Criterion C

The recipe program, named "**Salt and Pepper Recipes,**" uses the class structure shown in UML diagram in Criterion B. **MainFrame** controls most of the data processing and all the panels are associated with **MainFrame**. The **"Recipe" Object**, is used to standardize recipes for use within the program. A **"Recipe" ArrayList** is used within the MainFrame to keep track of what recipe is selected and what recipe to do operations on. The following are some components of the program.

Figure 1: MainFrame Variables

```
1   import java.awt.BorderLayout;
2   import java.io.*;
3   import java.util.ArrayList;
4   import javax.swing.*;
5
6
7   public class MainFrame extends JFrame {
8       private ViewPanel viewPanel;
9       private AddPanel addPanel;
10      private EditPanel editPanel;
11      private ScalePanel scalePanel;
12      private SortPanel sortPanel;
13      private HelpPanel helpPanel;
14      private Toolbar toolbar;
15      private JPanel currentPanel;
16
17      private ArrayList<Recipe> recipeArr = new ArrayList<~>();
18      private int recipePosition;
```

The program uses **Java Swing** for GUI and **MainFrame** itself uses **BorderLayout**. This allows for **MainFrame** to replace its **CENTER** in the layout with respective panels to make it easy for switching. Instead of instantiating a **JFrame** within **MainFrame**, **MainFrame** extends **JFrame** in order to obtain all the necessary methods for the Frame without having to redundantly append the name of the Frame before the method. Important imports also include **java.io.File** which will be used for recipe text files.

Figure 2: MainFrame Initial Method Calls

```
37          currentPanel = viewPanel; //Set the panel that is currently showing
38
39          updateRecipeArr();
40
41          recipePosition = 0;
42          updateViewPanel();
```

```
230     private void updateRecipeArr() {
231         File dir = new File( pathname: "Recipes");
232
233         File[] txtFiles = dir.listFiles(new FilenameFilter() {
234             public boolean accept(File dir, String name) { return name.endsWith(".txt"); }
237         });
238
239         for (File f : txtFiles) {
240             addRecipeToArr(f,  add: true);
241         }
242     }
243
244     private void updateViewPanel() {
245         if(recipeArr.size() != 0)
246             viewPanel.updateViewPanel(recipeArr.get(recipePosition));
247         else
248             viewPanel.updateViewPanel( r: null);
249     }
```

MainFrame calls **updateRecipeArr()** and **updateViewPanel()** initially in order to update the **MainFrame** with any recipe text files that have been previously saved. Since all stored program data is deleted when the program is closed, these methods repopulate **recipeArr** with the recipes found in the **Recipes** folder. The method generates an array of **Files** with **dir.listFiles** and filters for text files with the ending ".txt". It then uses a for-each loop, sending each file to **addRecipeToArr()** to be added. **updateViewPanel()** updates the **ViewPanel** with the current recipes by sending the recipe in **recipeArr** at the pointer instantiated by **recipePosition**. **recipePosition** allows the **MainFrame** to know exactly what recipe is being interacted with no matter what the user is doing.

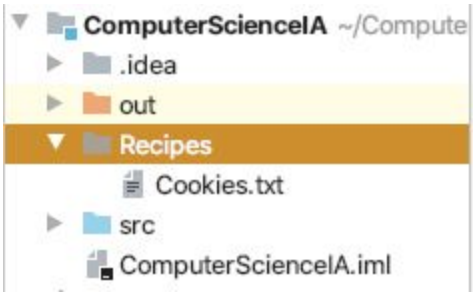
Figure 3: addRecipeToArr() Method

```
251     private void addRecipeToArr(File f, boolean add) {
252         try {
253             BufferedReader reader = new BufferedReader(new FileReader(f));
254             String readLine, title;
255             String intro = "";
256             String directions = "";
257
258             reader.readLine(); //Title
259             title = reader.readLine();
260
261             Recipe newRecipe = new Recipe(title);
262
263             reader.readLine(); //Introduction
264             readLine = reader.readLine();
265             while (!readLine.equals("Ingredients")) { //Reading introduction
266                 intro += readLine + "\n";
267                 readLine = reader.readLine();
268             }
269
270             newRecipe.addIntroduction(intro);
271
272             readLine = reader.readLine();
273             while (!readLine.equals("Directions")) { //Reading ingredients
274                 String[] splitStr = readLine.split( regex: "--",  limit: 2);
275                 if(splitStr.length == 2)
276                     newRecipe.addIngredient(splitStr[0], splitStr[1]);
277                 readLine = reader.readLine();
278             }
279
280             readLine = reader.readLine();
281             while (readLine != null) { //Reading directions
282                 directions += readLine + "\n";
283                 readLine = reader.readLine();
284             }
285
286             newRecipe.addDirections(directions);
287
288             reader.close();
289
290             if(add) recipeArr.add(newRecipe);
291             else {
292                 recipeArr.set(recipePosition, newRecipe);
293             }
294         }
295         catch(Exception e) {
296             System.out.println(e + " in addRecipeToArr");
297         }
298     }
```

addRecipeToArr() adds recipes to the array and replaces recipes in the array. It takes the **File f** as a parameter as well as a boolean add. “add” is true if the File is a new recipe and has to be added and is false when it has to be replaced because the File has changed. **addRecipeToArr()** uses **BufferedReader** to read the files. Each file is written in a standard way, so there are while loops to read each part

of the recipe until it hits another heading (“Title”, “Introduction”, “Ingredients”, etc). When reading directions, each line is split into the ingredient and the amount and added to the **Recipe**. Each direction is standardized so to separate the ingredient from the amount with “==”. The final **Recipe** object is either added to the end of **recipeArr** or replaced.

Figure 4: Recipes Folder and Example Recipe File




```
1  Title
2  Cookies
3  Introduction
4  A simple introduction to the recipe about Cookies.
5  Ingredients
6  3 Cups==Flour
7  2==Eggs
8  3 Cups==Sugar
9  Directions
10 1. Mix together sugar, butter, and eggs until smooth
11 2. Incorporate flour taking care not to over mix.
```

Figure 5: ViewPanel updateViewPanel() method

```
14  public class ViewPanel extends JPanel implements ActionListener {

40      public void updateViewPanel(Recipe r) {
41          textArea.setText(null);
42
43          if(r != null) {
44              textArea.append("----------------\n");
45              textArea.append("Title");
46              textArea.append("\n----------------\n");
47              textArea.append(r.getTitle());
48              textArea.append("\n\n----------------\n");
49              textArea.append("Introduction");
50              textArea.append("\n----------------\n");
51              textArea.append(r.getIntroduction());
52              textArea.append("\n----------------\n");
53              textArea.append("Ingredients");
54              textArea.append("\n----------------\n");
55              textArea.append(r.getIngredients());
56              textArea.append("\n----------------\n");
57              textArea.append("Directions");
58              textArea.append("\n----------------\n");
59              textArea.append(r.getDirections());
60          }
61          else {
62              textArea.append("There are no recipes to display.");
63          }
64
65          textArea.setCaretPosition(0);
66      }
```

With this method, **MainFrame** is able to pass a **Recipe** object to **ViewPanel**, and **ViewPanel** is able to use get methods to append to appropriate **JComponents** and display the recipe on the GUI.

Figure 6: ViewPanel GUI Example

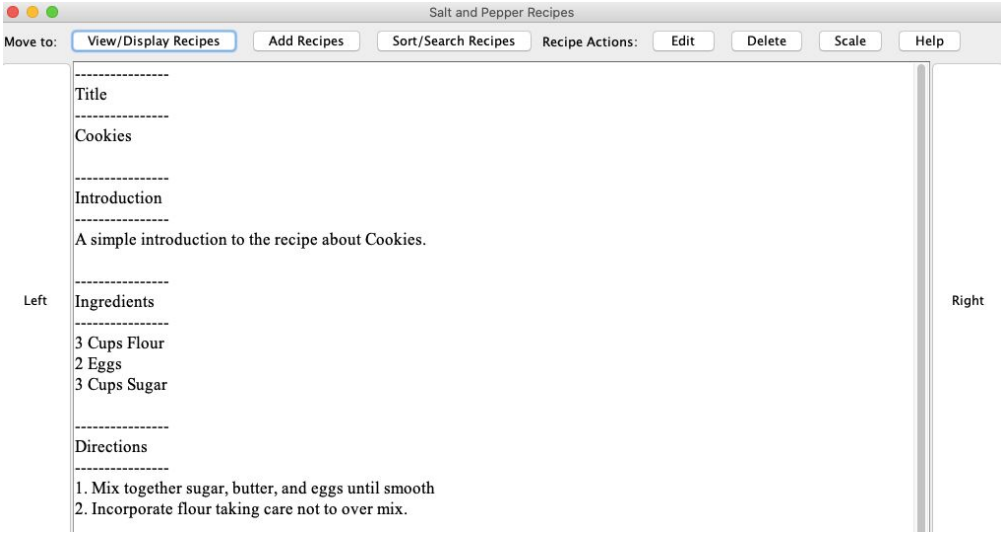


Figure 7: ToolbarListener Interface

```
3  public interface ToolbarListener {
4      public void changePanel(String panel);
5  }
```

Figure 8: Toolbar Class

```
20      private ToolbarListener buttonListener;
```

Figure 9: Toolbar Methods

```
56  public void setButtonListener(ToolbarListener buttonListener) { this.buttonListener = buttonListener; }
```

```
60  @   public void actionPerformed(ActionEvent e) {
61          JButton clicked = (JButton) e.getSource();
62
63          if(clicked == viewButton) {
64              if(buttonListener != null) {
65                  buttonListener.changePanel("viewPanel");
66              }
67          } else if(clicked == addButton) {
68              if(buttonListener != null) {
69                  buttonListener.changePanel("addPanel");
70              }
71          } else if(clicked == editButton) {
72              if(buttonListener != null) {
73                  buttonListener.changePanel("editPanel");
74              }
75          } else if(clicked == deleteButton) {
76              if(buttonListener != null) {
77                  buttonListener.changePanel("delete");
78              }
79          } else if(clicked == scaleButton) {
80              if(buttonListener != null) {
81                  buttonListener.changePanel("scalePanel");
82              }
83          } else if (clicked == sortButton) {
84              if(buttonListener != null) {
85                  buttonListener.changePanel("sortPanel");
86              }
87          } else if (clicked == helpButton) {
88              if (buttonListener != null) {
89                  buttonListener.changePanel("helpPanel");
90              }
91          }
92      }
93  }
```

As referenced in Criterion B, **Panel** classes use interface instances as one way to communicate with **MainFrame**. In this example, **ViewPanel** creates a

ToolbarListener variable with a method to set the **ToolbarListener**. **MainFrame** will use this method to set **buttonListener** in **Toolbar** and set **changePanel()**'s action. **Toolbar** can then communicate with **MainFrame** what panel has been clicked within the **actionPerformed** method. All **Panels** except **HelpPanel** have respective interfaces to help with their function.

Figure 10: MainFrame toolbar.setButtonListener()

```
44      toolbar.setButtonListener(new ToolbarListener() {
45          public void changePanel(String panel) {
46              JPanel changeToPanel = currentPanel;
47              switch(panel) {
48                  case "viewPanel":
49                      updateViewPanel();
50                      changeToPanel = viewPanel;
51                      break;
52                  case "addPanel":
53                      changeToPanel = addPanel;
54                      break;
55                  case "editPanel":
56                      if(currentPanel == viewPanel) {
57                          if (recipeArr.size() != 0) {
58                              changeToPanel = editPanel;
59                              editPanel.editPanel(recipeArr.get(recipePosition));
60                          } else {
61                              errorMessage( text: "There is no recipe to edit.");
62                              return;
63                          }
64                      }
65                      else
66                          informationMessage( text: "Move to \"View/Display Recipes\" Page to edit a recipe.");
67                      break;
```

```
68                  case "delete":
69                      if(currentPanel == viewPanel) {
70                          if(recipeArr.size() != 0) {
71                              try {
72                                  String[] options = {"Yes", "No"};
73                                  int n = JOptionPane.showOptionDialog( parentComponent: null,
74                                          message: "Are you sure you want to delete the recipe?",
75                                          title: "delete",
76                                          JOptionPane.YES_NO_OPTION,
77                                          JOptionPane.QUESTION_MESSAGE,
78                                          icon: null,
79                                          options,
80                                          options[0]);
81                                  if (n == JOptionPane.YES_OPTION) {
82                                      String recipePath = "Recipes/" + recipeArr.get(recipePosition).getTitle() + ".txt";
83                                      File deleteFile = new File(recipePath);
84                                      deleteFile.delete();
85                                      recipeArr.remove(recipeArr.get(recipePosition));
86                                      if (recipePosition != 0) recipePosition -= 1;
87                                      informationMessage( text: "Recipe has been deleted.");
88                                      updateViewPanel();
89                                  }
90                              } catch (Exception e) {
91                                  errorMessage( text: "Error: There is something wrong with files; Recipe could not be deleted.");
92                              }
93                          }
94                          else
95                              errorMessage( text: "Error: There is no recipe to delete.");
96                      }
```

```
97                  else
98                      informationMessage( text: "Move to \"View/Display Recipes\" Page to delete a recipe.");
99                  break;
100             case "scalePanel":
101                 if(currentPanel == viewPanel) {
102                     if (recipeArr.size() != 0) {
103                         changeToPanel = scalePanel;
104                     } else {
105                         errorMessage( text: "There is no recipe to scale.");
106                         return;
107                     }
108                 }
109                 else
110                     informationMessage( text: "Move to \"View/Display Recipes\" Page to scale a recipe.");
111                 break;
112             case "sortPanel":
113                 changeToPanel = sortPanel;
114                 break;
115             case "helpPanel":
116                 changeToPanel = helpPanel;
117                 break;
118         }
119
120         remove(currentPanel);
121         add(changeToPanel, BorderLayout.CENTER);
122         validate();
123         repaint();
124         currentPanel = changeToPanel;
125     }
126 });
```

Toolbar's implementation involves the use of a switch statement to switch between panels. **MainFrame**'s **currentPanel** tracks the panel that is currently shown. When **Toolbar**'s buttons are clicked, **Toolbar** passes a String of the panel that is clicked and **MainFrame** switches to the **Panel** with the use of local variable **changeToPanel** and removing the **currentPanel** and adding the new **Panel**. **Validate()** and **repaint()** refreshes **MainFrame**. Depending on the panel, different actions are taken to make sure the panel is shown correctly and no errors occur. **informationMessage()** and **errorMessage()** methods cause a pop up to appear, guiding users. The delete option shows the use of try and catch blocks throughout the program when dealing with Files. **Delete** shows a confirmation box and deletes the file and the object from **recipeArr** if confirmed. Edit, Delete, and Scale are made only to work if the user is currently on **ViewPanel** and looking at the recipe they want to act on.

Figure 11: MainFrame errorMessage() and informationMessage()

```
326 private void errorMessage(String text) {
327     JOptionPane.showMessageDialog( parentComponent: null, text, text, JOptionPane.ERROR_MESSAGE);
328 }
329
330 private void informationMessage(String text) {
331     JOptionPane.showMessageDialog( parentComponent: null, text, text, JOptionPane.INFORMATION_MESSAGE);
332 }
```

Figure 12: AddPanel()

```
27      public AddPanel() {
28          setLayout(new GridBagLayout());
29
30          panelLabel = new JLabel( text: "Add A Recipe");
31          titleLabel = new JLabel( text: "Title: ");
32          introductionLabel = new JLabel( text: "Introduction: ");
33          ingredientsLabel = new JLabel( text: "Ingredients: ");
34          directionsLabel = new JLabel( text: "Directions: ");
35
36          titleField = new JTextField();
37
38          introductionArea = new JTextArea();
39          ingredientsArea = new JTextArea();
40          directionsArea = new JTextArea();
41
42          confirmButton = new JButton( text: "Add Recipe");
43          clearButton = new JButton( text: "Clear");
44
45          clearText();
46
47          confirmButton.addActionListener( l: this);
48          clearButton.addActionListener( l: this);
49
50          ////////////GUI
51
52          gc.insets = new Insets( top: 0,  left: 40,  bottom: 20,  right: 40);
```

```
54              //Add Panel Label
55              gc.weightx = 2;
56              gc.weighty = 2;
57              gc.gridx = 0;
58              gc.gridy = 0;
59              gc.gridwidth = 5;
60
61              add(panelLabel, gc);
62              //Column 1
63              gc.gridwidth = 1; //reset gridwidth to default
64
65              gc.weightx = 1;
66              gc.weighty = 1;
67              gc.anchor = GridBagConstraints.LINE_END;
68
69              gc.gridx = 0;
70              gc.gridy = 1;
71              add(titleLabel, gc);
72
73              gc.gridx = 0;
74              gc.gridy = 2;
75              add(introductionLabel, gc);
76
77              gc.gridx = 0;
78              gc.gridy = 3;
79              add(ingredientsLabel, gc);
```

```
81          gc.gridx = 0;
82          gc.gridy = 4;
83          add(directionsLabel, gc);
84
85          //Column 2
86          gc.anchor = GridBagConstraints.LINE_START;
87          gc.weightx = 20;
88          gc.weighty = 2;
89          gc.fill = GridBagConstraints.BOTH;
90          //gc.gridwidth = 4;
91
92          gc.gridx = 1;
93          gc.gridy = 1;
94          add(titleField, gc);
95
96          gc.gridx = 1;
97          gc.gridy = 2;
98          add(new JScrollPane(introductionArea), gc);
99
100         gc.gridx = 1;
101         gc.gridy = 3;
102         add(new JScrollPane(ingredientsArea), gc);
103
104         gc.gridx = 1;
105         gc.gridy = 4;
106         add(new JScrollPane(directionsArea), gc);
107
```

```
108             //Buttons
109             gc.anchor = GridBagConstraints.CENTER;
110             gc.weightx = 5;
111             gc.weighty = 5;
112             gc.fill = GridBagConstraints.BOTH;
113
114             gc.gridx = 0;
115             gc.gridy = 5;
116             add(clearButton, gc);
117
118             gc.gridx = 1;
119             gc.gridy = 5;
120             add(confirmButton, gc);
121     }
```

AddPanel, **SortPanel**, **ScalePanel**, and **EditPanel** all use custom layouts with **GridBagLayout** and **GridBagContraints**.

Figure 13: AddPanel Methods and MainFrame Adding Recipe

```
125     public void clearText() {
126         titleField.setText(TITLE_DEFAULT);
127         introductionArea.setText(INTRODUCTION_DEFAULT);
128         ingredientsArea.setText(INGREDIENTS_DEFAULT);
129         directionsArea.setText(DIRECTIONS_DEFAULT);
130     }
131
132 @   public void actionPerformed(ActionEvent e) {
133         JButton clicked = (JButton)e.getSource();
134         if(clicked == confirmButton) {
135             if(buttonListener != null) {
136                 String title = titleField.getText();
137                 String intro = introductionArea.getText();
138                 String ingredients = ingredientsArea.getText();
139                 String directions = directionsArea.getText();
140
141                 buttonListener.addRecipe(title, intro, ingredients, directions);
142
143                 clearText();
144             }
145         }
146         else if(clicked == clearButton) {
147             clearText();
148         }
149     }
150 }
```

```
142 addPanel.setButtonListener(new AddRecipeListener() {
143     public void addRecipe(String title, String intro, String ingredients, String directions) {
144         String recipePath = "Recipes/" + title + ".txt";
145         File newRecipe = new File(recipePath);
146
147         try {
148             boolean created = newRecipe.createNewFile();
149             if(created) {
150                 writeToFile(newRecipe, title, intro, ingredients, directions);
151                 addRecipeToArr(newRecipe,  add: true);
152                 informationMessage( text: "Your recipe has been added.");
153             }
154             else {
155                 errorMessage( text: "Error: There is already a recipe with that name.");
156             }
157         }
158         catch (Exception e) {
159             errorMessage( text: "Error: There is something wrong with files; Recipe could not be created.");
160         }
161     }
162 });
```

AddPanel passes user inputs for creating a new recipe to **MainFrame** where **MainFrame** creates a new **File** with the recipe name as the name of the file. The file is written with **BufferedWriter** in the **writeToFile()** method and added to **recipeArr** with **addRecipeToArr**. Messages are displayed along the way.

Figure 14: MainFrame writeToFile() method

```
300     private void writeToFile(File recipeFile, String title, String intro, String ingredients, String directions) {
301         try {
302             BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(recipeFile,  append: false));
303
304             bufferedWriter.write( str: "Title");
305             bufferedWriter.newLine();
306             bufferedWriter.write(title);
307             bufferedWriter.newLine();
308             bufferedWriter.write( str: "Introduction");
309             bufferedWriter.newLine();
310             bufferedWriter.write(intro);
311             bufferedWriter.newLine();
312             bufferedWriter.write( str: "Ingredients");
313             bufferedWriter.newLine();
314             bufferedWriter.write(ingredients);
315             bufferedWriter.newLine();
316             bufferedWriter.write( str: "Directions");
317             bufferedWriter.newLine();
318             bufferedWriter.write(directions);
319             bufferedWriter.close();
320         }
321         catch (Exception e) {
322             System.out.println(e + " in writeToFile");
323         }
324     }
```

Figure 15: Recipe class

```
3   public class Recipe {
4       private String title, introduction, directions;
5       private Map<String, String> ingredients = new HashMap<String, String>();
6
7       public Recipe(String title) { this.title = title; }
10
11      public void addIntroduction(String introduction) { this.introduction = introduction; }
12
13      public void addDirections(String directions) { this.directions = directions; }
14
15      public void addIngredient(String amount, String ingredient) { ingredients.put(ingredient, amount); }
18
19      public String getTitle() { return title; }
20
21      public String getIntroduction() { return introduction; }
22
23      public String getDirections() { return directions; }
24
25      public String getIngredients() {
26          String ingredientStr = "";
27          for(Map.Entry<String, String> entry : ingredients.entrySet()) {
28              ingredientStr += entry.getValue() + " " + entry.getKey() + "\n";
29          }
30          return ingredientStr;
31      }
```

```
33      public String getIngredientsForEdit() {
34          String ingredientStr = "";
35          for(Map.Entry<String, String> entry : ingredients.entrySet()) {
36              ingredientStr += entry.getValue() + "--" + entry.getKey() + "\n";
37          }
38          return ingredientStr;
39      }
40
41      public void scale(boolean scaleUp, int amount) {
42          for(Map.Entry<String, String> entry : ingredients.entrySet()) {
43              String newIng, newAmount;
44              int oldAmount;
45              String[] splitIng = entry.getValue().split( regex: " ",  limit: 2);
46
47              if(splitIng.length == 2) {
48                  newIng = " " + splitIng[1];
49                  oldAmount = Integer.parseInt(splitIng[0]);
50              }
51              else {
52                  newIng = "";
53                  oldAmount = Integer.parseInt(entry.getValue());
54              }
55
56              if(scaleUp) {
57                  newAmount = String.valueOf(oldAmount*amount);
58              }
59              else {
60                  newAmount = String.valueOf(oldAmount/amount);
61              }
62
63              ingredients.replace(entry.getKey(), newAmount + newIng);
64          }
65      }
66
67      public String amountOfIngredient(String ingredient) {
68          if(ingredients.containsKey(ingredient)) {
69              return ingredients.get(ingredient);
70          }
71          else
72              return null;
73      }
74  }
```

Recipe uses a **HashMap** for the ingredients to make for easier processing while sorting. The key is the ingredient and the value is the amount. A **HashMap** entry loop is used to loop through all the entries and get keys and values. **scaleUp()** works by iterating through every entry and splitting each entry's value if there is a space in case the user added a unit to the amount (ex. "3 Cups" instead "3"). The number string is retained and parsed into an int. The amount is multiplied or divided and is converted back into a String and replaced in the **HashMap**.

Figure 16: SortPanel()

```
19      private ArrayList<Recipe> recipeArr;

109 @   public void updateRecipeArr(ArrayList<Recipe> recipeArr) { this.recipeArr = (ArrayList<Recipe>)recipeArr.clone(); }

128     public void sortArray() {
129         String typeStr = typeBox.getSelectedItem().toString();
130         String searchStr = nameField.getText();
131
132         if(typeStr.equals("Ingredient")) {
133             String noIngredient = "";
134
135             displayArea.setText(null);
136             for(Recipe r : recipeArr) {
137                 String amountStr = r.amountOfIngredient(searchStr);
138                 if(r.amountOfIngredient(searchStr) != null) {
139                     displayArea.append("\"" + r.getTitle() + "\" requires " + amountStr + " " + searchStr + "\n");
140                 }
141                 else
142                     noIngredient += "\"" + r.getTitle() + "\" does not require " + searchStr + "\n";
143             }
144
145             displayArea.append(noIngredient);
146
147
148         } else if (typeStr.equals("Recipe")) {
149             boolean swapped;
150             int n = recipeArr.size();

152             for(int i = 0; i<n-1; i++) {
153                 swapped = false;
154                 for(int j = 0; j<n-i-1; j++) {
155                     String firstTitle = recipeArr.get(j).getTitle();
156                     String secondTitle = recipeArr.get(j+1).getTitle();
157
158                     if(Math.abs(searchStr.compareTo(firstTitle)) > Math.abs(searchStr.compareTo(secondTitle))) {
159                         Recipe temp = recipeArr.get(j);
160                         recipeArr.set(j, recipeArr.get(j+1));
161                         recipeArr.set(j+1, temp);
162                         swapped = true;
163                     }
164                 }
165
166                 if(swapped == false) {
167                     break;
168                 }
169             }
170
171             displayArea.setText(null);
172             for(Recipe r : recipeArr) {
173                 displayArea.append(r.getTitle() + "\n");
174             }
175
176         }
177     }
178 }
```

SortPanel clones **recipeArr** from **MainFrame** and sorts the **ArrayList** with a Bubble Sort. **compareTo()** is used to organize recipe titles lexicographically. The absolute value is taken so that the value 0 from **compareTo()** (which means Strings are equal) is pushed to the front.

Figure 17: MainFrame sortPanel.setButtonListener()

```
219         sortPanel.setButtonListener(new SortRecipeListener() {
220             public void sortRecipe() { sortPanel.updateRecipeArr(recipeArr); }
223         });
```

Figure 18: ScalePanel Scale Function

```
92  @   public void actionPerformed(ActionEvent e) {
93          JButton clicked = (JButton)e.getSource();
94
95          if(clicked == confirmButton) {
96              if(buttonListener != null) {
97                  int amount;
98                  String amountStr = amountField.getText();
99                  String typeStr = typeBox.getSelectedItem().toString();
100                 boolean scaleUp;
101
102                 if (typeStr.equals("Up"))
103                     scaleUp = true;
104                 else
105                     scaleUp = false;
106
107                 if (amountStr == null) {
108                     String text = "There is no amount inputed";
109                     JOptionPane.showMessageDialog( parentComponent: null, text, text, JOptionPane.ERROR_MESSAGE);
110                     return;
111                 }
112
113                 try {
114                     amount = Integer.parseInt(amountStr);
115                 } catch (Exception ex) {
116                     String text = "Amount inputed is not valid; make sure it is an integer.";
117                     JOptionPane.showMessageDialog( parentComponent: null, text, text, JOptionPane.ERROR_MESSAGE);
118                     return;
119                 }
120
121                     buttonListener.scaleRecipe( submit: true, scaleUp, amount);
122                 }
123             }
124             else if(clicked == cancelButton) {
125                 if(buttonListener != null) {
126                     buttonListener.scaleRecipe( submit: false,  scaleUp: true,  amount: 0);
127                 }
128             }
129         }
130 }
```

ScalePanel checks the user inputted amount with a try and catch block to make sure the value is within integer range.

Figure 19: MainFrame scalePanel.setButtonListener()

```
188         scalePanel.setButtonListener(new ScaleRecipeListener() {
189             public void scaleRecipe(boolean submit, boolean scaleUp, int amount) {
190                 if(submit) {
191                     Recipe currRecipe = recipeArr.get(recipePosition);
192                     currRecipe.scale(scaleUp, amount);
193 
194                     String title = currRecipe.getTitle();
195                     String intro = currRecipe.getIntroduction();
196                     String ingredients = currRecipe.getIngredients();
197                     String directions = currRecipe.getDirections();
198 
199                     String recipePath = "Recipes/" + title + ".txt";
200                     File newRecipe = new File(recipePath);
201 
202                     try {
203                         writeToFile(newRecipe, title, intro, ingredients, directions);
204                         informationMessage( text: "Your recipe has been scaled.");
205                     }
206                     catch (Exception e) {
207                         errorMessage( text: "Error: There is something wrong with files; Recipe could not be scaled.");
208                     }
209                 }
210                 remove(scalePanel);
211                 updateViewPanel();
212                 add(viewPanel, BorderLayout.CENTER);
213                 validate();
214                 repaint();
215                 currentPanel = viewPanel;
216             }
217         });
```

Figure 20: EditPanel methods

```
110     public void setButtonListener(EditRecipeListener buttonListener) { this.buttonListener = buttonListener; }
111 
112 @   public void editPanel(Recipe r) {
113         titleField.setText(r.getTitle());
114         introductionArea.setText(r.getIntroduction());
115         ingredientsArea.setText(r.getIngredientsForEdit());
116         directionsArea.setText(r.getDirections());
117     }
118 
119 @   public void actionPerformed(ActionEvent e) {
120         JButton clicked = (JButton)e.getSource();
121         if(clicked == confirmButton) {
122             if(buttonListener != null) {
123                 String title = titleField.getText();
124                 String intro = introductionArea.getText();
125                 String ingredients = ingredientsArea.getText();
126                 String directions = directionsArea.getText();
127 
128                 buttonListener.editRecipe( submit: true, title, intro, ingredients, directions);
129             }
130         }
131         else if(clicked == cancelButton) {
132             if(buttonListener != null) {
133                 buttonListener.editRecipe( submit: false,  title: null,  intro: null,  ingredients: null,  directions: null);
134             }
135         }
136     }
```

Figure 21: MainFrame editPanel.setButtonListener()

```
164     editPanel.setButtonListener(new EditRecipeListener() {
165         public void editRecipe(boolean submit, String title, String intro, String ingredients, String directions) {
166             if(submit) {
167                 String recipePath = "Recipes/" + title + ".txt";
168                 File newRecipe = new File(recipePath);
169 
170                 try {
171                         writeToFile(newRecipe, title, intro, ingredients, directions);
172                         addRecipeToArr(newRecipe,  add: false);
173                         informationMessage( text: "Your recipe has been edited.");
174                 }
175                 catch (Exception e) {
176                     errorMessage( text: "Error: There is something wrong with files; Recipe could not be edited.");
177                 }
178             }
179             remove(editPanel);
180             updateViewPanel();
181             add(viewPanel, BorderLayout.CENTER);
182             validate();
183             repaint();
184             currentPanel = viewPanel;
185         }
186     });
```

Words: 905