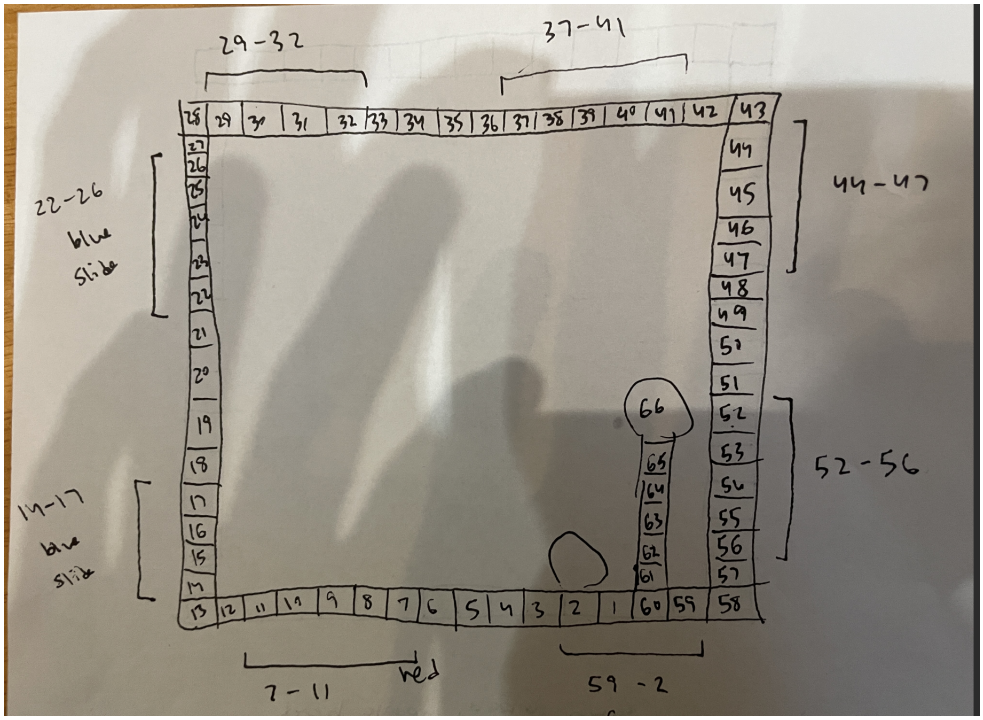
Alex Araki-Kurdyla
Leo McMenimen
Nathan Stamford
Derek Araki-Kurdyla
11/28/23

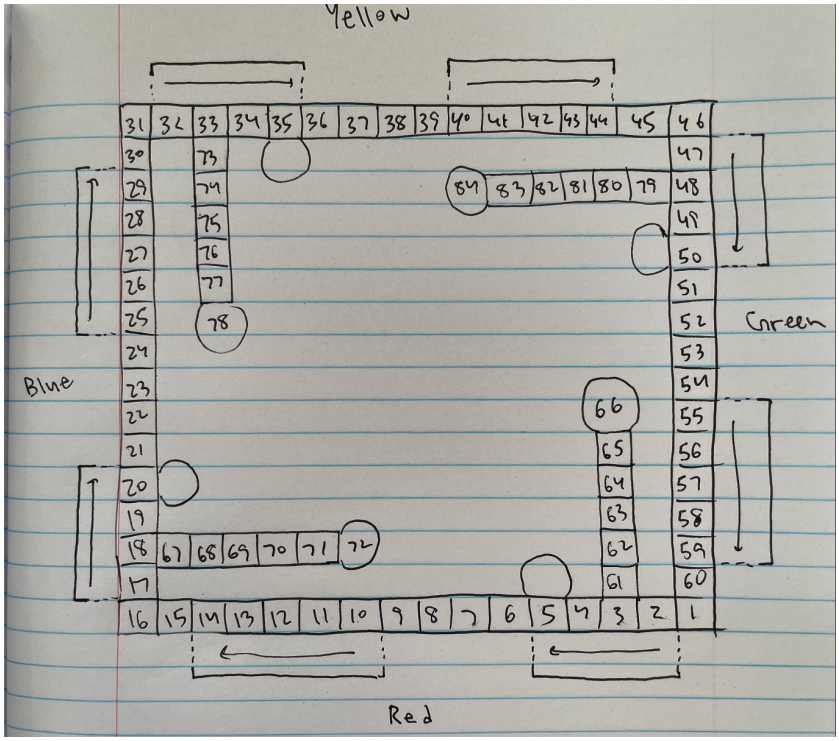
Design Manual

Our final project attempted to recreate the popular board game “Sorry” using JavaFx and Scene Builder. In order to accomplish this, we wrote 14 different java classes and 1 interface. We thought the best way to create a functional game was to create a “board” of spaces, and then write methods to manipulate these spaces and the pawns that will be on them in accordance with the rules of Sorry. All spaces on the “board” are assigned an enum value: OCCUPIED, UNOCCUPIED, START_SHORT_SLIDE_DIFFERENT, and START_LONG_SLIDE_DIFFERENT. We only marked the slides of different colors because you can only slide on an opposing color’s slide, so starting spaces for a slide of the same color would be irrelevant (see GameRules.txt for more detail).

In order for a player to move a pawn in our game, we also assigned each space on the board with an integer index. If the board was just a square perimeter, the job of moving pawns would be much easier. However, the addition of safety spaces hinders movement with indices. This is because except for the overlap of space 60 to space 1 (see image on next page), when moving a pawn, we would just be able to assign the landing space of a pawn to its current space plus the number of spaces it needs to move. This issue allows a pawn of one color from moving into a safezone of another color no matter how we indexed those spaces. In order to address this, we designed a solution in which pawns only see 66 indices, 1-66 (shown in the below image), in which instead of each color having access to all 84 indices, a pawn of a certain color only has access to the indices around the perimeter of the board plus the 6 safezone indices for that color. When it comes to moving a pawn, we could now use our initial solution in which we would just increment a pawn’s position by a certain number of spaces because there is no risk of it moving into another player’s safe zone. We would get the pawn’s index on the overall hashmap of 84 indices, convert it to its respective color index (using 1 of the 4 converter class objects we created), and then convert the respective color index plus(or minus) the number of spaces it needs to move back to the overall board index (again, using the same converter object as before).

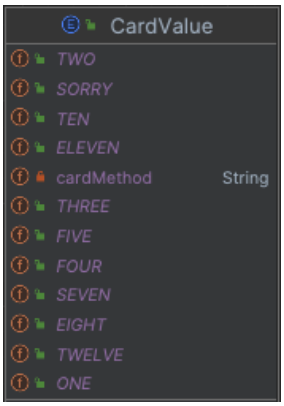


However, it does add another level of complexity because we have to have some sort of converter to convert a space on the individual color board to the overall board or vice versa. For reference, here is the overall board used:



In order to create the idea that the spaces on the overall board and the board from the perspective of the pawn are the same, we created four different classes (RedConverter, BlueConverter, YellowConverter, GreenConverter) that contain methods to convert the color index to the overall board index or vice versa. For example, our class BlueConverter contains two methods: convertToBoardIndex and convertToColorIndex. Both methods return an integer which is the index of the other “board” in the same space.

The Player class is where most of the game logic and turn taking occurs. At the start of the game, all 84 indices or spaces are marked as UNOCCUPIED. When the game starts and a player draws a card by pressing the “Draw” button on the UI, a function is called that initializes certain spaces on the board based on whose turn it is. For example if it is the red player’s turn, all of the opposing color’s start slide spaces (START_SHORT_SLIDE_DIFFERENT and START_LONG_SLIDE_DIFFERENT) will be initialized as UNOCCUPIED as well. Then, a card is drawn from the deck. The game carries out different actions depending on what card is drawn. The card can be one of 11 cases, which are all handled in a large switch statement in a method called findPossiblePawnMoves.

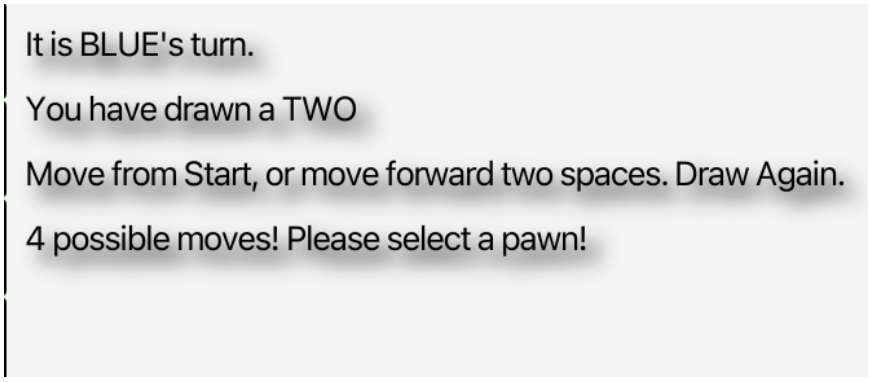


The findPossiblePawnMoves method contains a case for each possible CardValue enum, and each case varies based on the enum value:

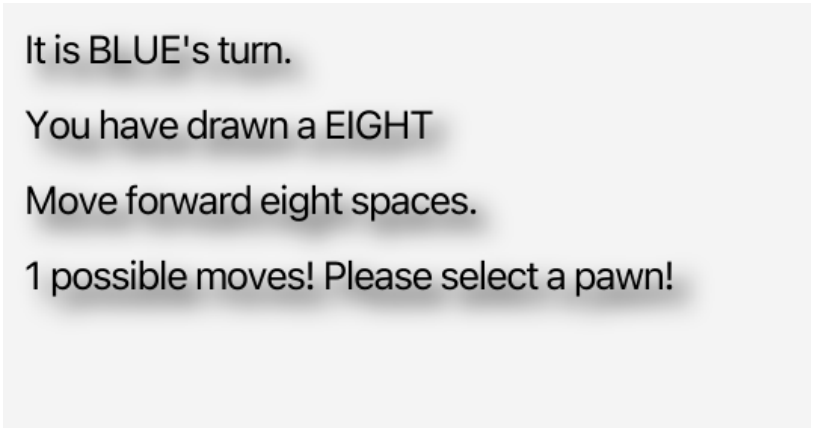
```
switch (drawnCard.getCardValue()) {
    case ONE -> {
        ArrayList<Pawn> possiblePawnMovesOnBoard = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSp
        ArrayList<Pawn> possiblePawnMovesAtStart = this.findPossiblePawnMovesAtStart(colorPawnsToIterateThrough);

        possiblePawnMoves.addAll(possiblePawnMovesOnBoard);
        possiblePawnMoves.addAll(possiblePawnMovesAtStart);

    }
    case TWO -> {
        ArrayList<Pawn> possiblePawnMovesOnBoard = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSp
        ArrayList<Pawn> possiblePawnMovesAtStart = this.findPossiblePawnMovesAtStart(colorPawnsToIterateThrough);

        possiblePawnMoves.addAll(possiblePawnMovesOnBoard);
        possiblePawnMoves.addAll(possiblePawnMovesAtStart);

    }
    case THREE -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 3);

    case FOUR -> possiblePawnMoves = this.findPossiblePawnMovesForBackwards(colorPawnsToIterateThrough, numSpaces: 4);

    case FIVE -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 5);

    case SEVEN -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 7);

    case EIGHT -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 8);

    case TEN -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 10);

    case ELEVEN -> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 11);

    case TWELVE-> possiblePawnMoves = this.findPossiblePawnMovesForForwards(colorPawnsToIterateThrough, numSpaces: 12);
```

Almost all of the cases use the method findPossiblePawnMovesForForwards, which, as the name suggests, finds all the possible moves for the current player based on the card that they drew. For case ONE and case TWO, another method is also used: findPossiblePawnMovesAtStart. This is because if a player draws a ONE or a TWO, they can choose to move a pawn from the start circle onto the board.

Both findPossiblePawnMovesForForwards and findPossiblePawnMovesAtStart take an ArrayList of pawn objects and an integer (numSpaces) as parameters. The ArrayList contains all of the 4 pawns of the player whose turn it currently is (no matter where they are on the board). The integer relates to the card the player drew. For example, if the player drew a 3, the integer parameter inputted into this method would be 3. These methods iterate through all of the pawns, and each time they check if the current index of the pawn plus the integer (or minus if the method is findPossiblePawnMovesForBackwards) is a valid move. If it is a valid move, the pawn is added to the ArrayList possiblePawnMoves. Some cases have different implementations based on the card that is drawn. For example, case FOUR calls findPossiblePawnMovesForBackwards instead of findPossiblePawnMovesForForwards. This is because if a player draws a FOUR, they have to move backwards four spaces. FindPossiblePawnMovesForBackwards has the same result as the other methods: it finds possible moves for the player whose turn currently is, it just checks different spaces. The UI mirrors this process by displaying text:



In this case, all of the blue pawns were still in the start area so the blue player could move any of the four pawns out onto the board, hence the 4 possible moves.



On the blue player's next turn, they drew an EIGHT. Since they only have one pawn out on the board, and therefore only one pawn can move according to the rules, there is only one possible move printed to the screen. When a player clicks the pawn they want to move and click the "Move!" button on the UI, the game updates the position of the pawn and moves the pawn by adding or subtracting the indices (from the pawn's point of view). Then, the new index the pawn moved to will be converted to an index on the overall board and that new space will be marked as OCCUPIED.

User Stories

Carla Cambridge: Carla will be looking for a relaxing time with her friends, where she does not have to focus and can take it easy and coast through the game. She appreciates games that are fun, quick to set up, and bring out some competition. Sorry is a great candidate for such a board game because it is relatively simple, easy to play, and easy to set up, but also has aspects of competition and can be very fun. Also, the games don't take that long and the interface guides the players throughout the whole game.

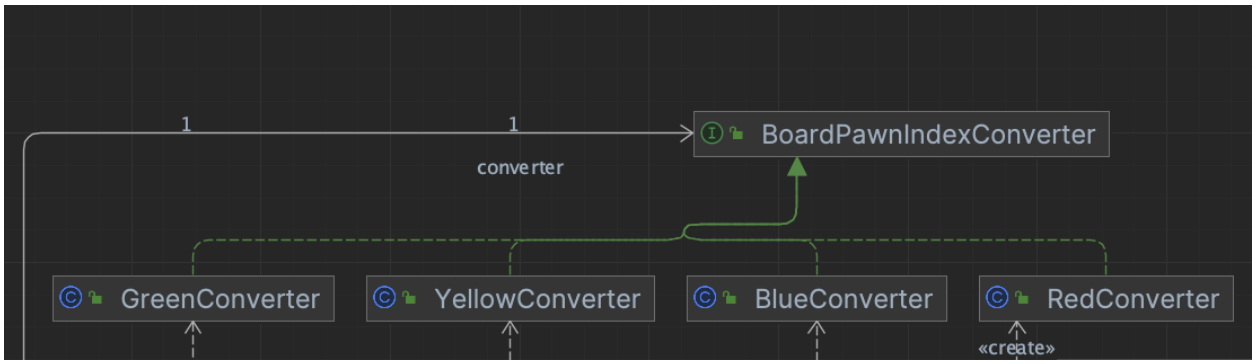
Greg Grumblebee: Greg Grumblebee is a middle aged dad of 3, who enjoys hosting regular family game nights, occasionally inviting the neighbors to join for a good time. Greg values games that are easy to pick up, engaging, and suitable for the whole family. Our implementation of Sorry is perfect for Greg for a multitude of reasons. First, up to 4 players can play our game which is perfect for his family game nights. Additionally, Sorry is very simple so his kids and new players will have no problem learning how to play and enjoying the game. Our interface is very easy to understand and use so Greg and his family will be very content playing.

Punishing Pete: Punishing Pete delights in sending opponents back to the start, using Sorry cards with glee. Watch out for his ruthless tactics! While Sorry may seem like a relatively worry free and straightforward game, there are some aspects that Pete will thoroughly enjoy. For example, Pete revels in his ability to send his opponents back to the start. While we did not completely finish all the implementations of the many cards that can be drawn, Pete will still have a lot of fun sending one of his opponent's pawns back to the start when he draws a Sorry card.

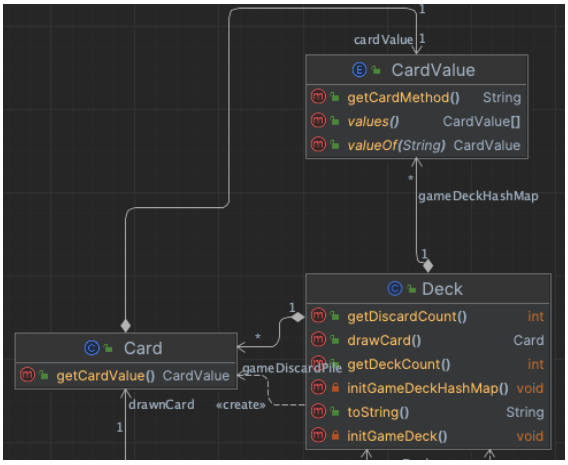
Scheming Steve: Scheming Steve's mind never rests. While others play, he's busy plotting how to 'accidentally' send them home. Always be on guard around Scheming Steve or be prepared to receive a fast loss. Since Sorry isn't the most strategic game in the world, Steve may not have a great time playing, However, there are some aspects of Sorry he would've enjoyed. For example, when a player draws a 7, they have the option to split the moves between two pawns. Steve could've used his restless mind to maximize this option to the best of its ability. However, we ran out of time so we could not implement this specific card.

These four user personas helped us decide what key aspects of Sorry we wanted to include in our JavaFX implementation. Due to time constraints, we could not completely achieve all that we wanted, but we still managed to create a working board game with an interface and sound game logic. In our opinion, the most important aspect of Sorry that we wanted to replicate was the base logic and the correct movements of pawns around the board. If we could get an interactive board game that simulates the basic rules and idea of Sorry, the project would be a success. We did end up accomplishing this, including the pawns being bumped back to the start, but we did not have time to implement some specific cards that have harder logic into them. For example, the cards SEVEN, TEN, ELEVEN and Sorry were not written how they work in the real board game. Each of these cards have something unique about them that make it difficult for us to get the correct movement in the time period we had to work on this project.

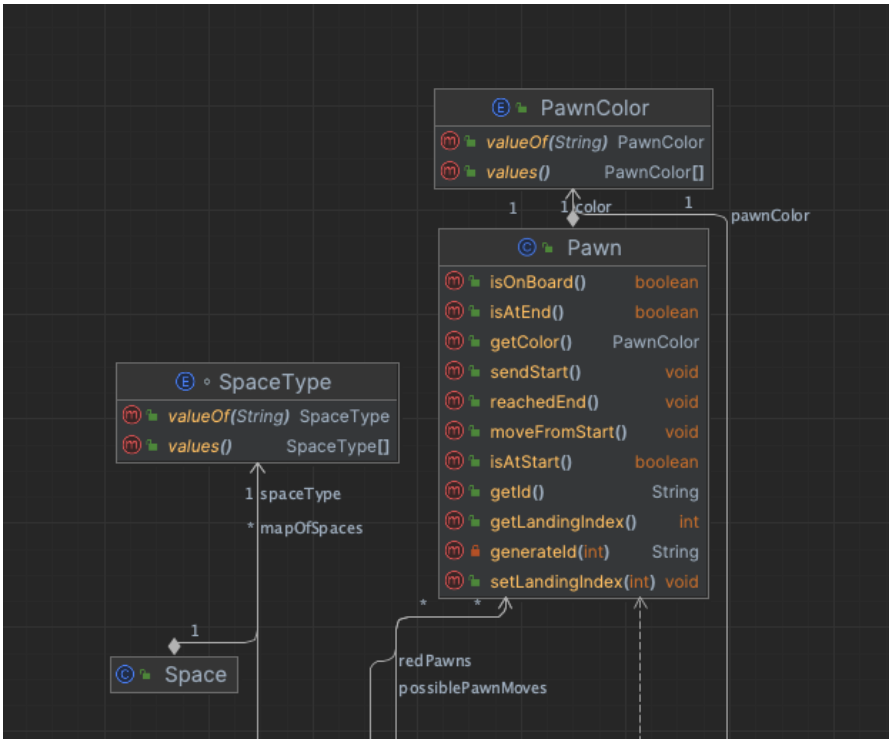
IntelliJ Generated UML Diagrams



This portion of the overall UML diagram showcases the interface implemented by all the converters. The interface contains two abstract methods to convert an index to the same index on the other board. There are four converters because there are four different sets of spaces initialized for each color. In order to match these different boards to the overall board used for the game, there needs to be a converter to convert the indexes between the boards. The difference between these boards is that the other color's safety spaces are not included. This is because an opposing color will never be able to enter these spaces, so it is efficient to exclude them. Also, it would be difficult to get a pawn to skip over these spaces entirely if they were included using our methods, so the board converters are the best way to go about representing the SORRY board.



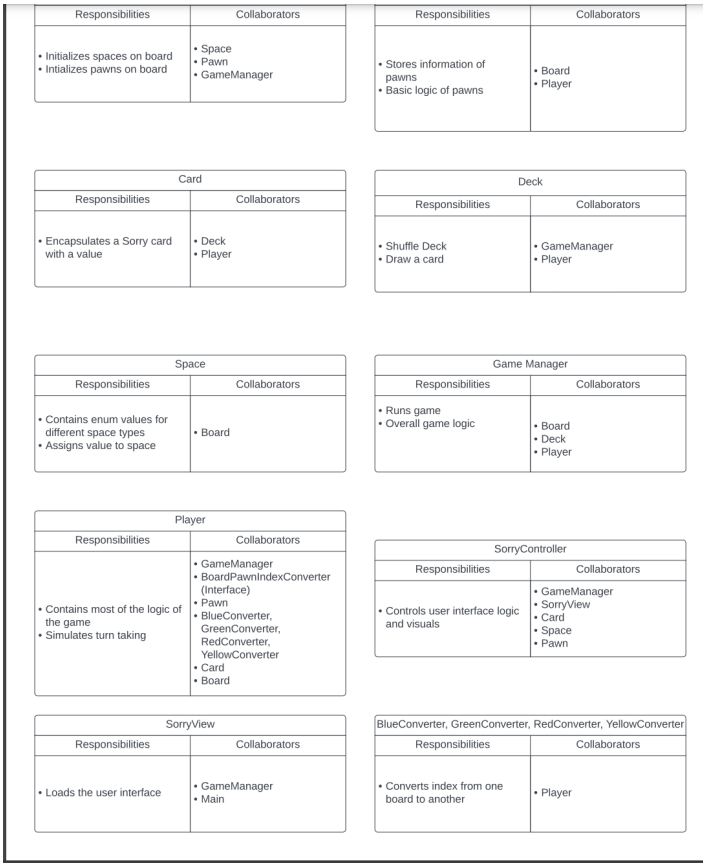
This portion of our UML diagram highlights the interaction between the Card and the Deck classes. CardValue is an enum value assigned to an object of the card class, and as you can see, the Deck class utilizes these enums in multiple methods. For example, the method initGameDeck() in Deck adds the correct number of cards with the corresponding enum values to the gameDeck being used.



This portion of our UML diagram highlights the interaction between the Pawn and Space class, as well as the respective enums of SpaceType and PawnColor. While the Space class is extremely simple (just contains a constructor), it's very important because all the spaces are initialized every turn. It was very important that we got this basic idea implemented because it made our more complex methods much easier to write. For example, the SpaceType enum is used frequently in the giant switch statement we have in the Player class to check if a space is

unoccupied and therefore a valid move. As for the Pawn class, it contains a lot of important boolean fields that are also used in a lot of other classes. For example, the private instance variable atStart is always set to true when a pawn is initialized (makes sense because all pawns begin on the start area). We also wrote a method in this class, isAtStart(), which returns the status of the atStart variable. The isAtStart() method is utilized by the Player class in the findPossiblePawnMovesAtStart method. In order for a move to be considered by this method, it has to be at the start area, in which this instance variable and method are extremely helpful.

CRC Cards



Responsibilities	Collaborators
• Initializes spaces on board • Intializes pawns on board	• Space • Pawn • GameManager

Responsibilities	Collaborators
• Stores information of pawns • Basic logic of pawns	• Board • Player

Card	
Responsibilities	Collaborators
• Encapsulates a Sorry card with a value	• Deck • Player

Deck	
Responsibilities	Collaborators
• Shuffle Deck • Draw a card	• GameManager • Player

Space	
Responsibilities	Collaborators
• Contains enum values for different space types • Assigns value to space	• Board

Game Manager	
Responsibilities	Collaborators
• Runs game • Overall game logic	• Board • Deck • Player

Player	
Responsibilities	Collaborators
• Contains most of the logic of the game • Simulates turn taking	• GameManager • BoardPawnIndexConverter (Interface) • Pawn • BlueConverter, GreenConverter, RedConverter, YellowConverter • Card • Board

SorryController	
Responsibilities	Collaborators
• Controls user interface logic and visuals	• GameManager • SorryView • Card • Space • Pawn

SorryView	
Responsibilities	Collaborators
• Loads the user interface	• GameManager • Main

BlueConverter, GreenConverter, RedConverter, YellowConverter	
Responsibilities	Collaborators
• Converts index from one board to another	• Player

To summarize, completing this project was extremely challenging but very rewarding. As a group, we effectively worked together to efficiently finish our project and submit a working virtual representation of Sorry. If we could redo this project from scratch, we would have thought more about connecting the game logic to the UI. This would've saved us from writing a lot of unnecessary methods and interactions between the user and the logic. However, we were successful and all learned a lot about both software development and working effectively in groups.