**JUMPIn’ Documentation**

**Game Design:**  The overall implantation of the game is based on the observer pattern/event model. Additionally, the game revolves around the concepts of:

**Test Driven Development:** Test-driven development methodology was used for the implementation of the design. Before implementing each class, tests were created that could be run to ensure that each class was performing the functions it is supposed to be. For example, a Rabbit test which tested each possible jump it could perform and ensured that expected output.

This approach to the development made it easy to develop the classes while using the tests as a reference to make sure that the implementation was on point. With each change to the class, tests were ran again to ensure that it was still implemented properly. After passing all initial tests, all the code was carefully merged in the development branch and extensive testing was performed to try to break the implementation by adding more tests and edge cases for the class to pass. This process was repeated until satisfaction of the capabilities of code was achieved.

**Correctness**: The approach of correctness was used during the software development cycle. Correctness is the adherence to the specifications that determine user’s interaction with the software and how the software should behave when it is used correctly.

Hence, during the development process of the project; following rules were strictly followed:

* Problems were defined completely such as for the rabbit jumping, the problem statement included multiple cases such as:
  + Rabbit jumping over the empty box/tile.
  + Rabbit jumping over the mushroom.
  + Rabbit jumping over the fox.
  + Rabbit jumping over other rabbit.
  + Rabbit jumping over an empty elevated board box/tile.
  + Rabbit jumping over a filled elevated board box/tile.
  + Rabbit jumping over a filled hole.
  + Rabbit jumping over an empty hole.
  + Rabbit jumping out of the board.
* The algorithm and the program logic were engineered according to the completely defined problem set. Such as for the example in the above point, it was made sure that the logic was implemented for each possible scenario.
* Reuse of the models and already existent code as much as possible. It was made sure that the code and is being re-used as much as possible and is not being written repeatedly. Such as for the integration if the Graphical User Interface; instead of rewriting the model again or replicating the entire code under view; it was made sure that the model code was being used and was integrated with the GUI.
* Proving of the correctness of algorithms during the design phase.
* Attention was paid to the clarity and simplicity of the game. It was made sure that the code was Highly cohesive and loosely coupled.
* Verification of each part of a program right after the developed. For this purpose; test driven development was used.

**Continuous Integration:** A CI process using Gradle and GitHub Actions was set. The project was managed by using the gradle build system which allowed automated testing, building and analysis. PMD for static code analysis as well as test coverage tools to monitor tests was used as well.

**Immutability**: Some of the classes in the project have been made immutable. Immutable class means that once an object is created, we cannot change its content. This concept has been used in the board implementation and the model.

**Persistent Data Structures**: Persistent Data Structures have been used in the game implementation. **PCollections** library was used for this purpose. A persistent data structure preserves the previous version of itself when being modified and is therefore effectively immutable. Persistent data structures allow both updates and queries on any version.

In the implementation of the game, many functions/operations perform small changes. Hence, copying the previous version wouldn't be efficient. To save time, memory and improve the efficiency, it is important to identify similarities between two versions and share as much data as possible. Hence, the use of pcollections library in the design and implementation of the game becomes handy.

**Project Structure:** The project is composed of four packages in total:

* **Model**
* **Solver**
* **Tui**
* **View**

The game instance can be started by running the Application.java file located outside the above-mentioned packages.

**Model:** All the classes under the model package are responsible for dealing with the processing of the user entered input. Here’s a brief explanation of what every doing:

* **Board.java:**  Creates and manages the state of the board by performing operations through defined methods. Also includes methods to delegate slide and jump commands to the animals on the board.
* **BoardHistory.java:** Responsible for managing the history of the board moves. This class achieves this by serializing its states. This class also contains options to undo and redo, which makes calls to the old board states that were serialized.
* **BoardItem.java:** An abstract class that deals with the state of the board items. It contains methods that generate the items for the board.
* **Containable.java:** An interface that represents board items that can be contained in others.
* **ContainerItem.java:** An abstract class that represents the hole objects on the board.
* **Coordinate.java:** Class that creates coordinates on the board.
* **DefaultBoard.java:** Class that creates the board default layout.
* **Direction.java:** An enum class that generates the directions. Has been implemented to deal with the String version of the game.
* **Elevated.java:** An interface that represents an elevated board item.
* **ElevatedBoardItem.java:** A class that is responsible for dealing with and the creation of items on the board that are elevated.
* **EmptyBoardItem.java:** The class that deals with the generation of empty space on the board.
* **Fox.java:** a class that represents a fox on the board, which can slide across the board to move.
* **GameState.java:** An enum class that deals with the generation of enums that keep track of the current game state.
* **Hole.java:** A class that deals with the creation and management of the holes on the board.
* **InvalidMoveException.java:** A class that deals with the exception when an attempted move is invalid.
* **MaybeMovable.java:** An interface that deals with the representation of the objects that maybe can move on the board and hence are type of moveable items.
* **MaybeObstacle.java:** Represents objects that may be obstacles on the board**.**
* **Movable.java:** An interface that the board items that can move.
* **Mushroom.java:** Represents a mushroom item on the board.
* **Orientation.java:** Enum class containing all the possible orientations a fox can be oriented.
* **Rabbit.java:** a class that represents a rabbit on the board, which can jump across the board to move.
* **SingleBoardItem.java:** An abstract class that specifies a board item with a single coordinate.

**Solver:** This package deals with the auto game solving implementation. It has following two classes:

* **Move.java:** Used to store valid moves for a object. These valid moves are used in the solver code.
* **Solver.java:** Solver class contains the logic for an algorithm that solves the board from any position.

**Tui:** This package contains classes that are responsible for the String representation of the game. It contains following classes:

* **ANSIColor.java:** This class contains the colors used in the text user interface.
* **ItemUIRepresentation.java:** An enum class that contains a UI representation for each item, usually first character(s) of the item's name.
* **JumpInClient.java:** This class is used for string input parsing and parses them into commands that can be applied to the board.
* **Main.java:** This class deals starts with the starting of the String version of the game.

**View:** This package deals with the creation of the Graphical User Interface for the game as well as is responsible for taking in the user input and doing processing according to it. It acts as a bridge between the GUI and the model. A brief description of its classes are as follows:

* **ApplicationPanel.java:** This class deals with the initialization of the GUI board outer components including the JToolBar i.e., menu bar.
* **Board.java:** This class deals with the initializes with the inner components of the GUI i.e., the board setup. Integrates the view board with the one present under the model.
* **ButtonBoardItem.java:** This class creates button items on the board.
* **Circle.java:** This class has a constructor that creates a circle of a specific color to be placed on the board.
* **ContainerItem.java:** This class that represents an item on the board GUI that can contain another item. Integrates the GUI ContainerItem class with the one in the model.
* **Elevated.java:** This class represents an Elevated item, which is a darker color and can contain another item. It is also responsible for the integration of the view elevated items with the one under the model package.
* **Fox.java:** Class that represents a fox object on the board GUI and integrates it with the logic laid out for the fox under the model package.
* **GUIBoardItem.java**: Creates a GUI item with its associated board object.
* **GuiColor.java:** Specifies the RGB values of colors to be used on the board.
* **Hole.java:** Item representing a Hole object in the GUI.
* **ImageResources.java:** A class containing all the images used by the GUI, with each image mapped to a string.
* **ItemClickEvent.java:** An event thrown by objects when they are clicked, specifying their coordinates.
* **ItemClickListener.java:** Interface for items that can be clicked; triggering an ItemClickEvent.
* **Mushroom.java:** Represents a mushroom object placed in the GUI.
* **Rabbit.java:** Represents a rabbit object placed on the GUI.
* **ToolBar.java:** This class creates an instance of the menu bar that contains items such as redo move, undo move, display message etc.

**Game Rules**

- Foxes can only slide.

-> Fox sliding is only legal if:

-> From the fox current location to the destination location, there are no obstacle.

-> Items that qualify as obstacles for foxes include rabbits, other foxes, elevate blocks/tiles on the board, mushrooms.

- Rabbits can only jump.

-> Rabbit jumping is only legal if:

-> There is an obstacle in-between the rabbit current location to the destination location.

-> Items that qualify as obstacles for rabbits include other rabbits, foxes, and mushrooms.

-> Rabbits can jump over multiple obstacles as well. If there is an obstacle (as defined in the line above) from the

rabbit current location to the destination location and none of the blocks/tiles in-between are empty, the move is legal.

- Mushrooms are non-moveable items and cannot be moved from their position on the board.

- To win the game, user must ensure that all the rabbits that are present on the board have jumped in the holes.

-> If the game is won, a pop-up window will be displayed; confirming the win of the user.

-> If the game is not won yet or still in progress, no such message will be displayed.

**Game Instructions**

- To play the game; please follow the instructions as follows:

- Left click on the item you want to move.

- Left click on the tile/block/box on the board where you want to place the item.

- If move is legal and follow the rules, the selected item will be moved to the destination box.

- If move is illegal, a pop-up window will be displayed; displaying the reason why the move was unsuccessful.