COMPSCI: 520 HW1

Theory and Practice of Software Engineering

Smruthi Sathyamoorthy and Sudharshan Govindan

Github: https://github.com/sudharshan1234/expense_tracker

Manual review

Non-functional requirements met:

Usability:

- The ExpenseTrackerView class offers a logical and user-friendly GUI with unambiguous labels and input fields.
- The GUI is made to be user-friendly for people who may not have substantial technical knowledge, making it simple to understand and engage with.

Example from the code:

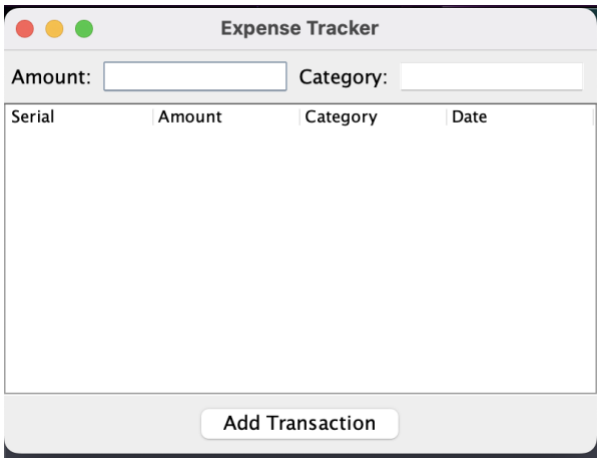
src > ExpenseTrackerView.java > ExpenseTrackerView > ExpenseTrackerView(DefaultTableModel)

```
53  public ExpenseTrackerView(DefaultTableModel model) {
54      setTitle(title:"Expense Tracker"); // Set title
55      setSize(width:600, height:400); // Make GUI larger
56      this.model = model;
57
58      addTransactionBtn = new JButton(text:"Add Transaction");
59
60      // Create UI components
61      JLabel amountLabel = new JLabel(text:"Amount:");
62      amountField = new JTextField(columns:10);
63
64      JLabel categoryLabel = new JLabel(text:"Category:");
65      categoryField = new JTextField(columns:10);
66      transactionsTable = new JTable(model);
67
68      // Layout components
69      JPanel inputPanel = new JPanel();
70      inputPanel.add(amountLabel);
71      inputPanel.add(amountField);
72      inputPanel.add(categoryLabel);
73      inputPanel.add(categoryField);
74      inputPanel.add(addTransactionBtn);
75
76      JPanel buttonPanel = new JPanel();
77      buttonPanel.add(addTransactionBtn);
78
79      // Add panels to frame
80      add(inputPanel, BorderLayout.NORTH);
81      add(new JScrollPane(transactionsTable), BorderLayout.CENTER);
82      add(buttonPanel, BorderLayout.SOUTH);
```

```
84      // Set frame properties
85      setSize(width:400, height:300);
86      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
87      setVisible(b:true);
```

The ExpenseTrackerView constructor initializes a graphical user interface (GUI) window. It sets the title of the window to "Expense Tracker" and adjusts the size to 600 pixels in width and 400 pixels in height. The provided DefaultTableModel (model) is stored for later use. It creates a button labeled "Add Transaction" and defines various UI components like labels and text fields for entering transaction details. These components are organized into two JPanel containers: inputPanel and buttonPanel. inputPanel holds labels and text field, while buttonPanel only contains the "Add Transaction" button. The panels are then positioned within the frame: inputPanel at the top, a scrollable table (created from the model) in the center, and buttonPanel at the bottom. The frame's size is further adjusted to 400 pixels in width and 300 pixels in height. Lastly, the default close operation is set to terminate the application, and the frame is made visible to the user.

Screenshot of the GUI:



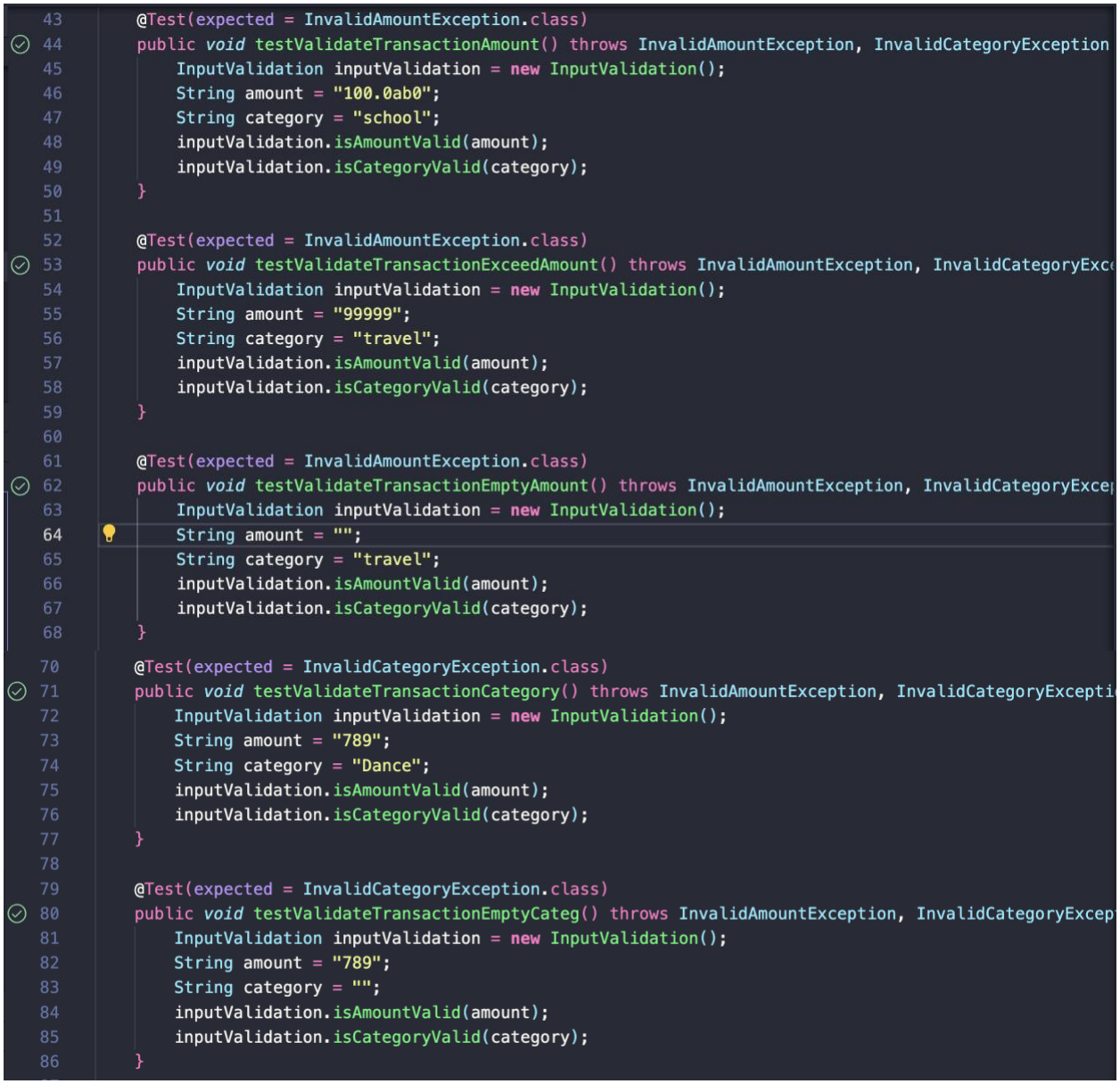
Testability:

- In this scenario, developers can use ExpenseTrackerTest.java to design and run unit tests to ensure that important components are working properly. This ensures that the application's various components work as planned.
- Developers can continue to build new unit tests to cover more elements of the software to improve testability. They can also think about employing testing frameworks or tools to speed up the testing process.

Example from the code:

```java
public class ExpenseTrackerTest {

    private ExpenseTrackerView view;
    private ExpenseTrackerApp app;

    @Before
    public void setup() {
        DefaultTableModel tableModel = new DefaultTableModel();
        tableModel.addColumn(columnName:"Serial");
        tableModel.addColumn(columnName:"Amount");
        tableModel.addColumn(columnName:"Category");
        tableModel.addColumn(columnName:"Date");
        view = new ExpenseTrackerView(tableModel);
        app = new ExpenseTrackerApp();
    }
```

```java
    @Test
    public void testAddTransaction() {
        // Create a new transaction
        double amount = 100.0;
        String category = "Food";
        Transaction transaction = new Transaction(amount, category);

        // Add the transaction to the view
        view.addTransaction(transaction);

        // Get the transactions from the view
        java.util.List<Transaction> transactions = view.getTransactions();

        // Verify that the transaction was added
        assertEquals(1, transactions.size());
        assertEquals(amount, transactions.get(index:0).getAmount(), 0.001);
        assertEquals(category, transactions.get(index:0).getCategory());
    }
```



```
43      @Test(expected = InvalidAmountException.class)
44      public void testValidateTransactionAmount() throws InvalidAmountException, InvalidCategoryException
45          InputValidation inputValidation = new InputValidation();
46          String amount = "100.0ab0";
47          String category = "school";
48          inputValidation.isAmountValid(amount);
49          inputValidation.isCategoryValid(category);
50      }
51
52      @Test(expected = InvalidAmountException.class)
53      public void testValidateTransactionExceedAmount() throws InvalidAmountException, InvalidCategoryExc
54          InputValidation inputValidation = new InputValidation();
55          String amount = "99999";
56          String category = "travel";
57          inputValidation.isAmountValid(amount);
58          inputValidation.isCategoryValid(category);
59      }
60
61      @Test(expected = InvalidAmountException.class)
62      public void testValidateTransactionEmptyAmount() throws InvalidAmountException, InvalidCategoryExce
63          InputValidation inputValidation = new InputValidation();
64          String amount = "";
65          String category = "travel";
66          inputValidation.isAmountValid(amount);
67          inputValidation.isCategoryValid(category);
68      }
70      @Test(expected = InvalidCategoryException.class)
71      public void testValidateTransactionCategory() throws InvalidAmountException, InvalidCategoryExcepti
72          InputValidation inputValidation = new InputValidation();
73          String amount = "789";
74          String category = "Dance";
75          inputValidation.isAmountValid(amount);
76          inputValidation.isCategoryValid(category);
77      }
78
79      @Test(expected = InvalidCategoryException.class)
80      public void testValidateTransactionEmptyCateg() throws InvalidAmountException, InvalidCategoryExcep
81          InputValidation inputValidation = new InputValidation();
82          String amount = "789";
83          String category = "";
84          inputValidation.isAmountValid(amount);
85          inputValidation.isCategoryValid(category);
86      }
```

This code contains a JUnit test case designed to evaluate the ExpenseTrackerView class's functionality. It begins by preparing the necessary testing environment. Creating a DefaultTablemodel with four specified columns is required. Using this paradigm, an instance of ExpenseTrackerView is subsequently created alongside an instance of ExpenseTrackerApp. Within this environment, the actual test, named testAddTransaction, is conducted. In this test, a new Transaction object with the quantity 100.0 and category "Food" is created. The addTransaction method is then used to add this transaction to the view. The test verifies the correctness of this operation by verifying if the list of transactions now contains one item, if the amount matches the

expected value, and if the category is as anticipated. This exhaustive test case ensures the correct operation of the ExpenseTrackerView class.

Furthermore, there are 5 more tests focusing on exception handling. These tests are aimed at the InputValidation class to ensure that it appropriately throws InvalidAmountException and InvalidCategoryException under various input scenarios.

Violated Non-Functional Requirements:

Debuggability:

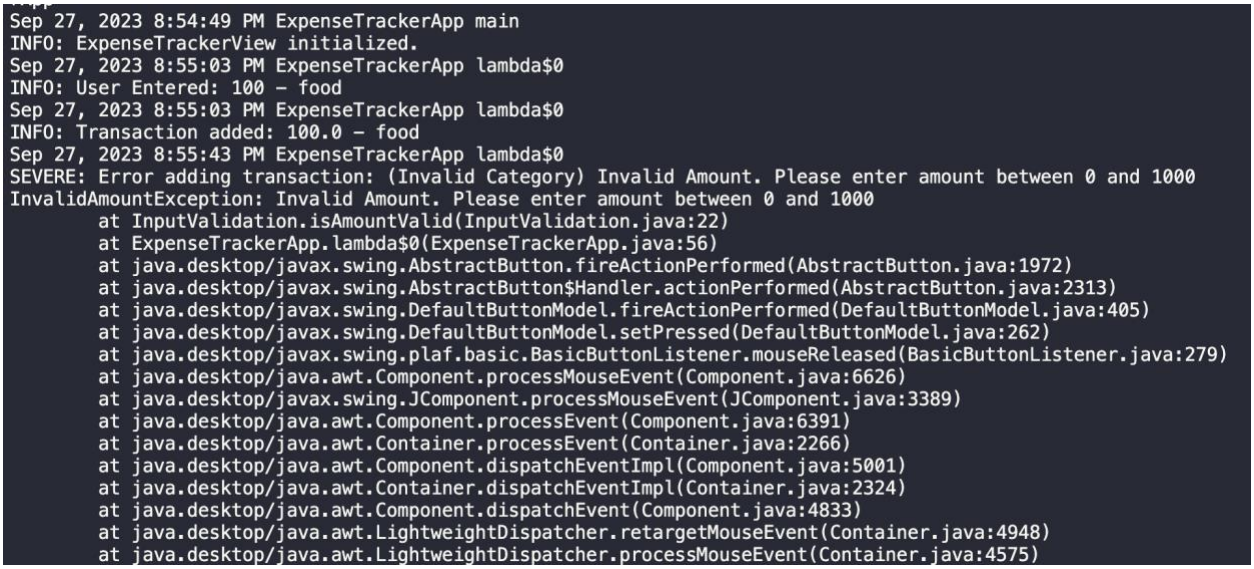
- Example: Throughout the codebase, there are no specific debug statements or logging mechanisms that provide insight into the internal state of the application or potential error conditions.
- Explanation: While the code is generally well-structured, it lacks explicit debug statements or logging mechanisms that could be beneficial for tracking the flow of execution, identifying potential issues, and troubleshooting specific scenarios.
- Illustrative Example:
  Logging was added to our code to check for errors while adding new transactions.

```
14    private static final Logger logger = Logger.getLogger(ExpenseTrackerView.class.getName());
```

```java
// Set up logging properties
LogManager.getLogManager().reset(); // Clear any default configurations
logger.setLevel(Level.ALL); // Log all messages

ConsoleHandler consoleHandler = new ConsoleHandler();
consoleHandler.setLevel(Level.ALL); // Log all messages

SimpleFormatter formatter = new SimpleFormatter();
consoleHandler.setFormatter(formatter);

logger.addHandler(consoleHandler);

logger.info(msg:"ExpenseTrackerView initialized."); // Example log statement
```

```java
view.getAddTransactionBtn().addActionListener(e -> {

    // Get transaction data from view
    String amount = view.getAmountField();

    String category = view.getCategoryField();
    try {

        inputValidation.isAmountValid(amount);
        inputValidation.isCategoryValid(category);
        logger.info("User Entered: " + amount + " - " + category);
        double amountNum = Double.parseDouble(amount);
        // Create transaction object
        Transaction t = new Transaction(amountNum, category);

        // Call controller to add transaction
        view.addTransaction(t);
        logger.info("Transaction added: " + t.getAmount() + " - " + t.getCategory());

    } catch (InvalidAmountException e2) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Invalid Amount !");
        logger.log(Level.SEVERE, "Error adding transaction: (Invalid Category) " + e2.getMessage(), e2);
        e2.printStackTrace();
    } catch (InvalidCategoryException e1) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"Invalid Category!");
        logger.log(Level.SEVERE, "Error adding transaction: (Invalid Category) " + e1.getMessage(), e1);
        e1.printStackTrace();
    }

});
```

Logging outputs in terminal:



- How to improve debuggability: We have added debug statements to capture key information during runtime such as addition of transaction and exceptions thrown while adding transactions. Incorporating exception handling and error messages can provide valuable information during debugging.

Modularity:

- Example: In ExpenseTrackerView.java, the class is responsible for both creating the GUI components (labels, buttons, tables) and directly interacting with transactions (e.g., adding transactions to the view). This blurs the lines between the UI and business logic, reducing the overall modularity.
- Explanation: While modularity is attempted through distinct classes, the ExpenseTrackerView class could benefit from a clearer separation between the GUI components and the business logic. Currently, it manages both aspects, which can make future code modifications and expansions more difficult.

- Illustrative Example:

src > ExpenseTrackerView.java > ExpenseTrackerView

```java
public class ExpenseTrackerView extends JFrame {

  private JTable transactionsTable;
  private JButton addTransactionBtn;
  private JTextField amountField;
  private JTextField categoryField;
  private DefaultTableModel model;
  private List<Transaction> transactions = new ArrayList<>();



  public JTable getTransactionsTable() {
    return transactionsTable;
  }

  public double getAmountField() {
    if(amountField.getText().isEmpty()) {
      return 0;
    }else {
    double amount = Double.parseDouble(amountField.getText());
    return amount;
    }
  }

  public void setAmountField(JTextField amountField) {
    this.amountField = amountField;
  }

  public String getCategoryField() {
    return categoryField.getText();
  }
```

```java
public void setCategoryField(JTextField categoryField) {
    this.categoryField = categoryField;
}

public JButton getAddTransactionBtn() {
    return addTransactionBtn;
}
public DefaultTableModel getTableModel() {
    return model;
}

public ExpenseTrackerView(DefaultTableModel model) {
    setTitle(title:"Expense Tracker"); // Set title
    setSize(width:600, height:400); // Make GUI larger
    this.model = model;

    addTransactionBtn = new JButton(text:"Add Transaction");

    // Create UI components
    JLabel amountLabel = new JLabel(text:"Amount:");
    amountField = new JTextField(columns:10);

    JLabel categoryLabel = new JLabel(text:"Category:");
    categoryField = new JTextField(columns:10);
    transactionsTable = new JTable(model);

    // Layout components
    JPanel inputPanel = new JPanel();
    inputPanel.add(amountLabel);
    inputPanel.add(amountField);
    inputPanel.add(categoryLabel);
    inputPanel.add(categoryField);
    inputPanel.add(addTransactionBtn);

    JPanel buttonPanel = new JPanel();
    buttonPanel.add(addTransactionBtn);

    // Add panels to frame
    add(inputPanel, BorderLayout.NORTH);
    add(new JScrollPane(transactionsTable), BorderLayout.CENTER);
    add(buttonPanel, BorderLayout.SOUTH);

    // Set frame properties
    setSize(width:400, height:300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(b:true);

}
```

- <u>How to improve modularity:</u> Refactor the ExpenseTrackerView class so that it adheres more rigorously to the MVC architectural pattern. This requires segregating GUI (View), business logic (Controller), and data management (Model) concerns. Consequently, each

component has a clearly defined function and can be modified and expanded independently.

Modularity: MVC architecture pattern

From the Application's Perspective:
Component A:

- Component Type: View
- Explanation: Component A consists of the input fields for "Amount" and "Category". It allows users to input the transaction details (amount and category).

Component B:

- Component Type: View
- Explanation: Component B is the table with columns "Serial", "Amount", "Category", and "Date". It displays the list of transactions.

Component C:

- Component Type: Controller
- Explanation: Component C is the "Add Transaction" button. It triggers an action in response to user interaction, which is typically handled by the controller to add a new transaction.

Corresponding Source Code:
Model:

- Source Code: Transaction.java
- Explanation: The Transaction.java file represents the model in the application. It defines the structure and behavior of a transaction.

One View:

- Source Code: ExpenseTrackerView.java
- Explanation: ExpenseTrackerView.java is responsible for displaying the GUI components, including the input fields (Amount and Category) and the table (Component A and Component B).

```
53    public ExpenseTrackerView(DefaultTableModel model) {
54      setTitle(title:"Expense Tracker"); // Set title
55      setSize(width:600, height:400); // Make GUI larger
56      this.model = model;
57
58      addTransactionBtn = new JButton(text:"Add Transaction");
59
60      // Create UI components
61      JLabel amountLabel = new JLabel(text:"Amount:");
62      amountField = new JTextField(columns:10);
63
64      JLabel categoryLabel = new JLabel(text:"Category:");
65      categoryField = new JTextField(columns:10);
66      transactionsTable = new JTable(model);
67
68      // Layout components
69      JPanel inputPanel = new JPanel();
70      inputPanel.add(amountLabel);
71      inputPanel.add(amountField);
72      inputPanel.add(categoryLabel);
74      inputPanel.add(addTransactionBtn);
75
76      JPanel buttonPanel = new JPanel();
77      buttonPanel.add(addTransactionBtn);
78
79      // Add panels to frame
80      add(inputPanel, BorderLayout.NORTH);
81      add(new JScrollPane(transactionsTable), BorderLayout.CENTER);
82      add(buttonPanel, BorderLayout.SOUTH);
83
84      // Set frame properties
85      setSize(width:400, height:300);
86      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
87      setVisible(b:true);
88
89    }
```

<u>One Controller:</u>
In the provided code, the ExpenseTrackerView class manages user interface components and contains transaction-related logic.

- Processing User Input:

    - The Controller must manage the logic for extracting user input (such as amount and category). This logic is currently present in the getAmountField() and getCategoryField() methods of ExpenseTrackerView.
- Incorporating Transactions:
    - The actual creation of a Transaction object and its addition to the model should occur within the Controller. This logic is presently present in the action listener of the ExpenseTrackerView.

```
22    public String getAmountField() {
23      return amountField.getText();
24    }
```

```
38    public String getCategoryField() {
39      return categoryField.getText();
40    }
```

```
123   public List<Transaction> getTransactions() {
124     return transactions;
125   }
126
127   public void addTransaction(Transaction t) {
128     transactions.add(t);
129     getTableModel().addRow(new Object[]{t.getAmount(), t.getCategory(), t.getTimestamp()});
130     refresh();
131   }
```

Extensibility:

You should provide a description of how to add *filtering* on the list of added transactions. You can think of filtering based on *category* types, *amount* or by *date*. For simplicity, think about applying one filter at a time. Your description should identify the set of fields and methods that need to be modified or introduced to support the extension. For each identified field or method, briefly describe the necessary changes to support the extension.

Answer:
To add filtering functionality to the list of added transactions based on category types, amount, or date, we can follow the below steps:

1. **<u>Introduce Filter Options:</u>**
    a. Fields:

i. Add fields in ExpenseTrackerView class to hold filter options (e.g., selectedCategory, minAmount, maxAmount, startDate, endDate).

b. Methods:
   i. Add getter methods in ExpenseTrackerView to retrieve filter options.

c. Description:
   i. Introduce fields like selectedCategory, minAmount, maxAmount, startDate, and endDate in ExpenseTrackerView to store user-selected filter options.
   ii. Create corresponding getter methods to retrieve these filter options.

2. **Modify refreshTable Method:**
   a. Methods:
      i. Update the refreshTable method in ExpenseTrackerView class.
   b. Description:
      i. Add logic to apply filters based on the selected category, amount range, and date range before updating the table. Only transactions that match the selected criteria should be displayed.

3. **Update GUI Components:**
   a. Methods:
      i. Add buttons or input fields to allow users to set filter criteria.
   b. Description:
      i. Introduce GUI components (e.g., buttons, input fields) to allow users to input filter criteria (category, amount range, date range).
4. **Handle User Interaction:**
   a. Methods:
      i. Implement event listeners for filter-related GUI components.
   b. Description:
      i. Implement event listeners for the filter-related GUI components. When a user interacts with these components (e.g., clicks a button or enters filter criteria), update the corresponding filter options in ExpenseTrackerView.
5. **Apply Filters in refreshTable:**
   a. Methods:
      i. Modify the refreshTable method in ExpenseTrackerView to apply the selected filters.
   b. Description:
      i. In the refreshTable method, before updating the table, apply the selected filters to the list of transactions. Only transactions that match the selected criteria should be displayed.
6. **Update the GUI to Reflect Filtered Results:**
   a. Methods:
      i. Modify the GUI components to display filtered results.
   b. Description:

        i. Update the GUI components (e.g., table, labels) to display the filtered results based on the selected filter options.

7. **<u>Add Clear Filters Option</u>**
    a. Methods:
        i. Introduce a button or option to clear applied filters.
    b. Description:
        i. Optionally, you can add a button or option to allow users to clear any applied filters and display the full list of transactions.