MONASH University

FIT3077

SPRINT THREE

Team Nine Bytes

SUBMITTED BY

Shyam Kamalesh Borkar (32801459)

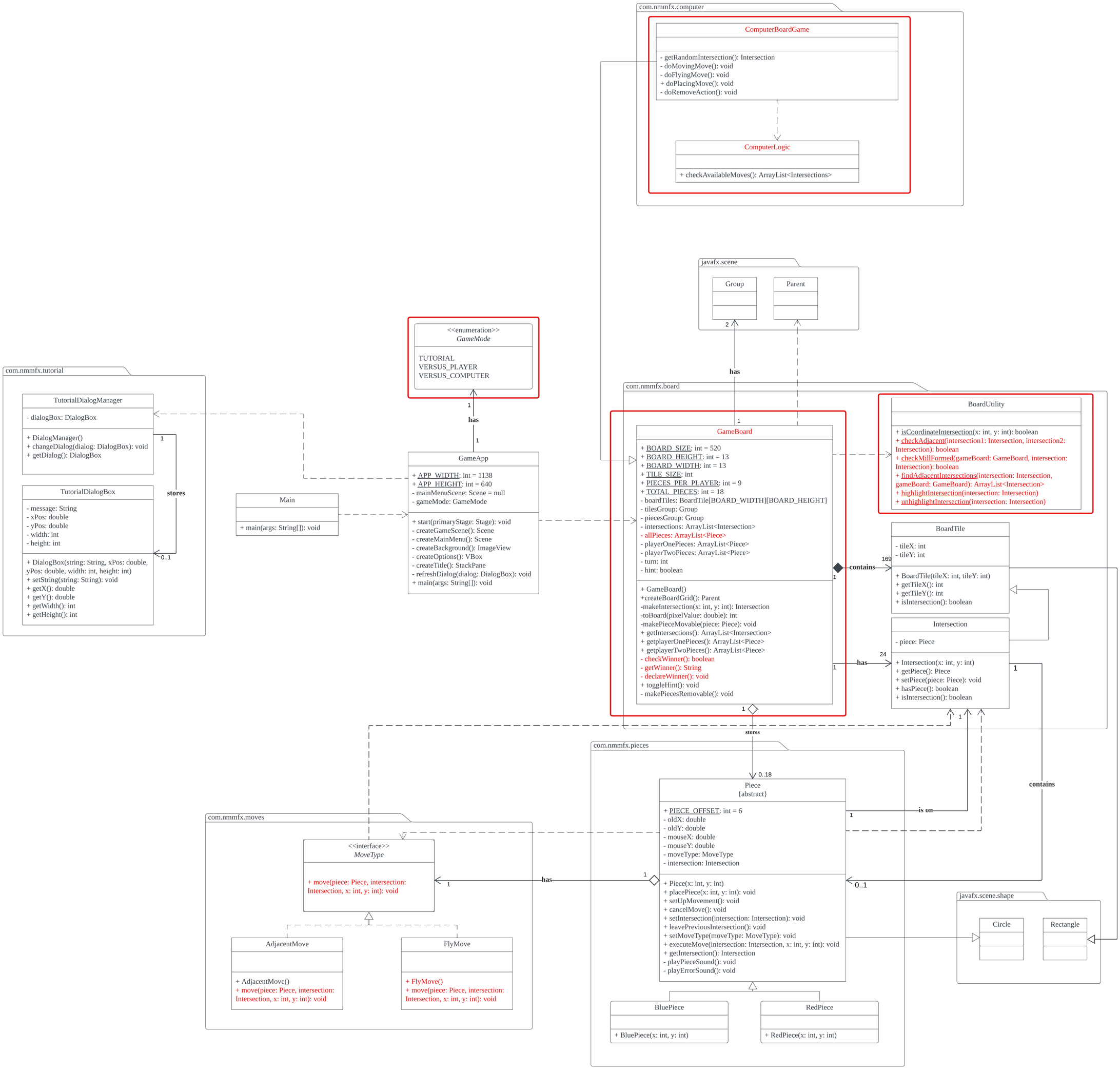
Lai Carson (32238436)

Richardo Husni (32767269)

Victor Lua (31362834)

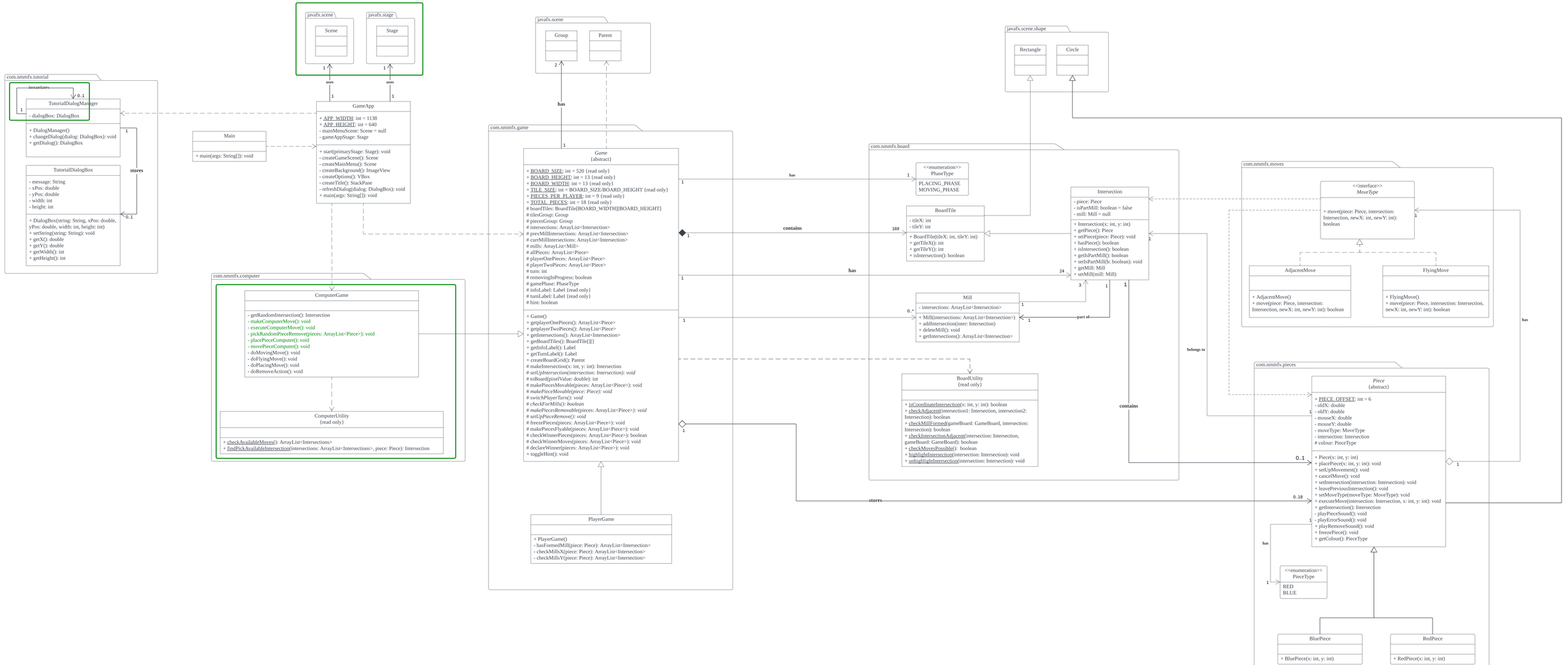
9 Men's Morris Sprint 2 UML Model

MA_Friday2PM_Team12



9 Men's Morris Sprint 3 Revised UML Class Diagram

MA_Friday2PM_Team12



Sprint 3 Video Demonstration: Youtube Unlisted Link (mp4 file available through moodle)

Design Rationale

Revised UML Design Decisions

Please refer to the provided UML class diagrams in this document for this section. For the simplicity of understanding all the revisions made to the UML class diagram from the previous sprint, we have included both of the diagrams: the UML class diagram from the previous sprint (Sprint 2) and the revised UML class diagram from the current sprint (Sprint 3). Both diagrams have been highlighted/annotated with colour to indicate the changes made. A **red colour annotation** has been used for the previous class diagram, indicating that changes have been made to it in the revised diagram or it has been removed from the new revised diagram, and a **green colour annotation** has been used for the new revised UML class diagram, indicating that something has been newly added or that highlighted part is different in some way compared to the previous diagram. A revised UML class diagram in pdf format without any annotations can be found in our team's gitLab repository.

The changes and the decision behind them will be highlighted below using terms like methods and class names from the diagram to further solidify the rationale behind the revision and put all the reasoning together. Anything that has been highlighted or annotated in the diagram and the reasoning is not covered is because that part of the diagram is related to the advanced requirements in Sprint 4 and will be covered in Sprint 4.

Mill and Intersection

A new Mill class has been introduced into the design architecture to represent a Mill in the game. In the earlier design diagram, it can be seen that the representation of a mill in the game is in the form of multiple methods. Later it was decided that representing Mills and their logic only with methods would not serve a great purpose in terms of design and the simple flow of logic and would ultimately lead to large lines of code under those methods. Hence, all the logic behind Mills in the game is in the form of attributes and methods in the Mill class, along with some changes to other classes/entities that take part in the mill, like the Intersection class.

There are two relationships between the Mill class and the Intersection class. The cardinality of the association relationship from a Mill to an Intersection is 1 at the Mill end and 3 at the intersection end. This is due to the fact that in the nine men's morris game a mill is formed when 3 adjacent intersections have a piece on them of the same colour (belonging to the same player). This cardinality allows to represent this logic elegantly.

The other relationship between the Mill and Intersection class is an association from the Intersection to the Mill class. The cardinality is 1 at the Intersection end to 0..1 at the Mill end. This is because as per the logic in our implementation an intersection can only take part in one mill at a time. When an intersection is already in a mill and another mill is formed which it is part of, it is then a part of the new mill.

Some new methods have also been added into the intersection class to support the logic behind forming mills and breaking mills.

Game and PlayerGame

In the previous design, it can be seen in our old UML class diagram that all the game-related logic was placed inside the GameBoard class. A couple of changes have been made regarding this part of the old diagram that can be seen in the new revised diagram. Firstly, all the logic that was present in the GameBoard class has been abstracted, such that the GameBoard class has now become an abstract class with abstract methods and common game-related logic placed inside it. The GameBoard class has been renamed to the Game class, which can be seen in the revised UML diagram, and placed into its own new package called game. Making such a change to the design will allow the architecture to follow the Do Not Repeat Yourself idea. Meaning that common code and logic will not be found in many places and written several times throughout the source code.

Now different types/versions of nine men's morris games can be made seamlessly by inheriting the common game attributes and methods from the Game class and providing their own implementation to the game by overriding and creating new methods. Hence, introducing an abstract class with common game logic promotes extensibility in the architecture. A new PlayerGame class has also been introduced into the current system. This class is for playing the normal nine men's morris game among two players and inherits from the abstract Game class while overriding its abstract methods to give its own unique implementation related to how the two player game is supposed to be played along with its own methods here and there.

ComputerGame and ComputerUtility

Similar to the PlayerGame class mentioned above, a new ComputerGame class is introduced in the revised UML diagram. The ComputerGame class was called the ComputerBoardGame class in the previous UML diagram, and the new ComputerUtility class was called the ComputerLogic class. The ComputerGame class will inherit from the abstract Game class just like the PlayerGame does and will include its own methods to carry out the logic behind a player playing with a computer. The ComputerUtility class will have methods related to helping the computer make decisions and choices against a normal player. This part of the UML is, however, part of the advanced requirements of the game and will be discussed in more detail later in the last sprint.

Piece and PieceType

The Piece class has had small changes made to it. There is a new enumeration that has been added to the design and implementation called PieceType. This enumeration has two values, which are RED and BLUE. The Piece class now has an association relationship with the PieceType enumeration. The cardinality of this relationship is 1 at the Piece end and 1 at the PieceType end. This is because a single piece can only be of one type. A new getter method has been added for the PieceType in the Piece class, and this will be used in other classes to help with different aspects of the game like winning and mill forming. Other new methods like freezePiece have been added to the Piece class, which will make the piece noninteractive on the board to allow the game to be played properly.

BoardUtility and Move

Very minor changes have been made to the BoardUtility class and the Move class. A couple of changes have been made to the methods of the class, as can be seen by comparing the two annotated diagrams. For example, the checkMillFormed method is no longer part of the BoardUtility class as all Mill functionality is now taken care of, as explained earlier by the Mill class and other classes.

The only thing changed in the Move interface is the return value of the move method. In the previous UML diagram and design, the move method would return void, meaning nothing. However, in the new design and diagram, the move method returns a boolean indicating whether the move was successful or not. As a result, there is a similar change in the AdjacentMove and FlyingMove (previously called FlyMove) classes.

PhaseType and GameMode

Another new enumeration has been added to the design. The enumeration is called PhaseType, and it has two values: PLACING_PHASE and MOVING_PHASE. There is a new association relationship between the abstract Game class and the PhaseType enumeration. The cardinality is 1 to 1 because each game can only be in one phase at a time. There is no FLYING_PHASE, as this phase is not applicable to all the players in the game at all times, and when a player is able to fly, the game is still considered to be in the MOVING_PHASE. The introduction of this enumeration into the design has helped with the flow of logic and the implementation of the game rules.

It is also noteworthy to mention that the GameMode enumeration from the previous diagram is no longer part of the architecture, as can be seen by its absence in the revised UML diagram. This is because the enumeration does not play any role in the architecture that was identified during the third sprint.

Quality Attributes (NFRs)

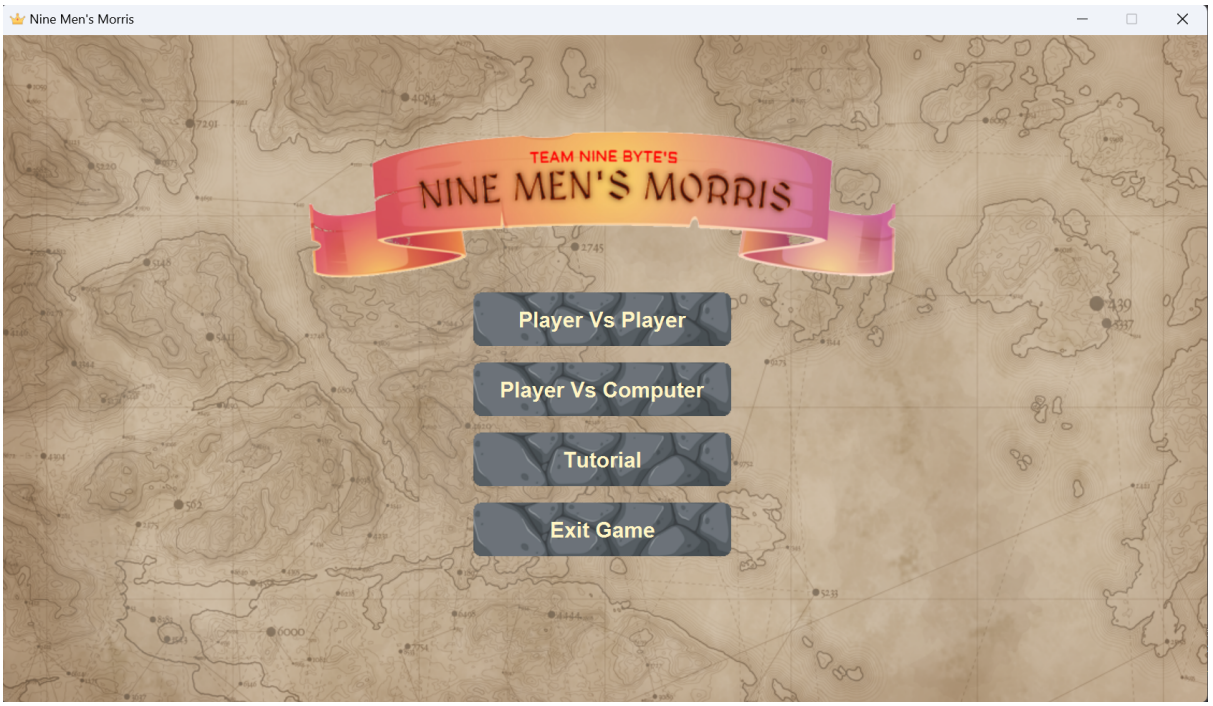


Figure 1: The main menu of our game

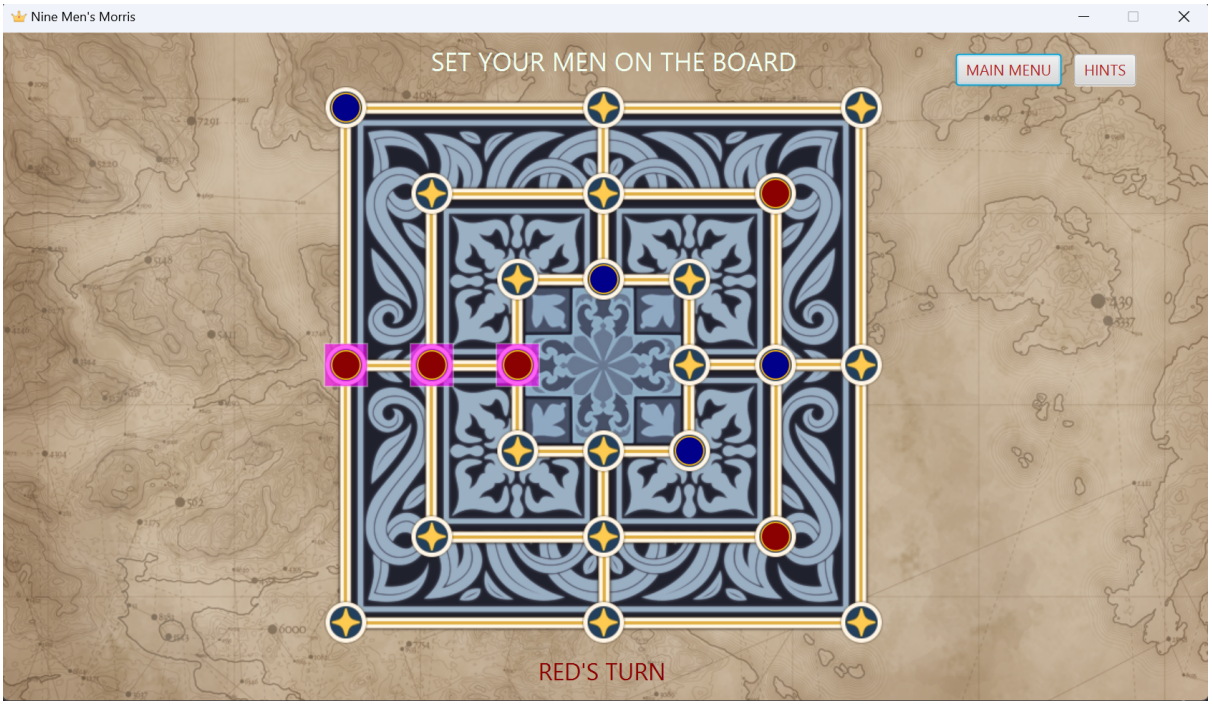


Figure 2: The appearance of a game in progress, illustrating how some of the quality attributes and human values are achieved

1. **Usability**

    Learnability

    The game expects many first time users, and players new to the game would not be immediately familiar with the rules. A game, or any application in general, is responsible for guiding its new users. If a user was presented with several applications, they would most likely pick the one that is easy to learn.

    Our game attempts to flatten the game's learning curve by displaying short messages at the top and bottom of the game board, stating what should be done, by whom, on a particular turn. This allows players to learn as they play the game, and familiarise themselves with the game rules step-by-step. This is also useful for experienced players, as they do not need to keep track of the turns and phases of the game.

    We intend to introduce a tutorial mode during our next sprint, as indicated by the Tutorial package in our UML. Since users that click on Tutorial Mode will most likely be first time users, we have considered adding pop-up notifications with more detailed explanations about game rules. Visual indicators to represent legal moves, such as colours around intersections, will also be included.

    Operable and user error protection

    A common trend in modern, widely-used applications (e.g. WhatsApp and online file converters) is simplification. For example, many applications utilise the drag and drop feature to make the application easier to use.

    Using a similar concept, we mimicked the real-world scenario whereby a player picks up a piece from the board, moves it around, then places it on another intersection when they slide or fly pieces. The user is given the option to click and drag their piece anywhere across the board. Different from the real-world scenario, however, is that the application additionally validates whether the piece was dropped on a valid position. This maintains the game's integrity by ensuring game rules are strictly followed. It also prevents users, especially new players, from breaking game rules that they were unaware (or maybe aware) of. The game 'bounces' the piece back to its original position if the move was indeed invalid, and allows the user to make their move again. Users do not need to drop their pieces precisely, as the game can 'snap' pieces to valid intersections as long as the piece was dropped close enough.

2. **Flexibility**

    Expandability

    Video games are constantly evolving to remain competitive and maintain the player's interest. It should be expected that more features will be introduced into the game in the future. Generally, as more features are added, code smells and poor design decisions become more apparent and these features

will take longer to implement, unless expandability was already considered during the project's inception.

Our team has introduced hinge points in the UML by abstracting numerous game concepts. For example, the Piece class is declared as an abstract class, since pieces can take on different colours, but are functionally identical. This eliminates a significant amount of code repetition, and adding a new type of token will not require the same logic to be written again - it saves a lot of time. The hinge is created on the abstract Piece class, between the client and Piece subclasses.

The Strategy design pattern was also introduced to implement the different phases (sliding and moving) of the game. The rules of each phase are encapsulated within the MoveType subclasses. A new phase can be quickly introduced by creating a new subclass. This adheres to the Open Closed principle, where the game's phase system can be extended, but no other classes have to be modified.

3. Reliability

Consistency

It is important to check that game rules are being followed at all times to ensure a fair game. No one likes to play video games that put their players at a disadvantage, even if it was due to a bug. The game must be able to consistently and correctly check for any rule violations whenever the state of the game changes.

Our game achieves this in several steps, by validating a player's move then analysing the consequence of that move at each turn.

At the start of each turn:

i. The game first determines which phase (placing, sliding or flying) a player is in by querying their current number of pieces in-play. This defines the rules that the player must follow for that turn.

ii. The player is then free to move any one of their pieces anywhere on the board.

iii. The game validates the move (see Usability - Operability and user error protection) by checking if the intersection is already occupied, and if the piece was moved legally, based on the previously defined rules for that player at the start of the turn.

iv. If the move was valid, the game executes the move.

v. The game detects if a new mill was formed, so the player can remove pieces belonging to the other player before passing their turn.

vi. A set of conditional statements is executed to check if the condition for a victory was met (the opposing player has 2 tokens or the opposing player's pieces have been trapped).

vii. The steps above are repeated at each turn.

We have considered additional edge cases the game might encounter:

i. Players cannot win during the placing phase even though the winning conditions are met, since each player can still place additional pieces on the board.

ii. A piece that forms two mills simultaneously, will form two different mills distinguishable by the game. The piece is recognised as being part of both mills.

iii. A piece that leaves an established mill to form a new mill will allow a player to remove opposing player's pieces.

iv. If all the pieces of the opposing player belong to mills, any piece can be removed to prevent a deadlock.

Most of these edge cases are shown in our video demonstration.

Human Value (Schwartz's Theory)

1. **Creativity**

   There are lots of Nine Men's Morris games on the internet. Browsing the Google App Store alone will show over twenty different versions. If we want users to choose our version of the game over the others, there is a need for our game to stand out. First impressions are usually formed based on appearance. Therefore, the visual appearance of our game is an important aspect to consider in order to capture a user's attention. In terms of UI, board and token design, our team noticed that many online implementations of Nine Men's Morris were visually identical.

   Our game attempts to stand out, visually, in multiple ways (refer to Figure 1 and Figure 2):

   i. It has a subtle background image

   ii. The game board has striking colours and intricate patterns, with intersections indicated clearly as stars

   iii. Adding blue are red tokens (instead of the usual black and white), with golden trims to match the game board

   iv. In the main menu, adding a game logo and buttons with hover effects

2. **Pleasure**

   People continue to play a specific game as long as it keeps providing them with some form of gratification. Games can provide this in multiple ways, winning is one example. Our team wanted both players to have the same experience while playing, and the 'pleasure' component to be persistent throughout the game - not just towards the end. To keep players invested in the game, the team introduced different forms of sensory feedback whenever an important event occurred.

   The game provides sensory feedback in multiple ways:

   i. Visual feedback as discussed in Creativity

   ii. Audio feedback. Different sounds are played whenever:

      a. A piece is placed or moved on the board

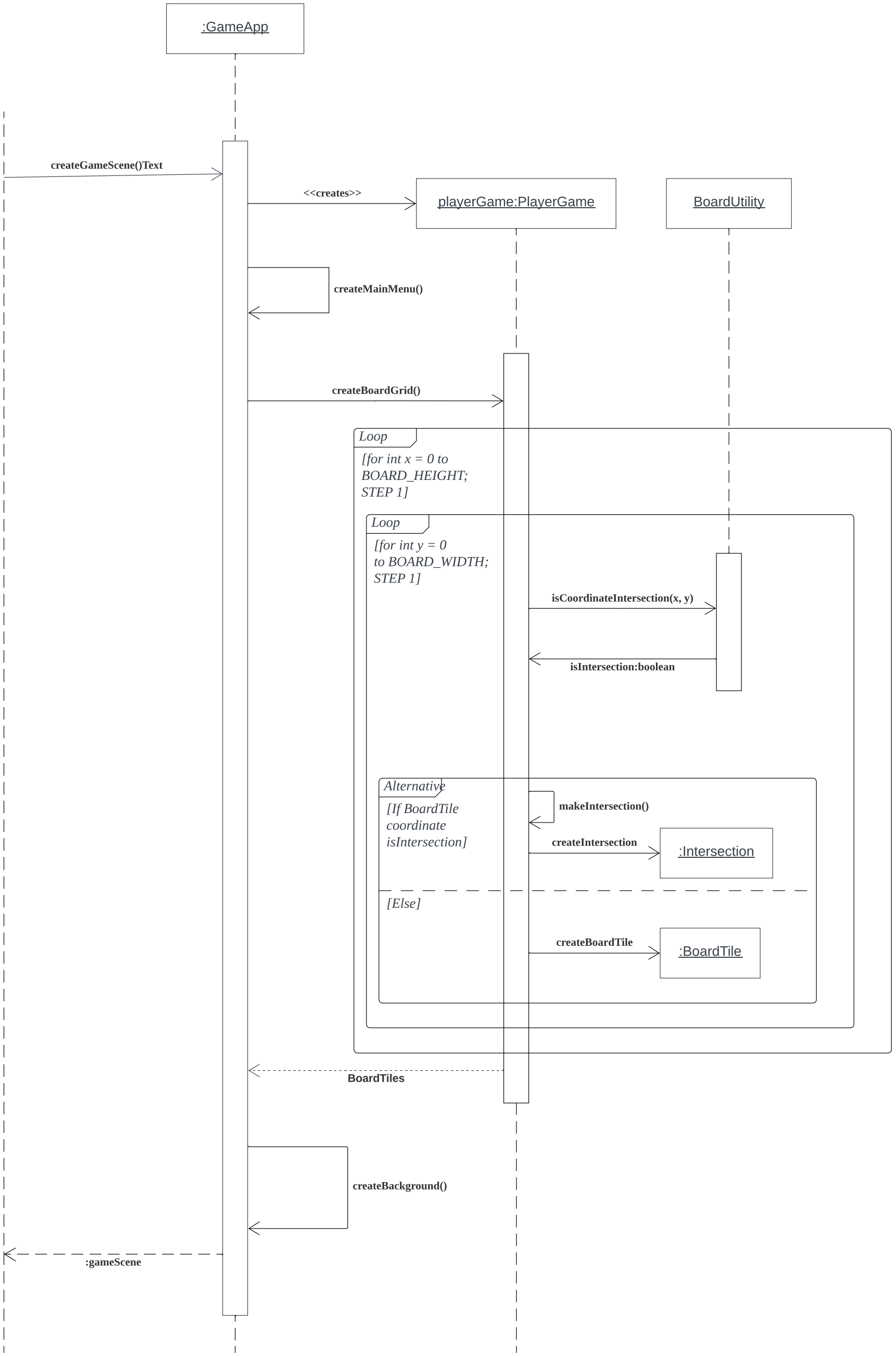
      b. A piece is placed in an invalid location on the board

      c. A mill is formed

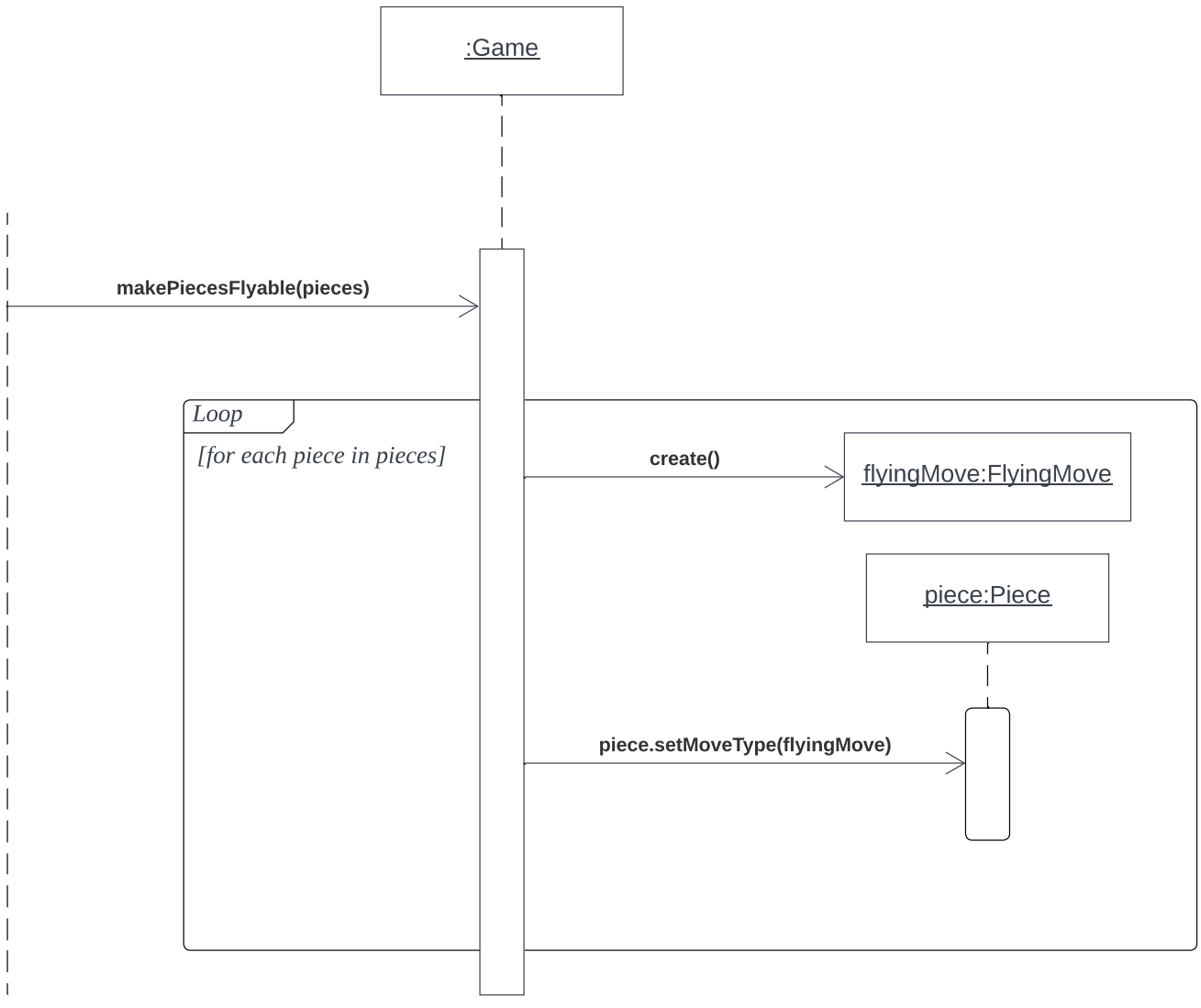
      d. A piece is removed

      e. A winner is declared

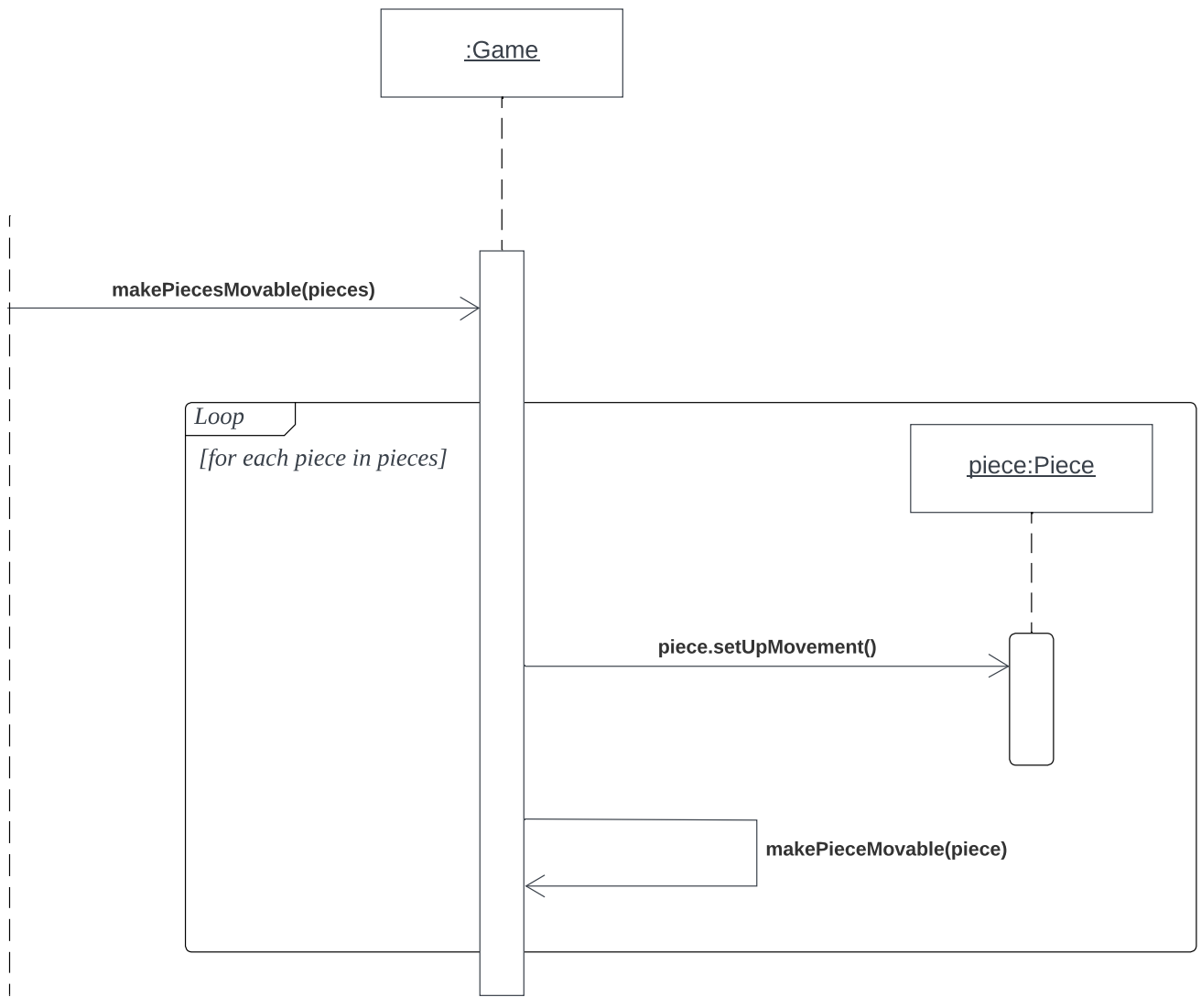
Bootstrap Sequence Diagram



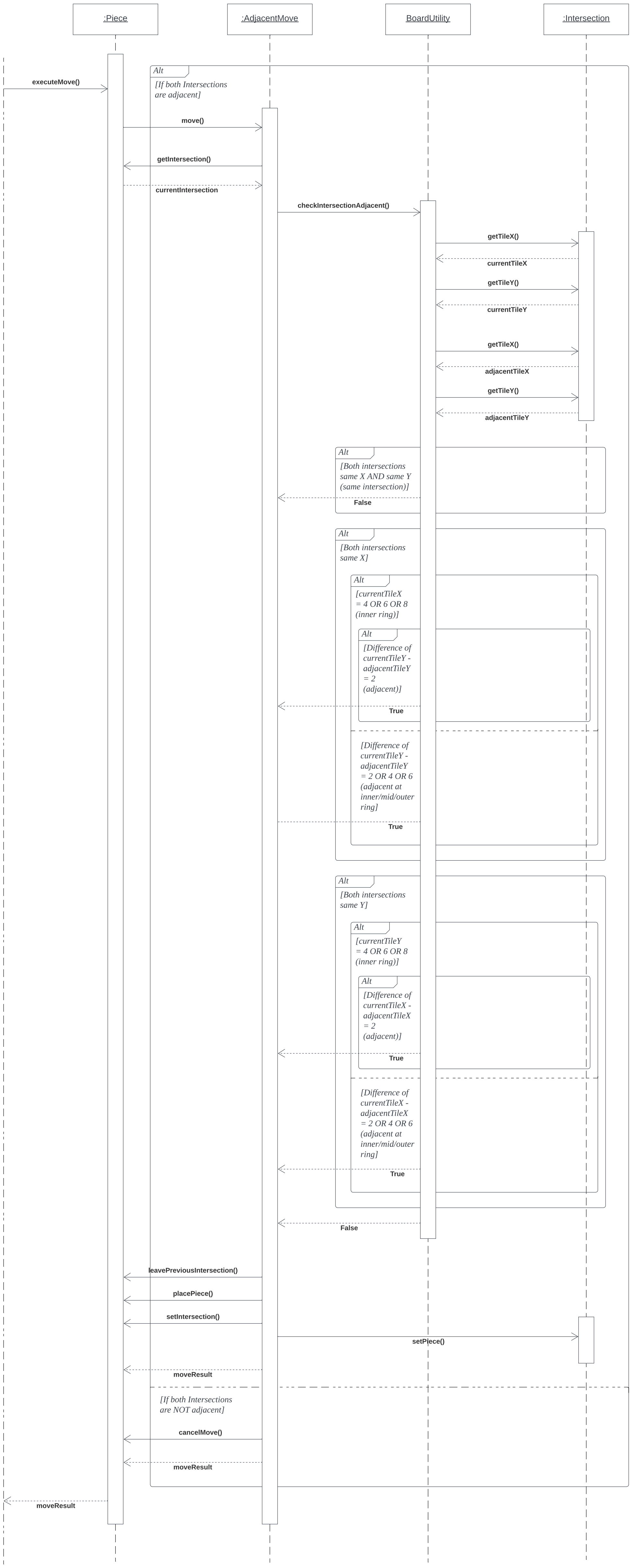
Making pieces flyable



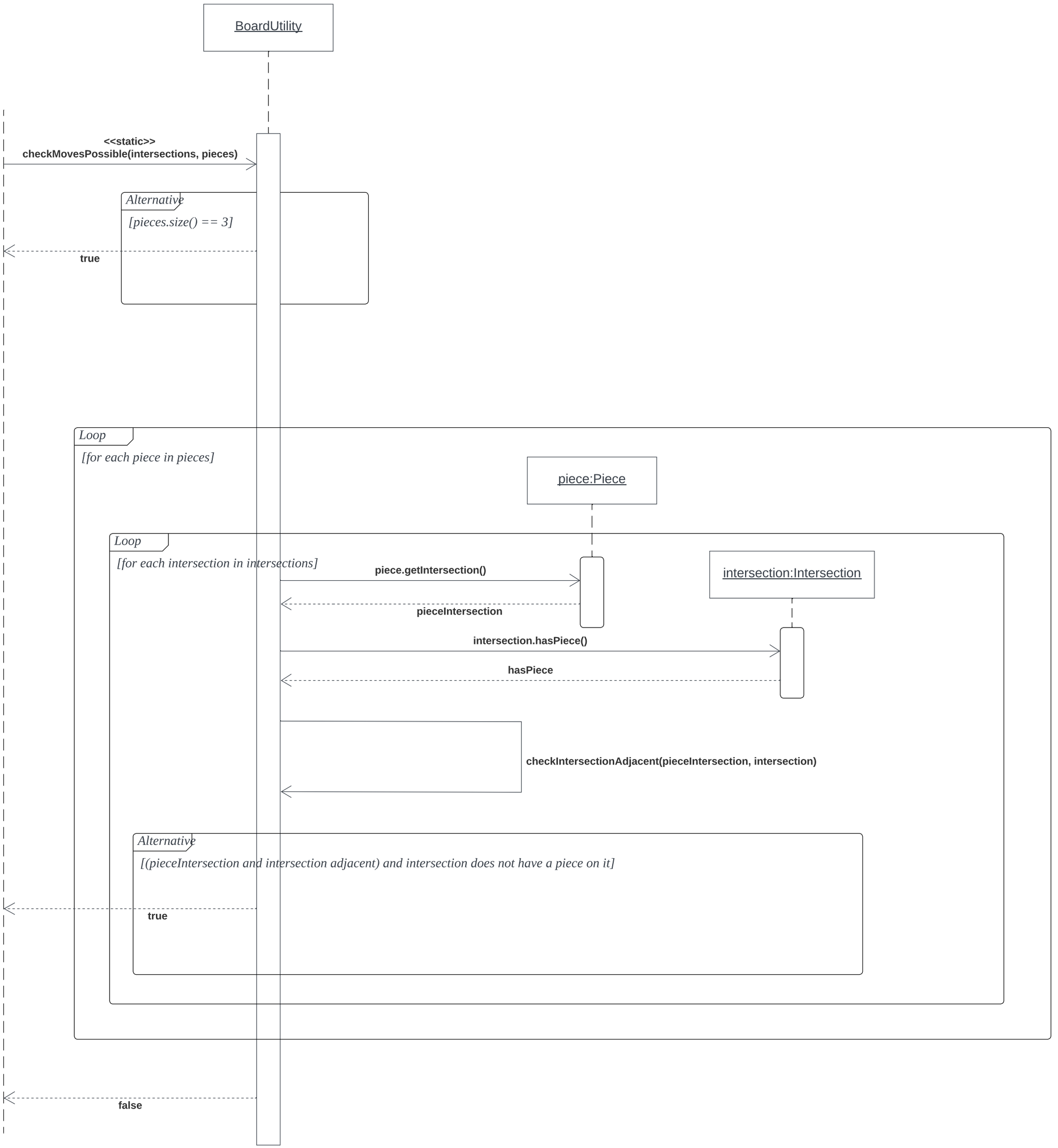
Making pieces Movable



Sliding pieces and adjacency verification



Checking if pieces have any place to move on the board



Mill detection along the x-axis