Code Smell & Refactoring

1. Dispensables: Duplicate Code

Problem occurrence: **TileMarker & TilePiece** (commit)

The TilePiece (Piece class) is responsible for rendering each individual tile, while TileMarker (Marker class) is responsible for marking tiles that are legal moves for the current player.

Originally, the colour scheme is stored individually in each class, with the Marker class fetching colour scheme from the Piece class at runtime, as the classes are intended to have the same colour when under the same colour state. However, this also required separately setting the colour scheme post-instantiation, and leaves duplicated methods (getPieceColour, getPieceInverseColour) and fields (pieceColour, colour states) in both classes.

Solution: The resolution is to create a superclass Tile and pull duplicate elements up, then change both classes to extend Tile.

Originally, merging both classes was also considered, but was rejected as the merged class may become excessively bloated. Additionally, it will also require the merged class to be responsible for both rendering the pieces and marking legal moves, breaking single responsibility and requiring additional parameters to be passed for construction.

Result: After refactoring, both classes will have their colour scheme updated at once when invoking Tile.setPieceColour(), as the class will pull the latest colour from superclass Tile instead of being stored separately in each class/instance. If new accessibility/customisation features are added that requires changing piece colours, the implementation will be simpler and easier as a result.

2. Couplers: Feature Envy

Problem occurrence: **MenuBar & MenuBarButton** (commit)

The MenuBar is a class that is responsible for rendering the menu bar in each screen, which will contain MenuBarButtons for players to interact with for different functions (go to settings, save, load, etc.).

As the project progresses, the MenuBarButton has become a de facto generic button class that is used beyond menu bars. Examples include the buttons in the main menu, and a subclass extending MenuBarButton in the settings screen. The MenuBar class was also responsible for generating button presets for each screen, as well as being used as a container for collections of MenuBarButtons.

This contravenes the single responsibility principle and separation of concerns, as the buttons were not instantiated separately from menuBar, causing problems in screens that do not require the menu bar rendered such as the main menu screen.

Solution: Therefore, the solution is to rename MenuBarButtons to simply Buttons to clarify the program structure, separate button preset generation methods to its own class, and remove the responsibility of button container from the MenuBar class.

We did not create a factory class for individual button generation as the buttons required for each panel are static, with all buttons having predetermined functionality at compile time. Thus, refactoring as such is rather overkill, taking away time that can be put towards developing other features when the current solution is sufficient for our purposes.

Result: After the refactor, buttons and the menu bar are now independent, allowing them to be instantiated and rendered separately, removing the need for MenuBar class to determine whether to render itself or not depending on the current active panel.

3. Coupler: Message Chains

Problem occurrence: **Gameplay** (commit)

The Gameplay panel class is responsible for rendering UI elements during gameplay. The paint() method will fetch the current game state every invocation to ensure it is consistent with the displayed game board.

To achieve this, the original implementation repeatedly calls GameLogic.getInstance().getBoard().getPiece(x, y) to retrieve the current state. The long method chain impedes code readability as there were multiple similar lines in the paint method. Additionally, as the method chain is inside a loop, even though game state does not change during paint(), the repeated game instance/board reference retrievals are unnecessary, adding unnecessary runtime to each paint() call.

Solution: Therefore, it is simplified by having paint() retrieve and store the reference to game state and game board at the start, and rewrite each message chain to use the stored references.

Result: The former method chains now require no chaining, improving readability of the code, as well as improving performance by removing the repeated reference retrievals.