Name: Linhao He
Collaborator: Jack Williams

MVC refactoring

1. Comments
2. Reduency
3. Encaplication
4. Naming conversion

I added comments for all functions to give a basic understanding of each method, then refactor the move function in rowgamecontroller.java because most of its code is redundant and able to compress to a for a loop. Second, the classes break encapsulation, so I changed them to privates and created setter and getter methods for each variable. Last, many functions use constant numbers without context, then for each constant number, I declare them as static constants with the corresponding name that gives some information.

Composite design pattern

Since we are applying the composite design pattern to the view, we need to declare 3 different classes that all implement RowGameView interfaces.

In RowGameBoardView (Component A) it will have variables

```
private final JButton[][] blocks;
```

A constructor that accepts Gui, initializes blocks, then assigns the block to gui.

And method

```
public void update(RowGameModel gameModel) {
   RowBlockModel[][] blockdata = gameModel.getBlocksData();
   for (all blocks do){
     blocks[row][column].setText(blockdata[row][column].getContents());
     blocks[row][column].setEnabled(blockdata[row][column].getIsLegalMove());
   }
}
```

The update function of RowGameBoardView.java will accept an existing RowGameModel, then get the NxN grid of `gameModel` through the `gameModel.getBlocksData()` function.

For each individual index of `blocks`, `setText` parameter to the `Contents` of `blockdata`

```
blocks[row][column].setText(blockdata[row][column].getContents());
```

For each individual index of **blocks**, `setEnabled` parameter to the `IsLegalMove` of `blockdata`.

```
blocks[row][column].setEnabled(blockdata[row][column].getIsLegalMove());
```

In RowGameStatusView (Component C) it will have variables and functions

```
private final JTextArea playerturn;
```

A constructor that accepts Gui, initializes `playerturn`, then assigns the `playerturn` to GUI.

The update function will accept an existing RowGameModel, then set the `playerturn` text to the return value of `getFinalResult()`.

```
public void update(RowGameModel gameModel) {
   this.playerturn.setText(gameModel.getFinalResult());

}
```

The RowGameGUI class will now have a new private collection of views, an update function that triggers all update functions in each component in the list. AddComp(component) adds a component to the list and RemoveComp(component) removes components from a list.

Observer design pattern

The observable field is the event listener added to each button in RowGameGUI; when it's clicked it will trigger the move function in RowGameController, causing updates in other components.

The Java Swift object that corresponds to the observer is `addActionListener` It causes the block to trigger the function whenever it's triggered.

The update method is a move function in RowGameController; it will update the entire grid and its corresponding components assuming the player places a valid move, then call the updateBlock and setPlayerTurnText() functions in RowGameGUI to update the view that is affected by the move function.