Marcia Gallant

Single Responsibility Principle:

- Separated the responsibility of updating the crate's position on the board from GameBoard to its own class: CratePositionUpdater. GameBoard responsibility is setting up the gameboard. It violates the single responsibility principle because it has another responsibility of updating the crate position. So, based on the single responsibility principle, the responsibility of updating the crates position when it is pushed by a turtle should be in its own class. I plan to solve it by creating a new class that has the responsibility of updating the crates position. This resolves the single responsibility principle because if I need to change how the gameboard is setup, I only have to change code in the gameboard class and if we need to change how the crate position is updated after a turtle moves it, I only need to change code in the CratePositionUpdater class.
- Separated the responsibility of setting all the neighbours for all the tiles in the gameboard to its own class: TileAttacher. The gameboard's responsibility is to set up the gameboard, so, having another responsibility of setting all the tile neighbours violates the single responsibility principle. So, I am creating a new class for the responsibility of setting all the tile neighbours. The solves the violation because if the way the gameboard is set up changes then we only have to worry about the gameboard class and if the way the tiles neighbours are set up changes, I only have to worry about the TileAttacher class.
- Separated the responsibility of updating the Robot turtle position to its own class: RobotTurtlePositionUpdater. The GameBoard's responsibility is setting up the gameboard, so, having the responsibility of updating robot turtles positions on the gameboard in that class violates the single responsibility principle because a class should not have more than one responsibility. I am separating it from the GameBoard class and making a new class for it, so, if the way we update the turtles position on the gameboard changes, we only have to worry about changing the RobotTurtlePositionUpdater.
- Separated the responsibility of calculating what the new position if one exists from the Robot Turtle class which deals with keeping track of the turtles position and made a new class RobotTurtlePosition calculator for the responsibility. The Robot Turtle class responsibility is to store information about the robot turtle. It also having the responsibility of keeping track of the turtles position violates the single responsibility principle. I am going to create a new class RobotTurtlePositionCalculator that will be responsible for calculating the new position if it exists. This solves the single responsibility principle because now if we have to change how we calculate the new position; we only have to worry about the RobotTurtlePositionCalculator class and if we need to change how the GameBoard is set up; we only have to worry about the GameBoard class.
- Separated the responsibility of checking if the turtle was able to move successfully (meaning no object or such was in the way) from the GameBoard class and made its own class: GameBoardMoveChecker. The gameboard having two different responsibilities violates Single responsibility principle. I am going to create a new class GameBoardMoveChecker that will be responsible for checking if the turtle can move successfully. This solves the single responsibility principle because now if we have to change how we check if the turtle can move; we only have

to worry about the GameBoardMoveChecker class and if we need to change how the GameBoard is set up; we only have to worry about the GameBoard class.

- I separated the responsibility of finding out what the invalid positions for crates and stone walls are from the GameBoardModel to its own class InvalidPositionCalculator. The GameBoardModel having two different responsibilities violates Single responsibility principle. The GameBoardModel is responsible for initializing all the different components of the game. I am going to create a new class InvalidPositionCalculator that will be responsible for determining what the invalid positions for crates and stone walls are. This solves the single responsibility principle because now if we have to change how we determine what the invalid positions are; we only have to worry about the InvalidPositionCalculator class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
- I separated the responsibility of how we moved that turtle from the GameBoardModel to its own class ObjectMover. The GameBoardModel having two different responsibilities violates Single responsibility principle. I am going to create a new class ObjectMover that will be responsible for moving the correct turtle according to the card chosen if possible. This solves the single responsibility principle because now if we have to change how we move the turtle; we only have to worry about the ObjectMover class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
- I separated the responsibility of creating Robot Jewels from the GameBoardModel to its own class RobotJewelCreator. The GameBoardModel having two different responsibilities violates Single responsibility principle. I am going to create a new class RobotJewelCreator that will be responsible for creating Robot Jewels. This solves the single responsibility principle because now if we have to change how we create Robot Jewels; we only have to worry about the RobotJewelCreator class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
- I separated the responsibility of creating Robot Turtles from the GameBoardModel to its own class RobotTurtleCreator. The GameBoardModel having two different responsibilities violates Single responsibility principle. I am going to create a new class RobotTurtleCreator that will be responsible for creating Robot Turtles. This solves the single responsibility principle because now if we have to change how we create Robot Turtles; we only have to worry about the RobotTurtleCreator class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
- I separated the responsibility of how we create card decks from the GameBoardModel to its own class CardDeckCreator. The GameBoardModel having two different responsibilities violates Single responsibility principle. I am going to create a new class CardDeckCreator that will be responsible for creating card decks for them game. This solves the single responsibility principle because now if we have to change what card should be in the card deck; we only have to worry about the CardDeckCreator class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
- I separated the responsibility of finding the next player from the GameBoardModel to its own class PlayerSwitcher. The GameBoardModel having two different responsibilities violates Single

responsibility principle. I am going to create a new class PlayerSwitcher that will be responsible for determining the next player. This solves the single responsibility principle because now if we have to change how determine the next player; we only have to worry about the PlayerSwitcher class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.

- I separated the responsibility of how we check which players won the game from the GameBoardModel to its own class WinChecker. The GameBoardModel having two different responsibilities violates Single responsibility principle. I am going to create a new class WinChecker that will be responsible for checking which players won the game. This solves the single responsibility principle because now if we have to change how we determine if a player won the game; we only have to worry about the WinChecker class and if we need to change how the GameBoardModel initializes the components of the game; we only have to worry about the GameBoardModel class.
-

Open Closed Principle:

- I also changed the way we look for invalid positions by storing all the positions of robot turtles and jewels in a single array and it can now be looped through and it adds the invalid positions. This follows the open closed principle and makes it easier to extend if we ever needed to have more invalid positions at the start. We used to look for invalid positions by having multiple if statements that look like:

```
invalidXPos = new int[numPlayers*2]; //since each player needs a turtle and a jewel
  invalidYPos = new int[numPlayers*2];

  if (numPlayers >= 1)
  {
        invalidXPos[index] = TURTLE_ONE_POSITION[0];
        invalidYPos[index] = TURTLE_ONE_POSITION[1];
        invalidXPos[index+1] = JEWEL_ONE_POSITION[0];
        invalidYPos[index+1] = JEWEL_ONE_POSITION[1];
        index = index + 2;
  }
```

  This violates the open closed principle because if we allow for more players then we will need more than 4 if statements. I am going to change the code in InvalidPositionCalculator to put it in a for loop with no if statements. This will follow open closed principle because I will not have to add additional if statements if we increase the amount of players that can play the game. I wrote a for loop that continues only if I < numPlayers, so, the right number of invalid positions are determined.

- I changed RobotJewelCreator code which used to have multiple if statements like:

```
if (numPlayers >= 1)
{
  //int x, int y, String c
  gameBoard.addRobotJewel(JEWEL_ONE_POSITION, JEWEL_ONE_COLOUR);

}
if (numPlayers >= 2)
{
```

```
        gameBoard.addRobotJewel(JEWEL_TWO_POSITION, JEWEL_TWO_COLOUR);
    }
    if (numPlayers >= 3)
    {
        gameBoard.addRobotJewel(JEWEL_THREE_POSITION, JEWEL_THREE_COLOUR);
    }
    if (numPlayers >= 4)
    {
        gameBoard.addRobotJewel(JEWEL_FOUR_POSITION, JEWEL_FOUR_COLOUR);
    }
```

It violates the open closed principle because if we need to extend our program to have more players, we need to add more if statements. I changed it to a for loop that loops numPlayers times. This solves the open closed principle violation because now we do not have to add if statements if we extend our program to have more players.

- I changed RobotTurtleCreator code in exactly the same way as RobotJewelCreator (getting rid of if statements and replacing them with a for loop). It violated the open closed principle the same way as RobotJewelCreator and was solved in the same way.
- I changed how to we create cards; the code used to be:

```
Card turnLeft = new Card(cardTypes[0], cardColours[0]);
 Card stepForward = new Card(cardTypes[1], cardColours[1]);
 Card turnRight = new Card(cardTypes[2], cardColours[2]);
 Card bug = new Card(cardTypes[3], cardColours[3]);
 cardDeck[0] = turnLeft;
 cardDeck[1] = stepForward;
 cardDeck[2] = turnRight;
cardDeck[3] = bug;
```

This violated open closed principle because we have to constantly create new cards and store them in a new spot in the array. Now, it is in a for loop which it creates 1 card and adds it to the deck per loop. Now, it is easily extendable because all we have to do is add a new string to cardTypes[] and to cardColours[] and we added a new card to the deck.

- I changed Old Playerswitcher code:

```
if (numPlayers > 1)
        {
                if (currentPlayer == currentPlayer.PLAYER_ONE)
                {
                        int next = findNextPlayer(0);
                        if (next == 1) //player 2
                        {
                                currentPlayer = currentPlayer.PLAYER_TWO;
                        }
                        else if (next == 2) //player 3
                        {
                                currentPlayer = currentPlayer.PLAYER_THREE;
                        }
                        else if (next == 3) //player 4
                        {
                                currentPlayer = currentPlayer.PLAYER_FOUR;
                        }
```

```
                    //else current player doesn't change
        }
```

It had more if statements for currentPlayer =2, 3, 4. This violates open closed principle because if we add more players, we will need to add more if statements. So, I put it in a for loop which loops through all the players to determine whose turn just passed and then finds the next player and returns it. This solves the principle because we do not need to add more if statements if we add more players into our game.

- displayNotificationOnWhoseTurnItIs method in the GameBoardController violated the open-closed principle:

```
protected void displayNotificationOfWhoseTurnItIs()
{
        Player currentPlayer = model.getCurrentPlayer();
        if(currentPlayer == Player.PLAYER_ONE)
        {
                view.changePlayersTurnIndicatorLabel(1);
        }
        else if(currentPlayer == Player.PLAYER_TWO)
        {
                view.changePlayersTurnIndicatorLabel(2);
        }
        if(currentPlayer == Player.PLAYER_THREE)
        {
                view.changePlayersTurnIndicatorLabel(3);
        }
        if(currentPlayer == Player.PLAYER_FOUR)
        {
                view.changePlayersTurnIndicatorLabel(4);
        }
}
```

  so if more players must be added into the game, I would have to add more if statements. I changed it to a for loop, so, now I just need to add a new Player enum value to pList in that method to extend it. It follows the open-closed principle now because we no longer have to add if statements for each possibility of the number of players.

- Similarly to displayNotificationOnWhoseTurnItIs method, validateCardChoice method in the GameBoardController violated the open-closed principle so if more players must be added into the game, I would have to add more if statements. I changed it to a for loop, so, now I just need to add a new Player enum value to pList in that method to extend it. It follows the open-closed principle now because we no longer have to add if statements for each possibility of the number of players.

Liskov Substituion Principle:

- drawCurrentGame has been changed so it no longer has to check each tileType, now it only must tell if it's a normal tile or not. The reason why it must still check if it's a normal tile or not is because there is a different normal tile image for each square on the board whereas the other tiles mostly only have 1 version and can go on any square or have a couple versions based on which direction they're facing and can still be on any square on the gameboard. In order for my program to work, the normal tiles have identifier numbers which correspond to what spot they are supposed to be in which none of the other tiles have. In order to make the tiles and other

objects drawn on the gameboard in only one method would mean I would have to add identifier numbers to each object and have one for each square even though the pictures would be all the same for the objects. This would bloat the imgs folder and seems like unnecessary work. Therefore, I've made a design trade-off to check for the normal tiles or not. It still mostly follows the Liskov substitution principle where all of the subtypes of the Tile class do not need to be known and it is still relatively easy to extend since if I add another tile type, I do not need to check its type.

Interface Segregation Principle ISP

- I've added interfaces such as modelDataInterface which holds all the methods the classes in the controller package needs. ControllerModelDataInterface holds all the methods from the GameBoardModel which the controller needs and ConverterModelDataInterface holds all the methods from GameBoardModel which the converter needs. This useful because it's not violating ISP anymore since we are not forcing the controller package classes to depend on any more methods than what they need.

Dependency Inversion Principle

- The controller classes used to depend on all the model classes which violates the dependency inversion principle. I added interfaces, ModelDataInterface and ConverterModelDataInterface to fix this. Now, it is not violating the dependency inversion principle because the controller classes are not depending on all the model classes but instead are depending on the interface instead. This means the change created a dependency inversion which means it no longer has a compile time dependency and only has a runtime dependency. That way it doesn't have to depend on all the small details.
- I've also added an interface called displayInterface which the controller now depends on instead of GameBoardDisplay. This is because it used to violate the dependency inversion principle because the GameBoardController depended on all the small details included in the view classes. Now that the GameBoardController no longer depends on anymore than necessary and it creates a dependency inversion.

Design Decisions:

- I kept checkIfObjectPresent since that is just a helper method to creating new stone walls and crates which is the responsibility of the GameBoard class inside the GameBoard class.
- I also made a design decision to keep robotJewelClaimed method in the GameBoard class because it is highly unlikely to change and quite overkill to make a class just for a method that makes it so two turtles cannot acquire the same jewel.
- ObjectMover responsibility is to make sure the turtle and any other objects are moved as desired or not moved if it's not possible. A design trade off that I made is that we need if statements to implement different functionality for all of the different cards. So, it's not really following the open-closed principle since if any cards are to be added we need new if statements to implement their functionality but this is a crucial part of the game. Each card does something different, so, if statements are necessary for our game to work.