Design Decisions

JumpInModel: JumpInModel contains all the logic of the actual gameplay of JumpIn. The user does not directly interact with the model, however the View that the user interacts with updates only when certain elements of the model change. The model handles the movement of every movable piece (rabbits and foxes). This includes checking whether a move is valid and updating the board when a movement is made. JumpInModel also stores the important board data, like previous moves, and redone moves, and whether the game is finished (user has won) or not.

JumpInView: The view uses the Java Swing library and displays all the content of the components. The view subscribes to the model and listens when the model is changed. Once it is changed, the view is notified and updates the buttons on the grid layout right away to reflect the model's changes.

JumpInController: JumpInController is a class that changes the values in the model to reflect actions taken by the user. The controller sets and handles the action events for the JButtons on each GUI (JumpInView and LevelEditorView). It also contains methods for changing the elements of the build GUI.

Board: The Board class contains a 2 dimensional Array of spaces to keep track of all the pieces currently on the board and their locations. This class is responsible for determining and incrementing how many holes are filled on the board, and providing that information to JumpInModel so that JumpInModel can determine when the game is done. Assembles the spaces and number of empty holes on the board together. The Board doesn't need to take in any parameters to create a new board. However, the Board uses a copy constructor to make replicas of itself and is used quite often when doing the undo and redo functions. The board is used quite often by the model so that it can "move" (set) the spaces of different pieces.

MoveableSpace: The Interface Moveable Space is implemented by Rabbit and FoxPart, the two pieces which the player can move, so that determining if the player has selected a valid piece to move is easier. The only method contained in moveableSpace is move which both moveable pieces must contain.

Space: Parent class to Hole, Mushrooms, FoxPart, and EmptySpace. Space is used as an abstraction to represent all the different pieces and empty spaces on the board. It is used quite often for creating and manipulating JButtons for the board of the game.

Hole: Hole is used to represent the goal of the game; to make all rabbits jump into their respective holes. When a hole is filled it sets a Boolean value of true, indicating it is filled. When the number of true values for holes filled matches the number of rabbits, the game is over (win). Holes, however, cannot be placed by the level editor as they are fixed in position by the games' rules.

Mushroom: Representation of an obstacle a rabbit can jump over. Mushrooms can be placed with the LevelEditorView as well. A Mushroom is very similar to a Space however a mushroom

EmptySpace: The EmptySpace class is a placeholder object for the locations on the board with no piece and after initialization acts solely as a destination for a move.

Rabbit: The Rabbit class contains the move function which is used to set the new position of the rabbit while updating the board state is done in Board and all the logic and determining whether a move is valid is done in JumpInModel.

FoxPart: The Fox class is the most complicated piece as each fox takes up two spaces. In order to keep track of this, each FoxPart contains a second fox part such that the two parts stay together when moving. Each fox part also has two booleans, isVertical which is used in determining valid moves for the fox and its orientation and isHead which determines which of the two FoxParts is the head and which is the tail. The moveBoth method is used in order to make moving the fox simpler by moving both pieces at the same time instead of one followed by the other, however the movement of individual fox parts can also be done via the move method inherited by the interface.

JumpInSolver: The JumpInSolver was used to help the user solve the game if he or she is stuck. The solver lists a series of pieces to move with directions in chronological order. The solver utilizes a Depth First Search algorithm to solve the puzzle. The solver is also used at the start of the game when building a level, to determine whether if the custom level made is solvable before starting the game. The solver stores the series of hints to move the pieces in a stack and prints them out in an ArrayList.

LevelEditorView: The LevelEditorView is a view that pops up at the start of the game. It allows the user to select pieces and place them on the board. The LevelEditorView, much like JumpInView subscribes to the model and listens when the model is changed. It updates the board once it gets notified. When you press “Play” on the level editor, the level you’ve just created will be generated. The level editor also has options for you to “Reset” a board you didn’t like originally when you designed it. The last saved level will be able to be opened from the builder screen, using the ‘Load template’ button.

SaveLoadJSON: SaveLoadJSON is a class dedicated to the saving and loading functionalities of the game. Saving a level formats the board into an array of spaces in a JSON file in the project directory (savedLevel.json). The load feature reads the JSON file line by line and parses it to recreate the board that the JSON file represents.

Written By: Benjamin Ransom

Revised by: Rafid Dewan (November 4, 2019)

Revised by: Rafid Dewan, Lazar Milojevic (December 2, 2019)