Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Волгоградский государственный технический университет»

Факультет электроники и вычислительной техники

Кафедра «Программное обеспечение автоматизированных систем»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе**

по дисциплине «Объектно-ориентированный анализ и программирование» на тему: «Проектирование программы с использованием объектно- ориентированного подхода»

(индивидуальное задание – вариант №27\_01)

Студент: Оганесян Артем Артакович Группа: ПрИн-366

Работа зачтена с оценкой « 1 » июня 2023 г.

Руководитель проекта, нормоконтроллер Литовкин Д.В.

Волгоград 2023 г.

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Волгоградский государственный технический университет»

Факультет электроники и вычислительной техники Направление 09.03.04 «Программная инженерия»

Кафедра «Программное обеспечение автоматизированных систем» Дисциплина «Объектно-ориентированный анализ и программирование»

Утверждаю

Зав. кафедрой Орлова Ю.А.

**ЗАДАНИЕ**

**на курсовую работу**

Студент: Оганесян А.А. Группа: ПрИн-366

1. Тема: «Проектирование программы с использованием объектно- ориентированного подхода» (индивидуальное задание – вариант №27) Утверждена приказом от «9» января 2023г. № 13-ст
2. Срок представления работы к защите « 01 » июня 2023 г.
3. Содержание пояснительной записки:

формулировка задания, требования к программе, структура программы, типовые процессы в программе, человеко-машинное взаимодействие, код программы и модульных тестов

1. Перечень графического материала:
2. Дата выдачи задания «15» февраля 2023 г.

Руководитель проекта: Литовкин Д.В.

Задание принял к исполнению: Оганесян А.А.

«15» февраля 2023 г.

Оглавление

[1 Формулировка задания 4](#_Toc136687600)

[2 Нефункциональные требования 5](#_Toc136687601)

[3 Первая итерация разработки 6](#_Toc136687602)

[3.1 Формулировка упрощенного варианта задания 6](#_Toc136687603)

[3.2 Функциональные требования (сценарии) 7](#_Toc136687604)

[3.3 Словарь предметной области 9](#_Toc136687605)

[3.4 Диаграмма классов 10](#_Toc136687606)

[3.5 Диаграммы последовательности 11](#_Toc136687607)

[3.6 Человеко-машинное взаимодействие 15](#_Toc136687608)

[3.7 Реализация ключевых классов 16](#_Toc136687609)

[3.8 Реализация ключевых тестовых случаев 30](#_Toc136687610)

[4 Вторая итерация разработки 37](#_Toc136687611)

**[4.1 Функциональные требования(сценарии) 37](#_Toc136687612)**

**[4.2 Словарь предметной области 39](#_Toc136687613)**

**[4.3 Структура программы на уровне классов 40](#_Toc136687614)**

**[4.4 Типовые процессы в программе 41](#_Toc136687615)**

[**4.5 Человеко-машинное взаимодействие 45**](#_Toc136687616)

[**4.6 Реализация ключевых классов 47**](#_Toc136687617)

[**4.7 Реализация ключевых тестовых случаев 62**](#_Toc136687618)

[5 Список использованной литературы и других источников 71](#_Toc136687619)

# 1 Формулировка задания

Игра "Запутанная схема"

Правила игры "Запутанная схема":

- количество узлов не менее 6

- минимальное количество нитей = количество узлов \* 2

- выигрыш определяется отсутствием пересекающихся нитей

* цель игры — распутать нитки

Подвариант 1: необходимо предусмотреть в программе точки расширения, используя которые можно реализовать вариативную часть программы (в дополнение к базовой функциональности).

Вариативность: предусмотреть появление узлов с различным поведением движения. Узлы должны быть визуально различимы.

НЕ изменяя ранее созданные классы, а используя точки расширения, реализовать:

узел, который не может двигаться

узел, который движется только по горизонтали

# 2 Нефункциональные требования

1. Программа должна быть реализована на языке Java SE 12 с использованием стандартных библиотек, в том числе, библиотеки Swing.
2. Форматирование исходного кода программы должно соответствовать Java Code Conventions, September 12, 1997.

# 3 Первая итерация разработки

## Формулировка упрощенного варианта задания

Игра "Запутанная схема". Правила игры:

- количество узлов не менее 6

- минимальное количество нитей = количество узлов \* 2

- выигрыш определяется отсутствием пересекающихся нитей

* цель игры — распутать нитки

## Функциональные требования (сценарии)

**Сценарии игры «Запутанная схема».**

Главный успешный сценарий

* + - 1. **Игрок** начинает **игру**
      2. Игра создает игровое **поле(**с помощью **уровня)** – **клубок**. **Клубок** состоит из **узлов** и **ниток**
      3. Пока в **клубке** не осталось пересечений **ниток**

3.1 **Игрок** двигает **узел**

3.2 **Нить** проверяет пересечение с другой **нитью**

3.3 Если пересечений с другой **нитью** нет

3.3.1 **Игра** просит **клубок** проверить, не осталось ли пересечений **ниток**

3.3.2 Если пересечений не осталось

3.3.2.1 **Игра** окончена

3.3.3 Иначе

3.3.3.1 Возвращаемся к действию 3.1

Дочерний сценарий – движение узла

1. **Игрок** двигает **узел**
   1. **Нити**, смежные с передвинутым **узлом**, проверяют пересечение с другими **нитями**
   2. Если пересечений нет
      1. **Игра** обращается к **клубку**, для проверки на окончание игровой сессии
   3. Иначе
      1. Возвращаемся к действию 1.1

Дочерний сценарий – определение конца игры

1. **Клубок** проверят все **нити**, на наличие пересечений между ними
   1. Если пересечения остались
      1. Дочерний сценарий – **движение узла**
   2. Иначе
      1. **Игра** считается завершенной

## Словарь предметной области

**Игра** – игра знает о поле. Игра начинает игровую сессию и заканчивает ее.

**Узел** – круглая область поля. Узел имеет направление перемещения. Узел можно двигать.

**Уровень** – начальная расстановка на поле. Определяет начальную расстановку узлов и нитей на поле.

**Клубок** – определяет, запутан ли он. Состоит из нитей и узлов.

**Нить** – знает о двух узлах – конце и начале нити. Нить может проверить себя на пересечение с другими нитями.

## Диаграмма классов

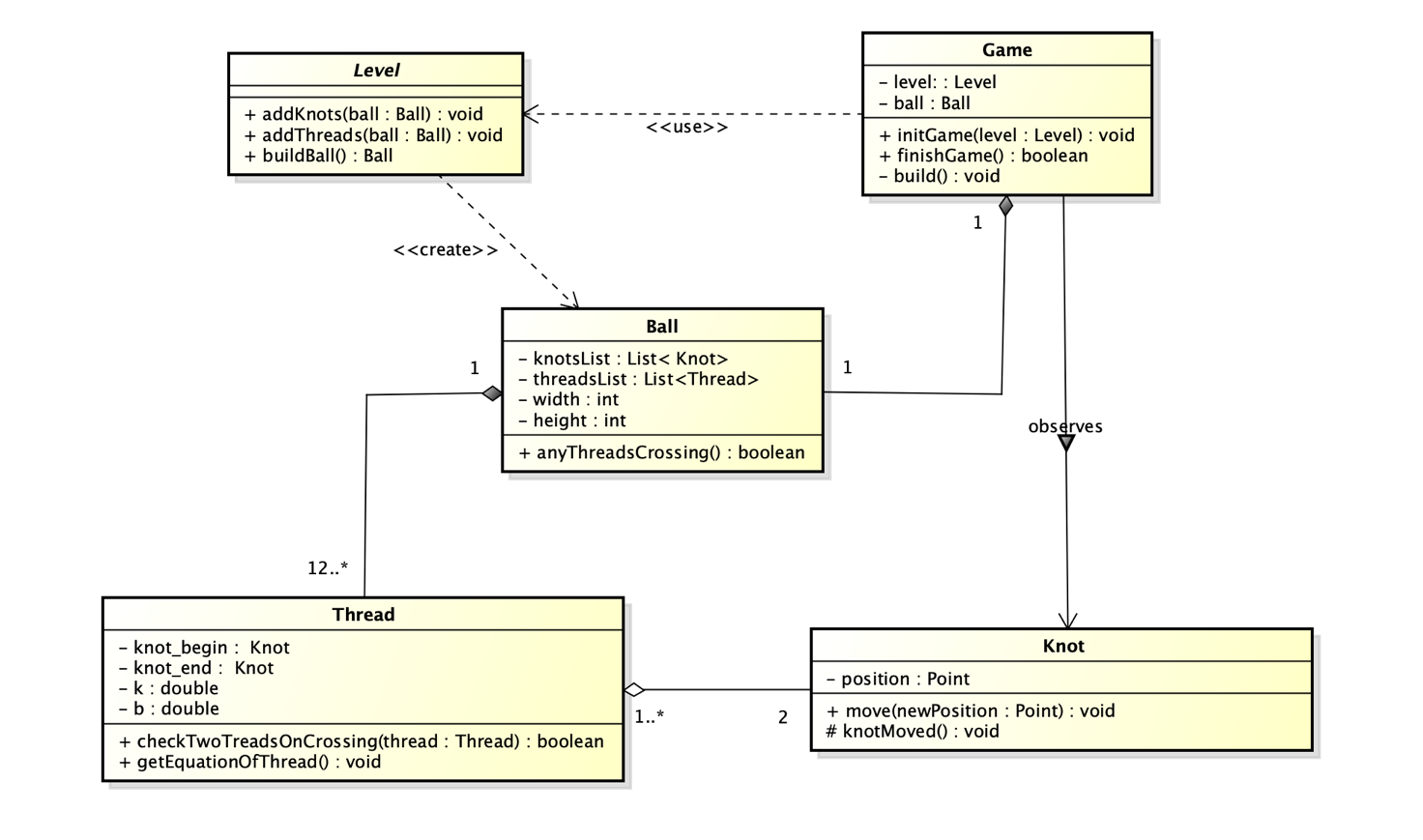


Диаграмма классов вычислительной модели

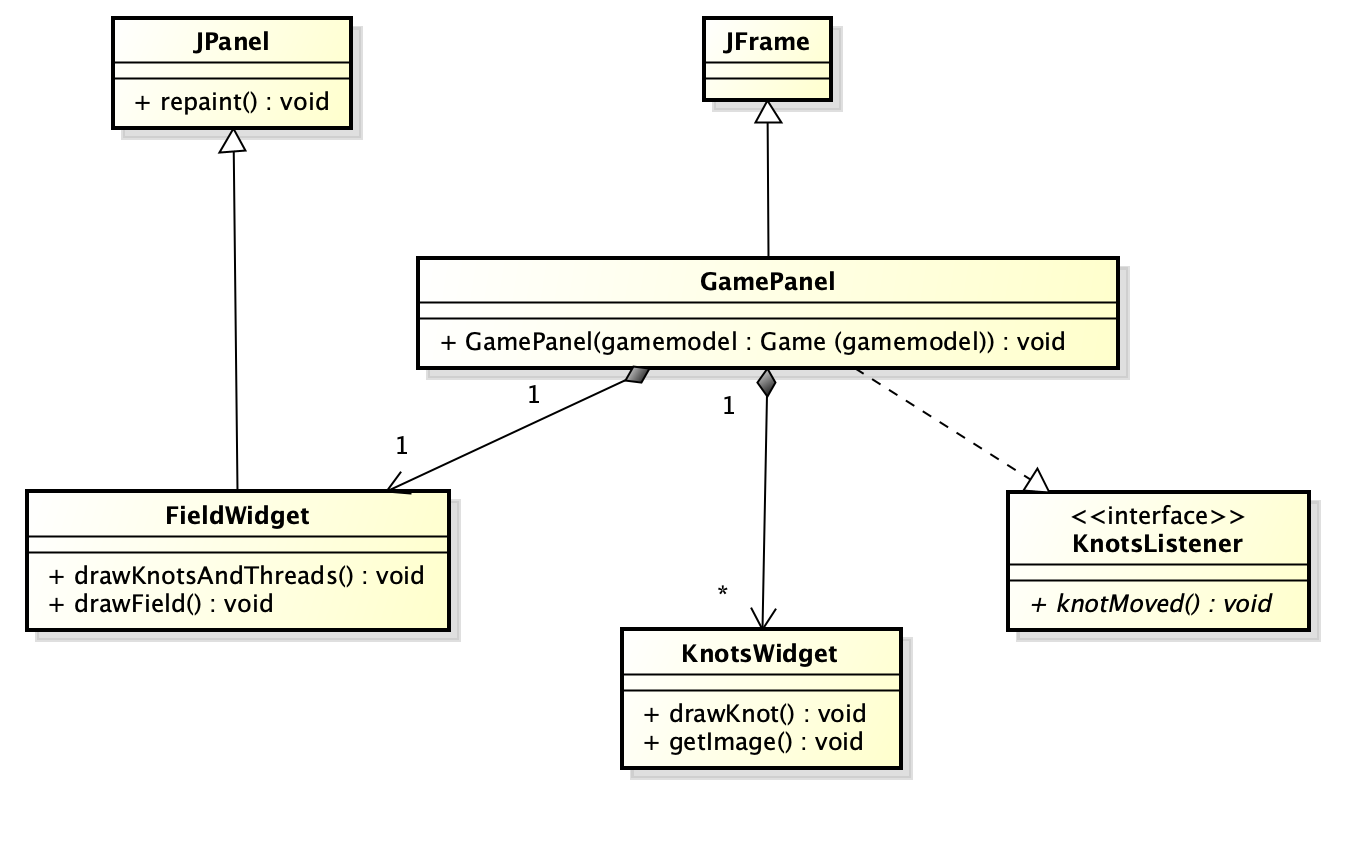
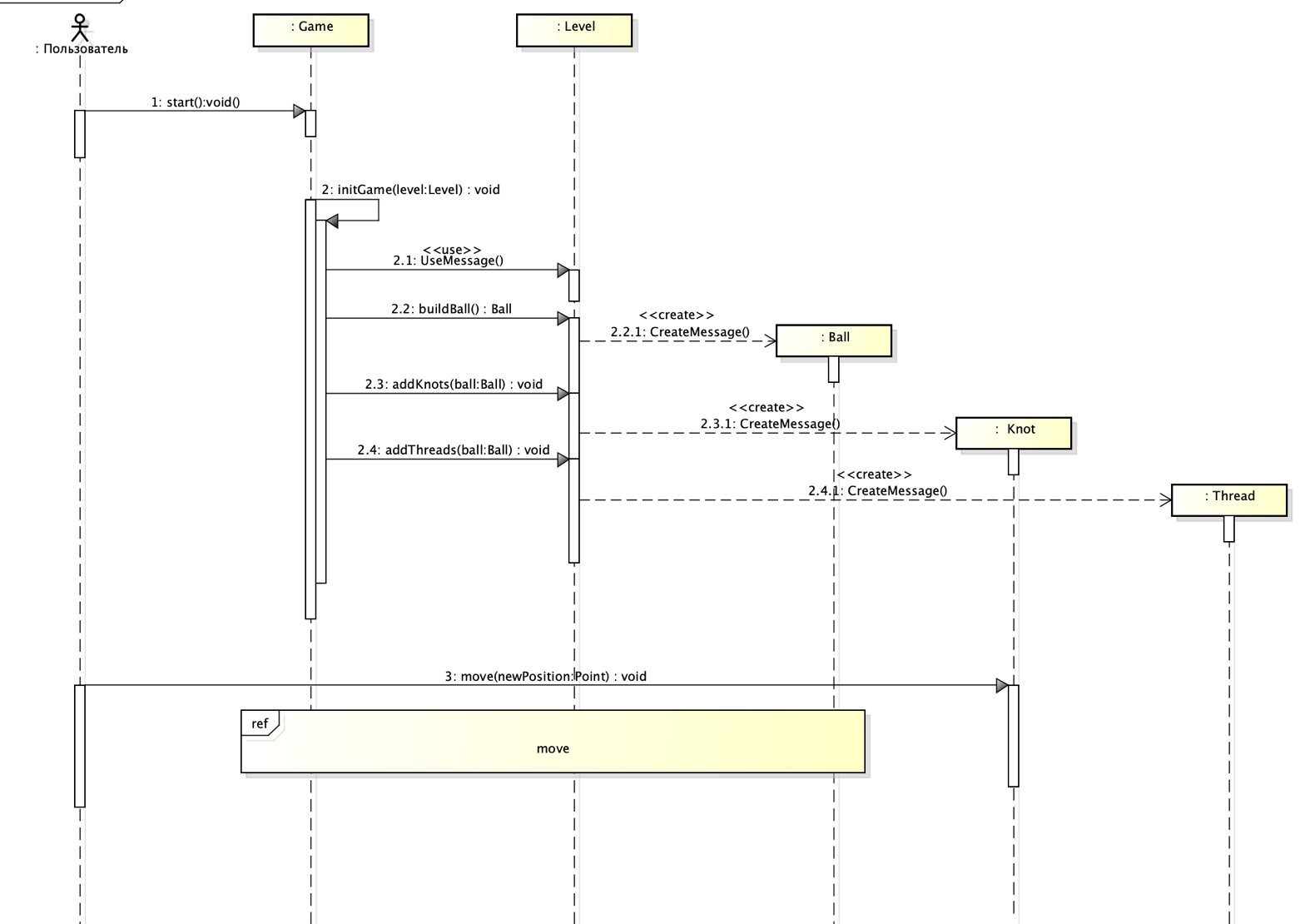
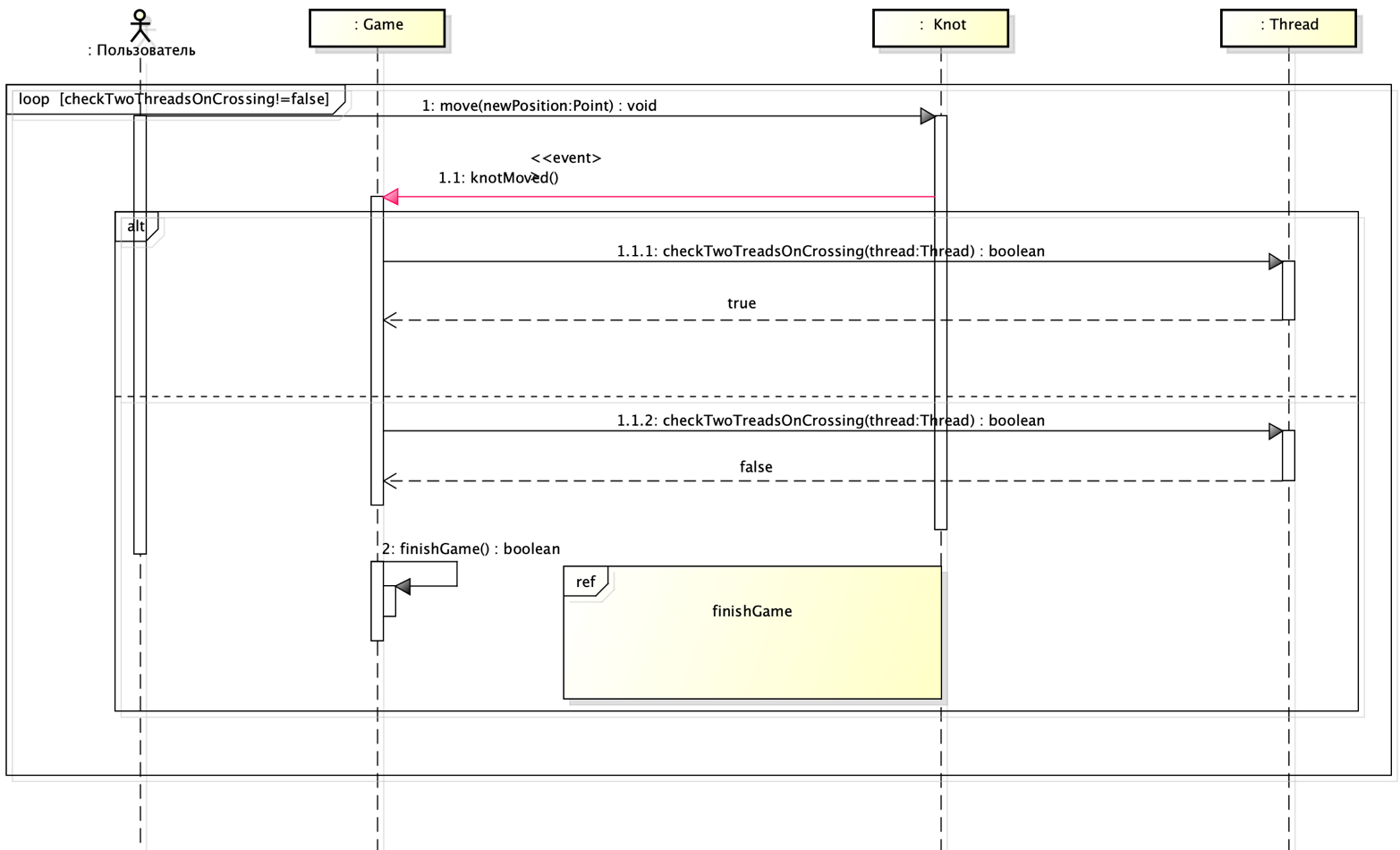
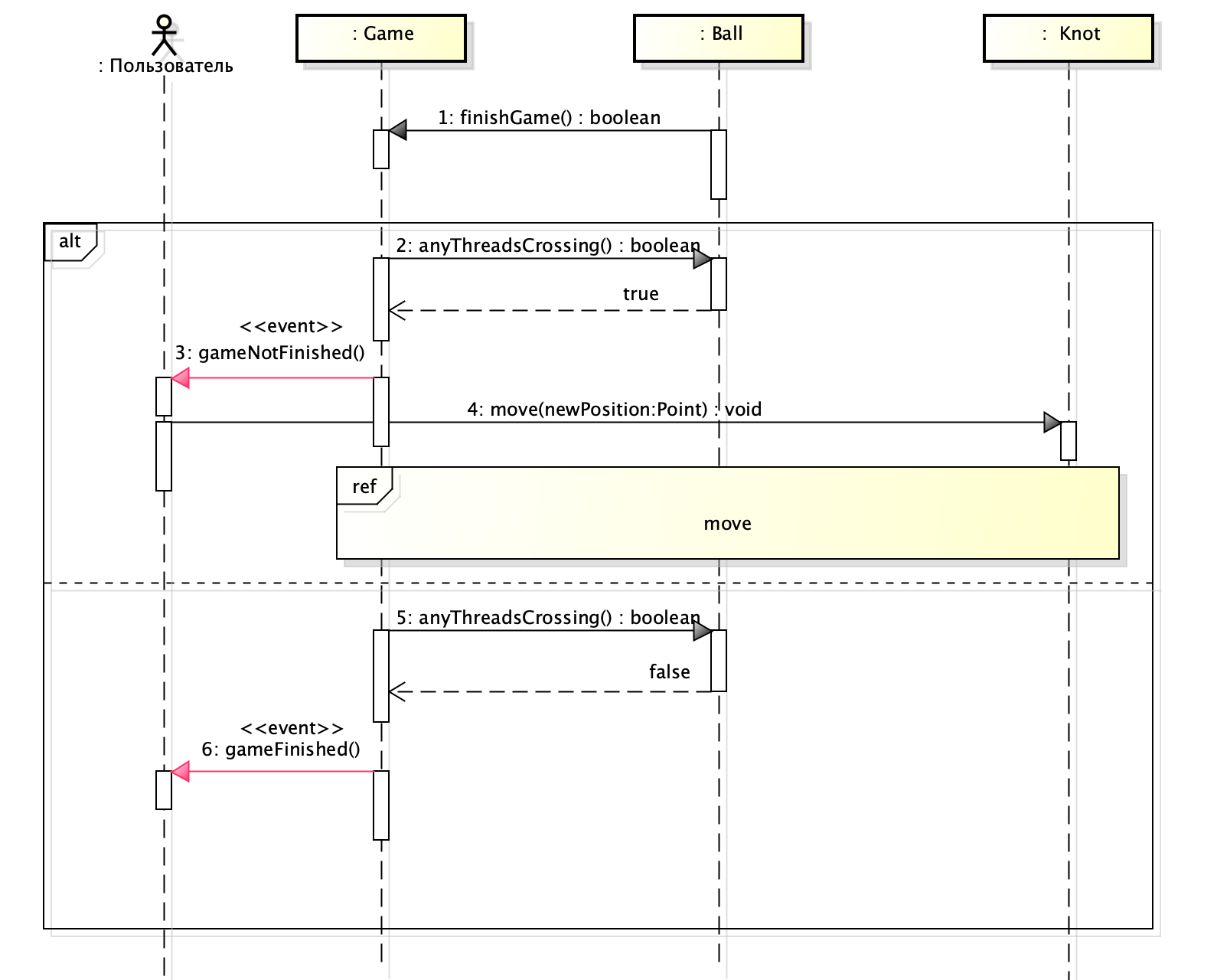


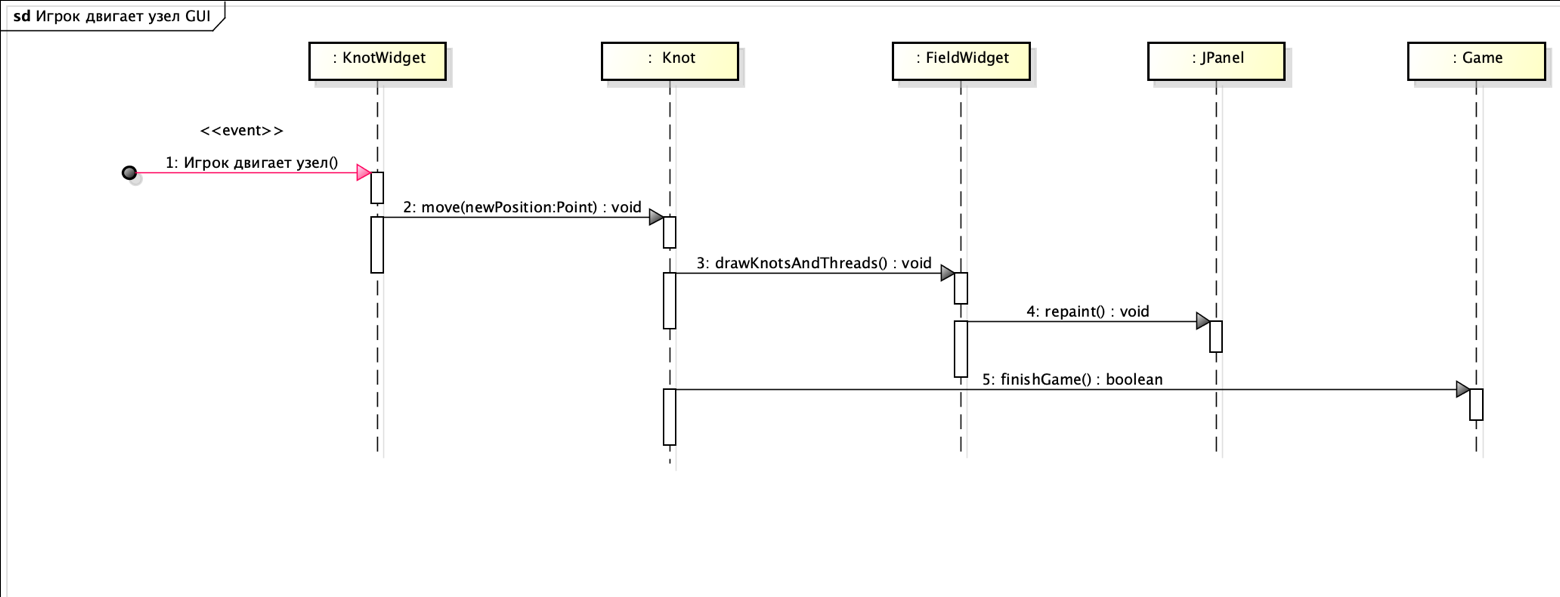
Диаграмма классов представления

## Диаграммы последовательности

1. **Общий игровой цикл**
2. **Движение узла**

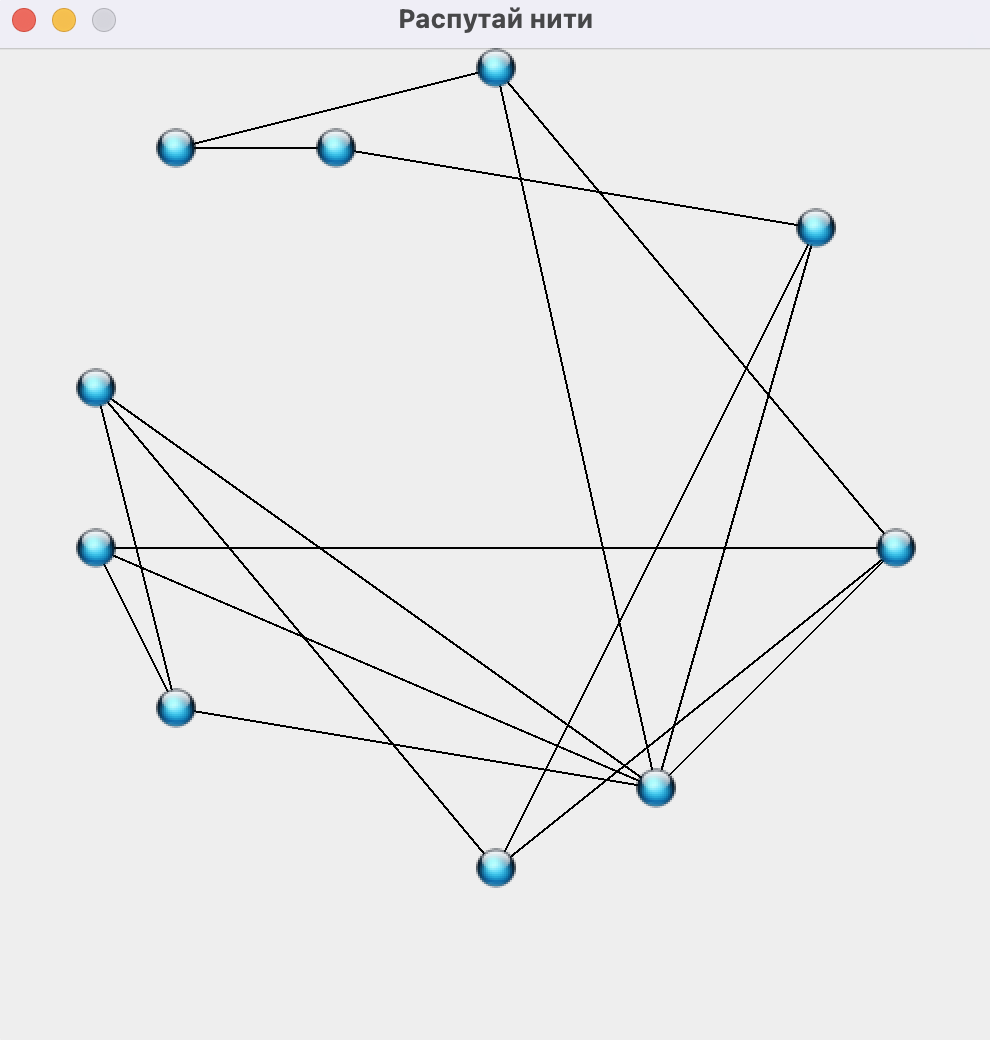
****

1. **Определение конца игры**
2. **Игрок двигает узел**



## Человеко-машинное взаимодействие

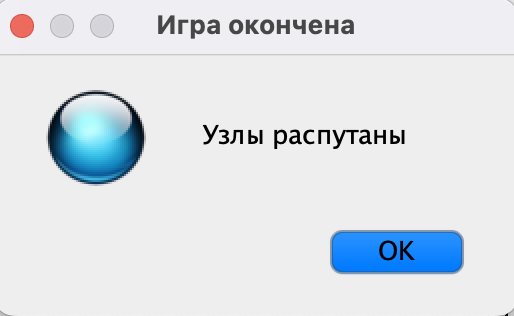
Общий вид главного окна представлен ниже. На нем располагается поле, узлы и нити.



По нажатию на узел, курсор изменяется (HAND\_CURSOR). Если курсор изменился, значит узел доступен для перемещения

**Окончание игры.**

По окончании игры выводится сообщение: «Узла распутаны»



Окончание игры

## Реализация ключевых классов

Класс Узел(Knot)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Нить(Thread)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Уровень(Level)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Клубок(Ball)

package main.model;  
  
import java.util.ArrayList;  
  
public class Ball {  
  
 private int width;  
 private int height;  
  
 */\*\*  
 \* Список узлов  
 \*/* private ArrayList<Knot> knots = new ArrayList<>();  
  
 */\*\*  
 \* Список нитей  
 \*/* private ArrayList<Thread> threads = new ArrayList<>();  
  
 public Ball(int width, int height){  
 if(width <= 0) throw new IllegalArgumentException("Field width must be more than 0");  
 if(height <= 0) throw new IllegalArgumentException("Field height must be more than 0");  
 this.width = width;  
 this.height = height;  
 }  
  
 public int getWidth() {  
 return this.width;  
 }  
 public int getHeight() {  
 return this.height;  
 }  
  
 protected void setKnots(ArrayList<Knot> knots){  
 this.knots = knots;  
 }  
 protected void setThreads(ArrayList<Thread> threads){  
 this.threads = threads;  
 }  
 public ArrayList<Knot> getKnots(){  
 return this.knots;  
 }  
 public ArrayList<Thread> getThreads(){  
 return this.threads;  
 }  
 public void addKnot(Knot k){  
 this.knots.add(k);  
 }  
  
 */\*\*  
 \* Проверить, не осталось ли пересечений нитей  
 \* @return false - пересечений не осталось, true - остались пересечения  
 \*/* public boolean anyThreadsCrossing(){  
 boolean threadsCrossing = false;  
 for(int i = 0; i < threads.size(); i++){  
 for(int j = i+1; j < threads.size(); j++){  
 threadsCrossing = getThreads().get(i).checkTwoThreadsOnCrossing(getThreads().get(j));  
 if(threadsCrossing)  
 return true;  
 }  
 }  
 return false;  
 }  
}

Класс Игра(Game)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import javax.swing.\*;  
  
import static javax.swing.JOptionPane.*showMessageDialog*;  
  
public class Game {  
 private Level level;  
  
 private Ball ball;  
  
 public Game(Level level){  
 this.level = level;  
 initGame();  
 }  
  
 */\*\*  
 \* Постоить поле и обстановку на нем  
 \*/* private void build(){  
 ball = level.buildBall();  
 }  
  
 */\*\*  
 \* Начать игру  
 \*/* public void initGame(){  
 build();  
 for(var i: ball.getKnots()){  
 i.addKnotListener(new KnotObserver());  
 }  
 }  
  
 */\*\*  
 \* Проверить закончилась ли игра  
 \* @return результат проверки  
 \*/* public boolean finishGame(){  
 return !ball.anyThreadsCrossing();  
 }  
  
  
 */\*\*  
 \* События  
 \*/* private class KnotObserver implements KnotListener {  
 @Override  
 public void knotMoved(KnotEvent e) {  
   
 }  
  
 @Override  
 public void knotReleased(KnotEvent e){  
 if(finishGame()){  
 *showMessageDialog*(null, "Узлы распутаны", "Игра окончена", JOptionPane.*WARNING\_MESSAGE*, new ImageIcon(getClass().getResource("/knot.png")));  
 }  
 }  
 }  
}

Классы событий(events)

package main.events;  
  
import main.model.Knot;  
  
import java.util.EventObject;  
  
public class KnotEvent extends EventObject {  
  
 private Knot knot;  
 public KnotEvent(Object source) {  
 super(source);  
 }  
  
 public void setKnot(Knot knot) { this.knot = knot; }  
  
 public Knot knot(){ return this.knot; }  
}

package main.events;  
  
import java.util.EventListener;  
  
public interface KnotListener extends EventListener {  
 void knotMoved(KnotEvent e);  
 void knotReleased(KnotEvent e);  
}

Виджеты

private int FRAME\_HEIGHT = 500;  
private int FRAME\_WIDTH = 500;  
private Level level;  
private final int IMAGE\_SIZE = 20;  
private JPanel panel;  
boolean isPressedMouse = false;  
boolean isChosenKnot = false;  
int numOfChosenKnot = -1;  
  
public static void main(String[] args) {  
  
 new Main().setVisible(true);  
}  
  
private Main(){  
 level = new Level();  
 level.buildBall();  
 Game game = new Game(level);  
 setImages();  
 InitPanel();  
 InitFrame();  
}  
  
private void InitPanel(){  
  
 panel = new JPanel(){  
 @Override  
 protected void paintComponent(Graphics g) {  
  
 super.paintComponent(g);  
  
 for(int i = 0; i < level.ball.getThreads().size();i++){  
 g.drawLine( (int) level.ball.getThreads().get(i).getBegin().getPosition().getX()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getBegin().getPosition().getY()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getEnd().getPosition().getX()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getEnd().getPosition().getY()\*IMAGE\_SIZE+IMAGE\_SIZE/2);  
 }  
  
 for(int i = 0; i < level.ball.getKnots().size(); i++ ) {  
 g.drawImage((Image) level.ball.getKnots().get(i).*image*, level.ball.getKnots().get(i).getPosition().x \* IMAGE\_SIZE, level.ball.getKnots().get(i).getPosition().y \* IMAGE\_SIZE, this);  
 }  
  
  
 }  
 };  
  
 panel.addMouseListener(new MouseAdapter() {  
 @Override  
 public void mousePressed(MouseEvent e) {  
 // System.out.println("Мышь зажата");  
 isPressedMouse = true;  
 panel.setCursor(Cursor.*getPredefinedCursor*(Cursor.*HAND\_CURSOR*));  
 for(int i = 0; i < level.ball.getKnots().size(); i++){  
 if(level.ball.getKnots().get(i).getPosition().getX() == e.getX()/IMAGE\_SIZE  
 && level.ball.getKnots().get(i).getPosition().getY() == e.getY()/IMAGE\_SIZE){  
 numOfChosenKnot = i;  
 isChosenKnot = true;  
 }  
 }  
 }  
 });  
  
 panel.addMouseMotionListener(new MouseAdapter() {  
 @Override  
 public void mouseDragged(MouseEvent e) {  
 Knot currentKnot = level.ball.getKnots().get(numOfChosenKnot);  
 currentKnot.fireKnotMoved();  
 int x = currentKnot.getPosition().x;  
 int y = currentKnot.getPosition().y;  
 if(isPressedMouse){  
 if(isChosenKnot){  
 if(e.getX() < FRAME\_WIDTH && e.getY() < FRAME\_HEIGHT && e.getY() > 0 && e.getX() > 0){  
 Point newPosition = new Point(e.getX()/IMAGE\_SIZE,e.getY()/IMAGE\_SIZE);  
 currentKnot.move(newPosition);  
 }  
 }  
 }  
 panel.repaint();  
 }  
 });  
  
 panel.addMouseListener(new MouseAdapter() {  
 @Override  
 public void mouseReleased(MouseEvent e){  
 panel.setCursor(Cursor.*getDefaultCursor*());  
 level.ball.getKnots().get(numOfChosenKnot).fireKnotReleased();  
 isPressedMouse = false;  
 numOfChosenKnot = -1;  
 isChosenKnot = false;  
 panel.repaint();  
  
 }  
 });  
  
 panel.setPreferredSize(new Dimension(FRAME\_WIDTH,FRAME\_HEIGHT));  
 add(panel);  
}  
  
private void InitFrame(){  
 pack();  
 setDefaultCloseOperation(WindowConstants.*EXIT\_ON\_CLOSE* );  
 setTitle("Запутанная схема");  
 setLocationRelativeTo(null);  
 setResizable(false);  
 setVisible(true);  
}  
  
private void setImages(){  
 for (int i = 0; i < level.ball.getKnots().size(); i++){  
 Knot.*image* = getImage("/knot2.png");  
 }  
}  
  
private Image getImage(String name){  
 ImageIcon icon = new ImageIcon(getClass().getResource(name));  
 return icon.getImage();  
}

## Реализация ключевых тестовых случаев

Тесты для класса Узел

package tests;  
  
import main.model.Knot;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
  
public class KnotsTest {  
  
 @Test  
 public void constructorTest(){  
 Knot actual = new Knot(new Point(1,1));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected, actual.getPosition());  
 }  
  
 @Test  
 public void moveTest(){  
 Knot actual = new Knot(new Point(1,1));  
 actual.move(new Point(2,2));  
 Point expected = new Point(2,2);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
}

Тесты для класса Клубок

package tests;  
  
import main.model.Ball;  
import org.junit.Assert;  
import org.junit.Test;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertThrows*;  
  
  
public class BallTests {  
  
 // Неверный параметр ширины передан в конструктор  
 @Test  
 public void constructorTestingBadWidth()  
 {  
 *assertThrows*(IllegalArgumentException.class, () -> new Ball(0, 10));  
 }  
  
 // Неверный параметр высоты передан в конструктор  
 @Test  
 public void constructorTestingBadHeight()  
 {  
 *assertThrows*(IllegalArgumentException.class, () -> new Ball(10, 0));  
 }  
  
 @Test  
 public void constructorTestingGood(){  
 Ball actual = new Ball(10,10);  
 int expectedWidth = 10;  
 int expectedHeight = 10;  
 Assert.*assertEquals*(expectedHeight, actual.getHeight());  
 Assert.*assertEquals*(expectedWidth, actual.getWidth());  
 }  
  
 @Test  
 public void threadsNotCrossing(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(6,2));  
 Knot knot4 = new Knot(new Point(10,6));  
 Knot knot5 = new Knot(new Point(-2,-4));  
 Knot knot6 = new Knot(new Point(-4,-2));  
  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
 Thread thread3 = new Thread(knot5,knot6);  
 Thread thread4 = new Thread(knot1,knot3);  
  
  
 ArrayList<Thread> threads= new ArrayList<>();  
 threads.add(thread1);  
 threads.add(thread2);  
 threads.add(thread3);  
 threads.add(thread4);  
  
 Ball ball = new Ball(10,10);  
 ball.setThreads(threads);  
 Assert.assertEquals(false, ball.anyThreadsCrossing());  
 }  
  
 @Test  
 public void threadsCrossing(){  
 Level level = new Level();  
 level = new Level();  
 level.buildBall();  
 Assert.assertEquals(true, level.ball.anyThreadsCrossing());  
 }  
  
  
}

Тесты для класса Уровень

package tests;  
  
import main.model.Ball;  
import main.model.Level;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
public class LevelTests {  
  
 @Test  
 public void buildBallTestingOnSizes(){  
 Level level = new Level();  
 Ball ball = level.buildBall();  
 int expectedWidth = 10;  
 int expectedHeight = 10;  
 Assert.*assertEquals*(expectedHeight, ball.getHeight());  
 Assert.*assertEquals*(expectedWidth, ball.getWidth());  
  
 }  
  
 @Test  
 public void buildBallTestingOnCountKnotsAndThreads(){  
 Level level = new Level();  
 Ball ball = level.buildBall();  
 int expectedKnotsCount = 6;  
 int expectedThreadsCount = 9;  
 Assert.*assertEquals*(expectedKnotsCount, ball.getKnots().size());  
 Assert.*assertEquals*(expectedThreadsCount, ball.getThreads().size());  
 }  
}

Тесты для класса Нить

package tests;  
  
import main.model.Knot;  
import main.model.Thread;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
  
import static org.junit.Assert.*assertThrows*;  
  
public class ThreadTests {  
  
 @Test  
 public void constructorTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(2,2));  
 Thread actual = new Thread(knot1,knot2);  
  
 Knot expKnot1 = new Knot(new Point(1,1));  
 Knot expKnot2 = new Knot(new Point(2,2));  
  
 Assert.*assertEquals*(expKnot1, actual.getBegin());  
 Assert.*assertEquals*(expKnot2, actual.getEnd());  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithPositiveNumbers(){  
 Knot knot1 = new Knot(new Point(2,3));  
 Knot knot2 = new Knot(new Point(6,4));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = 0.25;  
 double expb = 2.5;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithNegativeNumbers(){  
 Knot knot1 = new Knot(new Point(-2,3));  
 Knot knot2 = new Knot(new Point(6,-4));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = -0.875;  
 double expb = 1.25;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithZeroes(){  
 Knot knot1 = new Knot(new Point(0,3));  
 Knot knot2 = new Knot(new Point(1,0));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = -3;  
 double expb = 3;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithSimilarCoordinates(){  
 Knot knot1 = new Knot(new Point(0,0));  
 Knot knot2 = new Knot(new Point(0,0));  
 Thread thread = new Thread(knot1,knot2);  
  
 *assertThrows*(IllegalArgumentException.class, () -> thread.getEquationOfStraight());  
 }  
  
 @Test  
 public void twoThreadsCrossingTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(2,3));  
 Knot knot4 = new Knot(new Point(4,2));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(true,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsNotCrossingTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(6,2));  
 Knot knot4 = new Knot(new Point(10,6));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(false,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsNotCrossingWithNegativeNumbersTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(-2,-3));  
 Knot knot4 = new Knot(new Point(-4,-8));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
  
 Assert.assertEquals(false,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsCrossingWithNegativeNumbersTest(){  
 Knot knot1 = new Knot(new Point(-6,-2));  
 Knot knot2 = new Knot(new Point(4,-10));  
 Knot knot3 = new Knot(new Point(-2,-3));  
 Knot knot4 = new Knot(new Point(-4,-8));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(true, thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
  
}

Тесты для класса Игра

package tests;  
  
import main.model.Ball;  
import main.events.KnotEvent;  
import main.events.KnotListener;  
import main.model.Level;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
import static org.junit.jupiter.api.Assertions.*assertTrue*;  
  
public class GameTests {  
  
 private enum EVENT {*KNOTMOVED*}  
 private ArrayList<EVENT> knots\_events = new ArrayList<>();  
  
 private class KnotObserver implements KnotListener{  
 @Override  
 public void knotMoved(KnotEvent e) {  
 knots\_events.add(EVENT.*KNOTMOVED*);  
 }  
  
 @Override  
 public void knotReleased(KnotEvent e){}  
 }  
  
 @Test  
 public void knotEvent(){  
 Level level = new Level();  
  
 Ball ball = level.buildBall();  
  
 for(var i: ball.getKnots()){  
 i.addKnotListener(new KnotObserver());  
 }  
  
 ball.getKnots().get(0).move(new Point(4,4));  
 *assertTrue*(!knots\_events.isEmpty());  
 }  
  
}

# 4 Вторая итерация разработки

### 4.1 Функциональные требования(сценарии)

Главный успешный сценарий

**Игрок** начинает **игру**

**Игра** создает **игровое поле**(с помощью **уровня**) – **клубок**. **Клубок** состоит из **узлов** и **ниток**

Пока в **клубке** не осталось пересечений **ниток**

3.1 **Игрок** двигает **узел**

3.1.1 Если **узел** способен к передвижению

3.1.1.1 Если **узел** можно подвинуть по всем направлениям(вертикальному и горизонтальному)

3.1.1.1.1 **Узел** поменял свое положение и по вертикальной и по горизонтальной оси

3.1.1.2 Если **узел** можно подвинуть только по горизонтальному направлению

3.1.1.1.2 **Узел** поменял свое положение только по горизонтальной оси

3.1.2 Если **узел** не способен к передвижению

3.1.2.1 **Узел** не поменял свое положение

3.2 **Нить** проверяет пересечение с другой **нитью**

3.3 Если пересечений с другой **нитью** нет

3.3.1 **Игра** просит **клубок** проверить, не осталось ли пересечений **ниток**

3.3.2 Если пересечений не осталось

3.3.2.1 **Игра** окончена

3.3.3 Иначе

3.3.3.1 Возвращаемся к действию 3.1

Дочерний сценарий – движение узла

1. **Игрок** двигает **узел**
   1. Если **узел** способен к передвижению
      1. Если **узел** можно подвинуть по всем направлениям(вертикальному и горизонтальному)
         1. **Узел** поменял свое положение и по вертикальной и по горизонтальной оси

1.1.2 Если **узел** можно подвинуть только по горизонтальному направлению

1.1.2.1 **Узел** поменял свое положение только по горизонтальной оси

1.2 Если **узел** не способен к передвижению

1.2.1 **Узел** не поменял свое положение

1. **Нити**, смежные с передвинутым **узлом**, проверяют пересечение с другими **нитями**
2. Если пересечений нет

3.1 **Игра** обращается к **клубку**, для проверки на окончание игровой сессии

4. Иначе

4.1 Возвращаемся к действию 1

Дочерний сценарий – определение конца игры

1. **Клубок** проверяет все **нити**, на наличие пересечений между ними

2. Если пересечения остались

2.1 Дочерний сценарий – движение **узла**

3. Иначе

3.1 **Игра** считается завершенной

### 4.2 Словарь предметной области

**Игра** – игра знает о поле. Игра начинает игровую сессию и заканчивает ее.

**Узел** – круглая область поля. Узел имеет направление перемещения. Узел можно двигать.

**Узел, двигающийся по горизонтали** – круглая область поля. Узел имеет направление перемещения. При передвижении у узла меняется только положение по горизонтали, по вертикали остается неизменным.

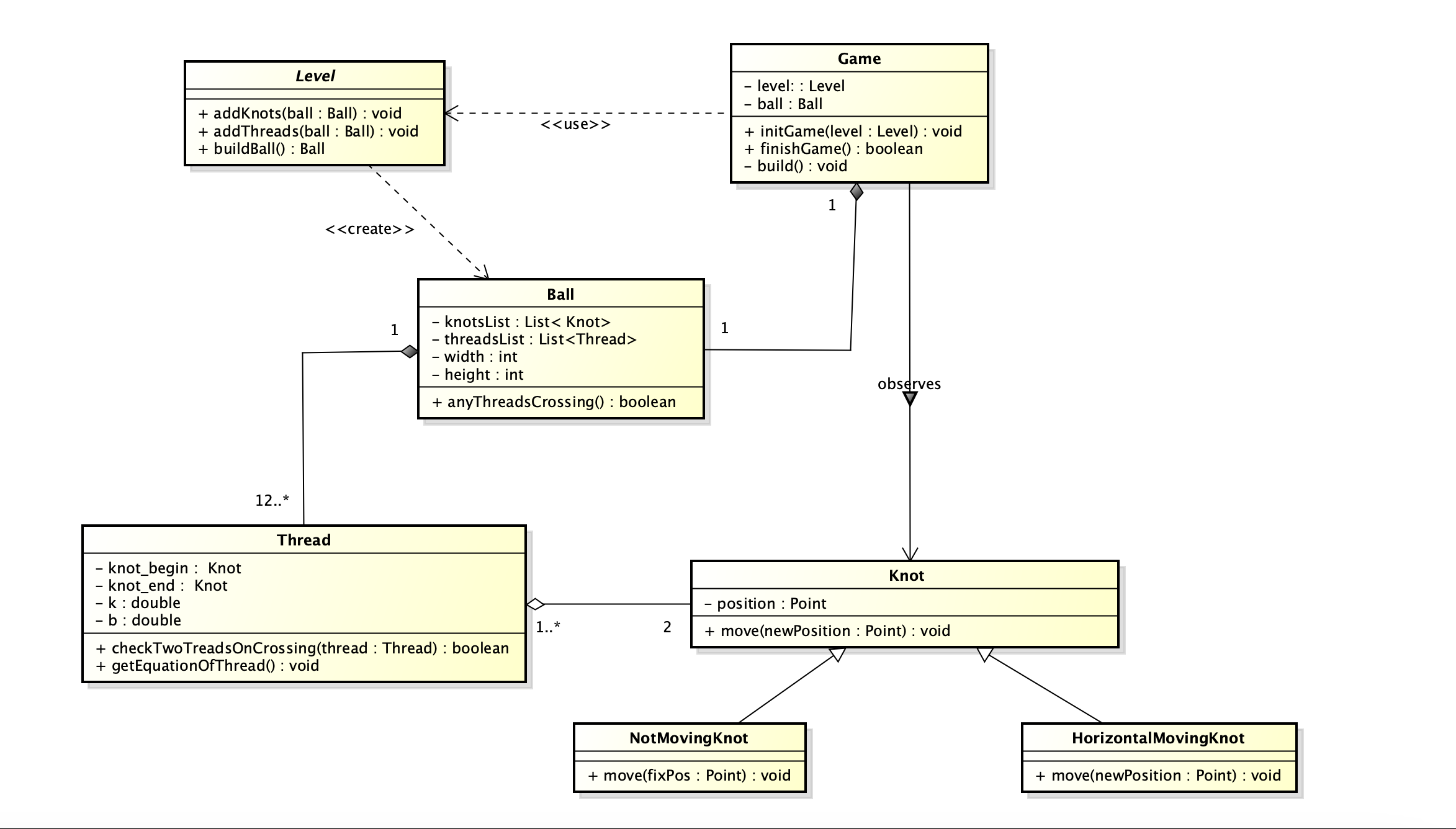
**Узел, не передвигающийся** – круглая область поля. Узел не имеет направление перемещения. Не передвигается.

**Уровень** – начальная расстановка на поле. Определяет начальную расстановку узлов и нитей на поле.

**Клубок** – определяет, запутан ли он. Состоит из нитей и узлов.

**Нить** – знает о двух узлах – конце и начале нити. Нить может проверить себя на пересечение с другими нитями.

### 4.3 Структура программы на уровне классов

****

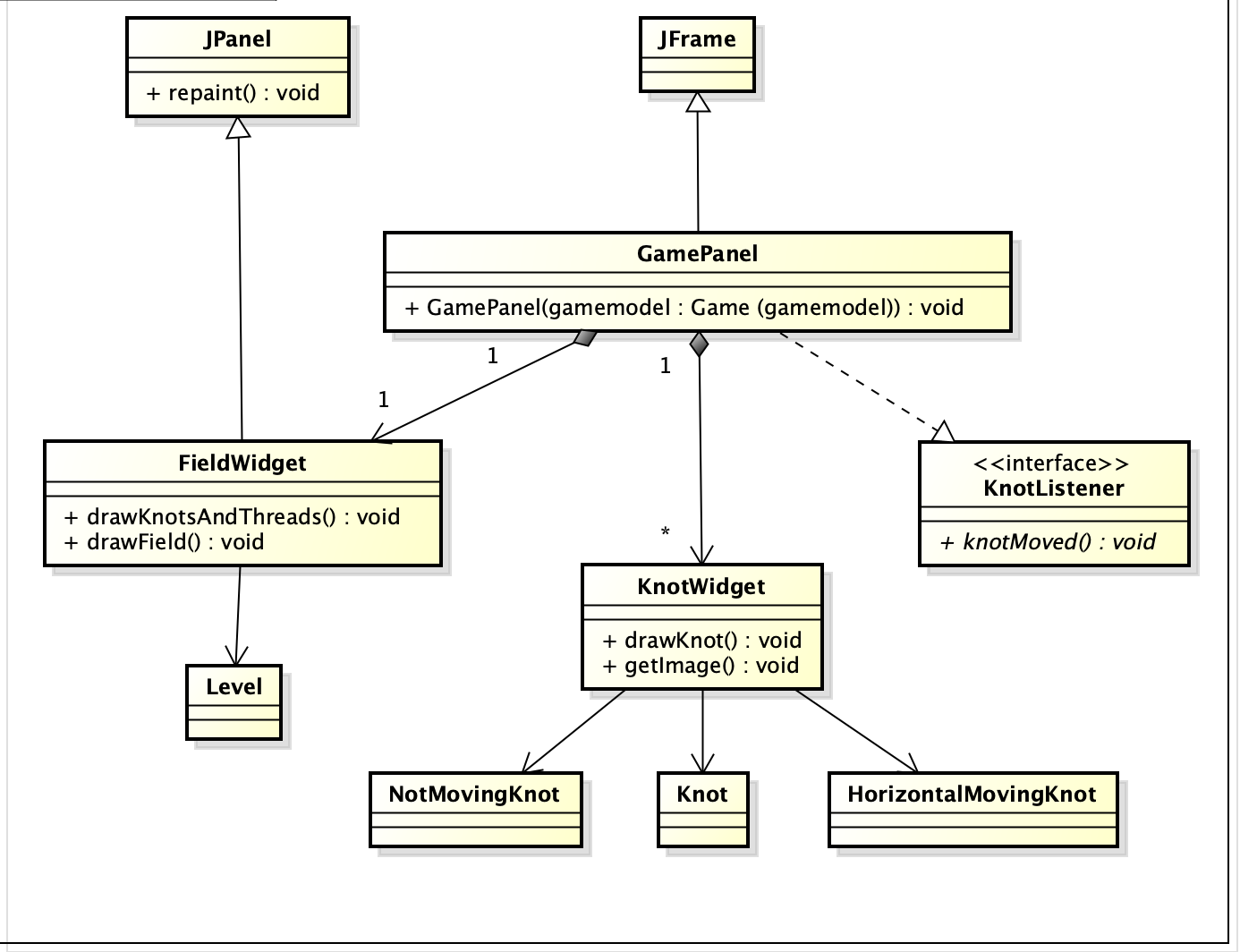
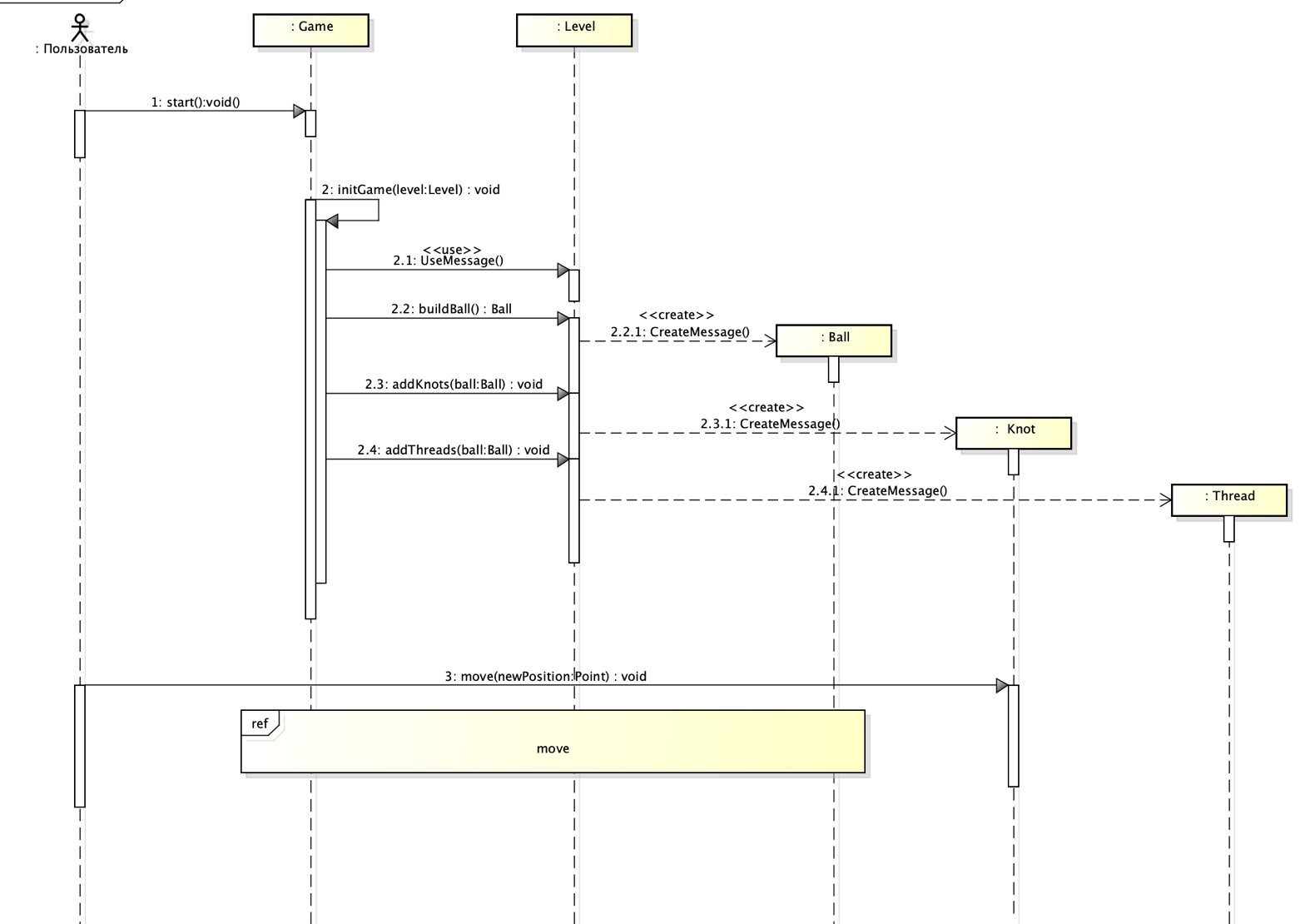
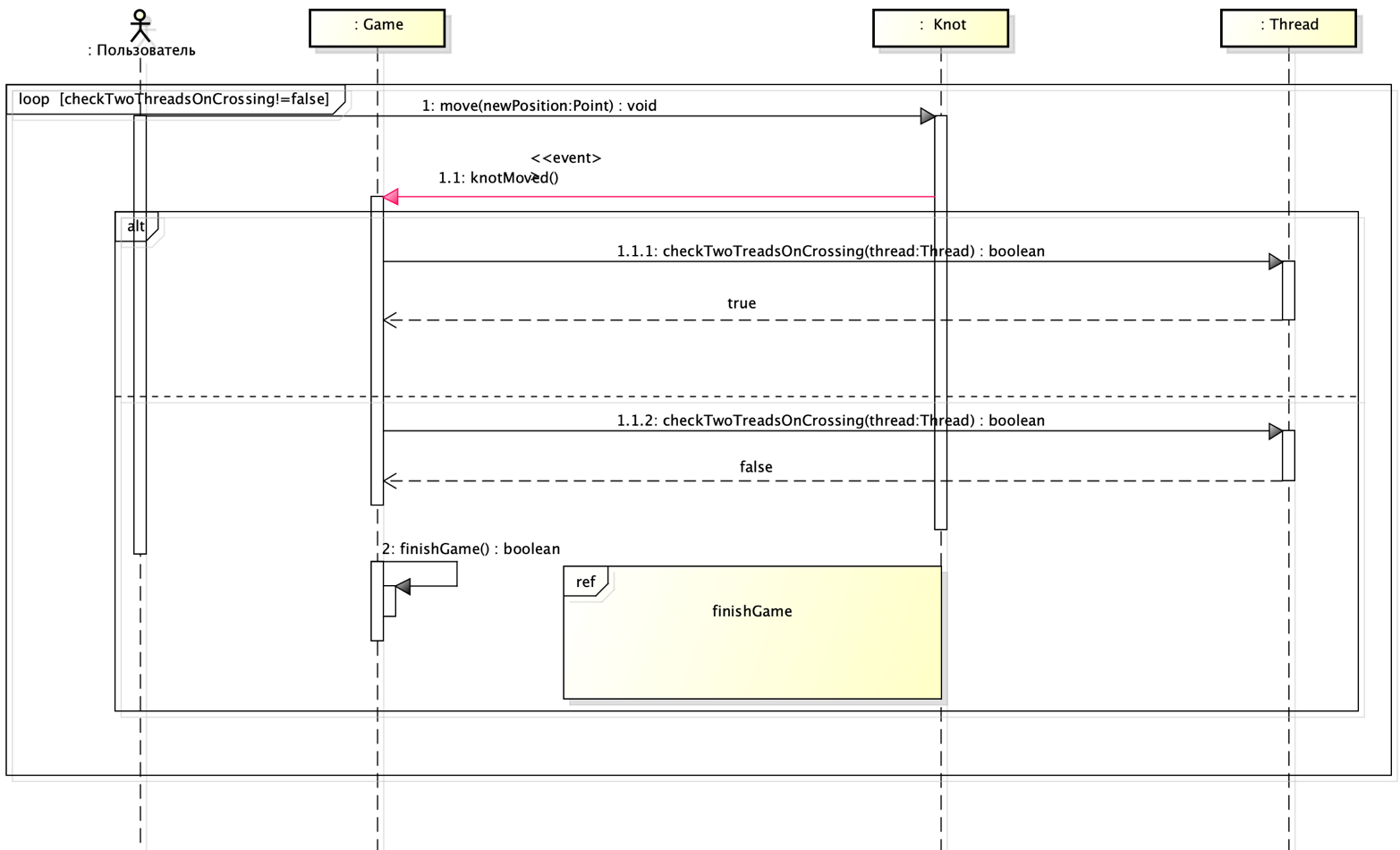
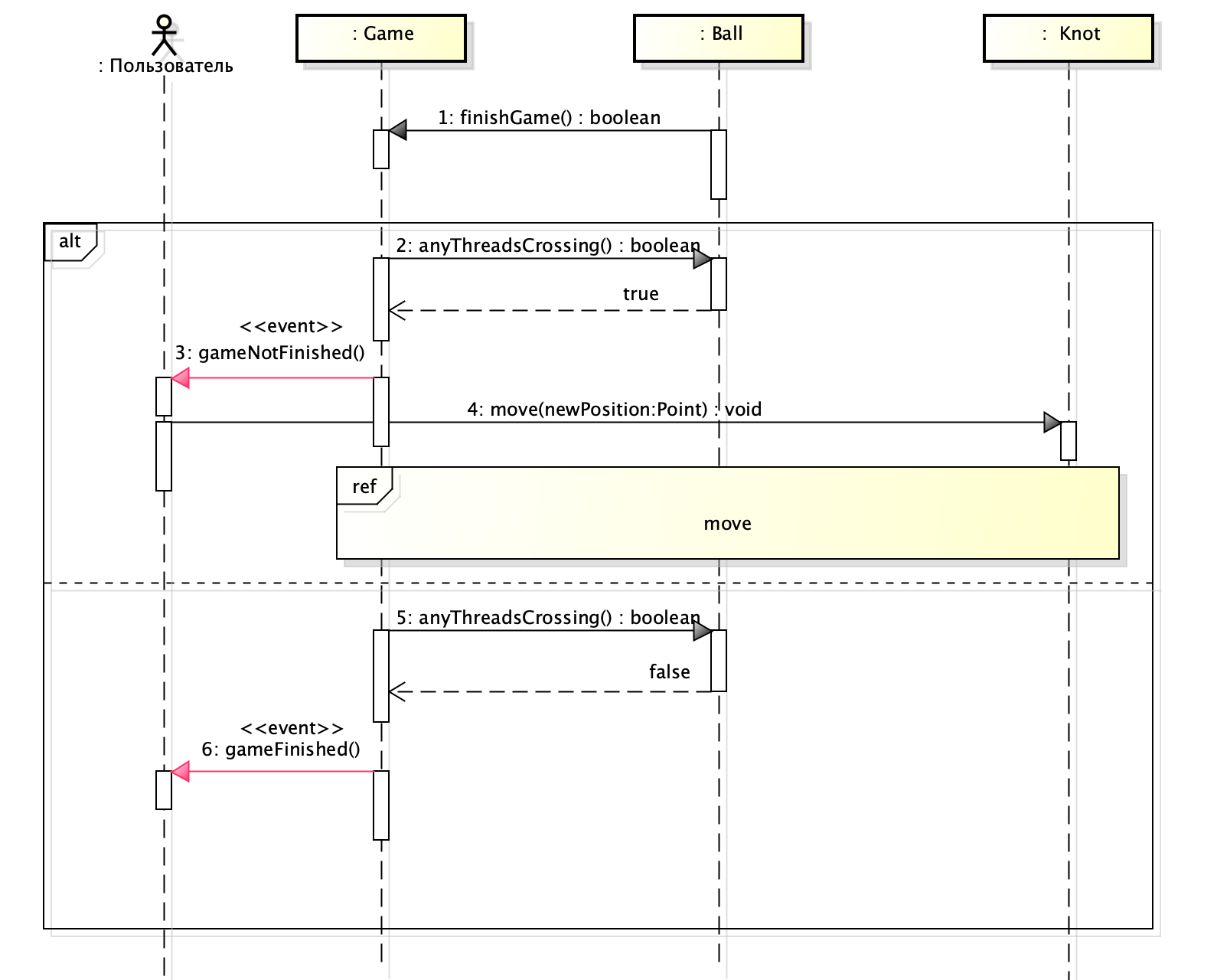
****Диаграмма классов вычислительной модели

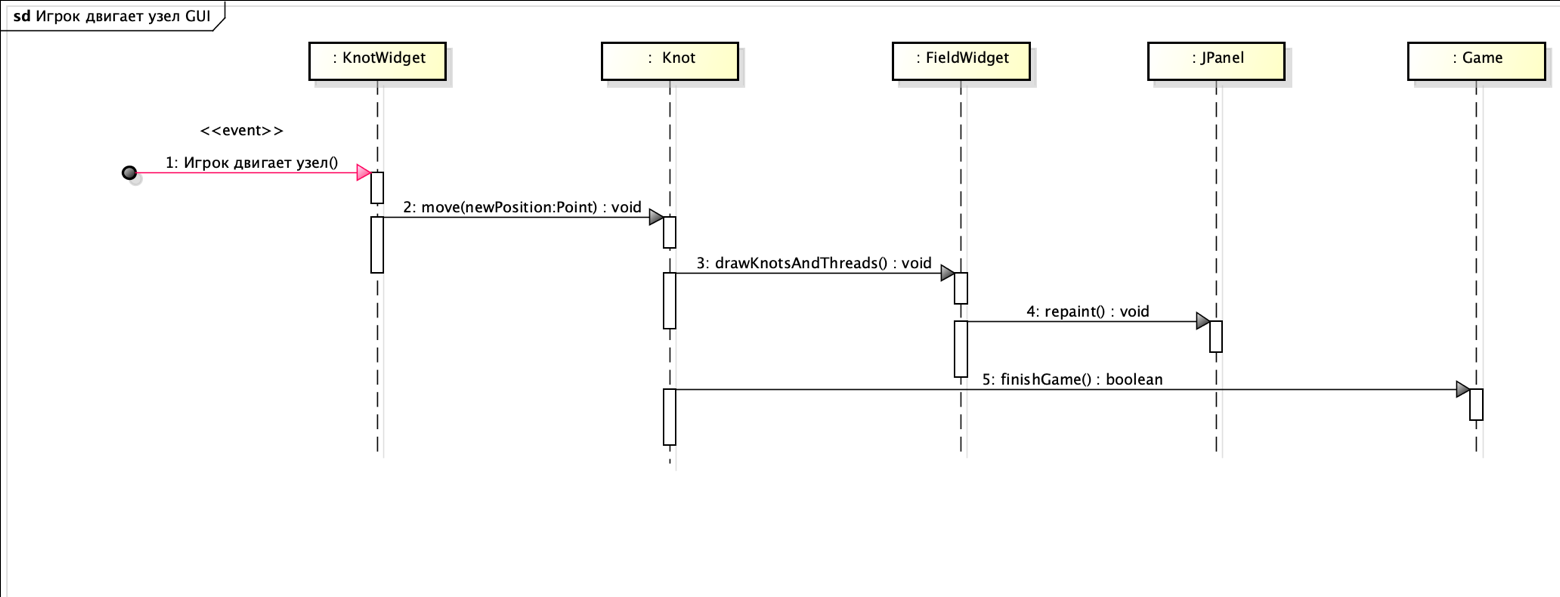
Диаграмма классов представления

### 4.4 Типовые процессы в программе

1. Общий игровой цикл
2. Движение узла

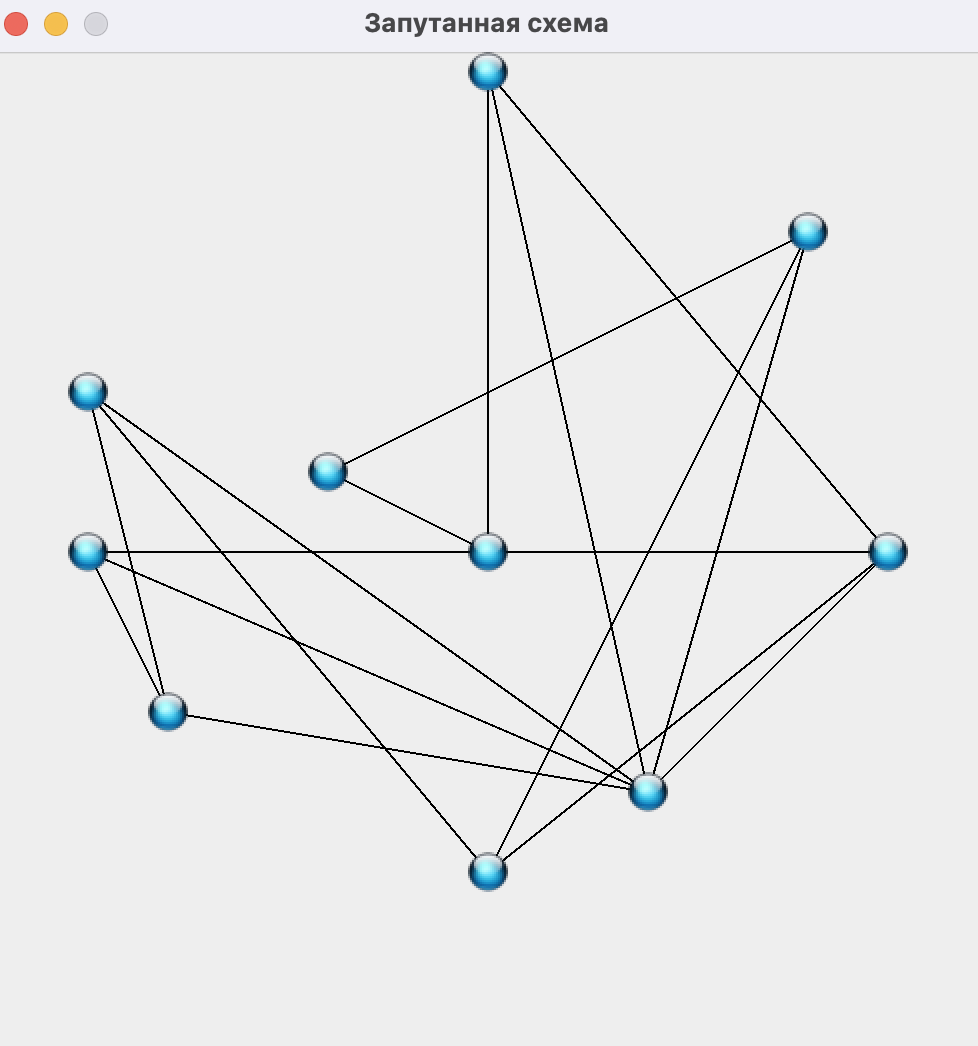
****

1. Определение конца игры
2. Игрок двигает узел

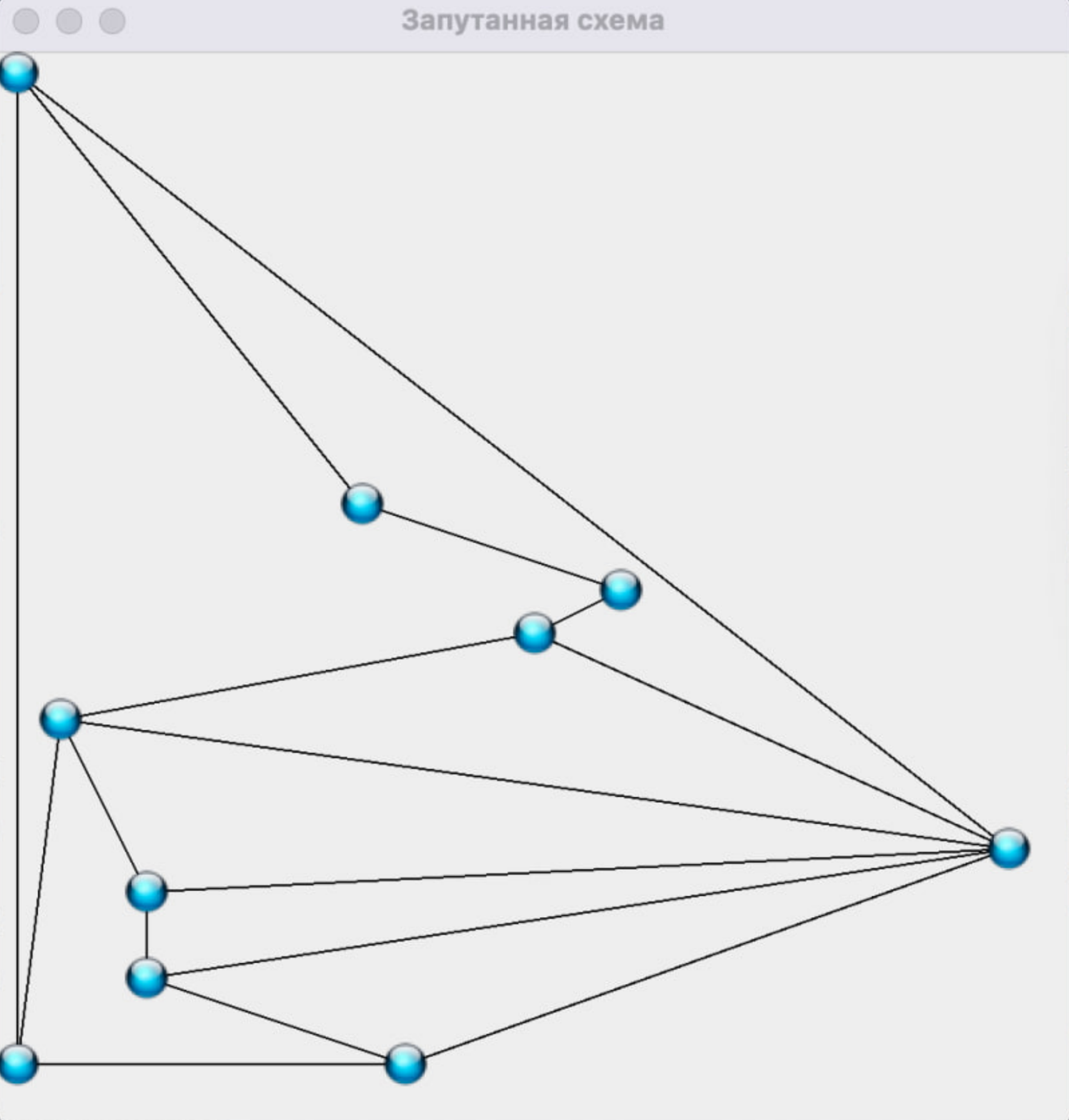


### 4.5 Человеко-машинное взаимодействие

Общий вид главного окна представлен ниже. На нем располагается поле, узлы и нити.



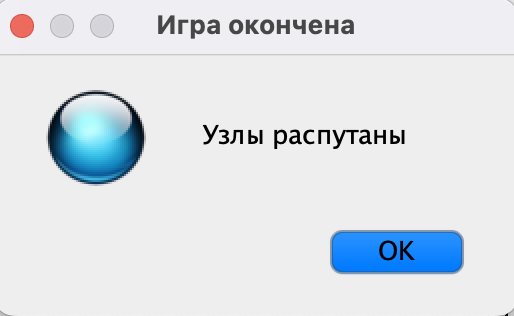
По нажатию на узел, курсор изменяется (HAND\_CURSOR). Если курсор изменился, значит узел доступен для перемещения



Пример выигрышной расстановки

**Окончание игры.**

По окончании игры выводится сообщение: «Узла распутаны»



Окончание игры

### 4.6 Реализация ключевых классов

Класс Узел(Knot)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Нить(Thread)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Уровень(Level)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
public class Knot {  
  
 private Point position;  
  
 public static Object *image*;  
  
 public Point getPosition(){  
 return this.position;  
 }  
  
 */\*\*  
 \*  
 \* @param position Позиция узла  
 \*/* public Knot(Point position){  
 this.position = position;  
 }  
  
 */\*\*  
 \*  
 \* @param newPosition Новая позиция узла, после передвижения  
 \*/* public void move(Point newPosition){  
 this.position = newPosition;  
 fireKnotMoved();  
 }  
  
 protected void setPosition(Point pos){  
 this.position.y = pos.y;  
 this.position.x = pos.x;  
 }  
  
 // ----------------Порождает события--------------------  
  
 private ArrayList knotListener = new ArrayList();  
  
 */\*\*  
 \*  
 \* Присоединяет слушателя  
 \*/* public void addKnotListener(KnotListener l){ knotListener.add(l);}  
  
 */\*\*  
 \*  
 \* Отсоединяет слушателя  
 \*/* public void removeKnotListener(KnotListener l){  
 knotListener.remove(l);  
 }  
  
 */\*\*  
 \*  
 \* Оповещает слушателей о событии  
 \*/* public void fireKnotMoved(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotMoved(event);  
 }  
 }  
  
 public void fireKnotReleased(){  
 KnotEvent event = new KnotEvent(this);  
  
 for (Object listener : knotListener){  
 ((KnotListener)listener).knotReleased(event);  
 }  
 }  
  
  
 */\*\*  
 \*  
 \* Перегрузка оператора  
 \*/* public boolean equals(Object obj) {  
 if (obj == null) {  
 return false;  
 }  
 if (obj.getClass() != this.getClass()) {  
 return false;  
 }  
 final Knot other = (Knot) obj;  
 if ((this.position == null) ? (other.position != null) : !this.position.equals(other.position)) {  
 return false;  
 }  
 return true;  
 }  
}

Класс Клубок(Ball)

package main.model;  
  
import java.util.ArrayList;  
  
public class Ball {  
  
 private int width;  
 private int height;  
  
 */\*\*  
 \* Список узлов  
 \*/* private ArrayList<Knot> knots = new ArrayList<>();  
  
 */\*\*  
 \* Список нитей  
 \*/* private ArrayList<Thread> threads = new ArrayList<>();  
  
 public Ball(int width, int height){  
 if(width <= 0) throw new IllegalArgumentException("Field width must be more than 0");  
 if(height <= 0) throw new IllegalArgumentException("Field height must be more than 0");  
 this.width = width;  
 this.height = height;  
 }  
  
 public int getWidth() {  
 return this.width;  
 }  
 public int getHeight() {  
 return this.height;  
 }  
  
 protected void setKnots(ArrayList<Knot> knots){  
 this.knots = knots;  
 }  
 protected void setThreads(ArrayList<Thread> threads){  
 this.threads = threads;  
 }  
 public ArrayList<Knot> getKnots(){  
 return this.knots;  
 }  
 public ArrayList<Thread> getThreads(){  
 return this.threads;  
 }  
 public void addKnot(Knot k){  
 this.knots.add(k);  
 }  
  
 */\*\*  
 \* Проверить, не осталось ли пересечений нитей  
 \* @return false - пересечений не осталось, true - остались пересечения  
 \*/* public boolean anyThreadsCrossing(){  
 boolean threadsCrossing = false;  
 for(int i = 0; i < threads.size(); i++){  
 for(int j = i+1; j < threads.size(); j++){  
 threadsCrossing = getThreads().get(i).checkTwoThreadsOnCrossing(getThreads().get(j));  
 if(threadsCrossing)  
 return true;  
 }  
 }  
 return false;  
 }  
}

Класс Игра(Game)

package main.model;  
  
import main.events.KnotEvent;  
import main.events.KnotListener;  
  
import javax.swing.\*;  
  
import static javax.swing.JOptionPane.*showMessageDialog*;  
  
public class Game {  
 private Level level;  
  
 private Ball ball;  
  
 public Game(Level level){  
 this.level = level;  
 initGame();  
 }  
  
 */\*\*  
 \* Постоить поле и обстановку на нем  
 \*/* private void build(){  
 ball = level.buildBall();  
 }  
  
 */\*\*  
 \* Начать игру  
 \*/* public void initGame(){  
 build();  
 for(var i: ball.getKnots()){  
 i.addKnotListener(new KnotObserver());  
 }  
 }  
  
 */\*\*  
 \* Проверить закончилась ли игра  
 \* @return результат проверки  
 \*/* public boolean finishGame(){  
 return !ball.anyThreadsCrossing();  
 }  
  
  
 */\*\*  
 \* События  
 \*/* private class KnotObserver implements KnotListener {  
 @Override  
 public void knotMoved(KnotEvent e) {  
   
 }  
  
 @Override  
 public void knotReleased(KnotEvent e){  
 if(finishGame()){  
 *showMessageDialog*(null, "Узлы распутаны", "Игра окончена", JOptionPane.*WARNING\_MESSAGE*, new ImageIcon(getClass().getResource("/knot.png")));  
 }  
 }  
 }  
}

Классы событий(events)

package main.events;  
  
import main.model.Knot;  
  
import java.util.EventObject;  
  
public class KnotEvent extends EventObject {  
  
 private Knot knot;  
 public KnotEvent(Object source) {  
 super(source);  
 }  
  
 public void setKnot(Knot knot) { this.knot = knot; }  
  
 public Knot knot(){ return this.knot; }  
}

package main.events;  
  
import java.util.EventListener;  
  
public interface KnotListener extends EventListener {  
 void knotMoved(KnotEvent e);  
 void knotReleased(KnotEvent e);  
}

Виджеты

private int FRAME\_HEIGHT = 500;  
private int FRAME\_WIDTH = 500;  
private Level level;  
private final int IMAGE\_SIZE = 20;  
private JPanel panel;  
boolean isPressedMouse = false;  
boolean isChosenKnot = false;  
int numOfChosenKnot = -1;  
  
public static void main(String[] args) {  
  
 new Main().setVisible(true);  
}  
  
private Main(){  
 level = new Level();  
 level.buildBall();  
 Game game = new Game(level);  
 setImages();  
 InitPanel();  
 InitFrame();  
}  
  
private void InitPanel(){  
  
 panel = new JPanel(){  
 @Override  
 protected void paintComponent(Graphics g) {  
  
 super.paintComponent(g);  
  
 for(int i = 0; i < level.ball.getThreads().size();i++){  
 g.drawLine( (int) level.ball.getThreads().get(i).getBegin().getPosition().getX()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getBegin().getPosition().getY()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getEnd().getPosition().getX()\*IMAGE\_SIZE+IMAGE\_SIZE/2,  
 (int) level.ball.getThreads().get(i).getEnd().getPosition().getY()\*IMAGE\_SIZE+IMAGE\_SIZE/2);  
 }  
  
 for(int i = 0; i < level.ball.getKnots().size(); i++ ) {  
 g.drawImage((Image) level.ball.getKnots().get(i).*image*, level.ball.getKnots().get(i).getPosition().x \* IMAGE\_SIZE, level.ball.getKnots().get(i).getPosition().y \* IMAGE\_SIZE, this);  
 }  
  
  
 }  
 };  
  
 panel.addMouseListener(new MouseAdapter() {  
 @Override  
 public void mousePressed(MouseEvent e) {  
 // System.out.println("Мышь зажата");  
 isPressedMouse = true;  
 panel.setCursor(Cursor.*getPredefinedCursor*(Cursor.*HAND\_CURSOR*));  
 for(int i = 0; i < level.ball.getKnots().size(); i++){  
 if(level.ball.getKnots().get(i).getPosition().getX() == e.getX()/IMAGE\_SIZE  
 && level.ball.getKnots().get(i).getPosition().getY() == e.getY()/IMAGE\_SIZE){  
 numOfChosenKnot = i;  
 isChosenKnot = true;  
 }  
 }  
 }  
 });  
  
 panel.addMouseMotionListener(new MouseAdapter() {  
 @Override  
 public void mouseDragged(MouseEvent e) {  
 Knot currentKnot = level.ball.getKnots().get(numOfChosenKnot);  
 currentKnot.fireKnotMoved();  
 int x = currentKnot.getPosition().x;  
 int y = currentKnot.getPosition().y;  
 if(isPressedMouse){  
 if(isChosenKnot){  
 if(e.getX() < FRAME\_WIDTH && e.getY() < FRAME\_HEIGHT && e.getY() > 0 && e.getX() > 0){  
 Point newPosition = new Point(e.getX()/IMAGE\_SIZE,e.getY()/IMAGE\_SIZE);  
 currentKnot.move(newPosition);  
 }  
 }  
 }  
 panel.repaint();  
 }  
 });  
  
 panel.addMouseListener(new MouseAdapter() {  
 @Override  
 public void mouseReleased(MouseEvent e){  
 panel.setCursor(Cursor.*getDefaultCursor*());  
 level.ball.getKnots().get(numOfChosenKnot).fireKnotReleased();  
 isPressedMouse = false;  
 numOfChosenKnot = -1;  
 isChosenKnot = false;  
 panel.repaint();  
  
 }  
 });  
  
 panel.setPreferredSize(new Dimension(FRAME\_WIDTH,FRAME\_HEIGHT));  
 add(panel);  
}  
  
private void InitFrame(){  
 pack();  
 setDefaultCloseOperation(WindowConstants.*EXIT\_ON\_CLOSE* );  
 setTitle("Запутанная схема");  
 setLocationRelativeTo(null);  
 setResizable(false);  
 setVisible(true);  
}  
  
private void setImages(){  
 for (int i = 0; i < level.ball.getKnots().size(); i++){  
 Knot.*image* = getImage("/knot2.png");  
 }  
}  
  
private Image getImage(String name){  
 ImageIcon icon = new ImageIcon(getClass().getResource(name));  
 return icon.getImage();  
}

Класс недвигающегося узла(NotMovingKnot)

package main.model;  
  
import java.awt.\*;  
  
public class NotMovingKnot extends Knot{  
 */\*\*  
 \* @param position Позиция узла  
 \*/* public NotMovingKnot(Point position) {  
 super(position);  
 }  
  
 @Override  
 public void move(Point fixPos){  
 fixPos = super.getPosition();  
 setPosition(fixPos);  
 fireKnotMoved();  
 }  
}

Класс узла, двигающегося только по горизонтали(HorizontalMovingKnot)

package main.model;  
  
import java.awt.\*;  
  
public class HorizontalMovingKnot extends Knot{  
  
 */\*\*  
 \* @param position Позиция узла  
 \*/* public HorizontalMovingKnot(Point position) {  
 super(position);  
 }  
 @Override  
 public void move(Point newPosition){  
 newPosition.y = super.getPosition().y;  
 setPosition(newPosition);  
 fireKnotMoved();  
 }  
}

### 4.7 Реализация ключевых тестовых случаев

Тесты для класса Узел

package tests;  
  
import main.model.Knot;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
  
public class KnotsTest {  
  
 @Test  
 public void constructorTest(){  
 Knot actual = new Knot(new Point(1,1));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected, actual.getPosition());  
 }  
  
 @Test  
 public void moveTest(){  
 Knot actual = new Knot(new Point(1,1));  
 actual.move(new Point(2,2));  
 Point expected = new Point(2,2);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
}

Тесты для класса Клубок

package tests;  
  
import main.model.Ball;  
import org.junit.Assert;  
import org.junit.Test;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertThrows*;  
  
  
public class BallTests {  
  
 // Неверный параметр ширины передан в конструктор  
 @Test  
 public void constructorTestingBadWidth()  
 {  
 *assertThrows*(IllegalArgumentException.class, () -> new Ball(0, 10));  
 }  
  
 // Неверный параметр высоты передан в конструктор  
 @Test  
 public void constructorTestingBadHeight()  
 {  
 *assertThrows*(IllegalArgumentException.class, () -> new Ball(10, 0));  
 }  
  
 @Test  
 public void constructorTestingGood(){  
 Ball actual = new Ball(10,10);  
 int expectedWidth = 10;  
 int expectedHeight = 10;  
 Assert.*assertEquals*(expectedHeight, actual.getHeight());  
 Assert.*assertEquals*(expectedWidth, actual.getWidth());  
 }  
  
 @Test  
 public void threadsNotCrossing(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(6,2));  
 Knot knot4 = new Knot(new Point(10,6));  
 Knot knot5 = new Knot(new Point(-2,-4));  
 Knot knot6 = new Knot(new Point(-4,-2));  
  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
 Thread thread3 = new Thread(knot5,knot6);  
 Thread thread4 = new Thread(knot1,knot3);  
  
  
 ArrayList<Thread> threads= new ArrayList<>();  
 threads.add(thread1);  
 threads.add(thread2);  
 threads.add(thread3);  
 threads.add(thread4);  
  
 Ball ball = new Ball(10,10);  
 ball.setThreads(threads);  
 Assert.assertEquals(false, ball.anyThreadsCrossing());  
 }  
  
 @Test  
 public void threadsCrossing(){  
 Level level = new Level();  
 level = new Level();  
 level.buildBall();  
 Assert.assertEquals(true, level.ball.anyThreadsCrossing());  
 }  
  
  
}

Тесты для класса Уровень

package tests;  
  
import main.model.Ball;  
import main.model.Level;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
public class LevelTests {  
  
 @Test  
 public void buildBallTestingOnSizes(){  
 Level level = new Level();  
 Ball ball = level.buildBall();  
 int expectedWidth = 10;  
 int expectedHeight = 10;  
 Assert.*assertEquals*(expectedHeight, ball.getHeight());  
 Assert.*assertEquals*(expectedWidth, ball.getWidth());  
  
 }  
  
 @Test  
 public void buildBallTestingOnCountKnotsAndThreads(){  
 Level level = new Level();  
 Ball ball = level.buildBall();  
 int expectedKnotsCount = 6;  
 int expectedThreadsCount = 9;  
 Assert.*assertEquals*(expectedKnotsCount, ball.getKnots().size());  
 Assert.*assertEquals*(expectedThreadsCount, ball.getThreads().size());  
 }  
}

Тесты для класса Нить

package tests;  
  
import main.model.Knot;  
import main.model.Thread;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
  
import static org.junit.Assert.*assertThrows*;  
  
public class ThreadTests {  
  
 @Test  
 public void constructorTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(2,2));  
 Thread actual = new Thread(knot1,knot2);  
  
 Knot expKnot1 = new Knot(new Point(1,1));  
 Knot expKnot2 = new Knot(new Point(2,2));  
  
 Assert.*assertEquals*(expKnot1, actual.getBegin());  
 Assert.*assertEquals*(expKnot2, actual.getEnd());  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithPositiveNumbers(){  
 Knot knot1 = new Knot(new Point(2,3));  
 Knot knot2 = new Knot(new Point(6,4));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = 0.25;  
 double expb = 2.5;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithNegativeNumbers(){  
 Knot knot1 = new Knot(new Point(-2,3));  
 Knot knot2 = new Knot(new Point(6,-4));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = -0.875;  
 double expb = 1.25;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithZeroes(){  
 Knot knot1 = new Knot(new Point(0,3));  
 Knot knot2 = new Knot(new Point(1,0));  
 double k;  
 double b;  
 Thread thread = new Thread(knot1,knot2);  
 thread.getEquationOfStraight();  
 k = thread.getK();  
 b = thread.getB();  
 double expk = -3;  
 double expb = 3;  
 Assert.*assertEquals*(expk,k, 0.0);  
 Assert.*assertEquals*(expb,b, 0.0);  
 }  
  
 @Test  
 public void checkingEquationOfLinesWithSimilarCoordinates(){  
 Knot knot1 = new Knot(new Point(0,0));  
 Knot knot2 = new Knot(new Point(0,0));  
 Thread thread = new Thread(knot1,knot2);  
  
 *assertThrows*(IllegalArgumentException.class, () -> thread.getEquationOfStraight());  
 }  
  
 @Test  
 public void twoThreadsCrossingTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(2,3));  
 Knot knot4 = new Knot(new Point(4,2));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(true,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsNotCrossingTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(6,2));  
 Knot knot4 = new Knot(new Point(10,6));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(false,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsNotCrossingWithNegativeNumbersTest(){  
 Knot knot1 = new Knot(new Point(1,1));  
 Knot knot2 = new Knot(new Point(3,3));  
 Knot knot3 = new Knot(new Point(-2,-3));  
 Knot knot4 = new Knot(new Point(-4,-8));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
  
 Assert.assertEquals(false,thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
 @Test  
 public void twoThreadsCrossingWithNegativeNumbersTest(){  
 Knot knot1 = new Knot(new Point(-6,-2));  
 Knot knot2 = new Knot(new Point(4,-10));  
 Knot knot3 = new Knot(new Point(-2,-3));  
 Knot knot4 = new Knot(new Point(-4,-8));  
  
 Thread thread1 = new Thread(knot1,knot2);  
 Thread thread2 = new Thread(knot3,knot4);  
  
 Assert.assertEquals(true, thread1.checkTwoThreadsOnCrossing(thread2));  
 }  
  
  
}

Тесты для класса Игра

package tests;  
  
import main.model.Ball;  
import main.events.KnotEvent;  
import main.events.KnotListener;  
import main.model.Level;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
import java.util.ArrayList;  
  
import static org.junit.jupiter.api.Assertions.*assertTrue*;  
  
public class GameTests {  
  
 private enum EVENT {*KNOTMOVED*}  
 private ArrayList<EVENT> knots\_events = new ArrayList<>();  
  
 private class KnotObserver implements KnotListener{  
 @Override  
 public void knotMoved(KnotEvent e) {  
 knots\_events.add(EVENT.*KNOTMOVED*);  
 }  
  
 @Override  
 public void knotReleased(KnotEvent e){}  
 }  
  
 @Test  
 public void knotEvent(){  
 Level level = new Level();  
  
 Ball ball = level.buildBall();  
  
 for(var i: ball.getKnots()){  
 i.addKnotListener(new KnotObserver());  
 }  
  
 ball.getKnots().get(0).move(new Point(4,4));  
 *assertTrue*(!knots\_events.isEmpty());  
 }  
  
}

Тесты для класса Недвигающийся узел

package tests;  
  
import main.model.NotMovingKnot;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
public class NotMovingKnotsTest {  
  
 @Test  
 public void moveOnXAndYTest(){  
 NotMovingKnot actual = new NotMovingKnot(new Point(1,1));  
 actual.move(new Point(2,2));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void moveOnXTest(){  
 NotMovingKnot actual = new NotMovingKnot(new Point(1,1));  
 actual.move(new Point(2,1));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void moveOnYTest(){  
 NotMovingKnot actual = new NotMovingKnot(new Point(1,1));  
 actual.move(new Point(1,2));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void notMoveTest(){  
 NotMovingKnot actual = new NotMovingKnot(new Point(1,1));  
 actual.move(new Point(1,1));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void negativeCoordsMoveTest(){  
 NotMovingKnot actual = new NotMovingKnot(new Point(-2,-7));  
 actual.move(new Point(-7,-2));  
 Point expected = new Point(-2,-7);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
}

Тесты для класса Узел, двигающийся по горизонтали

package tests;  
  
import main.model.HorizontalMovingKnot;  
import org.junit.Assert;  
import org.junit.jupiter.api.Test;  
  
import java.awt.\*;  
public class HorizontalMovingKnotsTest {  
  
 @Test  
 public void moveOnXAndYTest(){  
 HorizontalMovingKnot actual = new HorizontalMovingKnot(new Point(1,1));  
 actual.move(new Point(2,2));  
 Point expected = new Point(2,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void moveOnXTest(){  
 HorizontalMovingKnot actual = new HorizontalMovingKnot(new Point(1,1));  
 actual.move(new Point(2,1));  
 Point expected = new Point(2,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void moveOnYTest(){  
 HorizontalMovingKnot actual = new HorizontalMovingKnot(new Point(1,1));  
 actual.move(new Point(1,2));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void notMoveTest(){  
 HorizontalMovingKnot actual = new HorizontalMovingKnot(new Point(1,1));  
 actual.move(new Point(1,1));  
 Point expected = new Point(1,1);  
 Assert.*assertEquals*(expected,actual.getPosition());  
 }  
  
 @Test  
 public void negativeCoordsMoveTest(){  
 HorizontalMovingKnot actual = new HorizontalMovingKnot(new Point(-2,-7));  
 actual.move(new Point(-7,-2));  
 Point expected = new Point(-7,-7);  
 Assert.*assertEquals*(expected,actual.getPosition());

}

# 5 Список использованной литературы и других источников

1. Логинова, Ф.С. Объектно-ориентированные методы программирования. [Электронный ресурс] : учеб. пособие — Электрон. дан. — СПб. : ИЭО СПбУТУиЭ, 2012. — 208 с. — Режим доступа: http://e.lanbook.com/book/64040
2. Васильев, А.Н. Самоучитель Java с примерами и программами. [Электронный ресурс] : самоучитель — Электрон. дан. — СПб. : Наука и Техника, 2016. — 368 с. — Режим доступа: http://e.lanbook.com/book/90231
3. Программирование на языке Java. Конспект лекций. [Электронный ресурс] : учеб. пособие / А.В. Гаврилов [и др.]. — Электрон. дан. — СПб.: НИУ ИТМО, 2015. — 126 с. — Режим доступа: http://e.lanbook.com/book/91488