Royal Game of Ur

[student ids] April 2024

# 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.

# Design

The design section provides a breakdown of the folder structure and architec- tural components of the project. It outlines the organization of various directo- ries and modules. Additionally, it explains the flow of the system through the flowchart. It also explains different code design decision we have made during the project development.

## 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.

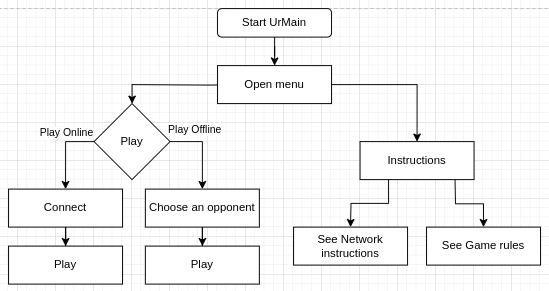


Figure 1: General Flowchart

## 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
* **PlayerPieceOnTile**: Contains information about a piece move, includ- ing the player, the tile moved from, and the tile moved to.
* **GameStartedEventSource**: Holds player setup options for the event of the game starting.
* **GameStartedWithServerEventSource**: Contains player setup options and a server action listener for the event of the game starting with a server.
* **GameStartedAsClient**: Similar to the above, but for starting the game as a client.
* **GameStash**: Holds information about the last roll and the piece moved.
* **Message**: Represents a message with a type and associated data.

## 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.

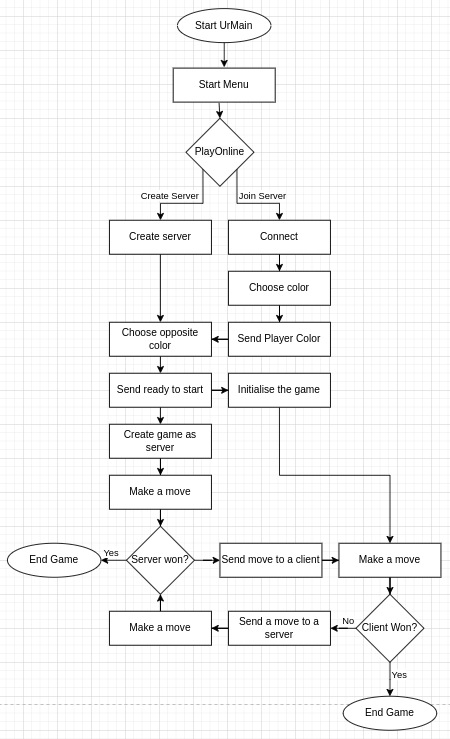
### MoveSelected

**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.

# 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.



6

Figure 2: Enter Caption

# GUI

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

# AI

The specification required the program to provide an AI opponent for a user to play against.

The implementation we provide allows the behavior of the opponent to be configured by both changing the evaluation process used (the AI’s ‘agent’) as well as their concept of a favorable game state (the agent’s ‘metric’). Some agent types are inherently ‘easier’ opponents, but where there is scope to do so, ‘Easy’ and ‘Hard’ difficulty versions are provided.

## RandomAgent

The simplest agent in the game is the RandomAgent who will choose randomly from the set of valid move options available to them at their turn. As with all players, in the case where there is no valid move option, the turn ends without any piece being moved.

## GreedyAgent

The GreedyAgent determines their next move as being that in their set of valid moves which will provide the ‘best’ expected game state (as determined by the selected metric) at the end of the current move. The GreedyAgent is imple- mented as a special case of the ExpectiminimaxAgent who only searches their game tree to the depth of the current move.

## ExpectiminimaxAgent

The most sophisticated agent is the ExpectiminimaxAgent who uses the expec- timinimax algorithm to recursively generate and traverse the game tree to a depth determined by the agent’s difficulty. ‘Harder’ difficulty agents will search the tree to a greater depth than their ‘easier’ counterparts.

### Expectiminimax

**Objective:** As per its name, the expectiminimax agent uses the expectimini- max algorithm to determine their next move. Expectiminimax is a variant of the minimax algorithm, which extends to stochastic games by its accommodation of random game events, e.g. a dice roll.

The minimax algorithm attempts to deduce an optimal move for a player given the current state of the game, a state evaluation metric and the player’s preference for the value of a state evaluated by this metric (i.e. maximal or

minimal metric value), and knowledge of the available transitions between game states (i.e. valid player moves). Expectiminimax extends the concept of a game state’s utility to the player by considering the likelihood that it will be reachable by the player due to the influence of random events on the game’s progression. The result of the expectiminimax algorithm is therefore the move a player should make that leads to the ‘best’ terminal state assuming optimal play by their opponent attenuated by the probability of the player being able to reach

this state.

**Implementation:** The expectiminimax algorithm works recursively, gener- ating a game tree up to a terminal depth, allocating values to the tree’s leaves and then backtracking, filling in parent node values by selecting the child node with the most desirable value for the opposing players.

The expectiminimax tree infixes ‘chance nodes’ with alternating maximizer and minimizer nodes. As can be deduced from the name, these nodes will select maximally/minimally valued child nodes at each level of the tree. These maximizer/minimizer nodes stand in place for the game’s players (say Max and Min) who are both assumed to play optimally at each turn. Game states awarded low values by the agent’s metric are those preferable to Min, and those given higher values are preferable to Max. By alternating these nodes in the tree (as the players alternate their turns in the game), the move chosen by the current player is the one that will lead to their best terminal state for an optimal opponent. If the opponent fails to play optimally, evaluation of the tree in the player’s subsequent turn from the progressed game state will select moves leading to even ‘better’ states than the ‘best worst-case’ reached with an optimal opponent.

Due to the stochastic element of ‘The Game of Ur’ (the dice roll), the ex- pectiminimax tree includes ‘chance nodes’ that do not feature in the minimax trees generated for non-stochastic games. The tree’s chance nodes correspond to the game’s random dice roll event and work to scale the utility of a game state by its probability by taking a probability-weighted average of the values of its children.

Due to the exponential increase in the size of the game tree with depth, attempting to evaluate the tree significantly deeply would take considerable time and hence violate the requirement of the AI to respond promptly. The evaluation depth is capped to a reasonable level to address this. To further aid the efficiency of the agent, the game tree generated by the agent is pruned as it is explored. The process of pruning eliminates nodes in the tree from consideration that will not influence the outcome of the algorithm. Although pruning does not resolve the algorithm’s exponential order, it does reduce it by some constant factor.

The methods of pruning a minimax algorithm game tree (most notably alpha-beta pruning) are only a partial solution to the problem of pruning an ex- pectiminimax tree due to the inclusion of chance nodes. The difficulty in pruning these ‘chance nodes’ is due to their value being calculated as the probability- weighted average of their child minimizer/maximizer nodes. By placing bounds on the possible values allocated to these child nodes (by the agent’s metric), we

can bound the average of these nodes and hence bound the value of the chance node without considering each of its children. By considering the ‘best’ (as concerns the chance node’s parent maximizer/minimizer node) possible value of the chance node currently being examined given the knowledge of the actual values of the node’s children that have been explored, and comparing this to the value of the node’s ‘best’ sibling, for whom we know the actual value, we can determine if further exploration of the current chance node will potentially yield a ‘better’ value than their sibling. If, even in the best case of child nodes, the chance node will not outscore its best sibling, then we do not consider it further.

The maximizer and minimizer nodes are pruned using the standard al- pha/beta technique which allows us to prune nodes which we can deduce will never be selected for as their parent (or grandparent etc.) has the choice of a ‘better’ child.

The value of alpha is the highest-valued node encountered so far in the tree; the value of beta is the lowest. These values are updated as the recursive calls return / we backtrack through the tree. If a node considered for the player Max is of a smaller value than alpha, then it will never be chosen for and so can be pruned. If a node considered for the player Min is of greater value than beta, then it will also never be chosen for and so can be pruned. Thus, we eliminate the need to examine nodes which present themselves as being of lower utility for the player.

# Testing

For testing we used deprecated version of our program that haven’t had any design on it. This way it was easier to see if something went wrong.

## Multiplayer testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test description | | Input | Expected Output | Actual Output |
| Test1: Roll test | | Press the roll button | Highlight all available  cells | Worked as expected |
| Evidence1 | | | | |
| Test2: Server move a piece test | | Press on a tile on a server turn | Output servers‘s tile  on a cell and pass the turn to the other player | Worked as expected |
| Evidence2 | | | | |
| Test3: Client move a piece test | | Dice roll — Press on a tile on a client turn | Output client‘s tile on  a cell and pass the turn to the other player | Worked as expected |
| Evidence3 | | | | |
| Test4: Client brings its  piece to the end of the board | | Dice Roll — Press on the 18th tile | Player2: 1 on the  18 tile and appropriate log message 6 | Worked as expected |
| Evidence4 | | | | |
| Test5:  roll | Handle 0 dice | Dice Roll 0 | Pass the turn to the  other player | NullPointerException |
|  |  | 10 |  | Evidence5 - console output |

Table 1: Test results

|  |  |  |  |
| --- | --- | --- | --- |
| Test6: Handle 0 dice  roll (fixed) | Dice Roll 0 | Pass the turn to the  other player | works as expected |
| Evidence6 - Log output | | | |
| Test7: Server brings its  piece to the end of the board | Dice Roll — Press on the 16th tile | Player1: 1 on the  16 tile and appropriate log message | Worked as expected |
| Evidence7 - GUI + terminal | | | |

Table 2: Test results (continued)

# Evaluation

dr

# Conclusion

1. **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
* h[ttps://www.w3sc](http://www.w3schools.com/java/javaabstract.asphttps)ho[ols.com/ja](http://www.w3schools.com/java/javaabstract.asphttps)va/ja[va*abstract.asphttps*](http://www.w3schools.com/java/javaabstract.asphttps): *//docs.oracle.com/javase/tutorial/java/IandI/ab*

1. **Appendix**