MONASH University

FIT3077

SPRINT FOUR

Team Nine Bytes

SUBMITTED BY

Shyam Kamalesh Borkar (32801459)

Lai Carson (32238436)

Richardo Husni (32767269)

Victor Lua (31362834)

Sprint 4 Video Demonstration: Youtube Unlisted Link (mp4 file available through moodle)

Chosen Advanced Requirements

We are a team of 4 so we have chosen the following 2 advanced requirements from the available advanced requirements in the assignment brief to design and implement:

(a) Considering that visitors to the student talent exhibition may not necessarily be familiar with 9MM, a **tutorial mode** needs to be added to the game. Additionally, when playing a match, there should be an option for each player to toggle "**hints**" that show all of the legal moves the player may make as their next move.

(c) A single player may play against the **computer**, where the computer will randomly play a move among all of the currently valid moves for the computer, or any other set of heuristics of your choice.

Revised User Stories

Most of the user stories remain the same from Sprint 1 except for a few. A couple of user stories have been removed in comparison to those in Sprint 1 due to the user stories being repetitive as mentioned in the Sprint 1 feedback.

Basic Requirements

Player

1. As a player, I want to see a game board, so that I know where to place and move my pieces.
2. As a player, I want to place my pieces on the board so that I can have up to 9 pieces on the board.
3. As a player, I want to move placed pieces on the board so that I can try to form a mill, meaning 3 of my pieces on a straight line.
4. As a player, I want to be able to choose and remove one of my opponent's pieces from the board, once I make a mill.

5. As a player, I want to be able to fly my pieces (move to any unoccupied spot on the board) once I am down to three pieces, so I am not limited to moving my remaining pieces to adjacent points.

6. As a player, I want the game to notify me once either player has 2 pieces left, so I know there is a winner.

7. As a player, I would like to end a game prematurely and return to the main menu so that I do not have to complete a game.

Game Board

8. As a game board, I want to have 24 intersections that can each only be occupied by a single piece, so that both players have enough space to place and move their pieces around.

9. As a game board, I want to know if a player has moved their piece to an adjacent available intersection on the board, so that I know a player is making a valid move.

10. As a game board, I want to make sure that pieces from a formed mill cannot be removed unless no other pieces are available, so that game rules are followed.

11. As a game board, I want to make sure that captured pieces cannot be played again, so that the game rules are obeyed.

12. As a game board, I want to indicate which player's turn it is, so that the correct player can make their turn.

Piece

13. As a game piece, I want to move to legal positions on the board based on game rules, so that the game can progress.

14. As a game piece, I want to move back to my original intersection if the player moves me to an illegal intersection, so that the game rules are obeyed.

Game State

15. As a game state, I want to keep track of which phase the game is in now so that I can identify what actions the pieces can make.

16. As a game state, I want to make sure that both sides have 3 or more pieces so that each side has a chance to form a mill.

17. As a game state, I want to keep track of each player's pieces to determine which phase each player of the game is currently in.

Advanced requirement (a) - Tutorial and Hints

18. As a novice player, I want the game to show prompts and guides in tutorial mode, so that I can learn the game as a beginner and improve my skills.

19. As a game player, I want an interactive tutorial experience, so that I can learn the game rules and mechanics easily.

20. As a player who needs assistance in game, I want to be able to toggle hints, so that I can visualise all possible moves for a piece and make better game decisions.

Advanced requirement (c) - Computer

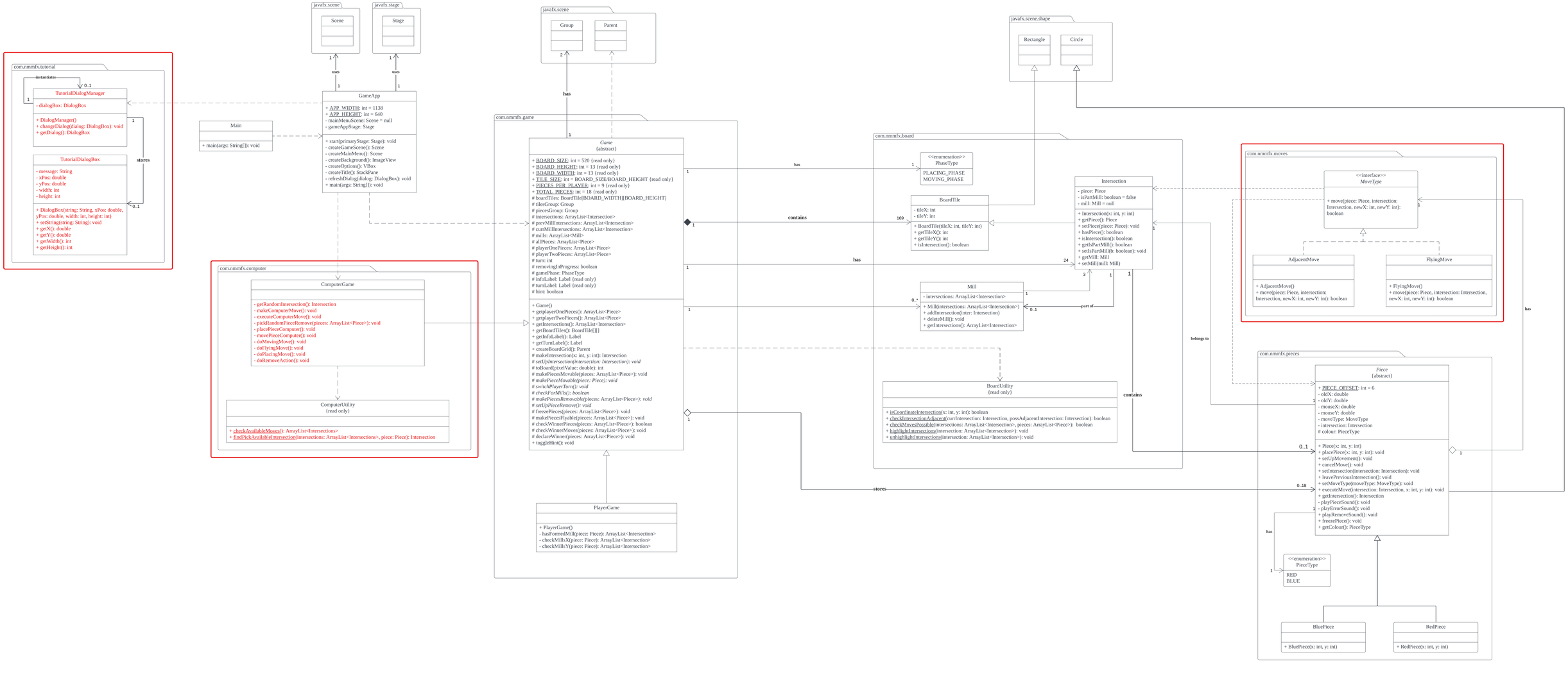
21. As a game player, I want to have an option to play against the computer, so that I can have someone to play with.

22. As a game computer player (bot), I want to be able to do the basic game interactions with randomness, so that I can play the game properly against a human player.

23. As a game computer player (bot), I want to delay my moves by a couple of seconds after the human player has made a move so that they have enough time to reciprocate the changes.

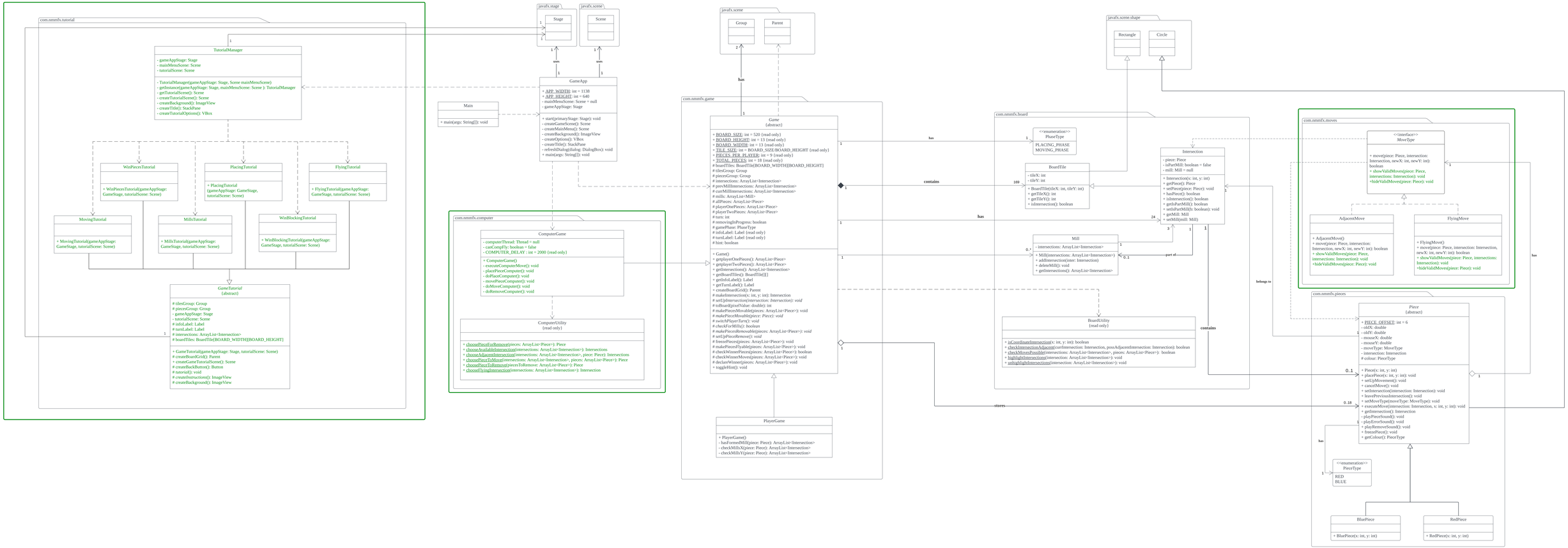
9 Men's Morris Sprint 3 UML Class Diagram

MA_Friday2PM_Team12



9 Men's Morris Sprint 4 Revised UML Class Diagram

MA_Friday2PM_Team12



Design Rationale

Revised UML Design Decisions

For this section, please refer to the UML class diagrams provided in this document. We have included both diagrams: the original UML class diagram from the prior sprint (Sprint 3) and the updated UML class diagram from the current sprint (Sprint 4), to make it easier to understand all the revisions made to the UML class diagram from the prior sprint. To show the changes, both diagrams have been colour-highlighted. A **green colour annotation** has been used for the new revised UML class diagram, indicating that something has been added or that the highlighted part is different in some way from the previous diagram.

A **red colour annotation** has been used for the previous class diagram to indicate that changes have been made to it in the revised diagram or that it has been removed from the new revised diagram. Our team's gitLab repository contains a revised UML class diagram for Sprint 4 in pdf format without any annotations. To further clarify the reasoning behind the revision and tie everything together, the changes and the decision behind them will be highlighted below using terms like methods and class names from the diagram.

Most of the changes that have been made to the UML class diagram are related to the advanced requirements in Sprint 4. The system design for the basic nine men’s morris game remains the same. Since most of the changes are regarding advanced requirements, the reasons behind the design will be discussed in more detail under their own dedicated sections later.

ComputerGame

The ComputerGame class methods have been changed to a minor extent. These method changes pertain to the actions of the computer when a human player plays against it, as can be seen from the two annotated diagrams. A couple of new variables have been added compared to the previous sprint, as these were the variables that were found to be needed during the implementation of the ComputerGame class. These variables are, for example, computerThread, which is a separate thread in the game in which delay can be introduced to give the effect that the computer takes time to make a decision, and COMPUTER_DELAY, which is a constant value indicating the delay in seconds the computer will take to respond.

ComputerUtility

It is the same class as before with added methods and changes in method names. During implementation, it was discovered that additional decision processes were needed for the computer to operate effectively and, as a result, these methods were added to the ComputerUtility static class.

More details about the design decisions behind the computer game can be found here.

MoveType, AdjacentMove and FlyingMove

MoveType is the interface that provides the required abstract methods for different types of moves. To introduce hints while following good design, new abstract methods were added to the MoveType interface related to the hints feature. As a result, classes like AdjacentMove and FlyingMove that implement the MoveType interface have adapted to these new changes in methods. There are only two newly added methods: *showValidMoves* and *hideValidMoves*. Design for the hints feature is discussed in detail here.

TutorialManger

The idea behind the previous tutorial design was to show static videos and cues to the user to teach them about the game and how to play. To make the tutorial more interactive and interesting, we abandoned that idea, changed the classes, and introduced a new design that will allow the user to interactively play with a board and learn about the different rules and mechanics of the team's nine men's morris game.

The TutorialManager class helps manage all the different types of tutorials for different game scenarios and provides access to these tutorials to the user through the user interface.

GameTutorial

GameTutorial is a new abstract class that is responsible for producing game-specific tutorials for the user. It provides the board logic and other UI aspects for use in the tutorial. Using this class, many different tutorial classes have been created by inheriting from *GameTutorial* to guide the user through the entire game from start to finish, explaining to the player the different game scenarios and giving the player hands-on experience. The classes that inherit from *GameTutorial* are *PlacingTutorial (a* tutorial for placing pieces), *MovingTutorial (a* tutorial for moving pieces), *MillsTutorial (a* tutorial for forming mills and removing pieces), *FlyingTutorial (a* tutorial for flying the pieces), *WinBlockingTutorial (a* tutorial for winning by blocking opponents), and finally *WinPiecesTutorial (a* tutorial for winning by removing opponents pieces).

The rationale behind the design of the Tutorial feature is further discussed here.

Advanced requirement (a) Design - Hints

Hints are visual indicators that show valid intersections a particular piece can move to by hovering over them. Each valid intersection is represented by an intersection highlighted in green. It can be toggled on and off (by pressing ‘Hints’ at the top right corner) at any point in the game except the placing phase, since any empty intersections are valid at this point.

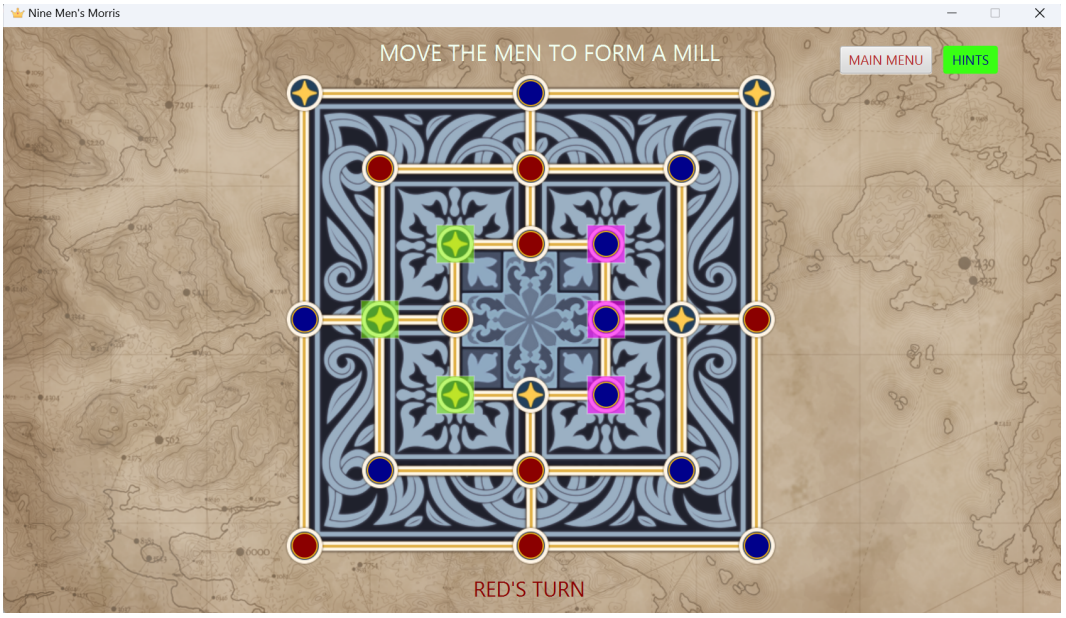


Figure: Hints (green intersections) being shown for a red piece. The cursor is hovering over the red piece.

1. **Hints Design 1**

   The first design concept for hints involved creating a new class to manage how hints were displayed. This class would decide which intersections would be highlighted based on the current state of the game. This initial design seemed obvious since the team was implementing a new feature, and the logic for this new feature should be encapsulated in a new class. This would adhere to the Single Responsibility Principle.

   A major drawback of this approach was that it needed a lot of information from other classes, especially the ‘Game’ class, in order to perform its functionalities. This led to our second design concept (see Design 2 below). An additional class may also introduce more dependencies and an additional layer of unnecessary complexity to our source code.

2. **Hints Design 2 (chosen)**

   Most of the information required to compute the hints was already within the ‘Game’ class. However, implementing hints in ‘Game’ would bloat the class even further. It did not make sense to add hints to ‘Game’ either, since it was a different set of responsibilities. Instead, the team noticed that the implementation of hints would be very similar to move validation done using the strategy pattern in the previous sprint. Hints are just visual indicators for valid moves for a particular piece, so the hints shown would have to change based on which phase a piece was in.

   The second design approach implements hints within each subclass of ‘MoveType’, which reuses the existing Strategy design pattern, since it already encapsulates the different logic of different phases (phase is ‘MoveType’ in our source code) in the game. Similar to how move validation is performed in our game, each piece is responsible for knowing its own phase. New abstract methods to show and hide hints are declared in ‘MoveType’, and overridden by the subclasses based on the rules of each phase. The ‘Game’ class does not need to know which phase the piece is in, and it only has to call the overridden methods of each piece to show hints. This approach also reuses some previously defined methods. The new methods that show or hide hints are invoked in methods that enable (allow them to move their pieces) and disable a player’s pieces respectively, so hints are only shown for the player that is taking their turn.

   This approach further enforces modularity and information hiding of the different game phases. No new classes or dependencies are created (as in Design 1) by reusing the old architecture, classes and methods, which minimises the complexity required to implement this new feature. This concept is also highly extensible. It adheres perfectly to the Open Closed Principle, since adding a new phase with new rules and hints only requires the addition of a new ‘MoveType’ subclass - no other classes need to be modified.

   The only drawback of this approach is that it violates the ‘Don’t Repeat Yourself’ rule within the same ‘MoveType’ subclass. Within the subclass, one method validates moves, while the new hints method validates the move again before showing hints. However, some refactoring can be done in order to isolate the movement validation method to make it a reusable component.

Hints Design 2 was chosen since it provided a lot more advantages with minimal drawbacks, and made both the source code and architecture ‘cleaner’ based on the reasons explained above.

Advanced requirement (a) Design - Tutorial

Tutorials are short, guided scenario-based games that teach the player about basic game rules. Players can select which part of the game they want to learn about based on the options in the tutorial menu. Each type of tutorial is presented with a typical game board on the left, and detailed explanations and instructions on the right that guide the player. Following through the instructions until the end completes a tutorial.

Figure: The tutorial menu, accessible from the game's main menu

Figure: The interface of a typical tutorial

1. **Tutorial Design 1**

   The initial design involved reusing the existing ‘ComputerGame’ code by having the tutorial play out as a normal game, with special pop-ups or notifications that guided a player as the game progressed. There would be conditional checks throughout to detect certain events (such as a phase change) in the game, and these events would serve as an indicator to display the pop-ups.

   Reusing existing code was an advantage since new features could be implemented without introducing much changes to our architecture. It is also a sign that our code was extensible and flexible to changes.

   This design had a few problems:

   a. Pop-ups are triggered by events. What happens if players get stuck and the events are never triggered?
   b. Players may lose interest since a game takes a while to conclude. It might take even longer for new players.
   c. If new rules are added into the game, more conditional blocks need to be added to detect new events. Eventually, the tutorial class would become convoluted and bloated with conditional blocks. These conditional blocks hint that they could be abstracted into their own class.

2. **Tutorial Design 2**

   The point (c) (explained in the previous section) was the main reason our team decided to rule out Option 1. To address this, game scenarios would be separated into different subclasses in this second design, inheriting common methods and attributes from a parent Tutorial class to abstract their commonalities. There will be a ‘TutorialManager’ class that sets up the UI for the tutorial menu, and creates a new game based on which tutorial is selected. From this point onwards, ‘TutorialManager’ would create the relevant subclass (to create the correct tutorial game), and the logic for each tutorial is contained within the subclass. This introduces a form of encapsulation and responsibility segregation to the tutorial component of our game.

   From the player’s perspective, this design would be more user friendly than Design 1, since a player has the option to select any game rule which they would like to learn about. These tutorial games are significantly shorter, but equally as informative too, since scenarios allow the player to jump to specific parts of the game, without the

need to play through an entire game. Players can also see all of the game rules in the tutorial menu at a glance, instead of relying on triggering events to discover them.

Unfortunately, this creates a dependency between each subclass with ‘TutorialManager’ since ‘TutorialManager’ is the class that instantiates the subclasses based on button presses. More game rules mean more tutorials, which may bloat the ‘TutorialManager’ class eventually. This is similar to the disadvantage discussed in Design 1, but to a lesser extent since creating a new object takes up less lines and is easier to understand than creating a new conditional block.

Tutorial Design 2 was chosen since it offers more advantages, and the disadvantage that comes with Design 2 is similar to Design 1, but to a lesser degree. On top of architectural reasons, the ‘Usability’ NFR also influenced the team’s decision, since it was an important aspect to consider in a highly user-interactive component like tutorial.

Advanced requirement (C) Design - Computer Game

In total there are only two classes/entities that are responsible for the computer game requirement in this project which are: ComputerGame and ComputerUtility.

ComputerGame

The *ComputerGame* class is the main class that houses all the functionality of the computer game requirements. This class inherits a lot of common game functionality from the abstract Game class. Due to this abstraction, most of the game functionality is already readily available, and not much was needed to be added to the *ComputerGame* class to satisfy the requirements of playing against a computer. The original plan was to have the *ComputerGame* to be a separate class on its own, thinking that the idea of playing with a computer would require completely different functionality. However, the team was able to see the commonalities between the ComputerGame and the normal *PlayerGame* and abstracted the common traits into the Game abstract class.

As a result of the abstraction, the design follows the DRY (do not repeat yourself) principle, meaning that there is no repeated code in the system. This class also follows the single responsibility principle, as it only has functionality for the game and executing computer moves and nothing else. The decision-making process is delegated to the other class, which is the *ComputerUtility* class.

The *ComputerGame* only has additional functionality that involves the computer executing moves depending on the different game phases and also involves using additional game threads to introduce delays when a computer makes its move.

ComputerUtility

The *ComputerUtility* class is a static class with static methods. The methods in this class just allow the computer in *ComputerGame* to make random game decisions while adhering to the game rules. For example, when the computer wants to decide which intersection to place its piece on in the placing phase, it will consult the *ComputerUtility* class to make the decision by passing the list of intersections on the board to the *ComputerUtility* class. The utility class then iterates through the list of intersections, producing a list of empty intersections (without a piece on them) and randomly returning one of the empty intersections as output for the computer to place its piece on.

The *ComputerUtility* class also follows the single responsibility principle, as it only helps the computer make game decisions, while the *ComputerGame* class is tasked with executing the chosen game decisions.