Mocking
(Pawn Game)

Overview

In this assignment, you will:

- Demonstrate your knowledge of using mocking for testing by testing a "Controller" in an application that uses the Model-View-Controller architecture.
- Keep track of your progress using version control.
- Write passing unit tests for the class that test the behaviour of the Controller.
- Report bugs in an issue tracking system.
- Determine how well your code is tested using code coverage.
- Check for memory leaks using a memory checker.

Overview of the Application

The Application Under Test (AUT) is a simplified version of the game of chess. The game is played on a 5x5 board; only pawns are used. Each player's pawns start on the back rows (i.e. rows 1 and 5). The rules that govern a pawn's movement remain relatively the same as in chess:

- Pawns can only move forward one space, except on their first move when they can move two spaces.
- A pawn that moves two spaces on its first move to "avoid capture" by an opponent's pawn can be taken "en passant" (see https://www.chess.com/terms/en-passant).
- There is no promotion of pawns when the opponent's back rank is reached, as there are no other pieces for which the pawn can be promoted.
- The game is over when the opponent's pawns can be captured (i.e. "last man standing").
    - For simplicity of the assignment, we'll assume that players will not move in such a way as to cause a stalemate where neither player will be able to capture the opponent's remaining pawns.

Instructions

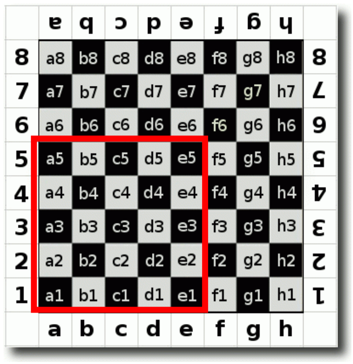
Setup

1. This assignment is done in pairs. Have one of the group members fork the assignment repository.
2. Add `mark3720` as `Reporter` so they can access your repository.
3. Submit the group's GitLab URL to Moodle.

Completing the Assignment

1. Create a unit test that uses mocking to test the behaviour of the `playGame()` method of `Chess`.
    a. You will need to create mock classes for the Model (i.e. `Board`) and View (i.e. `ChessUI`).

2. The behaviour of the `playGame()` method proceeds as follows:
    a. The method `setup()` is called to place the pieces on the board (i.e. `Board.placePiece()` is called for each piece).
    b. The board is drawn by calling `Board.draw()`. Also, this occurs after a player moves.
    c. The players will alternate moving their pieces starting with the player controlling the white pawns. `ChessUI.getLocation()` is used to get the location of the piece to move and where to move it. Players enter board coordinates using algebraic notiation (i.e. the lower left square is "A1" and the upper right square is "E5").

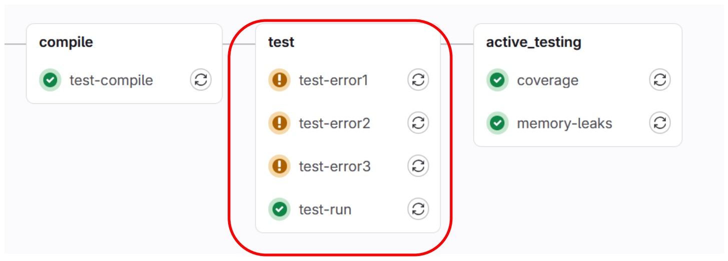

a b c d e f g h
8 a8 b8 c8 d8 e8 f8 g8 h8 8
7 a7 b7 c7 d7 e7 f7 g7 h7 7
6 a6 b6 c6 d6 e6 f6 g6 h6 6
5 a5 b5 c5 d5 e5 f5 g5 h5 5
4 a4 b4 c4 d4 e4 f4 g4 h4 4
3 a3 b3 c3 d3 e3 f3 g3 h3 3
2 a2 b2 c2 d2 e2 f2 g2 h2 2
1 a1 b1 c1 d1 e1 f1 g1 h1 1
a b c d e f g h

    d. A piece is moved by calling `Board.movePiece()`.
    e. The game checks to see if there is a winner by calling `Board.checkWinner()`.
    f. If there is a winner, `ChessUI.gameOver()` is called to announce the winner.
    g. If an illegal move is made, an exception is thrown and `ChessUI.invalidMove()` is called to inform the player.
3. A correct implementation of Chess.playGame() is provided, as well as three "mutants" where the method is incorrectly implemented. Your tests are to catch the errors with these implementations.
    a. Create an issue **in the group's repository** reporting what the fault is for each of the three "mutants".
4. A `Makefile` is provided to help you build and test your program, run static analysis, memory leak checking, style checking and code coverage.
    a. The Makefile has the following targets:
        i. `chessTest`: Builds an executable (`testChess`) that tests for the correct implementation of `Chess.playGame()`
        ii. `testChess_errors`: Builds an executable that tests for incorrect implementations of `Chess.playGame()`. The executables are:
            1. `testChess_error1`
            2. `testChess_error2`
            3. `testChess_error3`
        iii. `memcheck`: Runs `valgrind –memcheck` to check for memory leaks.

iv. `coverage`: Generates a coverage report for `Chess.cpp`

1. `Chess.cpp` is to have 100% coverage.

5. The pipeline is pre-configured – there is no need to change it.
   a. The "error" tests should not pass. The pipeline will look like:



Example

The following is an example set of moves and actions that can be used for testing the game.

1. White moves from A1 to A3.
2. Black moves from B5 to B4.
3. White moves from A3 to B4.
4. `Board.checkWinner()` returns `true`;

Grading

You will be graded based on your demonstrated understanding of unit testing using mocking, version control, and good software engineering practices,. Examples of items the grader will be looking for include (but are not limited to):

- 100% coverage of `Chess.playGame()` using unit test(s) that use mocking to test the behaviour. [20 marks]
- Issues correctly describing the three errors in the "buggy" implementations. [9 marks]
- The status of the most recent build in your repository's GitLab pipeline nearest the deadline (but not after the deadline) is green (i.e. passes). [5 marks]
- Memory leak checking shows no problems with your code. [5 marks]

Submission

- Submit on Moodle the GitLab URL of your repository.
  - You may be asked to give specific users read access to your repository.