Project Report

Design Patterns

MVC (Model-View-Controller)

The structure follows a MVC pattern where SR serves as the controller managing game state and logic, GameMenu/StartMenu as the view handling UI rendering, and HexStone/Hexagon as models encapsulating game data and operations.

SR - class serves as the entry point (main method) and controls the initialization (init method) of the game.
It manages the game board (board), tracks changeability and locking of board positions (isChangeable and isLocked arrays), and maintains game state variables (gameover, time_left, curr, etc.).
Also sets up the main JFrame for the game window.

GameMenu/StartMenu - class extends JPanel and utilizes paintComponent to display GUI as well as handle corresponding events.

HexStone - class possesses both hexagon and stone properties, storing its type (empty hexagon, stone occupied, hexagon not on the board), location on the board and also having an ArrayList of all neighboring hexstones (including ones not on the board) and valid neighboring hexstones (only including neighbors that are on the board).

Techniques

Hexagonal Grid Representation: Efficiently manage and draw a hexagonal game board using geometric calculations for mouse interaction and rendering.

```
private int[] getXP(int centerX, int centerY, int size){
   int[] xPoints = new int[6];

   // Angle for each vertex of the hexagon (in radians)
   double[] angles = {Math.PI / 6, Math.PI / 2, 5 * Math.PI / 6, 7 * Math.PI / 6, 3 * Math.PI / 2, 11 * Math.PI /
6};

   for (int i = 0; i < 6; i++) {
       xPoints[i] = (int) (centerX + size * Math.cos(angles[i]));
   }
   return xPoints;
}

private int[] getYP(int centerX, int centerY, int size){
   int[] yPoints = new int[6];

   // Angle for each vertex of the hexagon (in radians)
   double[] angles = {Math.PI / 6, Math.PI / 2, 5 * Math.PI / 6, 7 * Math.PI / 6, 3 * Math.PI / 2, 11 * Math.PI /
6};

   for (int i = 0; i < 6; i++) {
       yPoints[i] = (int) (centerY + size * Math.sin(angles[i]));
   }
   return yPoints;
}
```

```
//utilizes raycasting algorithm
public boolean withinHexagon(int x, int y, int[] hexagonX, int[] hexagonY) {

   int sides = 6;
   boolean inHexagon = false;

   for (int i = 0, j = sides - 1; i < sides; j = i++) {
       if (((hexagonY[i] > y) != (hexagonY[j] > y)) &&
               (x < (hexagonX[j] - hexagonX[i]) * (y - hexagonY[i]) / (hexagonY[j] - hexagonY[i]) + hexagonX[i])) {
           inHexagon = !inHexagon;
       }
   }

   return inHexagon;

}
```

Additionally, cube coordinates inspired by (https://www.redblobgames.com/grids/hexagons-v1/) were considered for more complex algorithms involving HexStones further away from each other and potentially a hashing-like method for determining relations using coordinates x, y, z. This implementation would have taken too long and was not continued, but can be expanded on in the future.

Attempts at more efficient initializations of the board were made by starting with the center hexagon and recursively creating its neighbors to create a tree-like structure going from inner to outer, constructing neighbor HexStone arrays in the constructor of each HexStone. The attempt created a StackOverflow error and also had a duplicate HexStone issue that was not further examined.

Methods and Experimental Results

Robot 1 uses a greedy algorithm approach as it makes a short-sighted, optimal move each step that loops through each available move and chooses the one that results in the most flips or the most (blue) stones for the robot. This often covers one of the author's considerations, which is to win the battle for individual hex rings by placing the six piece of the hex ring.

Robot 2 implements a simplified version of the considerations given by the author. The consideration looked at is "to attempt to play the last piece". This means that the robot should try to not play a move that puts the opponent in a position to complete a hex ring. Thus, this robot is implemented by storing all hex rings on the board and not allowing the move if it puts the hex ring in a state to be completed (unless forced to).

Testing both methods showed generally random results, robot 1 was not able to really take control of areas as it often one by one places in the top area of the board. Robot 2, dependent on the opponents actions, can perform similarly to Robot 1, but attempts to deny opponents the ability to complete hex rings. This appears to result in higher successes in Robot 2.

Persistence Implementation

Save and load feature captures the game state to a .json file through the use of external library org.json.

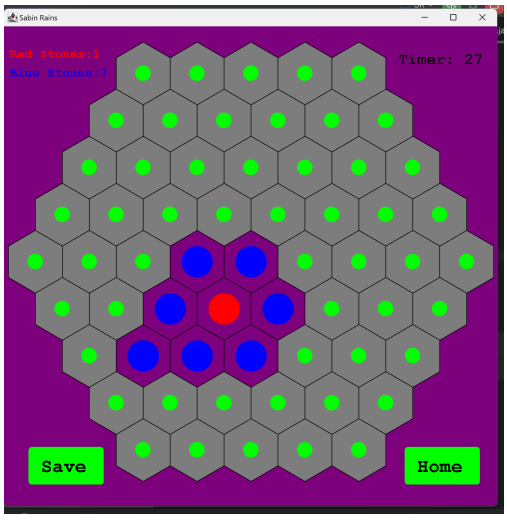
Writable - interface implemented by SR and HexStone that has the toJson() method which returns a JSONObject including the class attributes for the static SR and each HexStone object.

JsonReader - class represents a reader that reads SR information from JSON data stored in file

JsonWriter - class represents a writer that writes JSON representation of SR to file

Issues: Program currently marks any untaken area as locked when the file is loaded in. The issue is resolved after one move. Issue is probably caused by an unordered call of functions which does not account for the Lock check as in the next iteration of the checks, the problem is resolved.

Incorrect IsLocked array right after load:



Resolved: Error was in JsonReader, Object array was iterated incorrectly.

This persistence implementation can be extended to save and load games with robots. Moreover, multiple games can be stored and loaded using different files to have multiple saves each in their separate unique .json file.

References Used

SabinRains(2).jar was used as a guide for modeling of the GUI and event-driven programming