Royal Game of Ur

[student ids] April 2024

1. **Overview**

# Requirements

This project required us to create a digital version of the Royal Game of Ur, featuring a graphical user interface. The game needed to prevent illegal moves and recognise when a player had won. The Royal Game of Ur’s exact rules are not known, so we were required to select one of many different rule sets to implement. The player should be given the option to play against another person over a local network or play against an AI.

# Team Communication

At our first meeting, we broke the project into three parts: networking, building the interface, and AI. This way, each person could concentrate on a specific area but still help the project move forward together.

We had regular team meetings to share updates. If someone needed help or had questions about another’s work, they could have reached out to the team using a WhatsApp group.

Besides talking during meetings, we also focused on writing clear docstrings and commentaries in our code as well as writing clear commit messages. This helped everyone understand the code better, making it easier for us to work together.

In short, talking often and clearly, documenting our code well, and the right exploitation of git were crucial for our team communication.

1. **Design**

The design section provides a comprehensive breakdown of the folder structure and architectural components of the project. It outlines the organization of various directories and modules, shedding light on the rationale behind their arrangement. Additionally, it offers insights into the flow of the system through visual aids such as flowcharts, enhancing understanding of the project’s overall architecture and design principles.

# General structure

Our team organised our project in the following folders:

* + - **bin**: Contains compiled Java classes.
    - **build**: Contains build and run scripts.
    - **lib**: External libraries used in the project.
    - **src**: Source code files in Java format.
      * **ai**: Classes related to AI agents.

∗ **agent**: Classes implementing various AI agents.

* + - * **board**: Classes related to the game board.
      * **controller**: Classes responsible for controlling game logic.

∗ **action**: Classes representing game actions.

* + - * + **game**: Classes representing game-related actions.
        + **menu**: Classes representing menu-related actions.
      * **exceptions**: Custom exception classes.
      * **game**: Classes representing the game and player logic.
      * **main**: Main class to run the program.
      * **player**: Classes related to player management.
      * **server**: Classes related to server-client communication.
      * **states**: Classes representing different game states.
      * **ui**: User interface classes.
    - **tests**: Unit tests for the project.

Diagram below explains how does our program work.

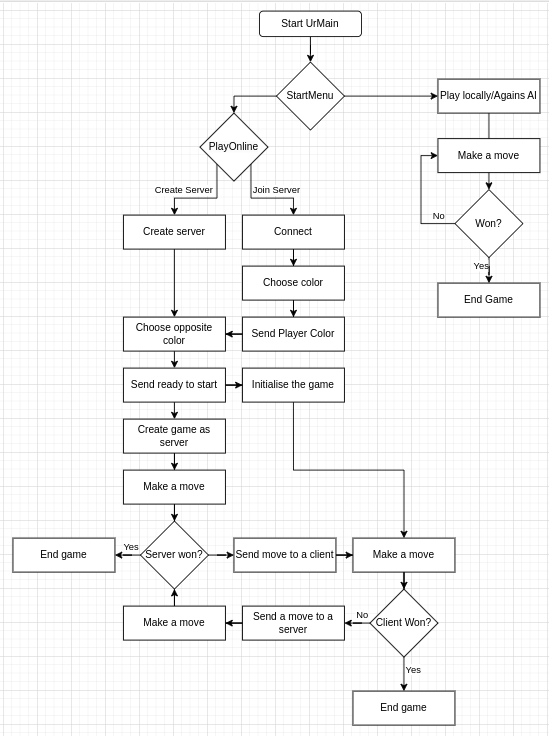


Figure 1: Game Flow Chart

# Using of records

In our project we used java records a lot as they help to model plain data aggregates with less ceremony than normal classes. Our program contains such

data aggregates as

* + - **BoardLayout**: Describes a board layout.
    - **PlayerOptions**: Record for player setup options, e.g. colour, human
    - **StartMenuClosed**: Describes a board layout.
    - **MenuClosedEventSourse**: Passed with MenuClosed event as source

# Abstract classes

The advantage of utilizing abstract classes in your Java project, as demonstrated through the files provided below, lies in the establishment of a robust framework that enhances code organization, reusability, and scalability. Each abstract class we have — Agent, Metric, Menu, PlayerController, and MoveSelected—serves a distinct purpose in structuring your project’s architecture. Here’s how each contributes to the overall design

## Agent

**Purpose:** Serves as a base for different types of AI agents (e.g., Rando- mAgent, GreedyAgent, ExpectiminimaxAgent) in a game.

**Advantage:** Facilitates the addition of new agent strategies without al- tering the game’s core logic. It abstracts common functionalities (such as storing player and game references) and requires subclasses to implement the specific decision-making logic (determineNextMove). This approach allows for easy scaling and testing of different AI behaviors under a unified interface.

## Metric

**Purpose:** Abstracts the concept of evaluating game states according to different criteria, useful in AI decision-making or game analytics.

**Advantage:** Offers a flexible structure for defining various game state evaluation metrics (e.g., maximizing advancement, post-board position- ing) essential for AI strategies. Implementing specific metrics becomes straightforward, with each adhering to a common contract to score game states. This design supports extensibility and enables the easy addition of new metrics as the game evolves.

## Menu

**Purpose:** Acts as a base for various menu interfaces in the game’s UI, interfacing with MenuController.

**Advantage:** Promotes UI consistency and simplifies the management of different menus (e.g., main menu, options menu) by centralizing common elements (like event handling setup) in one place. It allows for shared func- tionalities among all menus, reducing redundancy and facilitating changes across all menus through a single modification point.

## PlayerController

**Purpose:** Bridges the model (Player) with the view, controlling the game’s flow based on player actions.

**Advantage:** Abstracts the common functionalities of player controllers, whether AI or human, enabling a unified way to handle game actions (like move execution and validation). This separation of concerns enhances code maintainability, allowing for different player types to be managed under a consistent framework.

## MoveSelected

**Purpose:** Represents an abstract action event signaling the selection of a move, providing a clear mechanism for responding to user interactions or AI decisions within the game.

**Advantage:** While not an abstract class, it illustrates the use of ab- straction in event management, enabling a flexible and extendable way to handle various game actions seamlessly.

1. **Networking**

The networking aspect of the project involved setting up communication be- tween clients for multiplayer functionality. Our team decided to use TCP/IP- based protocol due to its reliability and ease of implementation. Eventually, we end up with the following cycle flowchart.

1. **GUI**

Creating a user-friendly graphical user interface (GUI) was crucial for ensuring an enjoyable gaming experience.

1. **AI**
2. **Testing**
3. **Evaluation**
4. **Conclusion**
5. **References**

* h[ttps://www.baeldung.com/ja](http://www.baeldung.com/java-record-keyword)v[a-record-k](http://www.baeldung.com/java-record-keyword)eyword

1. **Appendix**