Министерство науки и высшего образования Российской Федерации Пензенский государственный университет

Кафедра «Вычислительная техника»

Выполнил:

студент группы 22ВВВ1

Ганин И.Р.

Приняли:

Юрова О.В.

Пенза 2025

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

по курсу «Программирование на языке Java»

на тему «Разработка многомодульного приложения на языке Java»

# Содержание

[Содержание 4](#_Toc202136846)

[Введение 5](#_Toc202136847)

[1. Постановка задачи 6](#_Toc202136848)

[2. Теоретическая часть задания 7](#_Toc202136849)

[3. Выбор решения 8](#_Toc202136850)

[4. Описание протокола 9](#_Toc202136851)

[5. Описание программы 11](#_Toc202136852)

[6. Руководство пользователя 17](#_Toc202136853)

[Заключение 23](#_Toc202136854)

[Список используемых источников 24](#_Toc202136855)

[Приложение A. Листинг программы 25](#_Toc202136856)

[Приложение A.1. Файл «Card.java» 25](#_Toc202136857)

[Приложение A.2. Файл «Client.java» 27](#_Toc202136858)

[Приложение A.3. Файл «Server.java» 36](#_Toc202136859)

[Приложение A.4. Файл «GameLogic.java» 42](#_Toc202136860)

[Приложение Б. UML-диаграммы 48](#_Toc202136861)

[Приложение Б.1. UML – диаграмма вариантов использования. 48](#_Toc202136862)

[Приложение Б.2. UML – диаграмма классов. 49](#_Toc202136863)

[Приложение Б.3. UML – диаграмма деятельности. 50](#_Toc202136864)

[Приложение Б.4. UML – диаграмма развёртывания. 51](#_Toc202136865)

[Приложение Б.5. UML – диаграмма последовательности. 52](#_Toc202136866)

# Введение

Современные технологии информационного обмена играют ключевую роль в повседневной жизни человека. С развитием интернета и локальных сетей, системы передачи данных стали неотъемлемой частью общения, как в личной, так и в профессиональной сфере.

Целью данной курсовой работы является разработка игрового приложения клиент-серверной архитектуры «Дурак».

В рамках работы предполагается изучить основные аспекты построения сетевых приложений, взаимодействующих по модели "клиент-сервер". Будут рассмотрены вопросы обработки запросов, передачи данных через сеть.

Для реализации проекта используется язык программирования Java.

# Постановка задачи

В соответствии с заданием необходимо разработать систему программ клиент-серверной архитектуры, позволяющую пользователю получать информацию с сервера и взаимодействовать с ней.

Требуется разработать протокол обмена данными, который обеспечит передачу информации между клиентами и сервером.

Функции клиента:

* Графический оконный интерфейс пользователя
* Взаимодействие с сервером

Функции сервера:

* Организация взаимодействия с программой-клиентом
* Управление трафиком данных

Приложение должно обладать графическим интерфейсом и использовать следующие технологии:

1) Java Collections Framework

2) Механизм обработки исключительных ситуаций

3) Java Stream API

4) Java Multithreading

5) Сетевое взаимодействие.

Операционная система – Windows.

Язык программирования – Java.

Среда программирования: NetBeans.

# Теоретическая часть задания

2.1 Клиент-серверная архитектура

Клиент-серверная архитектура является одной из наиболее популярных моделей организации взаимодействия между компонентами программных систем. В данной архитектуре система состоит из двух основных частей: клиента и сервера. Клиент выступает инициатором взаимодействия, отправляя запросы серверу, который, в свою очередь, обрабатывает запросы и возвращает ответы.

Главные преимущества клиент-серверной архитектуры:

* Централизованное хранение данных, что упрощает их управление и защиту.
* Возможность масштабирования системы за счет увеличения числа клиентов или серверов.
* Гибкость и возможность расширения функциональности.

2.2 Протоколы передачи данных

Для организации взаимодействия между клиентами и серверами используются сетевые протоколы — набор правил, определяющих формат и порядок передачи данных.

Наиболее распространенные протоколы для передачи данных:

* TCP (Transmission Control Protocol): гарантирует доставку данных, контроль целостности и последовательности пакетов. Используется для создания надежных соединений.
* UDP (User Datagram Protocol): обеспечивает более быструю передачу данных за счет отказа от проверки целостности и подтверждения доставки, используется в задачах, где важна скорость.

Для реализации системы обмена сообщениями на основе клиент-серверной архитектуры часто используется протокол TCP, так как он обеспечивает надежность передачи данных.

# Выбор решения

Для разработки графического интерфейса программы используется Java.Swing. Эта библиотека предоставляет инструменты для создания текстовых полей, кнопок, списков и других элементов управления. Она позволяет реализовать нативный и легковесный интерфейс, адаптированный под ОС Windows.

Для организации сетевого взаимодействия используется библиотека java.net, в частности классы Socket и ServerSocket. Обмен данными между клиентами и сервером осуществляется по протоколу TCP/IP, который обеспечивает надёжную передачу данных с подтверждением доставки и возможностью повторной отправки при сбоях.

С помощью TCP клиент инициирует подключение к серверу, создавая объект Socket, а сервер в свою очередь принимает входящие соединения через ServerSocket. После установления соединения обе стороны могут передавать данные посредством потоков ввода и вывода, обеспечивая стабильную и последовательную коммуникацию в рамках сетевой игры.

# Описание протокола

Разработанная программа основывается на нижеописанном протоколе и состоит из клиента и сервера.

Все команды передаются как текстовые строки через PrintWriter/BufferedReader.

Формат — имя команды и (при необходимости) параметры, разделённые двоеточием.

Сервер при подключении игрока выдаёт ему ID в виде строки. Клиент читает эту строку и сохраняет её.

**Список команд**

* **TRUMP:{масть}** — Сервер сообщает игрокам козырную масть текущей игры.
* **GAME\_STARTED:{число\_игроков}** — Игра началась с указанным количеством игроков.
* **PLAYER\_ID:{id}** — Сервер сообщает клиенту его идентификатор игрока.
* **YOUR\_HAND:{карта1},{карта2},...** — Передача текущей руки игроку. Карты перечислены через запятую, каждая в формате "ранг масть" (например, 8 SPADE).
* **TURN:{игрок}:{фаза}** — Информация о текущем ходе: кто ходит и какая фаза хода (attack, defense, adding).
* **PLAYER\_THREW:{игрок}:{карты}** — Игрок совершил ход, выложив карты на стол (атакующие карты).
* **PLAYER\_COVERED:{игрок}:{карта}** — Игрок покрыл карту.
* **PLAYER\_ADDED:{игрок}:{карты}** — Игрок подкинул карты.
* **ROUND\_ENDED** — Раунд закончен, происходит обновление состояния.
* **PLAYER\_TAKE:{игрок}** — Игрок взял карты со стола.
* **TABLE:{атакующие\_карты}|{защитные\_карты}** — Обновление состояния столa, разделение символом ‘|’ между атакующими и защитными картами.
* **PHASE:ADDING\_CARDS** — Фаза добавления карт началась.
* **GAME\_OVER:{причина}** — Игра завершена. Причина окончания передается текстом (например, "Игрок 0 выиграл", "Ничья").
* **ERROR:{сообщение}** — Ошибка или недопустимая операция, сервер сообщает клиенту о проблеме.

Диаграмма последовательности представлена на рисунке 1.

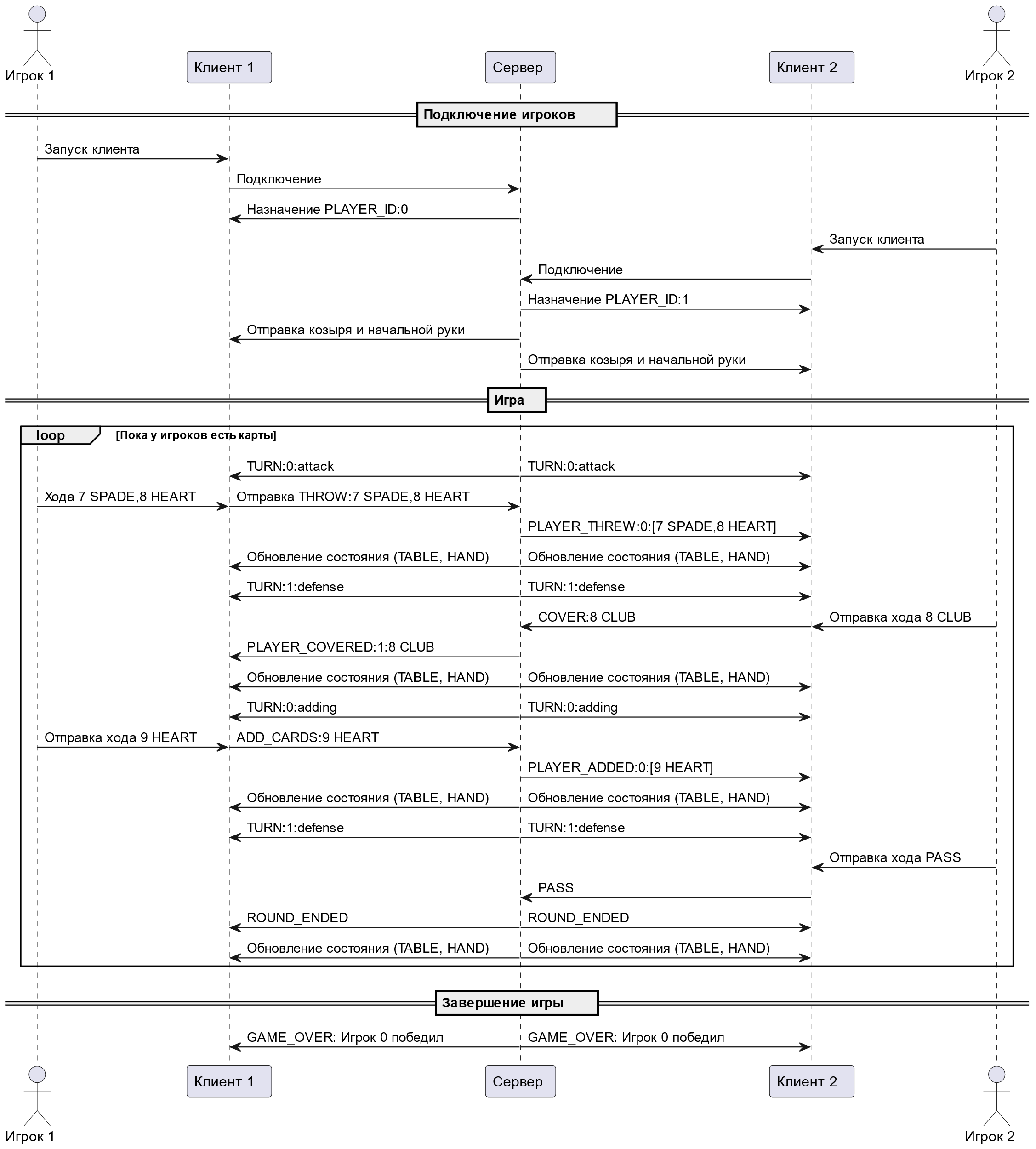


Рисунок 1 - Диаграмма последовательности

# Описание программы

**Файл GameLogic.java** (пакет fool)

* Перечисление RoundPhase

Определяет фазы раунда: ATTACKING, DEFENDING, ADDING\_CARDS, END\_ROUND.

* Конструктор GameLogic()

Инициализирует колоду, раздаёт по 6 карт каждому из двух игроков, выбирает козырь, устанавливает текущих атакующего и защитника.

* dealInitialCards()

Раздаёт игрокам стартовые карты из колоды.

* throwCards(List<Card> cards)

Вызывается, когда атакующий выкладывает карты на стол (доступно в фазах ATTACKING или ADDING\_CARDS).

* coverCard(Card attackCard, Card defenseCard)

Вызывается, когда защитник пытается покрыть атакующую карту, проверяет правила и козыри.

* addCards(List<Card> cards)

Вызывается, когда атакующий добавляет карты на стол, если это разрешено (фаза ADDING\_CARDS).

* takeCards()

Вызывается, когда защитник забирает все карты со стола, если не смог покрыть.

* endRound()

Завершает раунд, игроки добирают карты, обновляет активных игроков и фазу.

* isGameOver()

Проверяет, закончилась ли игра (нет карт у игроков и в колоде).

* Вспомогательные методы

refillHands(), canCover(), allCardsSameRank(), switchActivePlayerToOpponent() — логика добора карт, проверки покрытий, смены игроков и др.

**Файл Card.java (пакет fool)**

* Перечисления Suit и Rank

Suit: CLUB, DIAMOND, HEART, SPADE — масти карт.

Rank: SIX, SEVEN, ..., ACE — ранги карт с символами для отображения.

* Конструктор Card(Suit suit, Rank rank)

Создаёт карту с мастью, рангом и загружает иконку изображения.

* getSuit(), getRank()

Возвращают масть и ранг карты.

* getIcon()

Возвращает иконку для визуализации карты.

* toString()

Возвращает текстовое представление карты, например, “7 SPADE”.

**Файл Server.java (пакет fool)**

* main(String[] args)

Запускает сервер, инициализирует колоду, запускает цикл прослушивания входящих соединений. При подключении клиента создаёт новый поток ClientHandler и добавляет его в список клиентов. При достижении двух игроков запускает игру.

* initDeck()

Инициализирует и перемешивает полную колоду карт (4 масти, 9 рангов).

* startGame()

Создаёт новый объект GameLogic с колодой, распределяет игрокам ID, рассылает сообщения о начале игры и козыре, раздаёт стартовые карты.

* broadcast(String message)

Отправляет текстовое сообщение всем подключённым клиентам.

* sendHand(int playerId)

Отправляет конкретному игроку его текущую руку карт в текстовом формате.

* updateTable()

Формирует и рассылает сообщение с текущими картами на столе.

* processMove(int playerId, String move)

Метод для обработки ходов от игроков. Разбирает команду из строки, проверяет корректность хода по фазе и очереди, обновляет состояние игры через gameLogic и рассылает обновления другим клиентам.

* updateAllHands()

Обновляет и рассылает всем игрокам их руки карт.

* checkGameOver()

Проверяет условие окончания игры (отсутствие карт у игроков и в колоде), рассылает итоговое сообщение и закрывает соединения.

**Вложенный класс ClientHandler (в Server.java)**

* Поля

Socket socket — сокет клиента.

PrintWriter out — поток для отправки сообщений клиенту.

BufferedReader in — поток для чтения сообщений от клиента.

int playerId — идентификатор игрока.

* Конструктор ClientHandler(Socket socket)

Инициализирует потоки ввода-вывода для клиента.

* setPlayerId(int id) / getPlayerId()

Сеттер и геттер ID игрока.

* closeConnection()

Закрывает сокет клиента.

* run()

Основной цикл потока — читает сообщения от клиента, передаёт их в Server.processMove(). При отключении клиента вызывает удаление из списка и закрытие сокета.

* sendMessage(String msg)

Отправляет текстовое сообщение клиенту.

**Файл Client.java (пакет fool)**

* Поля

Socket socket — сокет для подключения к серверу.

BufferedReader in — поток для чтения сообщений с сервера.

PrintWriter out — поток для отправки сообщений на сервер.

int playerId — уникальный идентификатор игрока, полученный от сервера.

List<Card> hand — текущая рука карт игрока.

String trumpSuit — масть козыря.

GameLogic clientGameLogic — локальная копия логики игры для отображения и валидации ходов.

boolean running — флаг для управления основным циклом приема сообщений.

* Конструктор Client(String serverAddress, int port)

Создаёт подключение к серверу, инициализирует потоки ввода/вывода. Запускает поток для обработки входящих сообщений от сервера.

* startListening()

Запускает отдельный поток, который в бесконечном цикле читает сообщения с сервера, обрабатывает команды.

* sendMessage(String msg)

Отправляет строковое сообщение серверу.

* handleServerMessage(String message)

Обрабатывает входящие от сервера сообщения.

* makeMove(String command, List<Card> cards)

Формирует и отправляет ход на сервер в соответствующем формате, например THROW:7 SPADE,8 HEART.

* disconnect()

Закрывает соединение и потоки.

На рисунке 2 представлена диаграмма развертывания. На рисунке 3 представлена диаграмма деятельности.

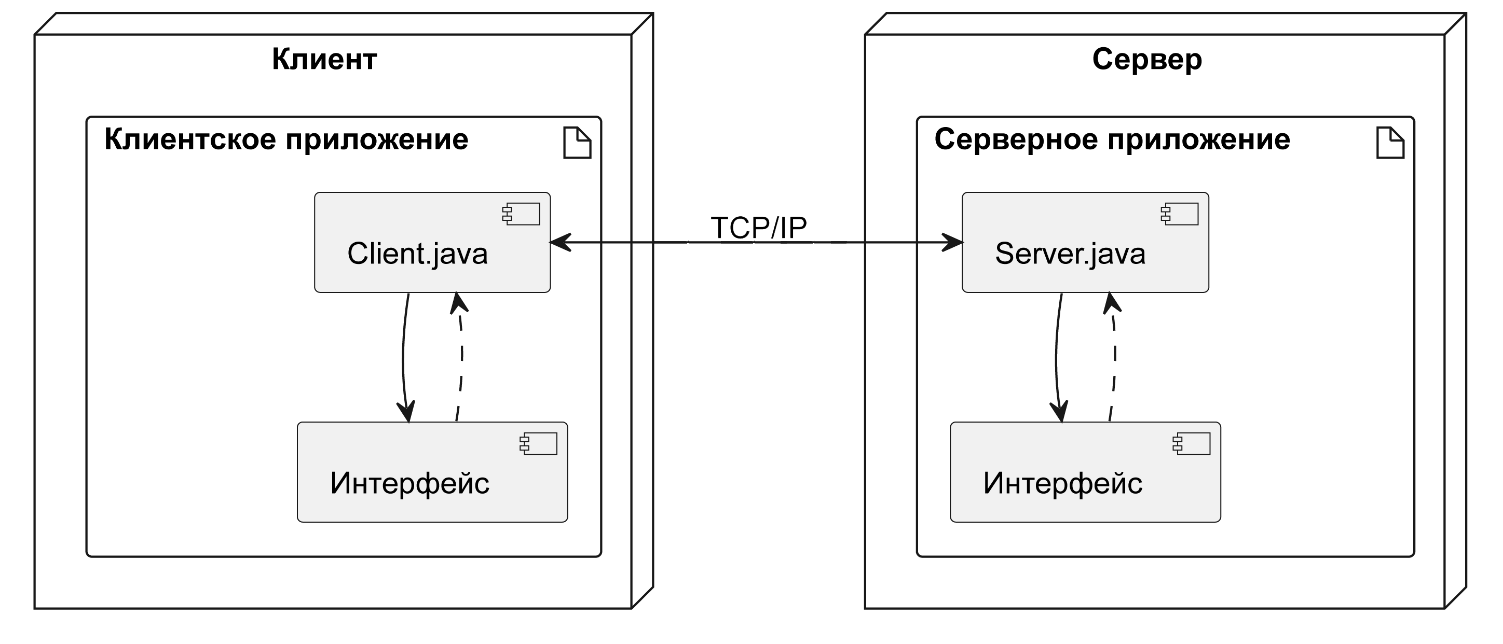


Рисунок 2 - Диаграмма развертывания

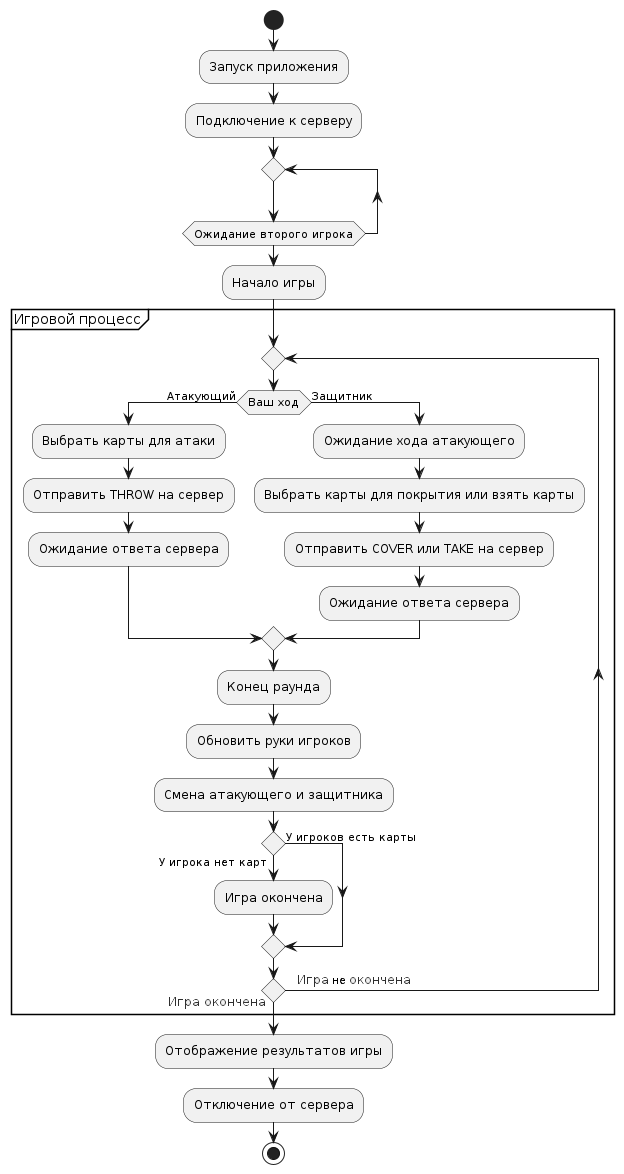
****

Рисунок 3 - Диаграмма деятельности

# Руководство пользователя

При запуске приложения происходит подключение к серверу и появляется главное окно. В нём пользователь видит свои карты и сообщения от сервера.

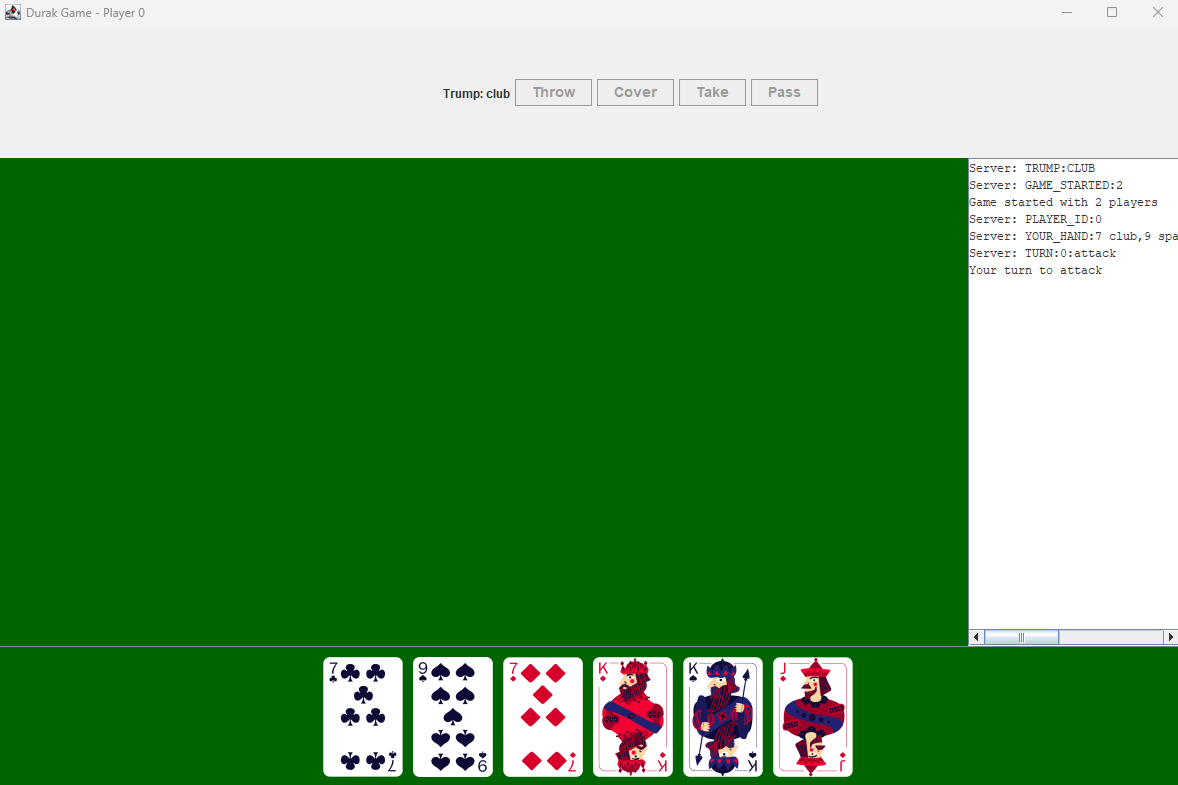


Рисунок 4 - Главное окно

Пользователь может выбрать карты для хода, после чего становится активной кнопка ‘Throw’.

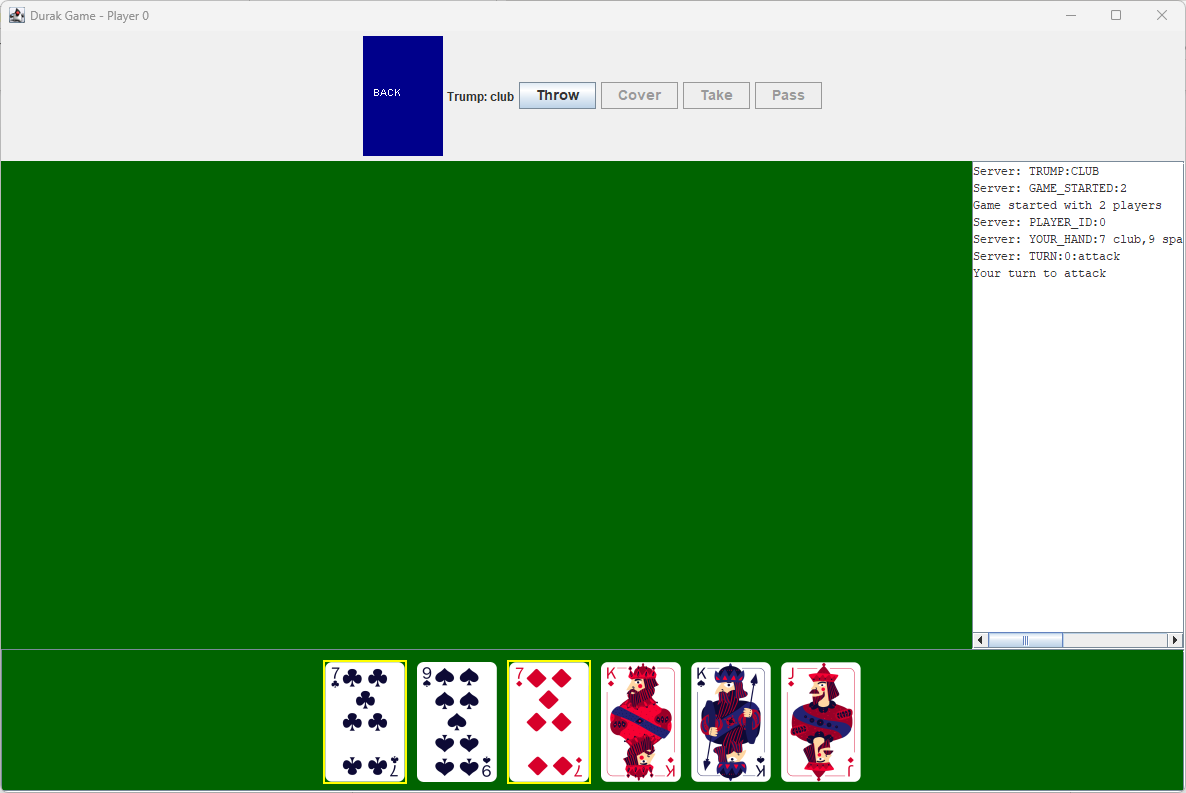


Рисунок 5 – Выбор карты

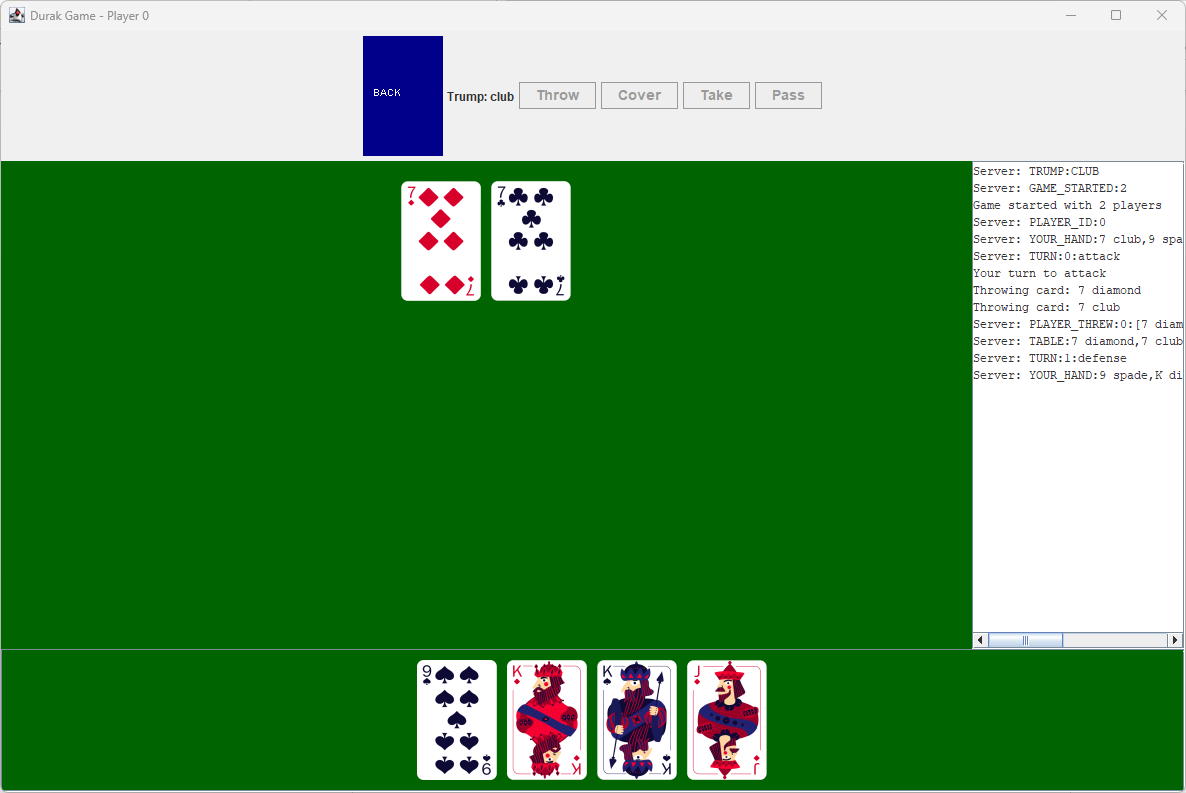


Рисунок 6 – Карты на столе

Второй игрок может взять карты или защититься от их.



Рисунок 7 – Игрок бъёт карты

Как только второй игрок побил карты, первый может докинуть карты или перейти к следующему раунду.



Рисунок 8 - Игрок докидывает карту

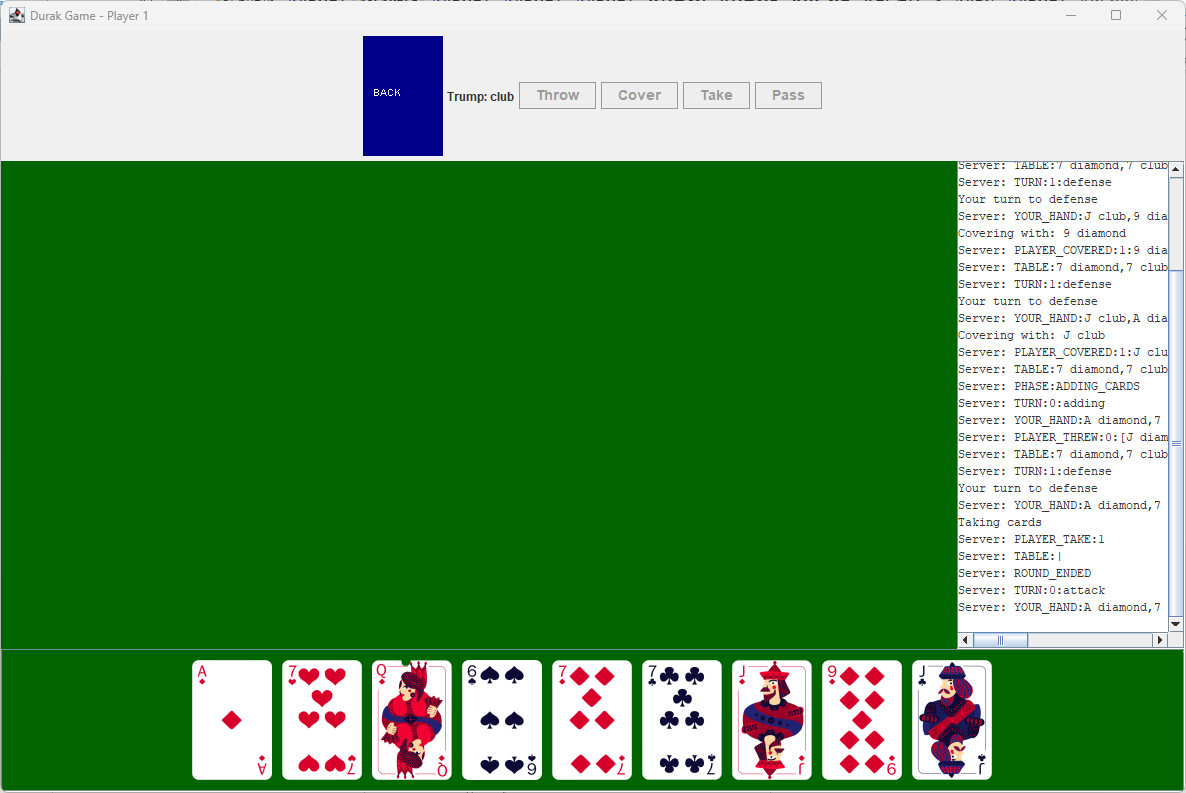


Рисунок 9 – Второй игрок берёт карты

В конце раунда игроку, у которого меньше 6 карт, выдаются карты из колоды

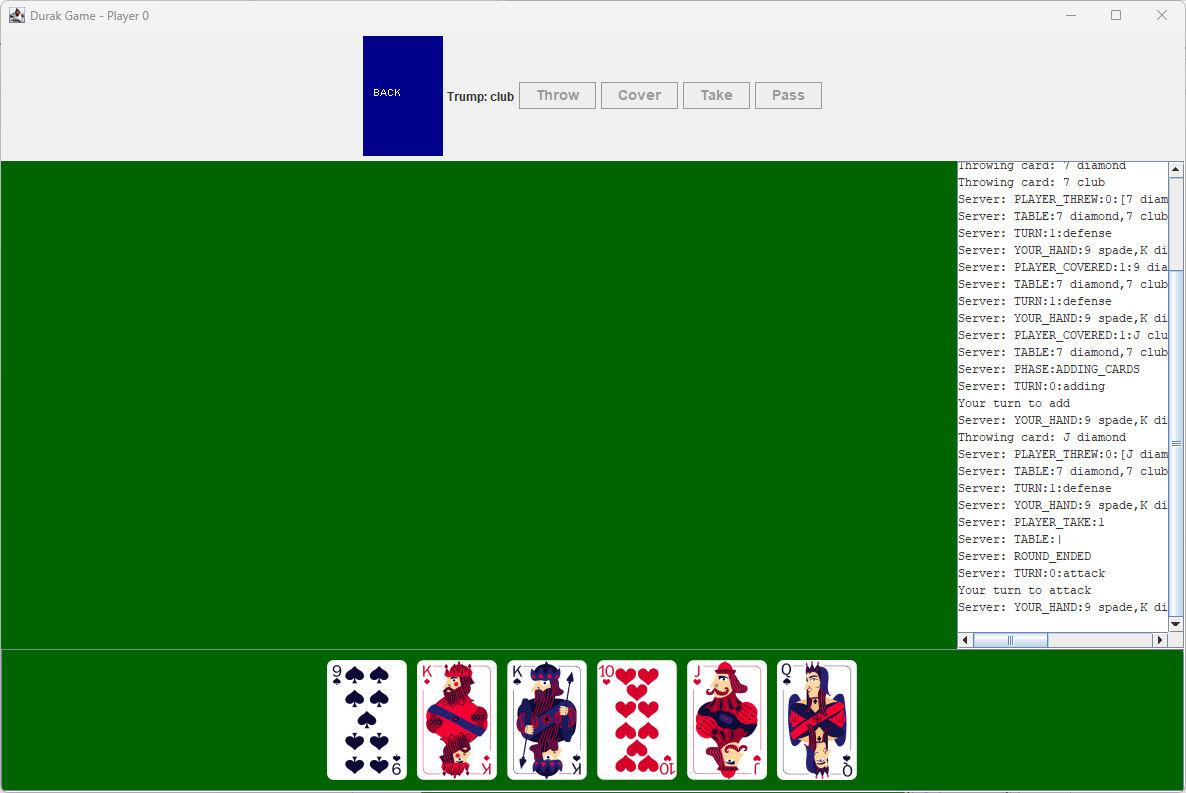


Рисунок 10 – Игрок получил карты из колоды

Игра продолжается до тех пор, пока у одного из игроков не закончатся карты.

Если у обоих игроков заканчиваются карты, то игра тоже заканчивается

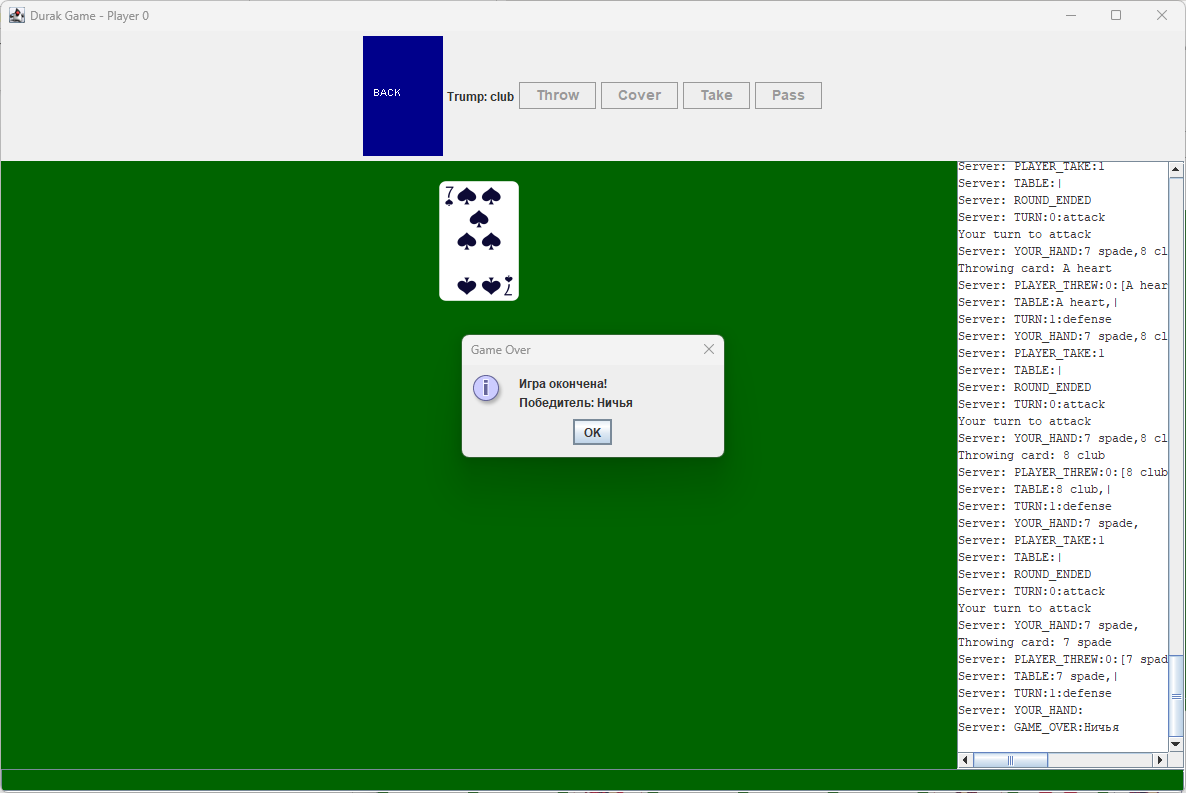


Рисунок 11 - Ничья

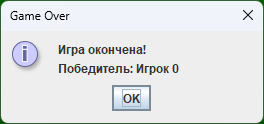


Рисунок 12 – Игрок побеждает

На рисунке 13 представлена диаграмма вариантов использования.

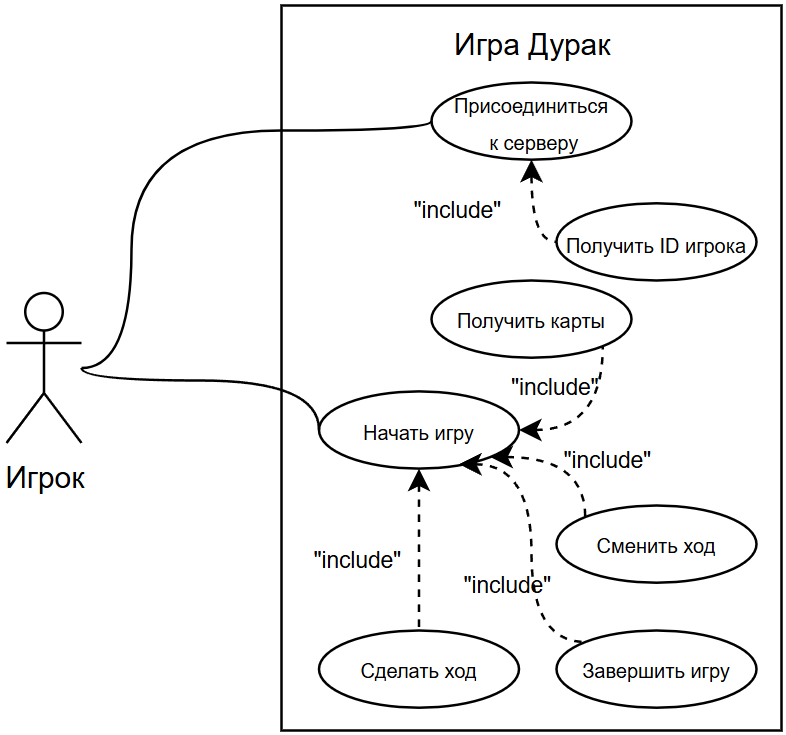


Рисунок 13 - Диаграмма вариантов использования

# Заключение

При выполнении данного курсового проекта были изучены принципы работы протоколов TCP/IP, приобретены практические навыки программирования на языке Java, получены навыки разработки сетевых приложений.

В результате выполнения курсовой работы была реализована клиент‑серверная игра «Дурак». В процессе разработки были изучены основы сетевого взаимодействия на сокетах Java (Socket/ServerSocket), были получены практические навыки многопоточности, был разработан собственный протокол передачи данных, который обеспечивает обмен именами игроков, синхронизацию состояний стола и передачу результатов.

Программа написана на языке Java с использованием среды программирования NetBeans.

# Список используемых источников

1. **Мухамедзянов Р. Р.** " Java. Серверные приложения."
2. stackoverflow.com [Электронный ресурс]. Режим доступа: https://stackoverflow.com// (Дата обращения 25.03.2025).
3. "Head First Java, Изучаем Java", Кэти Сьерра, Берт Бэйтс
4. "Java для чайников", Барри Бёрд
5. Шилдт Г. «Java. Полное руководство», 13‑е изд. IT‑Books, 2019.

# Приложение A. Листинг программы

## Приложение A.1. Файл «Card.java»

package fool;

import javax.swing.\*;

import java.util.Map;

import java.util.Objects;

public final class Card {

public enum Suit {

CLUB, DIAMOND, HEART, SPADE

}

public enum Rank {

SIX("6"), SEVEN("7"), EIGHT("8"), NINE("9"),

TEN("10"), JACK("J"), QUEEN("Q"), KING("K"), ACE("A");

private final String symbol;

Rank(String symbol) {

this.symbol = symbol;

}

public String getSymbol() {

return symbol;

}

public static Rank fromString(String s) {

return switch (s.toUpperCase()) {

case "6", "SIX" ->

SIX;

case "7", "SEVEN" ->

SEVEN;

case "8", "EIGHT" ->

EIGHT;

case "9", "NINE" ->

NINE;

case "10", "TEN" ->

TEN;

case "J", "JACK" ->

JACK;

case "Q", "QUEEN" ->

QUEEN;

case "K", "KING" ->

KING;

case "A", "ACE" ->

ACE;

default ->

throw new IllegalArgumentException("No rank with symbol " + s);

};

}

}

private static final Map<Rank, Integer> RANK\_VALUES = Map.of(

Rank.SIX, 6, Rank.SEVEN, 7, Rank.EIGHT, 8, Rank.NINE, 9,

Rank.TEN, 10, Rank.JACK, 11, Rank.QUEEN, 12, Rank.KING, 13,

Rank.ACE, 14

);

private final Suit suit;

private final Rank rank;

private final ImageIcon icon;

public Card(Suit suit, Rank rank, ImageIcon icon) {

this.suit = Objects.requireNonNull(suit);

this.rank = Objects.requireNonNull(rank);

this.icon = icon;

}

public Suit getSuit() {

return suit;

}

public Rank getRank() {

return rank;

}

public ImageIcon getIcon() {

return icon;

}

public int getRankValue() {

return RANK\_VALUES.get(rank);

}

public int compareTo(Card other, Suit trumpSuit) {

if (this.suit.equals(other.suit)) {

return Integer.compare(this.getRankValue(), other.getRankValue());

}

if (this.suit.equals(trumpSuit)) {

return 1;

}

if (other.suit.equals(trumpSuit)) {

return -1;

}

return 0;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

Card card = (Card) obj;

return suit == card.suit && rank == card.rank;

}

@Override

public int hashCode() {

return Objects.hash(suit, rank);

}

@Override

public String toString() {

return rank.getSymbol() + " " + suit.name().toLowerCase();

}

public String toNetworkString() {

return suit.name().charAt(0) + ":" + rank.name().charAt(0);

}

public static Card fromNetworkString(String str, ImageIcon icon) {

String[] parts = str.split(":");

Suit suit = Suit.valueOf(parts[0] + parts[0].substring(1).toLowerCase());

Rank rank = Rank.valueOf(parts[1] + parts[1].substring(1).toLowerCase());

return new Card(suit, rank, icon);

}

}

## Приложение A.2. Файл «Client.java»

package fool;

import fool.Card.Rank;

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.image.BufferedImage;

import java.io.\*;

import java.net.Socket;

import java.util.List;

import java.util.\*;

public class Client extends JFrame {

private JPanel attackPanel;

private JPanel defensePanel;

private JPanel playerHandPanel, tablePanel, controlsPanel;

private JButton throwButton, coverButton, takeButton, passButton;

private JTextArea logArea;

private JLabel trumpCardLabel;

private List<Card> playerHand = new ArrayList<>();

private List<Card> selectedCards = new ArrayList<>();

private List<Card> tableCards = new ArrayList<>();

private int playerId;

private PrintWriter out;

private BufferedReader in;

private Card.Suit trumpSuit;

private boolean isDefending = false;

private boolean isAdding = false;

private boolean isMyTurn = false;

private Map<String, ImageIcon> cardImages = new HashMap<>();

private static final String ASSETS\_PATH = "src/asset/";

public Client() {

System.out.println("Cards path: " + ASSETS\_PATH);

loadCardImages();

initComponents();

connectToServer();

}

private void loadCardImages() {

Card.Suit[] suits = Card.Suit.values();

Card.Rank[] ranks = Card.Rank.values();

for (Card.Suit suit : suits) {

for (Card.Rank rank : ranks) {

try {

String filename = String.format("%s-card(%s).png",

suit.name().toLowerCase(),

rankToString(rank));

String fullPath = ASSETS\_PATH + filename;

System.out.println("Loading: " + fullPath);

File file = new File(fullPath);

if (!file.exists()) {

System.err.println("File not found: " + fullPath);

continue;

}

ImageIcon originalIcon = new ImageIcon(file.toURI().toURL());

Image scaledImage = originalIcon.getImage()

.getScaledInstance(80, 120, Image.SCALE\_SMOOTH);

String cardKey = String.format("%s\_%s",

suit.name().toLowerCase(),

rankToString(rank));

cardImages.put(cardKey, new ImageIcon(scaledImage));

System.out.println("Success: " + filename);

} catch (Exception e) {

System.err.printf("Error loading %s-card(%s).png: %s%n",

suit.name().toLowerCase(),

rankToString(rank),

e.getMessage());

}

}

}

try {

String backFilename = "card-back.png";

String backPath = ASSETS\_PATH + backFilename;

System.out.println("Loading back: " + backPath);

File backFile = new File(backPath);

if (backFile.exists()) {

ImageIcon backIcon = new ImageIcon(backFile.toURI().toURL());

Image scaledBack = backIcon.getImage()

.getScaledInstance(80, 120, Image.SCALE\_SMOOTH);

cardImages.put("back", new ImageIcon(scaledBack));

System.out.println("Back loaded");

} else {

System.err.println("Back file not found, creating programmatically");

BufferedImage img = new BufferedImage(80, 120, BufferedImage.TYPE\_INT\_RGB);

Graphics2D g2d = img.createGraphics();

g2d.setColor(new Color(0, 0, 139));

g2d.fillRect(0, 0, 80, 120);

g2d.setColor(Color.WHITE);

g2d.setFont(new Font("Arial", Font.BOLD, 10));

g2d.drawString("BACK", 10, 60);

g2d.dispose();

cardImages.put("back", new ImageIcon(img));

}

} catch (Exception e) {

System.err.println("Error loading back: " + e.getMessage());

}

System.out.printf("Total loaded: %d images%n", cardImages.size());

}

private String rankToString(Card.Rank rank) {

return switch (rank) {

case SIX ->

"6";

case SEVEN ->

"7";

case EIGHT ->

"8";

case NINE ->

"9";

case TEN ->

"10";

case JACK ->

"J";

case QUEEN ->

"Q";

case KING ->

"K";

case ACE ->

"A";

};

}

private void initComponents() {

setTitle("Durak Game");

setSize(1200, 800);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLayout(new BorderLayout());

tablePanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));

tablePanel.setBackground(new Color(0, 100, 0));

playerHandPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));

playerHandPanel.setBackground(new Color(0, 100, 0));

controlsPanel = new JPanel(new FlowLayout());

controlsPanel.setBackground(new Color(240, 240, 240));

trumpCardLabel = new JLabel("Trump: ");

throwButton = new JButton("Throw");

coverButton = new JButton("Cover");

takeButton = new JButton("Take");

passButton = new JButton("Pass");

Font buttonFont = new Font("Arial", Font.BOLD, 14);

throwButton.setFont(buttonFont);

coverButton.setFont(buttonFont);

takeButton.setFont(buttonFont);

passButton.setFont(buttonFont);

controlsPanel.add(trumpCardLabel);

controlsPanel.add(throwButton);

controlsPanel.add(coverButton);

controlsPanel.add(takeButton);

controlsPanel.add(passButton);

logArea = new JTextArea(15, 30);

logArea.setEditable(false);

logArea.setFont(new Font("Monospaced", Font.PLAIN, 12));

JScrollPane scrollPane = new JScrollPane(logArea);

JScrollPane scrollPaneHand = new JScrollPane(playerHandPanel);

scrollPaneHand.setVerticalScrollBarPolicy(JScrollPane.VERTICAL\_SCROLLBAR\_NEVER);

add(scrollPaneHand, BorderLayout.SOUTH);

add(controlsPanel, BorderLayout.NORTH);

add(tablePanel, BorderLayout.CENTER);

//add(playerHandPanel, BorderLayout.SOUTH);

add(scrollPane, BorderLayout.EAST);

throwButton.addActionListener(this::throwAction);

coverButton.addActionListener(this::coverAction);

takeButton.addActionListener(this::takeAction);

passButton.addActionListener(this::passAction);

// Панели для карт атаки и защиты

attackPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));

defensePanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10, 10));

attackPanel.setOpaque(false);

defensePanel.setOpaque(false);

attackPanel.setBorder(null);

defensePanel.setBorder(null);

JPanel container = new JPanel();

container.setLayout(new BoxLayout(container, BoxLayout.Y\_AXIS));

container.setOpaque(false);

container.add(attackPanel);

container.add(defensePanel);

tablePanel.add(container, BorderLayout.CENTER);

add(tablePanel, BorderLayout.CENTER);

updateButtons(false);

}

private void throwAction(ActionEvent e) {

if (!selectedCards.isEmpty()) {

if (canThrowCard(selectedCards.get(0))) {

StringBuilder move = new StringBuilder("THROW");

for (Card card : selectedCards) {

move.append(":").append(card.getRank()).append(" ").append(card.getSuit());

logArea.append("Throwing card: " + card + "\n");

}

out.println(move.toString());

selectedCards.clear();

updateButtons(false);

} else {

logArea.append("Cannot throw - no cards with same rank on table!\n");

}

}

}

private boolean canThrowCard(Card card) {

if (tableCards.isEmpty()) {

return true;

}

String rankToMatch = card.getRank().name();

for (Card tableCard : tableCards) {

if (tableCard.getRank().name().equals(rankToMatch)) {

return true;

}

}

return false;

}

private void coverAction(ActionEvent e) {

if (!selectedCards.isEmpty() && selectedCards.size() == 1) {

Card card = selectedCards.get(0);

logArea.append("Covering with: " + card + "\n");

out.println("COVER:" + card.getRank() + " " + card.getSuit());

selectedCards.clear();

updateButtons(false);

}

}

private void takeAction(ActionEvent e) {

logArea.append("Taking cards\n");

out.println("TAKE");

selectedCards.clear();

updateButtons(false);

}

private void passAction(ActionEvent e) {

logArea.append("Passing\n");

out.println("PASS");

selectedCards.clear();

updateButtons(false);

}

private void updateButtons(boolean myTurn) {

isMyTurn = myTurn;

boolean canThrow = myTurn && !selectedCards.isEmpty()

&& (tableCards.isEmpty() || canThrowCard(selectedCards.get(0)));

boolean canCover = myTurn && isDefending && !selectedCards.isEmpty()

&& selectedCards.size() == 1;

throwButton.setEnabled(canThrow);

coverButton.setEnabled(canCover);

takeButton.setEnabled(myTurn && isDefending);

passButton.setEnabled(myTurn && isAdding);

}

private void connectToServer() {

try {

Socket socket = new Socket("localhost", 5555);

out = new PrintWriter(socket.getOutputStream(), true);

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

new Thread(() -> {

try {

String serverMessage;

while ((serverMessage = in.readLine()) != null) {

processServerMessage(serverMessage);

}

} catch (IOException e) {

e.printStackTrace();

}

}).start();

} catch (IOException e) {

JOptionPane.showMessageDialog(this,

"Failed to connect to server",

"Error",

JOptionPane.ERROR\_MESSAGE);

System.exit(1);

}

}

private void processServerMessage(String message) {

SwingUtilities.invokeLater(() -> {

logArea.append("Server: " + message + "\n");

if (message.startsWith("PLAYER\_ID:")) {

playerId = Integer.parseInt(message.split(":")[1]);

setTitle("Durak Game - Player " + playerId);

} else if (message.startsWith("GAME\_STARTED:")) {

logArea.append("Game started with " + message.split(":")[1] + " players\n");

} else if (message.startsWith("YOUR\_HAND:")) {

updateHand(message);

} else if (message.startsWith("TURN:")) {

String[] parts = message.split(":");

int current = Integer.parseInt(parts[1]);

isMyTurn = (current == playerId);

isDefending = parts[2].equals("defense");

isAdding = parts[2].equals("adding");

updateButtons(isMyTurn);

String phase;

if (isDefending) {

phase = "defense";

} else if (isAdding) {

phase = "add";

} else {

phase = "attack";

}

if (isMyTurn) {

logArea.append("Your turn to " + phase + "\n");

}

} else if (message.startsWith("TRUMP:")) {

trumpSuit = Card.Suit.valueOf(message.split(":")[1].toUpperCase());

trumpCardLabel.setText("Trump: " + trumpSuit.toString().toLowerCase());

ImageIcon trumpIcon = cardImages.get("back");

if (trumpIcon != null) {

trumpCardLabel.setIcon(trumpIcon);

}

} else if (message.startsWith("TABLE:")) {

updateTable(message);

} else if (message.startsWith("GAME\_OVER:")) {

String winner = message.substring("GAME\_OVER:".length());

JOptionPane.showMessageDialog(this, "Игра окончена!\nПобедитель: " + winner,

"Game Over", JOptionPane.INFORMATION\_MESSAGE);

dispose();

System.exit(0);

} else if (message.startsWith("PLAYER\_THREW:")

|| message.startsWith("PLAYER\_COVERED:")

|| message.startsWith("PLAYER\_TOOK:")

|| message.startsWith("PLAYER\_PASSED:")) {

} else if (message.startsWith("ERROR:")) {

String errorMsg = message.substring(6);

JOptionPane.showMessageDialog(this, errorMsg, "Error", JOptionPane.ERROR\_MESSAGE);

selectedCards.clear();

updateButtons(true);

logArea.append("Error received: " + errorMsg + ". Please choose another card.\n");

}

});

}

private void updateHand(String message) {

playerHand.clear();

String[] cards = message.substring(10).split(",");

for (String cardStr : cards) {

if (!cardStr.isEmpty()) {

String[] parts = cardStr.trim().split(" ");

if (parts.length == 2) {

try {

Card.Suit suit = Card.Suit.valueOf(parts[1].toUpperCase());

Card.Rank rank = Card.Rank.fromString(parts[0]); //

String key = suit.name().toLowerCase() + "\_" + rank.getSymbol();

ImageIcon icon = cardImages.get(key);

if (icon == null) {

icon = cardImages.get("back");

System.err.println("No image found for: " + key);

}

playerHand.add(new Card(suit, rank, icon));

} catch (IllegalArgumentException e) {

System.err.println("Invalid card: " + cardStr + " - " + e.getMessage());

}

}

}

}

showPlayerHand();

}

private void updateTable(String message) {

attackPanel.removeAll();

defensePanel.removeAll();

tableCards.clear();

String data = message.substring(6);

String[] parts = data.split("\\|", 2);

String attackingPart = parts.length > 0 ? parts[0] : "";

String defendingPart = parts.length > 1 ? parts[1] : "";

for (String cardStr : attackingPart.split(",")) {

cardStr = cardStr.trim();

if (!cardStr.isEmpty()) {

try {

String[] cardParts = cardStr.split(" ");

if (cardParts.length == 2) {

Card.Rank rank = Card.Rank.fromString(cardParts[0]);

Card.Suit suit = Card.Suit.valueOf(cardParts[1].toUpperCase());

String key = suit.name().toLowerCase() + "\_" + rank.getSymbol();

ImageIcon icon = cardImages.get(key);

if (icon != null) {

JLabel cardLabel = new JLabel(icon);

attackPanel.add(cardLabel);

tableCards.add(new Card(suit, rank, icon));

}

}

} catch (Exception e) {

System.err.println("Invalid attacking card: " + cardStr + " - " + e.getMessage());

}

}

}

int numCardsAtk = attackingPart.isEmpty() ? 0 : attackingPart.split(",").length;

int numCardsDef = defendingPart.isEmpty() ? 0 : defendingPart.split(",").length;

int shiftPx = 0;

if (numCardsDef > 1) {

shiftPx = (numCardsAtk - numCardsDef) \* 50; // Настраиваем под себя

}

defensePanel.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10));

defensePanel.add(Box.createRigidArea(new Dimension(shiftPx, 0)), 0);

// Парсим защищающие карты

for (String cardStr : defendingPart.split(",")) {

cardStr = cardStr.trim();

if (!cardStr.isEmpty()) {

try {

String[] cardParts = cardStr.split(" ");

if (cardParts.length == 2) {

Card.Rank rank = Card.Rank.fromString(cardParts[0]);

Card.Suit suit = Card.Suit.valueOf(cardParts[1].toUpperCase());

String key = suit.name().toLowerCase() + "\_" + rank.getSymbol();

ImageIcon icon = cardImages.get(key);

if (icon != null) {

JLabel cardLabel = new JLabel(icon);

defensePanel.add(cardLabel);

tableCards.add(new Card(suit, rank, icon));

}

}

} catch (Exception e) {

System.err.println("Invalid defending card: " + cardStr + " - " + e.getMessage());

}

}

}

attackPanel.revalidate();

attackPanel.repaint();

defensePanel.revalidate();

defensePanel.repaint();

tablePanel.revalidate();

tablePanel.repaint();

}

private void showPlayerHand() {

playerHandPanel.removeAll();

selectedCards.clear();

for (Card card : playerHand) {

JLabel cardLabel = new JLabel(card.getIcon());

cardLabel.setCursor(Cursor.getPredefinedCursor(Cursor.HAND\_CURSOR));

cardLabel.setBorder(null); // Убираем рамку по умолчанию

final boolean[] isSelected = {false};

cardLabel.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

if (isSelected[0]) {

cardLabel.setBorder(null);

selectedCards.remove(card);

isSelected[0] = false;

} else {

if (isAdding) {

boolean canSelect = false;

for (Card tableCard : tableCards) {

if (tableCard.getRank() == card.getRank()) {

canSelect = true;

break;

}

}

if (!canSelect) {

return;

}

} else

if (!selectedCards.isEmpty()) {

Rank currentRank = selectedCards.get(0).getRank();

if (card.getRank() != currentRank) {

return;

}

}

cardLabel.setBorder(BorderFactory.createLineBorder(Color.YELLOW, 2));

selectedCards.add(card);

isSelected[0] = true;

}

if (isAdding) {

passButton.setEnabled(true);

} else {

passButton.setEnabled(false);

}

if (!isDefending) {

throwButton.setEnabled(!selectedCards.isEmpty());

coverButton.setEnabled(false);

} else {

coverButton.setEnabled(!selectedCards.isEmpty());

throwButton.setEnabled(false);

}

playerHandPanel.revalidate();

playerHandPanel.repaint();

}

});

playerHandPanel.add(cardLabel);

}

playerHandPanel.revalidate();

playerHandPanel.repaint();

}

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

Client client = new Client();

client.setVisible(true);

});

}

}

## Приложение A.3. Файл «Server.java»

package fool;

import java.io.\*;

import java.net.\*;

import java.util.\*;

public class Server {

private static final int PORT = 5555;

static List<ClientHandler> clients = new ArrayList<>();

private static GameLogic gameLogic;

private static List<Card> fullDeck;

public static void main(String[] args) {

System.out.println("Server started. Waiting for connections...");

initDeck();

try (ServerSocket serverSocket = new ServerSocket(PORT)) {

while (true) {

Socket socket = serverSocket.accept();

System.out.println("New connection from: " + socket.getInetAddress());

ClientHandler clientThread = new ClientHandler(socket);

clients.add(clientThread);

clientThread.start();

if (clients.size() >= 2) {

startGame();

}

}

} catch (IOException e) {

System.err.println("Server error: " + e.getMessage());

}

}

private static void initDeck() {

fullDeck = new ArrayList<>();

Card.Suit[] suits = Card.Suit.values();

Card.Rank[] ranks = Card.Rank.values();

for (Card.Suit suit : suits) {

for (Card.Rank rank : ranks) {

fullDeck.add(new Card(suit, rank, null));

}

}

Collections.shuffle(fullDeck);

System.out.println("Deck initialized with " + fullDeck.size() + " cards");

}

private static void startGame() {

System.out.println("Starting game with " + clients.size() + " players");

gameLogic = new GameLogic(clients.size(), fullDeck);

broadcast("TRUMP:" + gameLogic.getTrumpSuit());

broadcast("GAME\_STARTED:" + clients.size());

for (int i = 0; i < clients.size(); i++) {

clients.get(i).setPlayerId(i);

clients.get(i).sendMessage("PLAYER\_ID:" + (i));

sendHand(i);

System.out.println("Player " + (i) + " initialized");

}

broadcastTurnInfo();

}

private static void broadcastTurnInfo() {

int current = gameLogic.getCurrentPlayer();

String phase;

if (gameLogic.getCurrentPhase() == GameLogic.RoundPhase.ATTACKING) {

phase = "attack";

} else if (gameLogic.getCurrentPhase() == GameLogic.RoundPhase.ADDING\_CARDS) {

phase = "adding";

} else {

phase = "defense";

}

String message = "TURN:" + current + ":" + phase;

System.out.println("Broadcasting turn info: " + message);

broadcast(message);

}

public static void broadcast(String message) {

System.out.println("Broadcasting: " + message);

for (ClientHandler client : clients) {

client.sendMessage(message);

}

}

public static void sendHand(int playerId) {

StringBuilder hand = new StringBuilder("YOUR\_HAND:");

for (Card card : gameLogic.getPlayerHand(playerId)) {

hand.append(card).append(",");

}

System.out.println("Sending hand to player " + (playerId) + ": " + hand);

clients.get(playerId).sendMessage(hand.toString());

}

public static void updateTable() {

StringBuilder table = new StringBuilder("TABLE:");

List<Card> attacking = gameLogic.getAttackingCards();

List<Card> defending = gameLogic.getDefendingCards();

for (Card card : attacking) {

table.append(card.toString()).append(",");

}

table.append("|");

for (Card card : defending) {

table.append(card.toString()).append(",");

}

System.out.println("Updating table: " + table);

broadcast(table.toString());

}

public static synchronized void processMove(int playerId, String move) {

int playerIndex = playerId;

String[] parts = move.split(":");

String command = parts[0];

try {

switch (command) {

case "THROW":

if (gameLogic.getCurrentPhase() == GameLogic.RoundPhase.DEFENDING) {

clients.get(playerIndex).sendMessage("ERROR:Not your turn to THROW");

return;

}

List<Card> throwCards = new ArrayList<>();

for (int i = 1; i < parts.length; i++) {

String[] cardData = parts[i].split(" ");

Card.Suit suit = Card.Suit.valueOf(cardData[1].toUpperCase());

Card.Rank rank = Card.Rank.fromString(cardData[0]);

throwCards.add(new Card(suit, rank, null));

}

if (gameLogic.throwCards(playerIndex, throwCards)) {

broadcast("PLAYER\_THREW:" + playerId + ":" + throwCards);

updateTable();

broadcastTurnInfo();

}

break;

case "COVER":

if (gameLogic.getCurrentPhase() != GameLogic.RoundPhase.DEFENDING) {

clients.get(playerIndex).sendMessage("ERROR:Not your turn to COVER");

return;

}

String[] defendData = parts[1].split(" ");

if (defendData.length < 2) {

clients.get(playerId).sendMessage("ERROR:Invalid card format in COVER");

return;

}

Card defendCard = new Card(

Card.Suit.valueOf(defendData[1].toUpperCase()),

Card.Rank.fromString(defendData[0]),

null

);

Card attackCard = gameLogic.getAttackingCards().get(gameLogic.getDefendingCards().size());

if (gameLogic.coverCard(playerIndex, attackCard, defendCard)) {

broadcast("PLAYER\_COVERED:" + playerId + ":" + defendCard);

updateTable();

if (gameLogic.getDefendingCards().size() == gameLogic.getAttackingCards().size()) {

gameLogic.setCurrentPhase(GameLogic.RoundPhase.ADDING\_CARDS);

gameLogic.switchActivePlayerToOpponent();

broadcast("PHASE:ADDING\_CARDS");

}

broadcastTurnInfo();

} else {

clients.get(playerIndex).sendMessage("ERROR:Invalid cover move");

}

break;

case "ADD\_CARDS":

if (gameLogic.getCurrentPhase() != GameLogic.RoundPhase.ADDING\_CARDS) {

clients.get(playerIndex).sendMessage("ERROR:Not your turn to ADD\_CARDS");

return;

}

List<Card> addCards = new ArrayList<>();

for (int i = 1; i < parts.length; i++) {

String[] cardData = parts[i].split(" ");

Card.Suit suit = Card.Suit.valueOf(cardData[1].toUpperCase());

Card.Rank rank = Card.Rank.fromString(cardData[0]);

addCards.add(new Card(suit, rank, null));

}

if (gameLogic.addCards(playerIndex, addCards)) {

broadcast("PLAYER\_ADDED:" + playerId + ":" + addCards);

updateTable();

gameLogic.setCurrentPhase(GameLogic.RoundPhase.DEFENDING);

gameLogic.switchActivePlayerToOpponent();

broadcastTurnInfo();

} else {

clients.get(playerIndex).sendMessage("ERROR:Failed to add cards");

}

break;

case "PASS":

if (gameLogic.getCurrentPhase() != GameLogic.RoundPhase.ADDING\_CARDS) {

clients.get(playerIndex).sendMessage("ERROR:Not your turn to PASS");

return;

}

gameLogic.endRound();

broadcast("ROUND\_ENDED");

updateTable();

broadcastTurnInfo();

break;

case "TAKE":

if (playerIndex != gameLogic.getDefenderIndex()) {

clients.get(playerIndex).sendMessage("ERROR:Not your turn to TAKE");

return;

}

gameLogic.takeCards(playerIndex);

broadcast("PLAYER\_TAKE:" + playerId);

updateTable();

gameLogic.endRound();

broadcast("ROUND\_ENDED");

broadcastTurnInfo();

break;

default:

clients.get(playerIndex).sendMessage("ERROR:Unknown command");

}

} catch (Exception e) {

clients.get(playerIndex).sendMessage("ERROR:" + e.getMessage());

}

updateAllHands();

if (gameLogic.isGameOver()) {

checkGameOver();

}

}

private static void endRound() {

System.out.println("Ending round");

broadcast("ROUND\_ENDED");

gameLogic.endRound();

updateTable();

updateAllHands();

broadcastTurnInfo();

if (gameLogic.isGameOver()) {

checkGameOver();

}

}

private static void checkGameOver() {

boolean gameOver = false;

String reason = "";

boolean deckEmpty = gameLogic.getDeck().isEmpty();

long playersWithCards = gameLogic.getPlayersHands().stream()

.filter(hand -> !hand.isEmpty())

.count();

if (deckEmpty && playersWithCards <= 1) {

gameOver = true;

List<List<Card>> playersHands = gameLogic.getPlayersHands();

for (int i = 0; i < playersHands.size(); i++) {

if (playersHands.get(i).isEmpty()) {

reason = "Игрок " + i;

}

}

}

if (playersWithCards <= 1) {

gameOver = true;

reason = "Ничья";

}

if (gameOver) {

broadcast("GAME\_OVER:" + reason);

for (ClientHandler client : clients) {

client.closeConnection();

}

clients.clear();

System.out.println("Game over: " + reason);

}

}

private static void updateAllHands() {

System.out.println("Updating all hands");

for (int i = 0; i < clients.size(); i++) {

sendHand(i);

}

}

public static synchronized void removeClient(ClientHandler client) {

System.out.println("Removing client: " + client.getPlayerId());

clients.remove(client);

if (clients.size() < 2) {

System.out.println("Not enough players, game cannot continue");

broadcast("GAME\_OVER:Not enough players");

}

}

}

class ClientHandler extends Thread {

private Socket socket;

private PrintWriter out;

private BufferedReader in;

private int playerId;

public ClientHandler(Socket socket) {

this.socket = socket;

try {

out = new PrintWriter(socket.getOutputStream(), true);

in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

} catch (IOException e) {

System.err.println("Error creating client handler: " + e.getMessage());

}

}

public void setPlayerId(int id) {

this.playerId = id;

}

public int getPlayerId() {

return playerId;

}

public void closeConnection() {

try {

socket.close();

} catch (IOException e) {

System.err.println("Err closing socket " + playerId + ": " + e.getMessage());

}

}

public void run() {

try {

System.out.println("Player " + playerId + " handler started");

String inputLine;

while ((inputLine = in.readLine()) != null) {

System.out.println("Received from player " + playerId + ": " + inputLine);

Server.processMove(playerId, inputLine);

}

} catch (IOException e) {

System.out.println("Player " + playerId + " disconnected");

} finally {

Server.removeClient(this);

try {

socket.close();

} catch (IOException e) {

System.err.println("Error closing socket: " + e.getMessage());

}

}

}

public void sendMessage(String msg) {

out.println(msg);

}

}

## Приложение A.4. Файл «GameLogic.java»

package fool;

import java.util.\*;

public class GameLogic {

public enum RoundPhase {

ATTACKING,

DEFENDING,

ADDING\_CARDS,

ROUND\_ENDED

}

private int numPlayers = 2;

private List<List<Card>> playersHands;

private List<Card> deck;

private List<Card> attackingCards;

private List<Card> defendingCards;

private RoundPhase currentPhase;

private int currentPlayer; // индекс атакующего

private int defenderIndex; // индекс защищающегося

private Card.Suit trumpSuit;

public GameLogic(int numPlayers, List<Card> fullDeck) {

this.numPlayers = numPlayers;

this.deck = new ArrayList<>(fullDeck);

this.playersHands = new ArrayList<>();

for (int i = 0; i < numPlayers; i++) {

playersHands.add(new ArrayList<>());

}

this.attackingCards = new ArrayList<>();

this.defendingCards = new ArrayList<>();

this.currentPhase = RoundPhase.ATTACKING;

this.currentPlayer = 0;

this.defenderIndex = 1;

this.trumpSuit = fullDeck.get(0).getSuit();

dealInitialCards();

}

private void dealInitialCards() {

for (int i = 0; i < 6; i++) {

for (List<Card> hand : playersHands) {

if (!deck.isEmpty()) {

hand.add(deck.remove(deck.size() - 1));

}

}

}

}

public Card.Suit getTrumpSuit() {

return trumpSuit;

}

public List<Card> getPlayerHand(int playerId) {

return playersHands.get(playerId);

}

public List<List<Card>> getPlayersHands() {

return playersHands;

}

public List<Card> getDeck() {

return deck;

}

public List<Card> getAttackingCards() {

return attackingCards;

}

public List<Card> getDefendingCards() {

return defendingCards;

}

public int getCurrentPlayer() {

return currentPlayer;

}

public int getDefenderIndex() {

return defenderIndex;

}

public RoundPhase getCurrentPhase() {

return currentPhase;

}

public void setCurrentPhase(RoundPhase phase) {

currentPhase = phase;

}

private void refillHands() {

for (int i = 0; i < numPlayers; i++) {

List<Card> hand = playersHands.get(i);

while (hand.size() < 6 && !deck.isEmpty()) {

hand.add(deck.remove(deck.size() - 1));

}

}

}

public boolean throwCards(int playerIndex, List<Card> cardsToThrow) {

if (playerIndex != currentPlayer) {

return false;

}

if (currentPhase != RoundPhase.ATTACKING && currentPhase != RoundPhase.ADDING\_CARDS) {

return false;

}

//if (!allCardsSameRank(cardsToThrow)) return false;

if (!playersHands.get(playerIndex).containsAll(cardsToThrow)) {

return false;

}

attackingCards.addAll(cardsToThrow);

playersHands.get(playerIndex).removeAll(cardsToThrow);

switchActivePlayerToOpponent();

currentPhase = RoundPhase.DEFENDING;

return true;

}

public boolean coverCard(int playerIndex, Card attackCard, Card defendCard) {

if (playerIndex != defenderIndex) {

return false;

}

if (currentPhase != RoundPhase.DEFENDING) {

return false;

}

if (!attackingCards.contains(attackCard)) {

return false;

}

if (defendingCards.size() >= attackingCards.size()) {

return false;

}

if (!playersHands.get(playerIndex).contains(defendCard)) {

return false;

}

if (!canCover(attackCard, defendCard)) {

return false;

}

defendingCards.add(defendCard);

playersHands.get(playerIndex).remove(defendCard);

if (defendingCards.size() == attackingCards.size()) {

currentPhase = RoundPhase.ADDING\_CARDS;

//switchActivePlayerToOpponent();

}

return true;

}

public boolean canAddCards(List<Card> cardsToAdd) {

if (currentPhase != RoundPhase.ADDING\_CARDS) {

return false;

}

if (cardsToAdd.isEmpty()) {

return false;

}

// Проверяем, что все карты одного номинала

if (!allCardsSameRank(cardsToAdd)) {

return false;

}

// Получаем все номиналы на столе

Set<Card.Rank> ranksOnTable = new HashSet<>();

for (Card c : attackingCards) {

ranksOnTable.add(c.getRank());

}

for (Card c : defendingCards) {

ranksOnTable.add(c.getRank());

}

// Проверяем, что номинал карты для подкидывания есть на столе

for (Card card : cardsToAdd) {

if (!ranksOnTable.contains(card.getRank())) {

return false;

}

}

return true;

}

public boolean addCards(int playerIndex, List<Card> cardsToAdd) {

if (playerIndex != currentPlayer) {

return false;

}

if (!canAddCards(cardsToAdd)) {

return false;

}

if (!playersHands.get(playerIndex).containsAll(cardsToAdd)) {

return false;

}

attackingCards.addAll(cardsToAdd);

playersHands.get(playerIndex).removeAll(cardsToAdd);

// После добавления ход переходит защитнику

currentPhase = RoundPhase.DEFENDING;

currentPlayer = defenderIndex;

return true;

}

public void takeCards(int playerIndex) {

if (playerIndex != defenderIndex) {

return;

}

List<Card> defenderHand = playersHands.get(playerIndex);

defenderHand.addAll(attackingCards);

defenderHand.addAll(defendingCards);

attackingCards.clear();

defendingCards.clear();

refillHands();

currentPhase = RoundPhase.ATTACKING;

}

public void endRound() {

attackingCards.clear();

defendingCards.clear();

refillHands();

currentPlayer = (currentPlayer + 1) % numPlayers;

defenderIndex = (currentPlayer + 1) % numPlayers;

currentPhase = RoundPhase.ATTACKING;

}

public boolean isGameOver() {

boolean deckEmpty = deck.isEmpty();

long playersWithCards = playersHands.stream().filter(hand -> !hand.isEmpty()).count();

return deckEmpty && playersWithCards <= 1;

}

private boolean allCardsSameRank(List<Card> cards) {

if (cards.isEmpty()) {

return false;

}

Card.Rank rank = cards.get(0).getRank();

for (Card c : cards) {

if (c.getRank() != rank) {

return false;

}

}

return true;

}

private boolean canCover(Card attackCard, Card defendCard) {

if (defendCard.getSuit() == attackCard.getSuit() && defendCard.getRank().ordinal() > attackCard.getRank().ordinal()) {

return true;

}

if (defendCard.getSuit() == trumpSuit && attackCard.getSuit() != trumpSuit) {

return true;

}

return false;

}

public void switchActivePlayerToOpponent() {

currentPlayer = (currentPlayer + 1) % 2;

}

}

# Приложение Б. UML-диаграммы

## Приложение Б.1. UML – диаграмма вариантов использования.

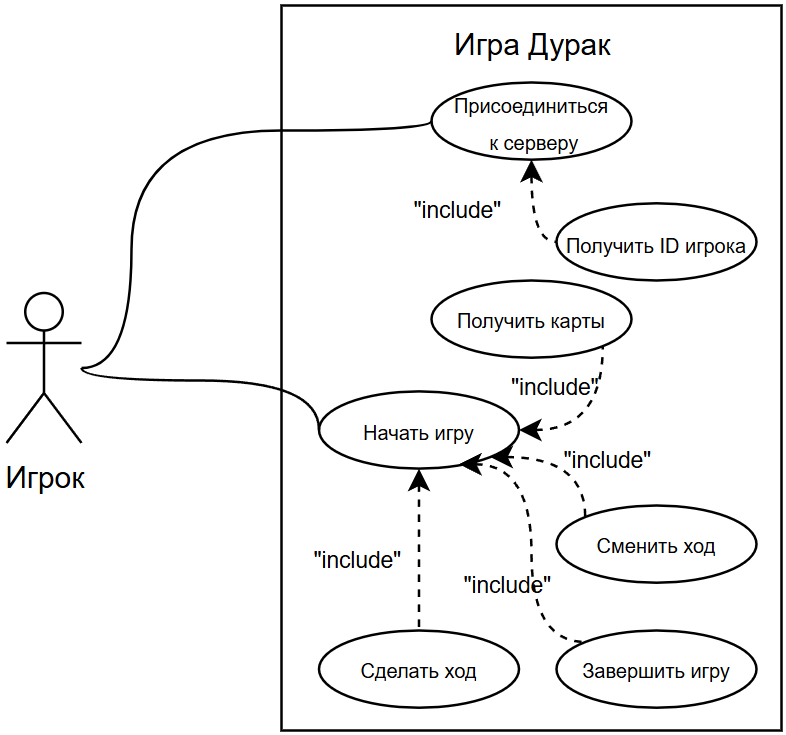


Рисунок 14 - UML – диаграмма вариантов использования

## Приложение Б.2. UML – диаграмма классов.



Рисунок 15 - UML – диаграмма классов

## Приложение Б.3. UML – диаграмма деятельности.

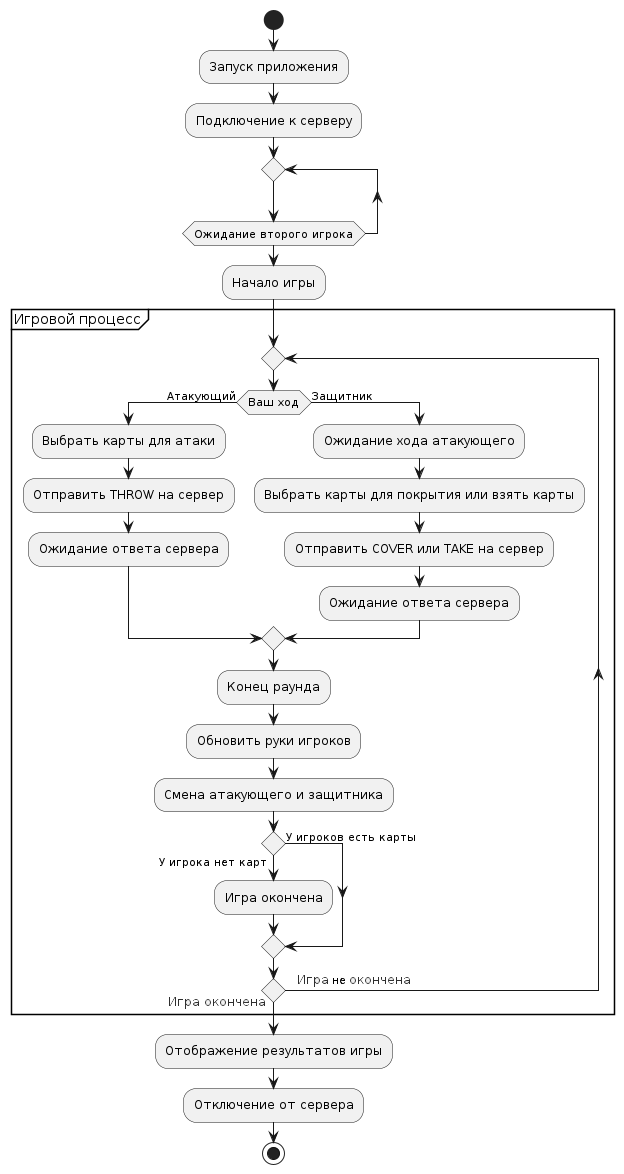


Рисунок 16 - UML – диаграмма деятельности

## Приложение Б.4. UML – диаграмма развёртывания.

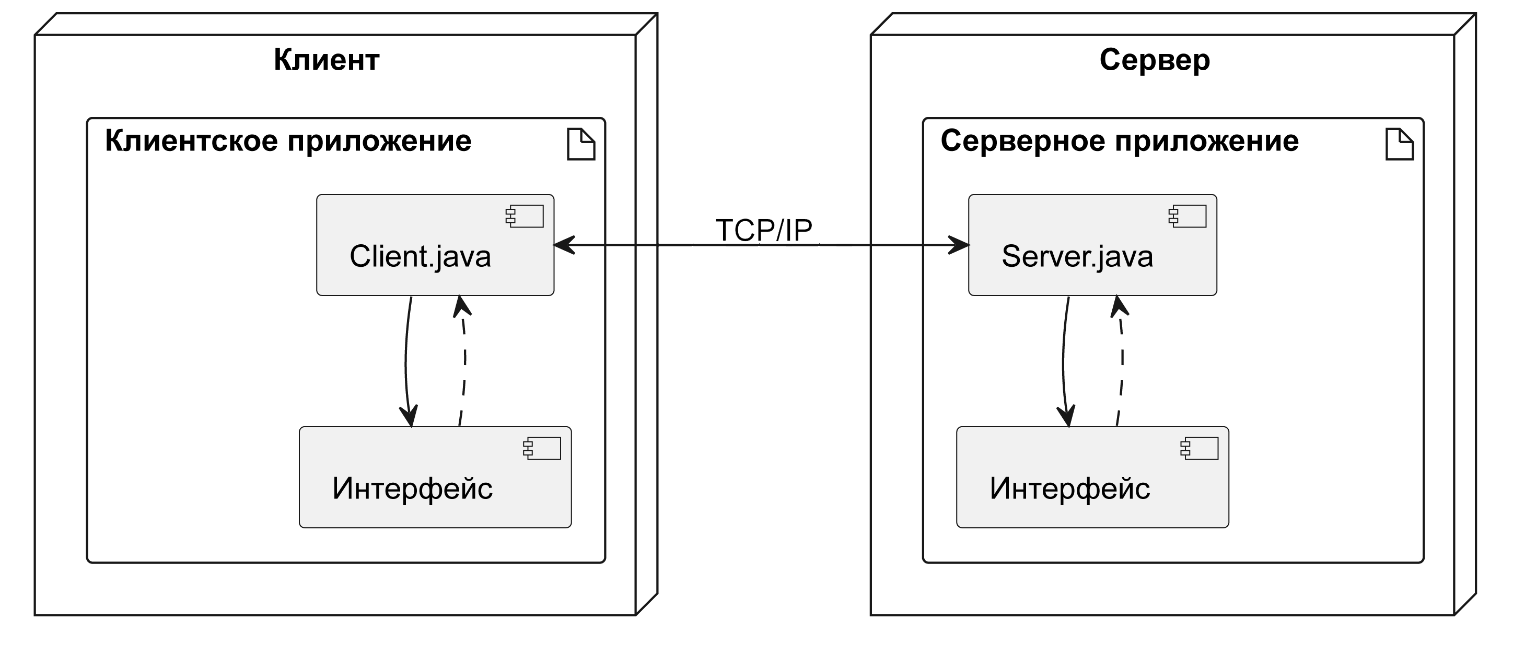


Рисунок 17 - UML – диаграмма развёртывания

## Приложение Б.5. UML – диаграмма последовательности.

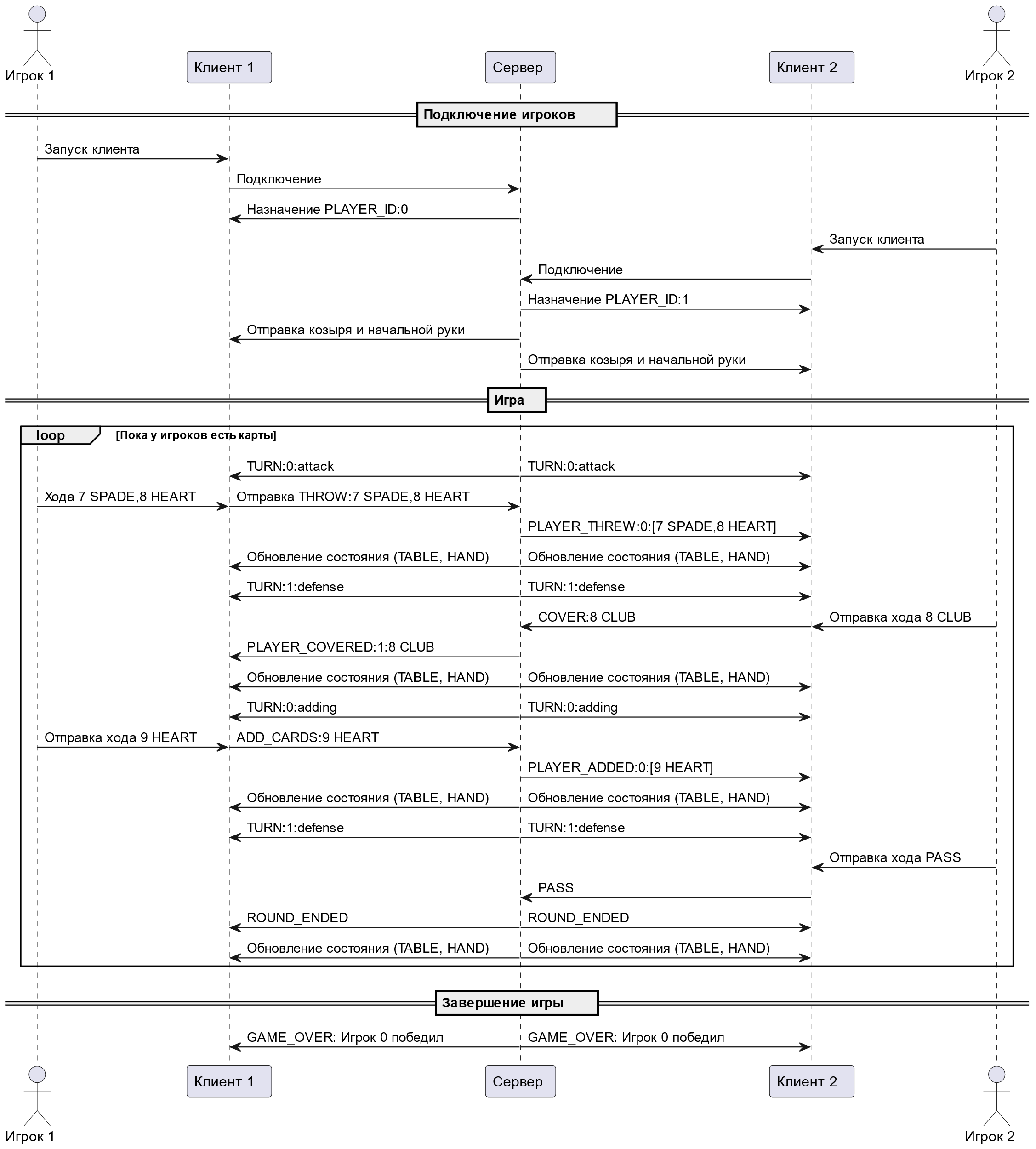


Рисунок 21 - UML – диаграмма последовательности