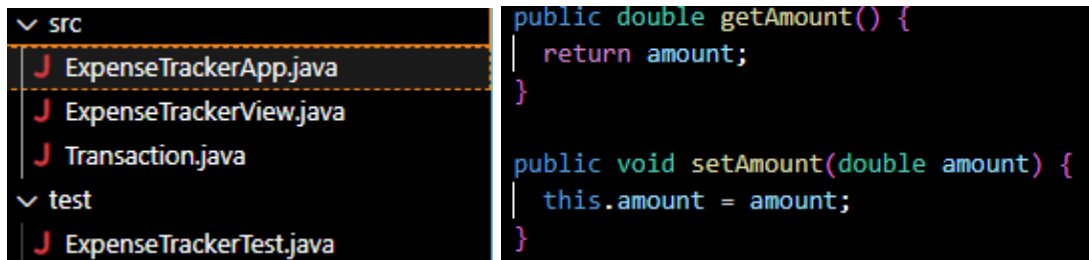


Manual Review

Non-Functional Requirements that are satisfied

- 1) Understandability : How easy it is to set up and understand the design and architecture of the system. Naming of the classes, the methods and the variables are appropriate (meaningful) and easy to relate. Comments for each class's methods are provided, moreover, there are comments for each and every step as we proceed in the logic flow. Lastly, a build file is provided which could be used with Apache ant and quickly create javadoc files for the system, and run the test cases.



The screenshot shows an IDE with a project structure on the left and code snippets on the right. The project structure includes a 'src' folder with 'ExpenseTrackerApp.java', 'ExpenseTrackerView.java', and 'Transaction.java', and a 'test' folder with 'ExpenseTrackerTest.java'. The code snippets show the 'getAmount()' and 'setAmount()' methods of the 'ExpenseTrackerApp' class.

```
public double getAmount() {  
    return amount;  
}  
  
public void setAmount(double amount) {  
    this.amount = amount;  
}
```

- 2) Debuggability : Easiness to remove/rectify some bug/error in the system. Due to good naming conventions that enhanced the understandability, it is also easy to debug the errors in the setup. If we observe one component is showing some unexpected behavior, we can directly jump to the corresponding method/variable responsible for the view/action and debug it.

Example: If some unexpected behavior is observed in the table layout - 'Serial' column comes after 'Amount' column - then we just need to check the flow of how the 'tableModel' variable is changed in the ExpenseTrackerApp and can debug it.

Non-Functional requirements that are violated

- 1) Testability : Testability should provide verification for components in a program. It should look for the flaw in the program. In the function testAddTranscation of class ExpenseTrackerTest, it does not cover impossible/edge cases, etc: negative amount, random category names.



The screenshot shows a code snippet for the 'testAddTranscation' method in the 'ExpenseTrackerTest' class. The method is annotated with '@Test' and contains code to create a new transaction with a fixed amount of 100.0 and a category of 'Food'.

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

Solution: Composing test cases with only normal parameters is insufficient. We could add another test case that assigns the mock-up variable `amount` to 0 or a negative number and think of the expected output.

- 2) Extensibility: Everything is specifically defined to the point that the application loses the ability to extend the features. In the class ExpenseTracker main function, category names are specified instead of being passed into the function, which could be difficult for future extensions since the number of categories (columns) could change.

```
6  ✓ public class ExpenseTrackerApp {
7
8  ✓ public static void main(String[] args) {
9
10     // Create MVC components
11     DefaultTableModel tableModel = new DefaultTableModel();
12     tableModel.addColumn("Serial");
13     tableModel.addColumn("Amount");
14     tableModel.addColumn("Category");
15     tableModel.addColumn("Date");
```

Solution: Instead of specifying the category names in the main function, we could extract these strings and make it an array or enum. It could make the code less repetitive, and if there's a new category, all we need to do is add a new element in the array.

Modularity: MVC architecture pattern

2.1.1 - Controller. In the context of MVC pattern, component A, the two text fields for amount and category should be seen as controller components since the main function here is for users to interact (entering a certain number or string) with them, instead of showing certain information.

2.1.2 - View. In an MVC architecture pattern, a view is a term used for what the user sees, it never interacts with the view - never updates it directly. In component B, a table of previously added transactions is being visualized by the user. This table changes/gets updated when the user interacts with the controller, but is not changed by the user itself.

2.1.3 - Controller. Component C is a button that is utilized by the user in order to add the transactions to the table. If we go by the definition of Controller in an MVC, it is a component of the system used by the user solely for interaction, therefore, it can be categorized as a Controller rather than a View.

2.2.1 - Model :

Class ExpenseTrackerView

- Function - refreshTable(), refresh() and getTransactions()

2.2.2 - View :

- Component B - tableModel attribute/field of class ExpenseTrackerApp

2.2.3 - Controller

- Component C - addTransactionButton of class ExpenseTrackerView

Extensibility

Additional components

Filtering is a technique to filter some of the data out of the entire dataset that does follow some constraints. We require the user to have options to filter on fields - 'amount', 'category' and 'date'. In order for users to enter these constraints we need 3 additional text fields to be present on the Graphical User Interface along with a filter button that triggers the filter operation.

```
private JTextField filterAmount;  
private JTextField filterCategory;  
private JTextField filterDate;  
private JButton filterTransactionBtn;
```

We have to add these additional fields to the ExpenseTrackerView class. Moreover, we have to add get() and set() methods for these private variables, to ensure future extensibility, while maintaining the encapsulation aspect. For the filterTransactionBtn, a new addActionListener method would be created in the ExpenseTrackerApp class, to trigger the filter operation and change the entries in the table, as soon as the button is clicked. (Brief description of implementation is mentioned later)

Functionality with each field

In the filterAmount TextField, we have an option to allow users to filter transactions based on the amount, but this field has a restriction (applied on all transactions when they were added) - Filter amount should be between 0 and 1000. The filterAmount would be processed in a way such

that all the transactions having more amount value than the filterAmount would be displayed, and the rest would be hidden (simply using >= comparison operator).

Similarly the filterCategory will only accept the String values that are in the following list: ['food','travel','bills','entertainment','others']. Java being a case sensitive language, may disagree between 'food' and 'Food' being the same, therefore the comparison would be made using equalsIgnoreCase() method. Transactions with matching categories will be displayed on the view (table).

Lastly the date column would filter transactions based on the date, that is, all the transactions made on that current date will be displayed. Our timestamp is given in a String format, of pattern "dd-MM-yyyy HH:mm". We will split the date and time, using split() and then we will compare the first string, that is date, with the filterDate value using equals(). Requirements for acceptance of the date value in the filterDate variable is that it should follow the pattern 'dd-MM-yyyy', moreover, dd should be between 01 and 31, MM should be between 01 and 12, and yyyy should be between 0001 and current year. Later we can put more constraints such that if MM is 02, then the range of dd changes to 01-28, etc.

Working/Changes in the Methods/Classes

1. addActionListener() function (for filterTransactionBtn) - reads the values from the newly created fields (validates the input values using InputValidation class) and pass them to a new method called filter() in ExpenseTrackerView class as a Transaction
2. filter(Transaction t) - Create a new list of transactions called filterTransactions. We iterate over the already added transactions using a for loop, for each transaction we check if all the conditions are satisfied. If yes, we add that transaction to the filterTransactions list, otherwise we skip it. Once we are done iterating, we call the refreshTable() function, and pass filterTransactions variable as an argument to it.
3. validateFilterAmount(double amount), validateFilterDate(String date), validateFilterCategory(String category) - new functions that are used for validation inside the addActionListener() function of the filterTransactionBtn. If the values are not complying with the requirements, then an error message is displayed on the GUI. What the validation conditions would be and how they will be performed are explained in the previous section.