OOSD Board Game

OOSD Board Game, for Object Oriented Software Design (ISYS1083), Group G3.

FOR TEACHERS: Please refer to PDF version of this document if you can't render markdown in your environment. Thanks.

Build environment

This program is written and tested in JDK 8 environments. Since lambda and some other newer JavaFX features have been used, JDK 7 will not work.

If you are using OpenJDK in some Linux distro, please remember to configure JavaFX separately.

You may also need to set Cofoja separately before compile. Please refer to this tutorial for more information.

Author (Assignment 1 branch)

Code written and debugged by Ming Hu (s3554025), partially reviewed by other group members, which are:

- Xuan Gia Khanh Nguyen (s3636905)
- Yixiong Shen (s3700889)
- Nguyen Tuan Manh (s3574923)

Main features implemented

- [x] Basics
  - [x] JavaFX GUI
  - [x] MVC design structure
- [ ] Gameplay
  - [x] Round based game demo
  - [x] Countdown timeout for each turn
  - [ ] Capture phrases (Assignment 2)

- [ ] Revert support (Assignment 2)

- [x] Board
  - [x] 8x8 board, 64 cells
- [x] Piece
  - [x] CSS styled
  - [x] Movable
- [x] Player
  - [x] Initial mark deduction info
  - [x] Turn based

Methods/Constructors with Cofoja (DbC as required)

- `controllers`
  - `GameLogic::commitMapChanges`
  - `GameLogic::timeout`
  - `GameLogic::selectPiece`
  - `GameLogic::placePiece`
  - `HomeController::commitUIChanges`
  - `HomeController::commitPlayerSelection`
    - useful for some misuse cases
- `helpers`
  - `BoardButtonHelper::parseClickResult`
  - `PieceFactory::createRandomPieceList`
  - `PieceFactory::createRandomCoordinateQueue`
- `models`
  - `Board::getPieceList`
    - useful for detecting logic issues in PieceFactory
  - `Board::setPieceList`
    - useful for detecting logic issues in PieceFactory
  - `BoardCellCoordinates`
  - `Coordinate`
  - `Coordinate::getPosX`
  - `Coordinate::getPosY`
  - `Coordinate::setPosX`
  - `Coordinate::setPosY`
  - `Piece::getStyle`

- `Piece::getAttackLevel`
- `Piece::applyAttack`
- `Piece::getCoordinate`
- `Piece::setCoordinate`
- `Player`
- `Player::getPlayerName`
- `Player::setPlayerName`

GRASP features / design structures

Low coupling

- Original Piece class is abstract class with different style/type of pieces extends it.
- UI controls are in a `Map<String, Object>`, where the strings are UI controls' ID and the objects are their references. Later in Assignment 2, if we need to increase the board size, we just need to draw the UI and adjust some logics, no need to care about models.

High cohesion

- `GameLogic` class and `HomeController` class
  - No method with long branch of code
  - Well categorised, `GameLogic` in charge of game logic only, while `HomeController` handles UI stuff.
  - But they also need to work together anyway...
- `Board` class, `Piece` class and `Coordinate` class
  - Model classes to store game status, with levels
  - Different level has its own responsibility, but they need to work together.
  - `Board` contains a list of `Piece`, each `Piece` has its own `Coordinate`

Controller

- This app is based on MVC design structures, so it has controllers
- Controllers are `GameLogic` which controls gameplay logic and `HomeController` which controls GUI ***(it also sounds like Pure Fabrication to some extent???)***

Polymorphism

- `ICoordinate` interface for `Coordinate` and `BoardButtonCoordinate`

Indirection

- `GameLogic` controls gameplay logic and `HomeController` controls GUI.
- If models need to control UI, they need to talk to `HomeController` first to ensure no invalid data is updated to UI.

SOLID features / design structures

Open/Close Principle

- `GameLogic::selectPiece` and `GameLogic::placePiece` are in **private** level
  - These two methods should not be changed and misused by others
- All the pieces extends an abstract Piece class
- Piece's attack level, total marks, styling string are in **final** to prevent design flaws caused by misuses.

Liskov Substitution Principle

- The `Board` contains a list of `Piece`
- Each `Piece` contains a `Coordinate`

Dependency Inversion Principle

- `ICoordinate` interface and `Piece` abstract class

The Don’t Repeat Yourself Principle

- `Piece` is abstract class, not interface. Common attributes (e.g. coordinates, marks, attack level) can stays in abstract class without re-implement them again in the sub-classes.
- It significantly reduces extra code.

Dependency Inversion

- The way of `GameLogic` class and piece list in `Board` class dealing with different pieces.