Cell Class:

```
import java.util.ArrayList;
import java.util.List;
/**
* Represents a single square on the checkerboard
*/
public enum Cell
{
  /**
   * An empty square
   */
  EMPTY(" "),
  /**
   * Square with a red piece in it
   */
  RED("●"),
  /**
   * Square with a black piece in it
   */
  BLACK("○"),
  /**
   * Square with a red piece that is kinged
   */
  RED_KING("♕"),
  /**
   * Square with a black piece that is kinged
   */
  BLACK_KING("♔");
  /**
   * The symbol to use for the cell
   */
  private final String symbol;
  Cell(String symbol)
  {
     this.symbol = symbol;
  }
  /**
   * Adding this to the piece's position moves them diagonally down right
   */
  private static final int RIGHT_DOWN = 7;
  /**
   * Adding this to the piece's position moves them diagonally down left
   */
  private static final int LEFT_DOWN = 9;
  /**
   * Adding this to the piece's position moves them diagonally up right
   */
  private static final int RIGHT_UP = -7;
```

```java
/**
 * Adding this to the piece's position moves them diagonally up left
 */
private static final int LEFT_UP = -9;
/**
 * Red king row start
 */
private static final int RED_KING_START = 56;
/**
 * Red king row end
 */
private static final int RED_KING_END = 64;
/**
 * Black king row start
 */
private static final int BLACK_KING_START = 0;
/**
 * Black king row end
 */
private static final int BLACK_KING_END = 8;
/**
 * Get the way a piece moves from the
 * @return A list of movement options for the piece
 */
public ArrayList<Integer> movement() {
    switch (this) {
        case RED:
            return new ArrayList<>(List.of(RIGHT_DOWN, LEFT_DOWN)); //Direction fixed
        case BLACK:
            return new ArrayList<>(List.of(RIGHT_UP, LEFT_UP));
        case RED_KING, BLACK_KING:
            return new ArrayList<>(List.of(RIGHT_DOWN, LEFT_DOWN, RIGHT_UP, LEFT_UP));
    }
    return new ArrayList<>();
}
/**
 * @return Is this red or red king
 */
public boolean isRedPiece() {
    return this.equals(RED) || this.equals(RED_KING);
}
/**
 * @return Is this black or black king
 */
public boolean isBlackPiece() {
    return this.equals(BLACK) || this.equals(BLACK_KING);
}
/**
 * @return Is this red or red king
```

```
     */
    public boolean isOpponent(Cell other) {
        switch (this) {
            case RED, RED_KING:
                return other.isBlackPiece();        //the three cases are fixed
            case BLACK, BLACK_KING:
                return other.isRedPiece();
            default:
              return false;
        }
    }
    /**
     * Try to promote a piece to king
     * @param cell_index The cell index
     * @return The new cell, promoted if at the promotion row
     */
    public Cell promoteIfReachedEnd(int cell_index) {
        switch (this) {
            case RED:
                if (RED_KING_START <= cell_index && cell_index < RED_KING_END) //fixed
                    return RED_KING;
                break;
            case BLACK:
                if (BLACK_KING_START <= cell_index && cell_index < BLACK_KING_END) //fixed
                    return BLACK_KING;
        }
        return this;
    }
    /**
     * @return If the cell is empty
     */
    public boolean isEmpty() {
        return this.equals(Cell.EMPTY);
    }
    @Override
    public String toString(){
        return symbol;
    }
}
```

GameState Class:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.NoSuchElementException;
import java.util.Scanner;
// Represents a GameState
public class GameState
{
  // Number of rows and columns on the board
  private static final int ROWS = 8;
  private static final int COLS = 8;
  private ArrayList<Cell> cells; // cells on the game board
  private boolean blackTurn;     // flag for current turn
  /**
   * Creates a new GameState
   */
  private GameState() {}
  /**
   * Creates a GameState
   * @param cells the 64 cells in the board
   * @param blackTurn true if it is black's turn in the state, false if red's
   */
  private GameState(ArrayList<Cell> cells,  boolean blackTurn) {
     this.cells = cells;
     this.blackTurn = blackTurn;
  }
  /**
   * Removes a piece from the game board
   * @param piece_index the index on the board of the piece to remove
   * @return a new GameState with the piece at the given index removed
   */
  private GameState removePiece(int piece_index) {
     GameState newState = new GameState();
     newState.cells = new ArrayList<>(this.cells); //fixed
     newState.blackTurn = this.blackTurn;        //fixed
     newState.cells.set(piece_index, Cell.EMPTY);
     return newState;
  }
  /**
   * Moves a piece and changes the turn to the other color
   * @param moved_cell_start_index the initial index of the piece to move
   * @param moved_cell_end_index the index to move the piece to
   * @return Updated GameState
   */
```

```
 private GameState movePieceFlipTurn(
      int moved_cell_start_index,
      int moved_cell_end_index
   ) {
      GameState newState = new GameState();
      newState.cells = new ArrayList<>(this.cells);

      Cell moved_cell = this.cells.get(moved_cell_start_index);
      newState.cells.set(moved_cell_start_index, Cell.EMPTY);
      newState.cells.set(moved_cell_end_index,
moved_cell.promoteIfReachedEnd(moved_cell_end_index));
      newState.blackTurn = !this.blackTurn; // fixed
      return newState;
   }
   /**
    * Get the row index of a Cell
    * @param cell The cell represented as [A-H][1-8]
    * @return the row index (0-7)
    */
   private int rowIndex(String cell) {
      if (cell.length() < 2) {
         return -1;
      }
      return cell.charAt(1) - '1';
   }
   /**
    * Get the col index of a Cell
    * @param cell The cell represented as [A-H][1-8]
    * @return the col index (0-7)
    */
   private int colIndex(String cell) {
      if (cell.length() < 2) {
         return -1;
      }
      return Character.toUpperCase(cell.charAt(0)) - 'A'; //fixed even though it is not a bug
   }
   /**
    * Populate the array of next possible moves for a piece
    * @param start_index The initial index of the piece
    * @param curr_index The curr index (as we are searching possible moves)
    * @param nextMoves The array to populate the moves in
    */
   private void populateNextMoves(
         int start_index,
         int curr_index,
         ArrayList<GameState> nextMoves
   ) {

      Cell start_cell = cells.get(start_index);
```

```
    for (int move : start_cell.movement()) {
        int next_index = curr_index + move;
        if (next_index < 0 || next_index >= cells.size()) {
            continue;                                   //fixed:new if statement added
        }

        if (cells.get(next_index).isEmpty()) {
            nextMoves.add(this.movePieceFlipTurn(start_index, next_index));
        } else if (start_cell.isOpponent(cells.get(next_index))) {

          next_index += move;

            if (next_index < 0 || next_index >= cells.size()) {
                continue;                                   //fixed: new if statement added
            }

            nextMoves.add(this.movePieceFlipTurn(start_index, next_index));
            this.populateNextMoves(start_index, next_index, nextMoves);
        }
    }
}
/**
 * Gets all the possible moves of a piece located at a cell
 * @param cell The cell represented as [A-H][1-8]
 * @return The list of moves
 */
public ArrayList<GameState> nextMovesFromCell(String cell) {
    int r = rowIndex(cell);
    int c = colIndex(cell);
    if (r < 0 || c < 0 || r >= ROWS || c >= COLS) {
        return new ArrayList<>();
    }
    int start_index = r * COLS + c;
    ArrayList<GameState> nextGameStates = new ArrayList<>();
    populateNextMoves(start_index, start_index, nextGameStates);
    return nextGameStates;
}
/**
 * Gets a string representation of the board
 * @return String of the board
 */
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    int horizontalSize = COLS * 2 - 1;
    sb.append("   A B C D E F G H\n");
    sb.append("  ┌").append("─".repeat(horizontalSize)).append("┐\n");
    for (int i = 0; i < ROWS; i++) {
        sb.append(i + 1).append(" │");
```

```
            for (int j = 0; j < COLS; j++) {
                sb.append(cells.get(i * COLS + j).toString()).append(" │ ");
            }
            sb.append("\n");
        }
        sb.append("  └").append("─".repeat(horizontalSize)).append("┘ \n");
        return sb.toString();
    }
    /**
     * Gets a GameState represented in a file, also given whose turn it is
     * @param filepath Path of the file containg the board
     * @param blackTurn true if it is black's turn, false if red's turn
     * @return the GameState
     * @throws FileNotFoundException Board file not found
     * @throws NoSuchElementException Thrown if error parsing board
     * @throws IllegalArgumentException Thrown if error parsing board
     */
    public static GameState fromFile(
        String filepath,
        boolean blackTurn
    ) throws FileNotFoundException, NoSuchElementException, IllegalArgumentException {
        File file = new File(filepath);
        Scanner reader = new Scanner(file);
        ArrayList<Cell> cells = new ArrayList<>();
        for (int i = 0; i < ROWS; i++) {
            String line = reader.nextLine();
            for (int j = 0; j < COLS; j++) {
                char data = line.charAt(j);
                switch (data) {
                    case 'r':
                        cells.add(Cell.RED);
                        break;
                    case 'b':
                        cells.add(Cell.BLACK);
                        break;
                    case 'R':
                        cells.add(Cell.RED_KING);
                        break;
                    case 'B':
                        cells.add(Cell.BLACK_KING);
                        break;
                    case '-':
                        cells.add(Cell.EMPTY);
                        break;
                    default:
                        throw new IllegalArgumentException("Invalid file format");
                }
            }
        }
```

```
        return new GameState(cells, blackTurn);
    }
    /**
     * Gets the current turn
     * @return true if it is black's turn, false if red's turn
     */
    public boolean isBlackTurn() {
        return this.blackTurn;
    }
    /**
     * Gets the board of the GameState
     * @return List of cells in the board
     */
    public ArrayList<Cell> getCells() {
        return this.cells;
    }
}
```

Main Class:

```
import java.io.FileNotFoundException;
public class Main {
    public static void main(String[] args) throws FileNotFoundException {

            //I added an error checking
            if (args.length < 3) {
                System.out.println("Usage: java Main <file> <cell> <turn>");
                return;
            }

        String stateFileName = args[0];
        String cell = args[1];
        boolean isBlackTurn = args[2].equals("1");
        GameState state = GameState.fromFile(stateFileName, isBlackTurn);
        for (GameState nextState : state.nextMovesFromCell(cell)) {
            System.out.println(nextState);
        }
    }
}
```

CellTest Class:

```
/**
* Tests for the Cell enum in the checkers game.
* Author: Michele Onton
* Date: 10-20-2025
*/
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class CellTest {
        /**
   * Tests that red and black pieces are correctly identified as their respective colors,
   * and that empty cells return false for both color checks.
   */

  @Test
  void testColorChecks() {

     assertTrue(Cell.RED.isRedPiece());
     assertTrue(Cell.BLACK.isBlackPiece());
     assertFalse(Cell.EMPTY.isRedPiece());
     assertFalse(Cell.EMPTY.isBlackPiece());

  }
  /**
   * Tests that red and black (including kings) are identified as opponents
   * and that same-color cells are not considered opponents.
   */

  @Test
  void testOpponentLogic() {

     assertTrue(Cell.RED.isOpponent(Cell.BLACK));
     assertTrue(Cell.RED_KING.isOpponent(Cell.BLACK_KING));
     assertFalse(Cell.RED.isOpponent(Cell.RED));
     assertFalse(Cell.BLACK.isOpponent(Cell.BLACK_KING));

  }
```

```
/**
   * Tests that promotion occurs when red or black pieces reach their respective end rows,
   * and that no promotion occurs for pieces in the middle of the board.
   */

  @Test
  void testPromotion() {

      assertEquals(Cell.RED_KING, Cell.RED.promoteIfReachedEnd(56));
      assertEquals(Cell.BLACK_KING, Cell.BLACK.promoteIfReachedEnd(0));
      assertEquals(Cell.RED, Cell.RED.promoteIfReachedEnd(20));

  }
  /**
   * Empty cells are correctly identified,
   * and verifies symbol representations for red and black pieces.
   */

  @Test
  void testIsEmptyAndToString() {

      assertTrue(Cell.EMPTY.isEmpty());
      assertEquals("●", Cell.RED.toString());

  }
}
```

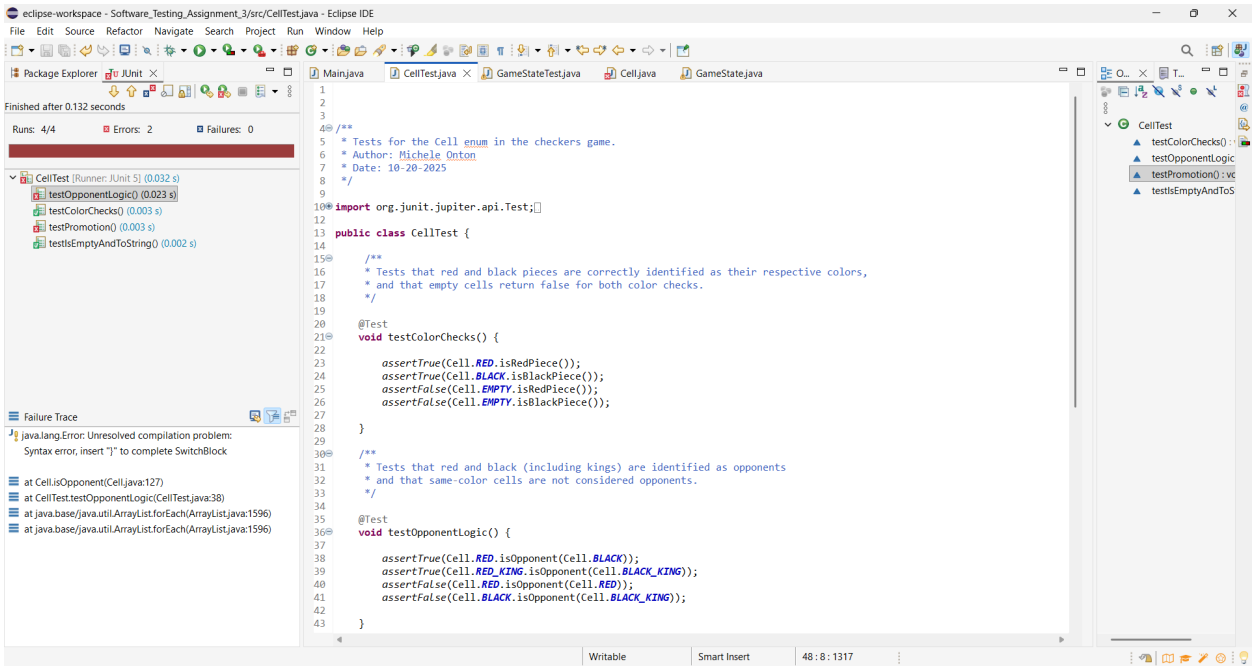
GameStateTest Class:

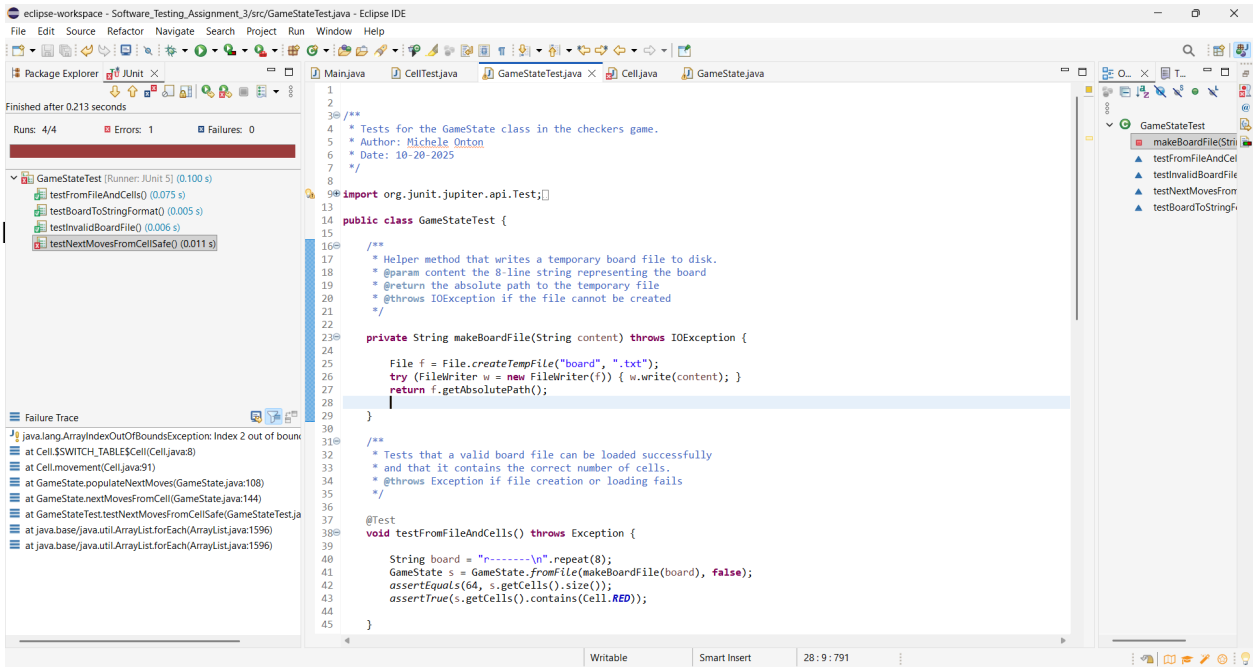
```
/**
* Tests for the GameState class in the checkers game.
* Author: Michele Onton
* Date: 10-20-2025
*/
import org.junit.jupiter.api.Test;
import java.io.*;
import java.util.*;
import static org.junit.jupiter.api.Assertions.*;
public class GameStateTest {
        /**
   * Helper method that writes a temporary board file to disk.
   * @param content the 8-line string representing the board
   * @return the absolute path to the temporary file
   * @throws IOException if the file cannot be created
   */

  private String makeBoardFile(String content) throws IOException {
```

```
    File f = File.createTempFile("board", ".txt");
    try (FileWriter w = new FileWriter(f)) { w.write(content); }
    return f.getAbsolutePath();

}
/**
 * Tests that a valid board file can be loaded successfully
 * and that it contains the correct number of cells.
 * @throws Exception if file creation or loading fails
 */

@Test
void testFromFileAndCells() throws Exception {

    String board = "r-------\n".repeat(8);
    GameState s = GameState.fromFile(makeBoardFile(board), false);
    assertEquals(64, s.getCells().size());
    assertTrue(s.getCells().contains(Cell.RED));

}
/**
 * Tests that a file with invalid characters throws an IllegalArgumentException.
 * @throws Exception if file creation fails
 */

@Test
void testInvalidBoardFile() throws Exception {

    String bad = "x-------\n".repeat(8);
    assertThrows(IllegalArgumentException.class,
            () -> GameState.fromFile(makeBoardFile(bad), true));

}
/**
 * Tests that nextMovesFromCell executes safely even if there are no valid moves.
 * This ensures no crashes or null returns from the method.
 * @throws Exception if file creation or loading fails
 */

@Test
void testNextMovesFromCellSafe() throws Exception {

    String board =
            "--------\n" +
            "--------\n" +
            "--------\n" +
            "---r----\n" + // red in the middle (D4)
            "--------\n" +
            "--------\n" +
```

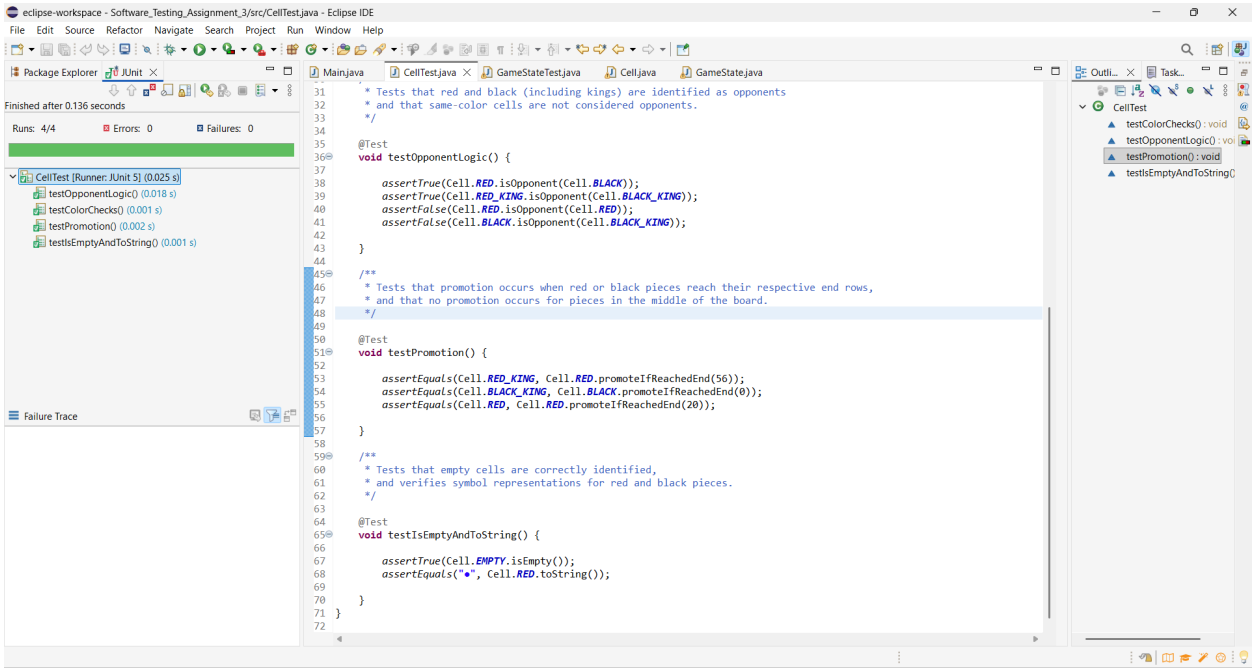
```
                "--------\n" +
                "--------\n";
        GameState s = GameState.fromFile(makeBoardFile(board), false);
        var moves = s.nextMovesFromCell("D4");
        assertNotNull(moves);

    }
    /**
     * Tests that the toString method formats the board correctly
     * and includes expected headers such as column labels.
     * @throws Exception if file creation or loading fails
     */

    @Test
    void testBoardToStringFormat() throws Exception {

        String board = "r-------\n".repeat(8);
        GameState s = GameState.fromFile(makeBoardFile(board), true);
        String str = s.toString();
        assertTrue(str.contains("A B C D E F G H"));

    }
}
```

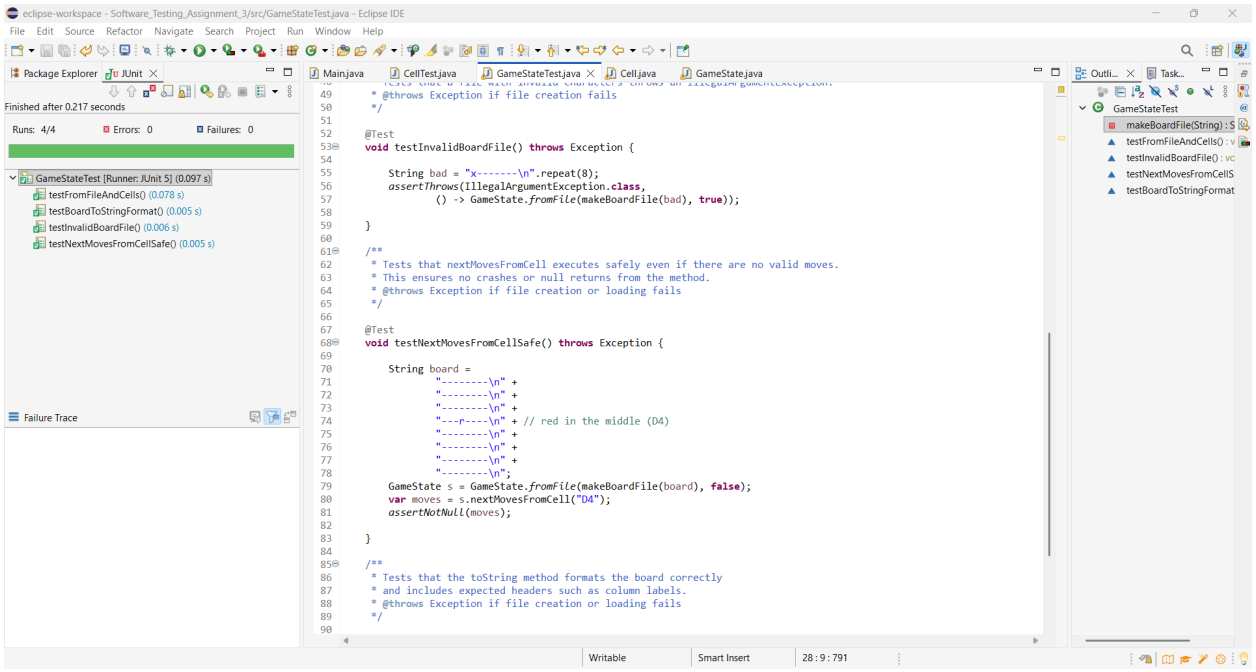
Failed Tests:



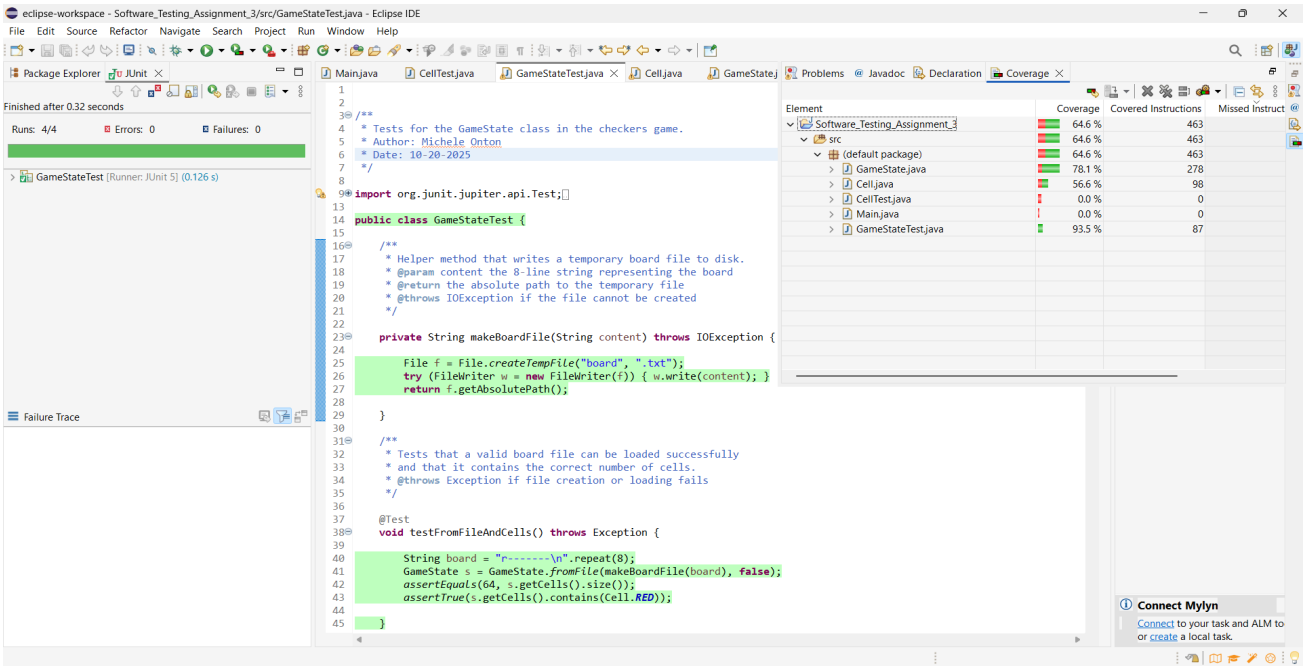


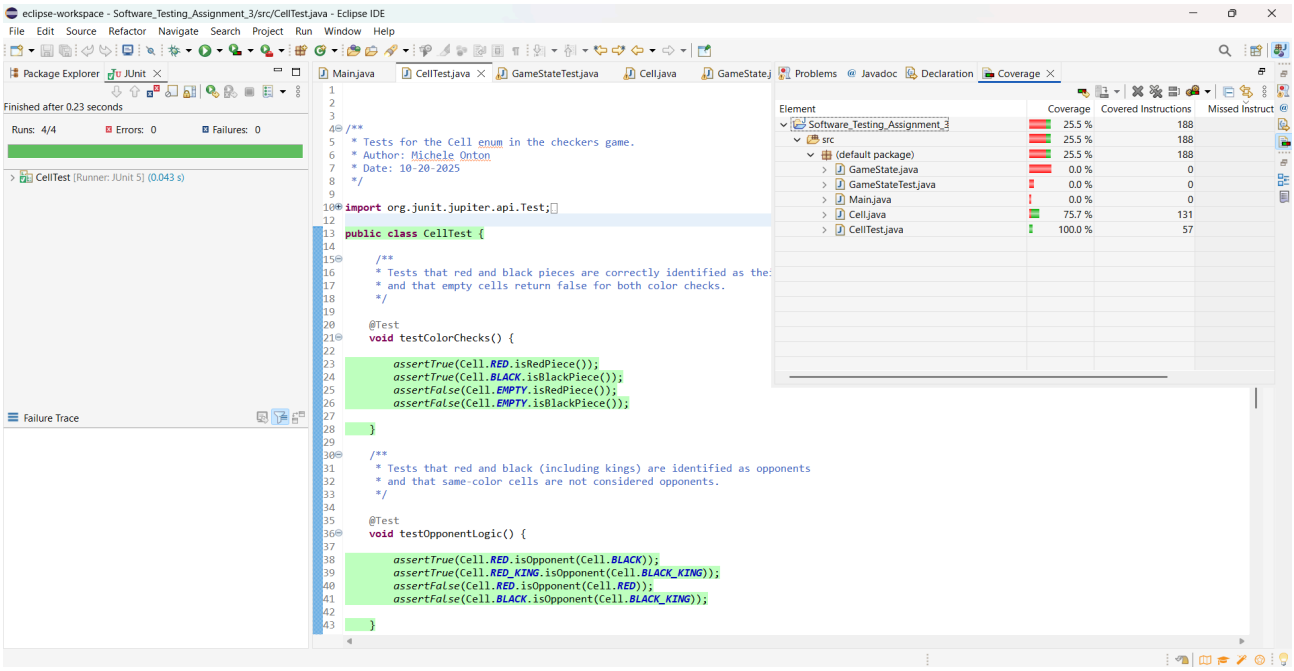
Passed Tests:





Coverage:





Description:

1. Red Piece Moves in Wrong Direction

Both RED and BLACK pieces use RIGHT_UP and LEFT_UP in their *movement()* method. In checkers, red pieces should move downwards while black pieces move upwards. This will cause red pieces to move backward instead of forward.

2. Promotion Row Boundaries Incorrect
The *promoteIfReachedEnd()* method uses exclusive inequalities (< and >), so pieces on the first or last cell of the promotion row will not promote to a king. The promotion logic should use inclusive bounds (>= and <) to cover the entire row.

3. Original GameState Modified in removePiece()
RemovePiece() sets newState.cells = this.cells and then modifies newState.cells. This mutates the original board, so all previous states sharing the same cells list will also see the change. States are no longer independent.

4. Opponent Detection Ignores King Pieces
isOpponent() only checks RED vs BLACK and ignores RED_KING and BLACK_KING. This causes jumps and captures against kings to be ignored, breaking normal game rules.

5. No Bounds Check in populateNextMoves()
When computing next_index = curr_index + move, there is no check if next_index is outside 0–63. Accessing cells.get(next_index) may throw an IndexOutOfBoundsException when the move would go off the board.

6. Jump Logic Does Not Remove Captured Pieces
When a piece jumps over an opponent, *populateNextMoves()* just moves the piece two steps forward without removing the jumped piece. This allows illegal jumps where the opponent piece remains on the board.

7. Turn Not Flipped After Move
movePieceFlipTurn() sets newState.blackTurn = this.blackTurn. The turn never switches, so the same player will always move repeatedly.

8. Column Index Calculation Case-Sensitive
colIndex() uses cell.charAt(0) - 'A'. If the user inputs lowercase columns like "c3", it will calculate the wrong column index. It should normalize the letter to uppercase first.

Overall Experience:

Working on this checkers project has been a valuable experience in designing, debugging, and testing object-oriented code. Implementing the “Cell” and “GameState” classes helped me understand the importance of proper management and recursive logic for generating possible moves. Debugging movement directions, promotion rules, and jump logic reinforced careful attention to edge cases and game rules enforcement. My tests have achieved a code coverage of 64.6%, covering most core functionality such as piece movement, promotion, and basic move generation, as well as file input parsing which I think was one of the most complicated parts of

the code and board setup. However, areas like multi-jump sequences, edge-of-board moves, captured piece removal, and turn flipping remain partially tested, highlighting the need for additional targeted tests to ensure all game rules are correctly enforced.

URL: https://github.com/MicheleOnton/Assignment3B-Testing-MicheleOnton.git