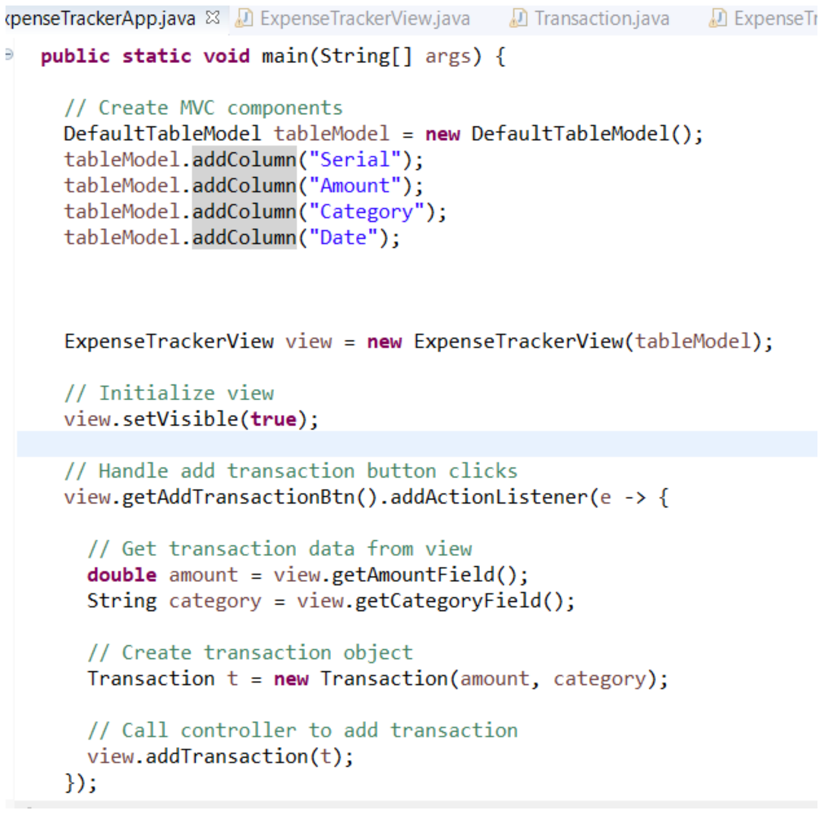
Group: Shreeya-Apoorva-hw

Shreeya Patil   Apoorva Rajan

Manual Review

- **Non-functional requirements that are satisfied**
    - Understandability:
      Non-functional requirements, such as "understandability," are essential for assessing the quality and maintainability of a Java codebase. The code depicts the use of Comments, Meaningful Variable and Method Naming. Modularization and Encapsulation is also followed because the code is broken down into smaller, modular components with clear responsibilities, and Single Responsibility Principle (SRP) is applied to ensure that each class has a single, well-defined purpose. There's also Consistent Code Style and Formatting. Hence because of many such reasons like these, we can say that Understandability is satisfied.

      Screenshot:

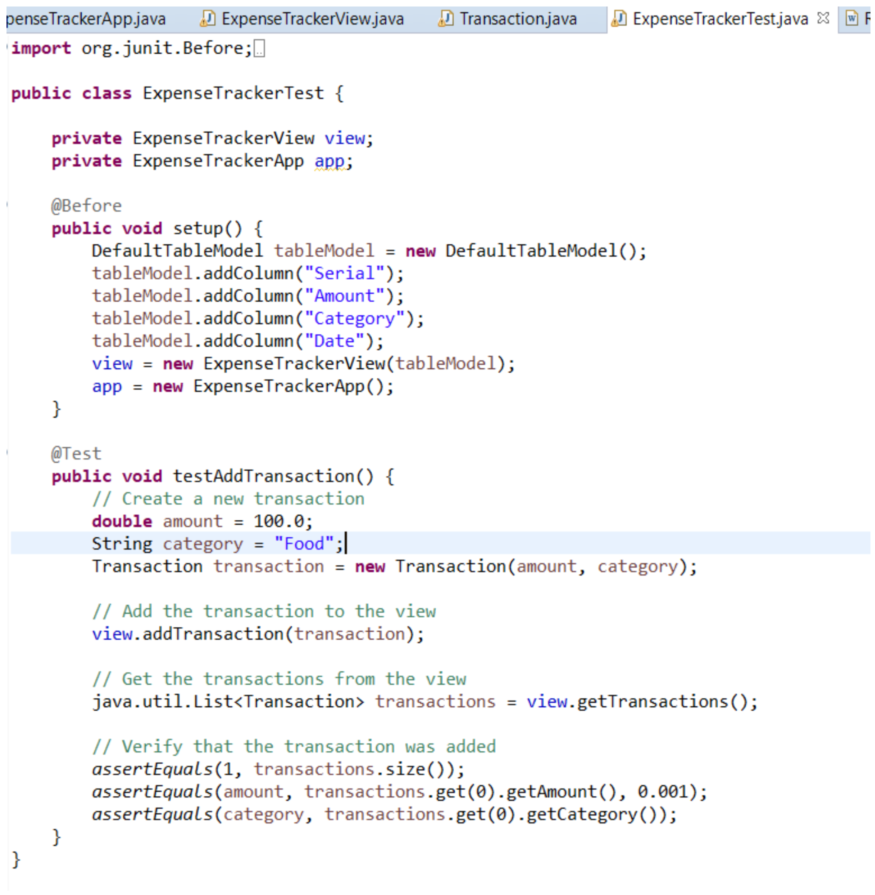


```java
public static void main(String[] args) {

  // Create MVC components
  DefaultTableModel tableModel = new DefaultTableModel();
  tableModel.addColumn("Serial");
  tableModel.addColumn("Amount");
  tableModel.addColumn("Category");
  tableModel.addColumn("Date");



  ExpenseTrackerView view = new ExpenseTrackerView(tableModel);

  // Initialize view
  view.setVisible(true);

  // Handle add transaction button clicks
  view.getAddTransactionBtn().addActionListener(e -> {

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

    // Create transaction object
    Transaction t = new Transaction(amount, category);

    // Call controller to add transaction
    view.addTransaction(t);
  });
```

    - Testability:

ExpenseTrackerTest.java class is used to test for errors in the code. Ensuring testability is an essential aspect of software development. Testability ultimately leads to more robust and maintainable software.

Screenshot:



```java
import org.junit.Before;

public class ExpenseTrackerTest {

    private ExpenseTrackerView view;
    private ExpenseTrackerApp app;

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

    @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(0).getAmount(), 0.001);
        assertEquals(category, transactions.get(0).getCategory());
    }
}
```

- **<u>Non-functional Requirements that are violated</u>**
    - <u>Data integrity:</u>

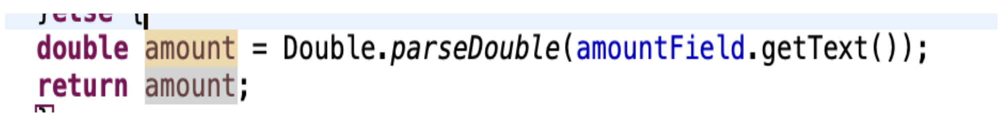
      The input data is not validated. Error messages related to Exception handling for wrong data types entered in the UI not handled. The user should be informed why the issue was caused.

      Data integrity requires that input data is validated to ensure it meets the expected format, constraints, and business rules. If the system does not perform proper input data validation, it may lead to incorrect or inconsistent data, violating data integrity.

      Effective exception handling is essential for maintaining data integrity. When incorrect or unexpected data types are entered through the user interface (UI), the system should handle these exceptions gracefully.

Failing to do so may result in data corruption or erroneous data, impacting data integrity.
Data integrity is also related to user experience. In cases of data validation errors, it is essential to provide clear and informative error messages to users. These messages should explain why the issue occurred, helping users understand and correct their input. Neglecting user-friendly error messages can affect user satisfaction and the overall usability of the system.

- Debuggability:
  No breakpoints or watch points in the code. Debuggability refers to the ease with which developers can debug and diagnose issues in the software code. It encompasses various aspects, including the ability to set breakpoints, watch variables, step through code, and obtain meaningful debugging information.
  No breakpoints or watch points in the code means that the software may lack essential debugging capabilities. This can make it challenging for developers to identify and resolve issues efficiently, potentially leading to longer debugging cycles, increased maintenance efforts, and reduced software quality.
  Therefore, the absence of breakpoints and watch points in the code can be seen as a violation of the non-functional requirement related to debuggability. To address this, developers may need to implement or enhance debugging features and tools within the software development environment to facilitate effective debugging and troubleshooting.



```
}else {
  double amount = Double.parseDouble(amountField.getText());
  return amount;
```

We can add a watch point where we retrieve the amount value from user input since we are not checking if the amount field has a value that can be parsed to double-type.

Modularity: MVC Architecture Pattern

2.1.1) Controller and View:
This component combines user interaction and data input. It provides a user interface for entering data, retains inputted values, and submits both the amount and category to the system. The model then processes this data and displays the results. Thus the view and controller are in play here as we can make changes through this component and also view the input being added.

2.1.2) View:
This component is responsible for presenting information to the user. It displays data such as the amount, category, timestamp of entry, and total amount.

2.1.3) Controller:
The controller handles user actions, such as clicking the "Add Transaction" button. It communicates with the model to add entries to the database and subsequently displays the relevant details.

2.2.1) The ExpenseTrackerApp.java class is the model. This class designs the model.

2.2.2) Component B is the view
ExpenseTrackerView.java displays the details in the table (Component B) and it is being called in the ExpenseTrackerApp.java file. The Component B just displays the details of the user's entries.

2.2.3) Component C is the controller
Component C shows the button "Add Transaction" which the user can click after entering the amount and category. On clicking of the "Add Transaction" button, the model uses the controller to update the table and finally display details in component B.

In ExpenseTrackerApp.java:

```
  // Handle add transaction button clicks
```

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

      // Get transaction data from view
      double amount = view.getAmountField();
      String category = view.getCategoryField();
      boolean isAmountValid = InputValidation.isAmountValid(amount);
      boolean isCategoryValid = InputValidation.isCategoryValid(category);
      if (isAmountValid && isCategoryValid) {

      // Create transaction object
      Transaction t = new Transaction(amount, category);

      // Call controller to add transaction
      view.addTransaction(t);
      } else {
          // Handle invalid input data (e.g., show an error message)
          // You can implement this part based on your application's requirements.
          // For example, display an error message to the user.
          System.out.println("Invalid input data.");
      }
    });
```

In ExpenseTrackerView.java:

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

  public void addTransaction(Transaction t) {
    transactions.add(t);
                    getTableModel().addRow(new     Object[]{t.getAmount(),     t.getCategory(),
t.getTimestamp()});
    refresh();
  }
```

Understandability

The javadoc file is created and there are incremental.

Extensibility:

We will consider adding filtering based on category:

Step 1: Modify ExpenseTrackerView.java

- In the ExpenseTrackerView.java file, add a new dropdown button labeled "Filter" with filter options, such as different categories e.g. Food, Travel, Bills, etc.

Step 2: Implement Filter Function

- Inside the ExpenseTrackerView class, create a function that can filter transactions from the 'transactions' list based on the selected category.

Step 3: Access and Manipulate the Table

- In the same class (ExpenseTrackerView.java), access the table that displays transactions.
- Clear all existing rows from the table.

Step 4: Add Filtered Rows

- After clearing the table, add the rows from the ‘transactions’ list that match the filter criteria to the table.

Step 5: Declare Action Trigger in ExpenseTrackerApp.java

- In the ExpenseTrackerApp.java file, declare the action trigger that invokes the filter function when the "Filter" dropdown selection changes.

Step 6: User Interaction

- Ensure that the user can interact with the "Filter" dropdown to select a category and trigger the filtering process.