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.
- Please be aware that our group contribution are not equal, i.e. not everyone has 25%. The rate is listed below:

Member	Rate	Works done
Ming Hu	45%	All coding works, including this README file; Diagram corrections; Presentation slide corrections
Yixiong Shen	18.3%	Presentation slides
Xuan Gia Khanh Nguyen	18.3%	Class diagrams
Tuan Manh Nguyen	18.3%	Final report

  Please refer to the contribution confirmation sheet for more details.

- **This README file is a reference for team member to understand the code and help them for designing diagram, presentation slide and reports. If there is any conflicting content with the final report, please refer to that document first.**

Author (Assignment 2 branch)

Code written, tested and debugged by Ming Hu (s3554025).

Icons

Icons are from Alibaba's IconFont, free for non-commercial use.

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

- [x] Basics
  - [x] JavaFX GUI
  - [x] Menu bar
  - [x] MVC design structure
- [x] Gameplay
  - [x] Round based game demo
  - [x] Countdown timeout for each turn
  - [x] Capture phrases (Assignment 2)
  - [x] Defensive mode (Assignment 2)
  - [x] Grade/win state (Assignment 2)
- [x] Board
  - [x] 8x8 board, 64 cells
  - [x] Un-do/Re-do support (Assignment 2)
  - [x] Status save/reload (Assignment 2)
  - [x] Board resizing (Assignment 2)
- [x] Piece
  - [x] CSS styled
  - [x] Movable
- [x] Settings (Assignment 2)
  - [x] Variable board size
  - [x] Variable piece count

    - [x] Custom player name
- [x] Player
    - [x] Initial mark deduction info
    - [x] Turn based
- [x] "About" window
- [x] Hotkey bindings on menu bar

Design patterns

Assignment 2 has implemented/refactored with 5 design patterns, which are:

- Prototype
    - located in `models.piece.PieceGenerator` class
        - It is used for creating pieces correctly
        - prevents using unnecessary initialization code.
- Command
    - located in `PieceFactory`, `PieceCreator` class
        - It shortens the code with lambda (no more if-else or switch-case needed)
        - simplify the process for adding new pieces/characters
- Decorator
    - located in package `models.pieces`
        - It simplifies the process for adding new pieces/characters
        - It allows flexible extension to a existing piece
- Chain of Responsibility
    - located in `helpers.reactions` package
        - It simplifies & decouples the request when the app creates notification/logging
- Abstract Factory
    - located in `models.factory` package (together with command pattern)
        - It limits direct access for concrete classes.

Methods/Constructors with Cofoja (DbC as required)

NOTICE: Since Cofoja may not work correctly on Intellij IDEA and the program may be failed to compile. The development environment follows the configurations in this tutorial and the project can pass compiling and testing with no issue.

These code mentioned below are using Cofoja:

- controllers
  - logic
    - `CompeteManager`
    - `GameLogic`
    - `StatusManager`
  - `HomeController`
  - `SettingController`
- helpers
  - reactions
    - `CrashReactions`
    - `DebugReactions`
    - `WarningReactions`
  - `BoardButtonHelper`
- models
  - board
    - `Board`
  - coordinate
    - `Coordinate`
  - factory
    - `PieceFactory`
    - `PlayerFactory`
    - `CoordinateFactory`
  - piece
    - some decorator classes
    - `PieceGenerator`
    - `SimplePiece`
  - player
    - `CommunismPlayer`
    - `CapitalismPlayer`

GRASP principles

Low coupling

- Piece model classes are using Decorator design pattern. The properties in a certain piece are decorated by decorator classes.

- Buttons in the board are automatically generated, not directly written into FXML. It also allows users to change the size of board between 6x6 to 18x18 dynamically.

High cohesion

- `GameLogic` class and `HomeController` class
    - No method with long section of code
    - Well categorised, `GameLogic` in charge of game logic only, while `HomeController` handles UI stuff.
- `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`

Creator

- Abstract factory pattern is used in this project for creating contents in the board.

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

- `Piece` interface for piece models
- `Player` interface for player models
- `PieceGenerator` interface for generating

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.
- Also for `CompeteManager` and `StatusManager`. If the data is invalid, then it won't be updated to UI. Instead, there will be a proper warning message/debug log will be shown.

SOLID principles

Single Responsibility Principle

- Game logic code are separated to three classes, which are `GameLogic`, `CompeteManager` and `StatusManager`.
    - `GameLogic` only responsible the low-level logics
    - `CompeteManager` handles the move range and attack range evaluation
    - `StatusManager` saves or restores the status or the board

Open/Close Principle

- `GameLogic::selectPiece` and `GameLogic::placePiece` are in **private** level
    - These two methods should not be changed and misused by others
- Piece models are in proper

Liskov Substitution Principle

- `SimplePiece` has the correct implementation under `Piece`

Interface segregation principle

- All piece methods in `Piece` interface are in use

Dependency Inversion Principle

- `Piece` interface for piece models
- `Player` interface for player models
- `PieceGenerator` interface for generating
- The way of `GameLogic` class and piece list in `Board` class dealing with different pieces.

The Don’t Repeat Yourself Principle

- Piece models are using decorator pattern and command pattern with lambda. It significantly reduces extra code in the initialization.