Design Choice Explanation (Milestone 4)

Authors: Sudarsana Sandeep

**Board.java:**

The board() method is used to create the 5 by 5 board and the method takes in a challenge level as a parameter. The addPiecesToBoard() method is used to add pieces onto the board. The checkwin() method checks if the game has been won by checking all the holes and incrementing the count if a rabbit is in there. It returns true if count is equal to the number of rabbits in the game. The move() method uses the coordinates of the object to be moved as well as the coordinates of the destination, to find out the object to be moved and asks that object (Rabbit/Fox) to move to that position. The method checks whether the object is a fox or rabbit and delegates the actual movement to that object.

**Slot.java:**

The slot(method) is used to create a slot on the board by taking in the x and y coordinates. A slot represents one unit on the board. The board will be created using slots. The setPos() method sets the initial value of a game piece or slot object itself. Slot also has getter methods to get the (x, y) coordinate of a piece on the board.

**Hole.java:**

This class inherits from Slot. An Array list is used to store the game piece type within the hole. With the use of the Array list we can then determine if a hole contains a rabbit or not. The hasGamePiece() and hasRabbit() methods are used to check whether the hole already has a piece inside. For the hasGamePiece() method it returns false if it is empty and true if there is an object in the list. For the hasRabbit() method it first uses one if statement to check if the list is empty or not and if the list is not empty it then uses another if statement to check whether the hole contains an instance of a rabbit. This will be important when the program needs to check whether all the rabbits are in holes. The getGamePiece() returns the piece inside the hole and removeGamePiece() method removes the piece from the hole.

**Mushroom.java:**

This class inherits from Slot. The Mushroom () method is used to create a mushroom on the board by taking in the position (x and y coordinates). This will help the to identify whether the rabbit can jump over a specific piece or not.

**Rabbit.java:**

This class inherits from Slot. The rabbit () method is used to create a rabbit on the board by taking in the position (x and y coordinates). It also sets the name of the specific rabbit since some rabbits have different colors. The canHop() method checks if a rabbit can perform the hop movement using recursion. Rabbit has a move() method which actually performs the move operation on rabbits, so the board class just calls this method. Whereas in milestone 1 the board was performing the move for all game pieces. We have done this to increase cohesion between the classes because in milestone 1 some classes were not doing anything other than constructing the object and board was doing all the movement.

**Fox.java:**

This class inherits from Slot. The fox () method creates a fox on the board by taking in the position of the fox. There are 2 parts to the fox. The first part of the fox is represented using xPos and yPos and the second part is represented using xPos2 and yPos2. The fox method also uses an if statement to check whether the fox is being placed vertically. The if statement returns true if the y positions are equal and false otherwise which would mean that the fox is being placed horizontally. This will help to determine which direction the fox can be moved in. The method getVertical() which returns a boolean type is used to get the result (whether isVertical equals true or false). The method setPos() sets the position of the fox on the board. The getTailX() method gets the location of the tail piece of the first fox and the getTailY() method gets the location of the tail of the second fox. The canSlide() method checks if the fox can perform the slide that was requested, this check is done using recursion. The move() method actually performs the move that was requested for the fox object. The move() method was added in the fox class to increase cohesion between classes, because in milestone 1 the board was performing the move operation for foxes.

**View.java:**

The view class is the view portion of the MVC for this project. In the previous milestone the view class was the view and the controller, but in milestone 3 we have decided to split them up because the view was too convoluted. The view has a JFrame, which contains a JMenuBar and JPanel’s. The JMenuBar contains a JMenu and some buttons depending on which panel one is on. The frame also consists of multiple panels, where each panel is a certain menu in the game. There is a panel which is the start menu, another panel which is the level select menu, another panel which contains the game board itself and where the game is actually played. The card layout from the previous milestone we had for the view has now been removed. To represent the board we are still using a grid bag layout.

**Controller.java:**

In milestone 3 we have decided to split up the View class and add a controller class. The controller class now has the actionPerformed method which was previously in the view class. The controller class acts as a middleman between the view and the model, so when the view changes the controller lets the model know the state needs to be changed and vice versa.

In milestone 3 our controller had one big actionPerformed method, coming to this iteration we have decided to split the actionPerformed into multiple methods for better coupling and cohesion in our project. The game has multiple screens/JPanels, so we split the actionPerformed for each panel, every panel/screen has its own actionPerformed method now specific to it.

**ActionStorage.java:**

The ActionStorage class is used to keep track of all the moveable game piece locations. This class was introduced to help with implementing the undo/redo features. We have decided to use a stack as the data structure to keep track of all the coordinates and believed as a group using a stack was the most logical choice. When implementing undo and redo we need to know the current location and previous locations of game pieces. With a stack, it’s just pushing and popping the coordinates from a stack to achieve the undo and redo functionality.

**Solver.java:**

The solver class attempts to solve the board using BFS and if a solution is found, it’s stored in a variable called solution. The solver consists of a list of possible moves (list of integers), the list of moves to win the game, a queue of moves, a variable for the color of the last fox moved, a Boolean to check if all the children of a node(moves) are checked, and the board that is being tested. The solver creates a new list of moves (or passed one recursively) and makes those moves to set the board to a certain state. The solver then picks one of the game pieces as a starting point and checks if it can make a proper move or moves. If so, it adds it to the queue and checks if those moves resulted in a win. The solver then recursively checks the children of that move and only checks the children once all the move’s neighbours were checked. Once all children are checked, it’s looped back until the queue is empty or a solution is found.

**LevelBuilder.java:**

The LevelBuilder class allows users/players to create their own levels on top of 5 premade levels that we as a group have made. The level builder consists of multiple buttons and JMenuitems which aid the user in constructing the level. LevelBuilder class uses multiple methods to give the level builder functionality such as adding/removing pieces and button highlighting to help the user out when creating levels. The created level is then checked by the solver to see if the level created was actually legal or not because it’s possible one could create a level that could not be solved. Finally, the level is saved in XML by using the Saver.java class and loaded to play using Loader.java class.

**Saver.java:**

The Saver.java class is used to convert to board into an XML file. This was achieved using JAXB and its marshaller. So the board is first converted into a 2d array of strings which is then converted to an XML file using the JAXB marshaller.

**SaveFile.java:**

The SaveFile.java class is used to convert a plain old java object into an XML file.

**Loader.java:**

The Loader.java class is used to convert an XML file into a board that the user can play. This was done using JAXB and its unmarshaller. So, the XML file is first converted into a 2d array of strings using the JAXB unmarshaller. Then the 2d array is converted into a board object which the user can now play.

The Saver, SaveFile, and Loader are all used for the save and load features of milestone 4 for the project. They also save and load the levels that are created by users.