Konobi

Software Development Methods project

Lorenzo Basile, Irene Brugnara, Roberto Corti, Arianna Tasciotti

Introduction

Our project

The goal of our project was to develop a command line version of **Konobi**, a board game for two players. The project also contains a client-server version of the game.

Our project

The goal of our project was to develop a command line version of **Konobi**, a board game for two players. The project also contains a client-server version of the game.

What tools did we use?

- Java 15
- Gradle
- TravisCI
- Git & GitHub

Konobi

Konobi is a drawless game and it can be played either on a go board or a chess board.

Two players, black and white, take turns at placing stones of their color on the board, starting with black. The aim of the players is to build chains of connected stones of their color.

Konobi

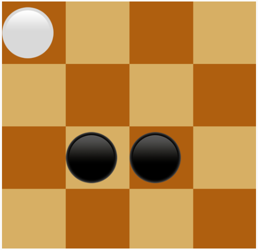
Konobi is a drawless game and it can be played either on a go board or a chess board.
Two players, black and white, take turns at placing stones of their color on the board, starting with black. The aim of the players is to build chains of connected stones of their color.

The game is won by the first player who connects the two opposite edges of the board.

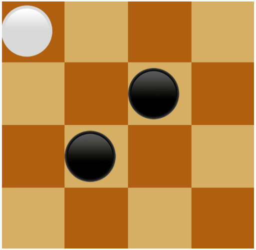
- Black: top $\leftrightarrow$ bottom
- White: left $\leftrightarrow$ right

Connections

Two like-colored stones can be:



Strongly connected



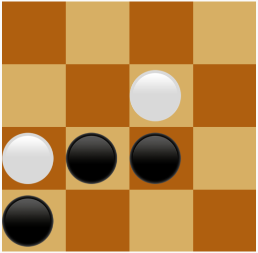
Weakly connected

A chain is a set of connected stones

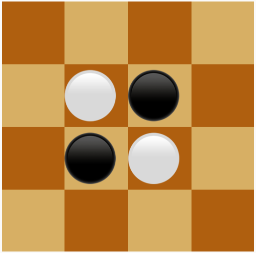
Placement rules

Not all moves are allowed:

- **Weak connections** to a certain stone are illegal unless it is impossible to make a placement that is both strongly connected to that stone and not weakly connected to another
- **Crosscut** placements are always illegal



Legal weak connection



Crosscut placement

Additional rules

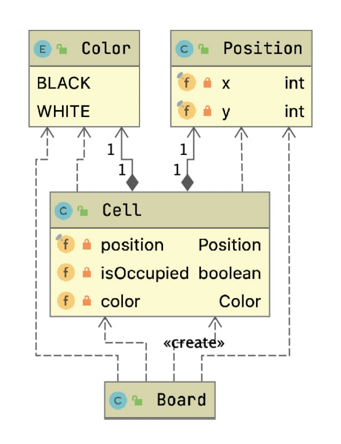
- **Pie rule**: at his first move, white can decide to switch colors with black instead of making a move
- **Mandatory pass**: if a player cannot make a move (because of placement restrictions), he has to pass

Basic entities

Cell

`Cell` represents the basic building block of the board

- `Position position`
- `Color color`
- `boolean isOccupied`



Cell class

Cell

When a cell is constructed it is empty: no color is associated to it and `isOccupied=False`, when a stone is placed in the cell a color is set and `isOccupied=True`.

Development history:

- From value `NONE` in enum `Color` to field `isOccupied` in class `Cell`
- Removed `Stone` data class

Board

A `Board` is represented by a set of Cells and extends `HashSet<Cell>` by overriding the `dimension()` method

Reasons for this choice of data structure:

- Usage of streams
- `Position` as field of `Cell`

The constructor of Board creates a set of empty cells

Connections

Strong connections

- We implemented a concept of orthogonal adjacency which only depends on the relative positions of two stones: the euclidean distance is 1
- Given the set of orthogonally adjacent stones, we implemented a method that filters only those with the same color and returns the set of strongly connected stones

Weak connections

- Same concept used for weak connections: two stones are diagonally adjacent if their square euclidean distance is 2.
- Then to obtain the set of weakly connected stones it is also necessary to filter out diagonally adjacent stones with common strong neighbors

Rules

Rule

Once provided all methods related to strong and weak connection into `Board` our next step was to provide a way to check whether a move is valid or not and to announce whether there is a chain.

Initially a `Rules` class was implemented...

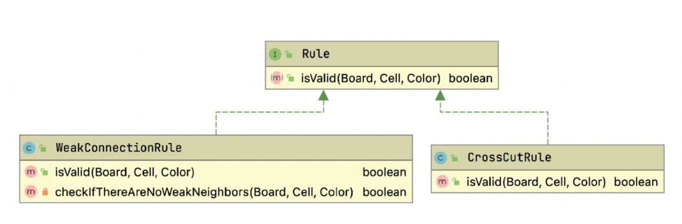
Rule

From commit eaba694

```java
public class Rules {
    Board board;

    public Rules(Board board) {
        this.board = board;
    }

    private boolean isLegalWeakConnectionPlacement(Cell cell) {
        Set<Cell> weakNeighbors = board.weakConnectionsOf(cell);
        Color stoneColor = cell.getColor();
        cell.reset();
        boolean condition = weakNeighbors.stream()
                                         .map(c->c.orthogonalNeighborsIn(board.cells))
                                         .anyMatch(s->s.stream()
                                                       .filter(c->!c.isOccupied())
                                                       .anyMatch(c->checkIfThereAreNoWeakNeighbors(c, stoneColor)));
        board.placeStone(cell.getPosition(), stoneColor);
        return !condition;
    }

    private boolean checkIfThereAreNoWeakNeighbors(Cell cell, Color stoneColor){
        board.placeStone(cell.getPosition(), stoneColor);
        Set<Cell> weakConnectionsOfCell = board.weakConnectionsOf(cell);
        cell.reset();
        return weakConnectionsOfCell.isEmpty();
    }

    private boolean isCrosscutPlacement(Cell cell) {
        Set<Cell> weakNeighbors = board.weakConnectionsOf(cell);
        Color stoneColor = cell.getColor();
        return weakNeighbors.stream()
                            .map(c->c.commonOrthogonalNeighborsWith(cell, board.cells))
                            .anyMatch(s->s.stream()
                                          .allMatch(c->c.isOccupied() && c.getColor()==stoneColor.oppositeColor()));
    }
```

Rule

Later we realized that there would be the possibility to abstract...



For a given `Board`, the `isValid` method will check if it is legal to place a stone of a given `Color` in the `Cell`.

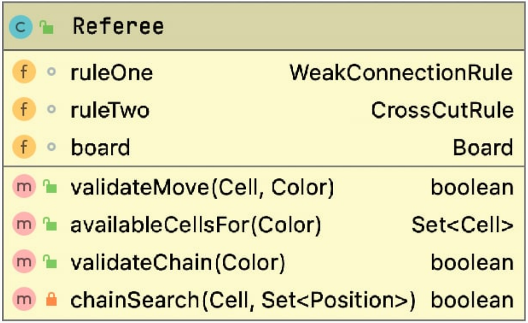
Having a `Rule` interface will allow the possibility to add new rules over the possible player's move.

Referee

Once implemented the logic of a valid move in the cell in `Rules` package our aim was to encapsulate all methods that could announce:

- a given move is legal w.r.t `WeakConnectionRule` and `CrossCutRule`
- the presence of a chain (i.e. end of the game)

Referee


Referee
ruleOne WeakConnectionRule
ruleTwo CrossCutRule
board Board
validateMove(Cell, Color) boolean
availableCellsFor(Color) Set<Cell>
validateChain(Color) boolean
chainSearch(Cell, Set<Position>) boolean