Order and Chaos

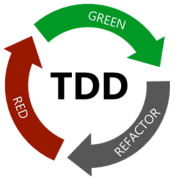
Project for Software Development Methods' course

Buscaroli Elena Liberatori Benedetta Mami Ciro Antonio Santacatterina Giovanni Valeriani Lucrezia

Practices

- Version Control

- Test Driven Development



- Continuous Integration

- Build Automation

Game & Rules

Game and Rules

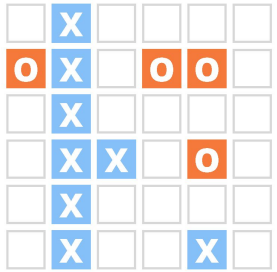
Order and Chaos is a variant of the game tic-tac-toe on a 6×6 game board.

Both players can place either Xs or Os, Order plays first, then turns alternate.

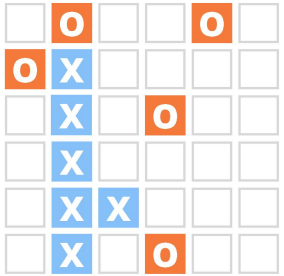
Order aims to get five like pieces in a row either vertically, horizontally, or diagonally.

Chaos aims to prevent this.

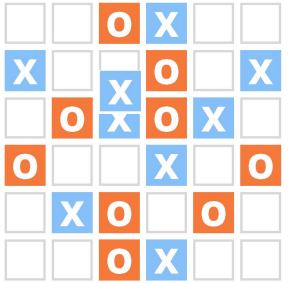
Six in a row does not qualify as a win for Order.



six in a row



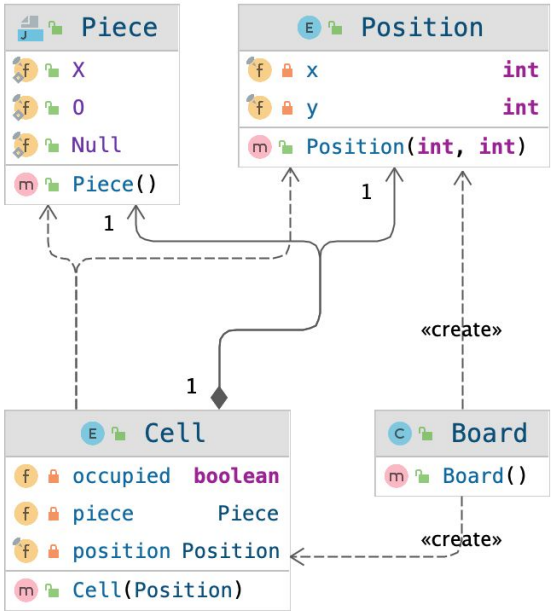
five in a row



all blocked

Building Blocks

Entities



The **Board** extends the **HashSet** class, whose type is class **Cell**.

Every **Cell** has a member called **Position**, a class containing two coordinates, x and y.

The Cell is either free or occupied by a **Piece**, an enum that can either be *X*, O or null. When a Position is free, the Piece it contains is *Null*.

Entities

These 4 entities allow us to update the status of the board given the provided inputs.

```
protected Position makeMove() {
    Position inputPosition = display.askPosition();
    Piece inputPiece =  display.askPiece();
    board.getCellAt(inputPosition).placePiece(inputPiece);
    this.BlockChecker.update(inputPosition);
    return inputPosition;
}

public Cell getCellAt(Position position) {
    return this.stream() Stream<Cell>
            .filter(c -> c.getPosition().equals(position))
            .findFirst() Optional<Cell>
            .orElse( other: null);
}
```

Game Flow

Main

```
public class Main {

    public static void main(String[] args) {

        Game game = new Game();
        game.playGame();

    }
}
```

Game flow

At every turn, the Game method **play()**:

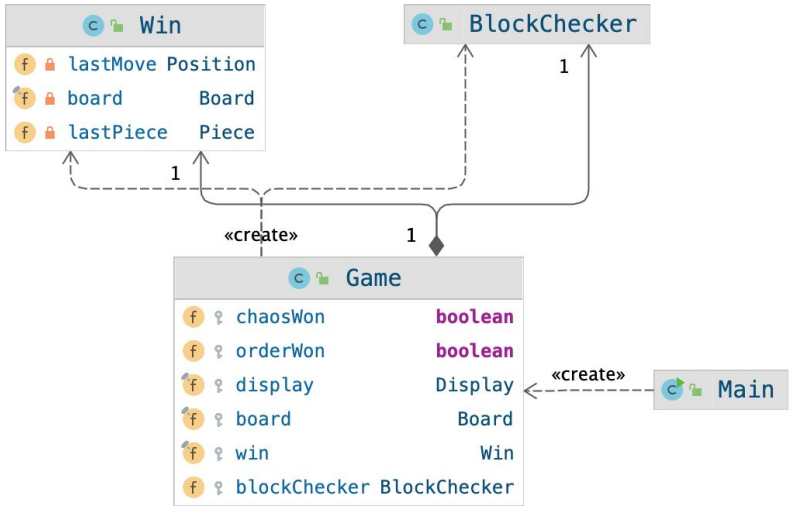
- Updates the board with the move given by the user
- Checks if a win condition is met

```
public void playGame() {
    this.init();
    this.play();
}

public void play(){
    while (!this.chaosWon && !this.orderWon) {
        this.display.displayPlayer(this.display.currentPlayer);
        this.display.changePlayer();
        Position lastMove = this.makeMove();
        this.checkBoard(lastMove);
        this.display.printBoard();
    }

    if(this.orderWon) {
        System.out.println("Order you won!");
    } else {
        System.out.println("Chaos you won!");
    }
}
```

End conditions



Order wins

Starting from the last move we extract the row, the column and the diagonals containing that cell.

```
public boolean checkWin(Position lastMove){
    this.lastMove = lastMove;
    this.lastPiece = board.getCellAt(lastMove).getPiece();
    return checkRow() || checkCol() || checkDiag() || checkAntiDiag();
}
```

Order wins

We check if any of those set contains a five pieces match and not six.

```
public boolean checkRowCol(Set<Cell> line) {
    return !allSixMatch(line) && (firstFiveMatch(line) || lastFiveMatch(line));
}

public boolean checkRow(){
    return checkRowCol(board.getRow(lastMove));
}
```

BlockChecker

```
public class BlockChecker extends HashSet<TreeSet<Cell>> {

    public BlockChecker(Board board) {
        super();
        for (Cell c : board) {
            if (board.getAntiDiag(c.getPosition()) != null)
                this.add((TreeSet<Cell>) board.getAntiDiag(c.getPosition()));
            if (board.getDiag(c.getPosition()) != null) this.add((TreeSet<Cell>) board.getDiag(c.getPosition()));
            this.add((TreeSet<Cell>) board.getCol(c.getPosition()));
            this.add((TreeSet<Cell>) board.getRow(c.getPosition()));
        }
    }

    public void update(Position position) {
        HashSet<TreeSet<Cell>> SetToRemove = this.stream()
                        .filter(s -> s.contains(new Cell(position)))
                        .filter(s -> (firstFiveBlocked(s) && lastFiveBlocked(s) || firstAndLastEquals(s)))
                        .collect(Collectors.toCollection(HashSet::new));
        SetToRemove.forEach(this::remove);
    }
```

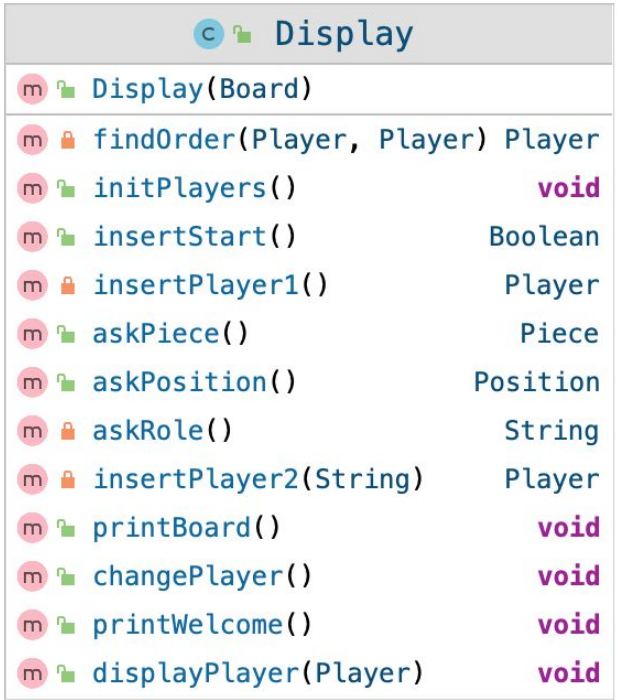
Chaos wins

Then chaos wins if BlockChecker, a Game member, is empty.

```
public void checkBoard(Position lastMove) {
    this.orderWon = this.win.checkWin(lastMove);
    this.chaosWon = this.blockChecker.isEmpty();
}
```

I/O & Exceptions Handling

I/O handling



Class **Display** takes care of game I/O both during the initialization and while playing the game.

Exceptions

Implemented three exceptions:

- NonValidPieceException()
- NonValidPosException()
- PosAlreadyOccupiedException()

used in **try-catch** structures in the **Display** class to assess whether the input position and piece given by the users are correct.

Testing

Test examples - Cell

Test the main methods of the **Cell** class:

- **isOccupied()**
- **getPosition()** to assess the cell is correctly initialized
- **getPiece()** through a parametrized test

```java
public class TestCell {
    private final Position position = new Position( x: 3,  y: 4);
    private final Cell cell = new Cell(position);

    @Test
    void testInstantiate() {
        assertFalse(cell.isOccupied());
    }

    @Test
    void testGetPosition() {
        assertEquals(position, cell.getPosition());
    }

    @ParameterizedTest
    @EnumSource(value= Piece.class, names={"X", "O"})
    void testPiece(Piece piece) {
        cell.placePiece(piece);
        assertEquals(piece, cell.getPiece());
    }
}
```

Test examples - Board

Test the main methods of the **Board** class:

- initialized with the correct dimension
- **placePiece()** correctly places the input piece
- **isFull()**

```java
@ParameterizedTest
@EnumSource(value=Piece.class, names={"X", "O"})
void testCheckPiece(Piece piece) {
    Position position = new Position( x: 4,  y: 4);
    board.getCellAt(position).placePiece(piece);
    assertEquals(board.getCellAt(position).getPiece(), piece);
}

@ParameterizedTest
@ValueSource(booleans = {true, false})
void testCheckFullBoard(boolean full) {
    if (full) {
        for (int i = 1; i <= 6; i++) {
            for (int j = 1; j <= 6; j++) {
                board.getCellAt(new Position(i, j)).placePiece(Piece.X);
            }
        }
    }
    assertEquals(full, board.isFull());
}
```

Test examples - Board

- parameterized tests to assess if **getCol()**, **getRow()** correctly return the required columns/rows
- similarly, for **getDiag()**, **getAntiDiag()**

```
@ParameterizedTest
@CsvSource({"2,2,0,X","2,2,0,O","2,1,1,X","2,1,1,O", "1,2,-1,X","1,2,-1,O"})
void testCheckGetDiag(int x, int y, int diff, Piece piece) {
    for (int i = 1; i <= 6; i++) {
        for (int j = 1; j <= 6; j++) {
            if (i-j == diff) {
                board.getCellAt(new Position(i, j)).placePiece(piece);
            }
        }
    }
    Set<Cell> diag = board.getDiag(new Position(x,y));
    if (diff < -1 || diff > 1) {
        assertNull(diag);
    } else {
        assertTrue(diag.stream().allMatch(c->c.getPiece() == piece));
    }
}
```

Test examples - Display

Test the methods asking the piece and position to the user:

- **askPiece()**
- **askPosition()**

by simulating the user input and assessing whether the piece/position is correctly acquired.

```java
@ParameterizedTest
@ValueSource(strings = {"X", "O"})
void testInputPiece(String expected) {
    System.setIn(new ByteArrayInputStream(expected.getBytes()));
    Piece exp_piece = Piece.valueOf(expected);
    assertEquals(exp_piece, display.askPiece());
}

@ParameterizedTest
@CsvSource({"1,1", "6,6", "6,1", "4,3"})
public void testInputPosition(String x, String y) {
    String simulatedUserInput = x + "," + y;
    System.setIn(new ByteArrayInputStream(simulatedUserInput.getBytes()));
    Position position = new Position(Integer.parseInt(x), Integer.parseInt(y));
    assertEquals(position, display.askPosition());
}
```

Test examples - Block

Parameterized tests to assess the correct update of the **BlockChecker** object, after placing pieces in the board and calling the **update()** method.

```java
@ParameterizedTest
@CsvSource({"3,3,4,4", "1,2,5,6"})
void testCheckBlockingDiag(int row1, int col1, int row2, int col2){
    Position pos1 = new Position(row1,col1);
    Position pos2 = new Position(row2,col2);
    game.board.getCellAt(pos1).placePiece(Piece.X);
    game.blockChecker.update(pos1);
    game.board.getCellAt(pos2).placePiece(Piece.O);
    game.blockChecker.update(pos2);
    assertFalse(game.blockChecker.contains((TreeSet<Cell>) game.board.getDiag(pos2)));
}

@ParameterizedTest
@ValueSource(ints={1, 2, 3, 4, 5, 6})
void testCheckBlockingLastAndFirstEquals(int col){
    Position pos1 = new Position( x: 1,col);
    Position pos2 = new Position( x: 6,col);
    game.board.getCellAt(pos1).placePiece(Piece.X);
    game.blockChecker.update(pos1);
    game.board.getCellAt(pos2).placePiece(Piece.X);
    game.blockChecker.update(pos2);
    assertFalse(game.blockChecker.contains((TreeSet<Cell>) game.board.getCol(pos2)));
}
```

Test examples - Game

Test the correctness of the **checkWin()** method of the Game class.

```
@ParameterizedTest
@ValueSource(booleans={true, false})
void testWinDiag(boolean win) {
    long numberOfCells = 6;
    if (win) { numberOfCells = 5; }
    game.board.getDiag(new Position( x: 1, y: 1)) Set<Cell>
            .stream() Stream<Cell>
            .limit(numberOfCells)
            .forEach(cell -> cell.placePiece(Piece.X));
    assertEquals(game.win.checkWin(new Position( x: 1, y: 1)), win);
}
```

Graphical User Interface

Graphical User Interface

JavaFx: GUI toolkit for the Java platform

Cascading Style Sheets (**CSS**): stylesheet language to easily change the application's look

GUI

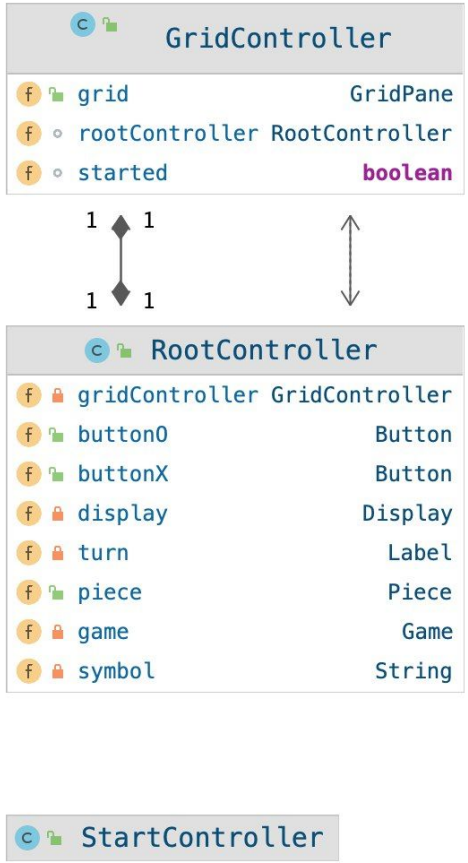
The design follows the Model-View-Controller (**MVC**) pattern:

- **Model**: core application, data and access/updates rules
- **View**: renders the contents of the Model (FXML files)
- **Controller**: translates the user's interactions with the View into actions performed by the Model

GUI is the main class (extends *javafx.application.Application*)

In addition:

- 4 **Controllers** handling the mouse inputs
- 6 **FXML source files** defining the user interface



Play by yourself!

Unix-like systems

Command line version : *./gradlew runConsole --console plain*

GUI version : *./gradlew run --console plain*

Windows systems

Command line version : *gradlew.bat runConsole --console plain*

GUI version : *gradlew.bat run --console plain*

Demo

Demo - Console

```
> Task :runConsole
Welcome to

If you want to know the rules write RULE, otherwise write PLAY to start playing the game.
RULE
RULES OF THE GAME
Order and Chaos is a variant of the game tic-tac-toe on a 6x6 gameboard.
The player Order strives to create a five-in-a-row of either Xs or Os either vertically, horizontally, or diagonally.
The opponent Chaos endeavors to prevent this.
Order plays first, then turns alternate.
Six-in-a-row does not qualify as a win.

If you want to know the rules write RULE, otherwise write PLAY to start playing the game.
PLAY
Insert your name and your role (order or chaos)!
Elena Chaos
You will play as order, insert your name!
Benedetta
Benedetta-order it's your turn!
Insert position as x,y
4,5
Insert piece
X

|     |     |     |     |     |     |
|     |     |     |     |     |     |
|     |     |     |     |     |     |
|     |     |     |     |  X  |     |
|     |     |     |     |     |     |
|     |     |     |     |     |     |
Elena-chaos it's your turn!
Insert position as x,y
3,2
Insert piece
O
```

Demo - GUI