

# KONOBİ 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

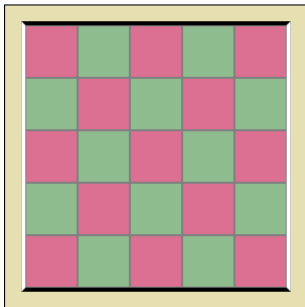
- ▶ IntelliJ;
- ▶ OpenJDK11 and JavaFX;
- ▶ GitHub;
- ▶ Gradle: building;
- ▶ TravisCI: continuous integrations;

# KONOBI GAME

# KONOBI

Konobi is a drawless connection game for two players: **Black** and **White**. It's played on 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

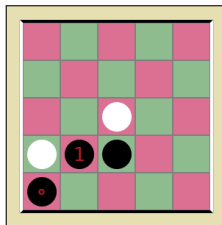
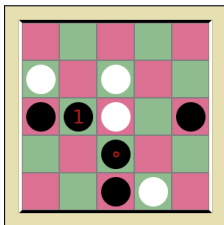
**Starting with Black**, the players take turns placing stones of their own color on empty points of the board, one stone per turn.

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

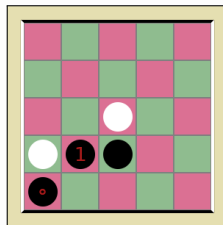
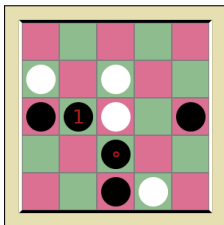
Legal moves:



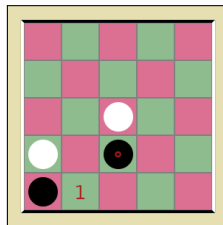
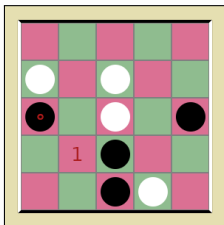


# LEGAL AND ILLEGAL MOVES

Legal moves:

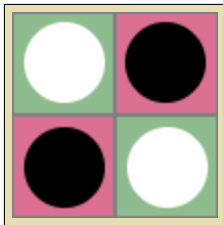


Illegal moves:



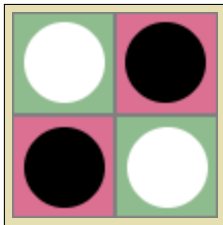
# KONOBI RULES CONT.

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.**

# PROJECT STRUCTURE

# PROJECT STRUCTURE

The project is subdivided in two main packages:

- ▶ `Core`;
- ▶ `UserInterface`.

# PROJECT STRUCTURE

The project is subdivided in two main packages:

- ▶ **Core;**
- ▶ **UserInterface.**

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



# PROJECT STRUCTURE

The project is subdivided in two main packages:

- ▶ **Core;**
- ▶ **UserInterface.**

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 **Cells**, and implements the **Iterable** interface. It conveys a notion of geometrical arrangement among the **Cells**.

**Player** class represents each of the two players.

# BUILDING BLOCKS - TDD

player second test and switch sides pigozzif committed 24 days ago ✓	 46e534e	
player first test pigozzif committed 24 days ago	 6d1909a	
Commits on Jan 23, 2020		
Cell coordinates test enricofallacara committed 25 days ago ✓	 d2682b5	
Cell coordinates test enricofallacara committed 25 days ago	 bb0106f	
Board size test enricofallacara committed 25 days ago ✓	 02f1d33	
Board size test enricofallacara committed 25 days ago	 009a3ce	
Fixed spacing pindri committed 25 days ago ✓	 1937a0e	
Color test pindri committed 25 days ago ✓	 63de5a7	
Coordinate test pindri committed 25 days ago ✓	 898abac	
Coordinate test pindri committed 25 days ago	 ce63025	
color test passed pindri committed 25 days ago ✓	 5959eb3	

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

# SRP AND BOARD



```
52 public boolean isStrongNeighbour(Point target, Point query) {
53     return Arrays.stream(
54         slice(Math.max(0, p.y - level),
55             Math.min(p.y + level + 1, size),
56             Math.max(0, p.x - level),
57             Math.min(p.x + level + 1, size))
58     ).anyMatch(x -> x != target.x || x != target.y);
59 }
60
61 public static boolean isStrongNeighbour(Point target, Point query) { return manhattanDistance(target.x, query.x, target.y, query.y) < 1; }
62
63 public static boolean isWeakNeighbour(Point target, Point query) { return manhattanDistance(target.x, query.x, target.y, query.y) < 2; }
64
65 @SafeVarargs
66 public final Stream<Cell> getNeighbours(Point point, int level, BiPredicate<Point, Point>... functions) {
67     return getMooreNeighbours(point, level).filter(cell -> Arrays.stream(functions).allMatch(x -> x.test(point, cell.getCoord())));
68 }
69
70 @SafeVarargs
71 public final Stream<Cell> getColoredNeighbours(Point point, int level, Color color, BiPredicate<Point, Point>... functions) {
72     return getNeighbours(point, level, functions).filter(x -> x.hasThisColor(color));
73 }
74
75 public boolean isOnBoard(Point point) {
76     return 0 <= point.x && point.x < size && 0 <= point.y && point.y < size;
77 }
78
79 public boolean isBorderingEdge(Point point, Color color) {
80     return (color == Color.WHITE ? point.x == size - 1 : point.y == size - 1);
81 }
82
83 private static double manhattanDistance(int x1, int x2, int y1, int y2) { return Math.abs(x1 - x2) + Math.abs(y1 - y2); }
84 }
```

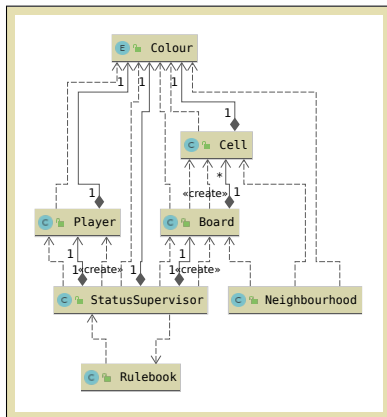
Board class was doing too much, so we performed an **Extract Class** refactor...

...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.

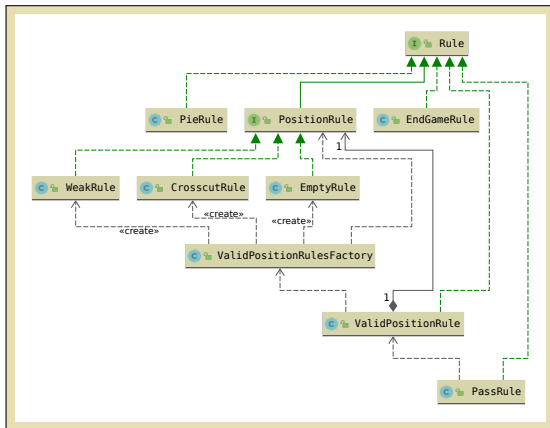
**StatusSupervisor** is in charge of holding the state of the game, and updating it whenever it changes (new move, pass rule, pie rule).

It is the middle-man that upper-level classes use to modify the status of the game.



# 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 **Preserve Whole Object**, allowing each of the classes to implement the **Rule** interface.





# 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 }
```



# RULES CONT.

ValidPositionRule class had something wrong...

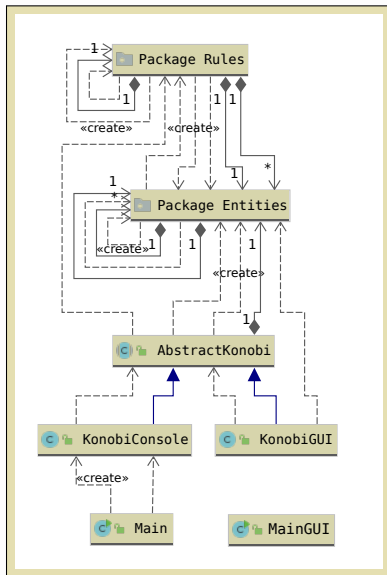
```
8 public class ValidPositionRule implements Rule{
9     private ArrayList<Rule> positionRules;
10
11     public ValidPositionRule() {
12         positionRules = new ArrayList<>(Arrays.asList(new EmptyRule(), new CrosscutRule(), new WeakRule()));
13     }
14
15     @Override
16     public boolean 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**), and PositionRule interface to be used inside it.

**AbstractKonobi** provides an abstraction for the game itself, containing functions to check and apply the various rules; it is extended by:

- KonobiConsole;
- KonobiGUI.



# UI PACKAGE

# INTERACTING WITH THE GAME

At first, we considered abstracting the console and the graphical interfaces with a common Java interface.

We realised this was leading us to *conceptualisation abuse*.

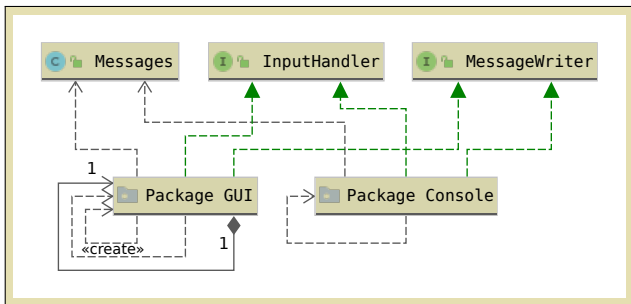
```
7  public interface UserInterface {  
8  
9      Point getInput(Player player);  
10     boolean askPieRule();  
11     void notifyEndGame(Player player);  
12     int askSize();  
13     void notifyPass();  
14     void display(Board board);  
15     void notifyInvalidMove();  
16     int initialize();  
17  
18 }
```

Implementation of `UserInterface` would have led to violations of the *SRP*.

The two interfaces are diverse enough, so we decided to create two distinct packages with different classes.

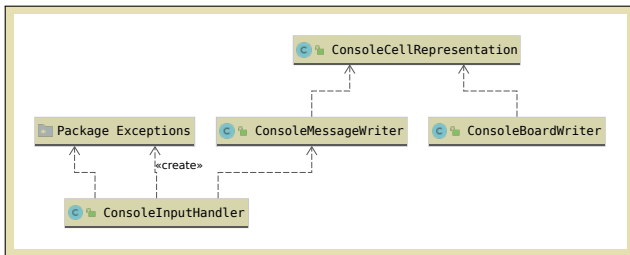
# COMMON ELEMENTS

The two interfaces do have something in common. They use the same **Messages** and they both implement an **InputHandler** to ask the users for input and a **MessageWriter** to communicate with them.



# CONSOLE USER INTERFACE

**ConsoleBoardWriter** handles the display of cells using **ConsoleCellRepresentation** to map the cell **Colour** to a symbol (convenient to avoid confusion with black/white console backgrounds).



# GRAPHICAL USER INTERFACE

The **GUI** class implements the game flow in a JavaFX application. **GUIBoardWriter** deals with the creation of an empty grid and its update after each move.

The **Events** package defines events for the rules (pie, pass and end-game rules); the events are processed by the **Handlers** package, which handles mouse inputs as well.

At each mouse input, **GUIMouseInputHandler** fires the rule events using **EventsFactory**.



# LONG METHOD SMELL IN GUI?

The GUI class is quite long: should this be regarded as a *Long method smell*?

```
117
118     Button rulesButton = createAndSetButton("Rules", width, height, (ActionEvent e) -> getHostServices().s
119
120     HBox hBox = new HBox();
121     hBox.getChildren().addAll(startButton, endButton, rulesButton);
122
123     pane.add(hBox, 0, 1);
124     hBox.setSpacing(15);
125     GridPane.setHalignment(hBox, HPos.CENTER);
126
127     Scene scene = new Scene(pane);
128
129     stage.setTitle("Konobi");
130     stage.setScene(scene);
131     stage.show();
132 }
133
134 @Override
135 public void stop() { Platform.exit(); }
136
137 public static void main(String[] args) { Application.launch(args); }
138
139 }
```

JavaFX applications are very verbose and (moderately) long methods should not be alarming.

# STARTING GAME

For portability, the project is shipped with the `gradlew` (`gradlew.bat` for Windows) executable to run the code without manually handling dependencies.

The console version of the game can be started using:

```
> ./gradlew runConsole
```

# STARTING GAME

For portability, the project is shipped with the `gradlew` (`gradlew.bat` for Windows) executable to run the code without manually handling dependencies.

The console version of the game can be started using:

```
> ./gradlew runConsole
```

The GUI version of the game can be started using:

```
> ./gradlew runGUI
```

# THANKS FOR YOUR ATTENTION

Thanks for your attention!

And now, let's test the project with a live demo.