Afonso Monteiro, Bruno Rosendo e Pedro Pinheiro
M.EIC ASMA

Intrigue

...

Game implementation using agents through the JADE framework

Problem Description

Intrigue is a turn-based board game where 5 players try to get as rich as possible during 5 rounds. Each owns a Palazzo with 4 cards (each is a job position, with a specific salary), 8 relatives (2 of each job type) and starts with 32000$. The values are parameterized in our implementation.

Every turn a player sends 2 of its relatives to look for work and employs other players' relatives in his. In the beginning of its turn, a player receives money based on the salary of the positions his relatives are employed.

The main interest of the game comes from the fact that players can bribe each other with the goal of getting their relatives employed in high paying jobs. There can also be conflicts where the players must negotiate to get the positions.

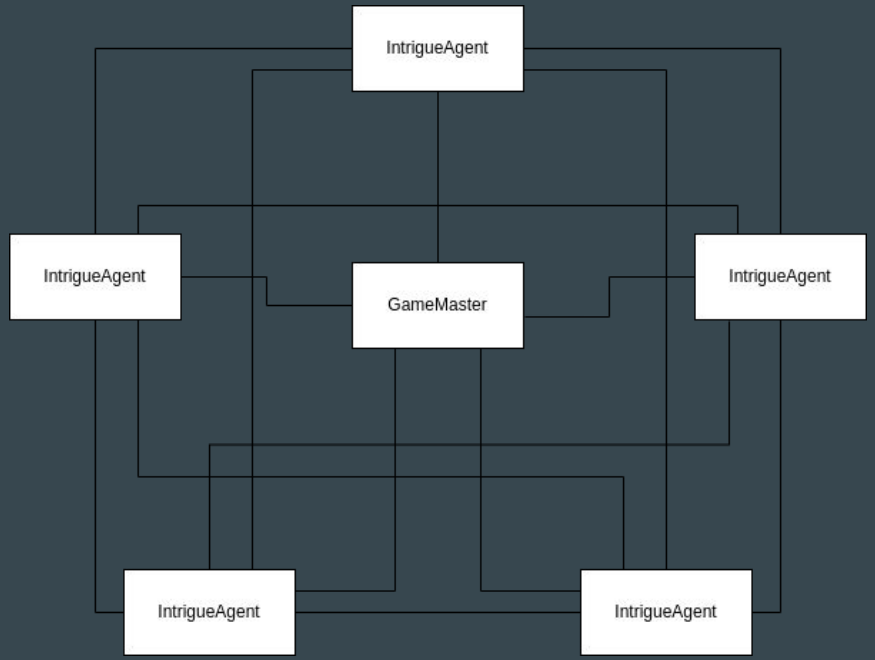
The complete rules can be found in https://boardgamegeek.com/boardgame/265/intrigue.

Global Diagram

There is a GameMaster, the agent responsible for keeping track of turns, rounds, and ending the game.

All the players extend *IntrigueAgent*. This allows us to test agents against mismatched opponents and easily implement different strategies and behaviours.

All agents are aware of the GameMaster, who provides instructions on their turn and actions to be performed. Additionally, agents are interconnected to broadcast their actions and engage in negotiations with each other.



Communication

Each agent can send **ACLMessages** to others, enabling them to share their actions, engage in negotiations, and keep their game state up-to-date.

These messages are labeled with **protocols** that specify the type of action to be performed or information to be updated, with *request* and *inform* being commonly used **performatives**.

Optionally, messages can carry in their **content** a serialized Java record with extra information, such as IDs or bribe amounts.

Protocols Used:

`ASSIGN_JOBS`	`COLLECT_INCOME`
`SEEK_EMPLOYMENT`	`JOBS_ASSIGNED`
`RESOLVE_CONFLICT`	`EMPLOYEES_SENT`
`NEW_TURN`	`BRIBE_OFFERED`

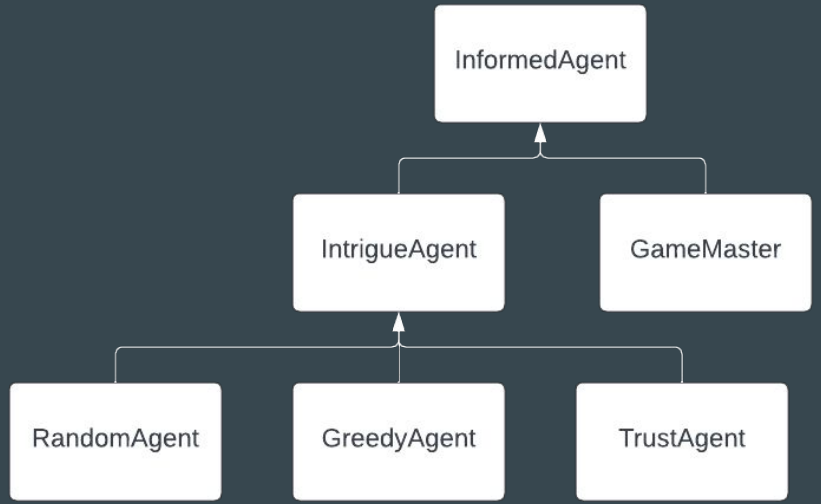
Content Example:

```
public record JobsAssigned(
        List<Integer> selectedPieceIndices,
        List<Integer> cardIndices,
        List<Integer> selectedPieceOwners // IDs
) implements Serializable {}
```

Agent Architecture

InformedAgents register in the service and actively listen for new agents that join. They also receive updates about the game and adjust their state accordingly.

IntrigueAgents are players who actively participate in the game. They are extended by implementing specific behaviours for various game actions, such as seeking jobs or giving bribes.



RandomAgent

- Game actions are randomly selected.
- Bribes are ignored and jobs are assigned randomly.
- Bribes are given with a random value between the minimum amount and current money.
- Employment is sought in random Palazzos.

GreedyAgent

- Best immediate action is chosen, ignoring possible cooperation.
- Low-paying jobs are assigned to players with the most money.
- Conflicts are solved by rewarding the player with the highest bribe.
- Previous best bribe is considered and $1 is added.
- Employment is sought in Palazzos with available and high-paying positions.

TrustAgent

- Trust factor is maintained for all players, and game actions are chosen based on trustworthiness.
- High-paying jobs are assigned to trusted players.
- Conflicts are solved by rewarding trusted players.
- Bribes are linearly proportional to the trust factor and larger than the previous best.
- Employment is sought in Palazzos of the most trusted players.

Agent Discovery

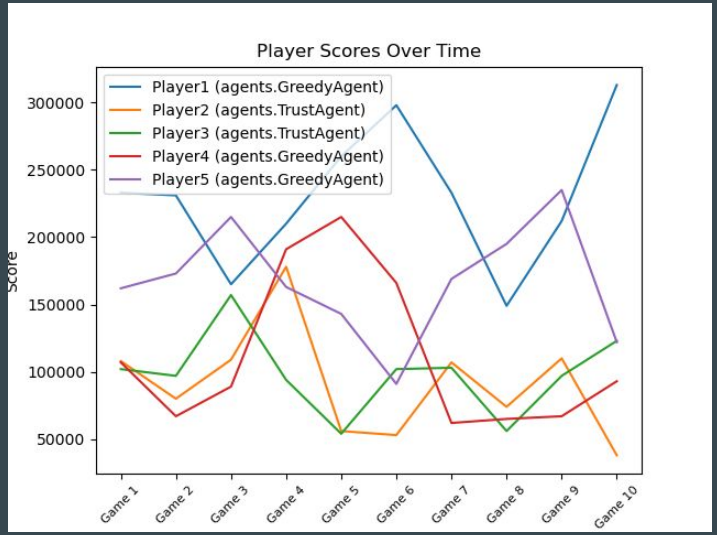
When an InformedAgent is launched, it registers itself in a **DFService** (directory facilitator, aka yellow pages) identified with "*intrigue-updates*" and adds two behaviours: ***AgentFinder*** and ***GameUpdateListener***.

AgentFinder will first search for existing agents in the DF and then subscribe to messages that inform when new agents arrive. This allows all informed agents to easily communicate their actions and requests.

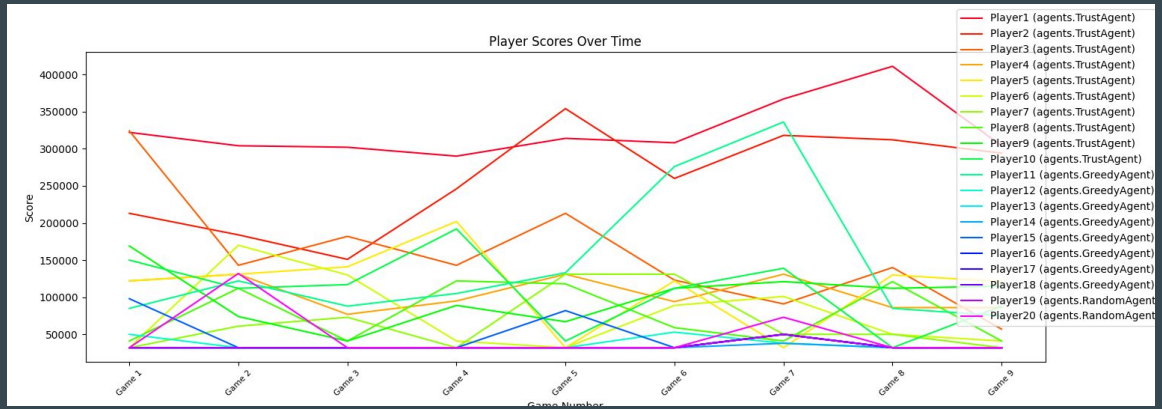
GameUpdateListener uses a message template to handle protocols related to game updates, decodes the content and calls the respective handlers for further processing.

Experiments and Results

To assess the performance of our system, we recorded the total amount of money each player accumulated in various games with different agent combinations. It's important to note that in these games, the player with the highest amount of money at the end is declared the winner.



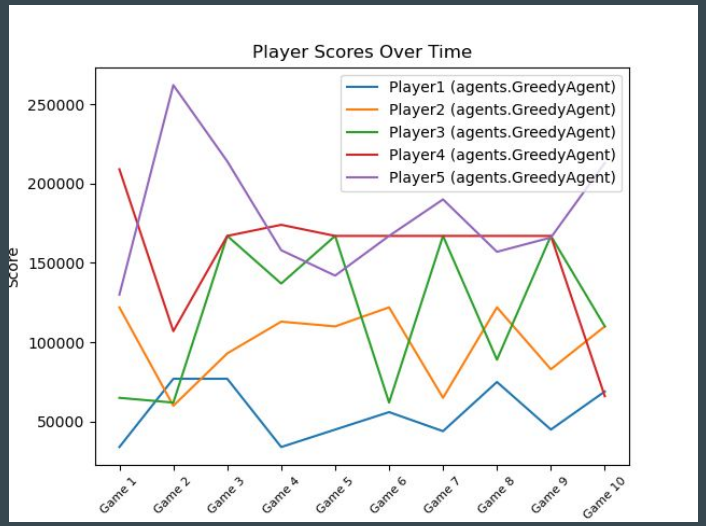
10 games with greedy and trust agents



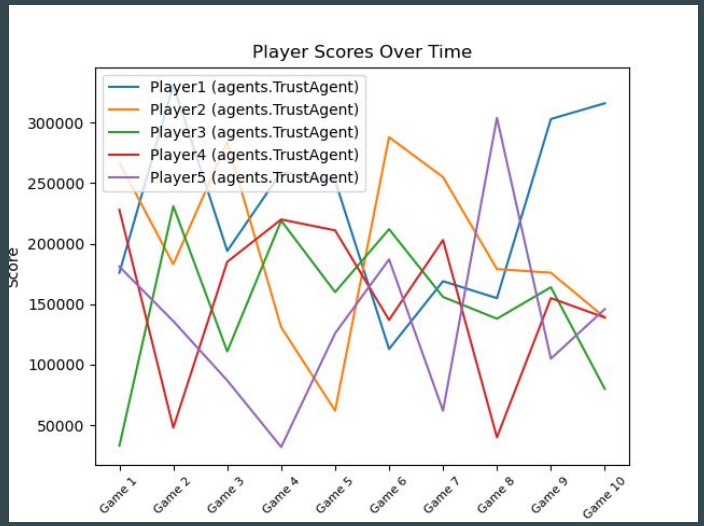
9 games with 20 agents (trust and greedy)

Experiments and results

The sequence in which agents play is inconsequential when they act randomly, but greedy agents tend to perform better when they play later, while trust-based agents tend to perform better when they play earlier. This can be corroborated by examining the number of wins each player accumulates in games with agents of the same type.



10 games with greedy agents



10 games with trust agents

Conclusions and Future Work

The project provided valuable insights into understanding the functioning of multi-agent systems and developing a distributed game with various types of agents interacting.

However, the most challenging aspect was managing communication and synchronization among the agents in the game. Although JADE's features were useful, the documentation was not very comprehensive, and some unexpected behaviors were encountered.

Despite these challenges, the chosen architecture was sensible for the problem at hand, and creating new agents and strategies became easier after overcoming initial hurdles.

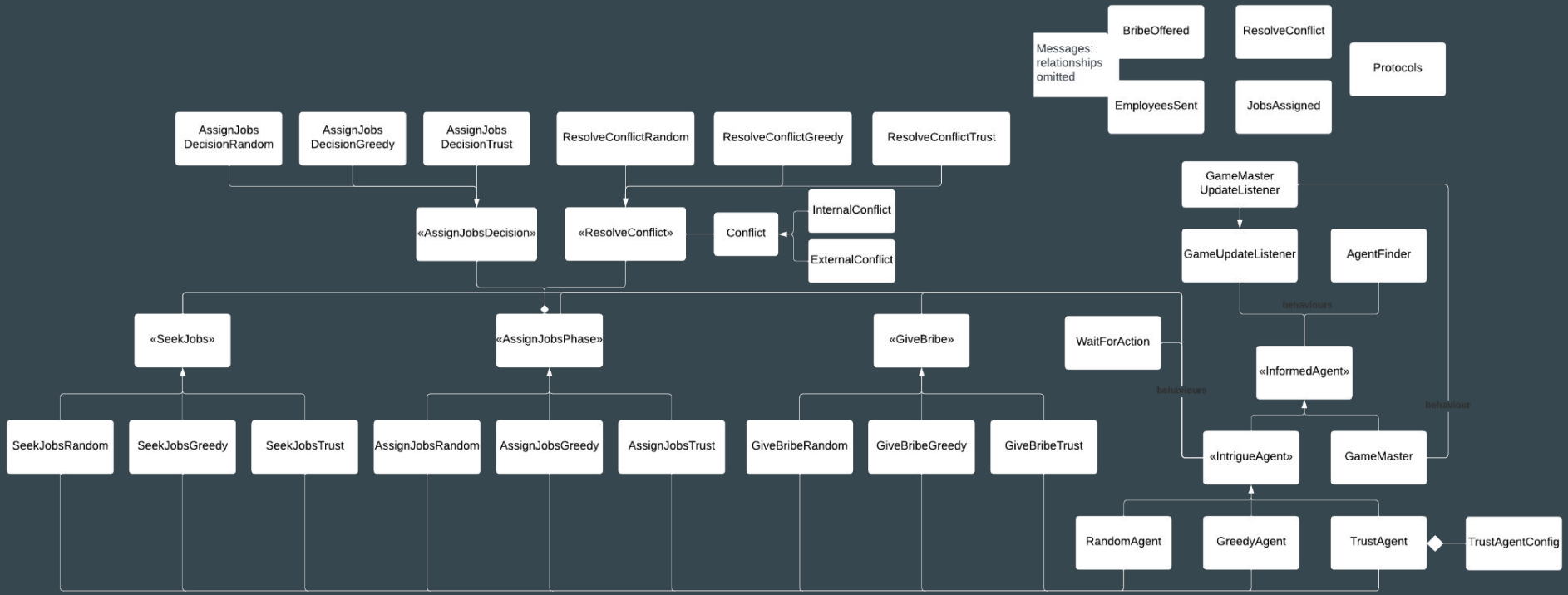
For future work, rigorous testing of the system is recommended to ensure robustness, as the experiments were limited and on the same machine. Additionally, further development of agents and behaviors, as well as experiments with different game rules, would be interesting. Improving the user interface would also enhance the overall immersive experience.

Afonso Monteiro, Bruno Rosendo e Pedro Pinheiro
M.EIC ASMA

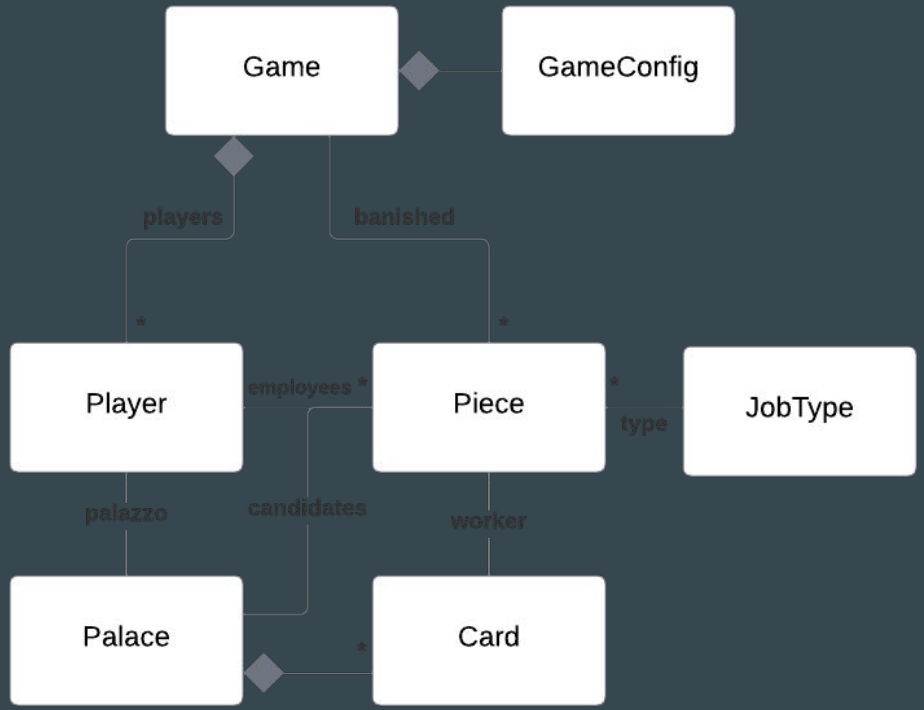
Additional Information

...

Complete Class Diagram - Agents and Behaviours



Complete Class Diagram - Game



Implemented Classes - IntrigueAgent

As aforementioned, every player agent implementation extends *IntrigueAgent*. This class waits for messages on setup and upon their receival passes them on to the appropriate methods. These methods are *getAssignJobsBehaviour, getSeekJobsBehaviour* and *getResolvedConflictBehaviour*. They are abstract and therefore must be overridden by specific agents. To facilitate this, we also created the abstract classes such as *SeekJobs, GiveBribe, ResolveConflict, AssignJobsPhase and AssignJobsDecision* . These allow the implementation of the agents to only focus on the things that differentiate them from one another and hide details such as messaging and determining which kind of conflicts had arisen.

Implemented Classes - InformedAgent and GameMaster

Both *IntrigueAgent* and *GameMaster* extend *InformedAgent.* This abstract class is responsible for using the Yellow Pages service, allowing agents to find others and register themselves. It also initiates the *GameUpdateListener* which receives and handles messages from other agents, informing the receiving agent of the actions other agents performed and allowing it to update its vision of the game.

*IntrigueAgent*s also use *WaitForAction,* a cyclic behaviour that waits for actions received from other players and the game master. As for the *GameMaster,* it extends the *GameUpdateListener* (*GameMasterUpdateListener*) so it can inform the players of their actions on the right stages of the game.

Implemented Classes - Conflicts

There are two types of conflicts in Intrigue:

- **External:** Two or more applicants of the same profession are waiting in the same park.
- **Internal:** A position has already been filled by one of the professions and one or more applicants from the same profession have shown up in the park looking to get the same job.

The key differences between external and internal conflicts lie in the order of players giving bribes and how conflicts are handled. In internal conflicts, the player who already holds the position always bribes first, and conflicts involving more valuable positions are resolved first.

To model these concepts, the game utilizes the *ExternalConflict* and *InternalConflict* classes, both of which extend the abstract *Conflict* class. The *InternalConflict* class also implements the *Comparable* interface, allowing it to be automatically sorted, such as by using a *TreeSet*.

How to Run - Experiments

To execute the experiments, you can utilize the provided main function in *Main.java*, which runs a specified game configuration multiple times and generates a CSV file containing the results. The configuration for the game and experiment parameters, such as the number of players and the name of the CSV file, can be adjusted in *config/GameConfig.java*.

Once the experiments are completed, you can analyze the results using the *test.py* Python script located in the root folder. This script generates and saves plots for visualizing the game outcomes. Please exercise caution when setting the name of the CSV file and the number of repetitions in the previous step.

Important: Remember to add the JADE library to the project's dependencies in your Java IDE to properly execute the project! The working directory should be *Intrigue/src/*.

How to Run - Normal Game

To initiate a single instance of a game, it is necessary to create a JADE container and provide it to the Game constructor, along with the desired type of player agents. The following template can serve as a reference:

```
public class Main {
    public static void main(String[] args) {
        String[] agents = {"agents.RandomAgent", "agents.GreedyAgent", "agents.RandomAgent", "agents.RandomAgent", "agents.RandomAgent"};
        Profile profile = new ProfileImpl();
        profile.setParameter(Profile.CONTAINER_NAME, s1: "Intrigue");
        AgentContainer gameContainer = jade.core.Runtime.instance().createMainContainer(profile);
        new Game(gameContainer, createAgents: true, agents);
    }
}
```

For additional customization, you can modify the configuration in *config/GameConfig.java*, where various game parameters such as the number of players and starting money are defined. This allows you to tailor the game settings to your specific requirements.

Execution Example

```
Agent player2@192.168.1.106:1099/JADE is ready.
Agent player5@192.168.1.106:1099/JADE is ready.
Agent player1@192.168.1.106:1099/JADE is ready.
Agent player3@192.168.1.106:1099/JADE is ready.
Agent player4@192.168.1.106:1099/JADE is ready.
Agent GameMaster@192.168.1.106:1099/JADE is ready.
  +--------+--------+--------+--------+--------+
          Round: 1 | Turn: 1
No pieces in the island.
Player 1
    | Money: 32000
    | Pieces: Scribe|Scribe|Minister|Minister|Alchemist|Alchemist|Healer|Healer|
    | No pieces in palace park
    | Card status:
        Card of value 1000 is not occupied
        Card of value 3000 is not occupied
        Card of value 6000 is not occupied
        Card of value 10000 is not occupied
Player 2
    | Money: 32000
    | Pieces: Scribe|Scribe|Minister|Minister|Alchemist|Alchemist|Healer|Healer|
    | No pieces in palace park
    | Card status:
        Card of value 1000 is not occupied
        Card of value 3000 is not occupied
        Card of value 6000 is not occupied
        Card of value 10000 is not occupied
```

The game starts by informing when all agents are ready to start the game. Subsequently, in each turn, the status of all players, along with their corresponding game pieces, is displayed on the console.

This process is repeated until the conclusion of the game, at which point the victorious player is declared.

```
Player 5
    | Money: 100000
    | Pieces:
    | No pieces in palace park
    | Card status:
        Card of value 1000 is occupied by Alchemist of 4
        Card of value 3000 is occupied by Healer of 1
        Card of value 6000 is occupied by Minister of 2
        Card of value 10000 is occupied by Scribe of 2
  +--------+--------+--------+--------+--------+

Game is over. The winner is player5 with 100000 money.
```