Konobi game

Software Development Method Project

Fallacara E., Indri P., Pigozzi F.

Introduction

The **goal** of our project is to implement the **Konobi game** in Java, giving also the user the opportunity to choose between two interfaces: **console version** or **GUI version**

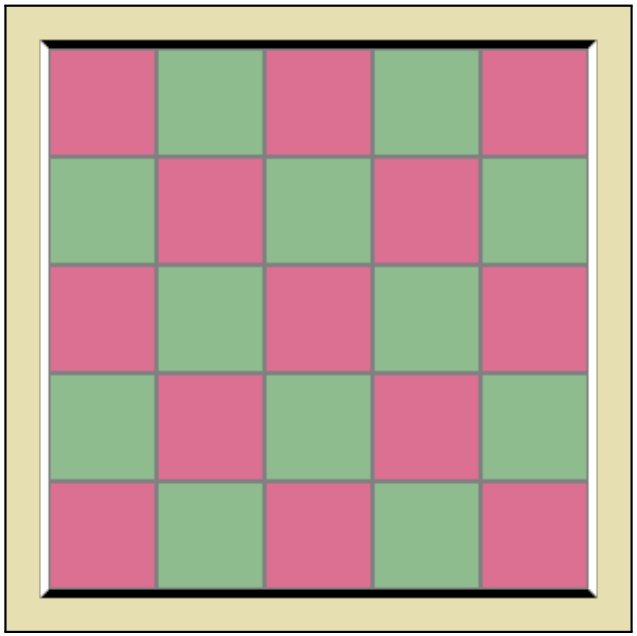
Tools

- IntelliJ;
- GitHub;
- Gradle: building;
- TravisCI: continuous integrations;
- JavaFX: ;
- Other?

Konobi

Konobi is a drawless connection game for two players: **Black** and **White**. It's played on the a square board, which is initially empty.

The top and bottom edges of the board are coloured black; the left and right edges are coloured white.



Konobi Rules

Starting with Black, the players take turns placing stones of their own color on empty points of the board, one stone per turn.

Konobi Rules

Starting with Black, the players take turns placing stones of their own color on empty points of the board, one stone per turn.

Two like-coloured stones are **strongly connected** if they are orthogonally adjacent to each other, and **weakly connected** if they are diagonally adjacent to each other without sharing any strongly connected neighbour.

Konobi Rules

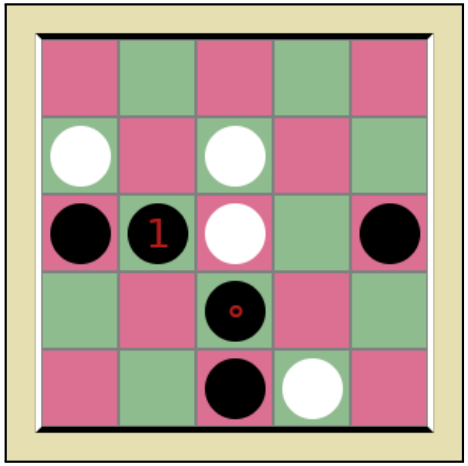
Starting with Black, the players take turns placing stones of their own color on empty points of the board, one stone per turn.

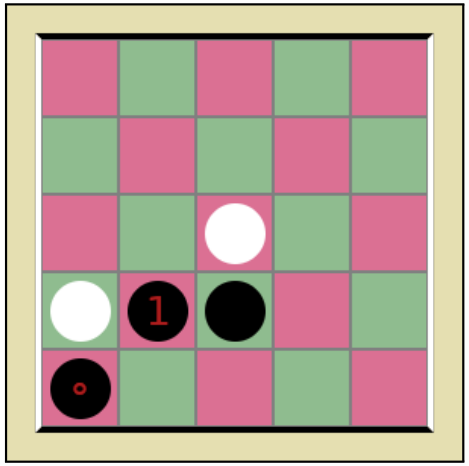
Two like-coloured stones are **strongly connected** if they are orthogonally adjacent to each other, and **weakly connected** if they are diagonally adjacent to each other without sharing any strongly connected neighbour.

It's **illegal** to make a weak connection to a certain stone unless it's impossible to make a placement which is both strongly connected to that stone and not weakly connected to another.

Legal and Illegal Moves

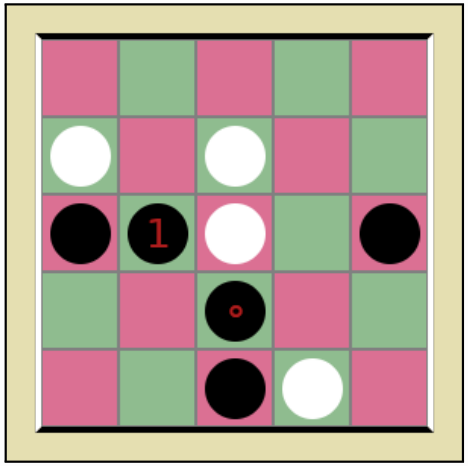
Legal moves:

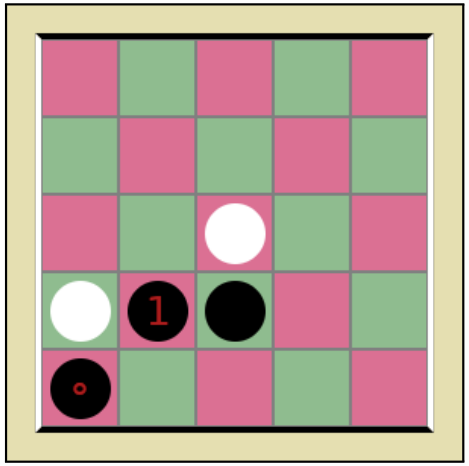




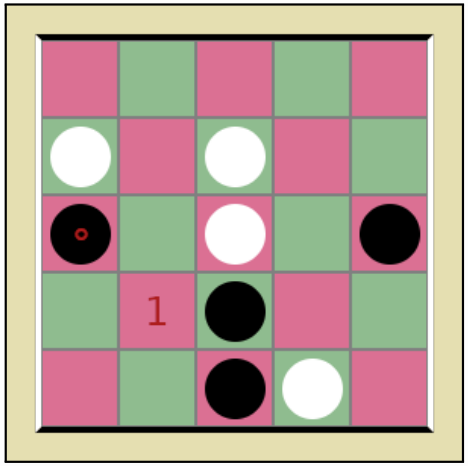
Legal and Illegal Moves

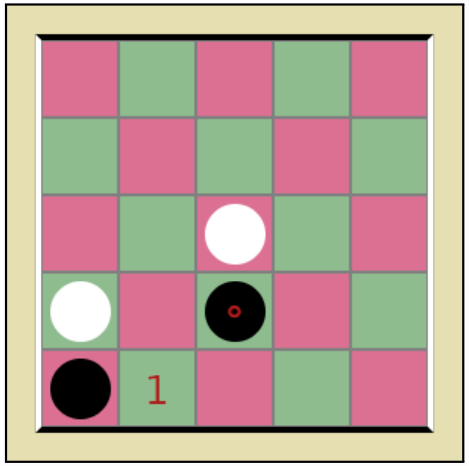
Legal moves:





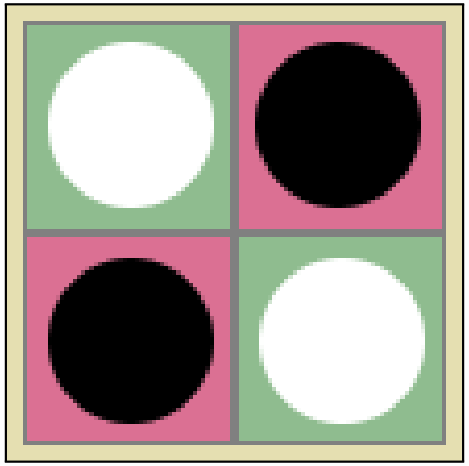
Illegal moves:





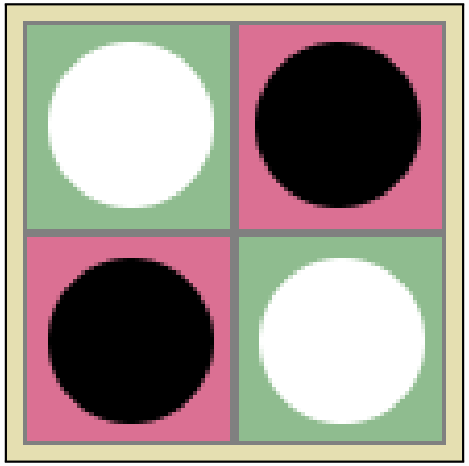
Konobi rules Cont.

It's also **illegal** to form a **crosscut**, i.e., a 2x2 pattern of stones consisting of two weakly connected Black stones and two weakly connected White stones.



Konobi rules Cont.

It's also **illegal** to form a **crosscut**, i.e., a 2x2 pattern of stones consisting of two weakly connected Black stones and two weakly connected White stones.



If a player can't make a move on his turn, he must **pass**. Passing is otherwise not allowed. There will always be a move available to at least one of the players.

Konobi rules Cont.

The **pie rule** is used in order to make the game fair. This means that White will have the option, on his first turn only, to change sides instead of making a regular move.

Konobi rules Cont.

The **pie rule** is used in order to make the game fair. This means that White will have the option, on his first turn only, to change sides instead of making a regular move.

The game is **won** by the player who completes a chain of his color touching the two opposite board edges of his color. **Draws are not possible**.

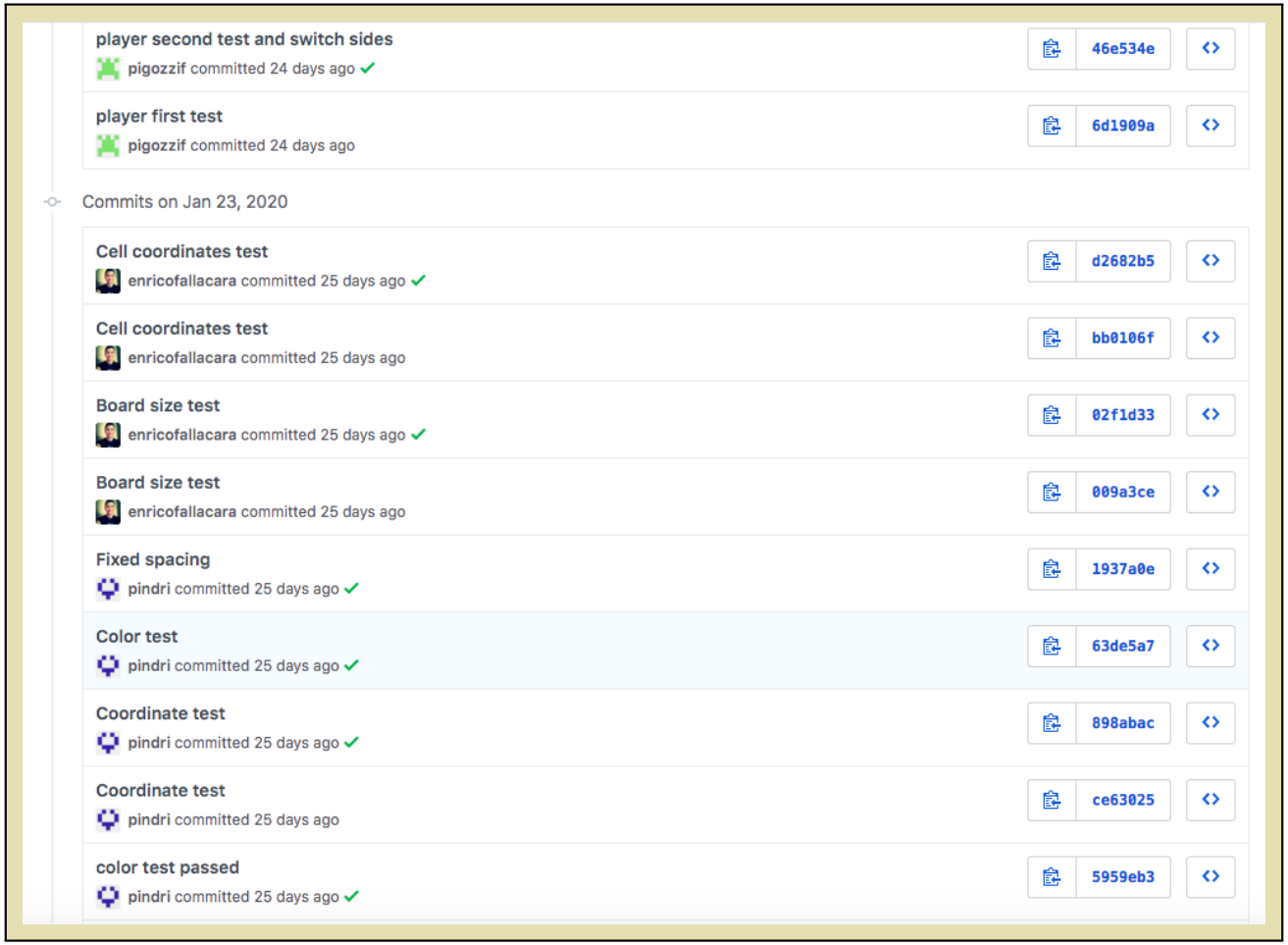
Building Blocks

Cell class is the fundamental building block of the game engine. It is associated to a **Colour**, and has a **Point** for the coordinates.

Board class is a collection of **Cell**s, and implements the **Iterable** interface. It conveys a notion of geometrical arrangement among the **Cell**s.

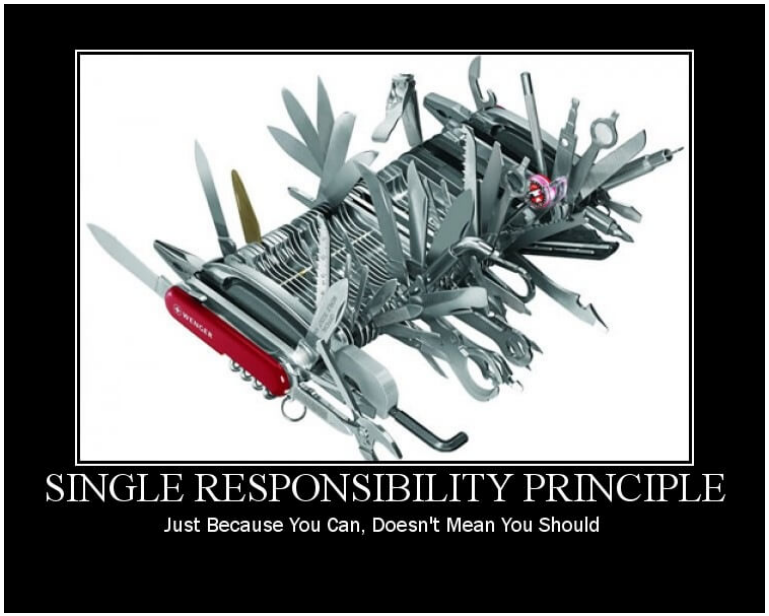
Player class represents each of the two players.

Building Blocks - TDD



Test Driven Development was adopted from the very onset, committing after every red-light/green-light pattern.

SRP and Board




```java
    public Stream<Cell> getMooreNeighbours(Point p, int level) {
        return Arrays.stream(
                slice(  Math.max(0, p.y - level),
                        Math.min(p.y + level + 1, size),
                        Math.max(0, p.x - level),
                        Math.min(p.x + level + 1, size))
                );
    }

    public static boolean isStrongNeighbour(Point target, Point query) { return manhattanDistance(target.x, query.x, target.y,

    public static boolean isWeakNeighbour(Point target, Point query) { return manhattanDistance(target.x, query.x, target.y, qu

    @SafeVarargs
    public final Stream<Cell> getNeighbours(Point point, int level, BiPredicate<Point, Point>... functions) {
        return getMooreNeighbours(point, level).filter(cell -> Arrays.stream(functions).allMatch(x -> x.test(point, cell.getCoo
    }

    @SafeVarargs
    public final Stream<Cell> getColoredNeighbours(Point point, int level, Color color, BiPredicate<Point, Point>... functions)
        return getNeighbours(point, level, functions).filter(x -> x.hasThisColor(color));
    }

    public boolean isOnBoard(Point point){
        return (0<= point.x && point.x < size) && (0 <= point.y && point.y < size);
    }

    public boolean isOnEndingEdge(Point point, Color color) {
        return (color == Color.white) ? point.x == size - 1 : point.y == size - 1;
    }

    private static double manhattanDistance(int x1, int x2, int y1, int y2) { return Math.abs(x1 - x2) + Math.abs(y1 - y2); }
}
```

Board class was doing too much, so we performed a **refactor**...

Neighbourhood

...and created the **Neighbourhood** class. It shows a **Monostate Pattern**, having only static methods to compute different flavours of neighbourhoods from an instance of **Board** and a target **Point**.

Building Blocks Cont.

StatusSupervisor is in charge of holding the state of the game, and updating it whenever it changes (new move, pass rule, pie rule).

It is employed as an interface between the **UI** module and the **core** module, allowing the two to communicate without knowing anything of each other.

Rules

The package **Rules** contains the true logic of the game. We started off by defining a class per rule, later to realize there was room for abstraction...

...we introduced **StatusSupervisor** as a **parameter object**, and allowed each of the classes to implement the **Rule** interface.

Each **Rule** can be queried by passing a **Supplier** for it to the **Rulebook**.

Rules Cont.

ValidPositionRule class had something wrong...

```java
public class ValidPositionRule implements Rule{
    private ArrayList<Rule> positionRules;

    public ValidPositionRule() {
        positionRules = new ArrayList<>(Arrays.asList(new EmptyRule(), new CrosscutRule(), new WeakRule()));
    }

    @Override
    public boolean isValid(Supervisor supervisor) {
        // TODO: spostare questo if in una nuova regola
        if(!supervisor.getBoard().isOnBoard(supervisor.getCurrentPoint()))
            return false;
        return positionRules.stream().allMatch(x -> x.isValid(supervisor));
    }
}
```

ValidPositionRule class had something wrong...

```java
public class ValidPositionRule implements Rule{
    private ArrayList<Rule> positionRules;

    public ValidPositionRule() {
        positionRules = new ArrayList<>(Arrays.asList(new EmptyRule(), new CrosscutRule(), new WeakRule()));
    }

    @Override
    public boolean isValid(Supervisor supervisor) {
        // TODO: spostare questo if in una nuova regola
        if(!supervisor.getBoard().isOnBoard(supervisor.getCurrentPoint()))
            return false;
        return positionRules.stream().allMatch(x -> x.isValid(supervisor));
    }
}
```

Rules Cont.

ValidPositionRule class had something wrong...

```java
public class ValidPositionRule implements Rule{
    private ArrayList<Rule> positionRules;

    public ValidPositionRule() {
        positionRules = new ArrayList<>(Arrays.asList(new EmptyRule(), new CrosscutRule(), new WeakRule()));
    }

    @Override
    public boolean isValid(Supervisor supervisor) {
        // TODO: spostare questo if in una nuova regola
        if(!supervisor.getBoard().isOnBoard(supervisor.getCurrentPoint()))
            return false;
        return positionRules.stream().allMatch(x -> x.isValid(supervisor));
    }
}
```

Violation was solved creating **ValidPositionRulesFactory** class, which follows the **Factory Pattern**.

Starting game

The console version of the game can be started using:

```
> ./gradlew runConsole
```

The GUI version of the game can be started using:

```
> ./gradlew runGUI
```

Console User Interface

- `ConsoleBoardWriter`: board display;
- `ConsoleCellRepresentation`: conversion between cell color and its representation;
- `ConsoleInputHandler`: player input handling;
- `ConsoleMessageWriter`: messages to the players.

Messages are contained in the `Messages` class: its messages are used by the GUI implementation as well.

Graphical User Interface

- `GUI`: implements the game flow in a `JavaFX` application;
- `GUIBoardWriter`: board and GUI display;
- `GUIAsker`: boh;
- `GUIMessageWriter`: messages to the players.

The `Events` package defines events for the rules (PieRule, PassRule and EndGameRule); the events are processed by the `Handlers` package, which handles mouse inputs as well.