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, which allows the two players to play remotely.

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, which allows the two players to play remotely.

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

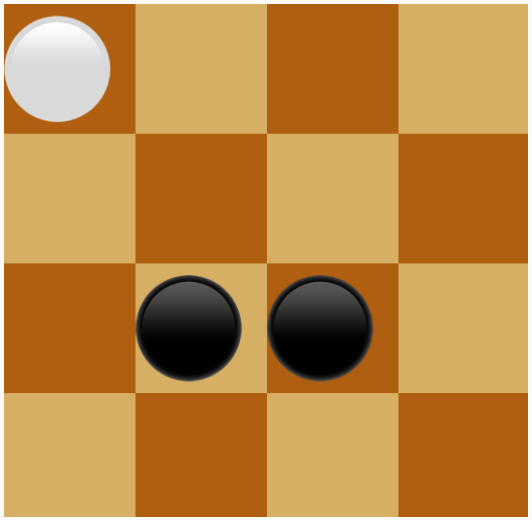
Konobi is a drawless game and it can be played either on a go board or a chess board.
Two players, black and white, take turns at placing stones of their color on the board, starting with black. The aim of the players is to build chains of connected stones of their color.

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

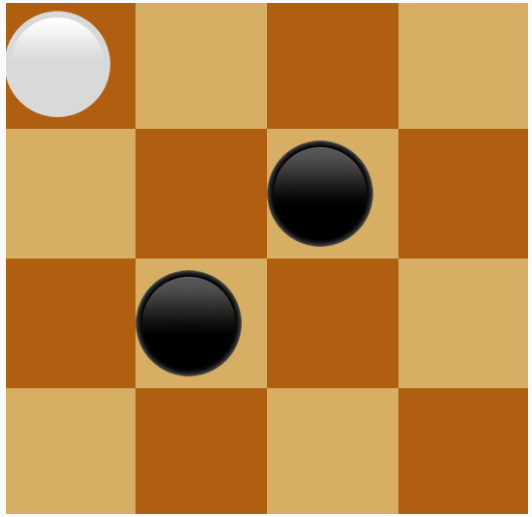
- Black: top $\leftrightarrow$ bottom
- White: left $\leftrightarrow$ right

Connections

Two like-colored stones can be:



Strongly connected



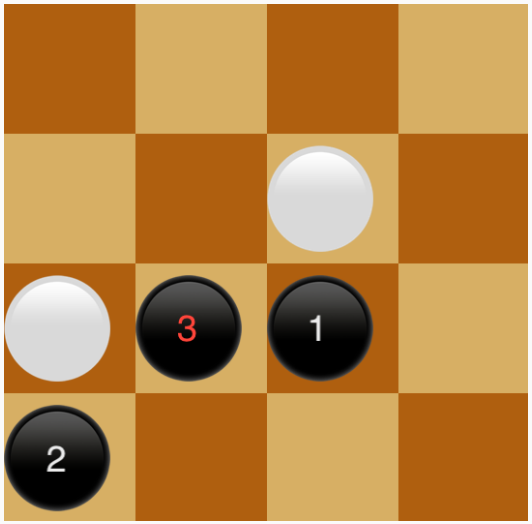
Weakly connected

A chain is a set of connected stones

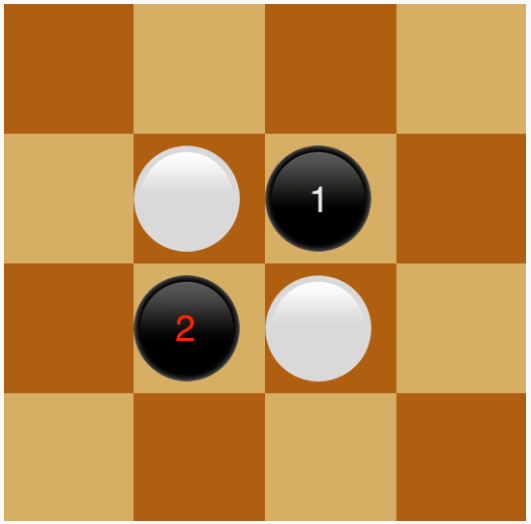
Placement rules

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

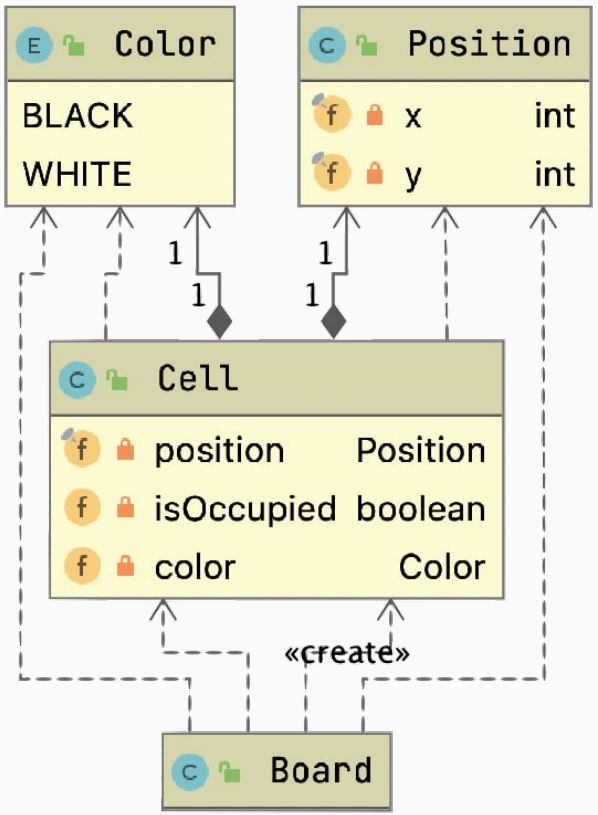
- **Pie rule**: at his first move, white can decide to switch colors with black instead of making a move.
- **Mandatory pass**: if a player cannot make a move (because of placement restrictions), he has to pass. It is guaranteed that at least one player can make a move.

Let's see a live demo

Basic entities

The basic building block: `Cell`

- When a cell is constructed it is empty: no color is associated to it and `isOccupied` is `False`
- When a stone is placed in the cell, a color is set and `isOccupied` becomes `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>`

Reasons for this choice of data structure:

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

Connections

Strong connections

in class `Position`:

```
public int squareEuclideanDistanceFrom(Position other) {
    return (int)Math.pow(x-other.x,2)+(int)Math.pow(y-other.y,2);
}
```

in class `Cell`:

```
private boolean isOrthogonallyAdjacentTo(Cell otherCell) {
    return this.position.squareEuclideanDistanceFrom(otherCell.position)==1;
}
public Set<Cell> orthogonalNeighborsIn(Set<Cell> cells) {
    return cells.stream()
                .filter(this::isOrthogonallyAdjacentTo)
                .collect(Collectors.toSet());
}
```

in class `Board`:

```
public Set<Cell> strongConnectionsOf(Cell cell) {
    return cell.orthogonalNeighborsIn(this).stream()
                                           .filter(Cell::isOccupied)
                                           .filter(c->c.hasSameColorAs(cell))
                                           .collect(Collectors.toSet());
}
```

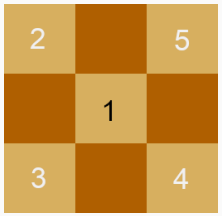
Weak connections

in class `Board`:

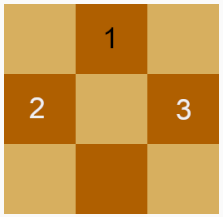
```
private Set<Cell> commonStrongConnectionsBetween(Cell cell, Cell otherCell) {
    Set<Cell> strongNeighborsOfCell = strongConnectionsOf(cell);
    Set<Cell> strongNeighborsOfOtherCell = strongConnectionsOf(otherCell);
    strongNeighborsOfCell.retainAll(strongNeighborsOfOtherCell);
    return strongNeighborsOfCell;
}
public Set<Cell> weakConnectionsOf(Cell cell) {
    return cell.diagonalNeighborsIn(this).stream()
                                         .filter(Cell::isOccupied)
                                         .filter(c->c.hasSameColorAs(cell))
                                         .filter(c->commonStrongConnectionsBetween(cell, c).isEmpty())
                                         .collect(Collectors.toSet());
}
```

Tests

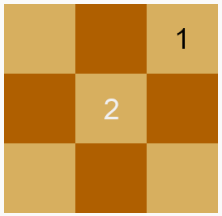
Different test cases for diagonal adjacency (same for orthogonal):



inner cell

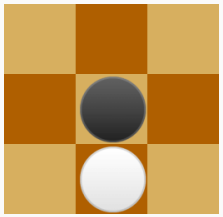


cell on edge

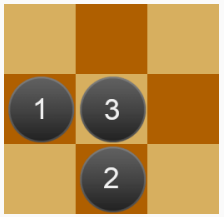


cell on corner

Example of tests for strong and weak connection:



the two stones are
not strongly connected



the two stones are
not weakly connected

Rules

Placement rules: how to implement them?

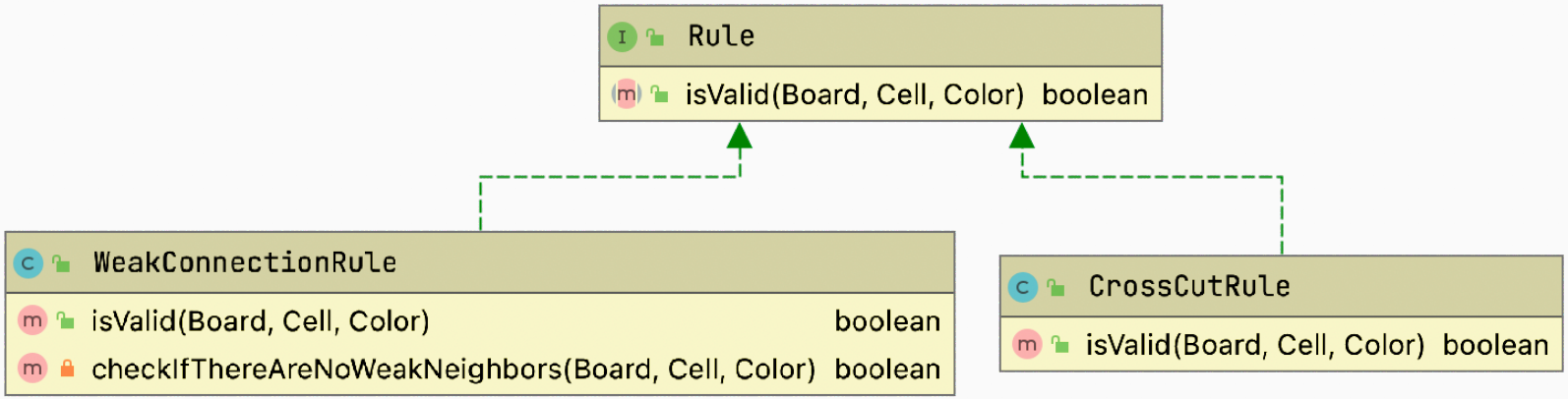
A feature that Konobi has to provide is that a player must make a move which respects placement rules:

- 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 (**Weak Connection Rule**)
- It's illegal to make a 2x2 pattern of stones consisting of two diagonally adjacent Black stones and two diagonally adjacent White stones. (**Crosscut Rule**)

Initially, a `Rules` class was implemented...

Rules package

...but later we realized that there could be a way 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`.

A `Rule` interface will allow the possibility to add new rules on the game.

How the rules were implemented

- WeakConnectionRule

```
public class WeakConnectionRule implements Rule{
    @Override
    public boolean isValid(Board board, Cell cell, Color stoneColor) {
        board.placeStone(cell.getPosition(), stoneColor);
        Set<Cell> weakNeighbors = board.weakConnectionsOf(cell);
        cell.removeStone();
        boolean isWeakPlacementInvalid = weakNeighbors.stream()
                                                .map(c->c.orthogonalNeighborsIn(board))
                                                .anyMatch(s->s.stream()
                                                            .filter(c->!c.isOccupied())
                                                            .anyMatch(c->checkIfThereAreNoWeakNeighbors(board, c, stoneColor)));
        return !isWeakPlacementInvalid;
    }

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

- CrossCutRule

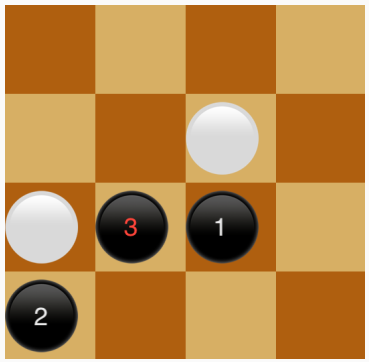
```
public class CrossCutRule implements Rule{
    @Override
    public boolean isValid(Board board, Cell cell, Color stoneColor) {
        board.placeStone(cell.getPosition(), stoneColor);
        Set<Cell> weakNeighbors = board.weakConnectionsOf(cell);
        boolean isThereACrossCut = weakNeighbors.stream()
                                                .map(c->c.commonOrthogonalNeighborsWith(cell, board))
                                                .anyMatch(s->s.stream()
                                                            .allMatch(c->c.isOccupied() && !c.hasSameColorAs(cell)));
        cell.removeStone();
        return !isThereACrossCut;
    }

}
```

How the rules were tested

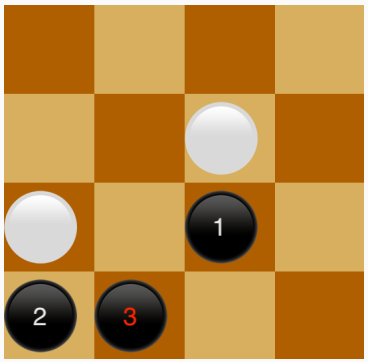
We implemented `WeakConnectionRule` and `CrossCutRule` by following TDD principles based on some examples...

Legal moves

```
@Test
public void blackStoneMakesLegalWeakConnectionPlacement() {
    Board board = new Board(5);
    WeakConnectionRule rule = new WeakConnectionRule();
    board.placeStone(at(1,1), Color.BLACK);
    board.placeStone(at(1,2), Color.WHITE);
    board.placeStone(at(3,2), Color.BLACK);
    board.placeStone(at(3,3), Color.WHITE);
    Cell cellToVerify = board.getCell(at(2,2));
    assertTrue(rule.isValid(board, cellToVerify, Color.BLACK));
}
```



Illegal moves

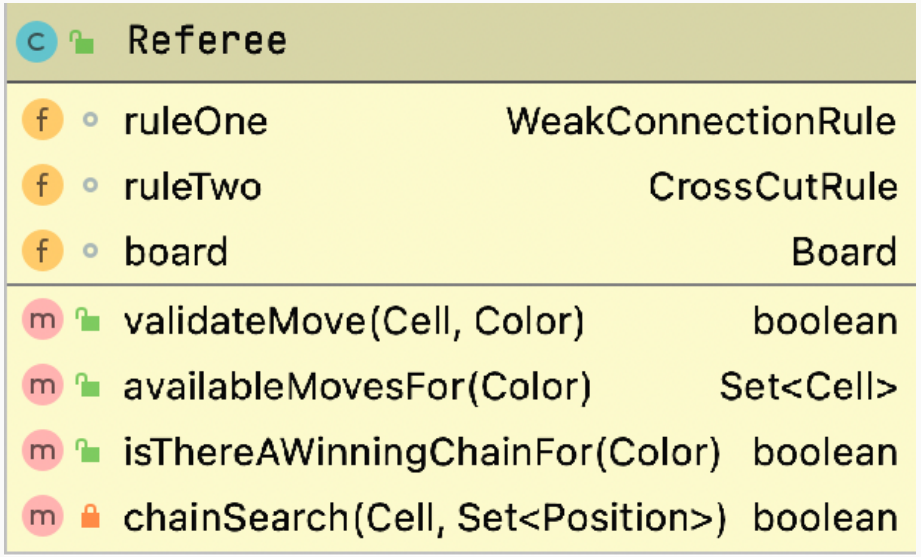



```
@Test
public void blackStoneMakesIllegalWeakConnectionPlacement() {
    Board board = new Board(5);
    WeakConnectionRule rule = new WeakConnectionRule();
    board.placeStone(at(1,1), Color.BLACK);
    board.placeStone(at(1,2), Color.WHITE);
    board.placeStone(at(3,2), Color.BLACK);
    board.placeStone(at(3,3), Color.WHITE);
    Cell cellToVerify = board.getCell(at(2,1));
    assertFalse(rule.isValid(board, cellToVerify, Color.BLACK));
}
```

A new entity comes: `Referee`

Given the logic of a valid move in `Rules` package, our aim was to group together all the methods needed to check if:

- a given move is legal w.r.t. `WeakConnectionRule` and `CrossCutRule`
- a winning chain is present
- the current player has to pass



How to find a winning chain

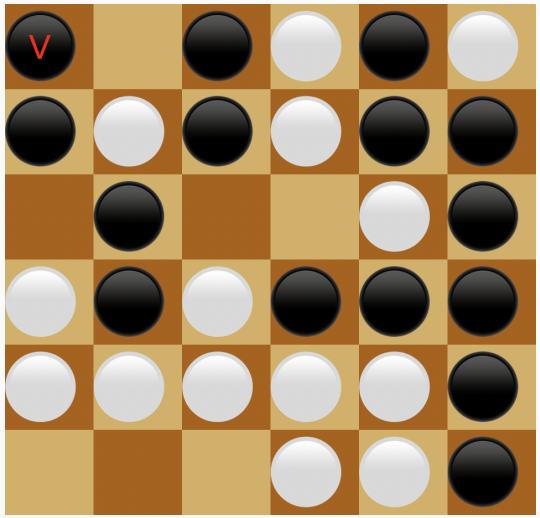
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.

```
public boolean isThereAWinningChainFor(Color color) {
    Set<Position> visitedCells = new HashSet<>();
    return board.stream()
                .filter(c->c.hasColor(color))
                .filter(c->c.isOnStartEdge(board.dimension()))
                .filter(Cell::isOccupied)
                .filter(c->!visitedCells.contains(c.getPosition()))
                .filter(c->c.hasColor(color))
                .anyMatch(c->chainSearch(c, visitedCells));
}

private boolean chainSearch(Cell source, Set<Position> visitedCells) {
    visitedCells.add(source.getPosition());
    if(source.isOnEndEdge(board.dimension())) return true;
    return board.connectionsOf(source).stream()
                                      .filter(c->!visitedCells.contains(c.getPosition()))
                                      .anyMatch(c->chainSearch(c, visitedCells));
}
```

Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

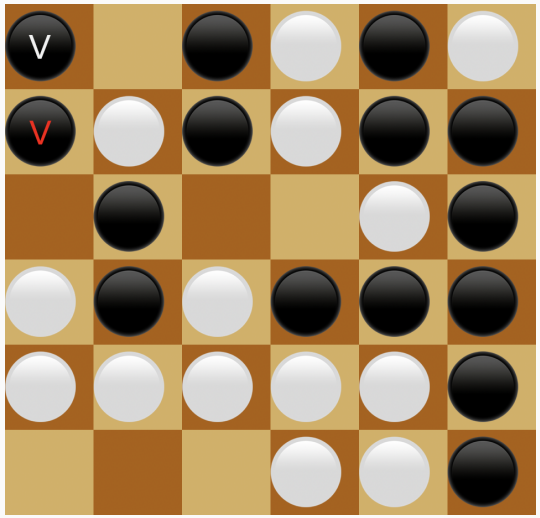
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

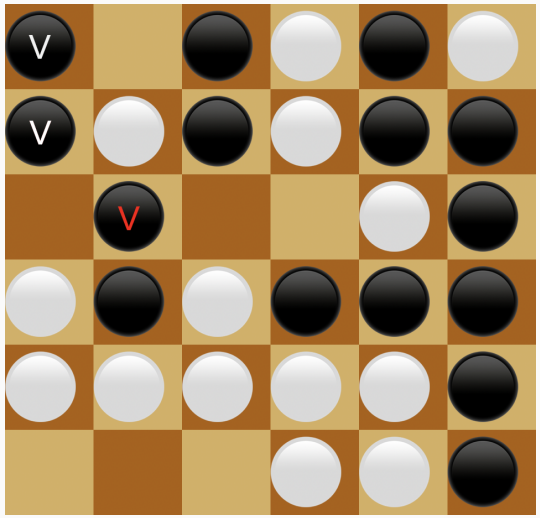
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

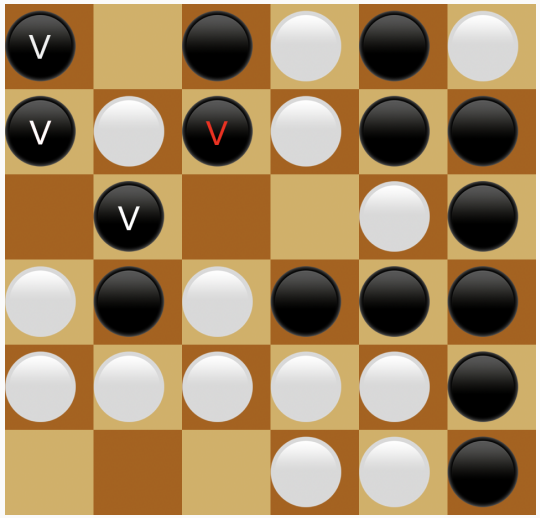
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

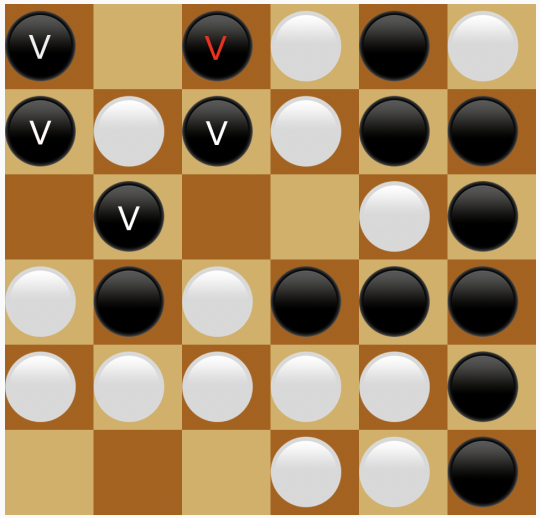
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

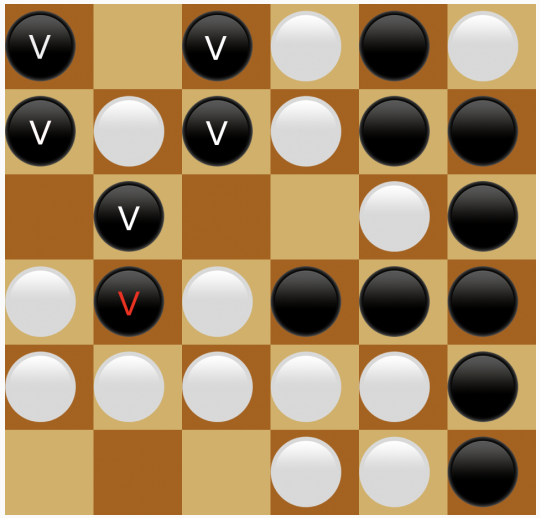
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

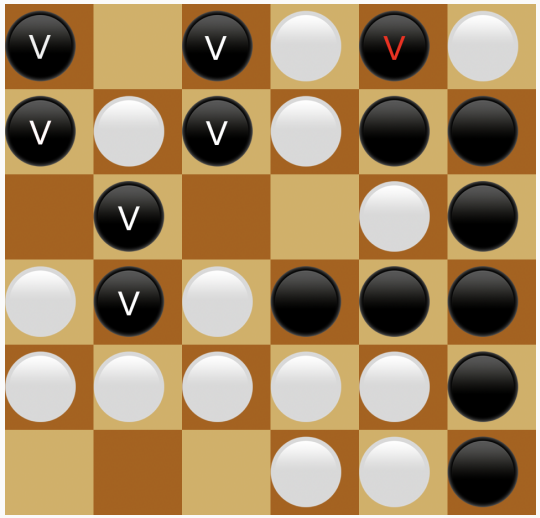
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

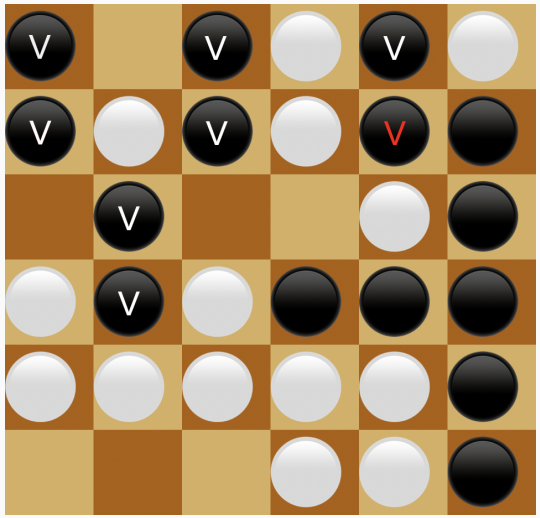
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

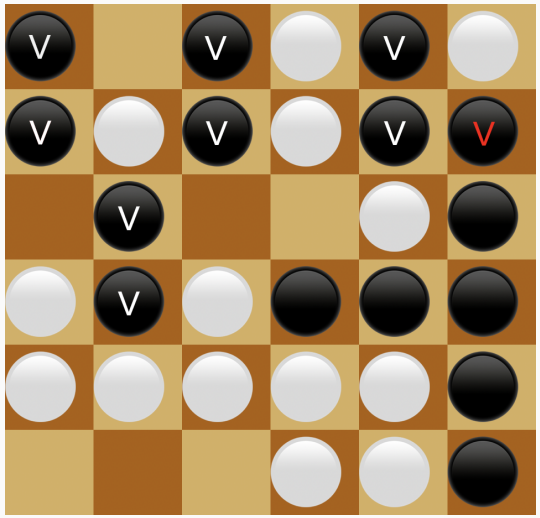
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

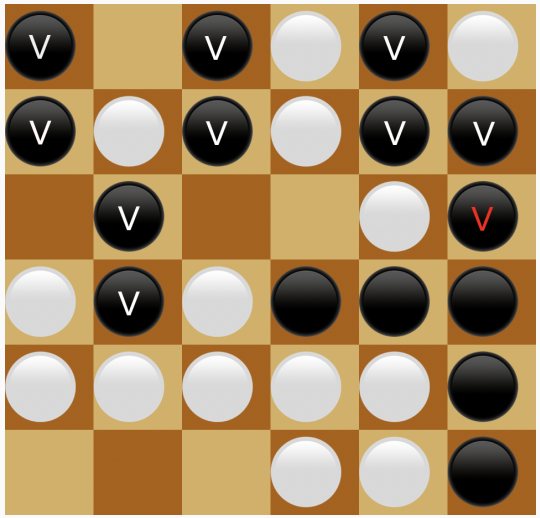
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

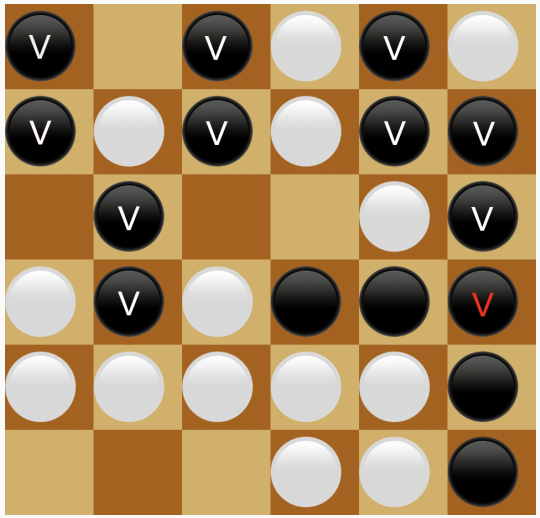
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

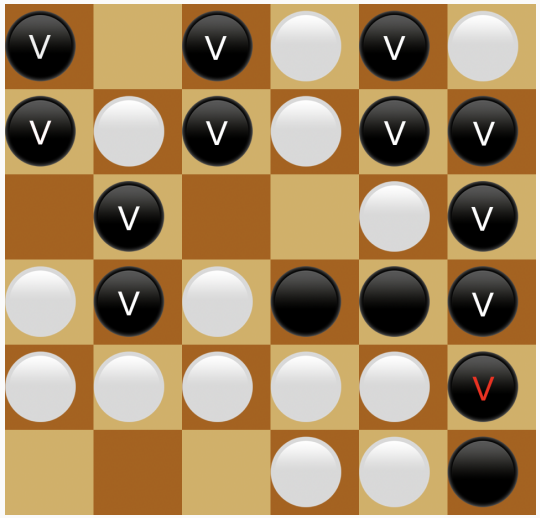
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

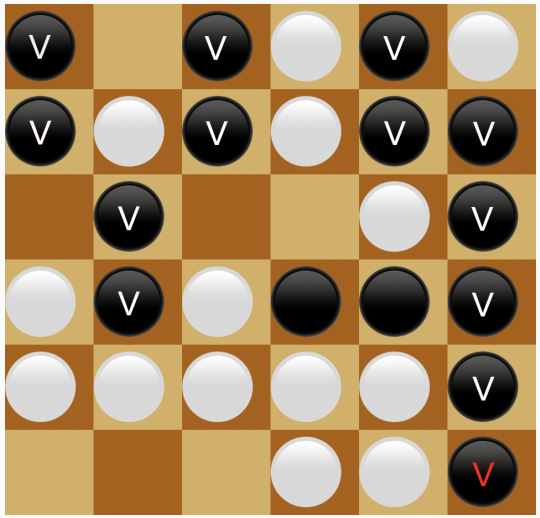
`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

How to find a winning chain

`isThereaWinningChainFor` method is used to find a chain of a given `Color`.



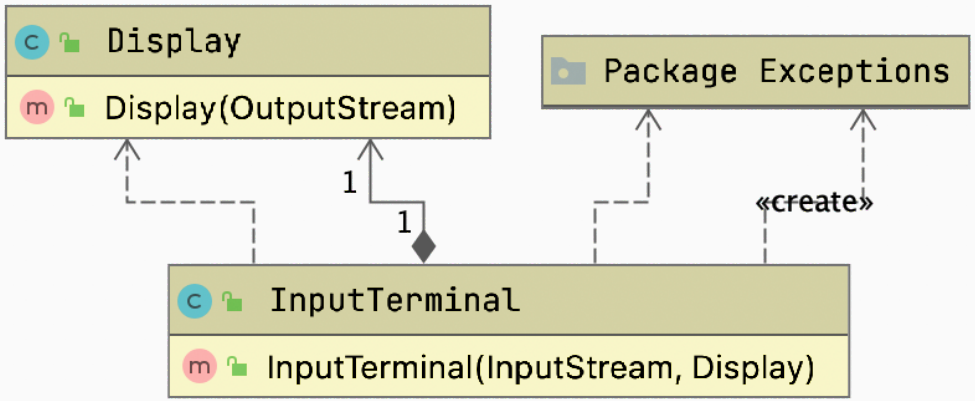
Recursive approach: starting from all stones placed on the start edge, we explore the set of their connected stones inside the board until we arrive in the end edge.

InputOutput

Handling user interaction: `InputTerminal` and `Display`

Classes `InputTerminal` and `Display` take care of game I/O.

- `Display`: communication with players through messages
- `InputTerminal`: management of player inputs



Sorting out exceptional events

Potential issues with user input:

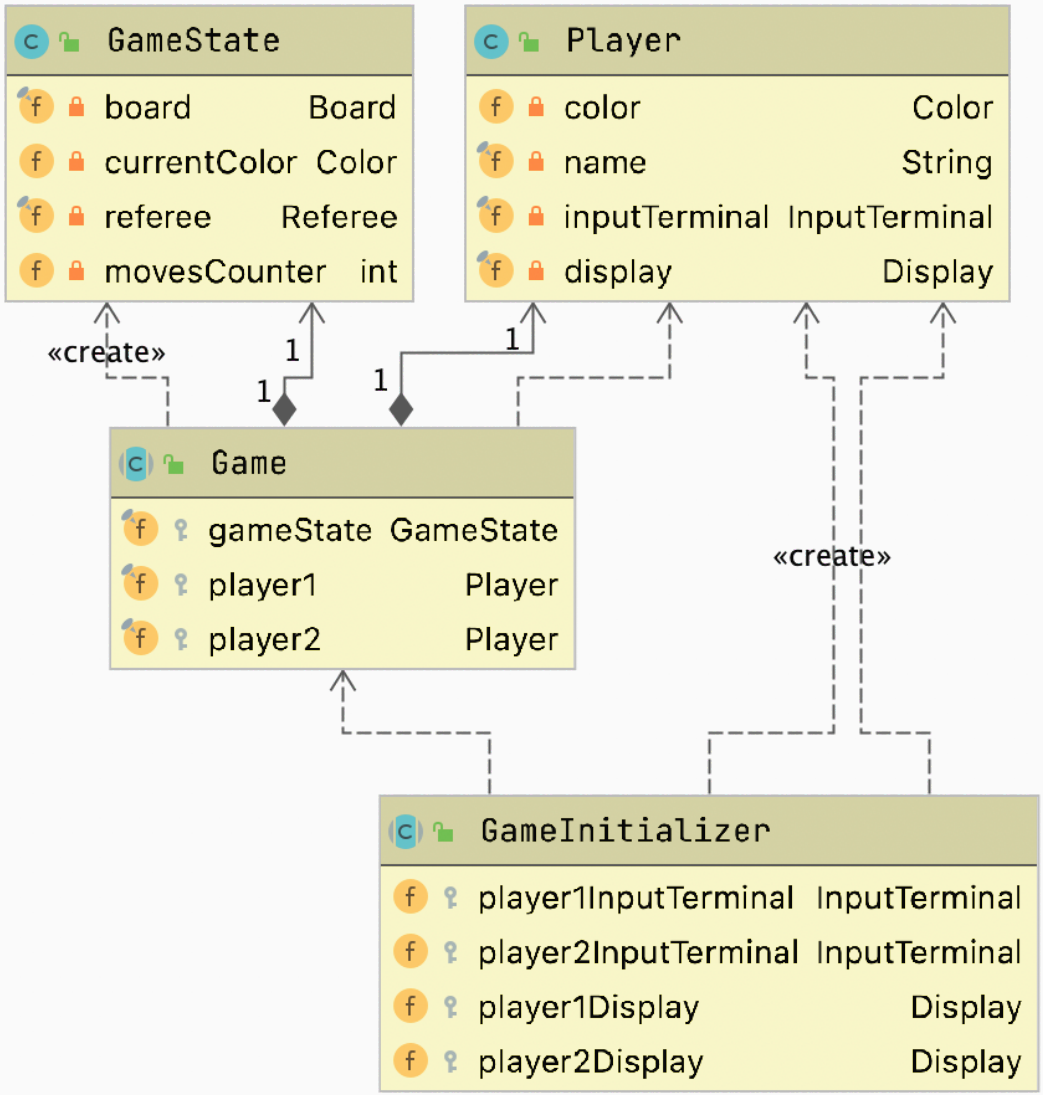
- negative number (board dimension, move coordinates)
- wrong answer (pie rule requires "y" or "n")
- number format (string instead of an integer)

How to deal with them:

- `throw` (and `catch`) custom exceptions extending `Exception` java class

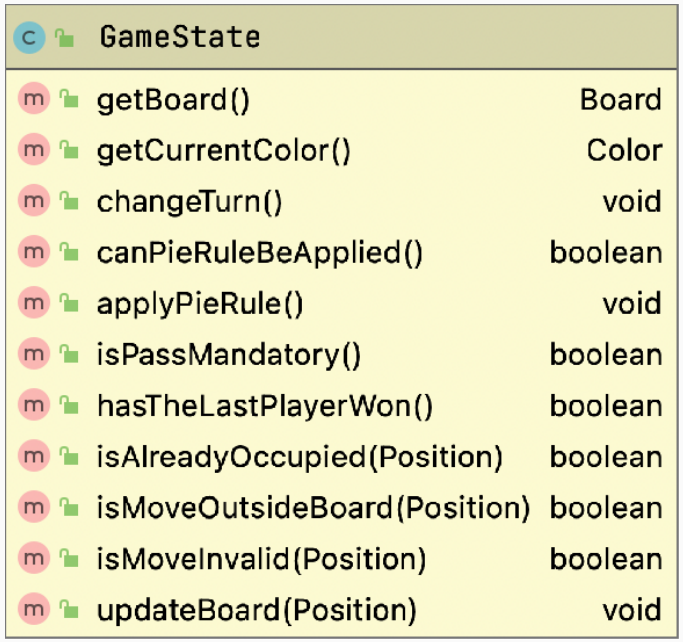
Game dynamics

Building blocks for game dynamics



Game flow management: `GameState`

`GameState` class keeps track of current state of the game.



`GameState` acts as an intermediary between `Referee` and `Board` on one side and the higher-level class `Game` on the other side.

Game execution: core methods

Based on the directives of `GameState`, `Game` manages the interactions with players.

Game execution: core methods

Based on the directives of `GameState`, `Game` manages the interactions with players.

```
public void play() {
    do {
        singleTurn();
    } while (!checkAndNotifyWin());
}
```

Game execution: core methods

Based on the directives of `GameState`, `Game` manages the interactions with players.

```
public void play() {
    do {
        singleTurn();
    } while (!checkAndNotifyWin());
}
```

```
private void singleTurn() {
    getCurrentDisplay().currentPlayerTurnMessage(getCurrentPlayer());
    if(gameState.canPieRuleBeApplied() && getCurrentInputTerminal().playerWantsToApplyPieRule(getCurrentPlayer())) {
        applyAndNotifyPieRule();
    } else {
        regularMove();
        printBoard(gameState.getBoard());
    }
    gameState.changeTurn();
}
```

Game execution: core methods

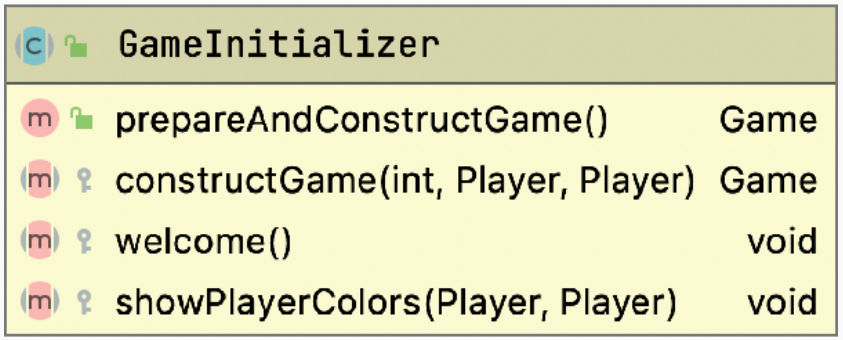
Based on the directives of `GameState`, `Game` manages the interactions with players.

```
public void play() {
    do {
        singleTurn();
    } while (!checkAndNotifyWin());
}

 private void singleTurn() {
     getCurrentDisplay().currentPlayerTurnMessage(getCurrentPlayer());
     if(gameState.canPieRuleBeApplied() && getCurrentInputTerminal().playerWantsToApplyPieRule(getCurrentPlayer())) {
         applyAndNotifyPieRule();
     } else {
         regularMove();
         printBoard(gameState.getBoard());
     }
     gameState.changeTurn();
 }

private void regularMove() {
    if(gameState.isPassMandatory()) {
        notifyMandatoryPass();
    }
    else {
        Position inputPosition = chooseNextMove();
        gameState.updateBoard(inputPosition);
    }
}
```

Preparing the game: `GameInitializer`

Abstraction for `Game` initialization:

- welcome message
- receiving input board dimension
- construct players
- construct game
- display player colors



Running the game

Two versions of Konobi

Konobi can be played by the two players on the same terminal or in a client-server version.
In the latter a server hosts the entire logic of the game: it creates a `ServerSocket` bound to a specified port number and waits for two clients to connect.

The two clients connect through `telnet` to the server, hence Java is not needed on the client side.

Inheritance in `Game` and `GameInitializer`

Code duplication between the two versions is avoided by exploiting polymorphism: abstract classes `Game` and `GameInitializer` allow for very general code in the form:

```
Game game = gameInitializer.prepareAndConstructGame();
game.play();
```

Depending on the version of Konobi, `game` is either a `GameConsole` or a `GameClientServer` and the method `play` calls different versions of abstract functions for the two versions.

Behind the scenes of `prepareAndConstructGame`

```java
public Game prepareAndConstructGame() {
    welcome();
    int dimension = player1InputTerminal.getDimension();
    String player1Name = player1InputTerminal.inputPlayerName(1);
    String player2Name = player2InputTerminal.inputPlayerName(2);
    Player player1 = new Player(Color.BLACK, player1Name, player1InputTerminal, player1Display);
    Player player2 = new Player(Color.WHITE, player2Name, player2InputTerminal, player2Display);
    Game game = constructGame(dimension, player1, player2);
    showPlayerColors(player1,player2);
    return game;
}
```

```java
@Override
protected GameConsole constructGame(int dimension, Player player1, Player player2){
    return new GameConsole(dimension, player1, player2);
}
```

`GameInitializerConsole`

```java
@Override
protected GameClientServer constructGame(int dimension, Player player1, Player player2){
    return new GameClientServer(dimension, player1, player2);
}
```

`GameInitializerClientServer`

Comparing `GameConsole` and `GameClientServer`

Client-Server version has to differentiate the messages for the two players:

```java
public class GameConsole extends Game {

    public GameConsole(int dimension, Player player1, Player player2) {
        super(dimension, player1, player2);
    }

    @Override
    protected void notifyPieRule() {
        getCurrentDisplay().playerColorsMessage(player1, player2);
    }

    @Override
    protected void notifyMandatoryPass(){
        getCurrentDisplay().passMessage(getCurrentPlayer());
    }

    @Override
    protected void printBoard(Board board){
        getCurrentDisplay().printBoard(board);
    }

    @Override
    protected void notifyEndOfMatch() {
        getOtherDisplay().winMessage(getOtherPlayer());
    }

}
```

`GameConsole`

```java
public class GameClientServer extends Game {

    public GameClientServer(int dimension, Player player1, Player player2) {
        super(dimension, player1, player2);
    }

    @Override
    public void notifyPieRule() {
        getCurrentDisplay().playerColorsMessage(player1, player2);
        getOtherDisplay().pieRuleHasBeenAppliedMessage();
        getOtherDisplay().playerColorsMessage(player1, player2);
    }

    @Override
    protected void notifyMandatoryPass(){
        getCurrentDisplay().passMessage(getCurrentPlayer());
        getOtherDisplay().passMessage(getCurrentPlayer());
    }

    @Override
    public void printBoard(Board board){
        getCurrentDisplay().printBoard(board);
        getOtherDisplay().otherPlayerHasMadeMoveMessage();
        getOtherDisplay().printBoard(board);
    }

    @Override
    public void notifyEndOfMatch() {
        getOtherDisplay().winMessage(getOtherPlayer());
        getCurrentDisplay().lossMessage(getCurrentPlayer());
    }

}
```

`GameClientServer`

Testing Konobi

To test the full execution of the game we designed a way to simulate it:

- Text files containing the list of inputs for some matches
- Standard input redirection on these files
- Standard output redirection to a `ByteArrayOutputStream`, then converted to a `String`
- `Main` execution
- Comparison between expected and actual winner name

Now let's connect to a server

Thank you for your attention!