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

и.o. зав. кафедрой \_\_\_\_\_\_\_\_\_\_ Сычев О.А.

**ЗАДАНИЕ**

**на курсовую работу**

Студент: Быкадоров М.Ю.

Группа: ПрИн-368

1. Тема: «Проектирование и реализация программы с использованием объектно-ориентированного подхода» (индивидуальное задание – вариант №47)

Утверждена приказом от «05» февраля 2025г. № 183-ст

2. Срок представления работы к защите « 06 »   июня  2025 г.

3. Содержание пояснительной записки:

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

4. Перечень графического материала:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Дата выдачи задания «14» февраля 2025 г.

Руководитель проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Литовкин Д.В.

Задание принял к исполнению: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Быкадоров М.Ю.

«14» февраля 2025 г.

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 и реализация программы с использованием объектно-ориентированного подхода»

(индивидуальное задание – вариант №47)

Студент: Быкадоров М.Ю.

Группа: ПрИн-368

Работа зачтена с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «      » \_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Руководитель проекта, нормоконтроллер \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Литовкин Д.В.

Волгоград 2025 г.

**Содержание**

[Формулировка задания 4](#_Toc200065801)

[2 Нефункциональные требования 5](#_Toc200065802)

[3 Первая итерация разработки 6](#_Toc200065803)

[3.1 Формулировка упрощенного варианта задания 6](#_Toc200065804)

[3.2 Функциональные требования (сценарии) 7](#_Toc200065805)

[3.3 Словарь предметной области 15](#_Toc200065806)

[3.4 Структура программы на уровне классов 20](#_Toc200065807)

[3.5 Типовые процессы в программе 21](#_Toc200065808)

[3.6 Человеко-машинное взаимодействие 28](#_Toc200065809)

[3.7 Реализация ключевых классов 29](#_Toc200065810)

[3.8 Реализация ключевых тестовых случаев 64](#_Toc200065811)

[4 Вторая итерация разработки 71](#_Toc200065812)

[4.1 Функциональные требования (сценарии) 71](#_Toc200065813)

[4.2 Словарь предметной области 72](#_Toc200065814)

[4.3 Структура программы на уровне классов 73](#_Toc200065815)

[4.4 Типовые процессы в программе 74](#_Toc200065816)

[4.5 Человеко-машинное взаимодействие 84](#_Toc200065817)

[4.6 Реализация ключевых классов 85](#_Toc200065818)

[4.7 Реализация ключевых тестовых случаев 87](#_Toc200065819)

[5 Список использованной литературы и других источников 90](#_Toc200065820)

# Формулировка задания

**Правила игры «Шахматы»:**

* Игра происходит на квадратной доске H x H, разделенном на ячейки;
* На доске находится две команды: черные и белые фигуры.
* У каждой команды 16 фигур: 2 ладьи, 2 слона, 2 коня, королева, король и 8 пешек.
* Команды ходят по очереди, начиная с белых.
* Игра может закончиться как ничьей, так и победой одной из команд.

**Дополнительные требования:**

Предусмотреть в программе точки расширения, используя которые можно

реализовать вариативную часть программы (в дополнение к базовой функциональности).

**Вариативность:**

Предусмотреть возможность создания новых типов фигур.

**Реализовать:**

Колдун – фигура, которая заменяет одну любую пешку команды. Колдун ходит в каждую из сторон на две клетки и умеет «перепрыгивать» другие фигуры. Также у каждой фигуры должны появиться жизни, которые уменьшаются при совершении фигурой перемещения; когда жизни заканчиваются, фигура умирает.

# 2 Нефункциональные требования

1. Программа должна быть реализована на языке Java SE 22 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

## 3.1 Формулировка упрощенного варианта задания

**Правила игры «Шахматы»**

* Игра происходит на квадратной доске H x H, разделенном на ячейки;
* На доске находится две команды: черные и белые фигуры.
* У каждой команды 16 фигур: 2 ладьи, 2 слона, 2 коня, королева, король и 8 пешек.
* Команды ходят по очереди, начиная с белых.
* Игра может закончиться как ничьей, так и победой одной из команд.

## 3.2 Функциональные требования (сценарии)

*Кому/что поручить*

1. **Сценарий** «Игра завершается победой одного из Игроков»
2. **По указанию пользователя,** игра стартует.
3. **По указанию** Игры создаётся Доска.
4. **По указанию** Игры, Доска создаёт связанные друг с другом ячейки и заселяется Фигурами.
5. **Игра** делает активной команду белых Фигур (белые Фигуры размораживаются).

**{**

1. **Делать**

**{**

5.1.Игрок выбирает Фигуру из своей команды, которую хочет переместить.

5.2.Игра проверяет, может ли Фигура двигаться.  
 5.3. **Если** Фигура может двигаться то

5.3.1.Фигура размораживается.

5.3.2.Игра сообщает Игроку, на какие клетки он может поставить свою Фигуру.

5.4. **По указанию** Игрока Фигура перемещается в указанную Ячейку.

5.5. **Если** внутри Ячейки была вражеская Фигура, то

5.1.5.1 Ячейка удаляет её.

5.6.Игра делает другую команду активной.

**} Пока** не поставлен мат одному из Игроков или не объявлен пат

1. Игра считает победителем игрока, поставившего мат противнику
2. Игра «замораживает» себя и все сущности, входящие в нее
3. **Сценарий завершается**
   1. **Альтернативный сценарий «**Досрочное завершение игры **пользователем».** Сценарий **выполняется в любой точке главного** сценария.
4. **По указанию пользователя**, программа завершается без определения победителя
5. **Сценарий завершается.**
6. **Дочерний сценарий «**Начальная расстановка заселяет доску Фигурами**»**
7. **По указанию** Игры, Начальная расстановка заселяет Доску:
   1. Доска создает и расставляет фигуры команды Черных в соответствии с правилами стандартных шахмат.
   2. Доска создает и расставляет фигура команды Белых в соответствии с правилами стандартных шахмат.
8. **Сценарий завершается.**
9. **Дочерний сценарий** «Игрок выбирает фигуру для хода»
10. Игрок выбирает Фигуру своей команды для совершения хода.
11. Игра проверяет, может ли игрок сходить этой Фигурой.
12. **Если** Игрок может сходить этой Фигурой то
    1. Игра сообщает Игроку, на какие Ячейки он может переместить свою Фигуру.
    2. Фигура «размораживается».
13. **Сценарий завершается.**
14. **Дочерний сценарий** «Фигура не может ходить, т.к. поставит под удар своего Короля»
15. **Если** Фигура своим перемещением поставит под Шах своего Короля, то
    1. Нет возможных Ходов для Фигуры.
16. **Сценарий завершается.**
17. **Альтернативный сценарий «**Фигура может своим ходом спасти от Шаха Короля**»**
18. **Если** Фигура может своим перемещением «спасти» от Шаха своего Короля, то
    1. Возможными Ячейками для перемещения становятся те, которые спасают от Шаха Короля.
19. **Сценарий завершается.**
20. **Альтернативный сценарий «**Движение Фигуры согласно стратегии её перемещения**»**
21. Возможными Ячейками для перемещения становятся те, на которые Фигура может попасть, учитывая свою стратегию перемещения.

2. **Сценарий завершается.**

1. **Дочерний сценарий «**Определение возможных ходов для Короля**»**
2. Король может переместиться на любые соседние Ячейки, внутри которых он не будет под шахом.
3. **Сценарий завершается.**
4. **Дочерний сценарий «**Игрок перемещает Фигуру на другую Ячейку**»**
5. Фигура «исчезает» из своей Ячейки.
6. **Если** в целевой Ячейке была вражеская Фигура то
   1. Вражеская Фигура исчезает из целевой Ячейки и считается «мертвой».
7. Фигура «появляется» в целевой Ячейке.
8. **Сценарий завершается.**
9. **Дочерний сценарий «**Шах Королю**»**
10. **Если** вражеская Фигура из своей Ячейки может попасть в Ячейку Короля, то
    1. Объявляется шах Королю.
11. **Сценарий завершается**
12. **Альтернативный сценарий «**Мат Королю**»**
13. Если при любом перемещении любой Фигуры команды всё ещё будет объявлен Шах их Королю, то
    1. Объявляется мат Королю.
14. **Сценарий завершается.**
15. **Альтернативный сценарий «**Пат**»**
16. Если Королю не объявлен шах, но никакая из Фигур не может переместиться, то
    1. Объявляется пат.
17. **Сценарий завершается.**
18. **Дочерний сценарий «**Проигравшим считается игрок, чьему Королю был объявлен мат**»**
19. **Если** какому-либо из Королей был объявлен мат, то
    1. Доска и все Фигуры на ней «замораживаются».
    2. Игрой объявляется победитель и проигравший.
    3. Игрокам предоставляется возможность начать заново или выйти из игры.
20. **Сценарий завершается.**
21. **Альтернативный сценарий «**Ничья, т.к. был объявлен пат**»**
22. **Если** был объявлен пат, то
    1. Доска и все Фигуры на ней «замораживаются».
    2. Игрой объявляется ничья.
    3. Игрокам предоставляется возможность начать заново или выйти из игры.
23. **Сценарий завершается.**

## 3.3 Словарь предметной области

**Словарь предметной области**

**Игра –** знает о поле и доске. Игра инициирует создание Доски и расстановку всех фигур с помощью Начальной расстановки. Игра определяет очередного активного Игрока, окончание игры и победителя

|  |  |
| --- | --- |
| Знает | * Доска * Начальная расстановка |
| Умеет | * Инициировать создание Доски и расстановку всех Фигур на ней с помощью Начальной расстановки * Определить очередного Игрока, который может обрабатывать запросы пользователя * Определять окончание игры * Определять победителя |
| Предназначение | * Организация общего игрового цикла |

**Доска –** квадратная область, состоящая из черных и белых Ячеек. Знает о Фигурах, находящихся на Доске.

|  |  |
| --- | --- |
| Знает | * Свои размеры (8х8) * Ячейки * Начальная расстановка |
| Умеет | * Создавать себя из ячеек * Предоставлять доступ к Ячейкам * Получать Фигуры, находящихся на ней * Замораживаться |
| Предназначение | * Контейнер Ячеек и Фигур, которые располагаются внутри Ячеек |

**Ячейка –** квадратная область Доски. Знает о 8 соседних Ячейках и Фигуре, находящейся в ней

|  |  |
| --- | --- |
| Знает | * Соседние Ячейки * Фигуру, находящаяся внутри Ячейки |
| Умеет | * Устанавливать соседство с другой ячейкой * Предоставлять доступ к Фигурам, располагающимся на Ячейке * Добавлять/извлекать Фигуры, располагающиеся на Ячейке. На одной Ячейке одновременно может располагаться только одна Фигура |
| Предназначение | * Контейнер Фигур, которые располагаются на Ячейке |

**Начальная расстановка –** умеет создавать Фигуры и размещать их на Доске. Позицию каждой из Фигуры Начальная расстановка определяет самостоятельно.

|  |  |
| --- | --- |
| Знает | * Поле * Требуемую расстановку всех Фигур на Доске |
| Умеет | * Создавать и размещать на Доске все Фигуры |
| Предназначение | * Фабрика сущностей * Расстановка Фигур на Доске |

**Фигура –** умеет единожды за ход перемещаться, после перемещения одной Фигуры остальным Фигурам этой команды запрещается перемещаться в течение этого хода. В зависимости от типа Фигуры они перемещаются по собственной стратегии (Конь буквой «Г», слон по диагонали и т.д.). Если ставится шах королю, право на перемещение имеют только те фигуры, которые могут «спасти» короля от смерти (в том числе сам король).

У каждой Фигуры есть свое время жизни в зависимости от типа Фигуры (пешка может сделать 6 шагов перед смертью и т.д.). Король не имеет время жизни, он может двигаться неограниченное количество раз; смерть для короля наступает только при «мате».

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца * Траекторию, по которой может двигаться Фигура |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю и типу Фигуры * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Пешка –** разновидность Фигуры. Время жизни составляет 6 ходов. Может за раз передвинуться на две или одну клеточку вперед при первом ходе и на одну при последующих, если перед пешкой стоит другая Фигура, двигаться она не может. Атаковать пешка может только по диагонали перед собой максимум на одну Ячейку.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Ладья –** разновидность Фигуры. Время жизни составляет 4 хода. Может за раз передвинуться либо по вертикали, либо по горизонтали в любую из сторон на неограниченное количество Ячеек. Если на пути Ладьи попадается вражеская Фигура, он её «съедает» и становится на её Ячейке. Если перед Ладьёй стоит Фигура союзник, «перепрыгнуть» через неё Ладья не может.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Конь –** разновидность Фигуры. Время жизни составляет 6 ходов. Может ходить буквой «Г» в любую из сторон перепрыгивая все фигуры на пути. Если при прыжке на целевой Ячейке стоит вражеская Фигура, Конь её «съедает», если же там стоит Фигура собственной команды, Конь туда «прыгнуть» не может.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Слон –** разновидность Фигуры. Время жизни составляет 5 ходов. Может ходить только по диагонали в любую из сторон на неограниченное количество Ячеек. Если на пути Слона попадается вражеская Фигура, он её «съедает» и становится на её Ячейке. Если перед Слоном стоит Фигура союзник, «перепрыгнуть» через неё Слон не может.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Королева –** разновидность Фигуры. Время жизни составляет 7 ходов. Может ходить как Ладья и Слон.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Поставить мат королю противника. |

**Король –** разновидность Фигуры. Не имеет время жизни, погибает только в том случае, если вражеская команда поставила ему мат. Может ходить в любую из сторон на одну Ячейку. Не может ходить на ту Ячейку, которую может атаковать вражеская Фигура.

|  |  |
| --- | --- |
| Знает | * Свою позицию, т.е. Ячейку-владельца |
| Умеет | * Перемещаться в другую Ячейку, учитывая возможный шах королю * Замораживаться/размораживаться |
| Предназначение | * Пережить вражеского короля. |

**Траектория –** сущность, внутри которой определяются правила того, как каждая из Фигур может ходить.

|  |  |
| --- | --- |
| Знает | - |
| Умеет | * Вычислять траектория движения каждой из Фигур |
| Предназначение | * Указать, каким образом каждая из Фигур может ходить. |

## 3.4 Структура программы на уровне классов

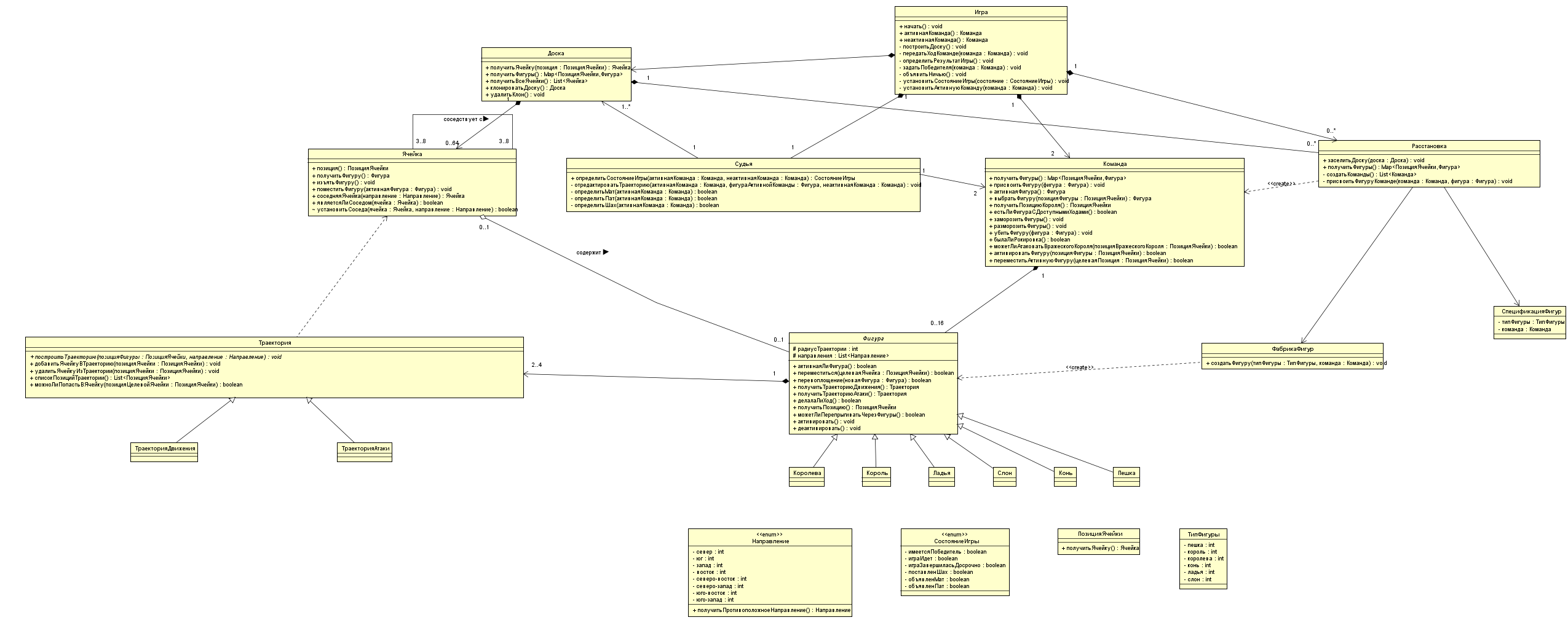


Рисунок 1 – Диаграмма классов вычислительной модели

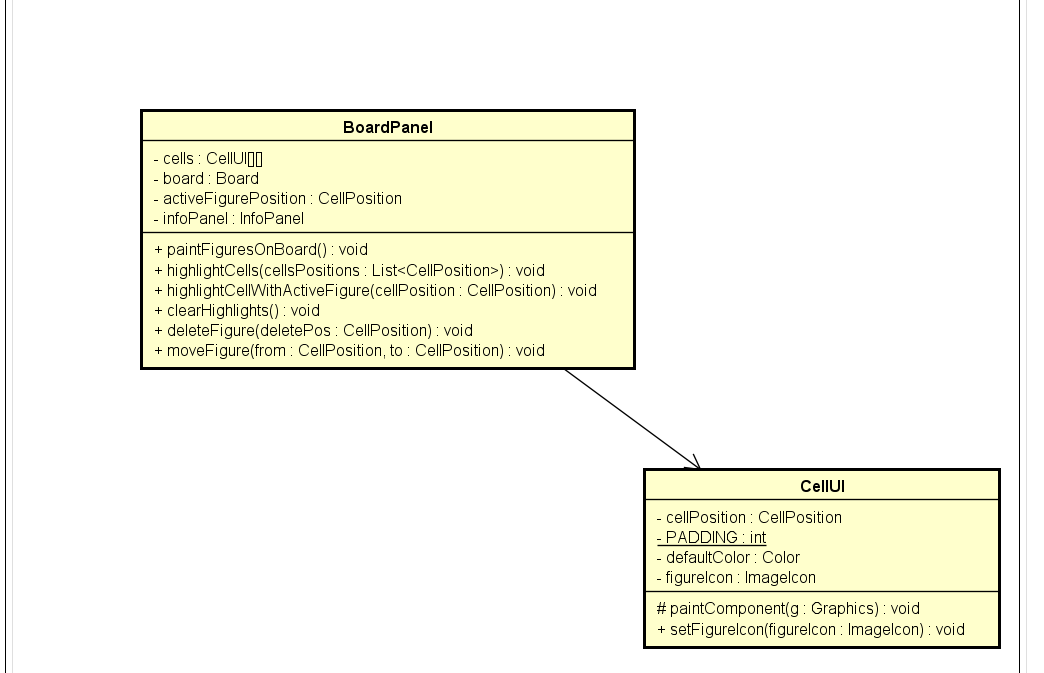


Рисунок 2 – Диаграмма классов представления

## 3.5 Типовые процессы в программе

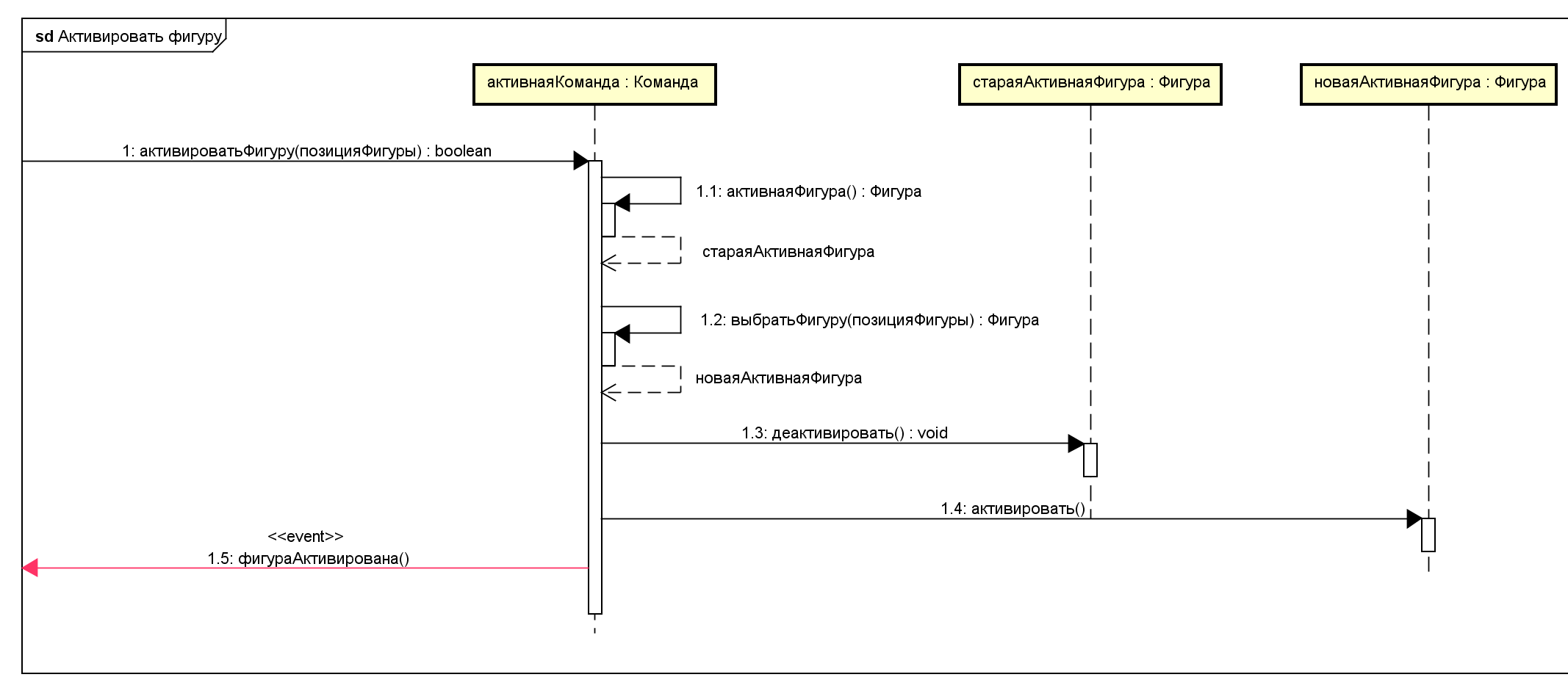


Рисунок 3 – Диаграмма последовательности для вычислительной модели. Активация фигуры.

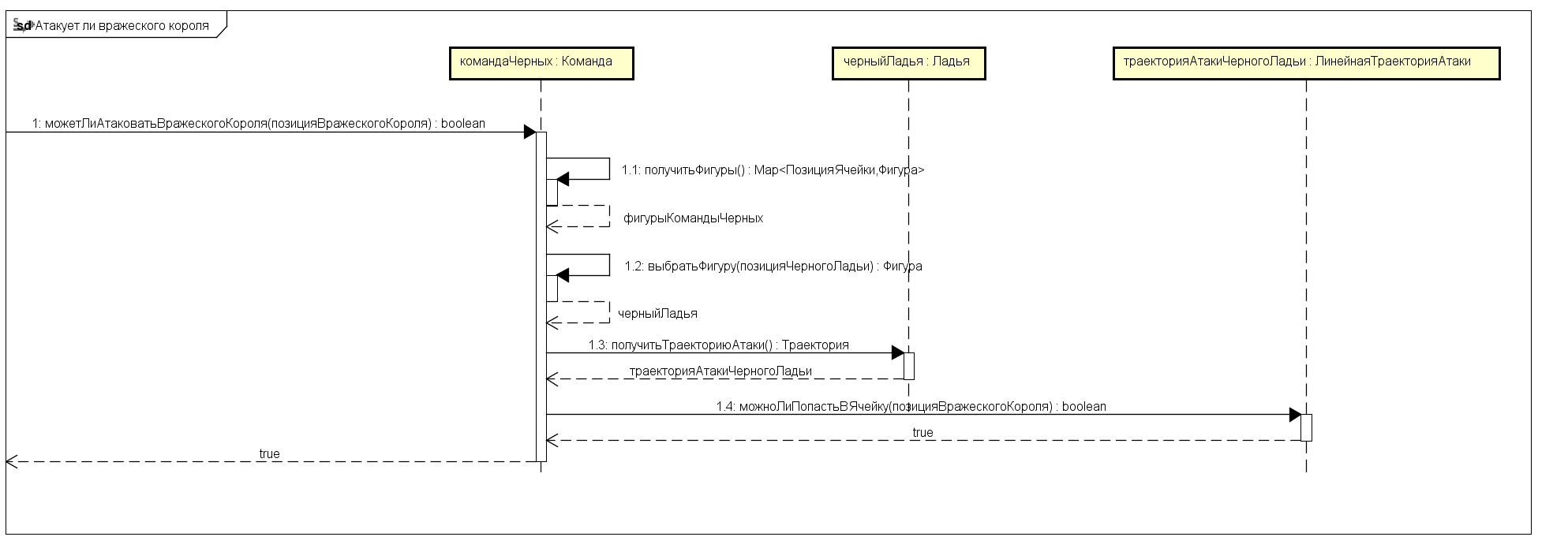


Рисунок 4 – Диаграмма последовательности для вычислительной модели.

Атакует ли вражеского короля.

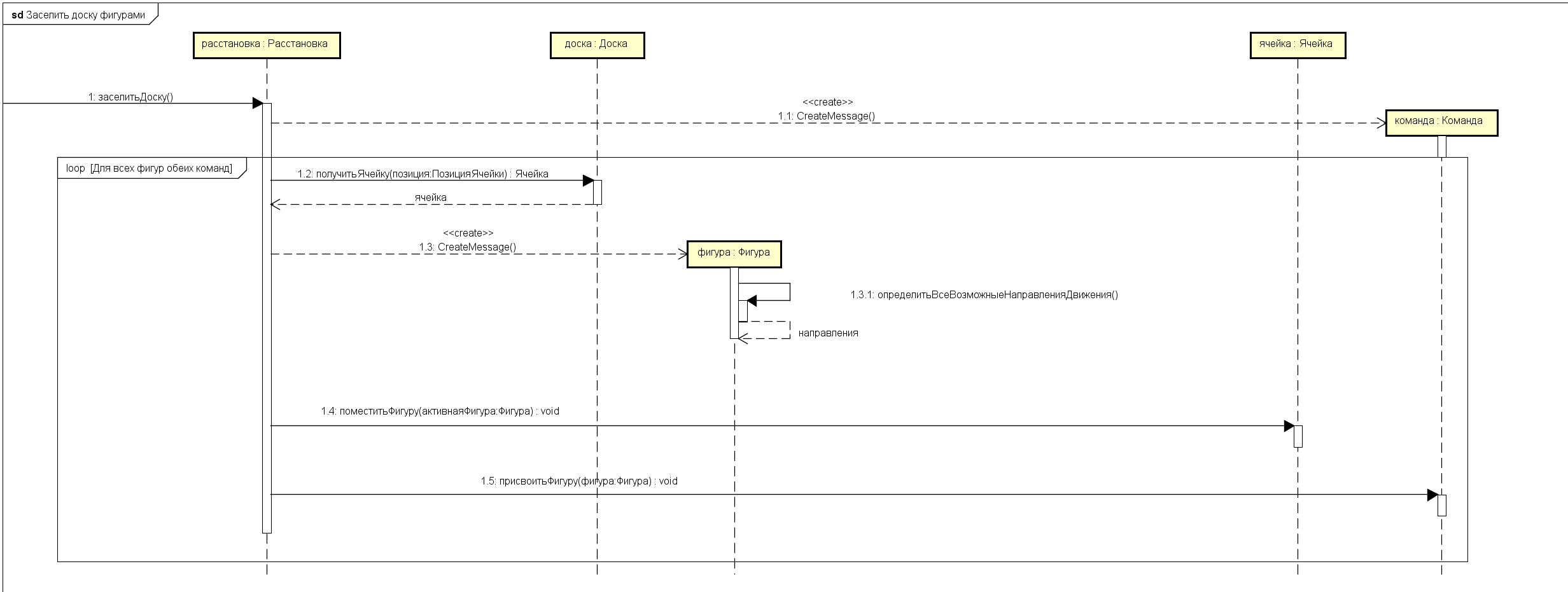


Рисунок 5 – Диаграмма последовательности для вычислительной модели.

Заселить доску фигурами

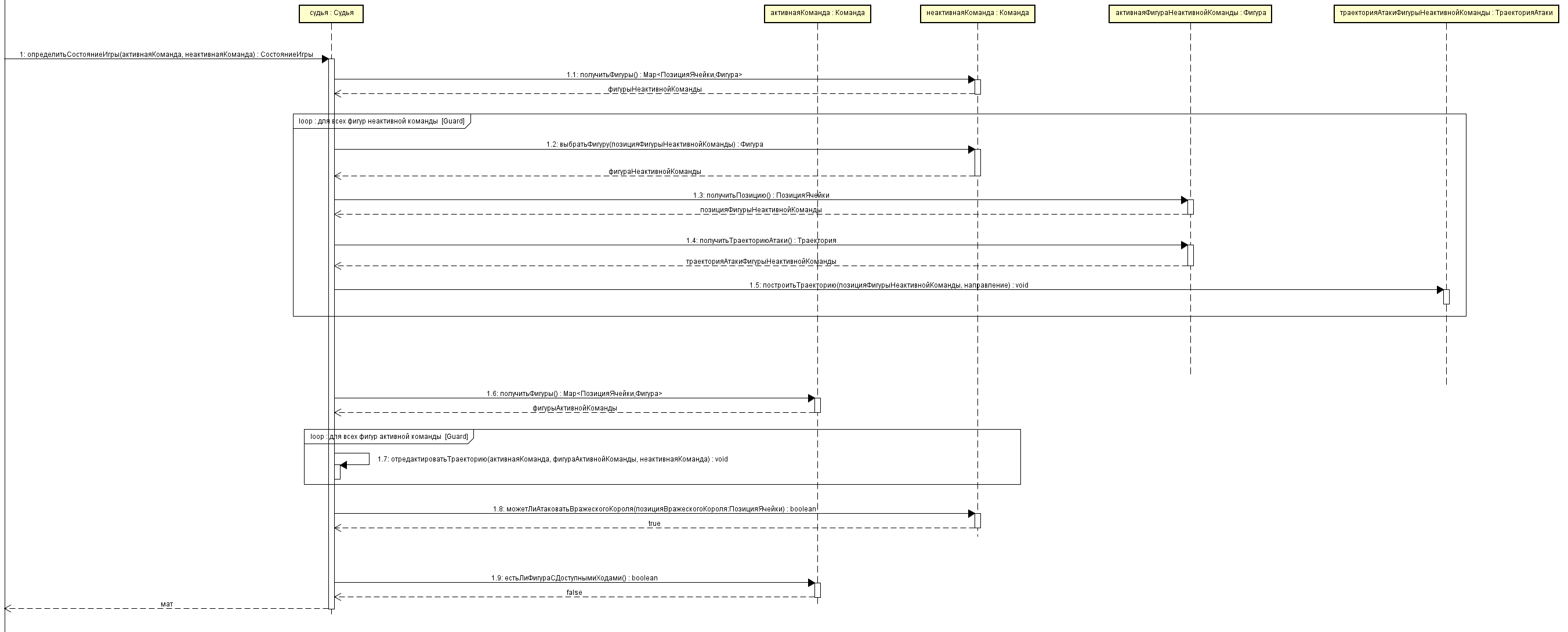


Рисунок 6 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – мат

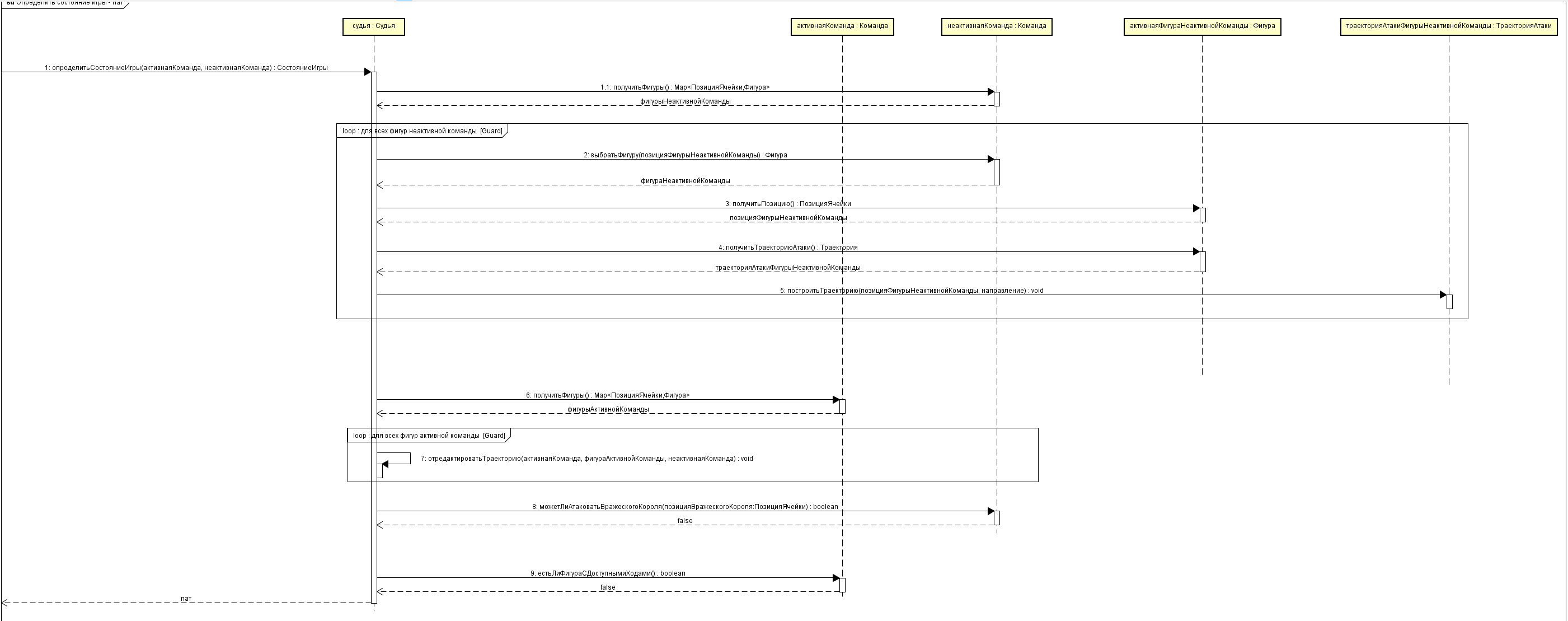


Рисунок 7 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – пат

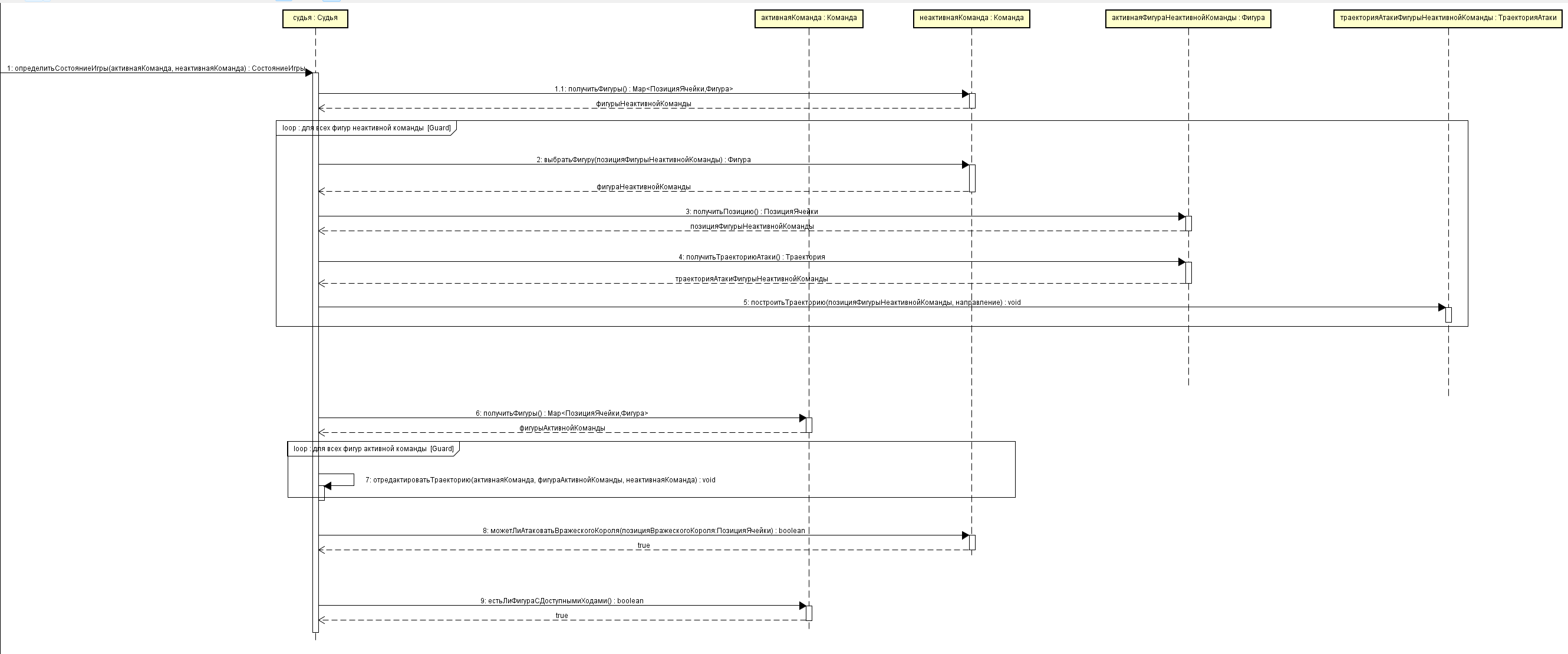


Рисунок 8 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – шах

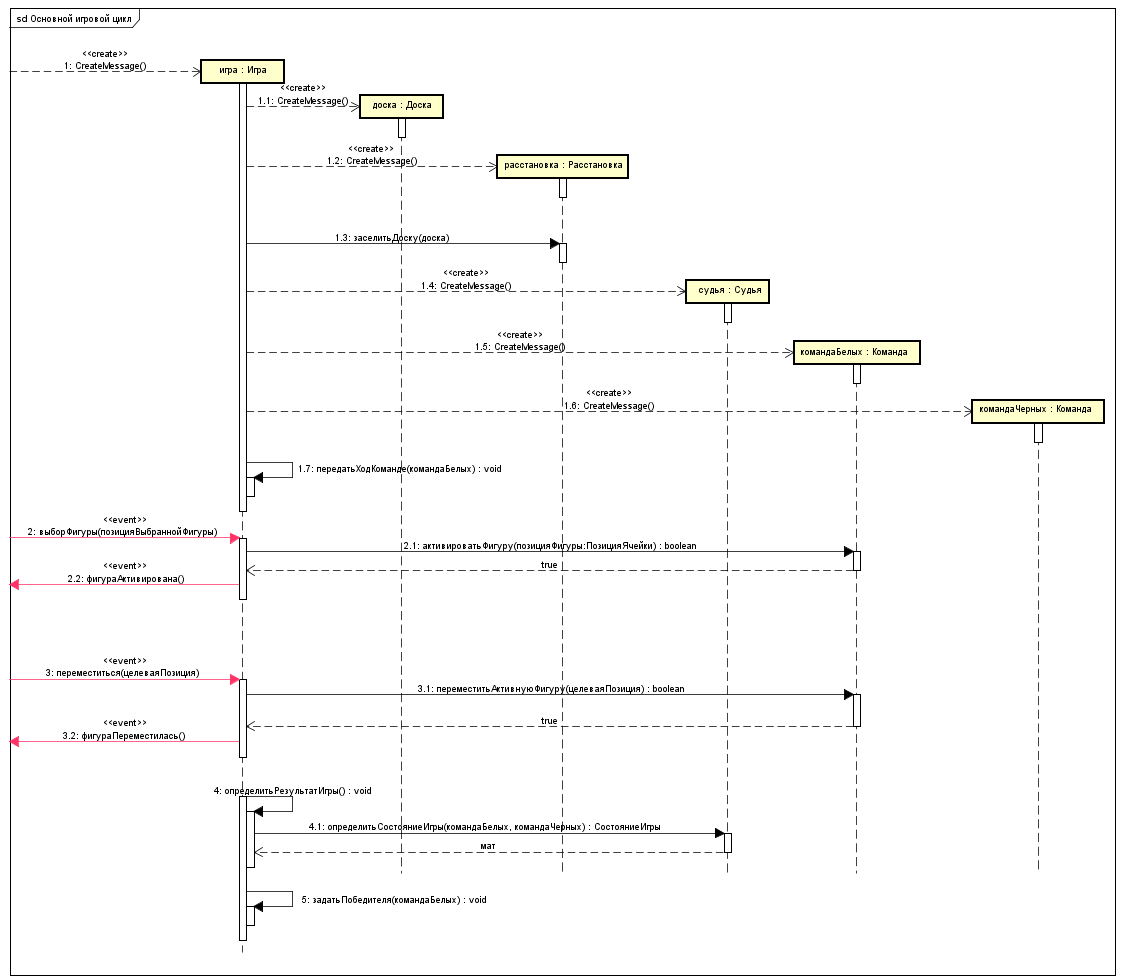


Рисунок 9 – Диаграмма последовательности для вычислительной модели.

Основной игровой цикл

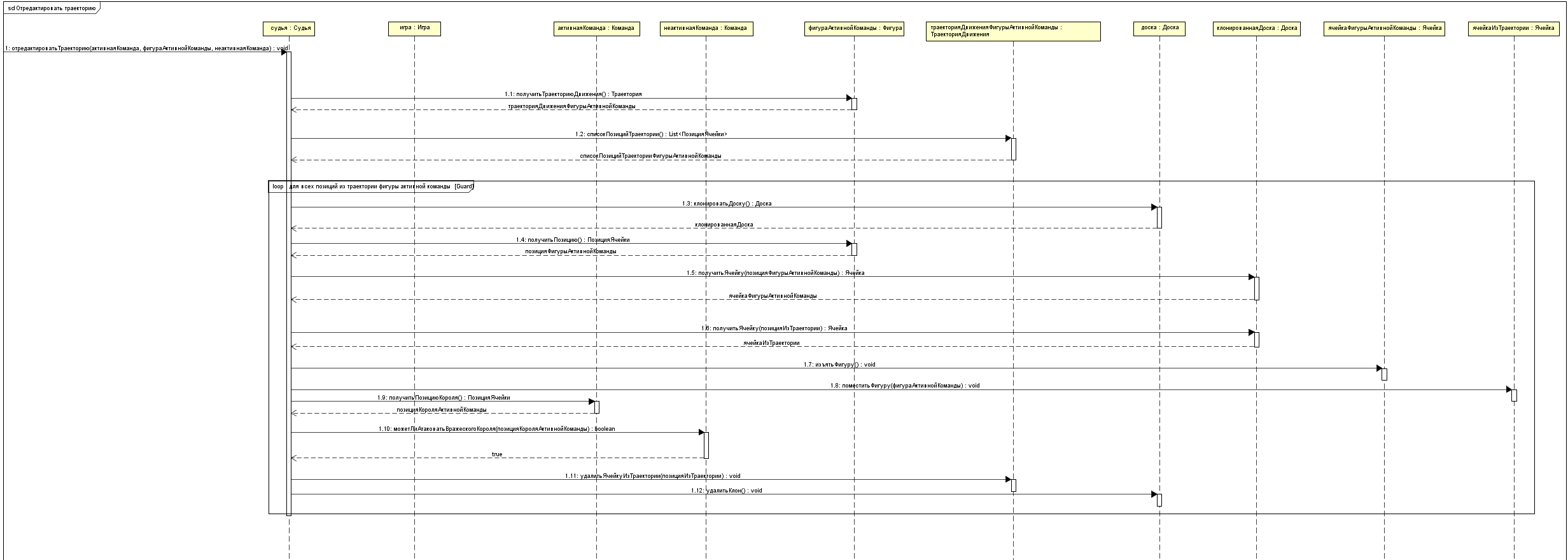


Рисунок 10 - Диаграмма последовательности для вычислительной модели.

Отредактировать траекторию

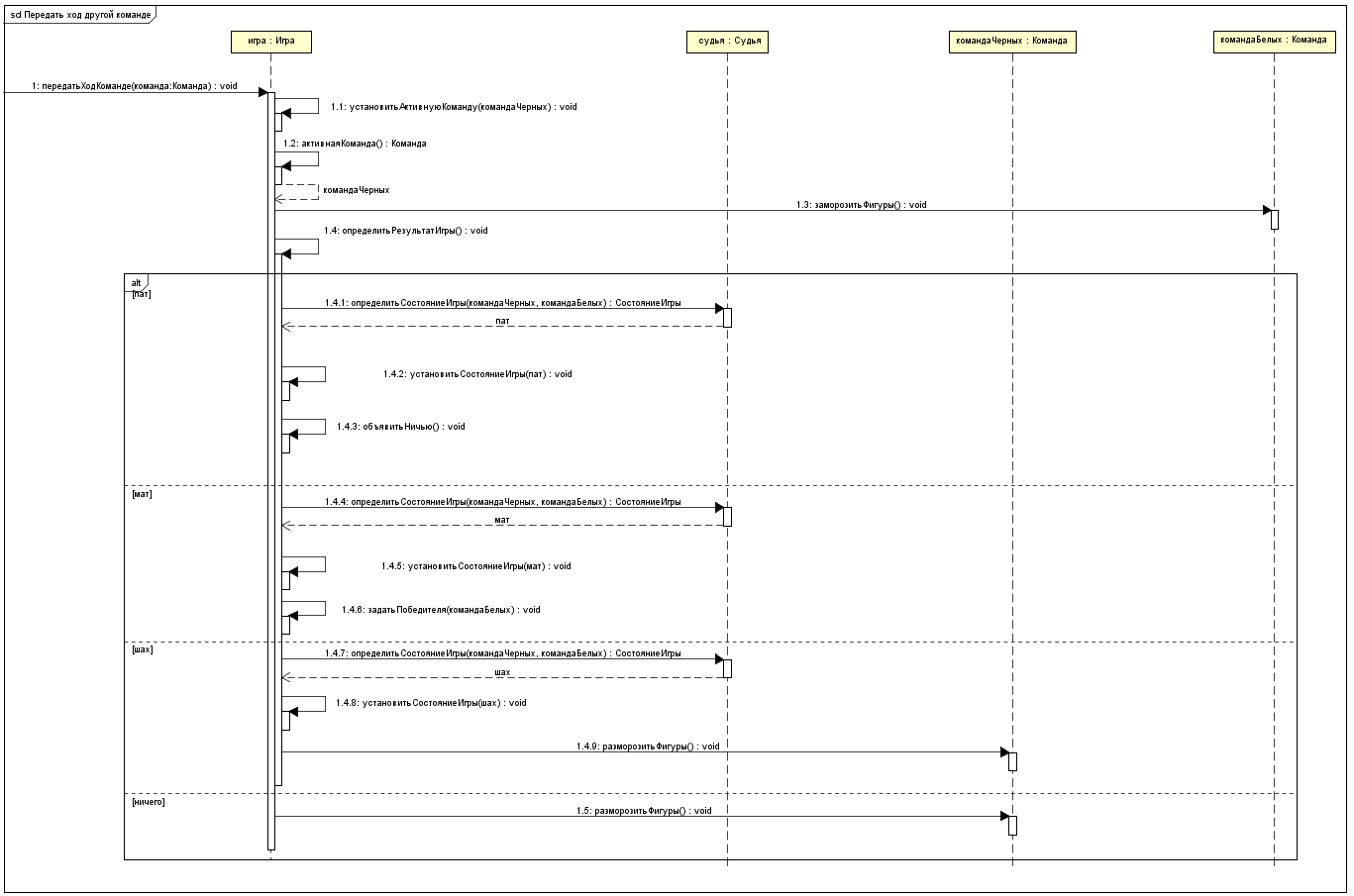


Рисунок 11 - Диаграмма последовательности для вычислительной модели.

Передать ход другой команде

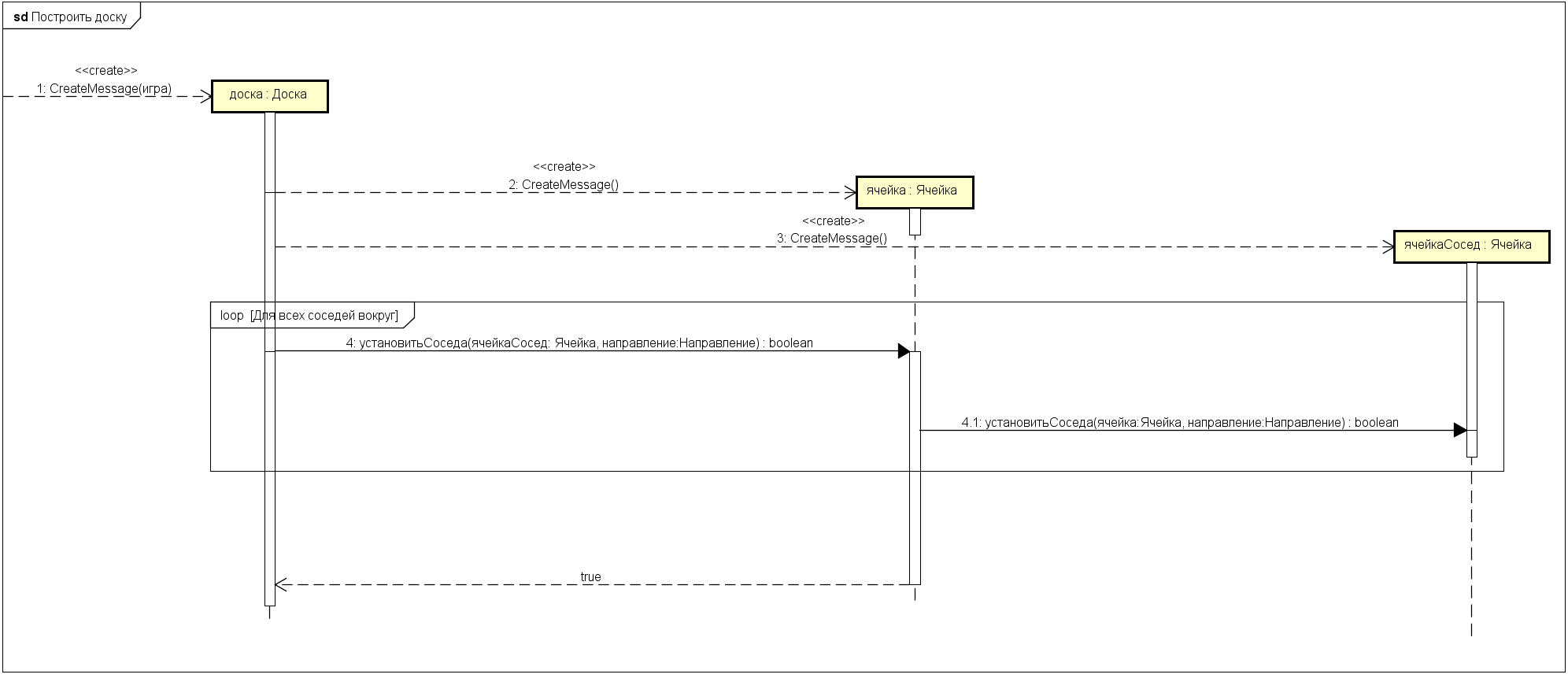


Рисунок 11 – Диаграмма последовательности для вычислительной модели.

Построить доску

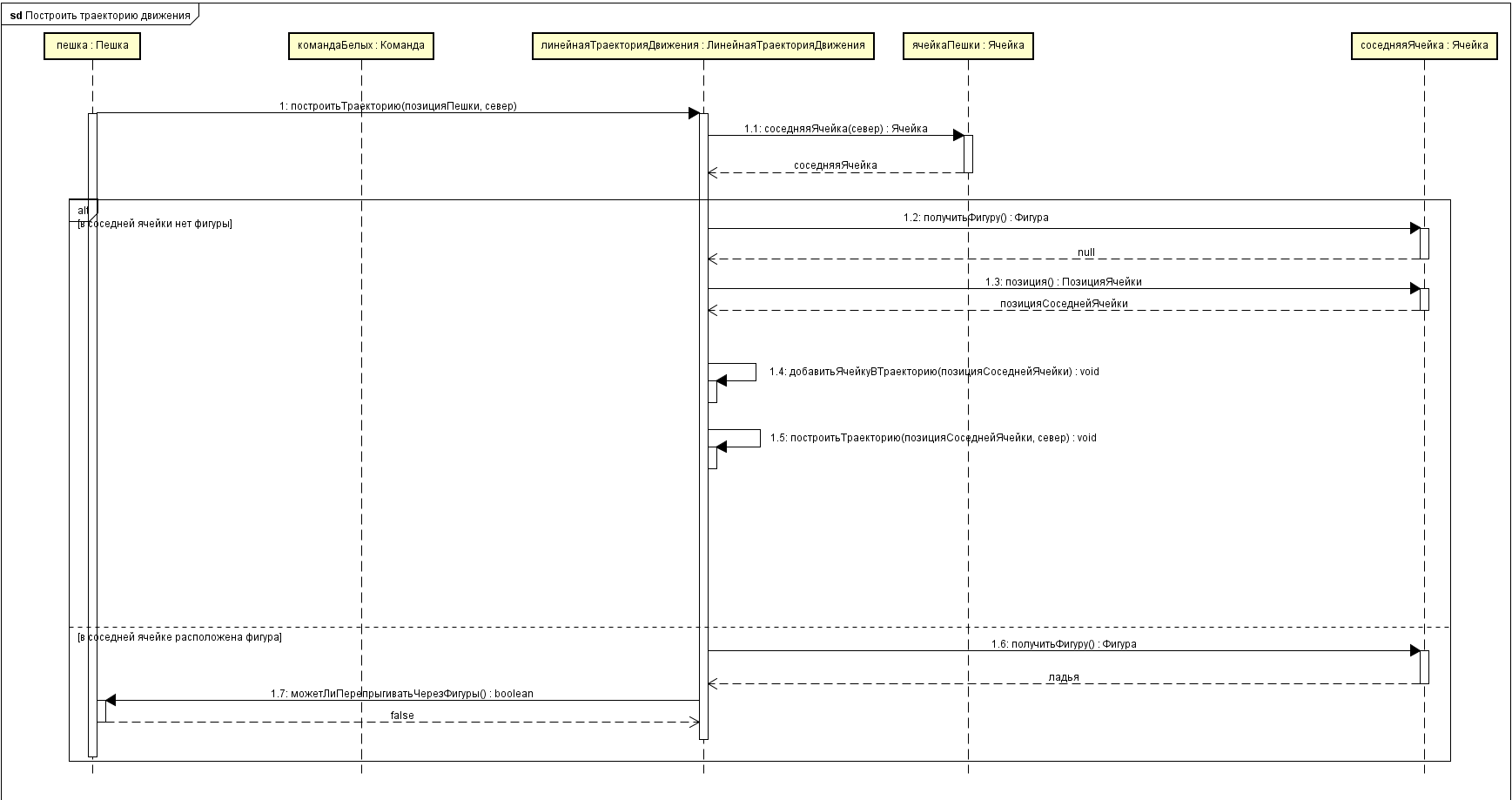


Рисунок 12 – Диаграмма последовательности для вычислительной модели.

Построить траекторию движения

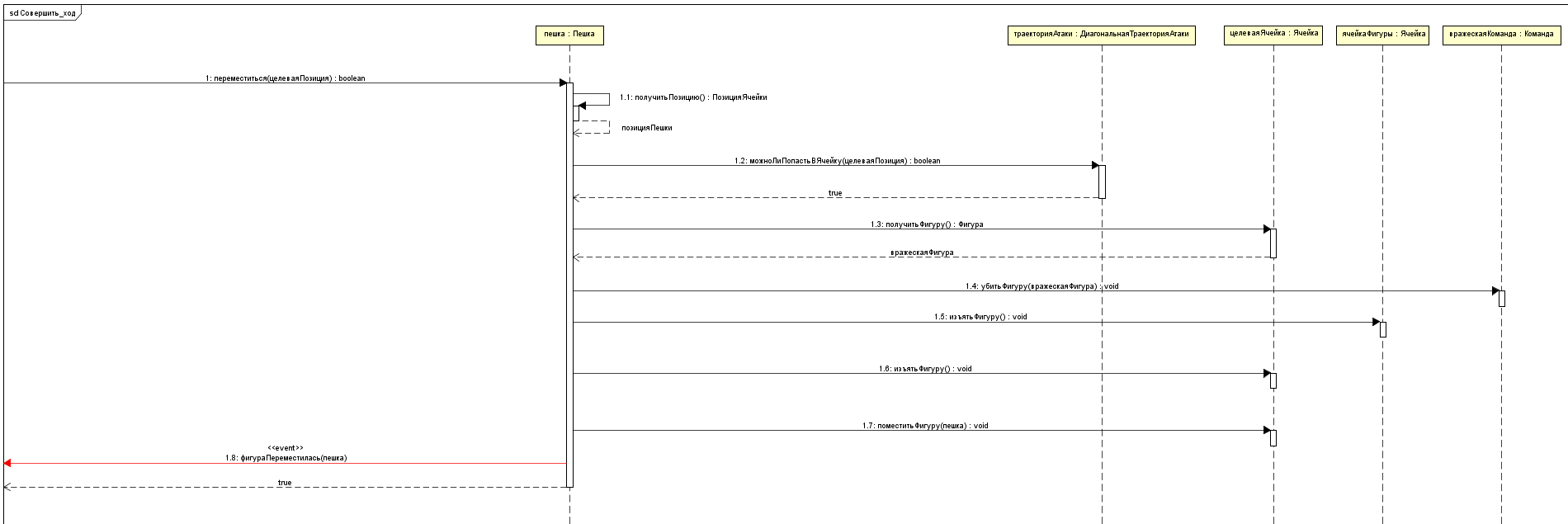


Рисунок 13 - Диаграмма последовательности для вычислительной модели.

Совершить ход

## 3.6 Человеко-машинное взаимодействие

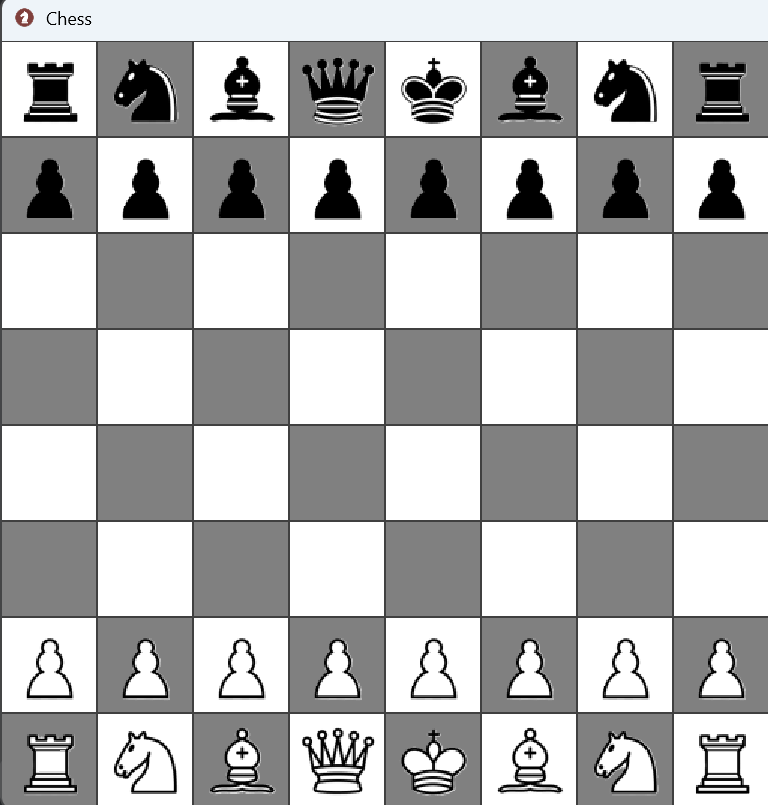
****

Рисунок 14 – общий вид главного окна программы

## 3.7 Реализация ключевых классов

package model.Figures;

import model.Cell;

import model.CellPosition;

import model.Direction;

import model.Team;

import model.Trajectories.Trajectory;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Фигура.

\*/

public abstract class Figure {

/\*\*

\* Команда.

\*/

protected Team team;

/\*\*

\* Количество жизней

\*/

protected int lives;

/\*\*

\* Количество жизней, которые отнимаются при апргрейде

\*/

protected int upgradeDamage;

/\*\*

\* Может ли фигура "перепрыгивать" другие фигуры

\*/

protected boolean canJumpOver = false;

/\*\*

\* Ходила ли фигура в течение игры

\*/

protected boolean hasMoved = false;

/\*\*

\* Ячейка

\*/

protected Cell cell;

/\*\*

\* Направления движения фигуры

\*/

protected List<Direction> movementDirections;

/\*\*

\* Направления атаки фигуры

\*/

protected List<Direction> attackDirections;

/\*\*

\* Траектории атаки фигуры

\*/

protected List<Trajectory> attackTrajectories;

/\*\*

\* Траектория движения фигуры

\*/

protected List<Trajectory> movementTrajectories;

/\*\*

\* Радиус траектории движения

\*/

protected int movementRadius;

/\*\*

\* Радиус траектории атаки

\*/

protected int attackRadius;

/\*\*

\* Смещение за один шаг

\*/

protected int[][] shiftPerStep;

/\*\*

\* Состояние фигуры (заморожена/разморожена)

\*/

protected boolean frozen = false;

/\*\*

\* Заморозить фигуру

\*/

public void freeze(){

this.frozen = true;

}

/\*\*

\* Разморозить фигуру

\*/

public void unfreeze(){

this.frozen = false;

}

/\*\*

\* Получить состояние фигуры (заморожена/разморожена)

\* @return заморожена ли фигура

\*/

public boolean isFrozen(){

return this.frozen;

}

/\*\*

\* Получить команду.

\*/

public Team getTeam() {

return this.team;

}

protected Figure(Team team){

this.team = team;

}

/\*\*

\* Вернуть тип фигуры

\* @return тип фигуры

\*/

public abstract FiguresTypes getFigureType();

/\*\*

\* Вернуть фигуру, в которую фигура может сделать Upgrade

\* @return тип фигуры

\*/

public abstract FiguresTypes getUpgradeFigureType();

/\*\*

\* Задать ячейку фигуре

\* @param cell ячейка

\*/

public void setCell(Cell cell){

this.cell = cell;

if (this.cell.getFigure() == null){

this.cell.setFigure(this);

}

}

/\*\*

\* Удалить ячейку фигуры

\*/

public void unsetCell(){

Cell currentCell = this.cell;

this.cell = null;

if (currentCell.getFigure() == this){

currentCell.unsetFigure();

}

}

/\*\*

\* Задать команду фигуре

\* @param team команда

\*/

public void setTeam(Team team){

this.team = team;

if (!this.team.getFigureList().contains(this)){

this.team.addFigure(this);

}

}

/\*\*

\* Отвязать команду от фигуры

\*/

public void unsetTeam(){

Team currentTeam = this.team;

this.team = null;

if (currentTeam.getFigureList().contains(this)){

currentTeam.deleteFigure(this);

}

}

/\*\*

\* Получить количество жизней

\* @return количество жизней

\*/

public int getLives(){

return this.lives;

}

/\*\*

\* Нанесение урона себе

\* @param damage кол-во урона

\*/

public void takeDamage(int damage){

if (getLives() != Integer.MAX\_VALUE){

this.lives = lives - damage;

}

}

/\*\*

\* Получить урон, который наносится при апргрейде

\* @return урон

\*/

public int getUpgradeDamage(){

return this.upgradeDamage;

}

/\*\*

\* Жива ли фигура

\* @return жива ли фигура

\*/

public boolean isAlive(){

return getLives() > 0;

}

/\*\*

\* Задать количество жизней фигуре

\* @param hp - количество жизней

\*/

public void setLives(int hp){

this.lives = hp;

}

/\*\*

\* Получить траектории движения фигуры

\* @return траектория движения фигуры

\*/

public List<Trajectory> getMovementTrajectories(){

return this.movementTrajectories;

}

/\*\*

\* Получить траектории атаки фигуры

\* @return траектория атаки фигуры

\*/

public List<Trajectory> getAttackTrajectories(){

return this.attackTrajectories;

}

/\*\*

\* Получить список ячеек из всех траекторий

\* @return список всех доступных позиций для движения

\*/

public List<Cell> getAllCellsFromTrajectories(){

List<Cell> allCells = new ArrayList<>();

// Для всех траекторий движения

for (Trajectory trajectory: getMovementTrajectories()){

allCells.addAll(trajectory.getCells());

}

// Для всех траекторий атаки

for (Trajectory trajectory: getAttackTrajectories()){

allCells.addAll(trajectory.getCells());

}

return allCells;

}

/\*\*

\* Получить список позиций ячеек их всех траекторий

\* @return список позиций доступных ячеек для движения

\*/

public List<CellPosition> getAllCellsPositionsFromTrajectories(){

List<CellPosition> allCellsPositions = new ArrayList<>();

// Для всех траекторий движения

for (Trajectory trajectory: getMovementTrajectories()){

// Для всех ячеек траектории

for (Cell cell: trajectory.getCells()){

allCellsPositions.add(cell.getPosition());

}

}

// Для всех траекторий атаки

for (Trajectory trajectory: getAttackTrajectories()){

// Для всех ячеек траектории

for (Cell cell: trajectory.getCells()){

allCellsPositions.add(cell.getPosition());

}

}

return allCellsPositions;

}

/\*\*

\* Получить радиус траектории движения

\* @return радиус траектории движения

\*/

public int getMovementRadius(){

return this.movementRadius;

}

/\*\*

\* Получить радиус траектории атаки

\* @return радиус траектории атаки

\*/

public int getAttackRadius(){

return this.attackRadius;

}

/\*\*

\* Задать радиус траектории движения

\* @param movementRadius радиус движения

\*/

public void setMovementRadius(int movementRadius){

this.movementRadius = movementRadius;

}

/\*\*

\* Задать радиус траектории атаки

\* @param attackRadius радиус атаки

\*/

public void setAttackRadius(int attackRadius){

this.attackRadius = attackRadius;

}

/\*\*

\* Получить позицию фигуры

\* @return позиция фигуры

\*/

public CellPosition getFigurePosition(){

if (this.cell != null){

return this.cell.getPosition();

}

return null;

}

/\*\*

\* Получить ячейку фигуры

\* @return ячейка фигуры

\*/

public Cell getCell(){

return this.cell;

}

/\*\*

\* Может ли фигура перепрыгивать другие фигуры

\*/

public boolean canJumpOver(){

return this.canJumpOver;

}

/\*\*

\* Ходила ли фигура в течение игры

\* @return ходила ли фигура в течение игры

\*/

public boolean hasMoved(){

return this.hasMoved;

}

/\*\*

\* Задать состояние ходила ли фигура в течение игры

\* @param hasMoved двигалась ли фигура в течение игры

\*/

public void setHasMoved(boolean hasMoved){

this.hasMoved = hasMoved;

}

/\*\*

\* Переместиться в заданную ячейку

\* @param targetCell целевая ячейка

\* @return последний ход

\*/

public UndoableMove moveTo(Cell targetCell){

// Фигура заморожена

if (isFrozen()){

return null;

}

// Объединить траектории в один лист

List<Trajectory> trajectories = new ArrayList<>(getMovementTrajectories());

trajectories.addAll(getAttackTrajectories());

UndoableMove undoableMove = new UndoableMove(this, targetCell);

// Для всех траекторий движения

for (Trajectory trajectory: trajectories){

// Целевая ячейка есть в траектории

if (trajectory.getCells().contains(targetCell)){

undoableMove.move();

}

}

takeDamage(1);

return undoableMove;

}

/\*\*

\* Смерть

\*/

public void die(){

unsetCell();

unsetTeam();

}

public abstract Figure cloneFigure();

} package model.Trajectories;

import model.Board;

import model.Cell;

import model.CellPosition;

import model.Direction;

import model.Figures.Figure;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

/\*\*

\* Траектория.

\*/

public abstract class Trajectory {

/\*\*

\* Список ячеек в траектории

\*/

protected List<Cell> cells;

/\*\*

\* Список направлений

\*/

protected final List<Direction> directions;

/\*\*

\* Количество возможных шагов за ход

\*/

protected int stepsCount;

/\*\*

\* Смещение за один шаг по строкам и колоннам

\*/

protected final int[] shiftPerStep;

/\*\*

\* @param directions направления

\* @param stepsCount количество возможных шагов

\* @param shiftPerStep смещение за один шаг по строкам и колоннам

\*/

public Trajectory(List<Direction> directions, int stepsCount, int[] shiftPerStep){

this.cells = new ArrayList<>();

this.directions = directions;

this.stepsCount = stepsCount;

this.shiftPerStep = shiftPerStep;

}

/\*\*

\* Получить список ячеек траектории

\*/

public List<Cell> getCells(){

return this.cells;

}

/\*\*

\* Можно ли попасть в заданную ячейку

\* @param targetCell целевая ячейка

\* @return можно ли переместиться

\*/

public boolean canGoToCell(Cell targetCell){

// Для всех ячеек траектории

for (Cell cell: this.cells){

// Ячейка есть в списке

if (cell.getPosition() == targetCell.getPosition()){

return true;

}

}

return false;

}

/\*\*

\* Построить траекторию

\* @param startCell стартовая клетка

\*/

public void buildTrajectory(Cell startCell){

// Очистка прошлой траектории

this.cells.clear();

// Можно ли перепрыгивать другие объекты на доске

boolean canJumpOver = startCell.getFigure().canJumpOver();

// Для всех направлений

for (Direction direction: this.directions){

if (stepsCount > 0) {

// Получить позицию в заданном направлении

int rowAfterStep = direction.getDeltaRow() \* shiftPerStep[0] + startCell.getPosition().getRow();

int colAfterStep = direction.getDeltaCol() \* shiftPerStep[1] + startCell.getPosition().getCol();

CellPosition positionAfterStep = new CellPosition(rowAfterStep, colAfterStep);

// Заданная позиция не выходит за пределы доски

if (Board.isInsideTheBoard(rowAfterStep, colAfterStep)) {

//Получить соседа в заданном направлении

Cell neighbourCell = findCellAt(startCell, new HashSet<>(), positionAfterStep);

buildDirectionTrajectory(neighbourCell, direction, this.stepsCount, canJumpOver);

}

}

}

}

/\*\*

\* Построить траекторию в заданном направлении

\* @param cell текущая клетка

\* @param direction направление

\* @param steps количество доступных шагов

\* @param canJumpOver можно ли перепрыгивать другие объекты на доске

\*/

private void buildDirectionTrajectory(Cell cell, Direction direction, int steps, boolean canJumpOver){

// Получить фигуру в ячейке

Figure figure = cell.getFigure();

// Уменьшаем количество доступных шагов

int remindSteps = steps - 1;

// Добавляем ячейку в траекторию

addCell(cell);

// Если нет фигуры, либо можно перепрыгнуть её

if (figure == null || canJumpOver) {

// Получить позицию в заданном направлении

int rowAfterStep = direction.getDeltaRow() \* this.shiftPerStep[0] + cell.getPosition().getRow();

int colAfterStep = direction.getDeltaCol() \* this.shiftPerStep[1] + cell.getPosition().getCol();

CellPosition positionAfterStep = new CellPosition(rowAfterStep, colAfterStep);

// Заданная позиция не выходит за пределы доски и остались ещё шаги

if (Board.isInsideTheBoard(rowAfterStep, colAfterStep) && remindSteps > 0){

// Строим траекторию в заданном направлении

Cell neighbourCell = findCellAt(cell, new HashSet<>(),positionAfterStep);

buildDirectionTrajectory(neighbourCell, direction, remindSteps, canJumpOver);

}

}

}

/\*\*

\* Найти ячейку в заданном направлении

\* @param currentCell текущая ячейка

\* @param visitedNeighbours уже проверенные ячейки

\* @param findPosition искомая позиция

\* @return ячейка с заданной позицией

\*/

private Cell findCellAt(Cell currentCell, Set<CellPosition> visitedNeighbours, CellPosition findPosition){

// Текущая ячейка не существует

if (currentCell == null){

return null;

}

// Ячейка уже проверялась

if (visitedNeighbours.contains(currentCell.getPosition())){

return null;

}

visitedNeighbours.add(currentCell.getPosition());

// Нашли искомую ячейку

if (currentCell.getPosition().equals(findPosition)){

return currentCell;

}

// Для всех соседей текущей ячейки

for (Cell neighbour: currentCell.getNeighbours().values()){

Cell found = findCellAt(neighbour, visitedNeighbours, findPosition);

if (found != null){

return found;

}

}

return null;

}

/\*\*

\* Добавить ячейку в траекторию

\* @param cell позиция ячейки

\*/

private void addCell(Cell cell){

this.cells.add(cell);

}

/\*\*

\* Удалить позиции ячеек из траектории

\* @param cellsToRemove список ячеек для удаления

\*/

public void deleteCells(List<Cell> cellsToRemove){

this.cells.removeAll(cellsToRemove);

}

/\*\*

\* Задать количество возможных шагов за ход

\* @param stepsCount количество возможных шагов за ход

\*/

public void setStepsCount(int stepsCount){

this.stepsCount = stepsCount;

}

}

package model;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* Доска.

\*/

public class Board {

/\*\*

\* Размер доски

\*/

private final static int BOARD\_SIZE = 8;

/\*\*

\* Список ячеек.

\*/

private final Map<CellPosition, Cell> cells;

public Board(){

this.cells = new HashMap<>();

for (int i = 0; i < BOARD\_SIZE; i++){

for (int j = 0; j < BOARD\_SIZE; j++){

CellPosition cellPosition = new CellPosition(i,j);

Cell cell = new Cell(cellPosition);

this.cells.putIfAbsent(cellPosition,cell);

}

}

// Связываем ячейки

assignNeighbours();

}

/\*\*

\* Связывание ячеек доски

\*/

private void assignNeighbours(){

// Перебираем все клетки доски

for (Cell cell: this.cells.values()){

// Перебираем всех возможных соседей клетки

for(Direction direction: Direction.values()){

int neighbourRow = cell.getPosition().getRow() + direction.getDeltaRow();

int neighbourCol = cell.getPosition().getCol() + direction.getDeltaCol();

// Входит ли соседняя ячейка в пределы доски

if (isInsideTheBoard(neighbourRow, neighbourCol)){

// Получаем ячейку соседа

Cell neighbourCell = getCellByPosition(new CellPosition(neighbourRow, neighbourCol));

// Добавляем соседа для ячейки

if (neighbourCell != null){

cell.setNeighbour(neighbourCell, direction);

}

}

}

}

}

/\*\*

\* Получить список ячеек.

\* @return список ячеек

\*/

public Map<CellPosition, Cell> getCells() {

return cells;

}

/\*\*

\* Получить ячейку по позиции

\* @param cellPosition позиция искомой ячейки

\* @return искомая ячейка

\*/

public Cell getCellByPosition(CellPosition cellPosition){

return this.cells.get(cellPosition);

}

/\*\*

\* Получить размер доски

\* @return размер доски

\*/

public static int getBoardSize(){

return BOARD\_SIZE;

}

/\*\*

\* Проверка на выход за пределы доски

\* @param row строка

\* @param col колонка

\* @return выходит ли позиция за пределы доски

\*/

public static boolean isInsideTheBoard(int row, int col){

return (row >= 0 && row < BOARD\_SIZE && col >= 0 && col < BOARD\_SIZE);

}

}

package model;

import model.Figures.Figure;

import model.Figures.FiguresFactory;

import model.Figures.FiguresTypes;

import model.events.\*;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CopyOnWriteArrayList;

/\*\*

\* Игра.

\*/

public class Game {

/\*\*

\* Доска

\*/

private Board board;

/\*\*

\* Судья

\*/

private Judge judge;

/\*\*

\* Активная команда

\*/

private Team activeTeam;

/\*\*

\* Неактивная команда

\*/

private Team inactiveTeam;

/\*\*

\* Активная фигура

\*/

private Figure activeFigure;

/\*\*

\* Список команд

\*/

private final List<Team> teams;

/\*\*

\* Список слушателей изменений фигур

\*/

private final List<FigureActionListener> figureListeners;

/\*\*

\* Список слушателей результатов игры

\*/

private final List<GameStatusActionListener> gameStatusListeners;

/\*\*

\* Список слушателей апргрейда фигуры

\*/

private final List<FigureUpgradedListener> figureUpgradedListeners;

public Game(){

this.teams = new ArrayList<>();

this.figureListeners = new CopyOnWriteArrayList<>();

this.gameStatusListeners = new CopyOnWriteArrayList<>();

this.figureUpgradedListeners = new CopyOnWriteArrayList<>();

}

/\*\*

\* Начало игры

\*/

public void start(){

// Создание доски, расстановки и судьи

this.board = new Board();

Placement placement = new Placement(this.board);

this.judge = new Judge(this.board);

// Инициализация команд

Team white = placement.getWhiteTeam();

Team black = placement.getBlackTeam();

this.teams.clear();

this.teams.add(white);

this.teams.add(black);

// Выбор того, кто ходит первым

this.activeTeam = white;

this.inactiveTeam = black;

// Размораживаем фигуры

unfreezeAll();

}

/\*\*

\* Выбор активной фигуры

\* @param pos позиция фигуры

\*/

public void onFigureSelected(CellPosition pos){

// Игнорировать, если выбрана фигура неактивной команды

Cell figureCell = board.getCellByPosition(pos);

Figure figure = figureCell.getFigure();

if (figure == null || figure.getTeam() != activeTeam || figure.isFrozen()){

clearSelection();

return;

}

// Запоминаем выбранную фигуру

this.activeFigure = figure;

// Оповещаем GUI о выборе фигуры

fireFigureActivated(activeFigure);

}

/\*\*

\* Сделать ход активной фигуры

\* @param targetCellPos целевая позиция ячейки

\*/

public void onFigureMoved(CellPosition targetCellPos){

// Не было активной фигуры

if (this.activeFigure == null){

return;

}

// Кликнули по фигуре своей команды

if (board.getCellByPosition(targetCellPos).getFigure() != null) {

if (board.getCellByPosition(targetCellPos).getFigure().getTeam() == this.activeTeam) {

clearSelection();

onFigureSelected(targetCellPos);

return;

}

}

// Кликнули по ячейке, которая не входит в траекторию фигуры

Cell targetCell = board.getCellByPosition(targetCellPos);

if (!this.activeFigure.getAllCellsFromTrajectories().contains(targetCell)){

clearSelection();

return;

}

// Двигаем фигуру

Cell fromCell = this.activeFigure.getCell();

this.activeTeam.moveFigure(targetCell, this.activeFigure);

// Оповещаем GUI о перемещении фигуры

fireFigureMoved(this.activeFigure, fromCell, targetCell);

// Если фигура истратила свои жизни при перемещении

if (!this.activeFigure.isAlive()){

// Убиваем фигуру и оповещаем GUI о её смерти

CellPosition pos = this.activeFigure.getCell().getPosition();

this.activeFigure.die();

fireFigureDeadFromItself(pos);

}

// Передать ход другой команде

clearSelection();

changeTeam();

}

/\*\*

\* Заморозить все фигуры

\*/

private void freezeAll(){

// Для каждой команды

for (Team team: this.teams){

// Для всех фигур команды

for (Figure figure: team.getFigureList()){

// Заморозить фигуру

figure.freeze();

}

}

}

/\*\*

\* Разморозить все фигуры

\*/

private void unfreezeAll(){

// Для каждой команды

for (Team team: this.teams){

// Для всех фигур команды

for (Figure figure: team.getFigureList()){

// Разморозить фигуру

figure.unfreeze();

}

}

}

/\*\*

\* Деактивация фигуры

\*/

private void clearSelection(){

if (this.activeFigure != null){

fireFigureDeactivated(this.activeFigure);

}

this.activeFigure = null;

}

/\*\*

\* Передать ход другой команде

\*/

private void changeTeam(){

// Заморозка фигур

freezeAll();

// Определить состояние игры

GameStatus gameStatus = this.judge.determineGameStatus(this.activeTeam, this.inactiveTeam);

// Был объявлен шах

if (gameStatus == GameStatus.CHECK){

fireCheck(this.inactiveTeam);

}

// Был объявлен пат

if (gameStatus == GameStatus.STALEMATE){

fireGameDrawn();

return;

}

// Был объявлен мат

if (gameStatus == GameStatus.CHECKMATE){

fireGameWon(this.activeTeam);

return;

}

// Передать ход другой команде

int activeTeamIndex = this.teams.indexOf(this.activeTeam);

this.inactiveTeam = this.activeTeam;

this.activeTeam = this.teams.get((activeTeamIndex + 1) % this.teams.size());

unfreezeAll();

}

/\*\*

\* Регистрация слушателей изменения состояния фигур

\* @param l слушатель

\*/

public void addFigureActionListener(FigureActionListener l){

figureListeners.add(l);

}

/\*\*

\* Удаление слушателей изменения состояния фигур

\* @param l слушатель

\*/

public void removeFigureActionListener(FigureActionListener l){

figureListeners.remove(l);

}

/\*\*

\* Регистрация слушателей результата игры

\* @param l слушатель

\*/

public void addGameFinishActionListener(GameStatusActionListener l){

gameStatusListeners.add(l);

}

/\*\*

\* Удаление слушателей результата игры

\* @param l слушатель

\*/

public void removeGameFinishActionListener(GameStatusActionListener l){

gameStatusListeners.remove(l);

}

/\*\*

\* Добавление слушателей апргрейда фигур

\*/

public void addFigureUpgradedListener(FigureUpgradedListener l){

figureUpgradedListeners.add(l);

}

/\*\*

\* Удаление слушателей апргрейда фигур

\*/

public void removeFigureUpgradedListener(FigureUpgradedListener l){

figureUpgradedListeners.remove(l);

}

/\*\*

\* Рассылка событий активации фигуры

\* @param figure фигура

\*/

private void fireFigureActivated(Figure figure){

FigureActivatedEvent event = new FigureActivatedEvent(this, figure);

for (var l: this.figureListeners){

l.figureActivated(event);

}

}

/\*\*

\* Рассылка событий деактивации фигуры

\* @param figure фигура

\*/

private void fireFigureDeactivated(Figure figure){

FigureClearSelectedEvent event = new FigureClearSelectedEvent(this, figure);

for (var l: this.figureListeners){

l.figureDeactivated(event);

}

}

/\*\*

\* Рассылка событий перемещения фигуры

\* @param figure фигура

\* @param from ячейка, из которой фигура переместилась

\* @param to ячейка, в которую фигура переместилась

\*/

private void fireFigureMoved(Figure figure, Cell from, Cell to){

FigureMovedEvent event = new FigureMovedEvent(this, figure, from, to);

for (var l: this.figureListeners){

l.figureMoved(event);

}

}

/\*\*

\* Рассылка событий смерти фигуры

\* @param figurePosition фигура

\*/

private void fireFigureDeadFromItself(CellPosition figurePosition){

FigureDeadFromItselfEvent event = new FigureDeadFromItselfEvent(this, figurePosition);

for (var l: this.figureListeners){

l.figureDeadFromItself(event);

}

}

/\*\*

\* Рассылка событий завершения игры в ничью

\*/

private void fireGameDrawn(){

GameStatusDrawEvent event = new GameStatusDrawEvent(this);

for (var l: this.gameStatusListeners){

l.gameFinishInDraw(event);

}

}

/\*\*

\* Рассылка событий завершения игры с победителем

\* @param winnerTeam команда победитель

\*/

private void fireGameWon(Team winnerTeam){

GameStatusWinnerEvent event = new GameStatusWinnerEvent(this, winnerTeam);

for (var l: this.gameStatusListeners){

l.gameFinishWithWinner(event);

}

}

/\*\*

\* Рассылка событий постановки шаха команде

\* @param teamWithCheck команда, которой объявлен шах

\*/

private void fireCheck(Team teamWithCheck){

GameStatusCheckEvent event = new GameStatusCheckEvent(this, teamWithCheck);

for (var l: this.gameStatusListeners){

l.teamHasCheck(event);

}

}

/\*\*

\* Рассылка событий апргейда фигур

\*/

private void fireUpgrade(){

for (var l: this.figureUpgradedListeners){

l.upgradedFigure();

}

}

/\*\*

\* Получить доску

\* @return доска

\*/

public Board getBoard(){

return this.board;

}

/\*\*

\* Работа с фигурой

\* @param cellPosition позиция ячейки

\*/

public void workWithFigure(CellPosition cellPosition){

if (this.activeFigure == null){

onFigureSelected(cellPosition);

}

else{

onFigureMoved(cellPosition);

}

}

/\*\*

\* Upgrade фигуры

\*/

public void upgradeFigure(){

// Уже был апгрейд или нет активной фигуры

if (this.activeTeam.isUpgraded() || this.activeFigure == null){

return;

}

// Получить тип фигуры, в который делаем апгрейд

FiguresTypes upgradeFigureType = this.activeFigure.getUpgradeFigureType();

// Фигура ни в кого не может быть апгрейднута

if (upgradeFigureType == null){

return;

}

// Запоминаем ячейку активной фигуры и её количество жизней после апргрейда

Cell activeFigureCell = activeFigure.getCell();

int activeFigureHp = activeFigure.getLives() - activeFigure.getUpgradeDamage();

// "Убиваем" активную фигуру и создаем новую

this.activeFigure.die();

Figure figure = FiguresFactory.createFigure(upgradeFigureType, this.activeTeam);

figure.setLives(activeFigureHp);

figure.setCell(activeFigureCell);

figure.setTeam(this.activeTeam);

this.activeTeam.setUpgraded(true);

// Рассылаем события об апгрейде фигуры

fireUpgrade();

// Передаем ход другой команде

clearSelection();

changeTeam();

}

}

package model;

import model.Figures.Figure;

import model.Figures.UndoableMove;

import model.Trajectories.Trajectory;

import java.awt.\*;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Судья.

\*/

public class Judge {

/\*\*

\* Доска

\*/

private final Board board;

public Judge(Board board){

this.board = board;

}

/\*\*

\* Определить состояние игры

\* @param activeTeam активная команда

\* @param inactiveTeam неактивная команда

\* @return состояние игры

\*/

public GameStatus determineGameStatus(Team activeTeam, Team inactiveTeam){

// Для всех фигур активной команды

for (Figure figure: activeTeam.getFigureList()){

// Перестроить траектории атаки

for (Trajectory trajectory: figure.getAttackTrajectories()){

trajectory.buildTrajectory(figure.getCell());

}

editTrajectories(figure, activeTeam);

}

// Для всех фигур неактивной команды

for (Figure figure: inactiveTeam.getFigureList()){

// Перестроить траектории движения и атаки

for (Trajectory trajectory: figure.getMovementTrajectories()){

trajectory.buildTrajectory(figure.getCell());

}

for (Trajectory trajectory: figure.getAttackTrajectories()){

trajectory.buildTrajectory(figure.getCell());

}

editTrajectories(figure, inactiveTeam);

}

// Может ли активная команда атаковать короля неактивной

boolean canAttack = activeTeam.canAttackEnemyKing(inactiveTeam.getKingCell());

// Есть ли у неактивной команды доступные ходы

boolean hasAvailableMoves = inactiveTeam.hasFigureWithAvailableMoves();

// У команды нет доступных ходов и её королю угрожают

if (canAttack && !hasAvailableMoves){

return GameStatus.CHECKMATE; // Объявлен мат

}

// У команды есть доступные ходы, но её королю угрожают

if (canAttack){

return GameStatus.CHECK; // Объявлен шах

}

// У команды нет доступных ходов и её королю не угрожают

if (!hasAvailableMoves) {

return GameStatus.STALEMATE; // Объявлен пат

}

return GameStatus.GAME\_IS\_ON;

}

/\*\*

\* Редактировать траекторию

\* @param figureActiveTeam фигура активной команды

\* @param activeTeam активная команда

\*/

private void editTrajectories(Figure figureActiveTeam, Team activeTeam){

// Объединить траектории в один лист

List<Trajectory> trajectories = new ArrayList<>(figureActiveTeam.getMovementTrajectories());

trajectories.addAll(figureActiveTeam.getAttackTrajectories());

// Клон текущего состояния игры

GameSnapshot gameSnapshot = GameSnapshotFactory.makeSnapshot(this.board);

Board clonedBoard = gameSnapshot.board(); // Клонированная доска

Team clonedActiveTeam;

Team clonedInactiveTeam;

// Клонируем активную и неактивную команду

if (activeTeam.getColor() == Color.WHITE){

clonedActiveTeam = gameSnapshot.whiteTeam();

clonedInactiveTeam = gameSnapshot.blackTeam();

}

else{

clonedActiveTeam = gameSnapshot.blackTeam();

clonedInactiveTeam = gameSnapshot.whiteTeam();

}

// Получаем клонированную фигуру

Figure clonedFigure = clonedBoard.getCellByPosition(figureActiveTeam.getCell().getPosition()).getFigure();

// Для всех траекторий

for (Trajectory trajectory: trajectories){

// Для всех ячеек траектории

List<Cell> cellsToRemove = new ArrayList<>();

for (Cell currentCell: trajectory.getCells()){

// Перемещаем клонированную фигуру на ячейку клонированной доски

UndoableMove lastMove = clonedFigure.moveTo(clonedBoard.getCellByPosition(currentCell.getPosition()));

boolean isAttacked;

// Перестраиваем траектории атаки вражеской команды

clonedInactiveTeam.buildAttackTrajectories();

isAttacked = clonedInactiveTeam.canAttackEnemyKing(clonedActiveTeam.getKingCell());

// Король находится под атакой

if (isAttacked){

// Добавить ячейку в список удаляемых

cellsToRemove.add(currentCell);

}

// Возвращаем фигуру обратно

lastMove.undo();

}

// Удалить ячейки из траектории

trajectory.deleteCells(cellsToRemove);

}

}

}

## 3.8 Реализация ключевых тестовых случаев

@Test

void testKingTrajectories() {

King king = new King(whiteTeam);

placePiece(king);

// King movement = attack (same), eight neighbors

Set<CellPosition> positions = allPositions(king);

assertEquals(8, positions.size());

for (Direction d : Direction.values()) {

CellPosition expected = new CellPosition(

centerPos.getRow() + d.getDeltaRow(),

centerPos.getCol() + d.getDeltaCol()

);

assertTrue(positions.contains(expected), "Missing " + d);

}

}

@Test

void testBishopTrajectories() {

Bishop bishop = new Bishop(whiteTeam);

placePiece(bishop);

// Bishop movement and attack are same diagonals; count = 14

// bishop.getMovementTrajectories only movement, attack separate but same

// We'll test movement only: 4 diagonals sum distances

Set<CellPosition> mv = new HashSet<>();

for (Trajectory t : bishop.getMovementTrajectories()) {

t.buildTrajectory(bishop.getCell());

for (Cell c : t.getCells()) mv.add(c.getPosition());

}

assertEquals(13, mv.size());

}

@Test

void testRookTrajectories() {

Rook rook = new Rook(whiteTeam);

placePiece(rook);

// Rook movement: 4 directions cardinal, distances sum to 14

Set<CellPosition> mv = new HashSet<>();

for (Trajectory t : rook.getMovementTrajectories()) {

t.buildTrajectory(rook.getCell());

for (Cell c : t.getCells()) mv.add(c.getPosition());

}

assertEquals(14, mv.size());

}

@Test

void testQueenTrajectories() {

Queen queen = new Queen(whiteTeam);

placePiece(queen);

// Queen movement: rook + bishop = 28

Set<CellPosition> mv = new HashSet<>();

for (Trajectory t : queen.getMovementTrajectories()) {

t.buildTrajectory(queen.getCell());

for (Cell c : t.getCells()) mv.add(c.getPosition());

}

assertEquals(27, mv.size());

}

@Test

void testBoardSize() {

// Должно быть 8×8 = 64 ячейки

assertEquals(64, board.getCells().size(), "Board should contain 64 cells");

}

@Test

void testGetCellByPositionValid() {

CellPosition pos = new CellPosition(5, 2);

Cell cell = board.getCellByPosition(pos);

assertNotNull(cell, "Cell at (5,2) must exist");

assertEquals(pos, cell.getPosition());

}

@Test

void testGetCellByPositionInvalid() {

// Вне диапазона

assertNull(board.getCellByPosition(new CellPosition(-1, 0)));

assertNull(board.getCellByPosition(new CellPosition(0, 8)));

assertNull(board.getCellByPosition(new CellPosition(8, 8)));

}

@Test

void testIsInsideTheBoardBoundaries() {

assertFalse(Board.isInsideTheBoard(-1, 0));

assertFalse(Board.isInsideTheBoard(0, -1));

assertTrue(Board.isInsideTheBoard(0, 0));

assertTrue(Board.isInsideTheBoard(7, 7));

assertFalse(Board.isInsideTheBoard(8, 0));

assertFalse(Board.isInsideTheBoard(7, 8));

}

@Test

void testNeighboursInCenter() {

Cell center = board.getCellByPosition(new CellPosition(3, 3));

Map<Direction, Cell> n = center.getNeighbours();

// У центральной клетки должны быть все 8 направлений

assertEquals(8, n.size(), "Center should have 8 neighbours");

for (Direction d : Direction.values()) {

assertTrue(n.containsKey(d), "Missing direction " + d);

Cell neighbour = center.getNeighbour(d);

assertNotNull(neighbour, "Neighbour at " + d + " must not be null");

// сосед должен располагаться по смещению из enum

assertEquals(

3 + d.getDeltaRow(),

neighbour.getPosition().getRow(),

"Row mismatch for " + d

);

assertEquals(

3 + d.getDeltaCol(),

neighbour.getPosition().getCol(),

"Col mismatch for " + d

);

}

}

@Test

void testOnFigureSelectedInvalid() {

// Empty square at (4,4)

game.onFigureSelected(new CellPosition(4, 4));

assertEquals(0, figureListener.activatedCount);

// Black piece at (1,0) when white's turn

game.onFigureSelected(new CellPosition(1, 0));

assertEquals(0, figureListener.activatedCount);

}

@Test

void testOnFigureMovedValidAndTeamSwitch() {

// Select white pawn at (6,0)

CellPosition from = new CellPosition(6, 0);

game.onFigureSelected(from);

// Move to (5,0)

CellPosition to = new CellPosition(5, 0);

game.onFigureMoved(to);

// Moved event

assertEquals(1, figureListener.movedCount);

assertEquals(from, figureListener.lastFrom);

assertEquals(to, figureListener.lastTo);

// After move, activeTeam is black => selecting a black pawn now is valid

figureListener.reset();

game.onFigureSelected(new CellPosition(1, 0));

assertEquals(1, figureListener.activatedCount);

}

@Test

void testOnFigureMovedInvalid() {

// Select white pawn at (6,0)

game.onFigureSelected(new CellPosition(6, 0));

// Attempt invalid move (not in trajectory)

game.onFigureMoved(new CellPosition(4, 4));

assertEquals(0, figureListener.movedCount);

}

@Test

void testNoStatusEventOnNormalMove() {

game.onFigureSelected(new CellPosition(6, 0));

game.onFigureMoved(new CellPosition(5, 0));

assertEquals(0, statusListener.checkCount);

assertEquals(0, statusListener.drawCount);

assertEquals(0, statusListener.winCount);

}

@Test

void testCheck() {

Figure k1 = new King(white);

Figure k2 = new King(black);

Figure r1 = new Rook(white);

k1.setTeam(white);

k2.setTeam(black);

r1.setTeam(white);

// Black king at (7,7), white rook attacking along row

place(k1, 0, 0);

place(k2, 7, 7);

place(r1, 7, 5);

white.buildAttackTrajectories();

white.buildMovementTrajectories();

black.buildAttackTrajectories();

black.buildMovementTrajectories();

// Black has (6,7) or (7,6) free => only CHECK

assertEquals(GameStatus.CHECK,

judge.determineGameStatus(white, black));

}

@Test

void testCheckmate() {

Figure k1 = new King(white);

Figure k2 = new King(black);

Figure r1 = new Rook(white);

Figure r2 = new Rook(white);

k1.setTeam(white);

k2.setTeam(black);

r1.setTeam(white);

r2.setTeam(white);

// Black king flanked by two white rooks:

// one at (6,6), one at (6,7): king at (7,7) no escape

place(k1, 0, 0);

place(k2, 7, 7);

place(r1, 6, 6);

place(r2, 6, 7);

white.buildAttackTrajectories();

white.buildMovementTrajectories();

black.buildAttackTrajectories();

black.buildMovementTrajectories();

assertEquals(GameStatus.CHECKMATE,

judge.determineGameStatus(white, black));

}

@Test

void testStalemate() {

// Black king at (7,7), no check but no legal moves:

// white king at (5,6) and queen at (6,5) cover all adjacents

place(new King(white), 5, 6);

place(new Queen(white), 6, 5);

place(new King(black), 7, 7);

white.buildAttackTrajectories();

white.buildMovementTrajectories();

black.buildAttackTrajectories();

black.buildMovementTrajectories();

assertEquals(GameStatus.STALEMATE,

judge.determineGameStatus(white, black));

}

# 4 Вторая итерация разработки

## 4.1 Функциональные требования (сценарии)

авав

## 4.2 Словарь предметной области

авав

## 4.3 Структура программы на уровне классов

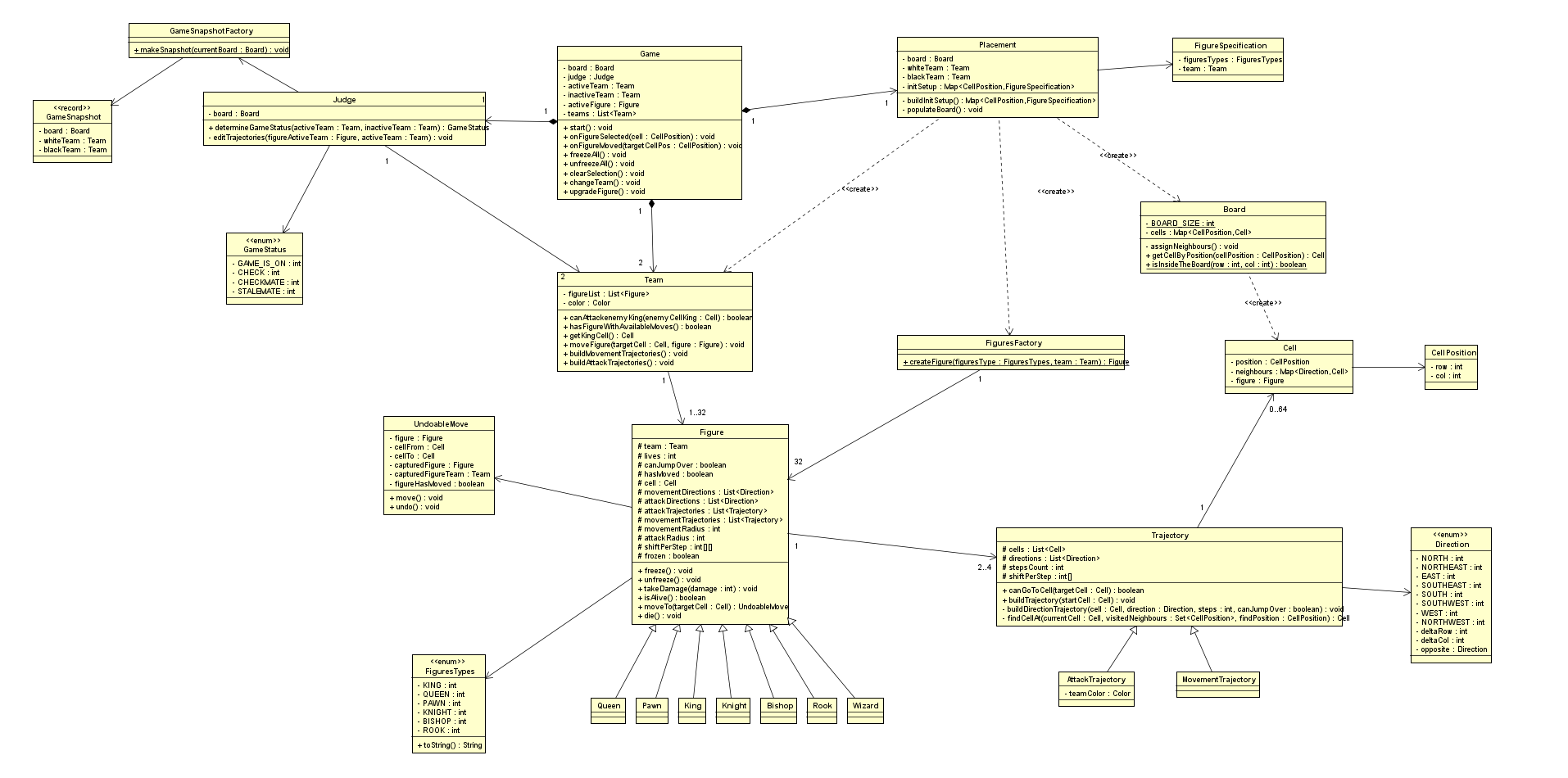


Рисунок 15 – Диаграмма классов вычислительной модели

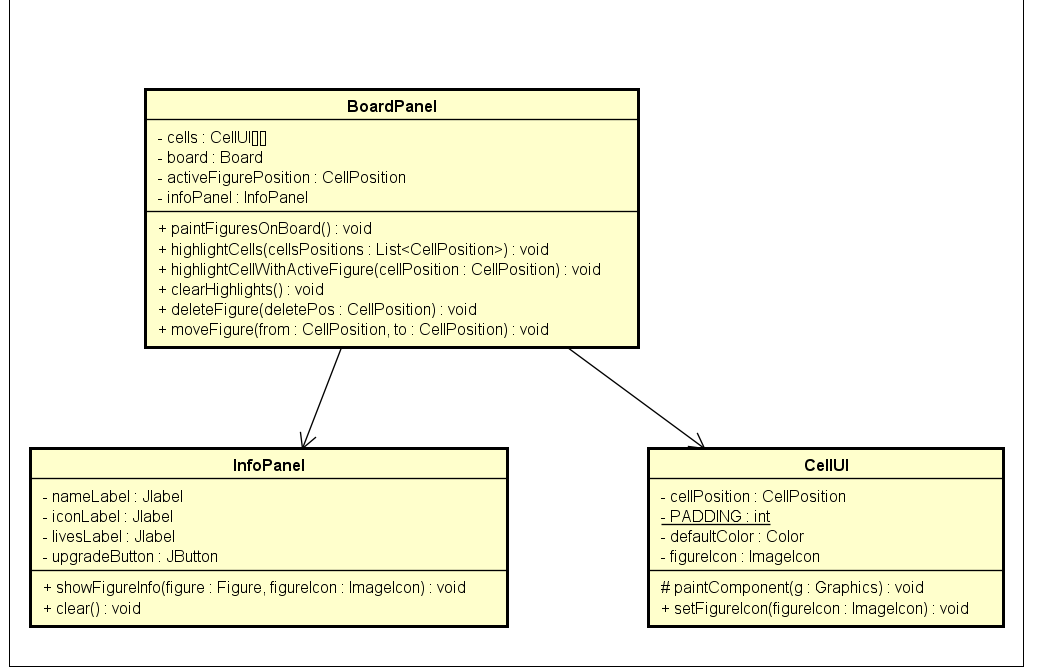


Рисунок 16 – Диаграмма классов представления

## 4.4 Типовые процессы в программе

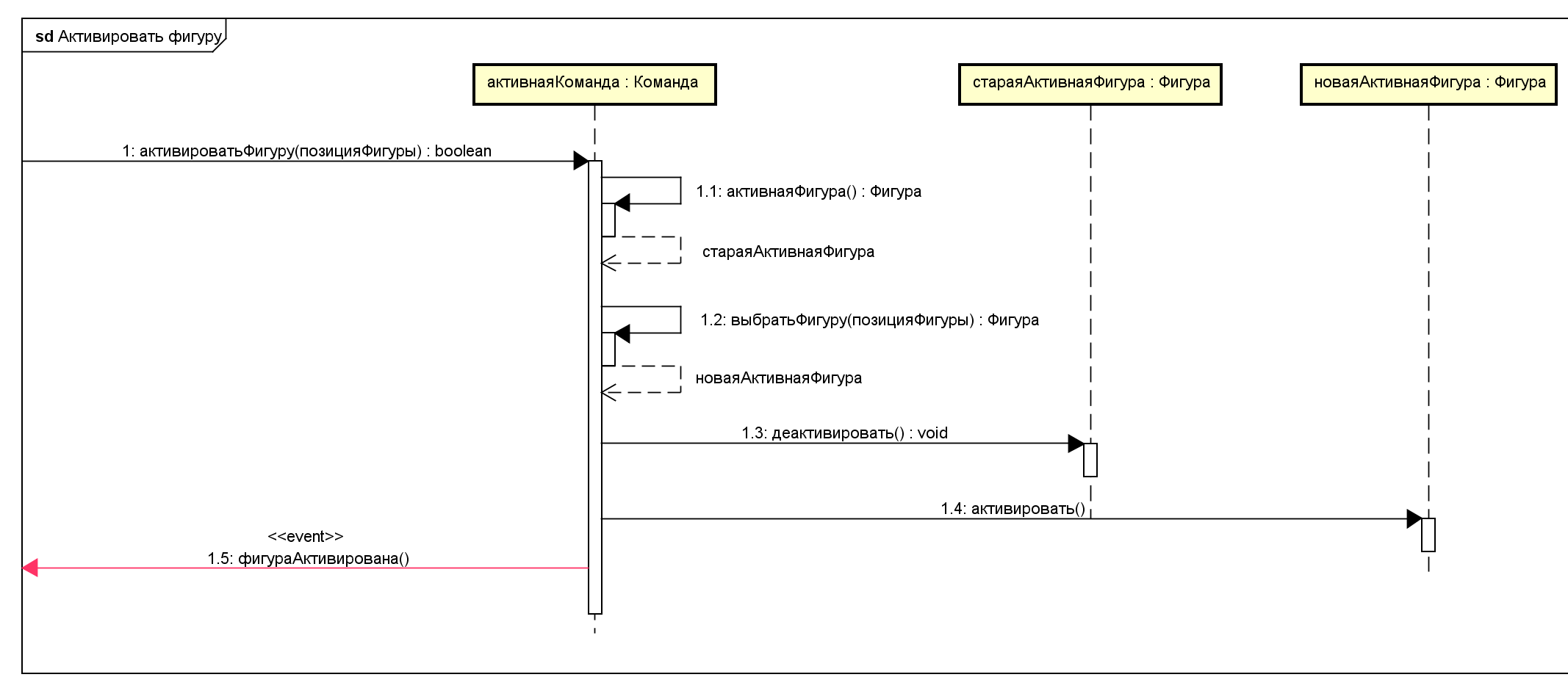


Рисунок 17 – Диаграмма последовательности для вычислительной модели. Активация фигуры.

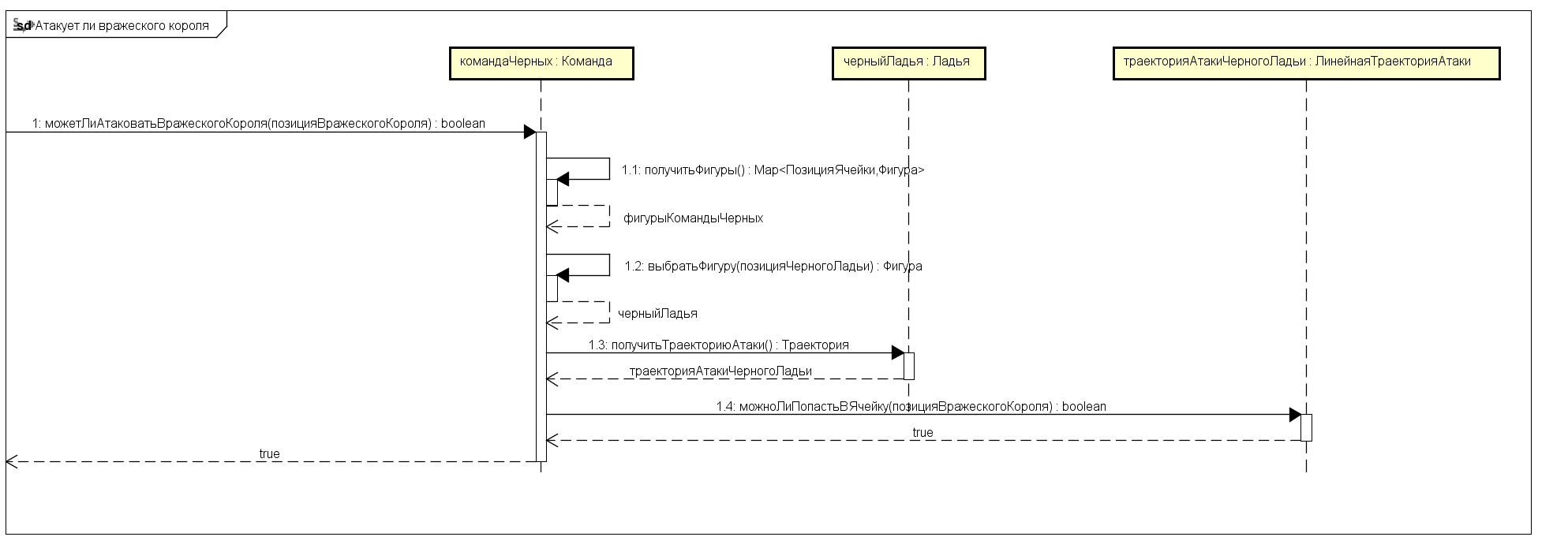


Рисунок 18 – Диаграмма последовательности для вычислительной модели.

Атакует ли вражеского короля.

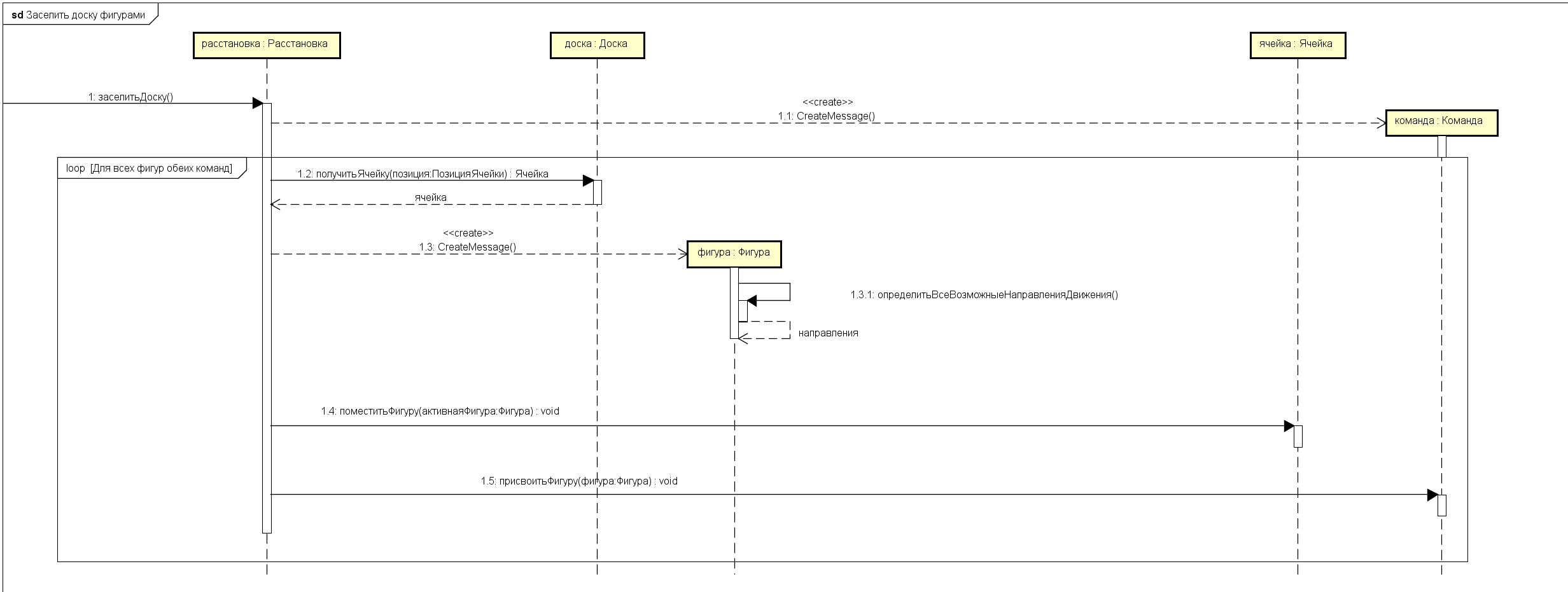


Рисунок 19 – Диаграмма последовательности для вычислительной модели.

Заселить доску фигурами

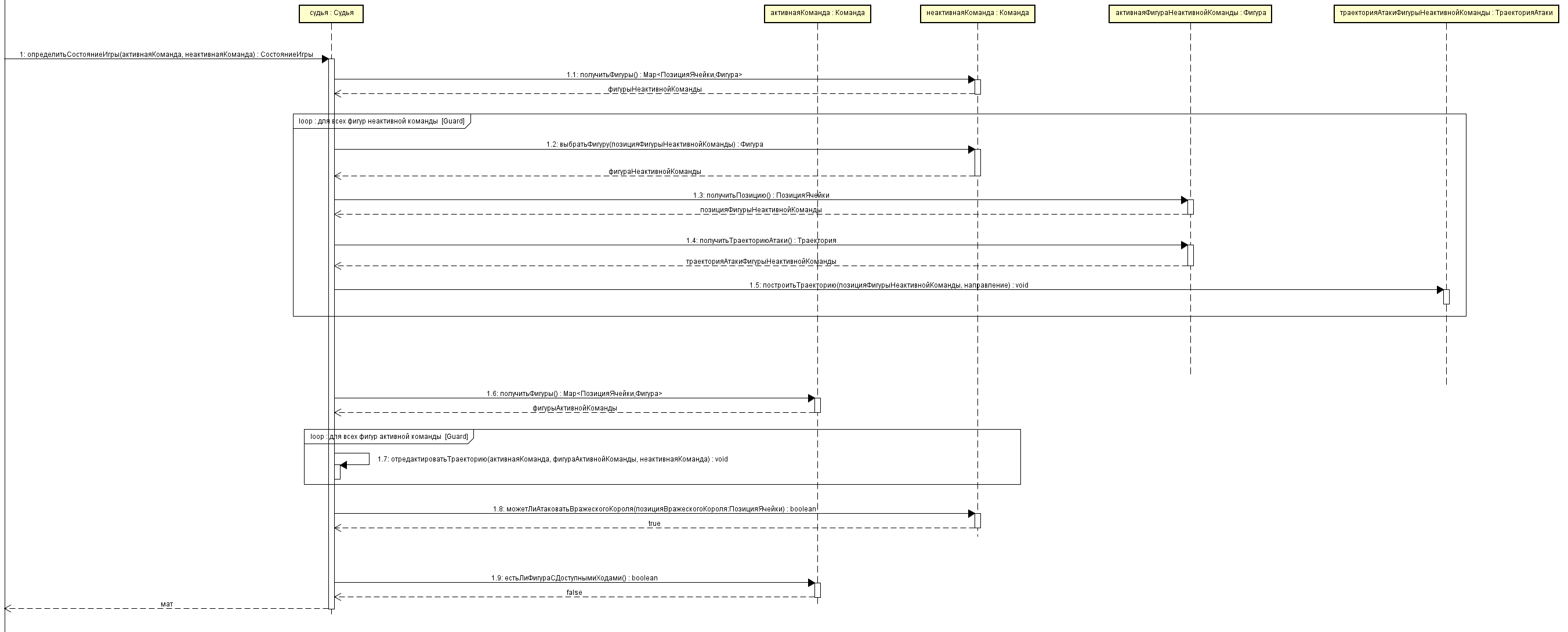


Рисунок 20 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – мат

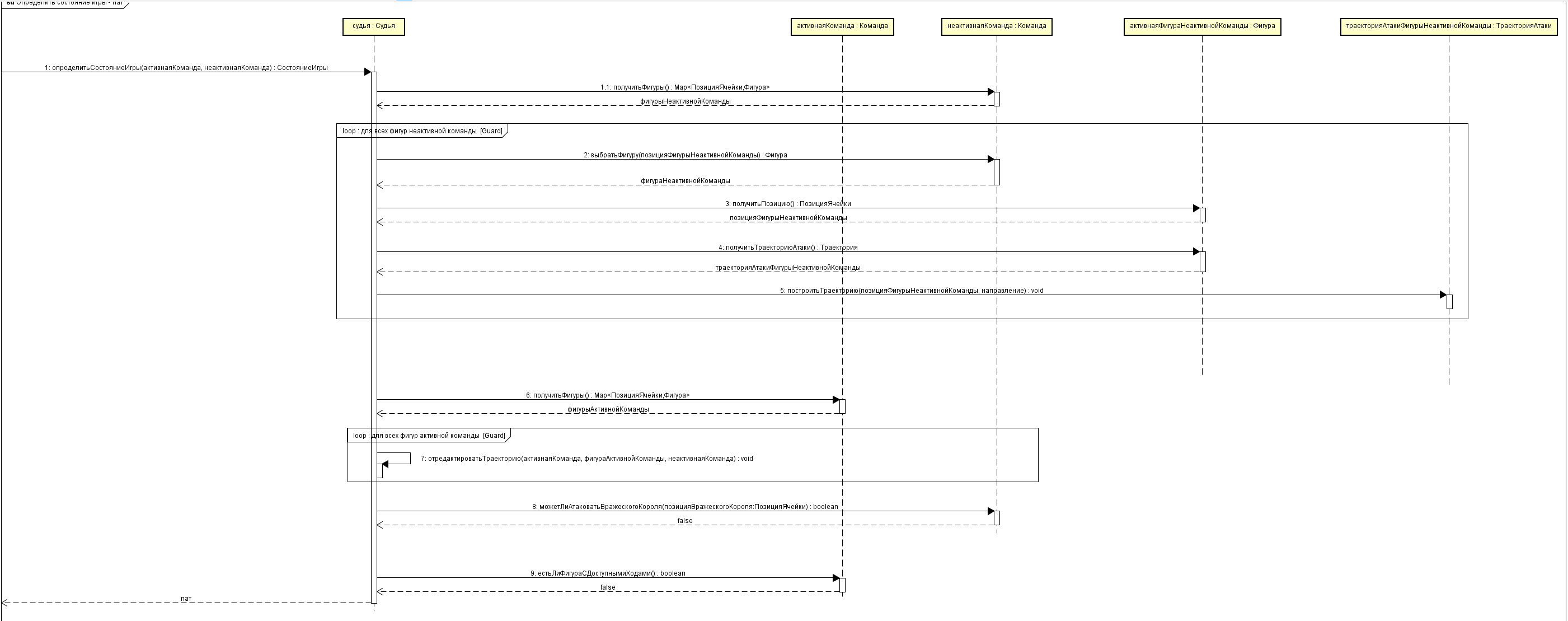


Рисунок 21 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – пат

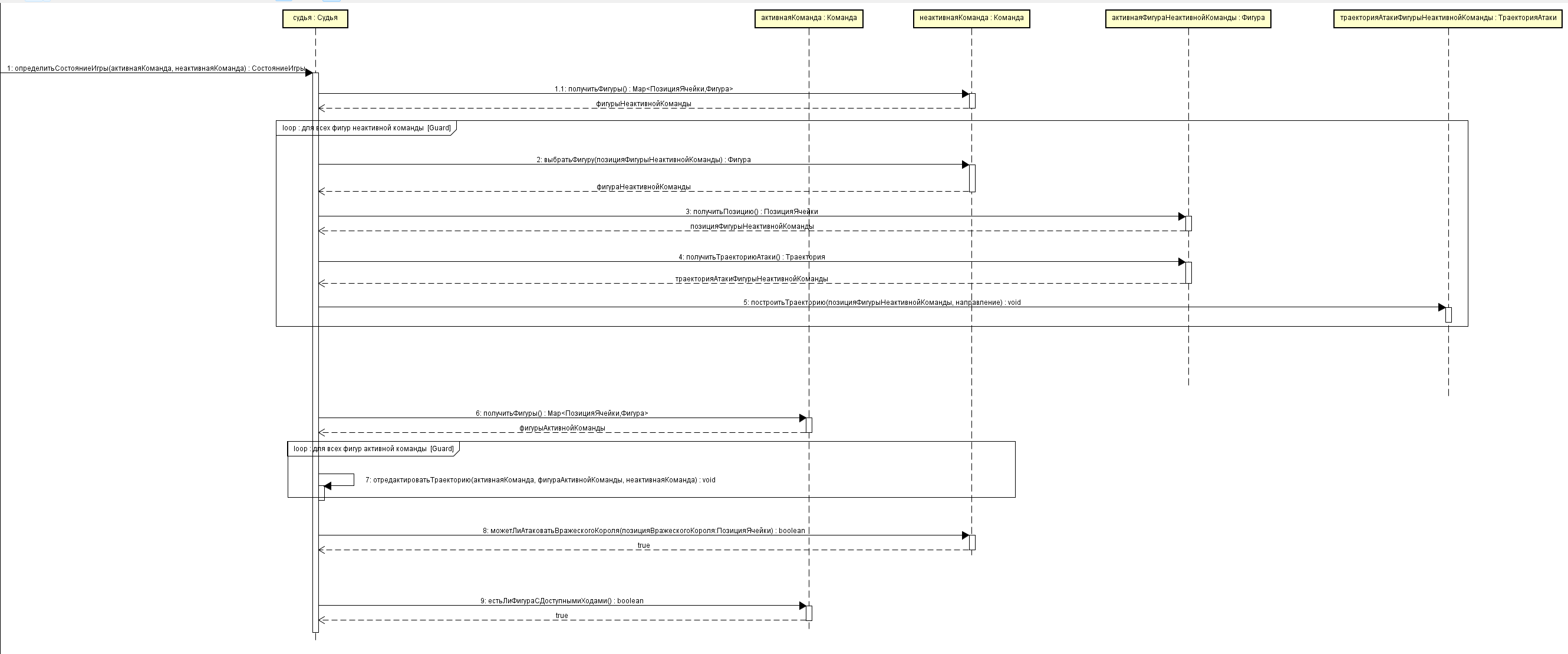


Рисунок 22 - Диаграмма последовательности для вычислительной модели.

Определение состояния игры – шах

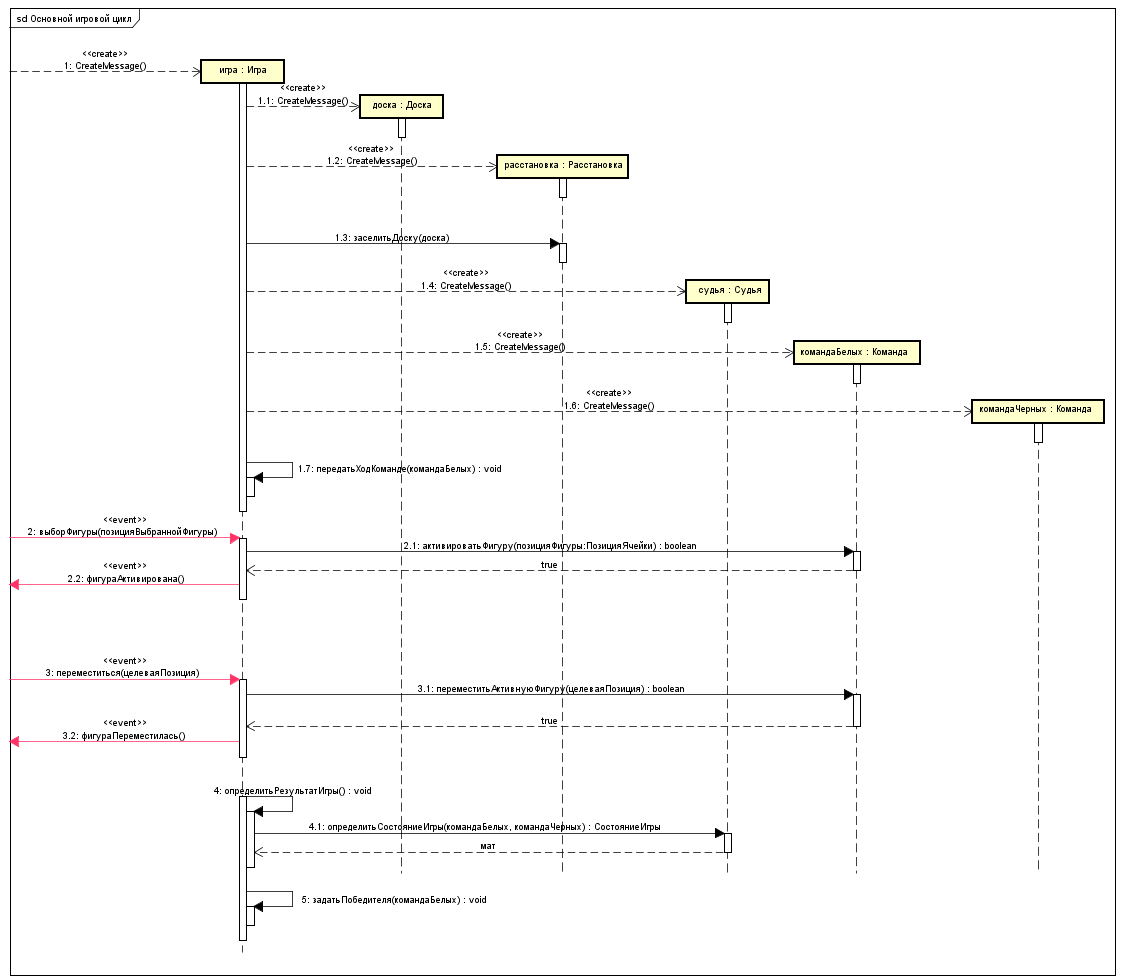


Рисунок 23 – Диаграмма последовательности для вычислительной модели.

Основной игровой цикл

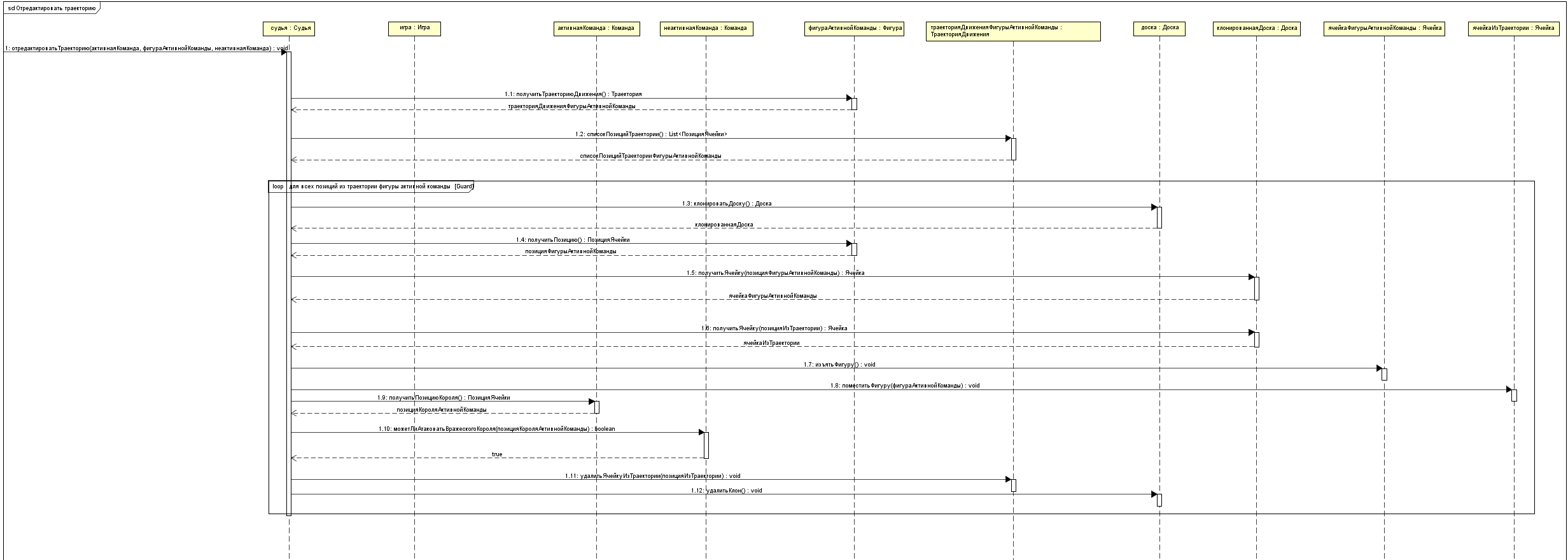


Рисунок 24 - Диаграмма последовательности для вычислительной модели.

Отредактировать траекторию

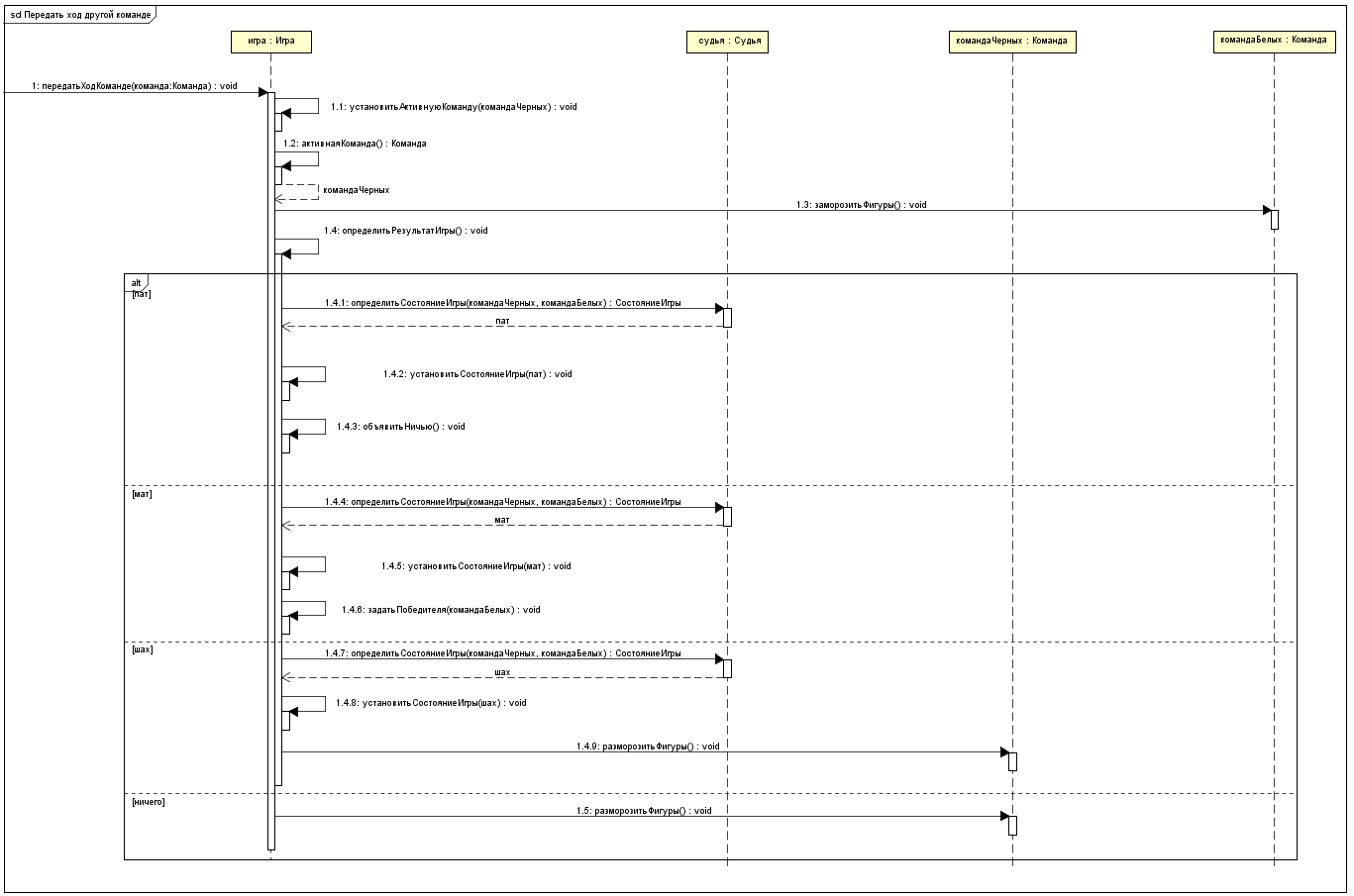


Рисунок 25 - Диаграмма последовательности для вычислительной модели.

Передать ход другой команде

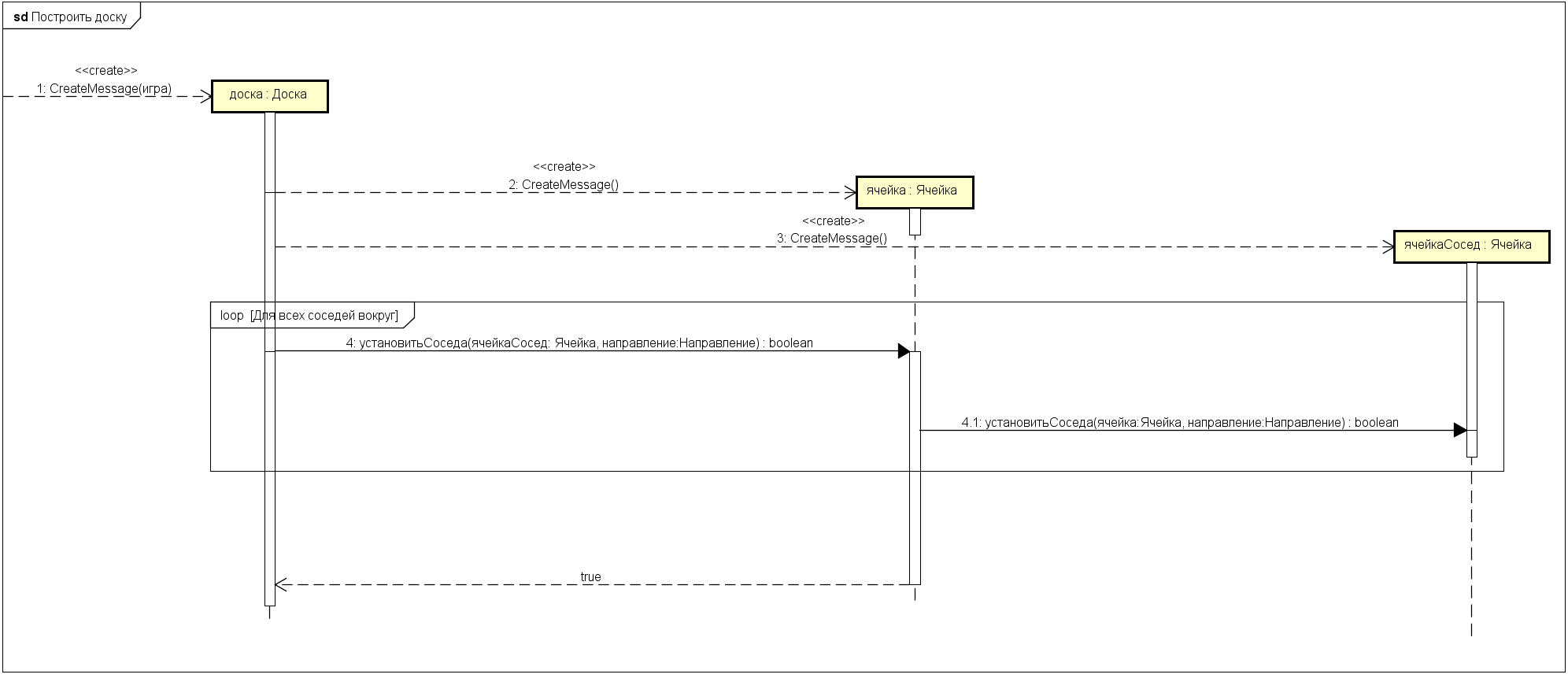


Рисунок 26 – Диаграмма последовательности для вычислительной модели.

Построить доску

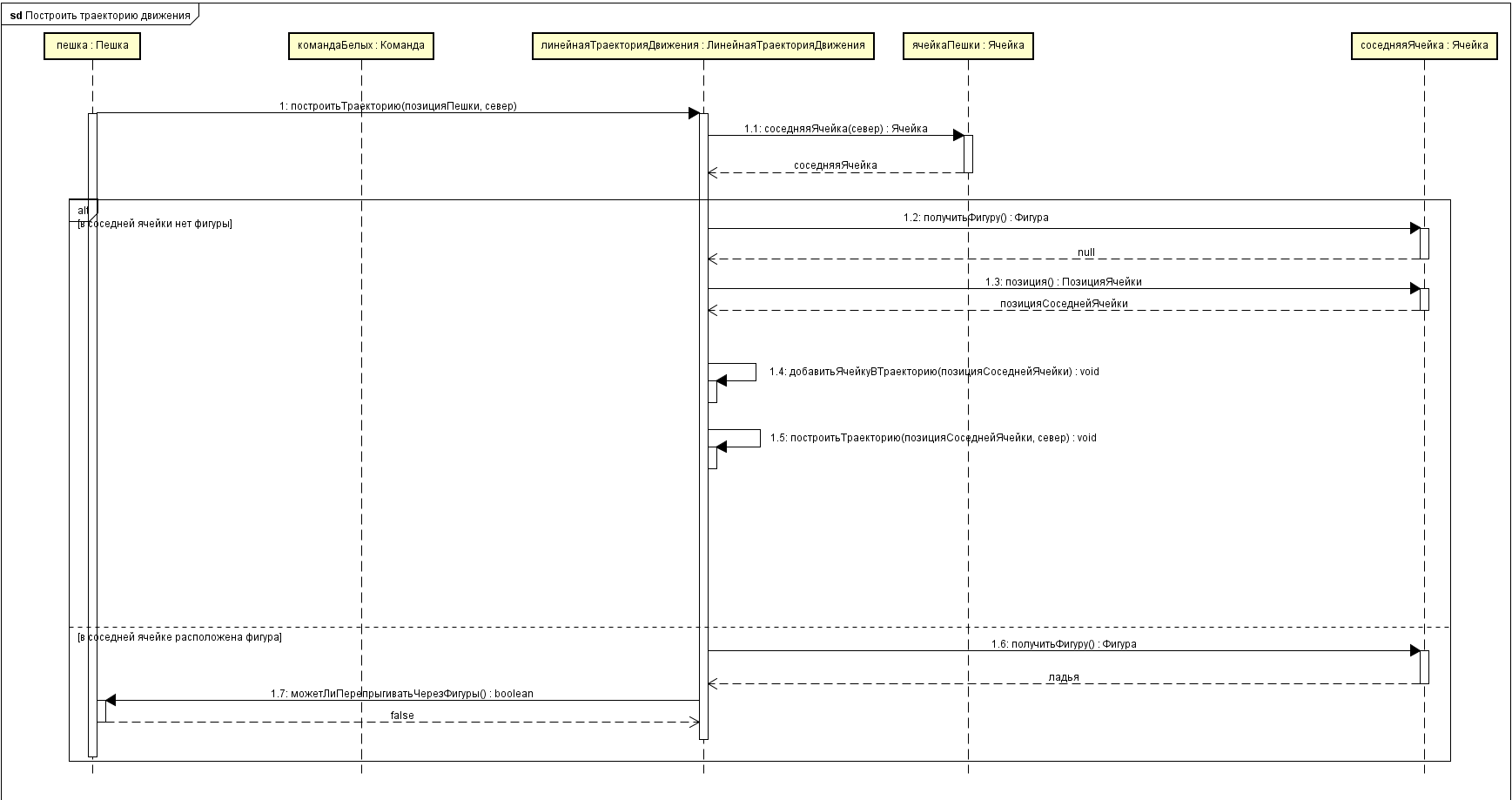


Рисунок 12 – Диаграмма последовательности для вычислительной модели.

Построить траекторию движения

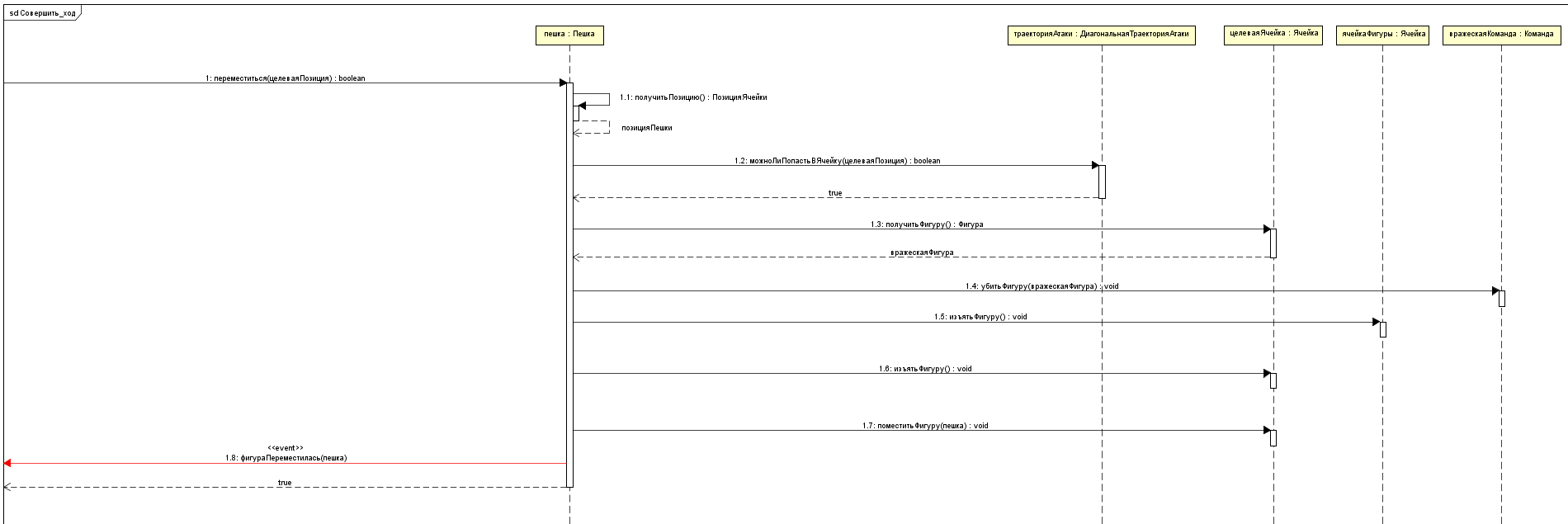


Рисунок 27 - Диаграмма последовательности для вычислительной модели.

Совершить ход

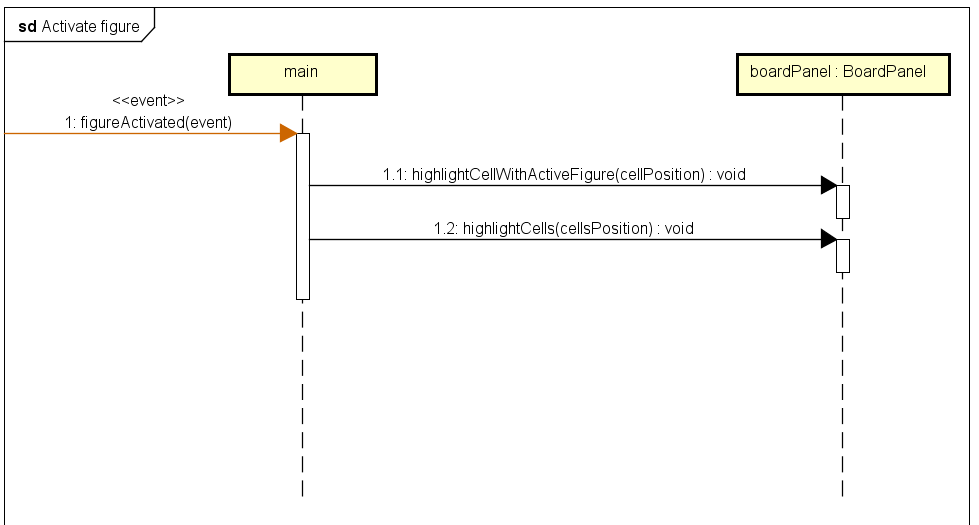


Рисунок 28 - Диаграмма последовательности для вычислительной модели.

Активировать фигуру (UI)

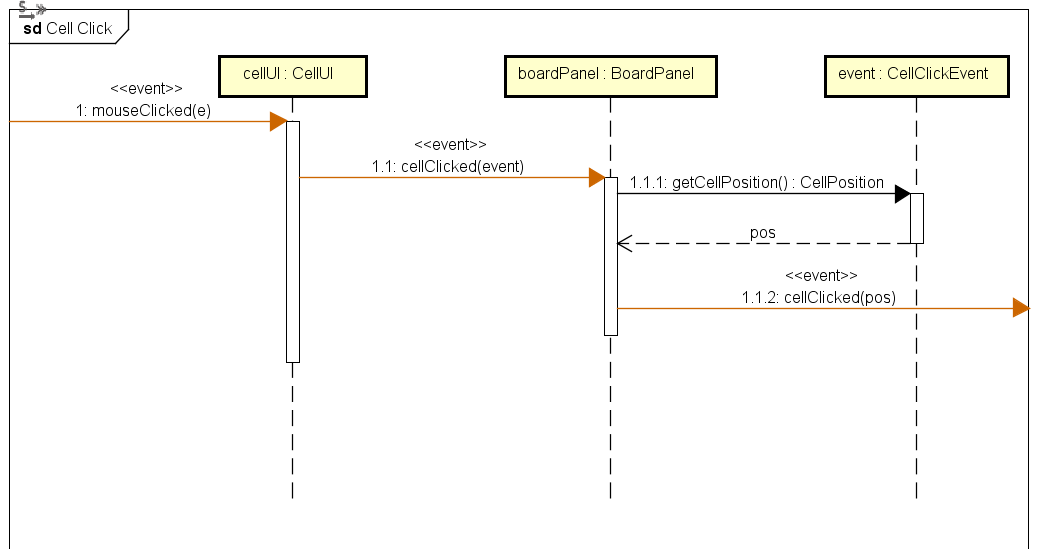


Рисунок 29 – Диаграмма последовательности для вычислительной модели.

Клик по ячейке

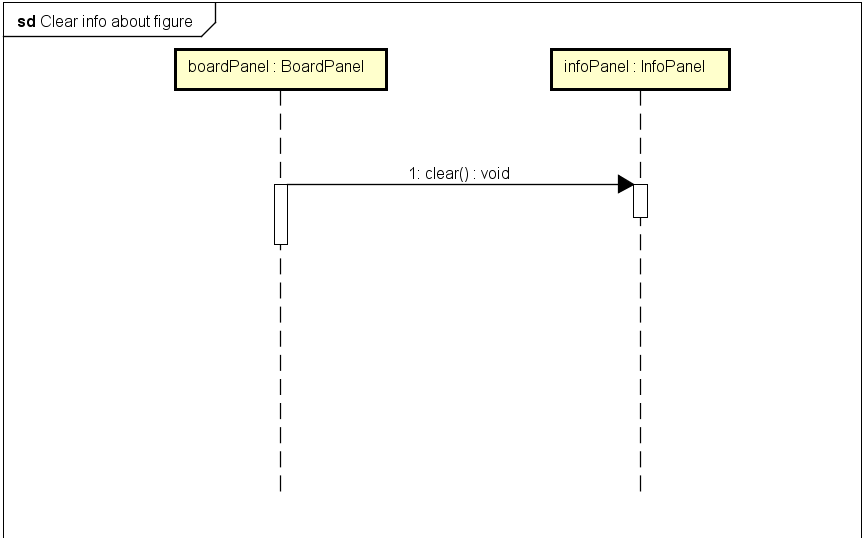


Рисунок 30 – Диаграмма последовательности для вычислительной модели.

Очистить информацию о фигуре

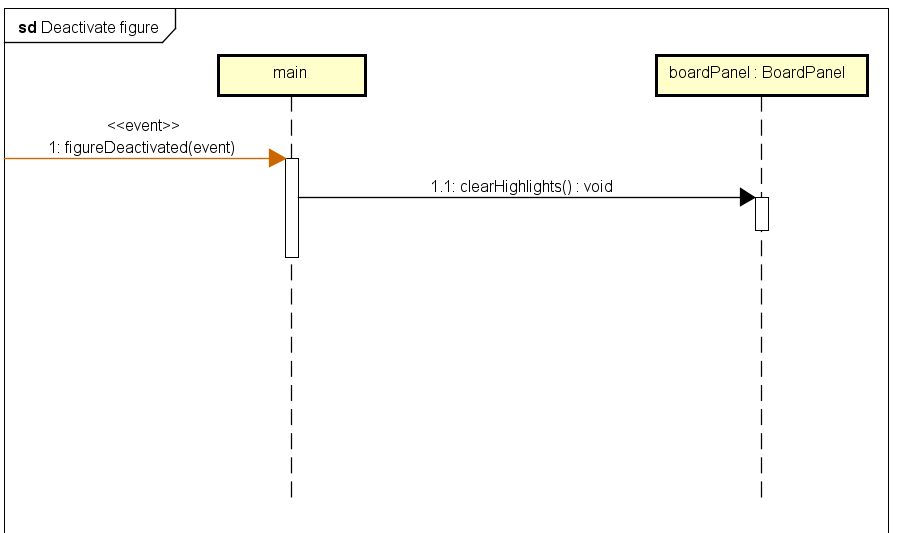


Рисунок 31 – Диаграмма последовательности для вычислительной модели.

Деактивировать фигуру

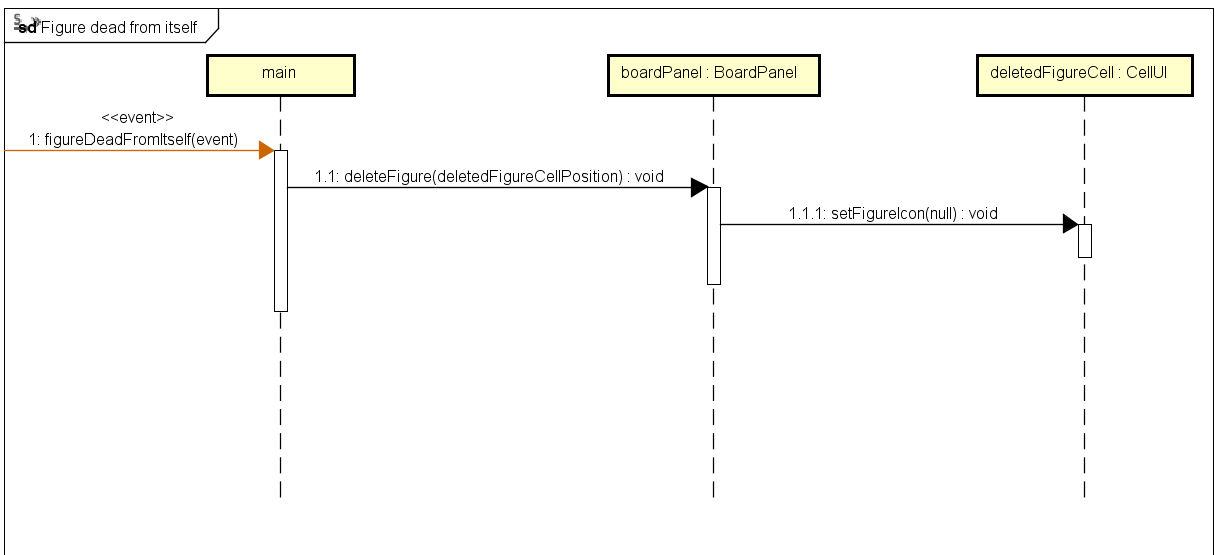


Рисунок 32 – Диаграмма последовательности для вычислительной модели.

Фигура убила себя

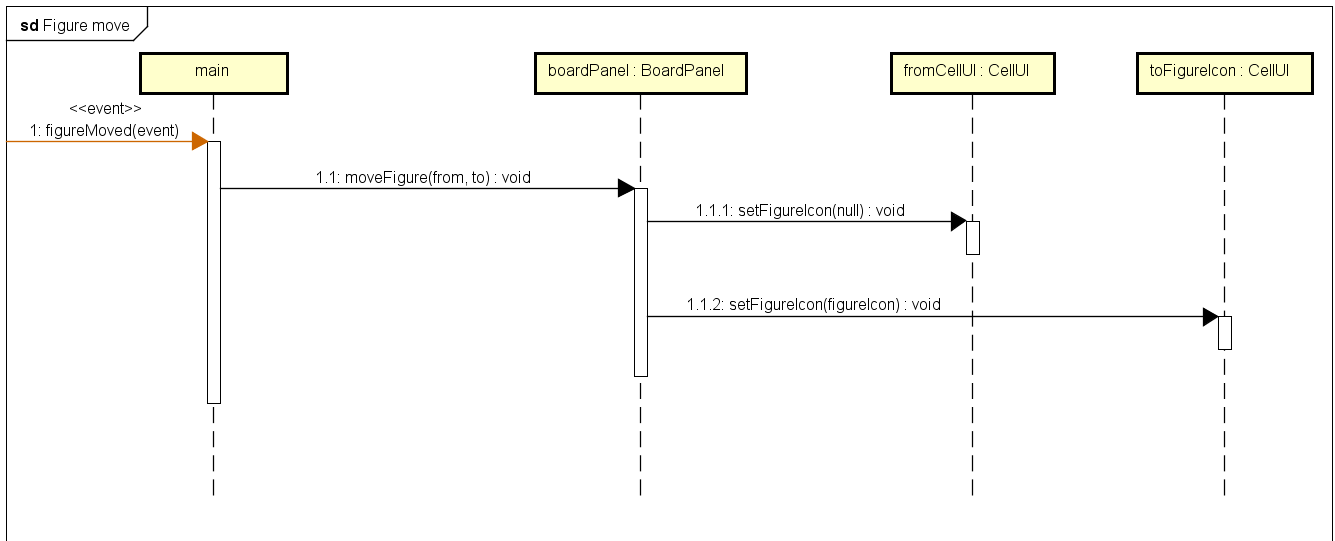


Рисунок 33 – Диаграмма последовательности для вычислительной модели.

Фигура переместилась

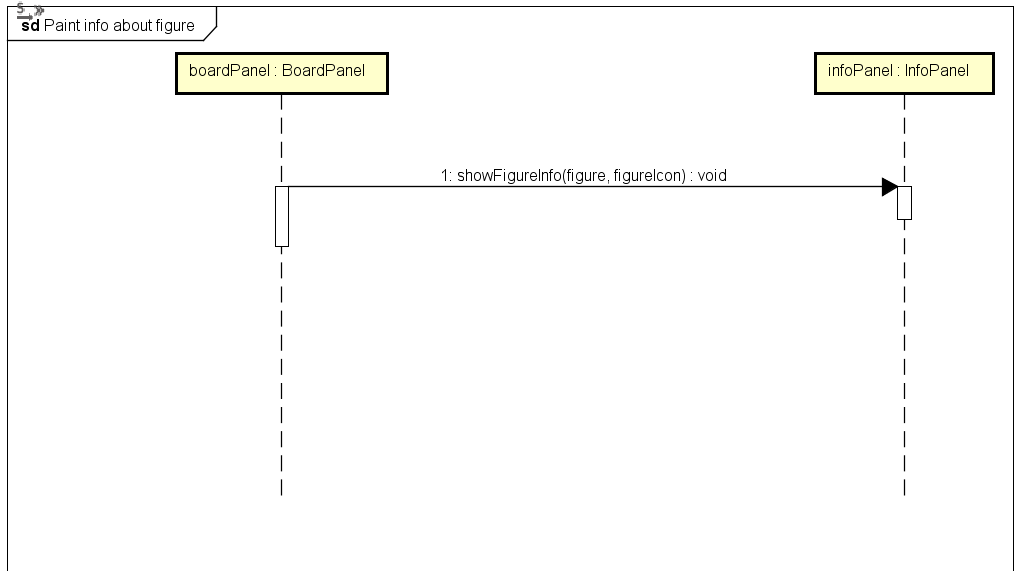


Рисунок 34 - Диаграмма последовательности для вычислительной модели.

Отрисовать информацию о фигуре

## 4.5 Человеко-машинное взаимодействие

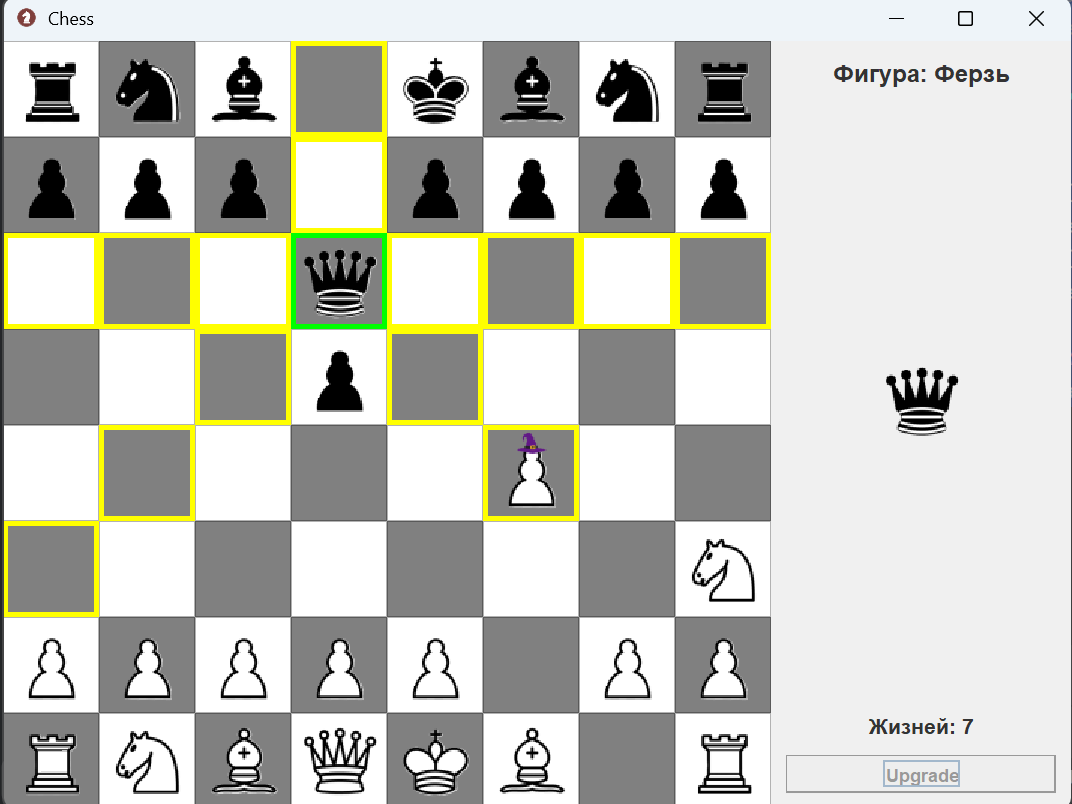


Рисунок 35 – Основное окно приложения

## 4.6 Реализация ключевых классов

package model.Figures;

import model.Direction;

import model.Team;

import model.Trajectories.AttackTrajectories.AttackTrajectory;

import model.Trajectories.MovementTrajectories.MovementTrajectory;

import java.util.List;

public class Wizard extends Figure{

public Wizard(Team team) {

super(team);

this.canJumpOver = true;

this.upgradeDamage = 1;

this.movementRadius = 2;

this.attackRadius = 2;

this.shiftPerStep = new int[][] { {1,1} };

this.movementDirections = List.of(Direction.NORTHWEST, Direction.NORTHEAST, Direction.SOUTHEAST, Direction.SOUTHWEST, Direction.NORTH, Direction.EAST, Direction.SOUTH, Direction.WEST);

this.attackDirections = List.of(Direction.NORTHWEST, Direction.NORTHEAST, Direction.SOUTHEAST, Direction.SOUTHWEST, Direction.NORTH, Direction.EAST, Direction.SOUTH, Direction.WEST);

this.movementTrajectories = List.of(new MovementTrajectory(this.movementDirections, this.movementRadius, this.shiftPerStep[0]));

this.attackTrajectories = List.of(new AttackTrajectory(this.team.getColor(), this.attackDirections, this.attackRadius, this.shiftPerStep[0]));

}

@Override

public FiguresTypes getFigureType() {

return FiguresTypes.WIZARD;

}

@Override

public FiguresTypes getUpgradeFigureType() {

return null;

}

@Override

public Figure cloneFigure(){

return new Wizard(this.team);

}

}

## 4.7 Реализация ключевых тестовых случаев

@Test

void testPawnUpgradeSuccess() {

// Выбираем белую пешку:

game.onFigureSelected(WHITE\_PAWN\_POS);

// Запомним старые жизни и тип

Figure pawnBefore = game.getBoard().getCellByPosition(WHITE\_PAWN\_POS).getFigure();

int oldLives = pawnBefore.getLives();

FiguresTypes oldType = pawnBefore.getFigureType();

assertEquals(FiguresTypes.PAWN, oldType);

// Выполняем апгрейд

game.upgradeFigure();

// Проверим, что событие апгрейда сработало ровно 1 раз

assertEquals(1, upgradeListener.count, "Должно быть ровно 1 событие апгрейда");

// В клетке должен быть новый объект фигуры (Wizard)

Figure upgraded = game.getBoard().getCellByPosition(WHITE\_PAWN\_POS).getFigure();

assertNotNull(upgraded, "После апгрейда в клетке должна стоять улучшенная фигура");

assertNotEquals(oldType, upgraded.getFigureType(), "Тип фигуры должен измениться");

assertEquals(FiguresTypes.WIZARD, upgraded.getFigureType(), "Тип должен стать WIZARD");

// Жизни уменьшились на upgradeDamage()

int expectedLives = oldLives - pawnBefore.getUpgradeDamage();

assertEquals(expectedLives, upgraded.getLives(),

"Количество жизней после апгрейда должно уменьшиться на upgradeDamage()");

// Проверим, что у команды выставился флаг «апгрейд совершен»

Team whiteTeam = upgraded.getTeam();

assertTrue(whiteTeam.isUpgraded(), "Флаг команды isUpgraded должен быть true");

// Если снова вызвать upgradeFigure(), изменений уже не будет:

upgradeListener.count = 0;

game.onFigureSelected(WHITE\_PAWN\_POS); // снова выбрать ранее апгрейженную фигуру

game.upgradeFigure();

// Никакого второго события не должно быть

assertEquals(0, upgradeListener.count, "При повторном апгрейде событие не должно срабатывать");

// Тип и жизни должны остаться прежними

Figure still = game.getBoard().getCellByPosition(WHITE\_PAWN\_POS).getFigure();

assertEquals(FiguresTypes.WIZARD, still.getFigureType(), "При повторном апгрейде тип не меняется");

assertEquals(expectedLives, still.getLives(), "При повторном апгрейде жизни не меняются");

}

/\*\*

\* 2) Если активная фигура — не пешка (например, конь), апгрейд не должен проходить:

\* - никакой апгрейд не срабатывает,

\* - команда не получает флаг isUpgraded,

\* - событий апгрейда не генерируется.

\*/

@Test

void testUpgradeWhenNotPawnDoesNothing() {

// Выберем, скажем, белого коня (по стандартной расстановке он на (7,1))

CellPosition knightPos = new CellPosition(7, 1);

game.onFigureSelected(knightPos);

Figure knightBefore = game.getBoard().getCellByPosition(knightPos).getFigure();

assertEquals(FiguresTypes.KNIGHT, knightBefore.getFigureType());

// Попытка апгрейда

game.upgradeFigure();

// Событий апгрейда не должно быть

assertEquals(0, upgradeListener.count, "Никакое событие апгрейда не должно сработать");

// Тип фигуры остался прежним (конь)

Figure knightAfter = game.getBoard().getCellByPosition(knightPos).getFigure();

assertEquals(FiguresTypes.KNIGHT, knightAfter.getFigureType(),

"Тип фигуры не должен измениться");

assertFalse(knightAfter.getTeam().isUpgraded(),

"Флаг команды isUpgraded не должен выставиться при попытке апгрейда не-пешки");

}

/\*\*

\* 3) Ещё проверим, что если ни одна фигура не выбрана (activeFigure = null),

\* вызов upgradeFigure() тоже ничего не делает.

\*/

@Test

void testUpgradeWithNoActiveFigureDoesNothing() {

// activeFigure ещё не назначена → сразу вызвать upgradeFigure()

game.upgradeFigure();

assertEquals(0, upgradeListener.count, "Без выбранной фигуры событие не срабатывает");

}

// ────────────────────────────────────────────────────────────────────

// Вспомогательный «слушатель» для подсчёта апгрейдов

// ────────────────────────────────────────────────────────────────────

private static class TestUpgradeListener implements FigureUpgradedListener {

int count = 0;

@Override

public void upgradedFigure() {

count++;

}

}

# 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>