

# OOSD Board Game

---

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

## IMPORTANT NOTICE FOR TEACHERS

---

- Please refer to PDF version of this document if you can't render markdown in your environment. Thanks.
- **The following Design Principle sections in this README document are just a for another reference, as it was written for a draft by Ming Hu for other group members without extra checking. If there is any conflicts against with the Design Principle section in the report, please refer to the report instead, not this README.**

## 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)

## 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.

## Main features implemented

---

- ☒ Basics

- ☒ JavaFX GUI
- ☒ MVC design structure
- ☐ Gameplay
  - ☒ Round based game demo
  - ☒ Countdown timeout for each turn
  - ☐ Capture phrases (Assignment 2)
  - ☐ Revert support (Assignment 2)
- ☒ Board
  - ☒ 8x8 board, 64 cells
- ☒ Piece
  - ☒ CSS styled
  - ☒ Movable
- ☒ Player
  - ☒ Initial mark deduction info
  - ☒ 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 principles

---

The following Design Principle sections are just a for a reference as it was written for a draft by Ming Hu for other group members. If there is any conflicts against with the Design Principle section in the report, please refer to the report instead, not this README.

### 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 principles

---

The following Design Principle sections are just a for a reference as it was written for a draft by Ming Hu for other group members. If there is any conflicts against with the Design Principle section in the report, please refer to the report instead, not this README.

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

## The Don't Repeat Yourself Principle

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