**Project Sprint #4**

Implement all the features that support a player (**human or computer**) to play a simple or general SOS game against another player (**human or** **computer**). The minimum features include **choosing human or computer for red and/or blue players**, **choosing the game mode (simple or general)**, **choosing the board size**, **setting up a new game**, **making a move (in a simple or general game)**, and **determining if a simple or general game is over**. The computer component must be able to play complete simple and general games. You are encouraged to consider basic strategies for winning simple or general games (e.g., against a poor human player). Optimal play is not required.

The following is a sample GUI layout. You should use a class hierarchy to deal with the computer opponent requirements. If your current code has not yet considered class hierarchy, it is time to refactor your code.

|  |  |  |
| --- | --- | --- |
| SOS Icon  Description automatically generated Simple game Icon  Description automatically generated General game Board size  8 | | |
| Blue player  Icon                          Description automatically generated Human  Icon  Description automatically generated S  Icon  Description automatically generated O  Icon                          Description automatically generated Computer | Chart, line chart  Description automatically generated | Red player  Icon  Description automatically generated Human  Icon  Description automatically generated S  Icon  Description automatically generated O  Icon  Description automatically generated Computer |
|  | Current turn: blue (or red) | New Game |

Figure 1. Sample GUI layout of the working program for Sprint 3

**Total points: 24**

1. **Demonstration (8 points)**

Submit a video of no more than five minutes, clearly demonstrating that you have implemented the computer opponent features and written some automated unit tests.

1. A complete simple game where the blue player is a human, the red player is the computer, and there is a winner
2. A complete general game where the blue player is the computer, the red player is a human, and there is a winner
3. A complete simple game where both sides are played by the computer
4. A complete general game where both sides are played by the computer
5. Some automated unit tests for the computer opponent.

In the video, you must explain what is being demonstrated.

1. **User Stories for the Computer Opponent Requirements (1 points)**

* **User Story Template**: As a <role>, I want <goal> [so that <benefit>]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **User Story Name** | **User Story Description** | **Priority** | **Estimated effort (hours)** |
| 8  .. | Computer move | As a player, I want the computer opponent to use intelligent strategies to make the game more challenging and unpredictable. | My priority is to put in a function that assure an Ai move whenever computer is the choice. | 3 hours |
| The flow of the game | As a player, I want the computer opponent to respond quicky to my moves, so that the game flows smoothly and doesn’t become tedious | My priority is to implement a random function that randomly put “S” or “O” as soon as it’s the computer turn so that it doesn’t feels like it's 2 humans that are playing. | 2 hours |
|  |  |  |  |  |

1. **Acceptance Criteria (AC) for the Computer Opponent Requirements (4 points)**

Add or delete rows as needed.

|  |  |  |  |
| --- | --- | --- | --- |
| **User Story ID and Name** | **AC**  **ID** | **Description of Acceptance Criterion** | **Status (completed, toDo, inPprogress)** |
| 8.Make a computer move | 8.1 | AC 8.1 Valid Computer Move  Given that the player has made a move  When it’s the computer’s turn to play  Then the computer should select a valid cell on the game board and place either an “S” or “O” symbol in that cell | Completed |
| 8.2 | AC 8.2 Invalid Computer Move  Given the current turn is the computer opponent  When the computer opponent tries to make an invalid move on a non-empty cell or a cell that is outside the grid boundaries  Then the move should be rejected and an error message should be displayed to the user, prompting them to try again. | Completed |
| 8.3 | AC 8.2 Check for SOS after Computer move  Given that the computer has made a move,  When the game board is updated with the computer’s move,  Then the game engine should check for any SOS sequence created by the computer’s move, and increment the computer’s score accordingly. | In Progress |
| 9. | 9.1 | AC 9.1 <scenario description>  Given  When  Then |  |
| … |  |  |

1. **Summary of All Source Code (1 points)**

|  |  |  |
| --- | --- | --- |
| Source code file name | Production code or test code? | # lines of code |
| Board.java |  |  |
| General.java |  |  |
| Simple.java |  |  |
| GUI.java |  |  |
| ComputerTest |  |  |
| GameMenuTest.java |  |  |
| GeneralTest.java |  |  |
| SimpleTest.java |  |  |
|  |  |  |
| Total | |  |

**You must submit all source code to get any credit for this assignment.**

1. **Production Code vs New User stories/Acceptance Criteria (2 points)**

Summarize how each of the new user story/acceptance criteria is implemented in your production code (class name and method name etc.)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **User Story ID and Name** | **AC ID** | **Class Name(s)** | **Method Name(s)** | **Status (complete or not)** | **Notes (optional)** |
| 8 | 8.1 | makeAiMove() | This function make sure of the valid computer move in the game | Completed |  |
|  | 8.2 | Random rng = **new** Random(); | This make sure of the flow of the game by randomly selecting between “S” and “O” | In progress |  |
|  | 8.3 | **public** **abstract** **void** checkForWin( | This function checks for the win by checking if SOS is formed or not. | Completed |  |

1. **Tests vs New User stories/Acceptance Criteria (2 points)**

Summarize how each of the new user story/acceptance criteria is tested by your test code (class name and method name) or manually performed tests.

6.1 Automated tests directly corresponding to some acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User Story ID and Name** | **Acceptance Criterion ID** | **Class Name (s) of the Test Code** | **Method Name(s) of the Test Code** | **Description of the Test Case (input & expected output)** |
| 1 | 1.1 |  |  |  |
|  | 1.2 |  |  |  |
|  | … |  |  |  |
| 2 | 2.1 |  |  |  |
|  | … |  |  |  |

6.2 Manual tests directly corresponding to some acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **User Story ID and Name** | **Acceptance Criterion ID** | **Test Case Input** | **Test Oracle (Expected Output)** | **Notes** |
| 1 | 1.1 |  |  |  |
|  | 1.2 |  |  |  |
|  | … |  |  |  |
| 2 | 2.1 |  |  |  |
|  | … |  |  |  |

6.3 Other automated or manual tests not corresponding to the acceptance criteria

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Number** | **Test Input** | **Expected Result** | **Class Name of the Test Code** | **Method Name of the Test Code** |
|  |  |  |  |  |
|  |  |  |  |  |

1. **Present the class diagram of your production code (3 points) and describe how the class hierarchy in your design deals with the computer opponent requirements (3 points)**?

For my production code called Board.java, it’s class diagram is as follow:

**board**: An abstract class that represents the game board. It has an int size, a 2D array of Cell type called grid, a char type called turn to represent the current turn, an instance of Random class called rng, and a List of arrays of int type called redWinningPatterns and blueWinningPatterns.

**initBoard():** A method that initializes the grid and clears redWinningPatterns and blueWinningPatterns lists. It also sets the currentGameState to PLAYING, turn to 'B', totalMoves to 0, bluePoints to 0, and redPoints to 0.

**getCell(row: int, column: int):** Cell: A method that returns the Cell type of the specified coordinates if it's inside the grid bounds.

**setCell(row: int, column: int, cell: Cell):** **void**: A method that sets the Cell type of the specified coordinates if it's inside the grid bounds.

**getTurn(): char**: A method that returns the current turn.

**setTurn(t: char): void**: A method that sets the current turn.

**getGameState(): GameState**: A method that returns the current game state.

**makeAiMove(): void**: A method that generates a random move for the AI player.

**makeMove(row: int, column: int): boolean**: A method that makes a move for the current player if the specified coordinates are valid. It also updates bluePoints and redPoints if a point is earned, and checks for a win condition or a draw.

**updateState(): void**: A method that updates the current game state.

**doNotSwitchTurn(): void**: A method that doesn't switch the current turn.

**switchTurn(): void**: A method that switches the current turn.

**checkSos(row: int, col: int): int**: A method that checks if there's a winning pattern around the specified coordinates and returns the number of points earned.

**BlueBoard**: A class that extends board and represents the blue player's game board. It has an int type called bluePoints to store the blue player's points and a List of arrays of int type called blueWinningPatterns to store the blue player's winning patterns.

**RedBoard**: A class that extends board and represents the red player's game board. It has an int type called redPoints to store the red player's points and a List of arrays of int type called redWinningPatterns to store the red player's winning patterns.

For my production code GUI.java, the class diagram is as follow:

The GUI class has several class-level fields like CELL\_SIZE, GRID\_WIDTH, GRID\_WIDHT\_HALF, CELL\_PADDING, SYMBOL\_SIZE, SYMBOL\_STROKE\_WIDTH, blueO, redO, redS, blueS, computerBlue, computerRed, game, gameBoardCanvas, gameStatusBar, boardSize, graph, gameMode, activePlayerBlue, and activePlayerRed. The class-level fields are initialized with default values or null.

The GUI class constructor sets the default close operation, bounds, and layout of the JFrame. It sets the title of the JFrame to "SOS Game". It creates a JLabel "SOS" and two radio buttons "Simple Game" and "General Game" for game mode selection. It creates a ButtonGroup for radio buttons and adds them to it. It creates a JLabel "Board Size" and a JTextField for board size selection. The JTextField is added with an ActionListener to handle the board size input. It creates JLabels and JRadioButtons for player selection with their corresponding ButtonGroups.

The GUI class has a private class GameBoardCanvas which extends JPanel and implements the MouseListener interface to handle mouse events. It has a method **paintComponent()** which overrides the parent method to draw the game board with symbols and lines. The GameBoardCanvas class also has a method **getCellAtPoint()** which returns the Cell object at the point where the mouse event occurred.

The GUI class implements the ActionListener interface to handle the events on the radio buttons and text field. It has a method **actionPerformed()** which checks the action command of the source object and updates the game mode, active players, and board size based on the user selection.

The GUI class also has a method **updateGameStatusBar()** which updates the game status bar with the current game state. It has a method **newGame()** which creates a new game board and sets the active player and game state. It has a method **computerMove()** which simulates the computer move for the selected player.

Overall, the class diagram of the given code includes the GUI class, **GameBoardCanvas** class, board interface, GameState enum, and Cell class. The GUI class has a HAS-A relationship with GameBoardCanvas, board interface, and Cell class. The **GameBoardCanvas** class has an IS-A relationship with JPanel and MouseListener interface. The board interface has an IS-A relationship with Serializable interface. The GameState enum has a list of possible game states. The Cell class has a field to store the cell value and a method to get and set the cell value.