# 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**.
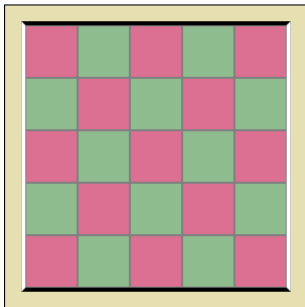
## Tools

- ▶ IntelliJ;
- ▶ OpenJDK11 and JavaFX;
- ▶ GitHub;
- ▶ Gradle: building;
- ▶ TravisCI: continuous integrations;
- ▶ Other?

# KONOBI GAME

# 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.

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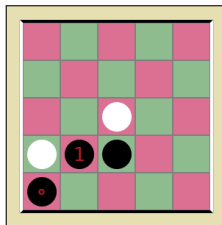
# Konobi Rules

**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.
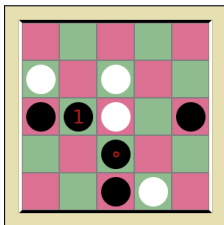
# Konobi Rules

**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.
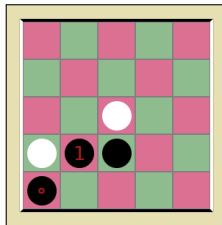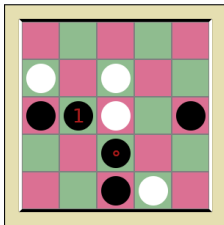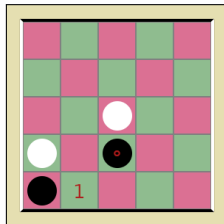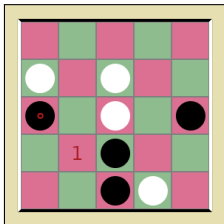
Legal moves:
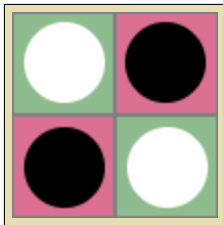
# Legal and Illegal Moves
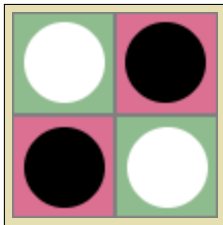
Legal moves:



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.

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.

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**.

# PROJECT STRUCTURE

The project is subdivided in two main packages:

► core
► user interface

The project is subdivided in two main packages:

- ▶ core
- ▶ user interface

The **core package** contains all the elements concerning the functional logic of the game.

# PROJECT STRCUTURE

The project is subdivided in two main packages:

► core
► user interface

The **core package** contains all the elements concerning the functional logic of the game.

The **UI package**, on the other hand, contains all the elements that are used to create the two different user interfaces: **command line** and **desktop interface**.

# CORE PACKAGE

# 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.

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

# SRP and Board



SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should



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

...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**.

**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...



```
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...



```
8   public class ValidPositionRule implements Rule{
9       private ArrayList<Rule> positionRules;
10
11      public ValidPositionRule() {
12          positionRules = new ArrayList<>(Arrays.asList(new EmptyRule(), new CrosscutRule(), new WeakRule()));
13      }
14
15      @Override
16      public boolean isValid(Supervisor supervisor) {
17          // TODO: spostare questo if in una nuova regola
18          if(!supervisor.getBoard().isOnBoard(supervisor.getCurrentPoint()))
19              return false;
20          return positionRules.stream().allMatch(x -> x.isValid(supervisor));
21      }
22  }
```



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

# UI Package

# 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.