MONASH University

FIT3077

SPRINT TWO

Team Nine Bytes

SUBMITTED BY

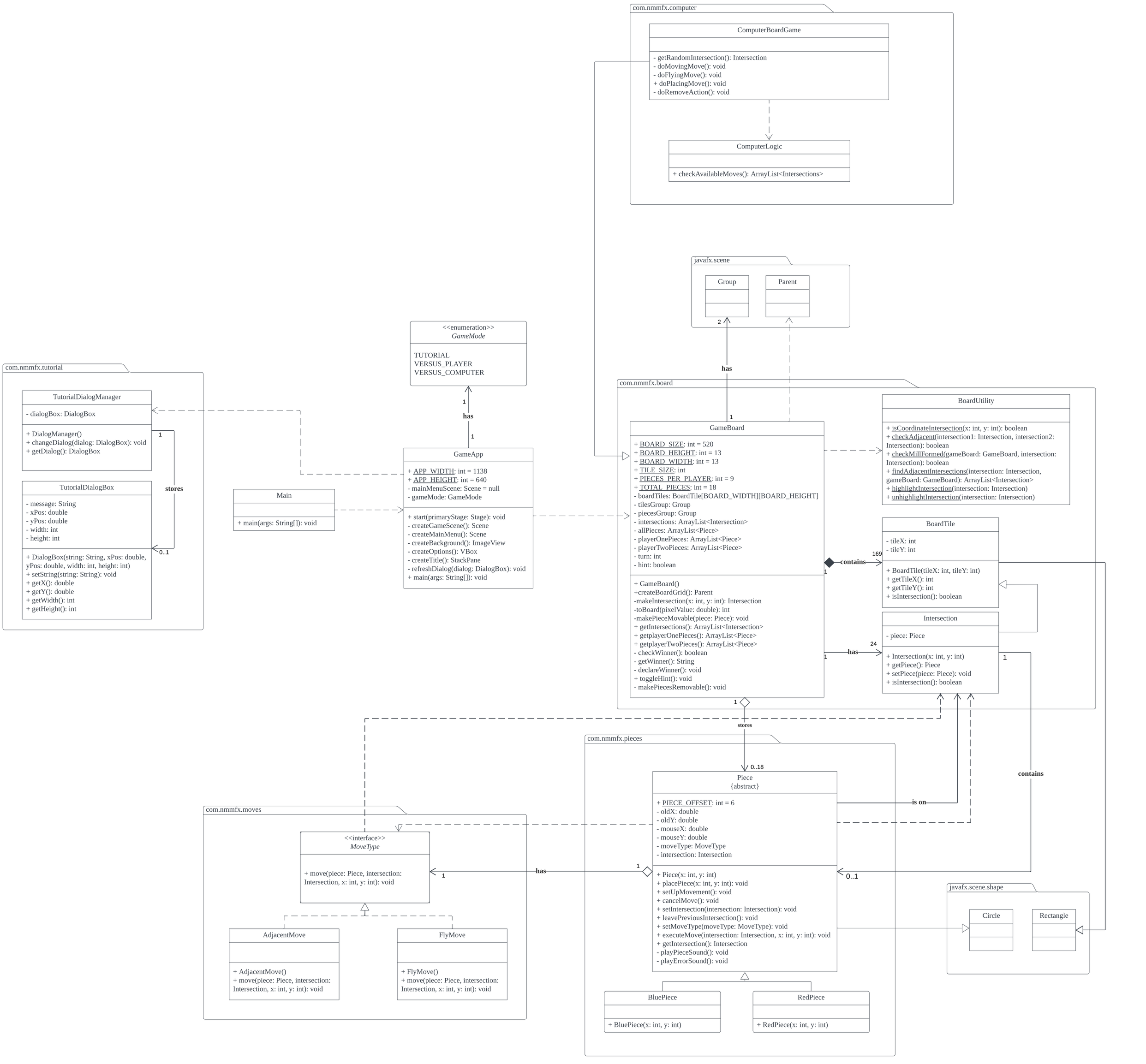
Shyam Kamalesh Borkar (32801459)

Lai Carson (32238436)

Richardo Husni (32767269)

Victor Lua (31362834)

9 Men's Morris Sprint 2 UML Model



NOTE: The components of the UML diagram that are not reflected in the source code of the project are design components for future requirements. This design rational only provides rationale for those components in the UML diagram that are implemented in code and are part of the work in progress game project.

The team would also like to point out some minor changes to the UI interactions in the nine men's morris game compared to our submitted Lofi prototype. In our initial prototype, the 9 tokens per player were placed on either side of the board. However, in our current implementation, there are no pieces on the side. The pieces will come into play when the player clicks on the intersections. For moving the pieces after they have been placed, in our current implementation, they need to be dragged onto an available intersection to be moved, in contrast to clicking on the piece and then clicking on an available intersection to move the piece there, which we depicted in our lofi prototype.

An executable file for the game is available in the form of a jar file in the root folder of the repository. Our game project is created using java and javafx using a jdk version of 17.0.2. To be able to run the jar file and try out what we have done so far, a Java Runtime Environment is required on the device. In addition, a jdk version of 17.0.2 or higher is required to be installed on the device to be able to run the jar file. This is because the project uses a jdk of 17.0.2. Not fulfilling the jdk requirements will render the jar file not runnable.

The relevant executable jar file can be found in the team's GitLab repository.

Design Rationale

Classes

BoardTile and Intersection

In the current implementation, GameBoard exists as a 13 by 13 collection of BoardTiles. This 13 by 13 grid is mapped to a background image representing the board. The user sees the image when they play the game, but not the grid. Each BoardTile represents a small fixed region in the GameBoard that a Piece may or may not move to. There are two types of tiles - a normal tile and an intersection, and Pieces may only occupy Intersections. Intersection extends BoardTile, since it has characteristics that are similar to a BoardTile (it has x and y coordinates which represent its position on the GameBoard), but an Intersection also needs to keep track of Pieces on it, which is why it has an additional association with a Piece. This approach adheres to the Single Responsibility Principle, since the Intersection itself keeps a pointer to a Piece if it is being occupied. This also enforces the Command-Query Separation Principle since the methods in GameBoard call the query methods in BoardTile to determine if a BoardTile is an Intersection, then queries the Intersection to find out if an Intersection is occupied. GameBoard then makes a decision and proceeds to call its own command methods to manipulate the position of Pieces around these Intersections based on game rules.

The original approach of the GameBoard was to have it exist as a single entity, without it being composed of GameTiles. This is possible since the Pieces can be placed in exact x and y coordinates on the window, but doing so would require some computation to find this exact position. This has the benefit of Pieces being able to occupy any relative position on the GameBoard, without it needing to ‘snap’ to any particular tile on the Board. This is also a drawback, since the code would be filled with formulas to calculate the positioning whenever a Piece was moved. These formulas would exist as a collection of magic numbers without any meaning, and reduce code readability. It would also be harder to map the positions of Pieces to the background image, since the coordinates of each position has to be computed individually and repeatedly, whereas the first approach maps all 169 (13 by 13) BoardTiles to the image whenever the board is initialised, so computing these positions only happens once.

Relationships

Composition between GameBoard and BoardTile

A GameBoard is composed of BoardTiles, and it keeps a record of all its BoardTiles in a collection attribute. Without a GameBoard, a BoardTile serves no purpose, since its x and y coordinate attributes are in reference to a GameBoard. If the GameBoard of a BoardTile is deleted, the BoardTile is deleted together with it, and it cannot be repurposed to fit into another GameBoard, since a GameBoard is initialised together with all its BoardTiles. This is why the relationship is not an aggregation. This introduces tight coupling between GameBoard and its BoardTiles, but this coupling is justified for the reasons previously mentioned - a BoardTile is useless without a GameBoard since they must work together to identify particular locations on the GameBoard.

This was the only option to connect BoardTiles with a GameBoard. A dependency relationship would not work, since a GameBoard must keep a reference to every BoardTile in order to differentiate them. Another approach was to have a separate class that was solely responsible for managing each Piece. The Game would prompt the Piece manager to iterate through its children (Pieces) and decide where a Piece could move every turn. However, this decision has to be made with information about the GameBoard (a Piece needs to know if the coordinates of its adjacent intersections, and determine if each of them are occupied), which the Piece object lacks. It could be argued that a PieceManager just needs an association to GameBoard to access this information, but that would require the creation of a new class and create a new association. It makes more sense for the GameBoard to make this decision, since it already has the required information at hand - the coordinates of every BoardTile.

Association between GameBoard and Intersection

A gameboard has a relationship with an intersection to allow for the gameboard to always have access to information regarding the intersection. Currently as part of the normal game this is not really needed because human players are the one’s that will manually interact with the board and based on that interaction an intersection or board tile will be evaluated. However, for future requirements such as the computer game, where the computer will make moves of its own it would need to have concrete information about the game elements to make proper decisions. For example, when the computer wants to take part in the placing phase of the game, it should have access to all the intersections in the game and place a piece on an intersection provided that the computer knows that the intersection is empty. The same goes for moving the pieces around.

Having an association with the intersection along with with the board tiles is a good design decision as this association will support future decisions and requirements while promoting good code health.

Aggregation between GameBoard and Piece

The relationship between GameBoard and Piece is very similar to the one between GameBoard and BoardTiles, as explained previously. GameBoard has a reference to all the Pieces it creates. However, unlike the BoardTiles, a Piece is not initialised together with the GameBoard, and will not be deleted when together with the GameBoard. Similar to

BoardTiles, coupling exists between Pieces and GameBoard, since a Piece must work together with GameBoard to determine where it will sit on the GameBoard, and keep a record of its own position on the GameBoard. A Piece can technically exist anywhere on the window, without the GameBoard, but its position would carry no meaning.

The alternative design where a class was created to manage all Pieces was considered, but not chosen for reasons explained in **‘Composition between GameBoard and BoardTile’**.

Inheritance

BoardTile and Intersection

For the design, we chose to use inheritance between the BoardTile class and the Intersection class. Multiple BoardTile objects are used to represent the board in a 13x13 grid manner. As part of the entire board system, we know that certain tiles will play the role of intersections (spots where pieces can be placed). These intersections are crucial parts of the nine men’s morris game. To cater to this, inheritance has been introduced between the BoardTile class and an Intersection class. An intersection will be visually represented as a tile, just like a normal board tile would. But an intersection would host additional functionality that would allow it to play and fulfil its role in the nine men’s morris game. This functionality includes the relationships between pieces. When creating the game board tile by tile if that particular tile has a specific coordinate in the tile grid system, an Intersection object will be used to represent that tile instead of a normal BoardTile. Since, the board has been chosen to be represented using a tile system, the use of inheritance in the Intersection class is the best design option.

Cardinalities

Intersection to Piece

The cardinality from intersection to piece is 1 at the intersection end and 0..1 at the piece end. This is due to the obvious logic that can be inferred from the nine men's morris game. Each intersection in the game needs to know whether it is occupied by a piece or not. Meaning whether a piece is placed at a particular intersection or not. Hence, the cardinality specifies the fact that an intersection has 0 pieces, indicating that the intersection is empty, or 1 piece, indicating that the intersection is occupied on the board. Having this relationship allows a lot of the game logic to exist. For example, when a piece is trying to be moved to another intersection that is legal, it is easy to check whether that particular instance of the intersection is occupied by another piece or not.

Piece to Intersection

For a piece to intersection relationship there is a one to one cardinality on both sides of the relationship. This is because a piece can only be at one intersection at a time at any point in the game. This cardinality will also allow the piece to be aware of which intersection it is on if an illegal move is made, so the piece can make its way back to its original intersection. Otherwise, if the move made is legal, the piece can communicate to its previous intersection that it is no longer being placed on it, which will also allow for the logic of a piece leaving the intersection.

GameBoard to BoardTile

The GameBoard to BoardTile relationship has a cardinality of 1 on the GameBoard side and a cardinality of 169 on the BoardTile side. This is because the entire game board is represented using a grid-based tile system, where all tiles together make up the game board. Since the GameBoard consists of 13 tiles horizontally and 13 tiles vertically, the total number of tiles adds up to 13x13, which is 169. This representation of the board makes the implementation of other design decisions much easier and smoother.

GameBoard to Intersection

The GameBoard to Intersection relationship has a cardinality of 1 on the GameBoard side and a cardinality of 24 on the Intersection side. The reason behind this is that the 9 men's morris game only has 24 intersections in total, hence the cardinality. The use of any other caradinality would result in misrepresentation of the game. Having this cardinality of 24 intersections would also allow the GameBoard to monitor the state of the intersections in the game.

Design pattern

The team noticed that a piece moves differently depending on the phase a player is in. It can either move to adjacent intersections, or fly to one. In other words, a Piece behaves differently, and will likely have to execute different, but similar code in order to move around. This move method has to exhibit polymorphism, but this can be designed and implemented in different ways.

Option 1: Do nothing

This approach has GameBoard deciding how a token should be moved by adding conditional statements to check which phase each player is in. Then, it executes one of two code blocks (a Piece can either move to adjacent intersections or fly) to move a Piece.

Pros:

1. Simple. No new classes or additional dependencies are created.
2. The GameBoard already has information about where each Piece resides on the board, and the locations of each intersection. It does not need to pass on information to other classes in order to delegate work.

Cons:

1. Violates the Single Responsibility Principle. The GameBoard keeps track and has to decide how every Piece should move. The board does not need to know every single way a Piece can move - this detail should be hidden from GameBoard.
2. Adding more ways a token can move will extend the conditional block, bloating the class.

Option 2: Use the State design pattern

In this approach, an interface would be created that defines an abstract method to move a Piece. For each unique way a Piece can move, create a child class (State) that implements this interface, such that each State implements its own game rules and logic within the abstract move method. There would be two States - one for adjacent movement and another for flying movements. The Piece and State would have a reference to each other in order to change the State of a Piece. The GameBoard (client) would only have to call the move method of a Piece, which in turn calls the overridden move method in State, since a Piece knows how to move based on its State attribute.

Pros:

1. GameBoard does not manage Pieces. Pieces manage themselves, by changing their own States. Adheres to the Single Responsibility Principle and GameBoard becomes less of a God class.
2. A Piece only knows how it can move based on its State attribute. It does not know of other States, and this promotes information hiding. This is possible because the State pattern uses aggregation over inheritance, which allows ‘blocks’ of functionality to be attached and detached from a class.

3. More maintainable and extensible. Just add a new child that implements the interface whenever a new type of movement is introduced. Adheres to the Open-Closed Principle.

Cons:

1. Creates a new association between every Piece and a State. Piece and State both need to update in order to reference each other, so that a piece can be moved correctly. These two classes are tightly coupled.
2. The GameBoard already has information about where each Piece resides on the board, and the locations of each intersection. GameBoard has to pass on information, such as intersection position to other classes in order to delegate the work of moving Pieces.
3. Adds additional design complexity and introduces newly coupled classes, since new classes and dependencies are added.

Option 3: Use the Strategy design pattern

The implementation of the Strategy pattern is almost identical to the State pattern. An interface defines an abstract method for child classes (Strategies) to implement the move method differently. However, in the Strategy pattern, Strategies do not need a reference to a Piece. Instead, the GameBoard (client) manages the Strategies used by each Piece, instead of the Pieces themselves. The work of actually moving the Pieces is still delegated to the individual Pieces, and subsequently the overridden move method in Strategies, just like the State pattern.

Pros:

1. The GameBoard already has information about where each Piece resides on the board, and the locations of each intersection. It should decide where a Piece can be allowed to move.
2. The GameBoard knows where a Piece can move, and the implementation details of how a Piece actually moves is in the Strategies, hidden from GameBoard.
3. A Piece only knows how it can move based on its Strategy attribute. It does not know of other Strategies, and this promotes information hiding. This is possible because the Strategy pattern uses aggregation over inheritance, which allows ‘blocks’ of functionality to be attached and detached from a class.
4. More maintainable and extensible. Just add a new child that implements the interface whenever a new type of movement is introduced. Adheres to the Open-Closed Principle.

Cons:

1. The GameBoard already has information about where each Piece resides on the board, and the locations of each intersection. GameBoard has to pass on information, such as intersection position to other classes in order to delegate the work of moving Pieces.
2. Violates the Single Responsibility Principle. The GameBoard keeps track and has to decide how every Piece should move.

3. Adds additional design complexity and introduces newly coupled classes, since new classes and dependencies are added.

Chosen approach

Option 3, where the Strategy pattern is used, was chosen to represent a Piece's polymorphic movement behaviour.

Option 1 was ruled out as the team agreed that the GameBoard class was already becoming bloated - it is noticeably longer than the other classes. The logic for a Piece's movement had to be moved out of the class in order to keep GameBoard maintainable, since adding new types of movement or implementing game rules in the future would exacerbate the problem.

Option 2 was ruled out as it offered benefits similar to the Strategy pattern in Option 3, but the tight coupling introduced as an association between each Piece and State was worse than the association between GameBoard and Piece in Option 3. The GameBoard requires information about every Piece anyway, so why introduce additional associations when we can utilise the existing association to solve the problem. Furthermore, changing the Strategy of a Piece requires the update of one reference (in GameBoard), but changing the State of a Piece requires two updated references (one in Piece, and another in State).