The goal of this assignment is to manually review and redesign an Expense Tracker App, according to the model-view-controller (MVC) architecture pattern.

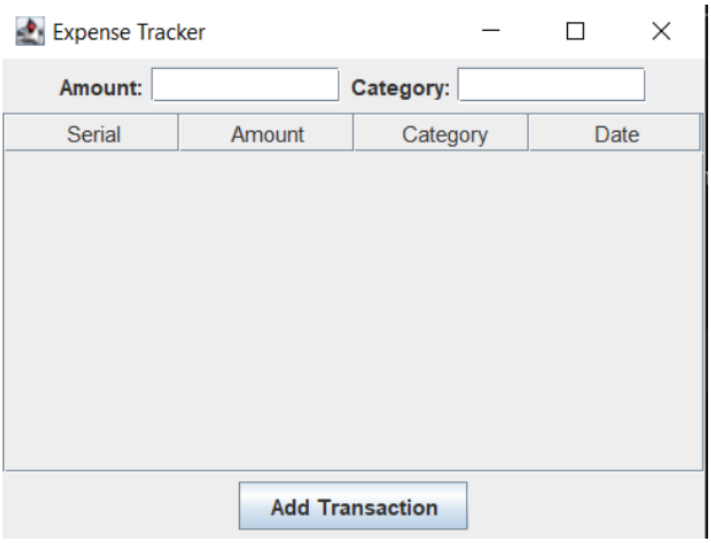


Figure 1: Screenshot of the 'Expense Tracker' UI

MANUAL REVIEW

- two examples of non-functional requirements that are satisfied

For the satisfied non-functional requirements, please follow the pattern:

- Brief summary of the non-functional requirement (a few keywords)
- An illustrative example from the code. The example should either specify a code element name (e.g., class, method, field) or be a screenshot. Line numbers should not be used.

Testability:
Brief Summary: The ease with which the software can be tested to ensure its functionality and correctness.
Example: The **'ExpenseTrackerTest'** class showcases the ability of the software to be tested. The method **'testAddTransaction'** is an illustrative example where a transaction is created, added to the view, and then verified if the transaction was added correctly. This structure facilitates unit testing, ensuring that the software behaves as expected.

Debuggability:
Brief Summary: The ease with which the software can be debugged to identify and fix issues.
Example: In the **'ExpenseTrackerView'** class, methods like **'getAmountField'** and **'getCategoryField'** help in isolating the part of the system to debug if any issues arise related to transaction values. These methods allow for easier tracking and fixing of bugs related to user input or transaction display by providing clear separation and access to crucial pieces of data. This modularity in the code helps in identifying and resolving issues in a structured manner.

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

- two examples of non-functional requirements that are violated

For the violated non-functional requirements, please follow the pattern:

- Brief summary of the non-functional requirement (a few keywords)
- An illustrative example from the code. The example should either specify a code element name (e.g., class, method, field) or be a screenshot. Line numbers should not be used.
- Explanation of this issue (could refer to general principles or poor design choices with respect to the desired non-functional requirement)
- How to fix the issue (a few sentences)

Extensibility:
Brief Summary: Ability to extend the functionality of the system with minimal changes.
Illustrative Example: In the 'ExpenseTrackerApp' and 'ExpenseTrackerView' classes, the handling of transactions is tightly coupled. The view.addTransaction(t) method call within the 'ExpenseTrackerApp' class adds the transaction directly to the view, which in turn updates the table model.

Explanation of Issue: The tight coupling between the view and the model restricts the extensibility of the system. Ideally, the system should follow a more decoupled design that adheres to the Model-View-Controller (MVC) pattern to allow for easier extension, such as adding new types of transactions or changing the way transactions are handled or stored.

How to Fix the Issue: Implement a controller class to handle the logic of adding, removing, and updating transactions. The controller should communicate between the model and the view, ensuring a separation of concerns and making the system more extensible.

```
public class ExpenseTrackerController {
  private ExpenseTrackerModel model;
  private ExpenseTrackerView view;
```

```
    public ExpenseTrackerController(ExpenseTrackerModel model, ExpenseTrackerView view) {
        this.model = model;
        this.view = view;
    }

    public void addTransaction(Transaction transaction) {
        model.addTransaction(transaction);
        view.refresh();
    }
    // ... other controller methods
}
```

Modularity:
Brief Summary: Division of a software system into separate, interchangeable components.
Illustrative Example: The ExpenseTrackerView class is responsible for both the presentation and some aspects of the data management of transactions (such as adding transactions directly to an internal list and updating the table model).

Explanation of Issue: The lack of modularity in the design makes the system harder to understand, maintain, and extend. For instance, adding a new feature like transaction editing or removal would require modifying the ExpenseTrackerView class, making it a bloated class with too many responsibilities.

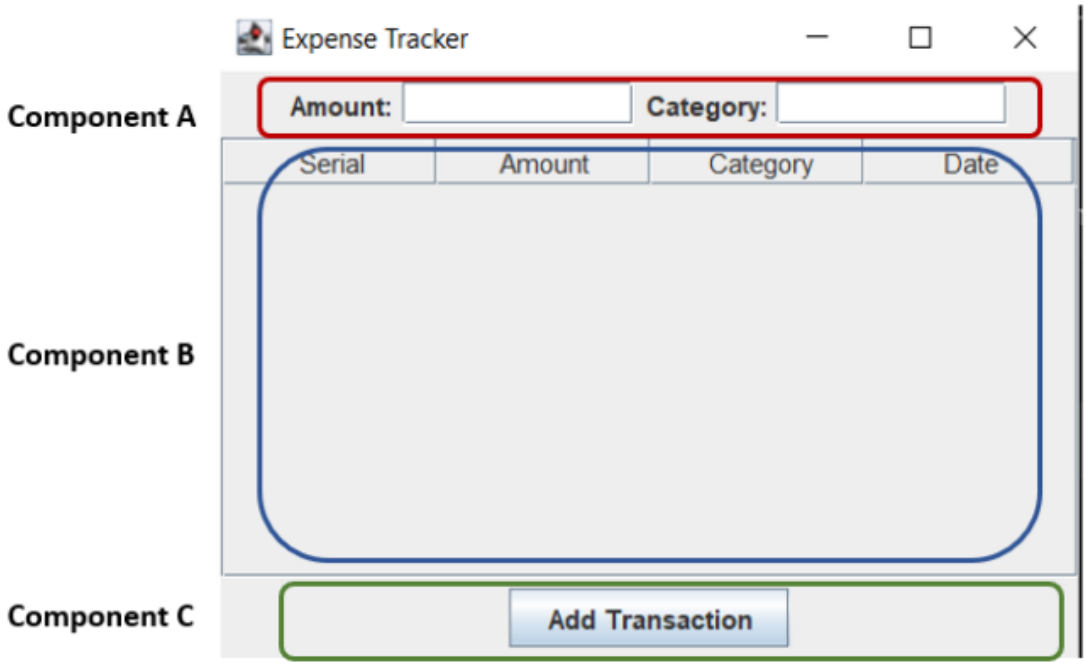
How to Fix the Issue: Adopting the MVC pattern can solve this issue by separating the concerns. Create a separate ExpenseTrackerModel class to handle the data management of transactions, leaving the ExpenseTrackerView class to focus solely on presentation. This separation will lead to a more modular design, where each class has a single responsibility.

```
public class ExpenseTrackerModel {
    private List<Transaction> transactions = new ArrayList<>();

    public List<Transaction> getTransactions() {
        return transactions;
    }

    public void addTransaction(Transaction transaction) {
        transactions.add(transaction);
    }
    // ... other model methods
}
```

MODULARITY: MVC ARCHITECTURE PATTERN



In Figure 2, identify the following components from the Application's perspective in the UI (not Java.swing perspective):

- 2.1.1) Component A: View, Controller, or both? Briefly explain what is being visualized and/or what user interaction is being provided.
- 2.1.2) Component B: View, Controller, or both? Briefly explain what is being visualized and/or what user interaction is being provided.
- 2.1.3) Component C: View, Controller, or both? Briefly explain what is being visualized and/or what user interaction is being provided.

Identify the original application source code (including all classes, fields, and methods) corresponding to the:

- 2.2.1) Model
- 2.2.2) One view (from the 3 components listed above)
- 2.2.3) One controller (from the 3 components listed above)

2.1 Component Identification:

2.1.1) Component A: View

Visualization: Component A has text fields that show the "Amount" and "Category" of the transaction the user desires to record.

User Interaction: The amount and category of a new transaction that users want to monitor may be entered.

2.1.2) Component B: View

Visualization: The table in Component B depicts the recorded transactions by listing their serial numbers, amounts, categories, and dates.

User Interaction: Although no particular interaction is specified, this component may enable the user to engage by choosing or viewing information of previous transactions.

2.1.3) Component C: Controller

Visualization: Component C is an "Add Transaction" button.

User Interaction: When this button is selected, a new transaction utilizing the information entered in Component A is recorded, and Component B is updated as a result.

2.2 Code Identification:

2.2.1) Model:

The Transaction class represents the model in this application. It encapsulates the data of a transaction, including amount, category, and timestamp.

```
public Transaction(double amount, String category) {
    this.amount = amount;
    this.category = category;
    this.timestamp = generateTimestamp();
}

public double getAmount() {
    return amount;
}
```

2.2.2) One view (Component B):

The ExpenseTrackerView class represents the view. Specifically, the transactionsTable field and related methods like refreshTable() are associated with Component B.

```
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 void refreshTable(List<Transaction> transactions) {
    // model.setRowCount(0);
    model.setRowCount(0);
    int rowNum = model.getRowCount();
    double totalCost=0;
    for(Transaction t : transactions) {
      totalCost+=t.getAmount();
    }
```

2.2.3) One controller (Component C):

The controller logic for Component C is embedded within the ExpenseTrackerApp class, within the ActionListener attached to the "Add Transaction" button.

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

In this setup, the ExpenseTrackerApp class acts as a simple controller by handling the user interaction, creating a new Transaction object, and updating the view.

Extensibility: Proposed extension [15 points]

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.

Adding filtering functionality to the list of transactions will involve creating a new method to handle the filtering logic, adding new UI components for the user to specify the filter criteria, and updating the method which refreshes the table to account for the current filter. Below are steps along with the necessary fields and methods to be modified or introduced to support this extension:

New UI Components:

Fields:

A **JComboBox** for filter categories: Allows the user to select a category to filter by.
A **JTextField** for filter amounts: Allows the user to enter an amount to filter by.
A **JButton** to activate the filter: Allows the user to apply the filter.
A **JButton** to clear the filter: Allows the user to clear the currently applied filter.
Methods:
createFilterComponents(): A new method to create and layout the filter UI components.

```
public void createFilterComponents() {
    JComboBox<String> categoryFilterComboBox = new JComboBox<>();
    JTextField amountFilterField = new JTextField(10);
    JButton applyFilterBtn = new JButton("Apply Filter");
    JButton clearFilterBtn = new JButton("Clear Filter");
```

New Fields in ExpenseTrackerView:
List<Transaction> filteredTransactions: A list to hold the transactions that meet the current filter criteria.

```
private List<Transaction> filteredTransactions = new ArrayList<>();
```

New Method in ExpenseTrackerView:
applyFilter(): A method to apply the current filter criteria to the list of transactions.

```
public void applyFilter(String category, Double amount) {
    filteredTransactions.clear();
    for (Transaction t : transactions) {
        if ((category != null && category.equals(t.getCategory())) ||
            (amount != null && amount.equals(t.getAmount()))) {
            filteredTransactions.add(t);
        }
    }
    refreshTable(filteredTransactions);
}
```

Modified Method in ExpenseTrackerView:
refresh(): Modify to call refreshTable with filteredTransactions instead of transactions if a filter is active.

```
public void refresh() {
    if (filteredTransactions.isEmpty()) {
        refreshTable(transactions);
```

```
    } else {
        refreshTable(filteredTransactions);
    }
}
```

New Method in ExpenseTrackerView:
clearFilter(): A method to clear the current filter.

```
public void clearFilter() {
    filteredTransactions.clear();
    refresh();
}
```

Action Listeners:
Add action listeners to the "Apply Filter" and "Clear Filter" buttons to call applyFilter() and clearFilter() respectively when clicked.

```
applyFilterBtn.addActionListener(e ->
applyFilter(categoryFilterComboBox.getSelectedItem().toString(),
amountFilterField.getText().isEmpty() ? null :
Double.parseDouble(amountFilterField.getText())));
clearFilterBtn.addActionListener(e -> clearFilter());
```

With these modifications, the ExpenseTrackerView class will now support filtering transactions based on category or amount. The filter UI components allow the user to specify the filter criteria, and the applyFilter and clearFilter methods manage the filtering logic and the display of the filtered transactions.

Model (ExpenseTrackerModel.java): A method to filter transactions based on the criteria.

```
public class ExpenseTrackerModel {
    private List<Transaction> transactions;
     public List<Transaction> filterTransactions(String filterType, Object
filterValue) {
        List<Transaction> filteredTransactions = new ArrayList<>();
        for (Transaction transaction : transactions) {
            switch (filterType) {
                case "Category":
                    if (transaction.getCategory().equals(filterValue)) {
```

```
                        filteredTransactions.add(transaction);
                    }
                    break;
                case "Amount":
                    if (transaction.getAmount() == (Double) filterValue) {
                        filteredTransactions.add(transaction);
                    }
                    break;
                case "Date":
                    // Assuming Transaction has a getDate method.
                    if (transaction.getDate().equals(filterValue)) {
                        filteredTransactions.add(transaction);
                    }
                    break;
            }
        }
        return filteredTransactions;
    }
}
```

View (ExpenseTrackerView.java): Filter fields and a filter button to the UI and method to update the table based on filtered transactions.

```
public class ExpenseTrackerView extends JFrame {
    private DefaultTableModel tableModel;
    private JTextField filterField;
    private JComboBox<String> filterTypeComboBox;
    private JButton filterBtn;
    public void updateTable(List<Transaction> transactions) {
        tableModel.setRowCount(0);  // Clear existing rows
        for (Transaction transaction : transactions) {
            Object[] row = new Object[4];
            row[1] = transaction.getAmount();
            row[2] = transaction.getCategory();
            // Assuming Transaction has a getDate method.
            row[3] = transaction.getDate();
            tableModel.addRow(row);
        }
```

```
    }

    public String getFilterField() {
        return filterField.getText();
    }

    public String getFilterType() {
        return (String) filterTypeComboBox.getSelectedItem();
    }

    public JButton getFilterBtn() {
        return filterBtn;
    }
}
```

Controller (ExpenseTrackerController.java): An action listener to the filter button to execute the filtering.

```
public class ExpenseTrackerController {
    private ExpenseTrackerView view;
    private ExpenseTrackerModel model;



    private void initController() {
                        view.getAddTransactionBtn().addActionListener(e   ->
addTransaction());
        view.getFilterBtn().addActionListener(e -> filterTransactions());
    }

    private void filterTransactions() {
        String filterType = view.getFilterType();
        String filterValue = view.getFilterField();
                             List<Transaction>    filteredTransactions    =
model.filterTransactions(filterType, filterValue);
        view.updateTable(filteredTransactions);
    }
}
```

In the proposed setup:

The **ExpenseTrackerModel** is responsible for filtering the transactions based on the given criteria.

The **ExpenseTrackerView** is updated to include UI elements for filtering, and a method to update the table based on filtered transactions.

The **ExpenseTrackerController** is updated to handle the action of filtering when the user presses the filter button.

This setup keeps the concerns of managing data, displaying data, and handling user actions separated according to the MVC pattern, while extending the functionality to include filtering of transactions.

By,

Hemangani Nagarajan

Harshita Loganathan

Git : https://github.com/HEMANGANI/CS520-Fall2023