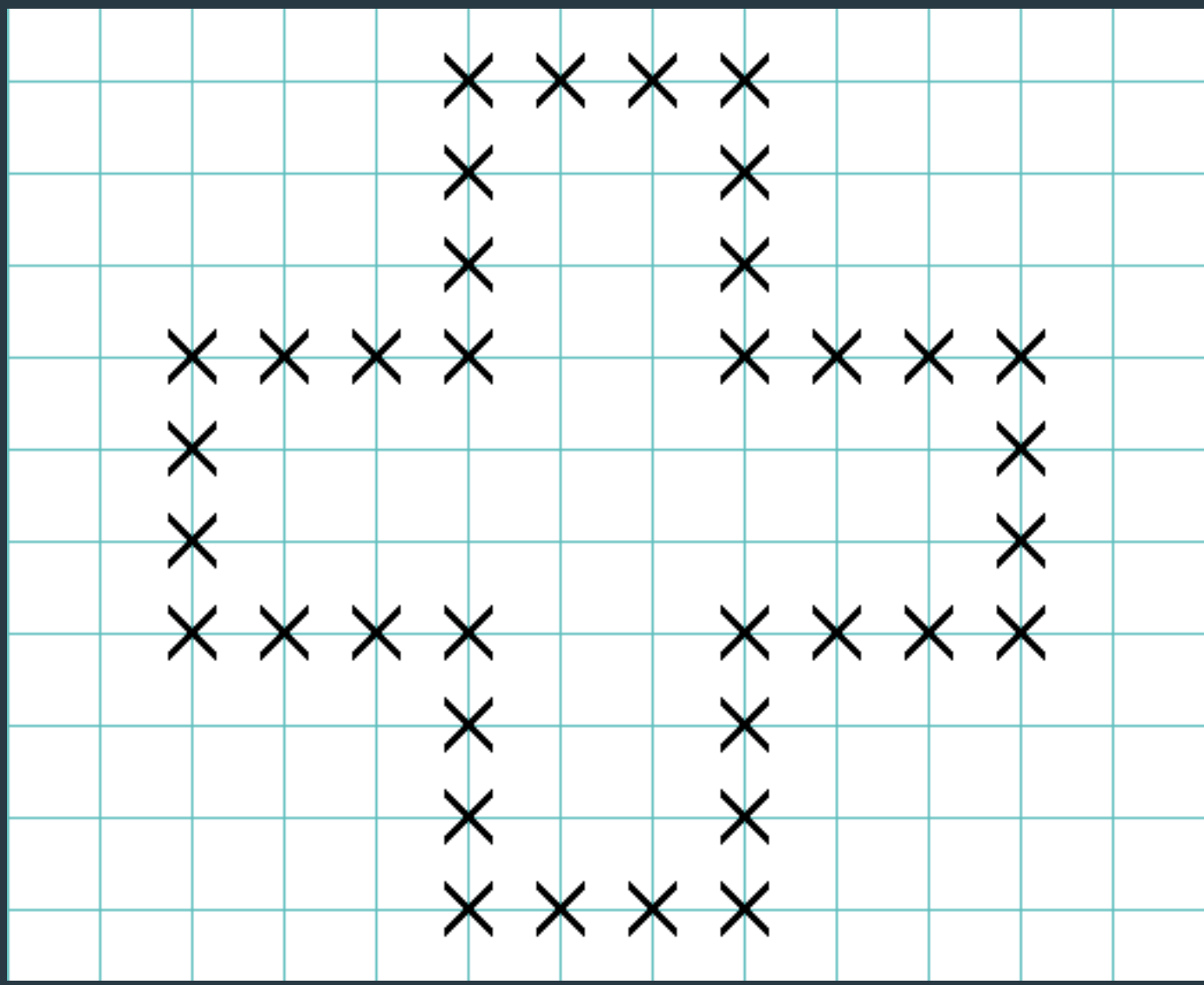
23/10/2023

Morpion Solitaire

A Java project by **NEVEU Pierre & DOUBABI Mustapha**



Introduction

Welcome to the Java documentation of our project, a digital adaptation of the game ***Morpion Solitaire,*** a single-player puzzle game. The objective is to fill the Maltese cross-shaped board with symbols (typically "X" or "O") according to specific rules, typically involving adjacent symbols.

This project aims to provide an engaging digital rendition of **Morpion Solitaire,** implementing the game's rich strategy.

This documentation serves as a guide for developers and users, elucidating the code's structure and the methods used to mirror the game's intricacies. Our design divides the code into key game components, such as Grid, Lines, and Points. Whether you're a developer interested in the project's technical aspects or a user keen to explore the digital ***Morpion Solitaire***, this document will provide valuable insights.

Key classes & enums

Here are the main classes that have been implemented; we'll go into more detail about the methods later.

CLASSES & ENUMS
GRID
LINE
ORIENTATION
POINT
SCOREBOARD
DIRECTION
STATUS
GAMEMANAGER

Grid class

Overview

The **Grid** class manages the game board, coordinates points, and lines. It offers methods for updating the grid, finding playable points, and checking point playability. The choice of a 25x25 grid size is due to the fact that no solution today can exceed this limit (the world record is held in an 18x22 grid).

- **Attributes:**
  - **size**: Length of the square grid's sides.
  - **grid**: A map associating point hash codes with their corresponding points.
  - **lines**: A set containing all grid lines.
  - **playablePoints**: A map linking points to sets of lines, identifying playable points.
  - **minPlayablePoint** and **maxPlayablePoint**: Define the sub-grid's bottom-left and top-right corners for playable point searches.
- **Methods:**
  - **updateGrid(Point playedPoint)**: Updates the grid with a played point and adjusts the playable point search area.
  - **updatePlayablePoints()**: Discovers and updates all playable points within the defined sub-grid.
  - **findLinesAround(Point point)**: Identifies possible lines that can be created from a played point.
  - **findLinesInDirection(Point point, Direction direction)**: Finds lines in a specific direction from a given point.
  - **getSubGrid(Point minPlayablePoint, Point maxPlayablePoint)**: Retrieves points within a specified sub-grid.
  - **checkPlayability(Point point)**: Checks point playability based on its presence in the playable points list.
  - **addLine(Line newLine)**: Incorporates a new line into the grid.
  - **getSize()**: Returns the grid's size.
  - **getLines()**: Provides all lines on the grid.
  - **getPlayablePoints()**: Offers access to the map of playable points.
  - **getGrid()**: Accesses the grid map.

The **Grid** class is an integral part of the game, responsible for managing the game board, tracking playable points, and facilitating game logic.

Line class

Overview

The **Line** class represents a collection of points forming a line on the game board. It includes methods for adding points to the line and offers information about the line's direction and points.

- **Attributes:**
  - **points**: A set of points that compose the line.
  - **direction**: Indicates the direction of the line, which can be horizontal, vertical, or diagonal.
- **Constructor:**
  - **Line(Set<Point> points, Direction direction)**: Initializes a line with a set of points and a specified direction.
- **Methods:**
  - **addPoint(Point point)**: Adds a point to the line.
  - **toString()**: Provides a string representation of the line, including its direction and points.
  - **equals(Object o)**: Compares two lines for equality based on their points.
  - **hashCode()**: Computes the hash code for the line using its points.
- **Getters:**
  - **getDirection()**: Retrieves the direction of the line.
  - **getPoints()**: Accesses the set of points that form the line.

The **Line** class serves as a fundamental component for representing lines within the game board, offering insights into the line's direction and constituent points.

Point class

Overview

The **Point** class represents a 2D coordinate point with x and y values. It includes methods for comparing, hashing, and moving points, as well as checking if a point is played.

- **Attributes:**
  - **x** and **y**: Integers representing the x and y coordinates of the point.
- **Constructors:**
  - **Point(int x, int y)**: Initializes a point with specified x and y values.
  - **Point(PlayedPoint p)**: Initializes a point based on a **PlayedPoint** object.
- **Methods:**
  - **toString()**: Generates a string representation of the point in the format "(x, y)".
  - **equals(Object o)**: Compares two points for equality based on their x and y coordinates.
  - **hashCode()**: Computes the hash code for the point using its x and y values.
  - **move(int x, int y)**: Updates the point's x and y coordinates.
  - **isPlayed()**: Determines if the point is a played point (instance of **PlayedPoint**).
  - **getX()**: Retrieves the x-coordinate of the point.
  - **getY()**: Retrieves the y-coordinate of the point.

The **Point** class serves as a fundamental representation of 2D coordinates, offering essential methods for managing and comparing points in the game board.

PlayedPoint subclass

Overview

The **PlayedPoint** class represents a point on the game board that has been played. It extends the base **Point** class and includes additional attributes and methods to manage played points.

- **Attributes:**
  - **playedPointsCount**: A static integer that counts the total number of played points.
  - **id**: An integer representing the order in which the current point was played.
  - **involvedInDirections**: A set of directions in which the played point is involved.
- **Constructors:**
  - **PlayedPoint(int x, int y)**: Initializes a played point with specified x and y values. It increments the played points count.
  - **PlayedPoint(Point p)**: Initializes a played point based on a **Point** object. It also increments the played points count.
- **Methods:**
  - **toString()**: Generates a string representation of the played point in the format "Played point: (x, y)".
  - **resetPlayedPointsCount()**: Resets the played points count to zero.
  - **getId()**: Retrieves the id of the played point.
  - **getCount()**: Retrieves the total count of played points.
  - **getInvolvedDirection()**: Retrieves the set of directions in which the played point is involved.

The **PlayedPoint** class extends the base **Point** class to specifically represent played points on the game board. It provides methods to access the order in which the point was played and the directions in which it is involved.

GameManager class

Overview

The GameManager class serves as a central manager for the game, responsible for setting up, starting, and maintaining the game state. It tracks the game version, player scores, and the game board. Additionally, it provides methods for starting and ending the game and displaying rankings.

- Attributes:
  - LOGGER: A logger instance for recording game-related messages.
  - classement: A scoreboard to keep track of player scores.
  - currentMod: A string indicating the game version (5T or 5D).
  - score: An integer representing the game score.
  - currentPlayer: A string storing the current player's name.
  - board: A reference to the game board represented by the Grid class.
- Constructor:
  - GameManager(Integer version): Initializes a GameManager object with the specified game version (5T or 5D).
- Methods:
  - setupGame(): Sets up the game by creating a game board, initializing the score, and preparing the scoreboard. It then starts the appropriate game version (5T or 5D).
  - startGameT(): Starts the game in the 5T version (currently unavailable).
  - startGameD(): Starts the game in the 5D version.
  - endParty(int score): Ends the game by recording the final score for the current player.
  - getScore(): Retrieves the game score.
  - getPlayerName(): Retrieves the name of the current player.
  - getGrid(): Retrieves the game board represented by the Grid class.
  - getVersion(): Retrieves the game version (5T or 5D).
  - displayRanking(): Displays the game rankings using the classement scoreboard.

The GameManager class functions as a central coordinator for the game, managing game setup, progression, and the final ranking of players.

ScoreBoard class

Overview

The **Scoreboard** class is responsible for managing and storing player scores in a text file. It allows adding player scores and writing them to the specified file.

- **Attributes:**
  - **path**: A constant string representing the file path where scores are stored.
  - **scores**: A **TreeMap** that associates player scores with player names.
- **Constructor:**
  - **Scoreboard()**: Initializes a **Scoreboard** object. It attempts to read existing scores from the file defined by **path** and stores them in the **scores** map.
- **Methods:**
  - **addScore(String playerName, int score)**: Adds a player's score to the **scores** map, associating it with the player's name.
  - **write()**: Writes the player scores from the **scores** map to the file specified by **path**. Existing scores in the file are overwritten.

The **Scoreboard** class facilitates the management of player scores and their persistence to a file. It allows adding new scores and updating the scoreboard file with the latest results.

Search method code

Here is the pseudo code of the method that looks for lines around a normal point that the latter could form.

```
For each unplayed point:
HashSet<Line> possibleLinesAround = new HashSet<>()
For each direction (Horizontal, Diagonal, Vertical):
                HashSet<Line> possibleLinesInDirection = new HashSet<>()
                HashSet<Point> points = new HashSet<>()
                        For each orientation (N, S, E, W, NW, NE, SW, SE):
                                For each neighbour in the orientation at a distance <= 4:
                                If (neighbour i is unplayed) or (neighbour i is in a line of the same direction)
                                                Break
                                else if points.size() == 4
                                        points.add(current unplayed point)
                                        possibleLinesInDirection.add(new Line(points, current direction)
                                        points.clear()
                                else
                                        points.add(current neighbour)
possibleLinesAround.addAll(possibleLinesInDirection)
```

Here's a step-by-step explanation of the pseudocode:

- **For each normal point:** We're checking each non-played point on the board.
- **Create a list to store lines:** We create an empty list to store the lines of points.
- **For each direction:** We're going to check in different directions.
- **Create a new line:** For each direction, we create an empty line.
- **If the line has 4 points, add it to the list:** If we find a line with 4 points, we add it to our list of lines and stop looking in that direction.

Search method code

- **For each of the 4 neighboring points:** We're going to look at the 4 points around our current point.
- **If the neighboring point is played and in a different direction:** If a neighboring point is already played and not in the same direction as the line we're checking, we add it to the line.
- **If not, exit the loop:** If the neighboring point doesn't meet the conditions in 7, we stop looking in that direction.
- **If the line has 4 points, add it to the list:** If we now have a line with 4 points, we add the initial point we started with, making a complete line, and add it to our list of lines.

This process repeats for each non-played point on the board, searching in different directions for lines of 4 points.

Class Diagram