Multi-Agent Connect Four

Ole August Støle
Sebastiaan Milis

Abstract

Agent-oriented programming (AOP) is a programming paradigm that is still in its infancy but is very promising for use in systems that interact with the real world. Since gamification is getting more and more ubiquitous and is by definition an interaction with the real world, this project investigates the use of AOP when programming a game. An example game, Connect Four, is developed in an AOP language (JADE) and this implementation is compared to implementations in other programming paradigms. This shows the big advantages of AOP. AOP is a great and very natural programming paradigm for reasoning about games and ensures great performance on a lot of important non-functional requirements.

Table of contents

Introduction (Omicini)

Throughout the history of software development, different programming paradigms have been proposed and used, all with their own advantages and disadvantages. A lot of languages that are used nowadays follow an imperative programming paradigm, either procedural or object-oriented but lately, declarative programming languages are gaining attention as well.

One can however notice that these programming paradigms do not perfectly fit the needs of the artificial systems that are being developed nowadays and are becoming a very important part of our world. Key aspects of these modern artificial systems are physical distribution, autonomy and intelligence and a fitting programming paradigm should thus fit these key aspects as much as possible. For these Multi-Agent Systems, the fitting programming paradigm seems to be agent-oriented programming (AOP). Although this programming paradigm is still in its infancy, one can expect it to be the dominant programming paradigm in the future. Development of AOP languages will therefore have to speed up rapidly.

Gamification, the application of typical elements of games to other areas, is getting more ubiquitous as a way of motivating humans when interacting with software systems and with the emergence of artificial systems, this will only get more ubiquitous. At this moment, games are mostly implemented using one of the other programming paradigms. In this project, the application of AOP in the domain of games is investigated by developing an example game as a Multi-Agent System, more precisely Connect Four. This gives the opportunity to investigate the suitability of AOP for games and to compare AOP as a programming paradigm for games to other paradigms that are currently being used.

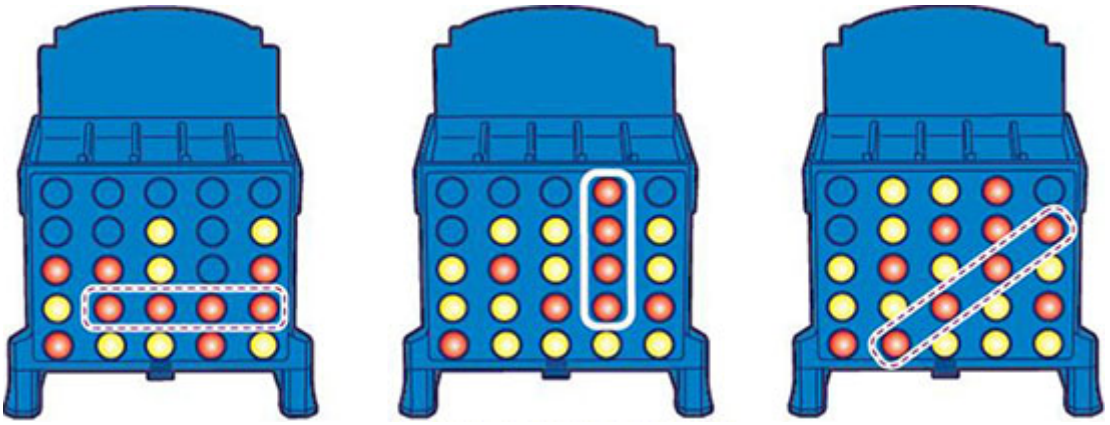
In the next section, some background information is provided. Afterwards the implementation of Connect Four as a Multi-Agent Systems is explained, the implementation is evaluated, and some further discussion is provided. At the end of this report, a conclusion of the whole project can be found.

Background

In this section, some background information is provided that is useful to follow and understand the upcoming sections and therefore the whole project.

Connect Four

The game of Connect 4 is a turn-based board game with 2 players participating. The board is a 6x7 upright grid. Each player has their own colored coins. One player begins by placing a coin within one of the 6 columns. This coin will fall straight down and take the lowest position in that column not yet occupied. The turn is then passed over to the next player. This loop is continued until the board reaches a terminal state. A player cannot place a coin in a column that's full, i.e. have 7 coins in it. There are four terminal states, three of them are winning states and one is the draw state. A winning state can be reached by connecting four coins of the same color either vertically, horizontally, or diagonally. If none of the winning states are reached but all available positions are taken by some coin, the game ends in a draw.



Winning states for Connect 4 (UltraBoardGames)

Connect X

The generic version of Connect 4 is Connect X. It's a made-up game where you're able to configure the game's width, height and number of connected pieces required to win. There can be more than 2 players and the rules of winning, as well as playing the game might be different. Winning could e.g. only be possible by having a vertical row or it might be possible to put coins in another location in a row than the lowest available one. One can thus see Connect X as a generic version of Connect 4 with a lot of parameters that are free to choose.

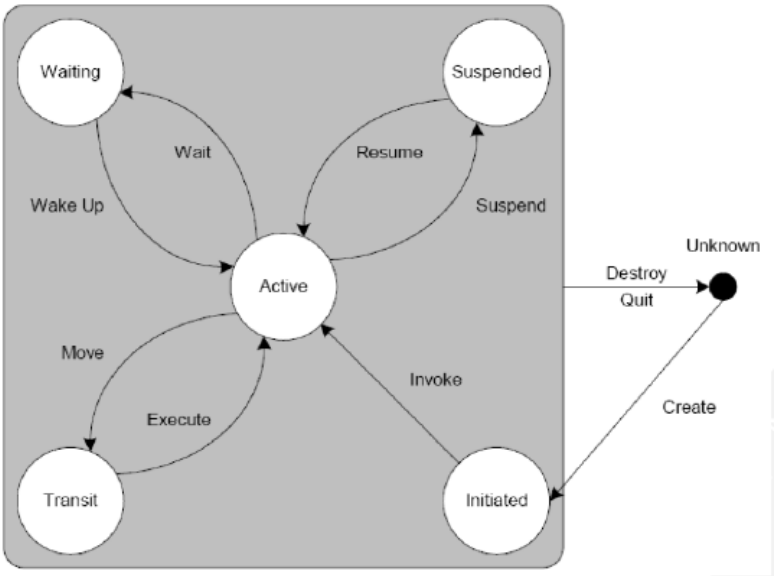
The game implemented in this project supports all variants of Connect X with a full implementation of the widely known Connect 4 variant.

JADE (Tilab) (Calegari and Omicini)

JADE stands for Java Agent DEvelopment Framework. It is a Java-based framework for developing agent-based applications according to the FIPA specifications.

Agents

In JADE, a Java and thus object-based middleware, agents are represented by Java objects. They however have a wide range of features promoting their autonomy. Every agent is executed in a single Java thread. They have a globally unique name and they can communicate via the below-mentioned ACC. The business logic of agents is expressed as behaviors. An agent is always in one of the FIPA states, as shown in the figure below.



FIPA states

Behaviors

A JADE behavior is an activity that has to be performed to complete a task. It can be a proactive as well as a reactive activity. Behaviors are run by the single Java thread containing the agent in a non-preemptive, round-robin way so the programmer should take this into account while developing agents and their behaviors.

Containers

Agents are spawned in containers of which one container acts as the main container (the first one). This container is responsible for maintaining a registry for all agents throughout the whole platform. Containers can be distributed over several hosts.

Agent Management System (AMS)

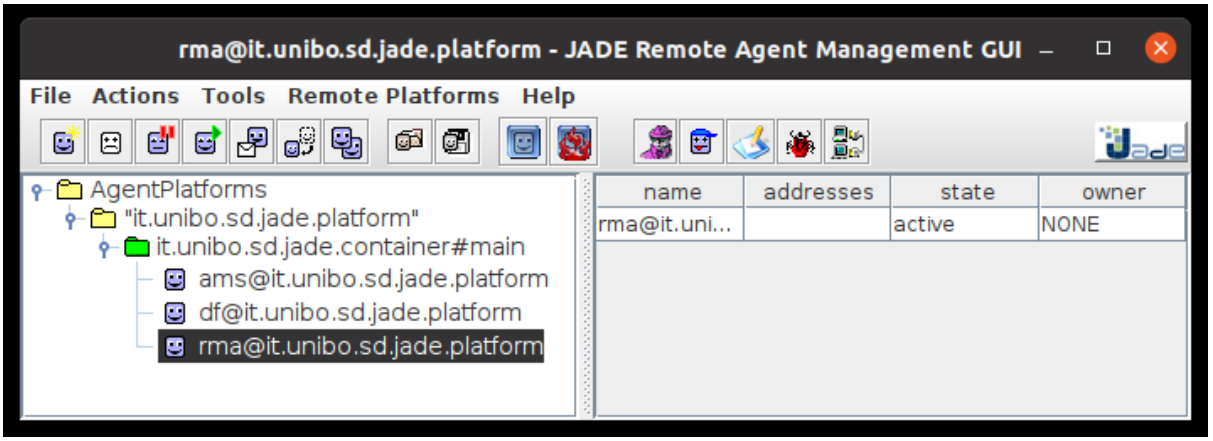
The Agent Management System (AMS) is the white pages service for a JADE platform. It keeps track of all agents throughout the whole platform (over all containers) and makes it possible to find agents in a white pages manner.

Agent Communication Channel (ACC)

The Agent Communication Channel (ACC) provides an asynchronous messaging service that makes it possible to send messages to all agents on the platform in a structured way. It follows the specifications of the FIPA ACL.

Remote Management Agent (RMA)

The Remote Management Agent (RMA) provides a GUI that can be used to inspect and control a JADE platform.

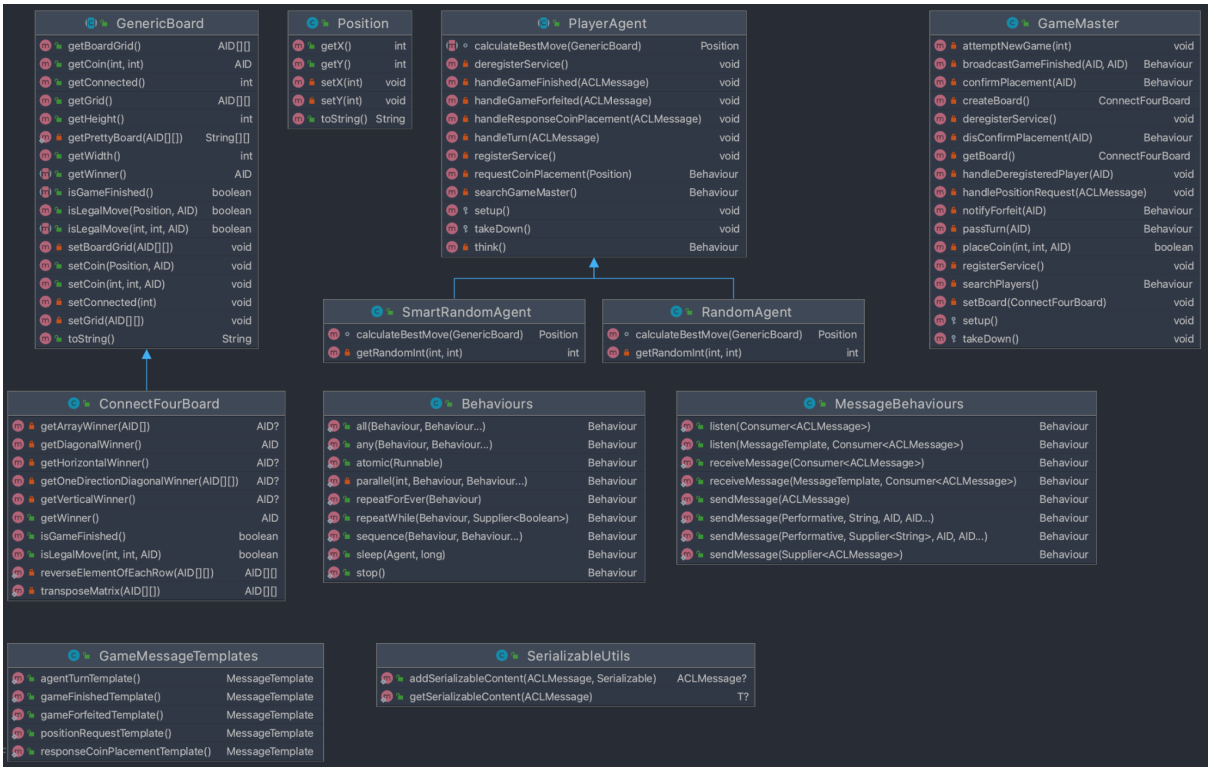


Screenshot of the Remote Management Agent GUI

Directory Facilitator (DF)

The directory facilitator allows agents to expose its services. Agents can register their service to the DF, which in turn enables other agents within the container to query agents offering a certain service.

Implementation



Overview of the project's main classes

Game agents

The game consists of two different agents, the GameMaster and the PlayerAgent.

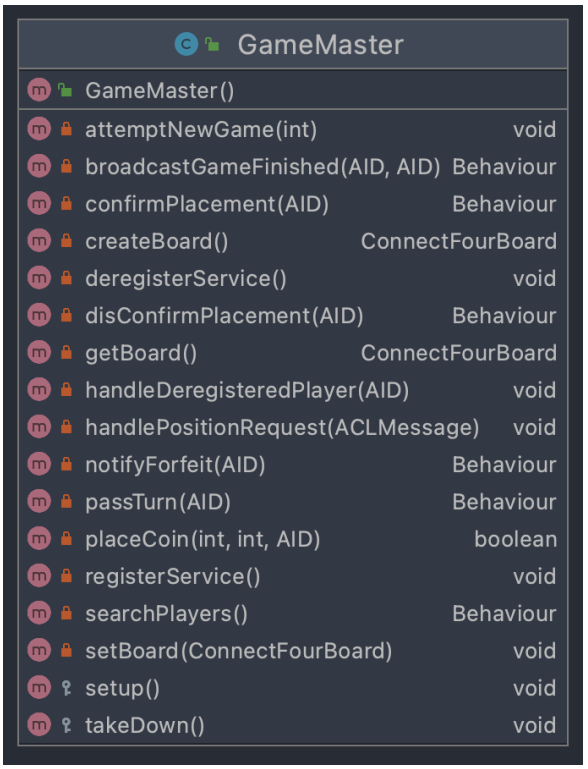
GameMaster
On creation, the game master will register its service to the Directory Facilitator (DF) as a "game-master". It will then initiate a ParallelBehaviour which will continue to execute until all its sub-behaviors terminate. One of the sub-behaviors is searching for agents who've registered service of "player". It is done by creating a *DFSubscriber* with handlers for *onRegister* and *onDeregister*. When a player is found (*onRegiser*), it's added to a list of waiting players. Whenever there're enough players in the queue to initiate a game, by default 2, a new board will be created and a game will begin.

As a game begins the game master will create a behavior to pass the turn to one of the player-agents. The turn is passed by sending an *ACLMessage* with a performative of *INFORM* and an ontology of "turn". As the turn is being passed, the game master keeps listening for a request to position a coin on the board. An incoming position request is handled by the following logic:

- Check whether there's an active game, i.e. checking if the board has been initiated.
- Check whether the requested move is legal.

- If the move isn't legal, the game master disconfirms the requested placement in a message to the requesting player agent.
- If the move is legal, the game master checks if the game is finished
- If the game isn't finished, the game master informs the other player that it's their turn.
- If the game is finished, the game master informs all the players that it's finished, resets the GameBoard, and attempts to begin a new game with the players in the waiting line.

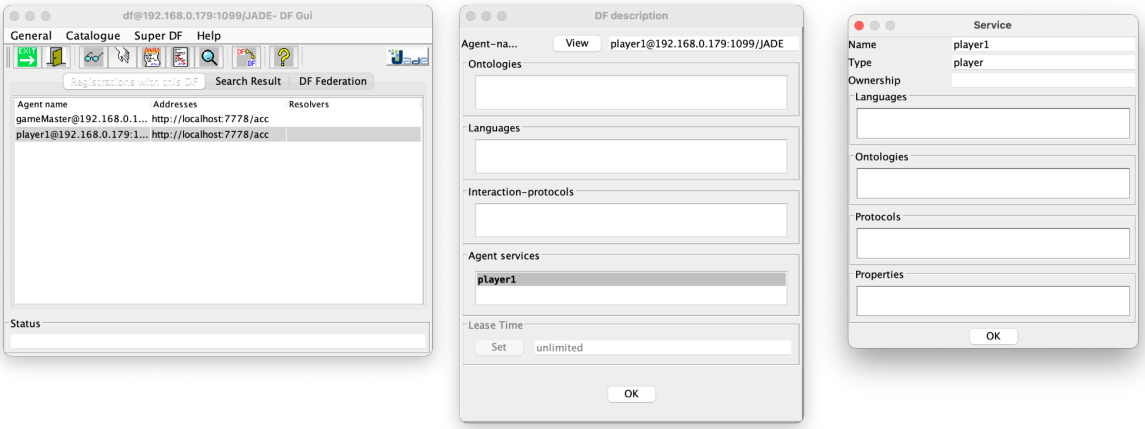
The process of searching for new players, initiating new games, and handling current games is continued until the game master itself deregisters.



The GameMaster class diagram

PlayerAgent

On creation, the player will register its service to the DF as a "player". It will then, in the same way as that of the game master, initiate a ParallelBehaviour which will continue to execute until all its sub-behaviors terminate.

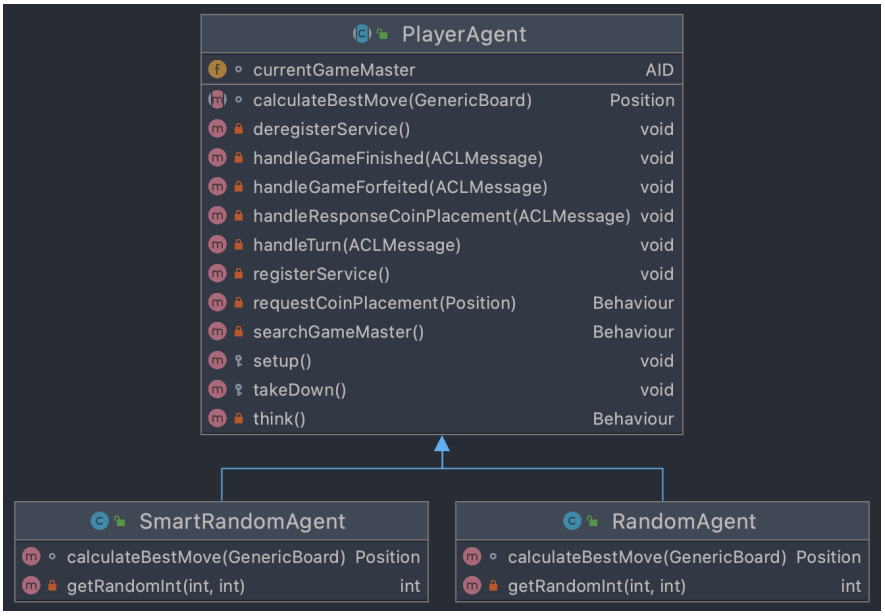


Example of how the DF has registered player1 as offering a service of type "player"

A player is a passive agent, relying on other agents to pass information it can act on. It doesn't contain any knowledge of the current state of the game until it's revealed in a serialized message from the game master. Because of this, most of the player's behaviors are listening and handling messages. A player listens and handles the following messages:

- Message informing agent it's its turn
- Message confirming/disconfirming the requested coin placement
- Message informing the game is finished
- Message informing the game has been forfeited by opponent agent

Most of the messages are handled in the same way by the superclass PlayerAgent, except for the decision of where to place a coin based on the board's state.



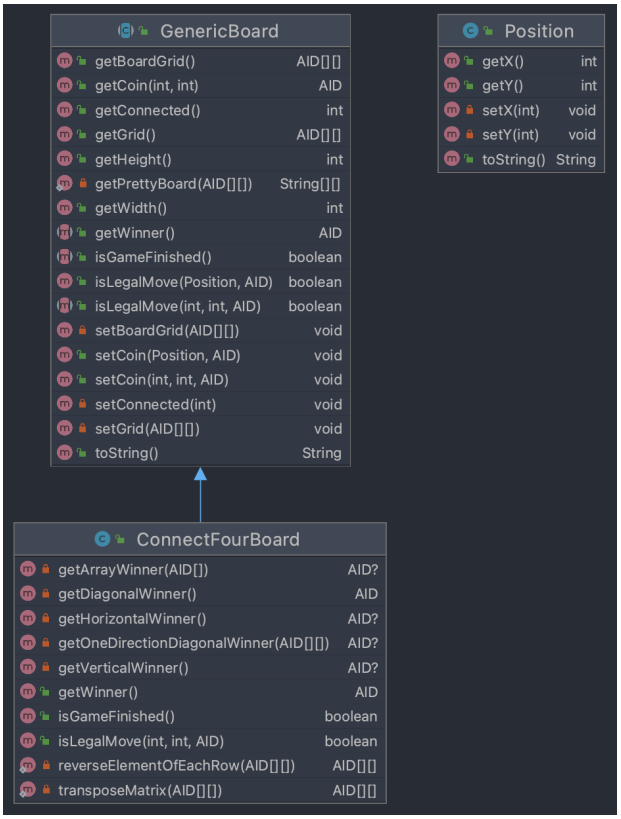
The PlayerAgent, SmartRandomAgent, and RandomAgent class diagram

The various kinds of player agents are RandomAgent and SmartRandomAgent. A RandomAgent will try to place a coin within the dimensions of the board and pass this position to the game master without validating its legality. This usually results in a long loop of disconfirms from the game master, followed by subsequent random placements. In contradiction, a SmartRandomAgent will create random placements until it finds a valid one, and then request that placement from the game master.

GameBoard

The ConnectFourBoard implements the abstract GenericBoard class. The generic version of the board allows any dimension of the board greater than 1x1, any amount of required connected coins within the range of the dimension, and offers methods for setting, getting, and validating the board state.

The board is represented by a 2D array containing agents' IDs. Whenever a coin is placed in the grid, the GameMaster checks whether the game is finished. A finished game is determined by checking if there's a horizontal, vertical, or diagonal winner. These functions leverage matrix characteristics to validate if the board has reached a terminal state.



The GenericBoard, ConnectFourBoard, and Position class diagram

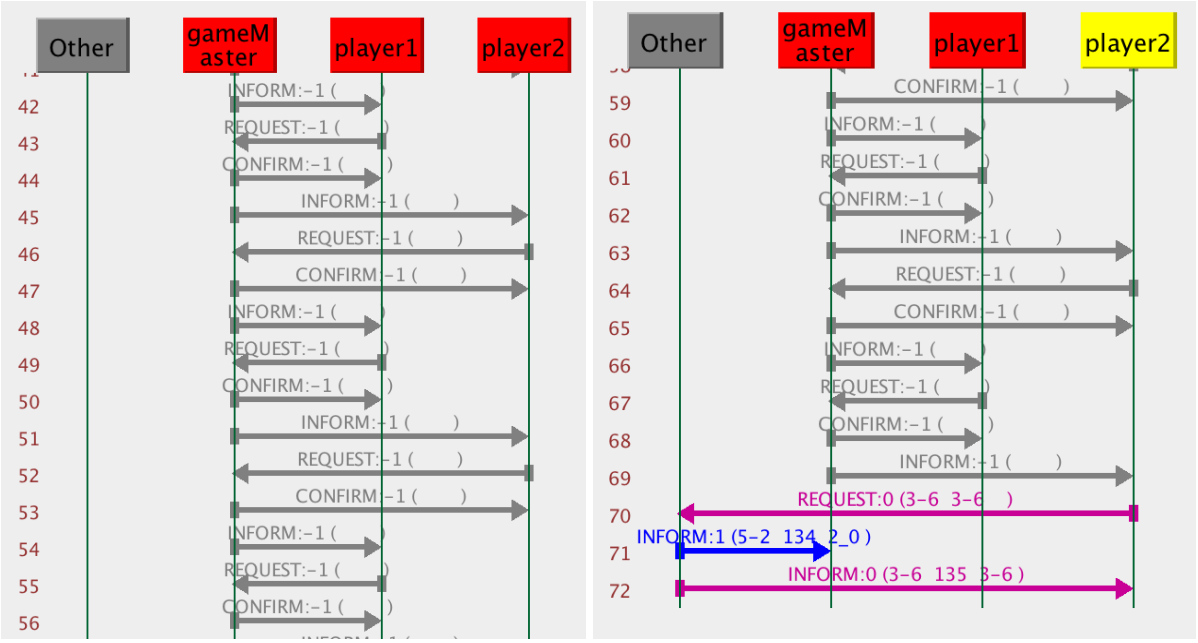
Environment

The agents are spawned within a platform and container, created when first running the program. When running the program, a platform and container are created, and one SmartRandomAgent and a GameMaster is then spawned within the container. Multiple agents can be spawned manually from the terminal or from the Remote Management Agent (RMA).

Templates
The templates used across the project are stored in GameMessageTemplates. The class exposes static templates primarily used within listen-behaviors. This makes it easier to abstract away and reuse common logic.

SerializableUtils
Since the board's passed from the GameMaster to the PlayerAgent, it must be serialized. In order to not try/catch the assignment/retrieval of serialized content to a message, a utility was created. The utility handles the assignment/retrieval of serialized content in a generic way and handles the potential error which might occur.

Game loop



Visualization of a game loop presented by the Sniffer-agent

By leveraging the *Sniffer*-agent, it's possible to visualize the game loop. In the example above, a game master and two players have been spawned within the same container. The figure to the left visualizes the loop where a player is informed that it's its turn, the player agent requests a placement, the placement is confirmed, and the game master passes the turn to the next player. In the figure to the right, the game reaches a terminal state and the losing player, player2, deletes itself in despair.

Evaluation & discussion

This project demonstrates the intuitive way of programming a game in the agent-oriented programming paradigm. It is clear that the idea of different autonomous and intelligent agents, interacting with each other in a structured manner, fits the dynamics of a game perfectly. This makes it easy to implement such game in an AOP language since thinking of it as an agent that represents the rules of the game itself and agents that represent the players playing the game is very intuitive.

Particular advantages are observed in terms of modifiability, maintainability and serviceability since every agent stands on its own so can be changed on its own and can be taken down for modifications without breaking the system. This project has shown as well that in an AOP paradigm, creating general systems that are easy to modify in the future is more natural than in other programming paradigms since everything (every agent) is automatically autonomous so not tightly integrated with each other.

AOP allows you to reason about one agent at a time without keeping other agents in mind, which is a lot easier to comprehend. This in turn favors important software requirements such as clearness, correctness, and verifiability.

The only thing connecting agents is their communication which is done as well in a structured manner. Interoperability-wise, this means that if all agents respect this structured way of communicating, they can interoperate with each other perfectly, independent of anything else (e.g. how they are implemented).

Comparison with other implementations

When comparing the implementation developed in this project to other open-source implementations in different programming paradigms, one can confirm the above-mentioned advantages of AOP compared to the other programming paradigms. It is often a lot harder to comprehend the implementation, which makes the implementation less clear and makes it harder to check for correctness. Modifications require an understanding of the whole codebase and making changes to this whole codebase since everything is tightly integrated with each other.

Of course, it is possible to create the Connect Four game in another programming paradigm while performing as well on above mentioned non-functional requirements (NFRs) as this project. The point is that in AOP, this is the natural result of programming the game, it is hard to not perform well on these NFRs whereas in the case of other programming paradigms, it’s the other way around. In other programming paradigms, it is hard to perform great on these NFRs and it’s your own responsibility.

Conclusion

To conclude, this project has demonstrated the advantages of using an agent-oriented programming paradigm when programming a game. The AOP paradigm suits the dynamics of a game very naturally and has a lot of advantages when looking into non-functional requirements. This combination of an easy-to-reason and use programming paradigm that automatically contains reliable performance on a lot of NFRs makes AOP a very promising programming paradigm for programming games, as well as broader, for every real-world interaction.

References

Calegari, Roberta, and Andrea Omicini. *Agents in Jade*. Slides for the Multi-Agent Systems course at Unibo. 2022.

Omicini, Andrea. *Drivers for Intelligent Systems*. Slides for Multi-Agent Systems course at Unibo. 2022.

Tilab. *Java Agent DEvelopment Framework: Jade Site*, https://jade.tilab.com/. Accessed 23 June 2022.

UltraBoardGames. “How to play Connect Four Shots | Official Game Rules.” *UltraBoardGames*, https://www.ultraboardgames.com/connect4/shots-game-rules.php. Accessed 23 June 2022.