Assignment Sprint 4 - User Stories and Design Rationale

Note: For Additional Documentation (Java versions, Generation and running of .jar file, etc.), refer to the README.md file in gitlab.

Chosen Advanced Requirements

1. Requirement a: A tutorial mode, and a hint button for each player to toggle hints that show all of the legal moves the player may make as their next move
2. Requirement c: A single player can play against the computer, where the computer will randomly play a move among all of the currently valid moves for the computer

Revised User Stories For Advanced Requirements

Req 1A - Tutorial Mode

1. As a game,
   I want to have a tutorial mode,
   So that I can teach the new players the rules of the game

2. As a beginner game player
   I want an option to play a tutorial game
   So that I can learn how to play the game.

3. As a beginner game player,
   I want the tutorial game to have a guide for game instructions,
   So that I can refer to them to know the game rules and mechanics.

4. As a game player,
   I want to see all legal moves highlights on the gameboard of a tutorial game,
   So that I can fully understand the options that are present at every move.

Req 1B - Hints

5. As a game player
   I want to have a hint option
   So that I display all my next legal moves on the board.

6. As a game player
   I want to get a hint that suggest the first legal move to play
   So that I can understand which piece to select and start my move.

7. As a game player
   I want to disable the hint option
   So that I can play the game on my own without help or suggestions

8. As a game player
   I want a button to toggle the hint function
   So that I can easily access the hints when I need them.

9. As a game player
   I want the hint to turn off automatically when I choose a new piece or make a valid move
   So that I may depend on the hint only when necessary and concentrate more on my own judgements

10. As a game player
    I want the hints to be visually highlighted
    So that I can easily identify all the legal moves available, in a clear and concise way.

Req 2 - Player vs Computer

11. As a computer
    I want to know all of the currently valid moves
    So that I may choose one move randomly to play

12. As a game player
    I want to compete against a computer
    So that I may hone my abilities and practise new ones

13. As a game,
    I want to include both player versus computer and player versus player modes in the main menu,
    So that player can choose the mode they want.

14. As a game player
    I want to play against the computer
    So that I can play 9MM even when there isn't another person available.

Design Rationale

Revised Class Diagram

Player Class (refer to Appendix 1.1)

The revised architecture for the *Player* class includes a new method *isComputer()*, that is introduced to identify if the player is a human or computer. In *ComputerPlayer*, this method returns true. It helps other classes, such as *GameManager*, to quickly identify the player type, ensuring proper handling of actions.

GameBoard Class (refer to Appendix 1.2)

The revised architecture for the *GameBoard* class includes a new method *getPosition()*, it accepts an integer as parameter and is responsible for returning an object of the Position class for that integer. It is introduced to aid in the process of highlighting a piece when a user wants to slide or jump. Therefore, it is used by the *handleSlide()* and *handleJump()* method of the *GameManager*.

GamePanel Class (refer to Appendix 1.3)

We have added a new feature in our game which highlights the piece that is currently selected by the player. We have introduced a new boolean variable, *pieceHighlight* to indicate if the player selected a valid piece and a new variable, *highlightPosition* that stores the position of the selected. When the player selects a piece on the game board, the *highlightPieceToMove()* method in *GamePanel* is called by *GameManager* to change the boolean to true and the value stored in *highlightPosition*. The game panel will then be repainted to highlight the selected piece by calling the *highlightPieceOnBoard()* method which calls the *highlight()* method which draws a circle based on the given position, colour and size.

Moreover, we have added a hint button in the game panel. When the player clicks the button, all the legal moves will be drawn on the game board. The *showHint()* method in *GameManager* will call the *displayHint()* method to change *showHint* to true and the value stored in *legalMoves*. The boolean, *showHint* is introduced to indicate if the button is currently enabled. The variable, *legalMoves* stores all the legal moves the player can make as their next move. The game panel will then be repainted to draw the hints on the game board by calling the *drawHintOnBoard()* method based on the *legalMoves*. The *drawHintOnBoard()* method highlights all the hints by calling the *highlight()* method.

Additionally, we have added a method, *disableShowHint()* which will disable the hint button by removing the hints displayed on the game board. To achieve this, we have made the *buttonPanel* to be an attribute because we need to access it outside of the *addButtons()* method, in the *disableShowHint()* method.

GameManager Class (refer to Appendix 1.4)

The revised architecture for position class includes a boolean variable *isPositionSelected*, its purpose is to indicate, when game state is slide or jump, if a piece has been selected to be moved on the board. Therefore, it is used by the handleSlide*()* and *handleJump()* method of the *GameManager*.

Moreover, a pivotal function of advance requirement 1 is introduced in the *GameManager* which is *showHint()*. The function is responsible for generating all legal moves when a player requests a hint for a selected piece. It does this by calling the performAction*()* function, of the *HintAction* class, that finds all legal moves a player can make based on the current game state. This method is called when an *ActionEvent* occurs on the hint button.

The code becomes more modular by isolating player action logic into its own function. Each function has a purpose, making code easier to read, comprehend, and maintain. Each method is responsible for one task. The codebase can reuse the *handlePlayerAction* method. Events other than mouse clicks can directly call the *handlePlayerAction* method without repeating the code in the *processClick* method. The *handlePlayerAction* method handles game state and clicked position decisions. It bridges user input (*processClick*) with action methods (*handlePlace*, *handleSlide*, etc.). The *processClick* method is simplified by this encapsulation. Overall, a *handlePlayerAction* method improves code organisation, reusability, encapsulation, and testability. It enhances code organisation and maintainability while keeping each method focused on its goal.

Architecture for the Advanced Requirements

Req 1a - Tutorial Mode

For the tutorial mode, we have reutilized the classes *Display*, *GamePanel* and *GameManager*. We have added a new JPanel, *TutorialGuide*, which consists of all the tutorial cards for displaying the game rules and illustrating the mechanics. This provides guidance to the players in understanding how to play the game. The swapping of *MainMenuPanel* and *TutorialGuide* is managed by the existing *Display*, which is responsible for managing all the panels. Moreover, the tutorial mode guides the player how to play the game by displaying all the legal moves they can make and playing with the computer player.

Why did we revise the architecture?

The architecture is revised because there were some slight communication issues between the *GameManager* and *GamePanel* when mutating game state to create scenarios. Furthermore, we cannot manually place pieces by calling *handlePlace()* as it violates the principle of encapsulation and the game state might not be updated correctly.

How easy was it to implement?

The implementation of the tutorial mode is trickier than we expected. We have to add the new panel, *TutorialGuide* as the other game modes should not consist of the tutorial cards (refer Appendix 2.1).

Req 1b - Hints

A feature of Tutorial Mode is to display users their next legal moves, for this purpose we have introduced a *HintAction* class. This class is responsible for providing the players with all the legal moves they can make as their next move depending on the current game state.

The idea behind this design architecture is to follow the design principle of Separation Of Concern, as "hints" is an independent feature. Moreover, this helps prevent high coupling between modules and enhances the cohesion of each module.
Additionally, adding the legal moves generator logic to any other class would have violated the design principle of Single Responsibility Principle. Therefore, the class *HintAction* is justified and abides by all rules.

Why did we revise the architecture?
The *showHint()* function was added to the *GameManger* class to improve its usability. As previously stated, the *GameManger* class functions as a controller class between the frontend and the backend in previous sprints. Our goal was to display the next legal moves as the user clicked the hint button, in other words, to conduct a backend operation when the user interacted with the GUI. As a result, we had to revise the *GameManager* class to include the *showHint()* function. It would then call the *performAction()* method in *HintAction* to find out all legal moves a player can make.

The *GamePanel* class has been revised with new methods such as *displayHint()*, *drawHintOnBoard()*, and *highlight()*. Because the "hint" function includes highlighting legal moves to improve user experience, we needed to add certain methods to the class directly managing the GUI, in our case the *GamePanel* class, to highlight appropriate positions using different colours.

How easy it was to implement
Implementing the "Hints" feature was quite simple due to the solid design concepts followed throughout the sprints. The fact that we evaluated the distinction between actions allocated to a component and other types of actions. As a result, we knew we needed to create a *HintAction* class, and because it is a user action rather than a piece action, we made the class directly inherit the *Action* interface, adhering to the DIP Solid Principle, and because the interface already had a *performAction()* method, it was simple to override the function and add in the legal moves logic (refer to Appendix 3.1). Additionally, for every position on the game board we store its neighbouring positions; this pre-production helped us thoroughly in finding legal moves for a player (refer to Appendix 3.2).
Furthermore, our whole project is built on the Model View Controller (MVC) software design pattern, which allows us to easily distinguish between frontend and backend activities. In this scenario, creating the highlighting portion of the "hint" function was made easy by just adding a few methods to the *GamePanel* class (refer to Appendix 3.3).

<u>Req 2 - Computer Player</u>
A *ComputerPlayer* class allows you to add a player versus computer element to the game. When there are no other human opponents available, players can play versus the computer. This separation of concerns follows the Single Responsibility Principle (SRP) by guaranteeing that the *ComputerPlayer* class is in charge of controlling the computer player's moves, while the *GameManager class* is in charge of the overall game. It introduces a new gaming option and increases the game's flexibility. Playing against a computer opponent gives players a competitive experience. The *ComputerPlayer* class enables you to create an

opponent that can make smart moves based on rules that have been set, improving the overall gameplay. The *ComputerPlayer* class can help with game testing and debugging, as it enables you to test the game's execution and simulate many different scenarios, resulting in a more refined and reliable game.

The separation of move selection implementation from *GameMananger* follows the design principle of separation of concern. It ensures that any changes or improvements made to the move selection algorithm can be done in *ComputerPlayer* and will not impact the core game logic in *GameManager*. For example, we can implement a more advanced AI algorithm to select the legal move in the *computeNextMove()* method without modifying GameManager. This promotes easier code maintenance and reduces the chances of side effects on the game functionality.

By designing the architecture in this way, we ensure a clear separation of responsibilities and minimise the dependencies between the classes, and allow easier updates and extensions in the future.

Why did we revise the architecture?

We have revised the *GameManager* to handle actions for both human and computer players. Previously, it was only responsible for handling actions related to human players. We have introduced a new method, *handlePlayerAction()*, which is responsible for validating and handling the player actions based on the current state of the game. The logic for handling actions are similar regardless of whether the player is a human or computer. Human players make a move by clicking on the game board in GUI; whereas, the computer player selects a random move. Both inputs will be then passed to the *handlePlayerAction()* for appropriate handling. Since the player who makes the first move is always the human player in the Player vs Computer game mode, we have implemented a checking condition after the human player's move is handled in *processClick()*, which checks whether the current player is a computer. This allows appropriate handling of moves to the computer player.

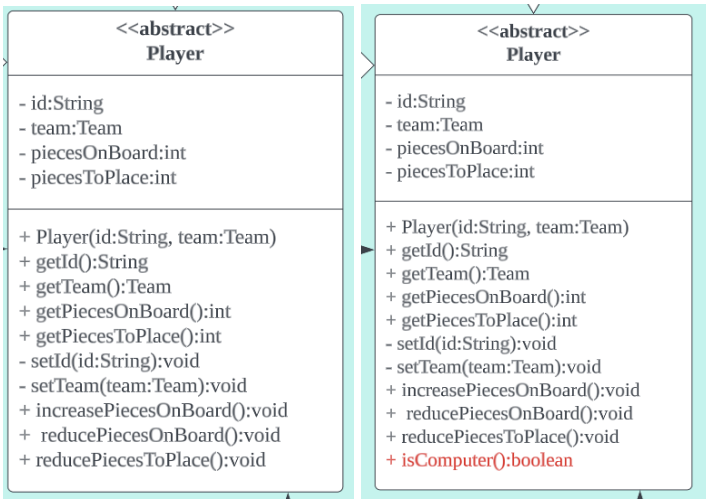
How easy it was to implement

The addition of the *ComputerPlayer* class to the existing code base involved minimal modifications to the overall structure and design. This was achieved by following SOLID principles and utilising an abstract *Player* class. By utilising inheritance, the code was able to be reused and the Open-Closed Principle was promoted (refer to Appendix 4.1). This was achieved by extending the capabilities of the abstract *Player* class without altering its current implementation. By adopting this approach, the code base became modular and maintainable, and at the same time, the desired computer player functionality was introduced.

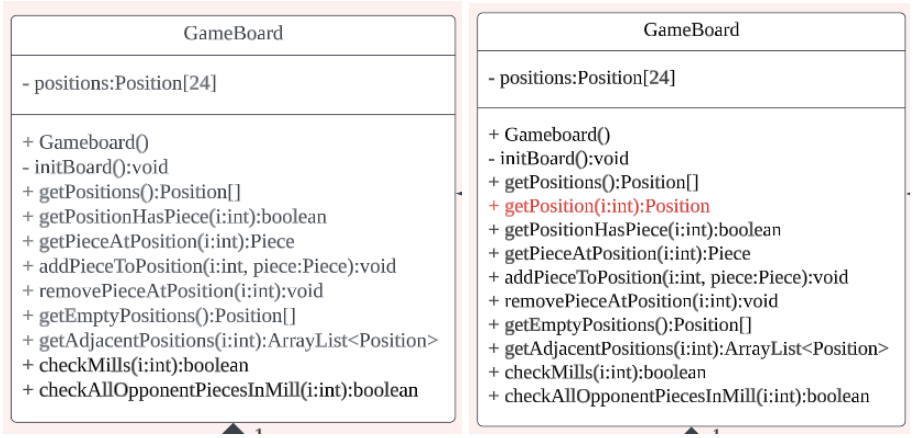
Appendix

Appendix 1: Revised Class Diagram

Appendix 1.1: Old Player Class (Left) vs New Player Class (Right)

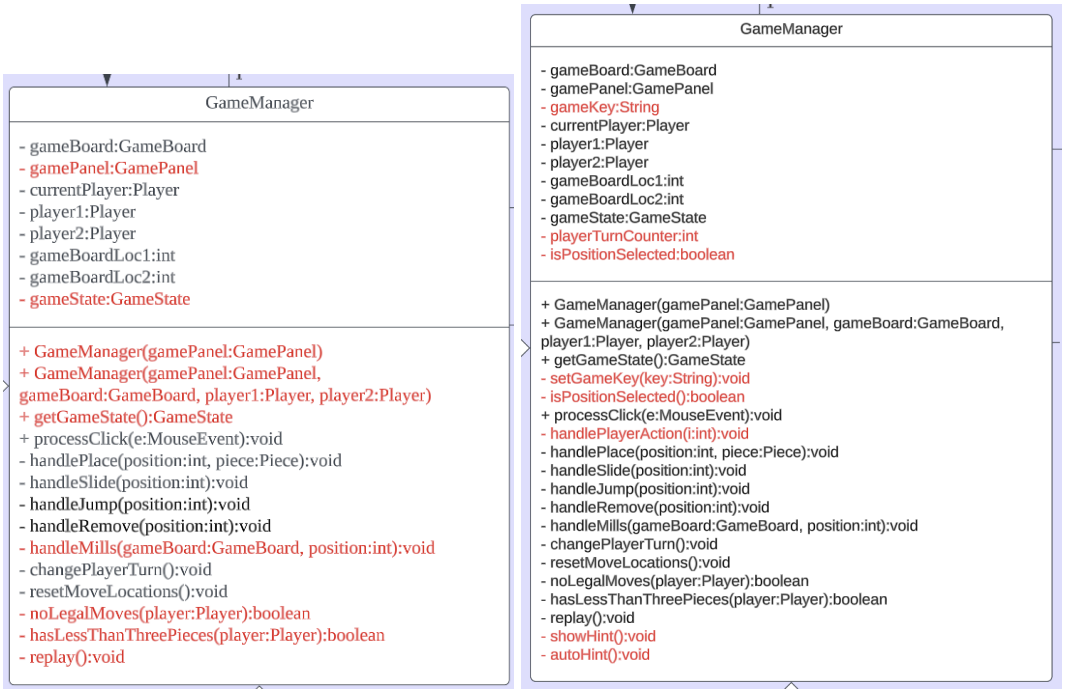


Appendix 1.2: Old GameBoard Class (Left) vs New GameBoard Class (Right)



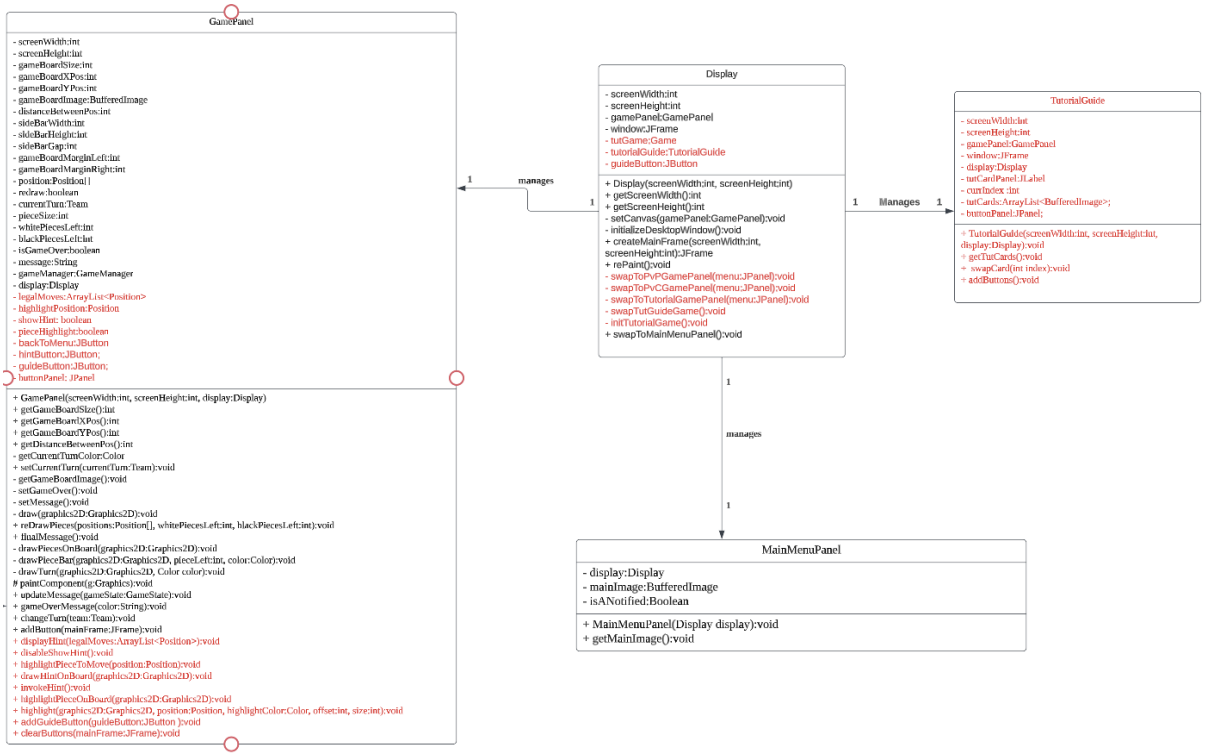
Appendix 1.3 Old GamePanel Class (Left) vs New GamePanel Class (Right)

Appendix 1.4 Old GameManager Class (Left) vs New GameManager Class (Right)



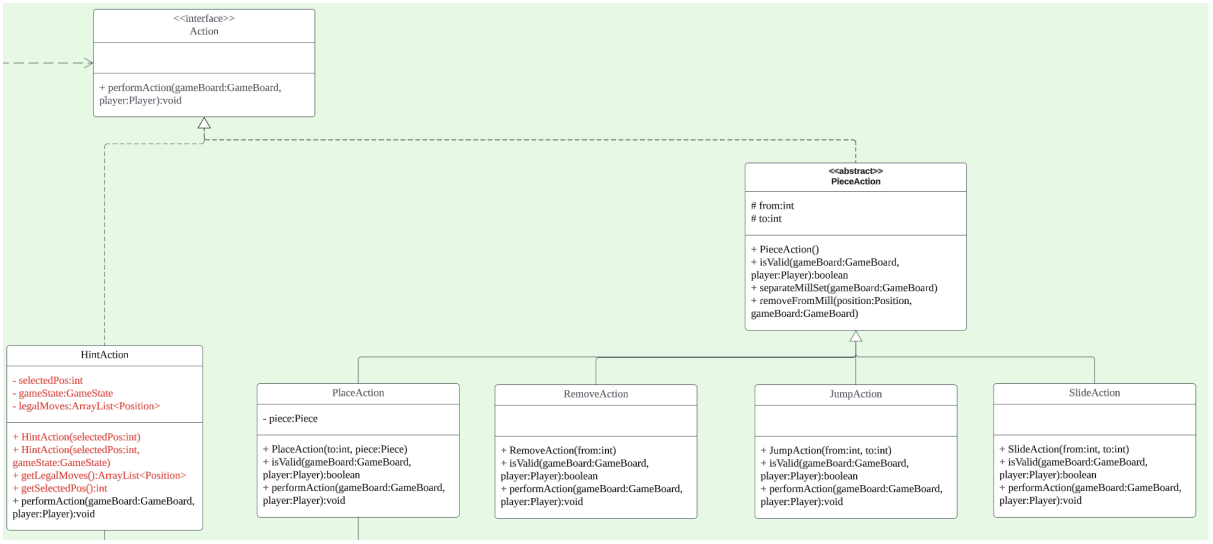
Appendix 2: Tutorial Mode

Appendix 2,1:



Appendix 3: Hint Button

Appendix 3.1: HintAction directly implement Action interface



Appendix 3.2A: Position stores its neighbours

```java
public class Position {

    // The id (unique)
    private int id;

    // The x coordinate
    private int x;

    // The y coordinate
    private int y;

    // The piece on this position
    private Piece piece;

    // The adjacency positions of this position
    private ArrayList<Position> neighbours = new ArrayList<>();
```

Appendix 3.2B: GameBoard initialises all the positions

```java
// The 24 positions of the game board
private Position[] positions = new Position[24];

// The 16 possible combinations of mills
public ArrayList<List<Position[]>> mills = new ArrayList<>();

// Constructor
/**
 * Creates a new GameBoard object.
 */
public GameBoard(){
    initBoard();
}

/**
 * Initialize all the 24 positions of the game board and all the 16 possible combinations of mills.
 */
private void initBoard() {
    // Add all 24 positions
    positions[0] = new Position(0,0,0);
    positions[1] = new Position(1,3,0);
    positions[2] = new Position(2,6,0);
    positions[3] = new Position(3,1,1);
```

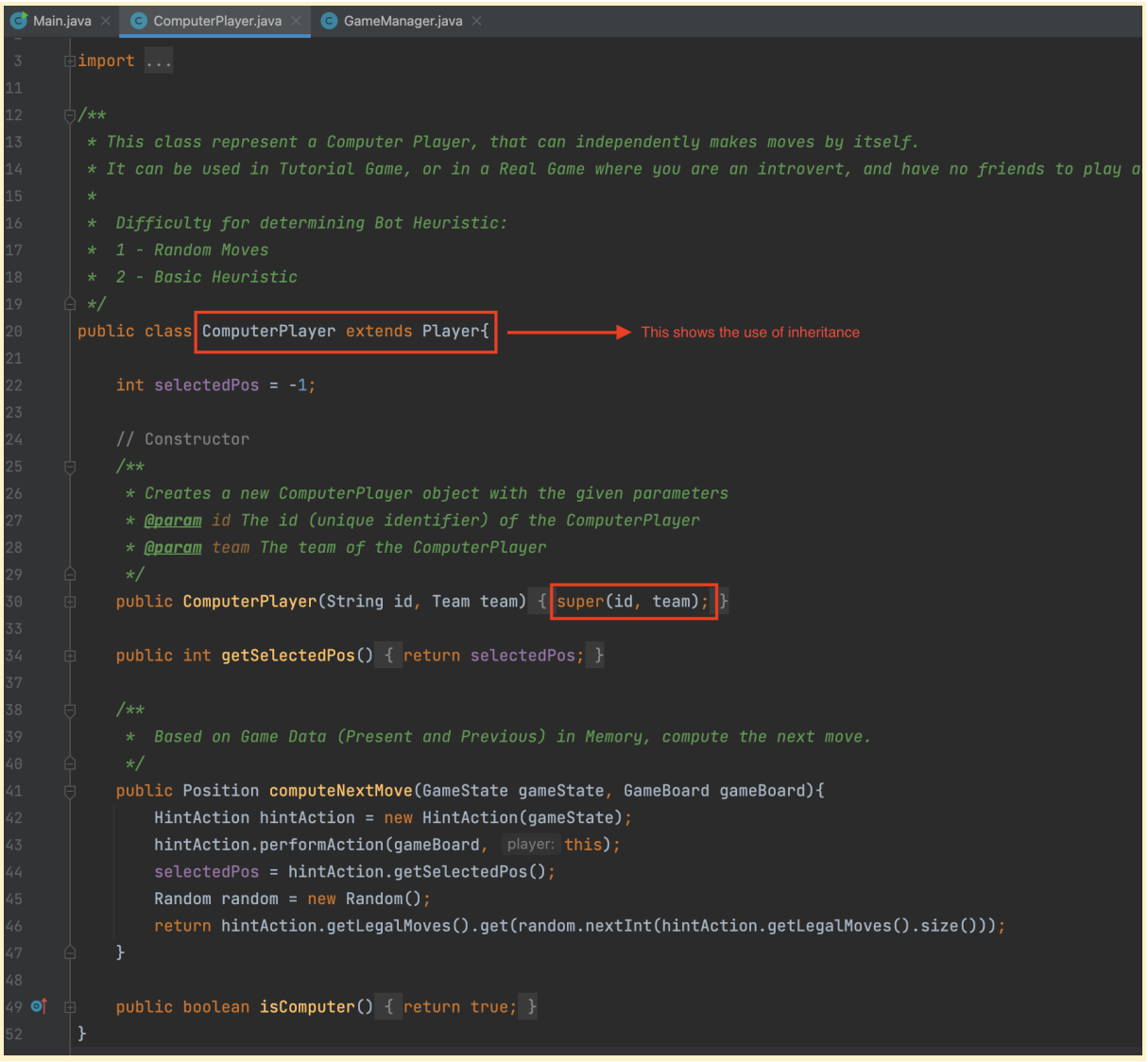
Appendix 3.2C: GameBoard adds the neighbours of each position

```java
    // Add all adjacent positions of each position
    positions[0].addNeighbour(positions[1]);
    positions[0].addNeighbour(positions[9]);
    positions[1].addNeighbour(positions[0]);
    positions[1].addNeighbour(positions[2]);
    positions[1].addNeighbour(positions[4]);
    positions[2].addNeighbour(positions[1]);
    positions[2].addNeighbour(positions[14]);
```

Appendix 3.3: New methods added to the GamePanel for highlighting the hints

Appendix 4: Computer Player

Appendix 4.1: Evidence of use of Open Closed Principle


Main.java
ComputerPlayer.java
GameManager.java
import ...
/**
* This class represent a Computer Player, that can independently makes moves by itself.
* It can be used in Tutorial Game, or in a Real Game where you are an introvert, and have no friends to play a
*
* Difficulty for determining Bot Heuristic:
* 1 - Random Moves
* 2 - Basic Heuristic
*/
public class ComputerPlayer extends Player{
This shows the use of inheritance
int selectedPos = -1;
// Constructor
/**
* Creates a new ComputerPlayer object with the given parameters
* @param id The id (unique identifier) of the ComputerPlayer
* @param team The team of the ComputerPlayer
*/
public ComputerPlayer(String id, Team team) { super(id, team); }
public int getSelectedPos() { return selectedPos; }
/**
* Based on Game Data (Present and Previous) in Memory, compute the next move.
*/
public Position computeNextMove(GameState gameState, GameBoard gameBoard){
HintAction hintAction = new HintAction(gameState);
hintAction.performAction(gameBoard, player: this);
selectedPos = hintAction.getSelectedPos();
Random random = new Random();
return hintAction.getLegalMoves().get(random.nextInt(hintAction.getLegalMoves().size()));
}
public boolean isComputer() { return true; }
}