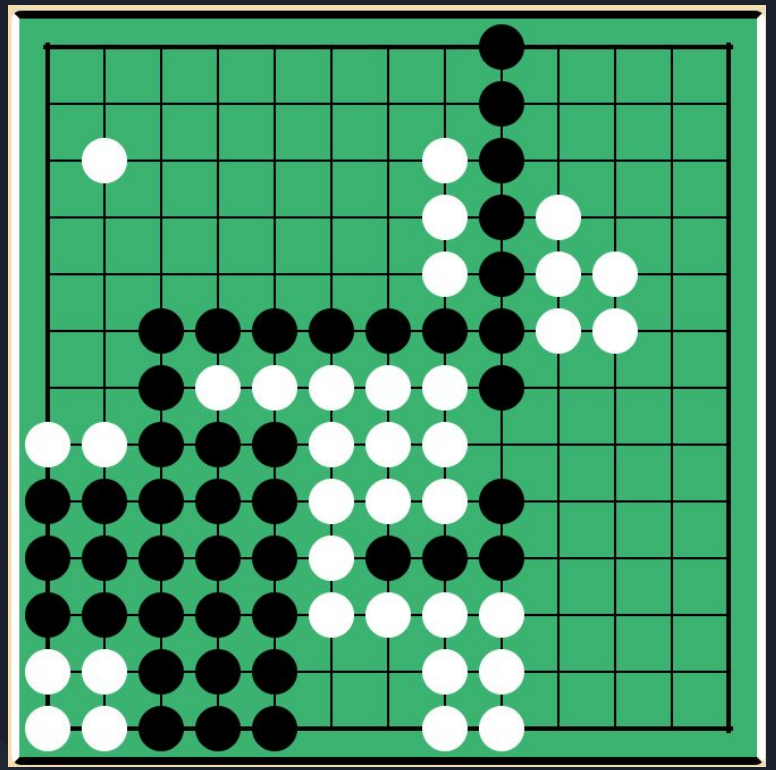
# Quentin Game



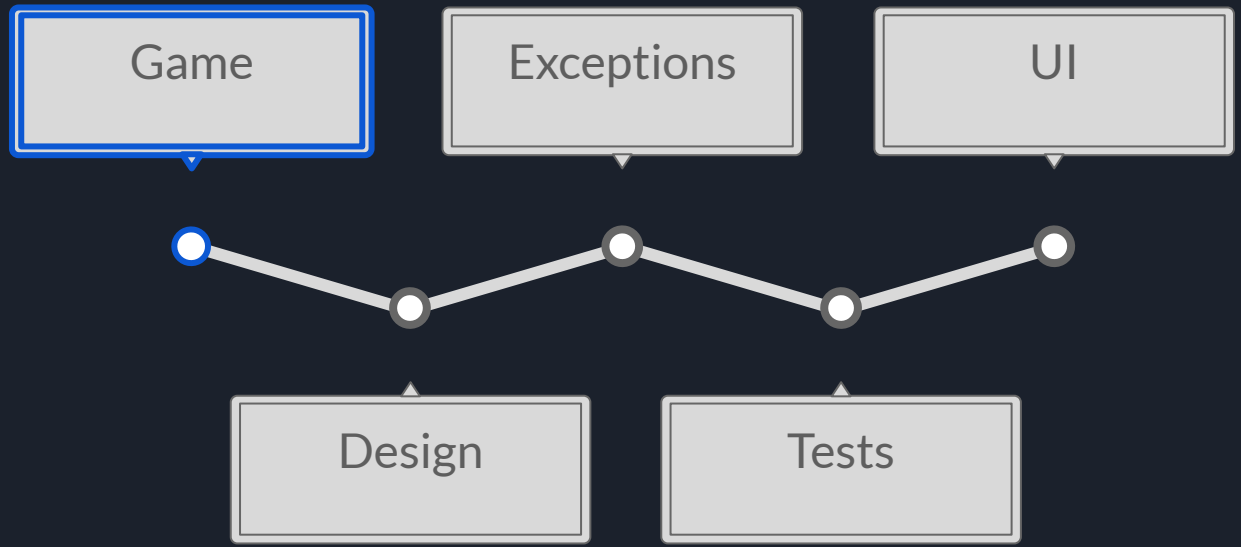
Eros Fabrici

Pietro Morichetti

Dogan Can Demirbilek

Ionut Alexandru Pascariu

Stefano Simonetto


Game
Exceptions
UI
Design
Tests

# Quentin Game Introduction

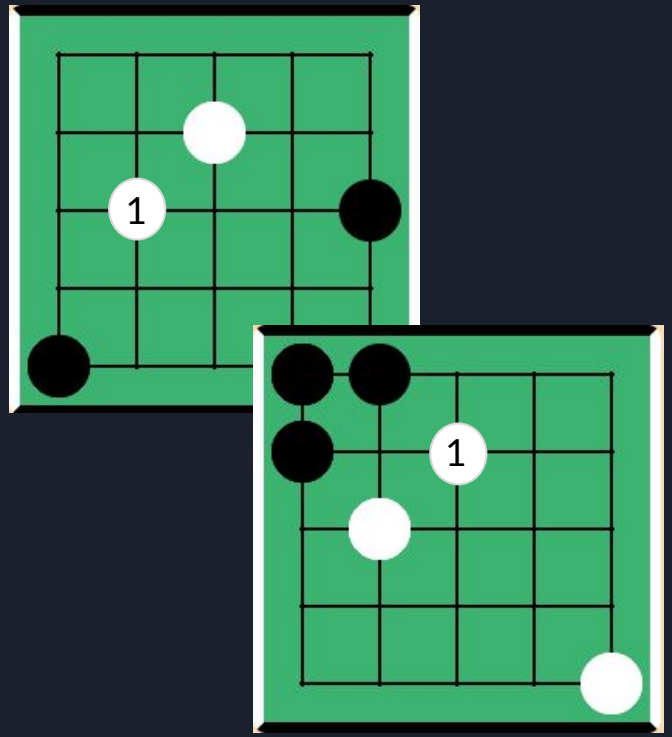
- Quentin is a drawless connection game for two players: Black and White.
- It's played on the intersections (points) of a square board, which is initially empty.
- The top and bottom edges of the board are colored black.
- The left and right edges are colored white.
- The game is won by the player who completes a chain of his color touching the two opposite board edges of his color.

# Rules Explanation: Illegal Move

Move is illegal if two diagonally adjacent colour alike stones do not share a colour alike orthogonal stone.

Legality of the moves are checked at the **end of the turns**, meaning that after territories are filled, if above rule was not followed, move is illegal and filled territory is reverted.

In both examples, White 1 is illegal with respect the previously mentioned rule.



# Rules Explanation: Pie Rule

The pie rule is used in order to make the game fair.
That means white player has the opportunity to switch colour with the opposite player, instead of making a regular move and just during its first turn.

NO


YES

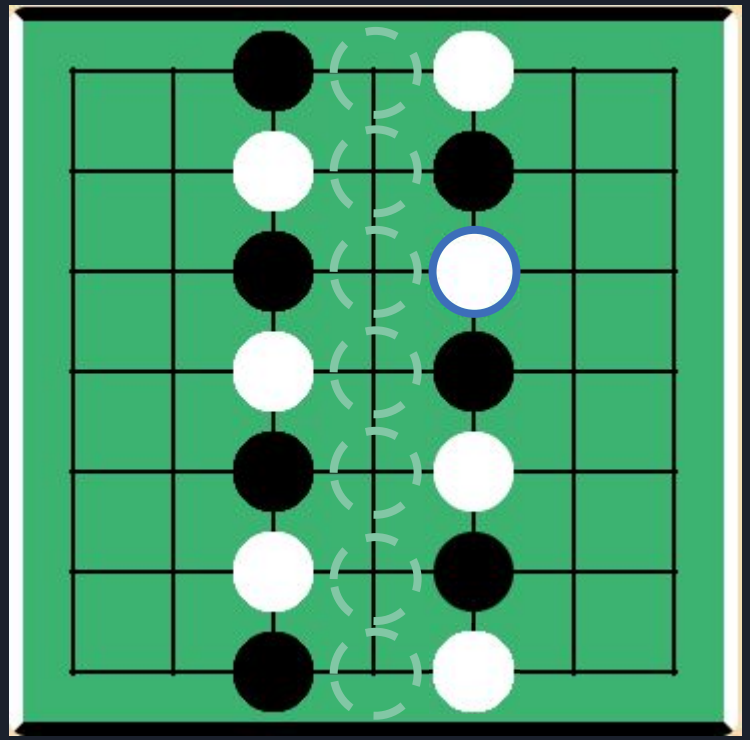
# Rules Explanation: Pass Rule

- If a player cannot 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.

# Rules Explanation: Regions and Territories

A **region** is a set of orthogonally adjacent empty points completely surrounded by stones or board edges.
If all those points are orthogonally adjacent to at least two stones, said region is also a **territory**.

After each turn, every territory is filled with stones of the player who has the majority number.
Territories with the same number of black and white stones are filled with stones of the opponent's colour.



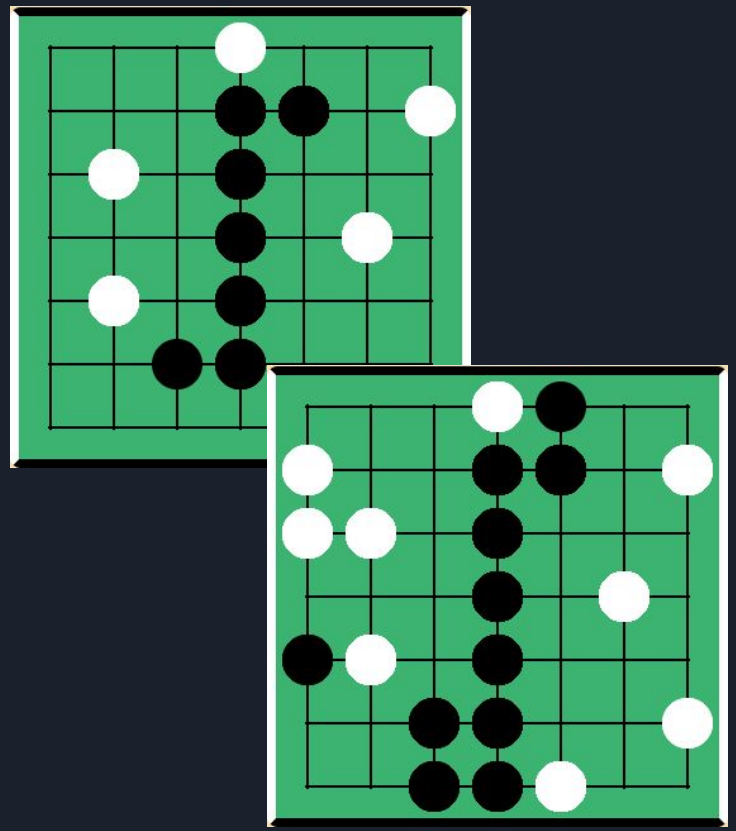
# Rules Explanation: Chain and Winning

A **chain** is a set of like-colored, orthogonally adjacent stones and they are necessary to match the win condition.

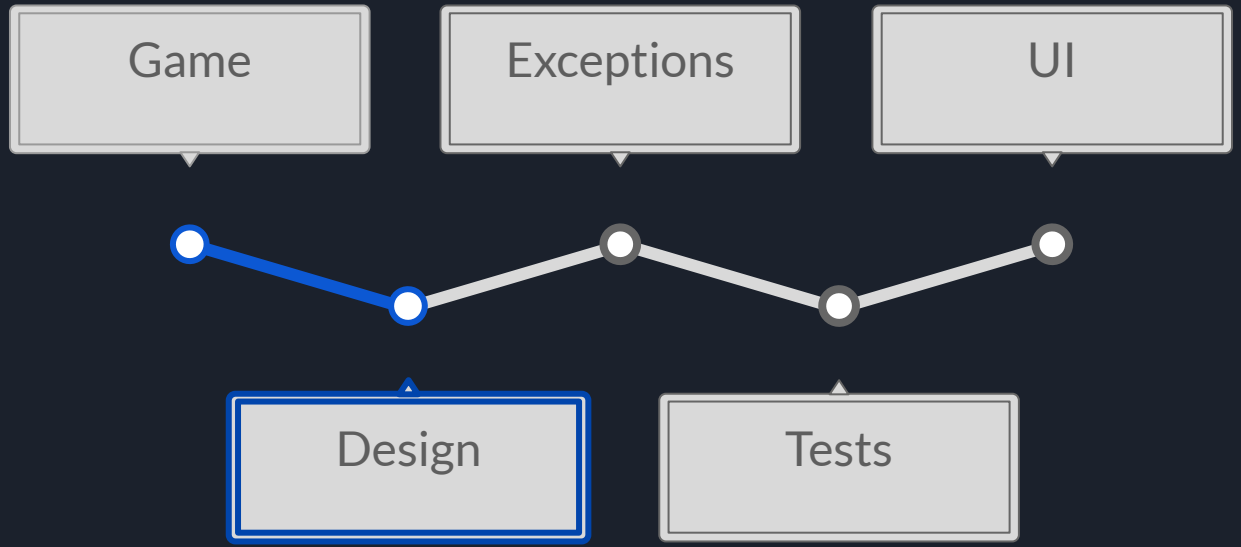
To win the game, one of the two players has to create a chain starting from one side and reaching the opposite side of the board.

White Player - left ⟺ right

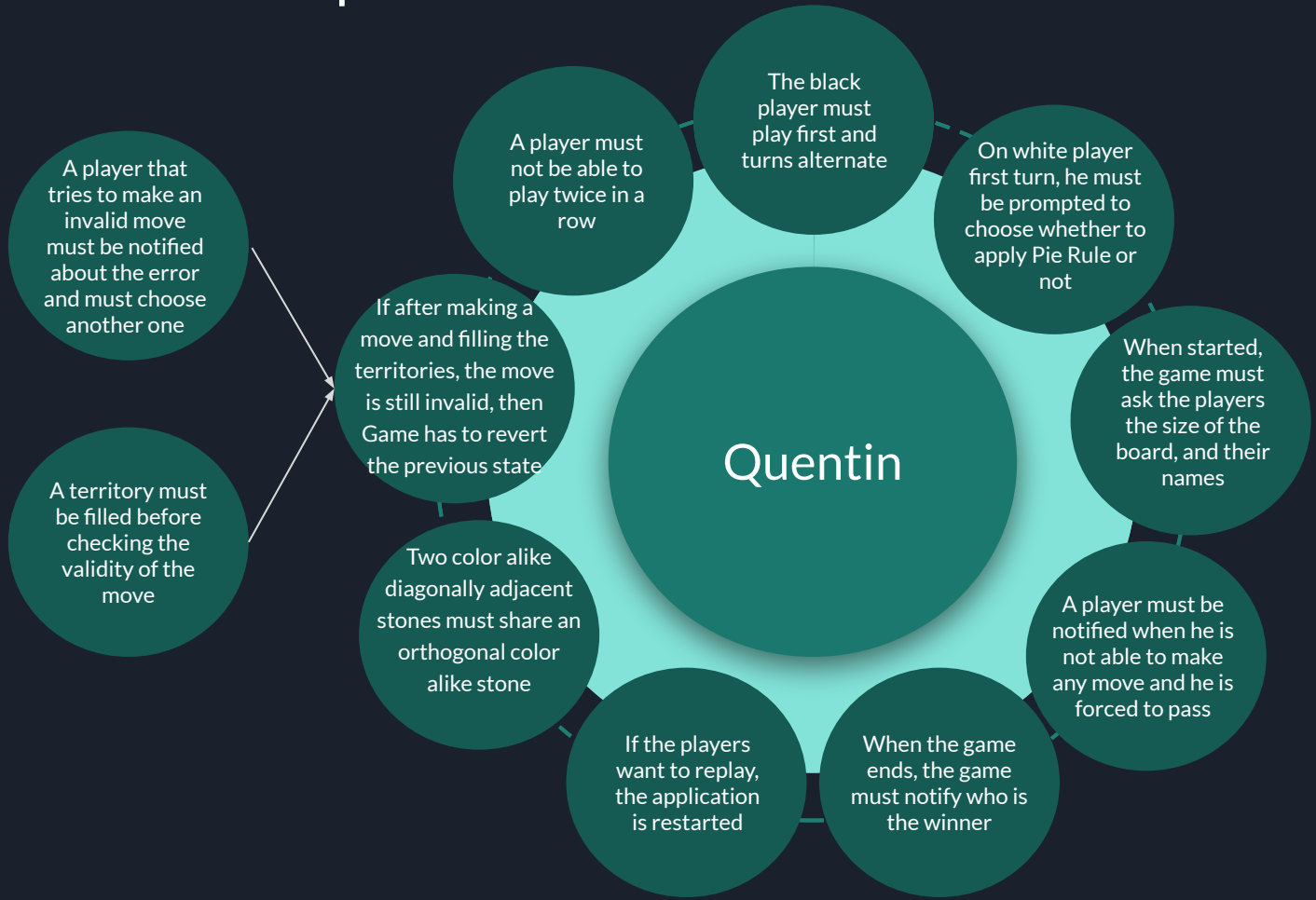
Black Player - top ⟺ bottom



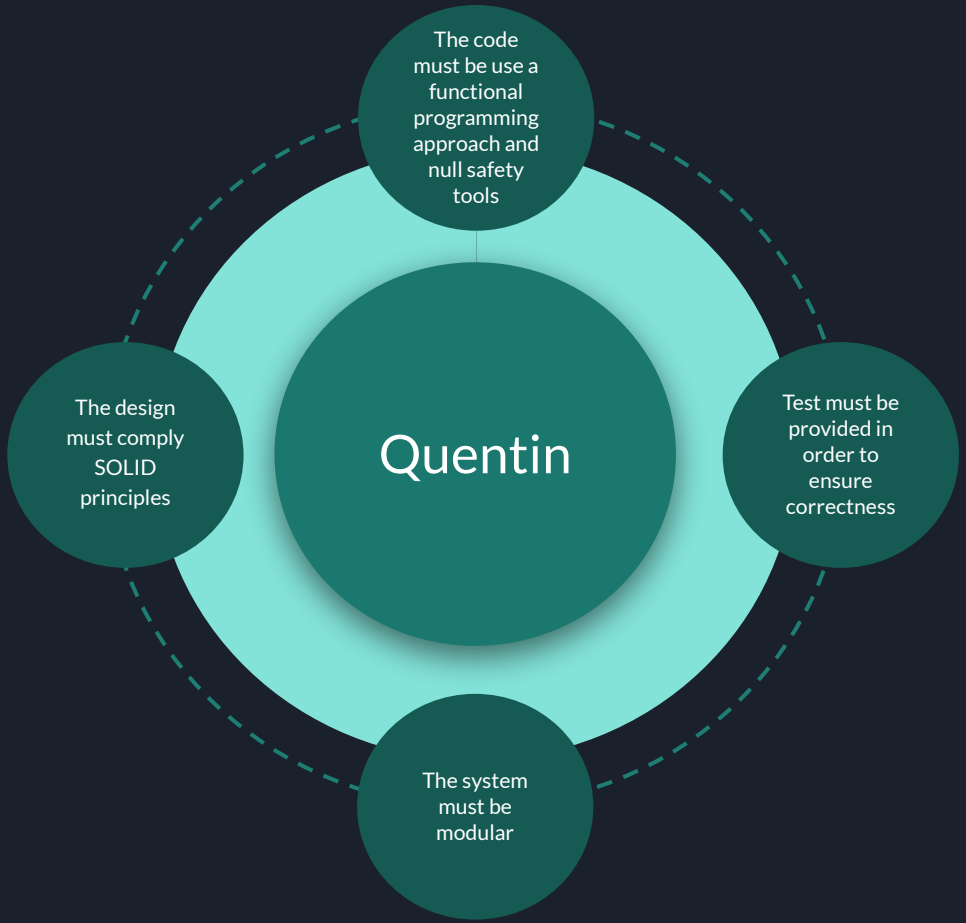
# LIVE DEMO


Game
Exceptions
UI
Design
Tests

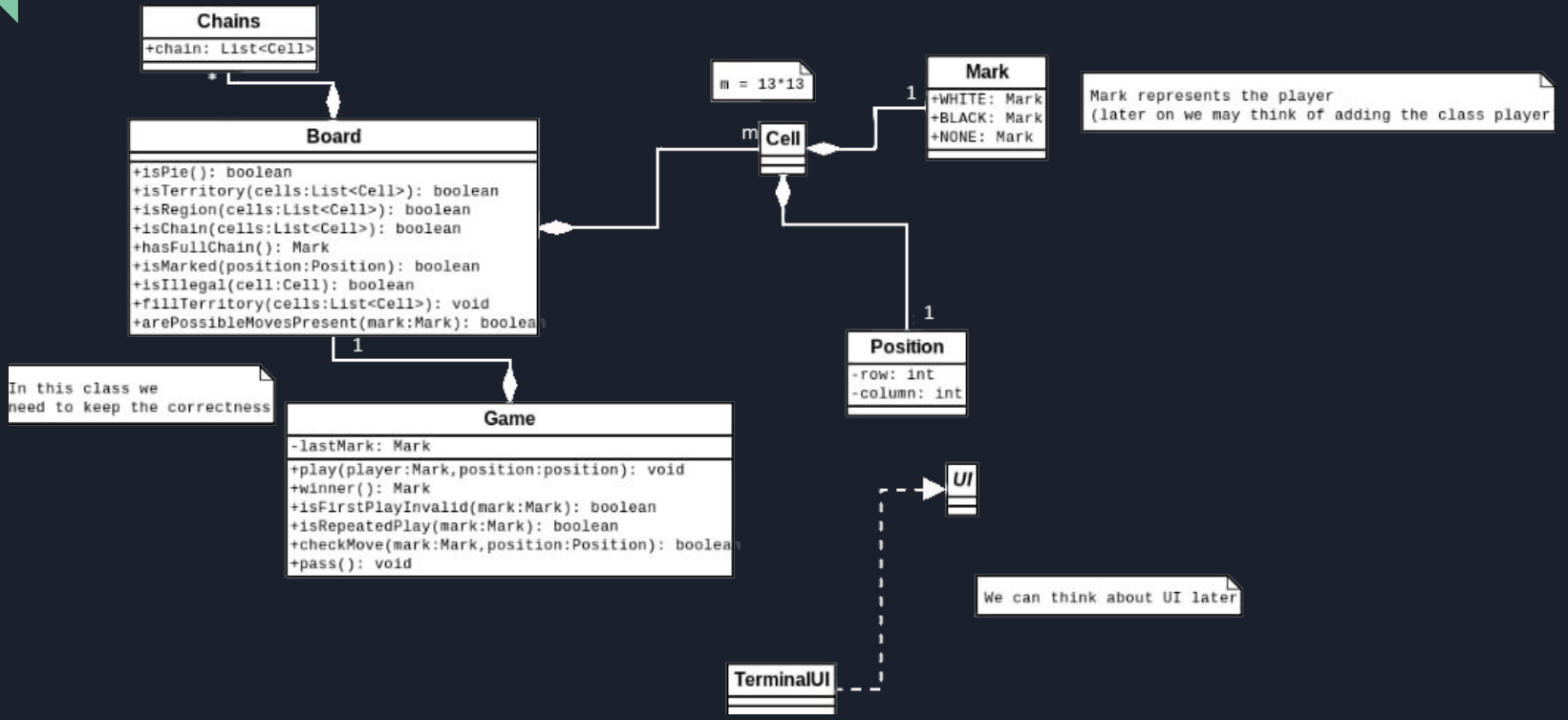
# Functional Requirements



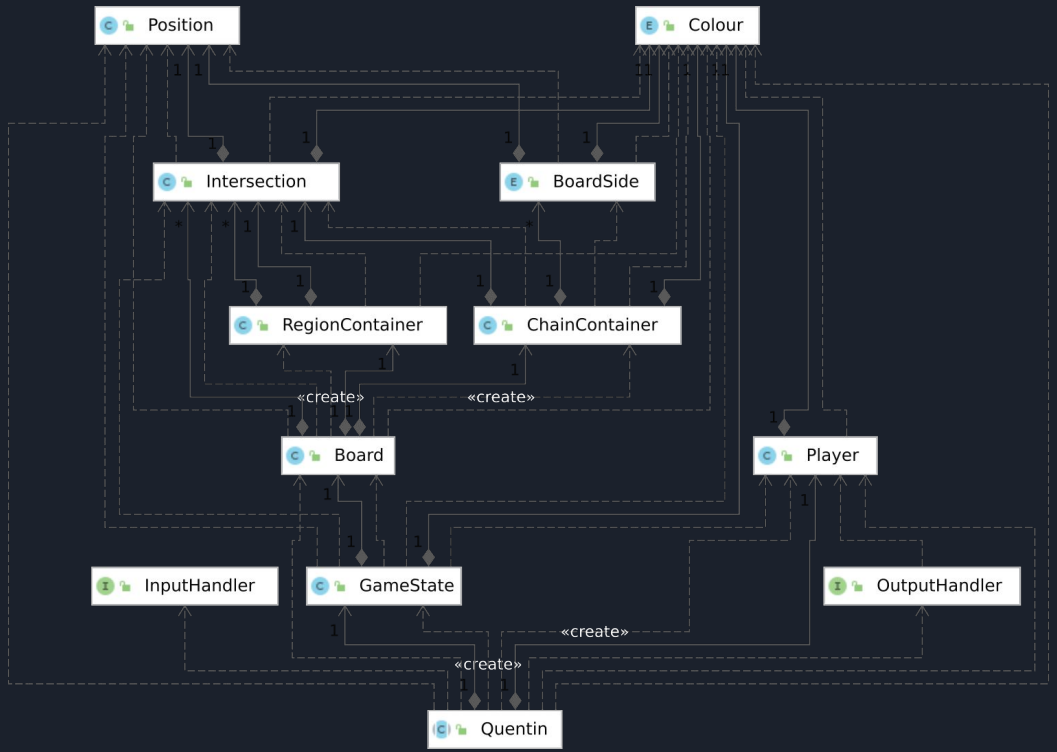
# Non-Functional Requirements



# First Design



# Core package Final Design



# Core Classes

**1** Abstract class

Quentin

- Game class ensures to follow the rules
- Ready for extension
- Milestone methods:
  - play
  - makeMove

**2** Enum

BoardSide
Stone

- Representation of the sides of the board
- Milestone methods:
  - initialiseSide
  - isAdjacentTo

**7** Classes

Board
ChainContainer
Intersection
Player
Position
RegionContainer
GameState

- Logical interaction perspective
- Milestone methods:
  - addStoneAt
  - fillTerritories
  - isChainConnectedToSameColorSides

# Snippet of Code: Fill Territories

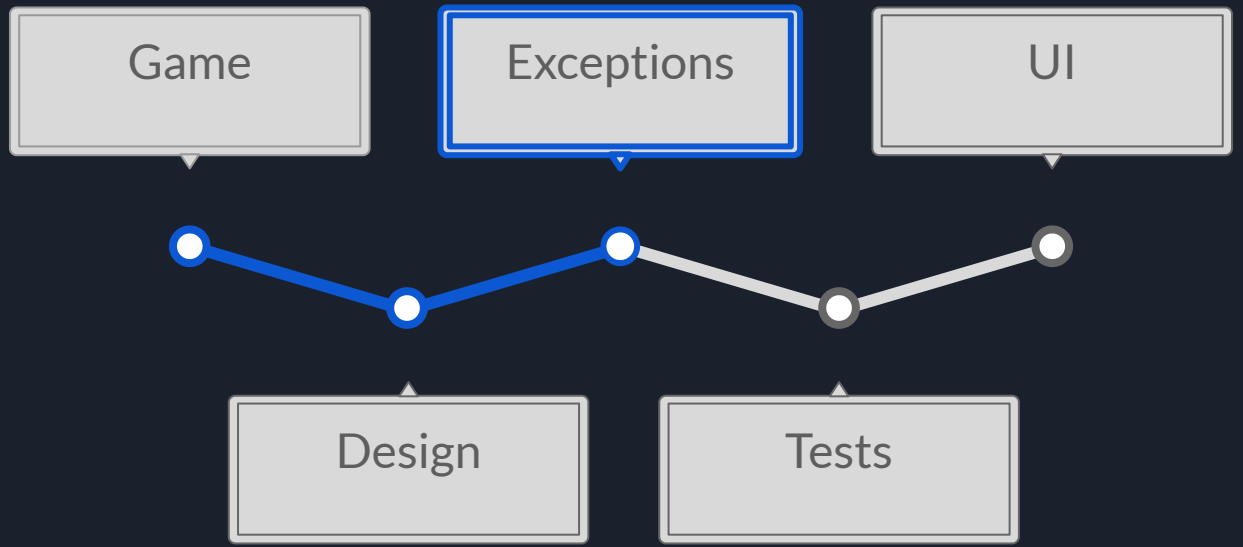
```
protected Set<Position> fillTerritories(Colour lastPlay) {
    Map<Set<Intersection>, Optional<Colour>> territoriesToFill =
            regionsContainer.getTerritoriesAndStonesToFill(lastPlay);
    territoriesToFill
            .forEach((territory, stone) ->
                    territory.stream()
                            .map(Intersection::getPosition)
                            .forEach(emptyIntersectionPosition -> addStoneAt(stone.orElseThrow(), emptyIntersectionPosition))
            );
    return territoriesToFill.entrySet().stream()  Stream<Map<K, V>.Entry<Set<Intersection>, Optional<Colour>>>
            .flatMap(entry -> entry.getKey().stream())  Stream<Intersection>
            .map(Intersection::getPosition)  Stream<Position>
            .collect(Collectors.toSet());
}
protected Map<Set<Intersection>, Optional<Colour>> getTerritoriesAndStonesToFill(Colour lastPlay) {
    return getTerritories().stream()
            .map(territory -> {
                Optional<Colour> color = getColorToFillTerritory(territory, lastPlay);
                return Map.entry(territory, color);
            })
            .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
}
```

# Snippet of Code: Illegal Move Rule

```
private boolean isIllegalMove(Intersection intersection) {
    Set<Intersection> colorAlikeOrthogonalIntersections =
            board.getOrthogonallyAdjacentIntersectionsOfColour(intersection);
    return board.getDiagonallyAdjacentIntersectionsOfColour(intersection).stream()
            .anyMatch(diagonalIntersection ->
                    board.getOrthogonallyAdjacentIntersectionsOfColour(diagonalIntersection).stream()
                            .noneMatch(colorAlikeOrthogonalIntersections::contains)
            );
}
protected Set<Intersection> getDiagonallyAdjacentIntersectionsOfColour(Intersection intersection) {
    return intersections.stream()
            .filter(intersection::isDiagonalTo)
            .filter(diagonalIntersection -> diagonalIntersection.hasStone(intersection.getColour().orElseThrow()))
            .collect(Collectors.toUnmodifiableSet());
}
protected Set<Intersection> getOrthogonallyAdjacentIntersectionsOfColour(Intersection intersection) {
    return intersections.stream()
            .filter(intersection::isOrthogonalTo)
            .filter(orthogonalIntersection -> orthogonalIntersection.hasStone(intersection.getColour().orElseThrow()))
            .collect(Collectors.toUnmodifiableSet());
}
```

# Snippet of Code: Pie Rule

```
protected void applyPieRule() { Stream.of(playerOne, playerTwo).forEach(Player::changeSide); }
protected boolean applyPieRuleIfPlayerWants(Player currentPlayer) {
    if (inputHandler.askPie(currentPlayer.getName())) {
        applyPieRule();
        outputHandler.notifyPieRule(getPlayers());
        return true;
    } else {
        return false;
    }
}
protected void changeSide() { colour = colour.getOppositeColor(); }
public Colour getOppositeColor() {
    return switch (this) {
        case BLACK -> WHITE;
        case WHITE -> BLACK;
    };
}
```


Game
Exceptions
UI
Design
Tests

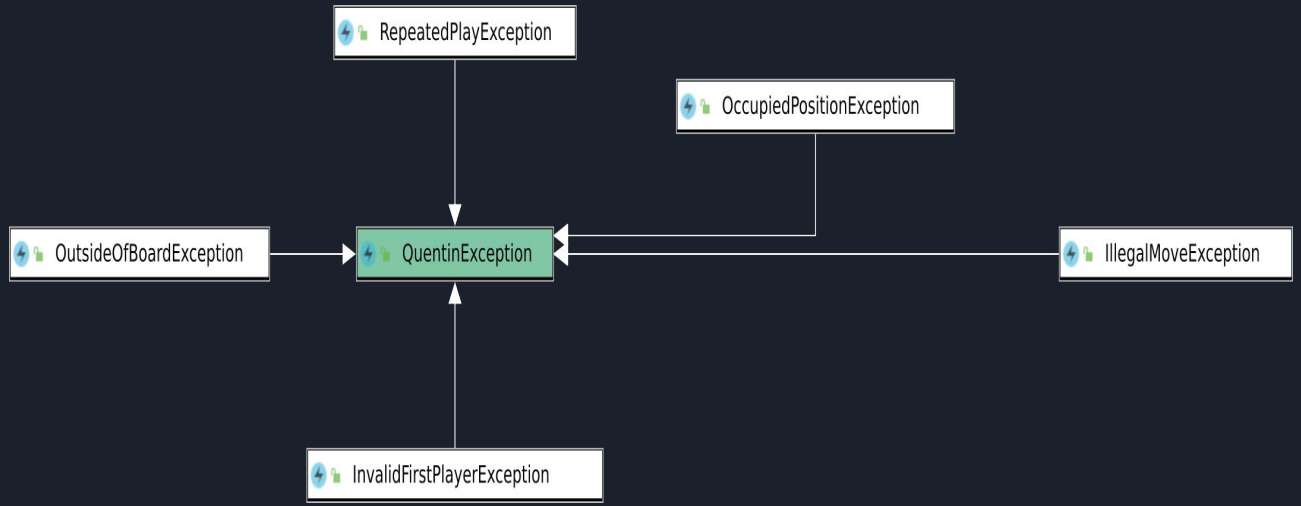
# Exceptions

Unexpected inputs could create some issues, they might be classified in:

- Inputs violating game constraints
- Inputs do not admitted for invalid format

Players are warned about the exception occurred through a message

# Exceptions Diagram



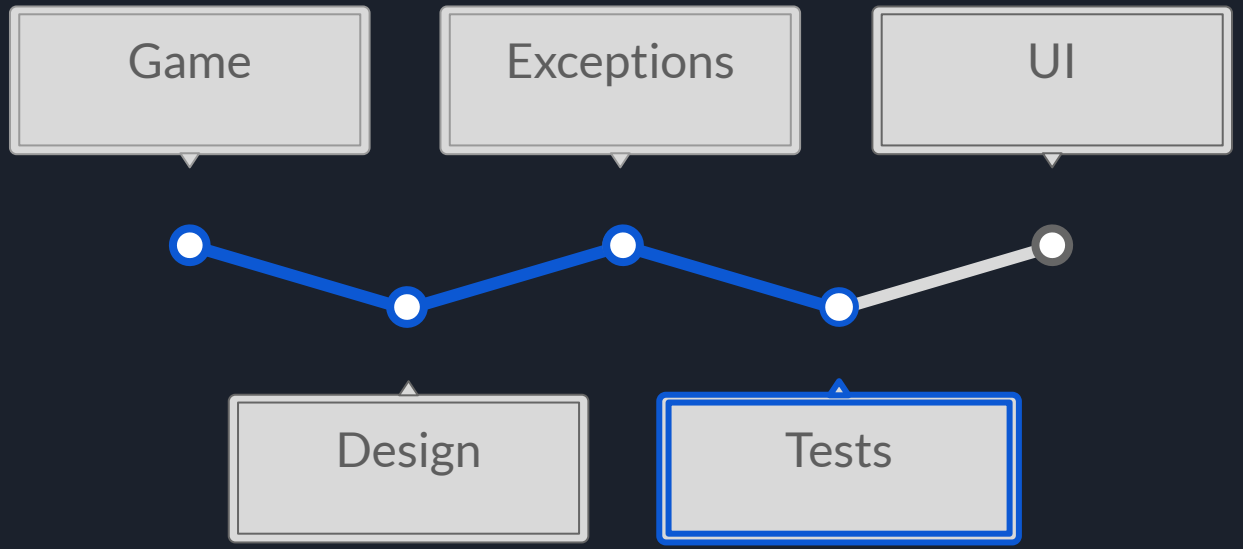
# Exceptions

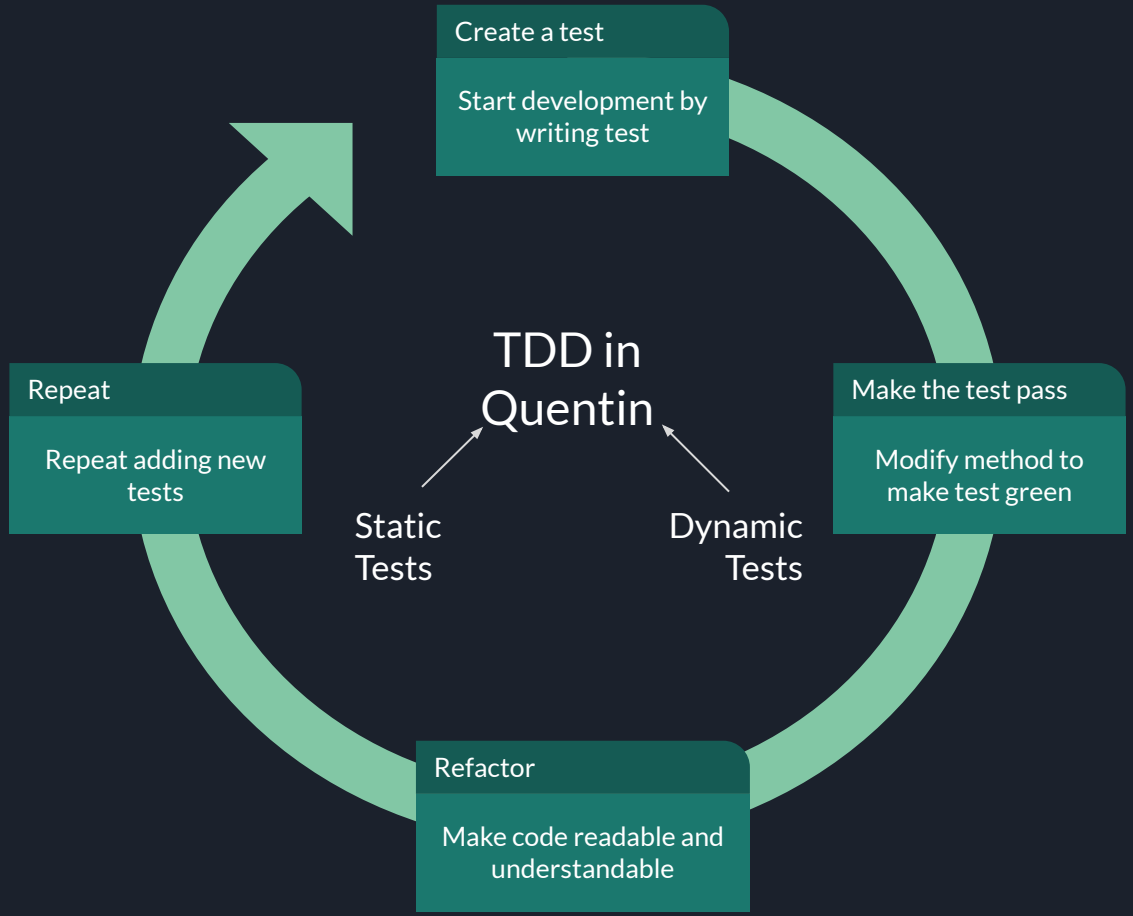
**QuentinException**

| Game Constraints |
| --- |
| IllegalMove |
| InvalidFirstPlayer |
| OccupiedPosition |
| OutsideBoard |
| RepeatedPlay |

| Invalid Input Format |
| --- |
| InputMismatch |
| NoSuchElementException |

Quentin exception class
extends RunTimeException class

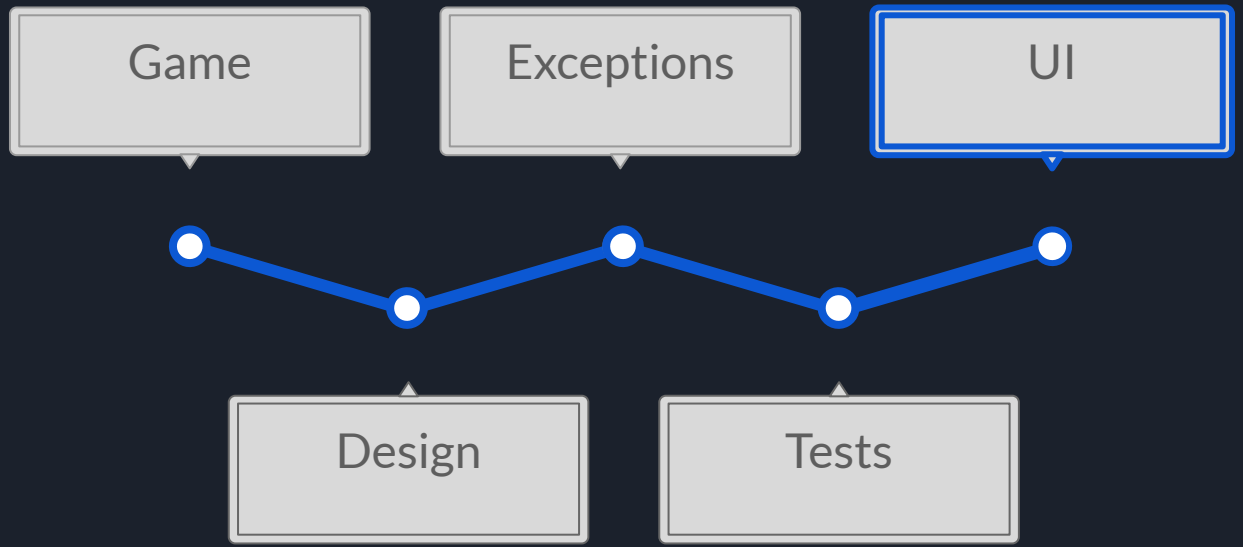

Game
Exceptions
UI
Design
Tests


Create a test
Start development by writing test
Make the test pass
Modify method to make test green
Refactor
Make code readable and understandable
Repeat
Repeat adding new tests
TDD in Quentin
Static Tests
Dynamic Tests

# Test Types



| Static Test | Dynamic Test |
| --- | --- |
| Static tests which are fully specified at the compile time | Generated during runtime |
| Parameterized tests with method sources | Usage of streams, collections |
| | Test methods can NOT be static or private |


Game
Exceptions
UI
Design
Tests

UI: User Interface

- CLI
- GUI

# User Interface

# Interfaces in UI

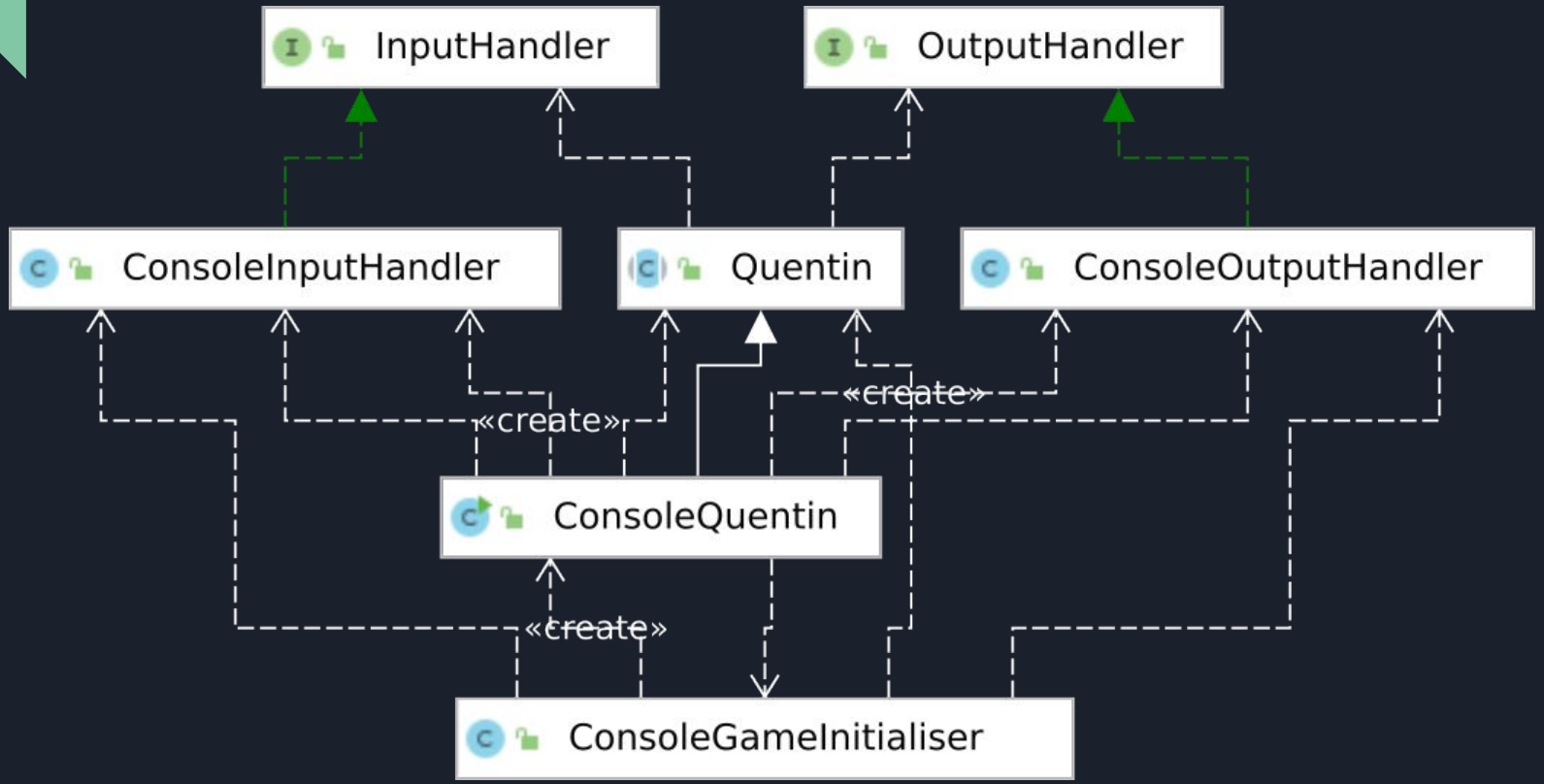
User Interface has two components that are Input and Output Handlers to make the interaction between the software and user.

- OutputHandler : pop up messages to the players like questions or notifications
- InputHandler : Player decision

# UI

Command Line Interface

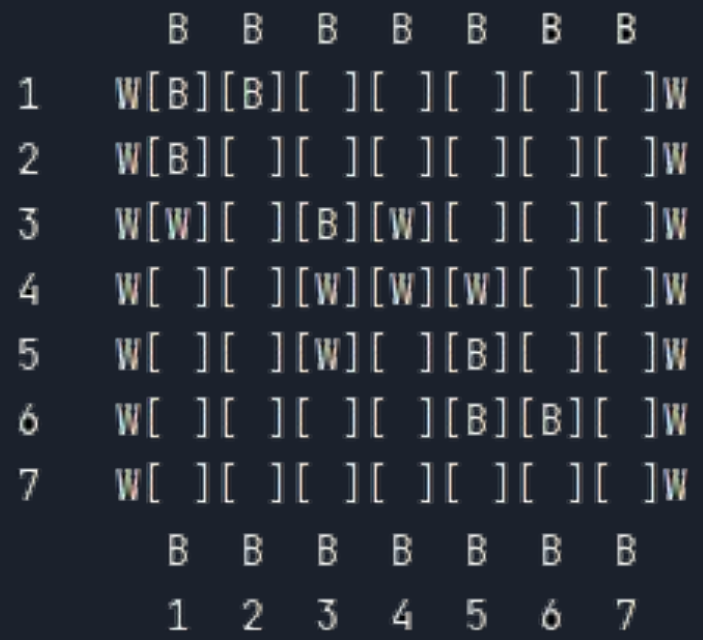
# CLI Package Diagram



# CLI Classes

ConsoleInputHandler and ConsoleOutputHandler are implementation of interfaces *InputHandler* and *OutputHandler*

ConsoleGameInitialiser get the needed data for creating the game instance

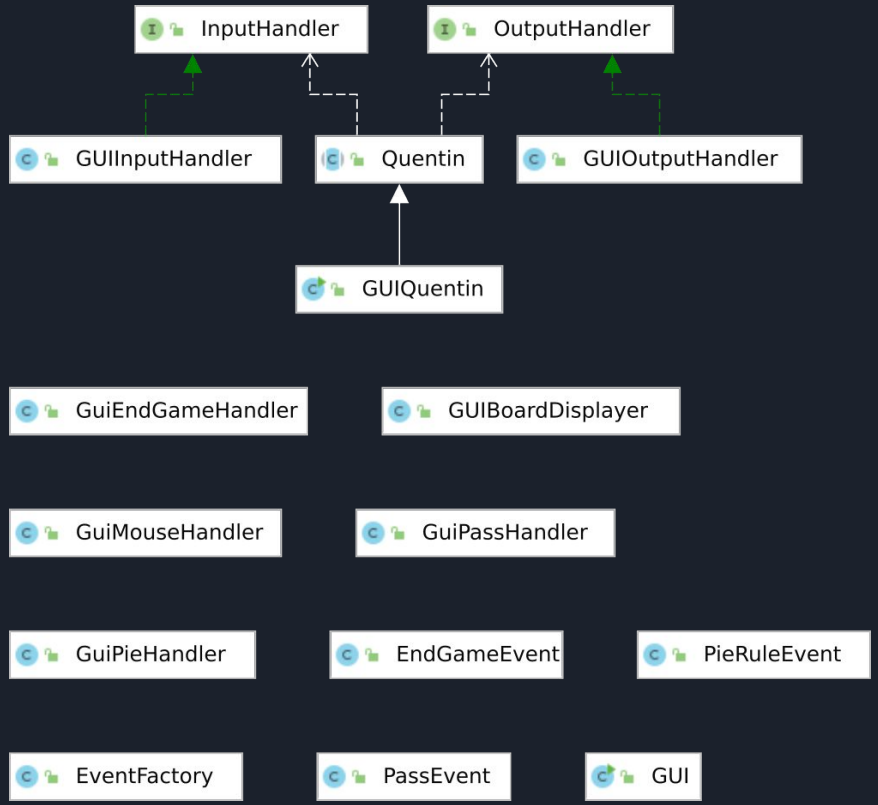


# UI

Graphical User Interface

- Events
- Handlers
- Display Game

# GUI Package Diagram

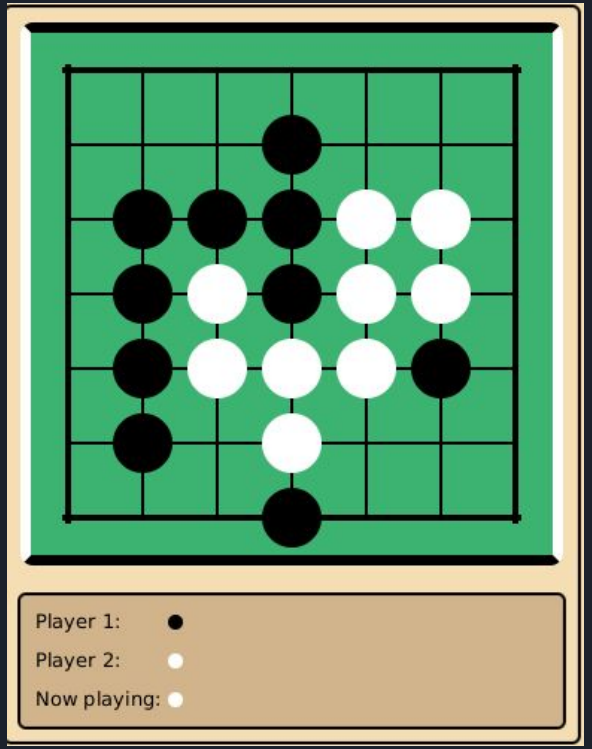


# Graphical User Interface

GUI allows users to interact with the game

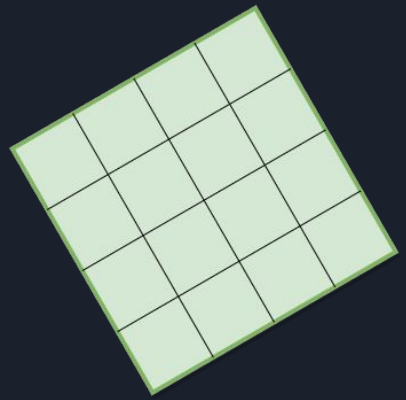
The actions in a GUI are usually performed through direct manipulation of the graphical elements:

- Button
- Grid
- Geometrical Object

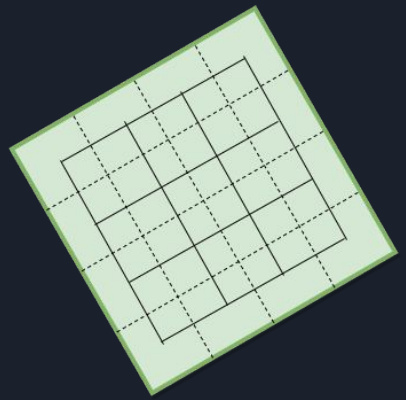


# Display Board

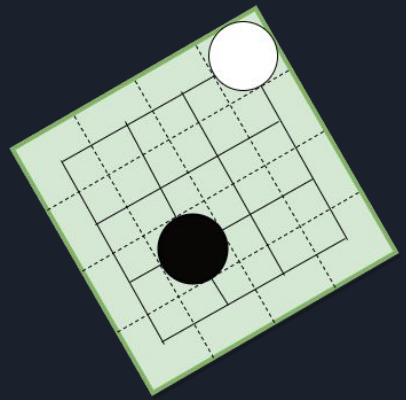
GridPane Obj



GridPane Obj + Line Obj



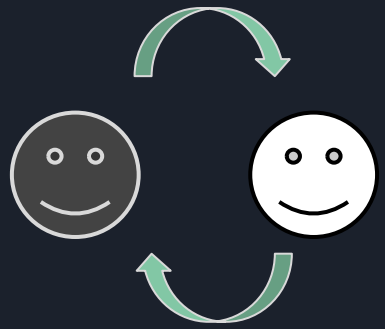
GridPane Obj + Line Obj + Circle Obj



Other functionality:

- Create and manage players label

# Events

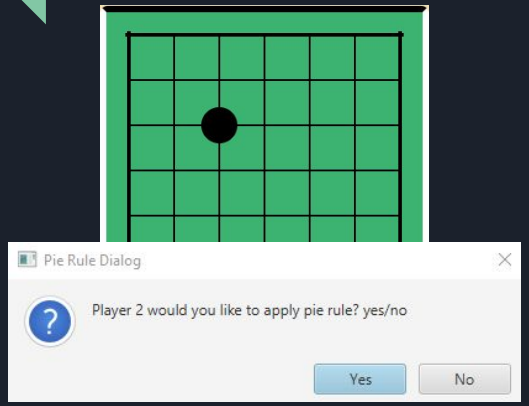


Pie Event

Pass Event

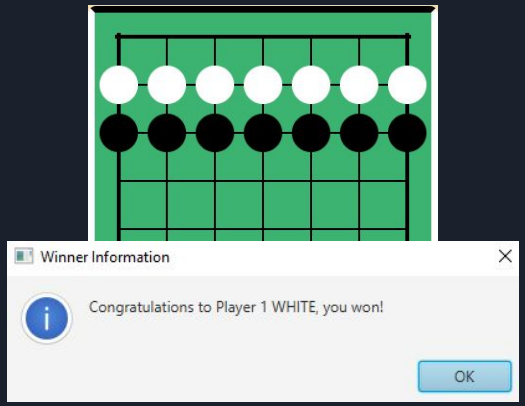

EndGame Event

# Handlers 1/2




Pie
Handler

Pass
Handler




EndGame
Handler

# Handler 2/2

## Mouse Handler

The most important handler because it constitutes the link between the player action and the game reaction.

- Define a new Position in the Board
- Check if the Player is able to play
- Trigger other handlers

**USER**

**GAME**

# THANKS FOR YOUR ATTENTION