SWE4203: Project Milestone 2

Student name/ ID

Jack Huynh - 3708644

Nguyen Do - 3702089

EST. 1785 UNB

UNIVERSITY OF NEW BRUNSWICK

Issue 1: Placement conflict error after (1,2), (1,3), (2,1) and (3,1)

Task 1

- Test 1: Fail
  - Moves are made normally until moves at (1,2), (1,3), (3,1) or (2,1), when player X or O attempts to place a move at (1,1), (2,2) or (3,3) after 1 of those 4 moves.
  - Expected result: The game incorrectly returns a placement_confict error and skip that player's turn.

- Test 2: Pass
  - Player X places a move at (1,1) on their first move.
  - Expected result: No error, as this is the correct behavior for the first move.

- Test 3: Pass
  - Player O places a move at (3,3) on move #2.
  - Expected Result: No error since this is a valid move before the third move.

- After the issue is fixed, the system is expected to have no error during placement phase.

Task 2: Candidate impact set (CIS)

- When modifying the code, the methods/classes that are expected to be affected are:

- Game.java: This is the main class that contains methods related to the issues.
- PlayerResult play (Player player, int x, int y) method: This is the method in charge of placing the move and storing data of the game.
- Gamemanager.java: Game state of the system handling player interaction and update the state of the game.
- Index.css: HTML browser that runs all the class above to run a functional game program.

Task 3:

- See Git commit history.

Task 4 Actual impact set (AIS)

- PlayerResult play (Player player, int x, int y) method.
- Game.java
- Index.js and GameManager.java didn't get any impact from the changes since it only call the method when running to get the return result.

- There is no missing CIS during this test case.
- There is no ripple effect occurring when we fix the code.

Task 5

Limitation by architectural elements:
Single responsible principal violation:

- Ideally, a class should have one responsibility, but the Game.java class currently handle multiple responsibilities. This makes isolated changes more difficult and increase the risk of ripple effects.

Potential architectural modifications:
Separation of concerns:

- Pro: By separating the game logic from the communication logic into different classes or components, changes to the game rules would not affect the streaming logic and vice versa.
- Cons: Introducing more component could increase the complexity of the system. Requires more coordination between components.

Issue 2: Skipping player's turn on occupied position.

Task 1

- Test case 1: Pass
    - Player X selected position (1,1)
    - Player O selected position (2,2)

    - Expected result: No turn is skipped, and the game continues normally.

- Test case 2: Pass
    - Player X selected position (1,1)
    - Player O selected position (1,1)
    - Expected result: The game prompts player O to choose a different position because (1,1) is already occupied.

- Test case 3: Fail.
    - Player X selected position (1,1)
    - Player O attempts selected position (1,1) and is skipped due to the error.
    - Player X selects position (2,2)
    - Expected result: Player O's turn is skipped, highlighting the issue.

- After the issue has been fixed, the program is expected to ask the player to place again without being skipped.

Task 2: Candidate impact set (CIS)

- Game.java: Main class that game logic methods are in.
- PlayerResult play (Player player, int x, int y) method: Main function where the implementation code is placed.
- GameManager.java: Class that calls play methods to run the game.

Task 3:

- See Git commit history.

Task 4: Actual impact set (AIS)

- Game.java
- PlayerResult play method.

- There is no missing CIS in this case, GameManager.java runs the updated code without error.
- There is no ripple effect occurring when we fix the code.

Task 5

Architecture limitations

- The play method in the game class handles move validation, state change and turn management all in one place. This can result in bigger error in notifications due to state inconsistencies or logic errors.

Potential architectural

Decoupled validation logic

- Pros: Isolating move validation into its own method or class can make the code more readable and easier to debug.
- Cons: May increase the complexity of the codebase with more classes or methods to manage.

State pattern for game states.

- Pros: Implementing a state pattern can make the transition between game states more explicit and manageable.
- Cons: It introduces more classes and can overcomplicate design for a simple game.

Issue 3: Winning notification condition is coded wrong.

Task 1

- Test 1 (Fail)
  - A player achieves a winning condition in Tic-Tac-Toe game.
  - Expected result: Returns “No winner for this game” notification.

- Test 2 (Pass)
  - A player wins by standard rules.
  - Expected result: A winning notification.

- Test 3 (Pass)
  - The game state return as draw
  - Expected result: “No winner for this game” notification.

- After the issues are fixed, the notification when the game ends will be correctly displayed.

Task 2: Candidate impact set (CIS)

- Index.js: Main class where the setWinnerDisp method are located.
- setWinnerDisp method: the actual method that handles displaying the notification by checking the state of the game.

Task 3:

See Git commit history.

Task 4: Actual impact set (AIS)

- Index.js
- setWinnerDisp method

- There is no missing CIS in this case.
- There is no ripple effect occurring when we fix the code.

Task 5:

Architecture limitation:

- If the logic for sending notifications is directly tied to the game state without a separate layer for handling communication, this can result in erroneous notifications due to state inconsistencies or logic errors.

Potential Architecture modifications:

Observer pattern for notifications

- Pros: Allows for a clear separation of concerns where the game state is observed, and notifications are sent as a response to changes rather than being embedded in the game logic.
- Cons: Might require a significant refactor of the existing code and introduce new abstraction to manage.