Iteration 3 Refactoring Report

Major Refactoring 1: Refactor of duplicate code in persistence layer.

Description:
In our previous code, we duplicated the additem method for several of our databases (fridge, grocery, and favourites). The only difference between these classes is which sql table it interacts with. To rectify this, we refactor our code to build a template add in our database helpers class, and simply pass the sql table to be operated on, drastically reducing the code smells in this space.

Previous:

```
public void addItem(FridgeItem Fridge, String user, String password) {
        String name = Fridge.getFoodItem().getName();
        int fridgeEnum;
        int creationEnum;
        int amount = Fridge.getStockableItem().getStock();
        LocalDate date = Fridge.getExpDate();
        if (Fridge.getFoodItem().getStockType() == StockType.values()[0]) {
                fridgeEnum = 0;
        } else {
                fridgeEnum = 1;
        }

        if (Fridge.getFoodItem().getCreator() == CreationType.values()[0]) {
                creationEnum = 0;
        } else {
                creationEnum = 1;
        }

        try {
                Connection con = DriverManager.getConnection(url, user, password);

                PreparedStatement statement = con.prepareStatement(queryInsert);

                statement.setInt(2, fridgeEnum);
                statement.setString(1, name);
                statement.setInt(3, amount);
                statement.setInt(4, creationEnum);
                statement.setInt(6, amount);
                if (date == null) {
                        statement.setDate(5, null);
                } else {
                        statement.setDate(5, Date.valueOf(date));
                }
                statement.executeUpdate();

        } catch (SQLException e) {
                e.printStackTrace();
        }

}
```

```
public void addGroceryItem(FridgeItem Fridge, String user, String password) {
        String name = Fridge.getFoodItem().getName();
        int fridgeEnum;
        int creationEnum;
        int amount = Fridge.getStockableItem().getStock();
        LocalDate date = Fridge.getExpDate();
        if (Fridge.getFoodItem().getStockType() == StockType.values()[0]) {
                fridgeEnum = 0;
        } else {
                fridgeEnum = 1;
        }

        if (Fridge.getFoodItem().getCreator() == CreationType.values()[0]) {
                creationEnum = 0;
        } else {
                creationEnum = 1;
        }

        try {
                Connection con = DriverManager.getConnection(url, user, password);

                PreparedStatement statement = con.prepareStatement(queryInsert);

                statement.setInt(2, fridgeEnum);
                statement.setString(1, name);
                statement.setInt(3, amount);
                statement.setInt(4, creationEnum);
                statement.setInt(6, amount);
                if (date == null) {
                        statement.setDate(5, null);
                } else {
                        statement.setDate(5, Date.valueOf(date));
                }
                statement.executeUpdate();

        } catch (SQLException e) {
                e.printStackTrace();
        }
```

After:

```
public void addItem(FridgeItem Fridge, String user, String password) {
        String type = "fridgeitem";
        helper.adder(type, user, password, Fridge);

}
```

Major Refactoring 2: Refactoring Multi-line UI Component Initialization to Custom Extensions

Description:
For the initialization of UI components using the Java Swing library, multiple lines of code were required to set up fairly basic parameters. Some of the common parameters were:

- *Background/Foreground Color* to ensure that the UI component matches the color scheme of the application.
- *Border* which was usually used to set up padding (extra space) around UI elements to ensure proper layout spacing was kept.
- *Text* which was the output string that the user viewed on top of the UI element.

There were also certain parameters that always needed to be set for UI elements of the same type.

- JPanel always needs *Layout* parameter to be set to define the structure of the encompassed elements
- JButton always needed an *ActionListener* parameter to provide interactivity.
- GridBagConstraints required 5 parameters (*gridx, gridy, weightx, weighty, fill*)

To promote reusable, readable and shorter code we created custom swing elements in the src.presentationLayer.swingExtensions package and simply use constructors to perform most of the parameter initialization.
Since these UI elements were used most frequently (15+ times), the refactoring was very impactful.

Example Refactoring for JPanel to CustomBoxPanel:

Previous:	**After:**

`JPanel p = new JPanel();`
`p.setLayout(new BoxLayout(p,`
`BoxLayout.X_AXIS));`
`p.setBackground(Color.black);`
`p.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));`
`this.add(p);`
	`JPanel p = new`
	`CustomBoxPanel(Color.black,`
	`BoxLayout.X_AXIS,`
	`10);`
	
	`this.add(p);`

Refactored Classes:

- **JPanel → CustomPanel and CustomBoxPanel**
- **JButton → CustomButton**
- **GridBagConstraints → GridConstraintsSpec** (static utility method class)

Major Refactoring 3: Refactoring Reused Useful GUI Elements

Description:
Some GUI elements were being repeated either in the form of their structure or behavior so this led to refactoring these elements into entire classes of their own which accomplish specific tasks that are required at many points in the application. There are 3 instances of this in the application and all 3 have associate refactored classes in the src.presentationLayer.swingExtension:

1) *CustomToggleButton.java*: Used when we want the user to turn some functionality on or off based on a boolean that can be changed when clicking a button. This was encompassed into an extension of the CustomButton to provide both the major refactoring 2 and the ability to switch between 2 text labels on the button and access to a boolean indicating if the toggle was on or off. This functionality was used: (1) to switch between ExpressiveView and CompressedView in the 'Your Fridge and Groceries' panel, (2) Turn the smart feature on or off, (3) To change between restocking/consumption mode in 'Your Fridge and Groceries' panel when smart feature is on.
2) *DateInputField.java:* Used when we want the user to input 3 field: Date, Month and Year. This requires 3 very specific dropdowns and specialized input validation so both the structure and functionality was encompassed into a reusable class used by both types of item addition in the *AddSelectView* and *AddCreateView* classes when assigning expiry dates to items.
3) *InputField.java:* Used as a recurring functionality throughout the program. Provides an interface which provides 2 functionalities: (1) textual input retrieving and (2) clearing input fields achieved by the *getInput()* and *clearField()* methods respectively. This interface was used by the classes to retrieve and clear input from different fields (dropdowns and text fields).

Example Refactoring for Button Toggle Functionality:

Previous:

```
Class A extends JPanel {
    JButton b;
    boolean on;
    ClassA() {
        b = new JButton("On");
        b.addActionListener(this)
        on = true;
    }
    public void actionPerformed( … ) {
         if (on) {
              b.setText("Off");
              on = false;
              // some functionality
         } else {
              b.setText("Off");
              on = true;
              // some functionality
         }
    }
}
```

After:

```
Class A extends JPanel {
    CustomToggleButton b;

    ClassA() {
        b = new
CustomToggleButton("On", "Off", "On");
        b.initToggle(true);
    }

    public void actionPerformed( … ) {
         if (b.isOn()) {
              // some functionality
         } else {
              // some functionality
         }
         b.toggle();
    }
}
```