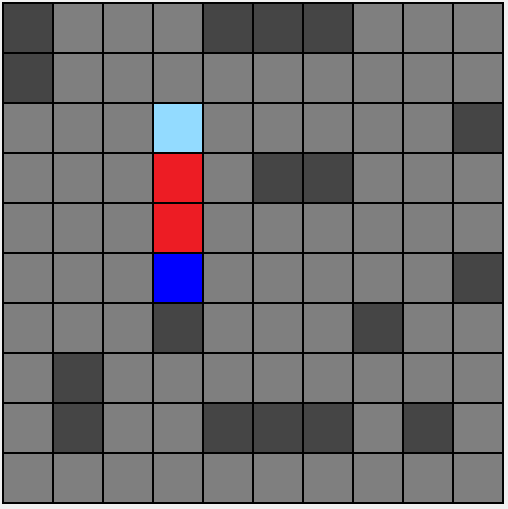


Рисунок 1.3 – третий этап и переход к первому этапу

Как только количество живых кораблей на поле станет равным нулю, то в зависимости от того, чье это поле, будет выведено сообщение с результатом игры (рисунок 1.4).

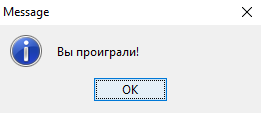


Рисунок 1.4 – сообщение о результате игры

# 

# 

# 2 Проектирование программы

## 2.1 Анализ реализуемых требований к курсовой работе

Согласно требованиям к курсовой работе мы должны написать программу, используя объектно-ориентированный подход – такой подход, при котором основными концепциями являются понятия объектов и классов и связи между ними.

2.2 Описание классов

Главным классом, содержащим метод main, является класс GameLauncher, откуда создаются объекты GameLogic и GameView.

Класс GameLogic имеет поля:

* public Field playerField – поле игрока;
* public Field botField – поле компьютера;
* public AI ai – объект, отвечающий за поведение компьютера во время игры;
* private ArrayList<PanelField> panelFields – список полей, отображающихся на форме;
* private boolean currentShot – переменная, определяющая чей ход;
* private boolean enableShot – переменная, разрешающая сделать выстрел.

Методы класса GameLogic:

* public GameLogic() – конструктор класса GameLogic, инициализирующий поля playerField, botField, ai и вызывающий метод newGame();
* public void newGame() – метод, инициализирующий поля currentShot, enableShot и вызывающий методы setShips() у объектов playerField и botField;
* public void doShot(int x, int y) – метод, производящий выстрел по входящим в качестве параметров координатам;
* public void add(PanelField panelField) – метод, добавляющий входящее в качестве параметра поле panelField в список panelFields;
* public void updatePanelFields() – метод, обновляющий значение списка panelFields и перерисовывающий эти поля.

Класс GameView имеет поля:

* private GameLogic logic – объект класса GameLogic;
* private GameController controller – объект класса GameController;
* private JMenuItem jMenuNewGame – пункт NewGame в меню;
* private JMenuItem jMenuItemExit – пункт Exit в меню;
* public PanelFieldPlayer panelFieldPlayer – поле игрока на форме;
* public PanelFieldBot panelFieldBot – поля компьютера на форме.

Методы класса GameView:

* public GameView(GameLogic logic) – конструктор класса GameView, инициализирующий поля logic и controller, добавляющий поля panelFieldPlayer и panelFieldBot в список panelFields в классе GameLogic и вызывающий метод attachController();
* private void buildUI() – метод, рисующий интерфейс пользователя;
* private void AttachController() – метод, добавляющий слушатель мыши.

Класс GameController имеет поля:

* public GameLogic logic – объект класса GameLogic;
* public GameView view – объект класса GameView;

Методы класса GameController:

* public GameController(GameLogic logic, GameView view) – конструктор класса GameController, иницилизирующий поля logic и view;
* public void actionPerformed(ActionEvent e) – слушатель мыши по командам меню;
* public void mousePressed(MouseEvent e) – слушатель мыши по полям игроков;

Абстрактный класс PanelField имеет поля:

* private Field field – поле, отображаемое на форме;

Методы абстрактного класса PanelField:

* public PanelField(Field field) – конструктор абстрактного класса PanelField, инициализирующий поле field;
* protected abstract Color getColor(int state) – метод, возвращающий цвет;
* protected void paintComponent(Graphics g) – метод, рисующий поля на форме;
* public void update() – метод, перерисовывающий поля на форме;

Классы PanelFieldPlayer и PanelFieldBot наследуются от абстрактного класса PanelField и возвращают цвет клеток в зависимости от принадлежности поля.

Класс Field имеет поля:

* public final static int SHOT\_MISSED – состояние промаха;
* public final static int SHOT\_INJURED – состояние ранения;
* public final static int SHOT\_KILLED – состояние убийства;
* private Cell[][] cells – двумерный массив клеток типа Cell;
* private ArrayList<Ship> ships – список кораблей на данном поле;
* private int width = 10 – ширина поля;
* private int height = 10 – высота поля;
* private int liveShips – количество живых кораблей на данном поле;

Методы класса Field:

* public void setShips() – метод, устанавливающий корабли на данном поле;
* public int doShot(int x, int y) – метод, совершающий выстрел по входящим в качестве параметров координатам и возвращающий состояние этого выстрела;
* public boolean isBound(int x, int y) – метод, проверяющий, попадают ли заданные координаты в рамки поля.

Класс Cell имеет поля:

* public final static int CELL\_WATER = 1 – клетка воды;
* public final static int CELL\_AROUND = 2 – клетка вокруг корабля;
* public final static int CELL\_WELL = 3 – клетка здорового корабля;
* public final static int CELL\_INJURED = 4 – клетка раненого корабля;
* public final static int CELL\_KILLED = 5 – клетка убитого корабля;
* public final static int CELL\_MISSED = 6 – клетка промаха;
* public int x – координата x клетки;
* public int y – координата y клетки;
* private int state – состояние клетки;
* private boolean mark – флаг выстрела по клетке;
* private Ship ship – корабль, принадлежащий клетке;

Методы класса Cell:

* public Cell(int x, int y) – конструктор класса Cell, инициализирующий координаты, состояние и флаг выстрела;
* public int doShot() – метод, совершающий выстрел по данной клетке и возвращающий состояние выстрела и клетки.

Класс Ship имеет поля:

* public final static int SHIP\_WELL = 1 – состояние здорового корабля;
* public final static int SHIP\_INJURED = 2 – состояние раненого корабля;
* public final static int SHIP\_KILLED = 3 – состояние убитого корабля;
* public int x – координата x клетки;
* public int y – координата x клетки;
* public int dx – направление корабля по x;
* public int dy – направление корабля по y;
* private int health – здоровье корабля;
* private int size – размер корабля;
* private int state – состояние корабля;
* private Field field – поле, к которому принадлежит данный корабль;
* private ArrayList<Cell> cells – список клеток корабля;
* private ArrayList<Cell> arounds – список клеток вокруг корабля;

Методы класса Ship:

* public Ship (Field field, int size) – конструктор класса Ship, инициализирующий поля размера, здоровья, поля, состояния, списков и вызывающий методы getPlace() и setShip();
* private void getPlace() – метод получающий случайные координаты;
* private boolean checkPlace() – метод, определяющий, подходит ли место для установки корабля по заданным координатам;
* private void setShip() – метод, устанавливающий корабль по заданным координатам;
* private boolean byPass(PlaceShip placeShip) – метод обхода вариантов установки корабля, который возвращает разрешение на установку;
* public int doShot() – метод, совершающий выстрел по данному кораблю и возвращающий состояние этого выстрела;

Абстрактный класс PlaceShip имеет поля:

* public Field field – поле, на которое устанавливается данный корабль;

Методы абстрактного класса PlaceShip:

* public PlaceShip(Ship ship) – конструктор абстрактного класса PlaceShip, инициализирующий поле field;
* public abstract boolean setShip(int x, int y) – метод установки корабля;
* public boolean setAround(int x, int y) – метод установки клеток вокруг корабля.

Класс PlaceShipCheck наследуется от абстрактного класса PlaceShip и имеет методы:

* public PlaceShipCheck(Ship ship) – конструктор класса PlaceShipCheck, вызывающий конструктор родительского класса;
* public boolean setShip(int x, int y) – метод, возвращающий разрешение на установку корабля по входящим в качестве параметров координатам;

Класс PlaceShipSet наследуется от абстрактного класса PlaceShip и имеет поля:

* private Ship ship – устанавливаемый корабль;

Методы класса PlaceShipSet:

* public PlaceShipSet(Ship ship) – конструктор класса PlaceShipSet, вызывающий конструктор родительского класса и инициализирующий поле ship;
* public boolean setShip(int x, int y) – метод, устанавливающий корабль по входящим в качестве параметров координатам;
* public boolean setAround(int x, int y) – метод, устанавливающий клетки вокруг корабля в соответствующее состояние.

Класс AI имеет поля:

* public Field field – поля, с которым оперирует компьютер;
* public AIBase action – объект, определяющий поведение компьютера в зависимости от ситуации на поле;
* public Random random – объект выбора случайных чисел;

Методы класса AI:

* public AI(Field field) – конструктор класса AI, инициализирующий поля field, action и random;
* public int doShot() – метод, совершающий выстрел компьютером;

Абстрактный класс AIBase имеет поля:

* protected AI ai – объект поведения компьютера;
* protected int x – координата x;
* protected int y – координата y;
* protected int dx – направление по x;
* protected int dy – направление по y;

Методы абстрактного класса AIBase:

* public AIBase(AI ai) – конструктор абстрактного класса AIBase, инициализирующий поле ai;
* protected abstract int doShot() – метод, совершающий выстрел;
* protected void setPosition(int x, int y) – метод, инициализирующий координаты x и y;
* protected void setDirection(int dx, int dy) – метод, инициализирующий направления dx и dy;

Класс AIRandom наследуется от абстрактного класса AIBase и имеет методы:

* public AIRandom(AI ai) – конструктор класса AIRandom, вызывающий конструктор родительского класса;
* public int doShot() – метод, совершающий выстрел в случайную клетку из всех доступных на поле и возвращающий состояние выстрела.

Класс AIAround наследуется от абстрактного класса AIBase и имеет методы:

* public AIAround(AI ai) - конструктор класса AIAround, вызывающий конструктор родительского класса;
* public int doShot() – метод, совершающий выстрел в случайную клетку из доступных вокруг раненой клетки корабля и возвращающий состояние этого выстрела.

Класс AIKilling наследуется от абстрактного класса AIBase и имеет методы:

* public AIKilling(AI ai) - конструктор класса AIKilling, вызывающий конструктор родительского класса;
* protected int doShot() – метод, совершающий выстрел в случайную клетку из доступных в одном или двух направлениях вокруг двух и более раненых клеток корабля и возвращающий состояние этого выстрела;
* public void draw(ArrayList<Cell> cells, int i, int j) – метод, формирующий список доступных клеток для выстрела.

## 2.3 Объектная декомпозиция

На рисунке 2.1 представлена объектная декомпозиция программы.

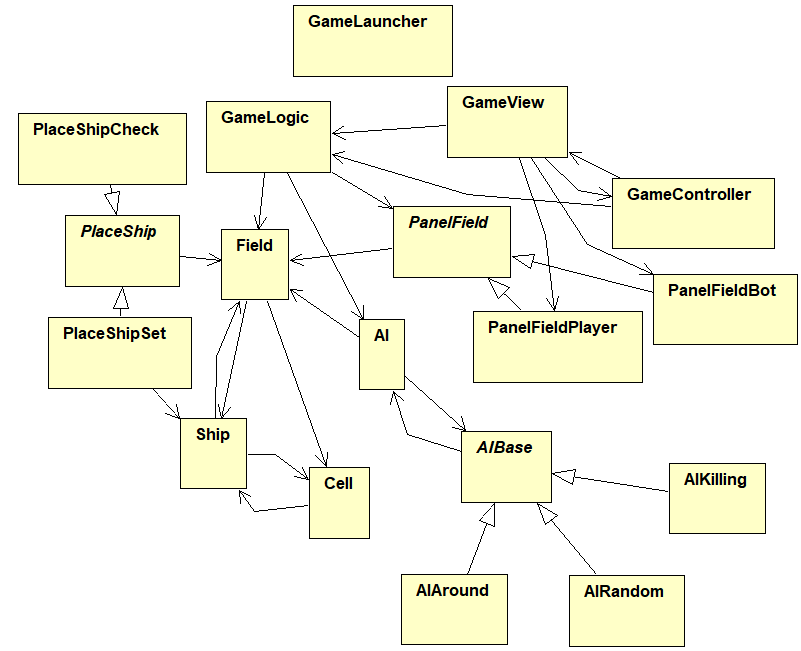


Рисунок 2.1 – объектная декомпозиция программы

# 3 Разработка алгоритмов

Алгоритм подготовки программы представлен на рисунке 3.1. После подготовки программа находится в состоянии ожидания действий пользователя. Алгоритм поведения программы после нажатия кнопки мыши по непомеченным клеткам противника представлен на рисунке 3.2.

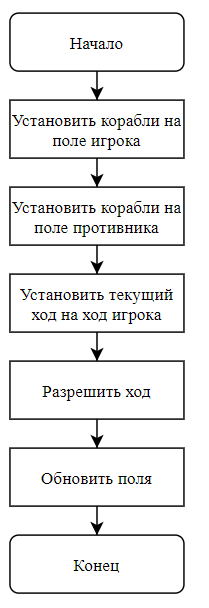


Рисунок 3.1 – алгоритм подготовки программы

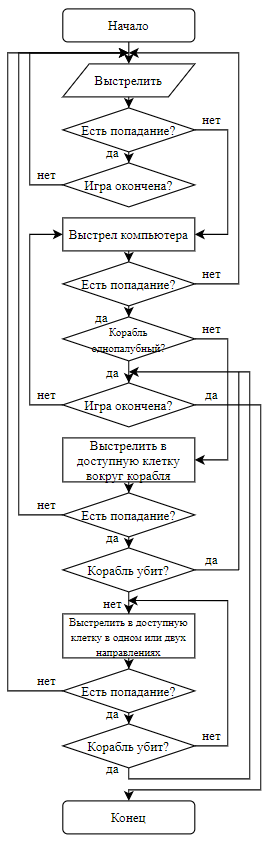


Рисунок 3.2 – алгоритм поведения программы после выстрела игрока

4 Тестирование программы

В ходе работы программы ошибок не возникает.

На рисунке 4.1 продемонстрирован скриншот окончания выполнения программы.

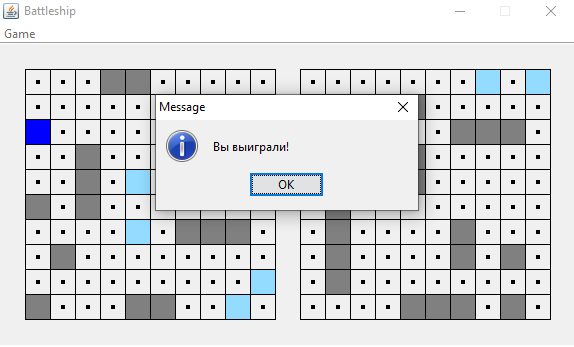


Рисунок 4.1 – Скриншот окончания выполнения программы

# 5 Руководство пользователя

Игра “Морской бой” обладает доступным, интуитивно понятным интерфейсом, который позволит пользоваться данным приложением любым пользователям. Игровая площадка имеет размер 10 на 10 клеток, на которой перед игрой будут случайным образом расставлены корабли на поле игрока и противника. После расстановки начинается бой. Он представляет собой поочередные выстрелы игрока и компьютера. При попадании в корабль противника участник боя получает возможность проведения внеочередного выстрела. Игра заканчивается при уничтожении одним из участников всех кораблей противника. Во время выполнения программы пользователь в любой момент может начать новую игру или выйти, выбрав соответствующее действие в пункте меню “Game”.

# 

# Заключение

Итак, на основе знаний, полученных за курс объектно-ориентированного программирования, была написана программа для игры «Морской бой». В ходе выполнения курсовой работы была поэтапно разобрана поставленная задача на примере, что помогло лучше определить принцип работы программы. После рассмотрения условий задачи на примере было начато проектирование программы. Проанализировав требования выполнения курсовой работы, была составлена объектная декомпозиция программы. Далее были разработаны алгоритмы выполнения программы, что является важнейшим пунктом для написания программы. В результате тестирования программы не было выявлено никаких ошибок.

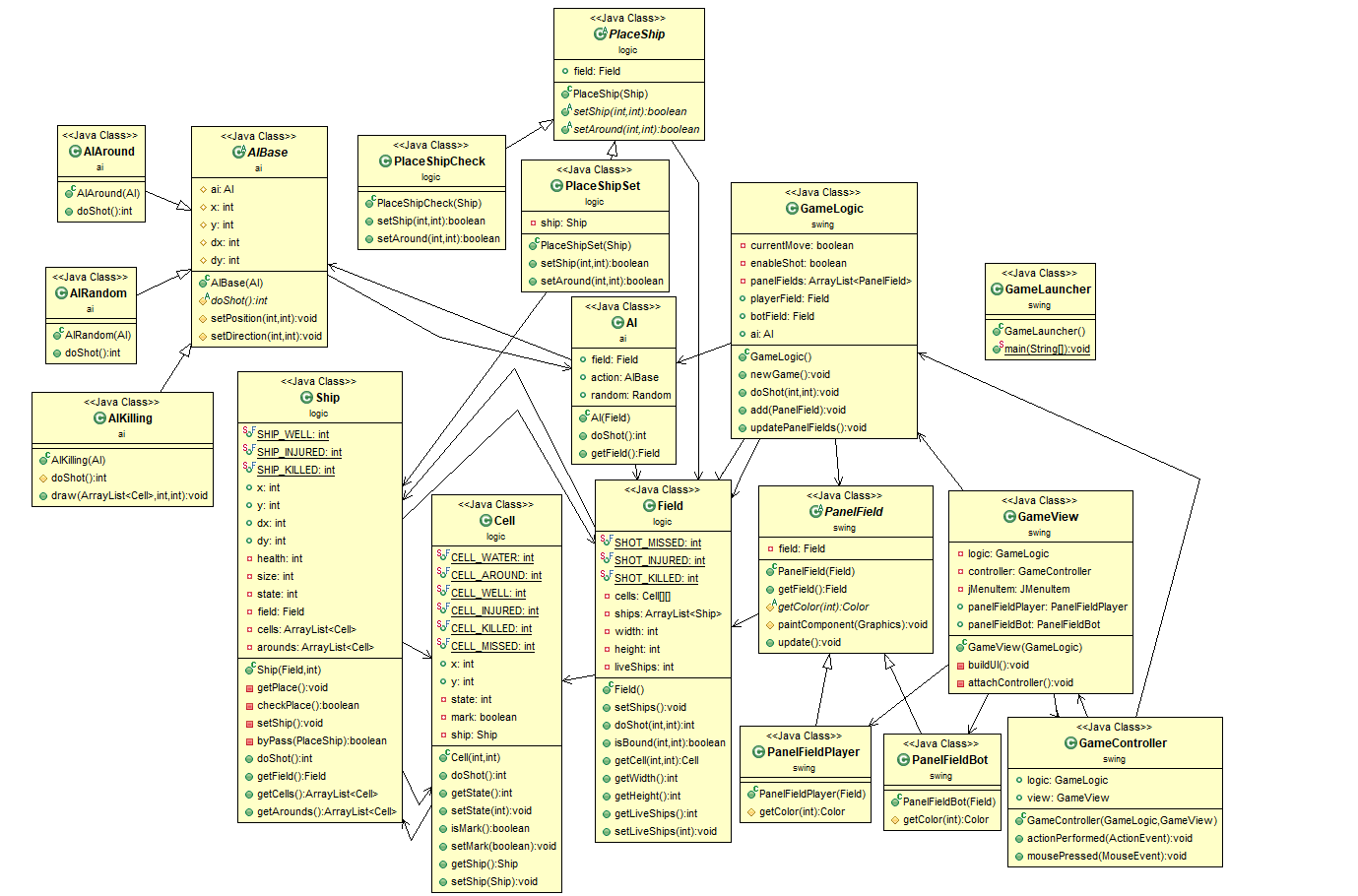
# Список используемых источников

1. Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К. Объектно-ориентированное программирование. Москва, издательство МГТУ имени Н.Э. Баумана, 2003.
2. Академик [Электронный ресурс]. URL:https://dic.academic.ru/ (дата обращения: 25.05.2018).
3. Герберт Шилдт. Java 8. Полное руководство.
4. JavaRush [Электронный ресурс]. URL: https://javarush.ru/

# 

# Приложение А

UML-диаграммы

Рисунок 1 – UML-диаграмма классов

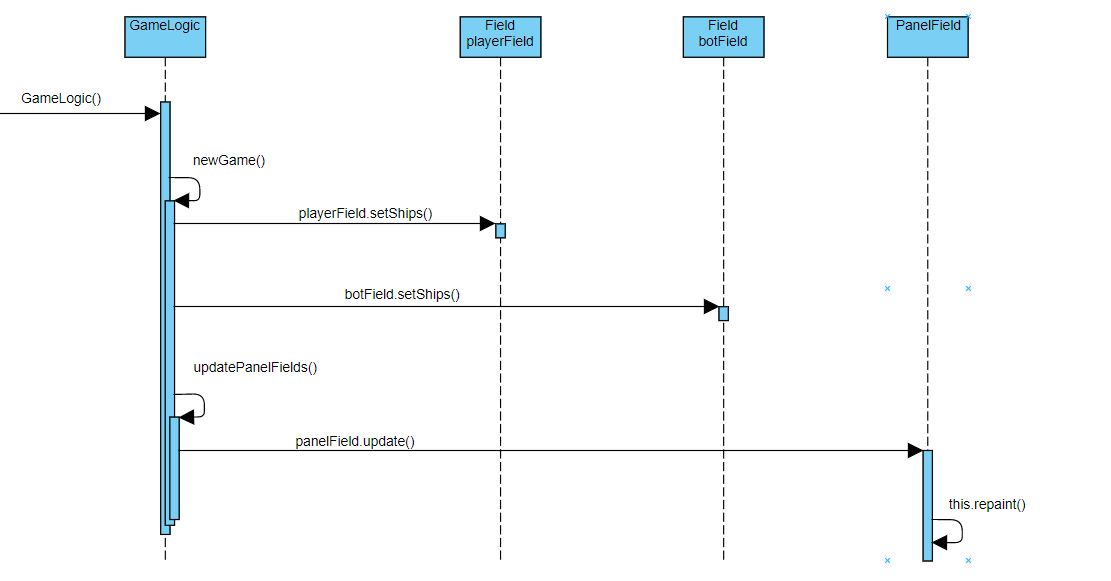


Рисунок 2 – UML-диаграмма последовательностей (подготовка)

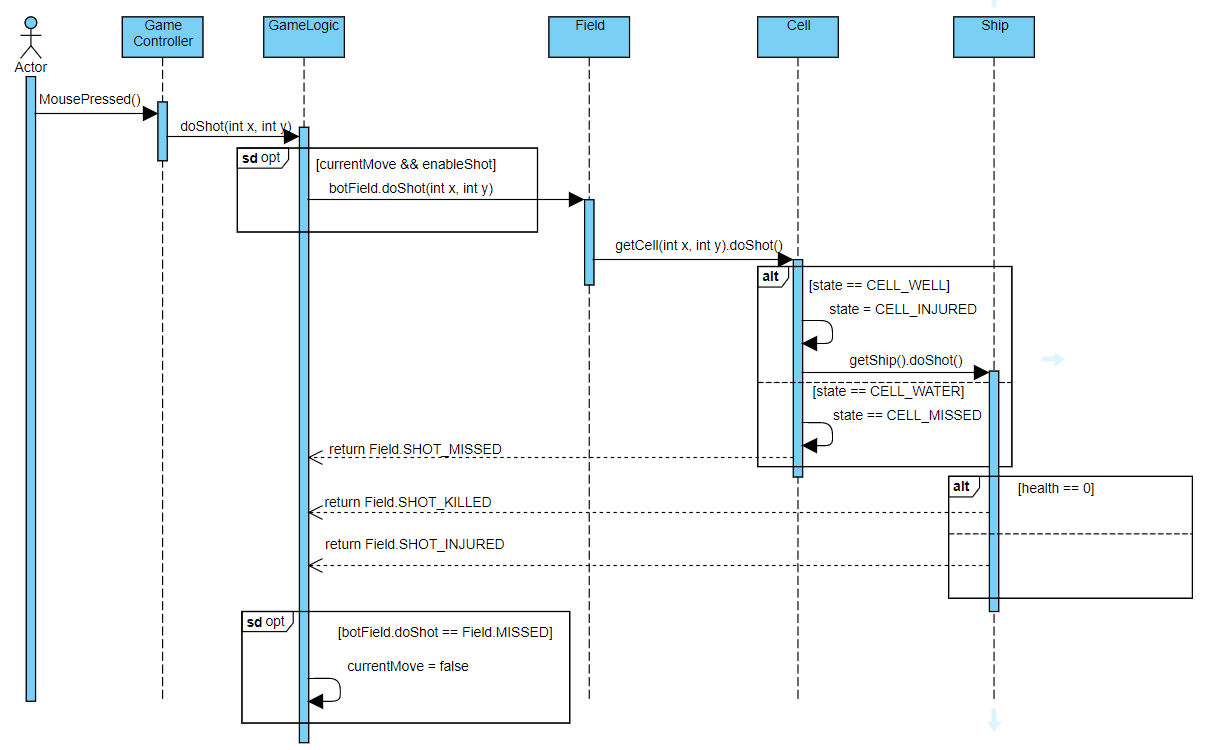


Рисунок 3 – UML-диаграмма последовательностей (ход игрока)

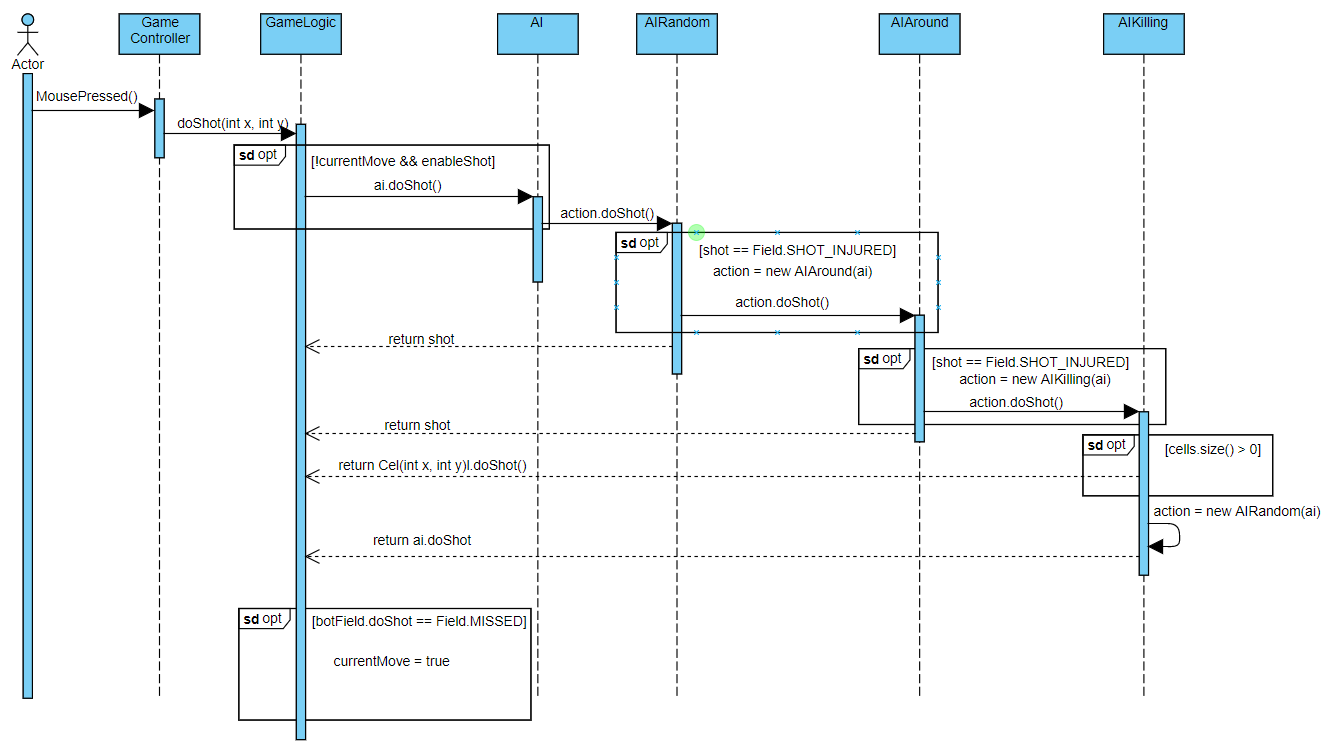


Рисунок 4 – UML-диаграмма последовательностей (ход компьютера)

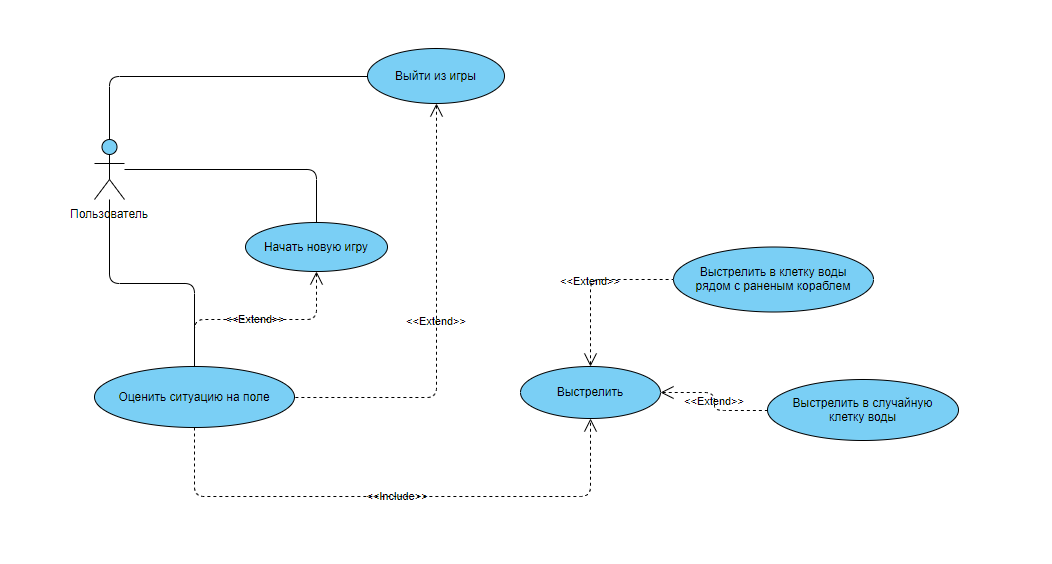


Рисунок 5 – UML-диаграмма использования

Приложение **Б**

Текст основной программы

package swing;  
  
import javax.swing.\*;  
  
public class GameLauncher {  
 public static void main(String[] args) throws ClassNotFoundException, UnsupportedLookAndFeelException, InstantiationException, IllegalAccessException {  
 UIManager.*setLookAndFeel*(UIManager.*getSystemLookAndFeelClassName*());  
 GameLogic logic = new GameLogic();  
 GameView view = new GameView(logic);  
 view.setVisible(true);  
 }  
}

package swing;  
  
import ai.AI;  
import logic.Field;  
import javax.swing.\*;  
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class GameLogic {  
 private boolean currentMove; // false - ходит бот, true - ходит игрок  
 private boolean enableShot;  
 private ArrayList<PanelField> panelFields = new ArrayList<>();  
 public Field playerField;  
 public Field botField;  
 public AI ai;  
  
 public GameLogic(){  
 playerField = new Field();  
 botField = new Field();  
 ai = new AI(playerField);  
 newGame();  
 }  
  
 public void newGame(){  
 playerField.setShips();  
 botField.setShips();  
 currentMove = true;  
 enableShot = true;  
 updatePanelFields();  
 }  
  
 public void doShot(int x, int y){  
 if (currentMove && enableShot){  
 if (botField.getCell(x, y).isMark()){  
 return;  
 }  
 if (botField.doShot(x, y) == Field.*SHOT\_MISSED*){  
 currentMove = false;  
 }  
 }  
 if (!currentMove && enableShot){  
 while (ai.doShot() != Field.*SHOT\_MISSED*);  
 currentMove = true;  
 }  
 updatePanelFields();  
 if (enableShot && botField.getLiveShips() == 0){  
 enableShot = false;  
 JOptionPane.*showMessageDialog*(null, "Вы выиграли!");  
 }  
 if (enableShot && playerField.getLiveShips() == 0){  
 enableShot = false;  
 JOptionPane.*showMessageDialog*(null, "Вы проиграли!");  
 }  
 }  
  
 public void add(PanelField panelField){  
 panelFields.add(panelField);  
 panelField.update();  
 }  
  
 public void updatePanelFields(){  
 Iterator<PanelField> i = panelFields.iterator();  
 while (i.hasNext()){  
 PanelField panelField = i.next();  
 panelField.update();  
 }  
 }  
}

package swing;  
  
import javax.swing.\*;  
import java.awt.\*;  
import java.awt.event.MouseAdapter;  
import java.awt.event.MouseEvent;  
  
public class GameView extends JFrame {  
 private GameLogic logic;  
 private GameController controller;  
 private JMenuItem jMenuNewGame;  
 private JMenuItem jMenuExit;  
 public PanelFieldPlayer panelFieldPlayer;  
 public PanelFieldBot panelFieldBot;  
  
 public GameView(GameLogic logic){  
 this.logic = logic;  
 buildUI();  
 this.logic.add(panelFieldPlayer);  
 this.logic.add(panelFieldBot);  
 this.controller = new GameController(logic, this);  
 attachController();  
 }  
  
 private void buildUI(){  
 this.setTitle("Battleship");  
 this.setResizable(false);  
 this.setBounds(460, 290, 580, 350);  
  
 Dimension dimension = Toolkit.*getDefaultToolkit*().getScreenSize();  
 this.setLocation((dimension.width-this.getWidth())/2, (dimension.height - this.getHeight())/2);  
  
 this.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 this.getContentPane().setLayout(null);  
  
 panelFieldPlayer = new PanelFieldPlayer(logic.playerField);  
 panelFieldPlayer.setBounds(25, 45, 251, 251);  
 this.getContentPane().add(panelFieldPlayer);  
  
 panelFieldBot = new PanelFieldBot(logic.botField);  
 panelFieldBot.setBounds(300, 45, 251, 251);  
 this.getContentPane().add(panelFieldBot);  
  
 JMenuBar jMenuBar = new JMenuBar();  
 jMenuBar.setBounds(0, 0, 577, 20);  
 this.getContentPane().add(jMenuBar);  
  
 JMenu menuGame = new JMenu("Game");  
 jMenuBar.add(menuGame);  
  
 jMenuNewGame = new JMenuItem("New game");  
 menuGame.add(jMenuNewGame);  
  
 jMenuExit = new JMenuItem("Exit");  
 menuGame.add(jMenuExit);  
 }  
  
 private void attachController(){  
 jMenuNewGame.addActionListener(controller);  
 jMenuExit.addActionListener(controller);  
 panelFieldBot.addMouseListener(new MouseAdapter() {  
 @Override  
 public void mousePressed(MouseEvent e) {  
 controller.mousePressed(e);  
 }  
 });  
 }  
}

package swing;  
  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.awt.event.MouseEvent;  
  
public class GameController implements ActionListener {  
 public GameLogic logic;  
 public GameView view;  
  
 public GameController(GameLogic logic, GameView view){  
 this.logic = logic;  
 this.view = view;  
 }  
  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 String s = e.getActionCommand();  
 if (s.equals("New game")){  
 logic.newGame();  
 }  
 if (s.equals("Exit")){  
 System.*exit*(0);  
 }  
 }  
  
 public void mousePressed(MouseEvent e){  
 PanelField panelField = view.panelFieldBot;  
 int x = e.getX() / (panelField.getWidth() / panelField.getField().getWidth());  
 int y = e.getY() / (panelField.getHeight() / panelField.getField().getHeight());  
 if (panelField.getField().isBound(x, y)) {  
 logic.doShot(x, y);  
 }  
 }  
}

package swing;  
  
import logic.Cell;  
import logic.Field;  
import javax.swing.\*;  
import java.awt.\*;  
  
public abstract class PanelField extends JPanel {  
 private Field field;  
  
 public PanelField(Field field){  
 this.field = field;  
 }  
  
 public Field getField(){  
 return field;  
 }  
  
 protected abstract Color getColor(int state);  
  
 @Override  
 protected void paintComponent(Graphics g){  
 super.paintComponent(g);  
 for (int i = 0; i < field.getWidth() + 1; i++) {  
 g.drawLine(i \* (getWidth() / field.getWidth()), 0, i \* (getWidth() / field.getWidth()), (getHeight() / field.getHeight()) \* field.getHeight());  
 }  
 for (int i = 0; i < field.getHeight() + 1; i++) {  
 g.drawLine(0, i \* (getHeight() / field.getHeight()), (getWidth() / field.getWidth()) \* field.getWidth(), i \* (getHeight() / field.getHeight()));  
 }  
 for (int i = 0; i < field.getWidth(); i++) {  
 for (int j = 0; j < field.getHeight(); j++) {  
 int state = field.getCell(j, i).getState();  
 g.setColor(getColor(state));  
 if (state == Cell.*CELL\_MISSED*){  
 g.fillRect(j \* (getHeight() / field.getHeight()) + ((getHeight() / field.getHeight()) / 2) - 1, i \* (getWidth() / field.getWidth()) + ((getWidth() / field.getWidth()) / 2) - 1, 4, 4);  
 }  
 else {  
 g.fillRect(j \* (getHeight() / field.getHeight()) + 1, i \* (getWidth() / field.getWidth()) + 1, (getHeight() / field.getHeight()) - 1, (getWidth() / field.getWidth()) - 1);  
 }  
 }  
 }  
 }  
  
 public void update(){  
 this.repaint();  
 }  
}

package swing;  
  
import logic.\*;  
import java.awt.\*;  
  
public class PanelFieldBot extends PanelField{  
 public PanelFieldBot(Field field) {  
 super(field);  
 }  
  
 @Override  
 protected Color getColor(int state) {  
 switch (state){  
 case Cell.*CELL\_WATER*: return new Color(147, 219, 255);  
 case Cell.*CELL\_AROUND*: return Color.*white*;  
 case Cell.*CELL\_WELL*: return new Color(147, 219, 255);  
 case Cell.*CELL\_INJURED*: return Color.*red*;  
 case Cell.*CELL\_KILLED*: return Color.*gray*;  
 case Cell.*CELL\_MISSED*: return Color.*black*;  
 }  
 return Color.*white*;  
 }  
}

package swing;  
  
import logic.\*;  
import java.awt.\*;  
  
public class PanelFieldPlayer extends PanelField{  
 public PanelFieldPlayer(Field field) {  
 super(field);  
 }  
  
 @Override  
 protected Color getColor(int state) {  
 switch (state){  
 case Cell.*CELL\_WATER*: return new Color(147, 219, 255);  
 case Cell.*CELL\_AROUND*: return Color.*white*;  
 case Cell.*CELL\_WELL*: return Color.*blue*;  
 case Cell.*CELL\_INJURED*: return Color.*red*;  
 case Cell.*CELL\_KILLED*: return Color.*gray*;  
 case Cell.*CELL\_MISSED*: return Color.*black*;  
 }  
 return Color.*white*;  
 }  
}

package logic;  
  
import java.util.ArrayList;  
  
public class Field {  
 public final static int *SHOT\_MISSED* = 1;  
 public final static int *SHOT\_INJURED* = 2;  
 public final static int *SHOT\_KILLED* = 3;  
 private Cell[][] cells;  
 private ArrayList<Ship> ships;  
 private int width = 10;  
 private int height = 10;  
 private int liveShips;  
  
 public void setShips(){  
 setLiveShips(0);  
 cells = new Cell[getWidth()][getHeight()];  
 for (int i = 0; i < getWidth(); i++) {  
 for (int j = 0; j < getHeight(); j++) {  
 cells[j][i] = new Cell(j, i);  
 }  
 }  
 ships = new ArrayList<Ship>();  
 for (int i = 4; i > 0; i--) {  
 for (int j = (5 - i); j > 0; j--) {  
 Ship ship = new Ship(this, i);  
 ships.add(ship);  
 }  
 }  
 for (int i = 0; i < getWidth(); i++) {  
 for (int j = 0; j < getHeight(); j++) {  
 Cell cell = cells[j][i];  
 if (cell.getState() == Cell.*CELL\_AROUND*){  
 cell.setState(Cell.*CELL\_WATER*);  
 }  
 }  
 }  
 }  
  
 public int doShot(int x, int y){  
 int shot = getCell(x, y).doShot();  
 return shot;  
 }  
  
 public boolean isBound(int x, int y){  
 return !((x < 0) || (x > (getWidth() - 1)) || (y < 0) || (y > (getHeight() - 1)));  
 }  
  
 public Cell getCell(int x, int y) {  
 return cells[x][y];  
 }  
  
 public int getWidth() {  
 return width;  
 }  
  
 public int getHeight() {  
 return height;  
 }  
  
 public int getLiveShips() {  
 return liveShips;  
 }  
  
 public void setLiveShips(int liveShips) {  
 this.liveShips = liveShips;  
 }  
}

package logic;  
  
public class Cell {  
 public final static int *CELL\_WATER* = 1;  
 public final static int *CELL\_AROUND* = 2;  
 public final static int *CELL\_WELL* = 3;  
 public final static int *CELL\_INJURED* = 4;  
 public final static int *CELL\_KILLED* = 5;  
 public final static int *CELL\_MISSED* = 6;  
 public int x;  
 public int y;  
 private int state;  
 private boolean mark;  
 private Ship ship;  
  
 public Cell(int x, int y){  
 this.x = x;  
 this.y = y;  
 this.state = *CELL\_WATER*;  
 this.mark = false;  
 }  
  
 public int doShot(){  
 mark = true;  
 if (state == *CELL\_WELL*){  
 state = *CELL\_INJURED*;  
 return getShip().doShot();  
 }  
 else{  
 if (state == *CELL\_WATER*){  
 state = *CELL\_MISSED*;  
 }  
 }  
 return Field.*SHOT\_MISSED*;  
 }  
  
 public int getState() {  
 return state;  
 }  
  
 public void setState(int state) {  
 this.state = state;  
 }  
  
 public boolean isMark() {  
 return mark;  
 }  
  
 public void setMark(boolean mark) {  
 this.mark = mark;  
 }  
  
 public Ship getShip() {  
 return ship;  
 }  
  
 public void setShip(Ship ship) {  
 this.ship = ship;  
 }  
  
}

package logic;  
  
import java.util.ArrayList;  
import java.util.Random;  
  
public class Ship {  
 public final static int *SHIP\_WELL* = 1;  
 public final static int *SHIP\_INJURED* = 2;  
 public final static int *SHIP\_KILLED* = 3;  
 public int x;  
 public int y;  
 public int dx;  
 public int dy;  
 private int health;  
 private int size;  
 private int state;  
 private Field field;  
 private ArrayList<Cell> cells;  
 private ArrayList<Cell> arounds;  
  
 public Ship (Field field, int size){  
 this.size = size;  
 this.health = size;  
 this.field = field;  
 this.state = Ship.*SHIP\_WELL*;  
  
 do {  
 this.getPlace();  
 } while (!checkPlace());  
  
 this.cells = new ArrayList<>();  
 this.arounds = new ArrayList<>();  
 this.setShip();  
  
 getField().setLiveShips(getField().getLiveShips() + 1);  
 }  
  
 private void getPlace(){  
 Random random = new Random();  
 this.x = random.nextInt(getField().getWidth());  
 this.y = random.nextInt(getField().getHeight());  
 this.dx = 0;  
 this.dy = 0;  
 if (random.nextInt(2) == 1){  
 this.dx = 1;  
 }  
 else{  
 this.dy = 1;  
 }  
 }  
  
 private boolean checkPlace(){  
 return byPass(new PlaceShipCheck(this));  
 }  
  
 private void setShip(){  
 byPass(new PlaceShipSet(this));  
 }  
  
 private boolean byPass(PlaceShip placeShip){  
 int i, m, n;  
  
 for (i = 0; i < size; i++) {  
 n = x + i \* dx;  
 m = y + i \* dy;  
 if (!placeShip.setShip(m, n)){  
 return false;  
 }  
 n = x + i \* dx - dy;  
 m = y + i \* dy - dx;  
 if (!placeShip.setAround(m, n)){  
 return false;  
 }  
 n = x + i \* dx + dy;  
 m = y + i \* dy + dx;  
 if (!placeShip.setAround(m, n)){  
 return false;  
 }  
 }  
  
 for (i = -1; i < 2; i++) {  
 n = x + i \* dy - dx;  
 m = y + i \* dx - dy;  
 if (!placeShip.setAround(m, n)){  
 return false;  
 }  
 n = x + i \* dy + (dx \* size);  
 m = y + i \* dx + (dy \* size);  
 if (!placeShip.setAround(m, n)){  
 return false;  
 }  
 }  
 return true;  
 }  
  
 public int doShot(){  
 if (health != 0){  
 health--;  
 if (health == 0){  
 getField().setLiveShips(getField().getLiveShips() - 1);  
 state = Ship.*SHIP\_KILLED*;  
 for (Cell cell : cells){  
 cell.setState(Cell.*CELL\_KILLED*);  
 }  
 for (Cell cell : arounds){  
 cell.setState(Cell.*CELL\_MISSED*);  
 cell.setMark(true);  
 }  
 return Field.*SHOT\_KILLED*;  
 }  
 else {  
 state = *SHIP\_INJURED*;  
 }  
 }  
 return Field.*SHOT\_INJURED*;  
 }  
  
 public Field getField() {  
 return field;  
 }  
  
 public ArrayList<Cell> getCells() {  
 return cells;  
 }  
  
 public ArrayList<Cell> getArounds() {  
 return arounds;  
 }  
}

package logic;  
  
public abstract class PlaceShip {  
 public Field field;  
  
 public PlaceShip(Ship ship){  
 this.field = ship.getField();  
 }  
  
 public abstract boolean setShip(int x, int y);  
 public abstract boolean setAround(int x, int y);  
}

package logic;  
  
public class PlaceShipCheck extends PlaceShip{  
 public PlaceShipCheck(Ship ship){  
 super(ship);  
 }  
  
 @Override  
 public boolean setShip(int x, int y){  
 if (field.isBound(x, y)){  
 return field.getCell(x, y).getState() == Cell.*CELL\_WATER*;  
 }  
 else{  
 return false;  
 }  
 }  
  
 @Override  
 public boolean setAround(int x, int y){  
 return true;  
 }  
}

package logic;  
  
public class PlaceShipSet extends PlaceShip{  
 private Ship ship;  
  
 public PlaceShipSet(Ship ship){  
 super(ship);  
 this.ship = ship;  
 }  
  
 @Override  
 public boolean setShip(int x, int y){  
 field.getCell(x, y).setState(Cell.*CELL\_WELL*);  
 ship.getCells().add(field.getCell(x, y));  
 field.getCell(x, y).setShip(ship);  
 return true;  
 }  
  
 @Override  
 public boolean setAround(int x, int y){  
 if (field.isBound(x, y)){  
 field.getCell(x, y).setState(Cell.*CELL\_AROUND*);  
 ship.getArounds().add(field.getCell(x, y));  
 }  
 return true;  
 }  
}

package ai;  
  
import logic.\*;  
  
import java.util.Random;  
  
public class AI {  
 public Field field;  
 public AIBase action;  
 public Random random;  
  
 public AI(Field field){  
 this.field = field;  
 this.action = new AIRandom(this);  
 this.random = new Random();  
 }  
  
 public int doShot(){  
 return action.doShot();  
 }  
  
 public Field getField() {  
 return field;  
 }  
}

package ai;  
  
public abstract class AIBase {  
 protected AI ai;  
 protected int x;  
 protected int y;  
 protected int dx;  
 protected int dy;  
  
 public AIBase(AI ai){  
 this.ai = ai;  
 }  
  
 protected abstract int doShot();  
  
 protected void setPosition(int x, int y){  
 this.x = x;  
 this.y = y;  
 }  
  
 protected void setDirection(int dx, int dy){  
 this.dx = dx;  
 this.dy = dy;  
 }  
}

package ai;  
  
import logic.Cell;  
import logic.Field;  
  
import java.util.ArrayList;  
  
public class AIRandom extends AIBase{  
 public AIRandom(AI ai){  
 super(ai);  
 }  
  
 @Override  
 public int doShot(){  
 ArrayList<Cell> cells = new ArrayList<>();  
 for (int i = 0; i < ai.getField().getWidth(); i++) {  
 for (int j = 0; j < ai.getField().getHeight(); j++) {  
 Cell cell = ai.getField().getCell(j, i);  
 if (!cell.isMark()){  
 cells.add(cell);  
 }  
 }  
 }  
 if (cells.size() == 0){  
 return Field.*SHOT\_MISSED*;  
 }  
 Cell cell = cells.get(ai.random.nextInt(cells.size()));  
 int shot = cell.doShot();  
 if (shot == Field.*SHOT\_INJURED*){  
 ai.action = new AIAround(ai);  
 ai.action.setPosition(cell.x, cell.y);  
 }  
 return shot;  
 }  
}

package ai;  
  
import logic.Cell;  
import logic.Field;  
  
import java.util.ArrayList;  
  
public class AIAround extends AIBase{  
 public AIAround(AI ai) {  
 super(ai);  
 }  
  
 @Override  
 public int doShot() {  
 int n, m;  
 ArrayList<Cell> cells = new ArrayList<>();  
 for (int i = 0; i < 2; i++) {  
 n = y;  
 m = x + i \* 2 - 1;  
 if (ai.getField().isBound(m, n)){  
 Cell cell = ai.getField().getCell(m, n);  
 if (!cell.isMark()){  
 cells.add(cell);  
 }  
 }  
 n = y + i \* 2 - 1;  
 m = x;  
 if (ai.getField().isBound(m, n)){  
 Cell cell = ai.getField().getCell(m, n);  
 if(!cell.isMark()){  
 cells.add(cell);  
 }  
 }  
 }  
 if (cells.size() > 0){  
 Cell cell = cells.get(ai.random.nextInt(cells.size()));  
 int shot = cell.doShot();  
 if (shot == Field.*SHOT\_INJURED*){  
 ai.action = new AIKilling(ai);  
 ai.action.setPosition(x, y);  
 ai.action.setDirection(cell.x - x, cell.y - y);  
 }  
 return shot;  
 }  
 ai.action = new AIRandom(ai);  
 return ai.doShot();  
 }  
}

package ai;  
  
import logic.Cell;  
  
import java.util.ArrayList;  
  
public class AIKilling extends AIBase{  
 public AIKilling(AI ai) {  
 super(ai);  
 }  
  
 @Override  
 protected int doShot() {  
 ArrayList<Cell> cells = new ArrayList<>();  
 draw(cells, dx, dy);  
 draw(cells, -dx, -dy);  
  
 if (cells.size() > 0){  
 return cells.get(ai.random.nextInt(cells.size())).doShot();  
 }  
  
 ai.action = new AIRandom(ai);  
 return ai.doShot();  
 }  
  
 public void draw(ArrayList<Cell> cells, int i, int j){  
 int m = x;  
 int n = y;  
  
 do{  
 m += i;  
 n += j;  
 } while ((ai.getField().isBound(m, n)) && (ai.getField().getCell(m, n).getState() == Cell.*CELL\_INJURED*));  
  
 if (ai.getField().isBound(m, n)){  
 Cell cell = ai.getField().getCell(m, n);  
 if (!cell.isMark()){  
 cells.add(cell);  
 }  
 }  
 }  
}