

Srinanda Kishore Yallapragada
Aditi Bansal
CS 520 Homework 1

Part 1: Manual Review:

Satisfied Non-Functional Requirements: Testability: Testability is a non functional requirement that describes the extent to which an application's code and features can be easily tested. The expense tracker application includes JUnit tests in ExpenseTrackerTest.java which has a JUnit testing setup and reflects the testability of the ExpenseTrackerView with a test called testAddTransaction.

```
@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());
}
}
```

Transaction.java is also set up to be tested easily as most of its public non-constructor methods are getter or setter functions whose effects can be validated easily through simple unit testing

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

public class Transaction {

    private double amount;
    private String category;
    private String timestamp;

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

    public double getAmount() {
        return amount;
    }

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

    public String getCategory() {
        return category;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public String getTimestamp() {
        return timestamp;
    }

    private String generateTimestamp() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy HH:mm");
        return sdf.format(new Date());
    }

}
```

Understandability: Understandability requires the application to be clear and documented enough so that another developer can understand, contribute and maintain the codebase with ease. Every part of the ExpenseTrackerApp class is commented to explain the purpose of the different parts of the code. This aids the understandability and readability of the code as a developer is not left wondering what certain function calls achieve.

```
// 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);
});
```

Violated Non-Functional Requirements: Extensibility: Extensibility describes how easy it is to accommodate future contributions to the code with minimal modifications to the existing codebase. Adding a new column to our application currently requires modifying the Transaction Java class, including adding a new property, getter, setter, and updating the constructor. This makes any extension to the transaction structure cumbersome. A possible fix would be to store the data as a list, allowing access via a function call similar to `transaction.get(AMOUNT)`, where AMOUNT is a numeric enum. This way, adding a new column would only require defining a new enum value, and appending an element to the list rather than modifying the entire class.

```
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");  
}
```

Modularity: Modularity requires the architecture of our software to be broken into components so that each has its own independent functionality and can be called from other functions as necessary. The refreshTable function should refresh the table in the view. However, it also includes the logic to sum the total cost which should in fact be in a separate function as it is not core to the refresh functionality. A possible fix is to separate out the code for the summation as a new function and call that new function from refreshTable.

```
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();  
    }  
  
    // Add rows from transactions list  
    for(Transaction t : transactions) {  
        model.addRow(new Object[]{rowNum+=1,t.getAmount(), t.getCategory(),  
t.getTimestamp()});  
    }  
    Object[] totalRow = {"Total", null, null, totalCost};  
    model.addRow(totalRow);  
  
    // Fire table update  
    transactionsTable.updateUI();  
}
```

Part 2: Modularity - MVC Architecture (24 Points)

2.1

A- Controller: The data is processed when the user hits the add transaction button. The user updates this component and not the model, so this would not be a view.

B- View: Displays transaction details from the stored data.

C- Controller: The component triggers a change to the model when clicked. it processes the data the user has inputted for the model and adds that thus its a controller.

2.2

Model:

Class:

Transaction.java

Fields:

private double amount;

private String category;

private String timestamp;

Methods:

public Transaction(double amount, String category)

public double getAmount()

public String getCategory()

public String getTimestamp()

private String generateTimestamp()

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;

public class Transaction {

    private double amount;
    private String category;
    private String timestamp;

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

    public double getAmount() {
        return amount;
    }

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

    public String getCategory() {
        return category;
    }


    public void setCategory(String category) {
        this.category = category;
    }

    public String getTimestamp() {
        return timestamp;
    }

    private String generateTimestamp() {
        SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy HH:mm");
        return sdf.format(new Date());
    }

}
```

Also we track this because it tracks all the transitions and that's part of the model in the sense that it acts like a database for as long as the application is running



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

View:

Class:

ExpenseTrackerView.java

Fields:

private JTable transactionsTable;

private DefaultTableModel model;

Methods:

public JTable getTransactionsTable()

public DefaultTableModel getTableModel()

public void refreshTable(List transactions)


```

private JTable transactionsTable;
private JTable transactionsTable;

public JTable getTransactionsTable() {
    return transactionsTable;
}

public void refresh() {

    // Get transactions from model
    List<Transaction> transactions = getTransactions();

    // Pass to view
    refreshTable(transactions);
}

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();
    }

    // Add rows from transactions list
    for(Transaction t : transactions) {
        model.addRow(new Object[]{rowNum+=1,t.getAmount(), t.getCategory(),
t.getTimestamp()});
    }
    Object[] totalRow = {"Total", null, null, totalCost};
    model.addRow(totalRow);

    // Fire table update
    transactionsTable.updateUI();
}

public DefaultTableModel getTableModel() {
    return model;
}

```

Controller:

Class:

ExpenseTrackerView.java

Fields:

```
private JTextField amountField;  
private JTextField categoryField;
```

Methods:

```
public double getAmountField()  
public void setAmountField(JTextField amountField)  
public double getCategoryFieldField()  
public void setCategoryField(JTextField categoryField)
```

```
public class ExpenseTrackerView extends JFrame {  
  
    private JTextField amountField;  
    private JTextField categoryField;  
  
    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();  
    }  
  
    public void setCategoryField(JTextField categoryField) {  
        this.categoryField = categoryField;  
    }  
  
    // Other view methods  
}
```

Part 3: Understandability - Documentation (12 Points)

PACKAGE: **CLASS** USE TREE INDEX SEARCH HELP

Unnamed Package > InputValidation

Contents

Description

Constructor Summary

Method Summary

Constructor Details

InputValidation()

Method Details

amountInputValidation(double)

categoryInputValidation(String)

Class InputValidation

java.lang.Object[?]
InputValidation

public class InputValidation
extends Object[?]

Constructor Summary

Constructors	Description
InputValidation()	

Method Summary

All Methods	Static Methods	Concrete Methods
Modifier and Type	Method	Description
static boolean	amountInputValidation(double amount)	
static boolean	categoryInputValidation(String [?] category)	

Methods Inherited from class java.lang.Object[?]

clone[?], equals[?], finalize[?], getClass[?], hashCode[?], notify[?], notifyAll[?], toString[?], wait[?], wait[?], wait[?]

Constructor Details

InputValidation

public InputValidation()

Method Details

amountInputValidation

public static boolean amountInputValidation(double amount)

categoryInputValidation

public static boolean categoryInputValidation(String[?] category)

127.0.0.1:5500/Documents/CSS20/hw1/expense_tracker/doc/InputValidation.html#amountInputValidation(double)

Jdocs generated and committed. please see github at <https://github.com/Srinanda-Yallapragada/hw1>.
git or uploaded .git folder

Part 4: Code Modification - Input Validation (12 Points)

```
if (!InputValidation.amountInputValidation(amount)) {
    JOptionPane.showMessageDialog(new JFrame(),
        "Amount needs to be between 0 and 1000");
    return;
}

if (!InputValidation.categoryInputValidation(category)) {
    JOptionPane.showMessageDialog(new JFrame(),
        "Category needs to be in food, travel, bills, entertainment or
other.");
    return;
}

public class InputValidation {

    public static boolean amountInputValidation(double amount) {
        if (amount < 1000 && amount > 0) {
            return true;
        }
        return false;
    }

    public static boolean categoryInputValidation(String category) {
        String[] validCategories = { "food", "travel", "bills", "entertainment",
"other" };

        for (String validCategory : validCategories) {
            if (validCategory.equals(category)) {
                return true;
            }
        }
        return false;
    }
}
```

Part 5: Extensibility - Adding a Filtering Feature (12 Points)

We would need to add an update view button so that we don't have to keep refreshing the view every time we update one of the filters. This update button would apply for category, amount and date based filters.

Category: We know that the only possible categories that we can filter the existing data from are "food", "travel", "bills", "entertainment", and "other". This means that our filtering will not need to take into account new categories.

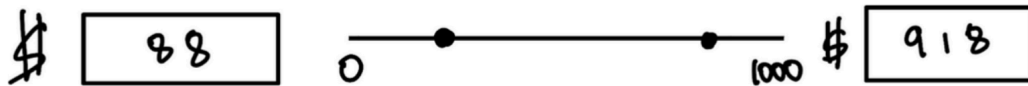
I believe that a checkbox based dropdown system would be the best for taking in the input on what categories to filter from. This would look something like the image below

FOOD	<input type="checkbox"/>
TRAVEL	<input checked="" type="checkbox"/>
BILLS	<input checked="" type="checkbox"/>
ENTERTAIN	<input type="checkbox"/>
OTHER	<input type="checkbox"/>

This design would allow users to select the categories they want to see and will also have multiple combinations of filters based on their needs. We would have a `filterCategory()` function that would

take in a list of the selected categories. This function would return a list of transactions that only contain the selected categories. We would implement this in ExpenseTrackerView as this modifies the view but does not make any updates to the model.

Amount: We know that the amount must be greater than 0 and less than a 1000 from the input validation section of this homework. We also know that the min value must be strictly lesser than the max value. I believe that having two input boxes, one for min and one for max, would be the appropriate way to get the information needed to filter my amount. I would use a slider with text boxes to achieve this. It would look like the image below



The text boxes allow the user to input finer margins while the slider gives a visual indication of the range that the user is viewing. We would have a filterAmount function that would take in two doubles as an input and return a list of transactions whose amount is between the minimum and maximum values given to the function. This would be implemented in the ExpenseTrackerView file as there are no updates made to the model and only the view is updated.

Date: Similar to the amount, we need two dates to create a range from which we can filter dates. We would have an input for a start date and an input for an end date. If we design it right, we do not have to change input validation. This would involve designing the UI such that it can only input a valid date. This can be visualized as seen below

Start date

11 / 11 / 05

End Date

12 / 12 / 06

^

< Nov 2005 >				
		1	2	3
4	5	6	7	8
9	10	.	.	.
-			-	

Similar to the other filter features, we would have a `filterDate()` function in the `ExpressTrackerView` file which would take in a start date and an end date as parameters. It would return a list of transactions between the start and end dates. We would not have to implement any validation checking (or minimal validation checking) as we have designed the input such that it only accepted valid dates using the UI.

Now we consider another function that would be implemented in `ExpressTrackerView` that is `filterData()` which takes in all the elements from the Category, Date and Amount and calls each of the filter functions. The order it calls the functions does not matter, but it would call the first filter, pass the results to the second filter and so on until we get a final list of filtered transactions that meet all the criteria provided by the user.

There would also be modifications in the code of `ExpressTrackerApp` to update the loop of using the application. There would be a listener for the update view button which would call the `filterData` function and update the view to show the user the filtered transaction they requested.

`ExpressTrackerApp` would have the following additions A checkbox dropdown for category filtering. A min/max input field (with optional slider) for amount filtering. Two date pickers for date filtering. An “Update View” button that triggers filtering when clicked. Updates to the loop and a listener for the update view button

`ExpressTrackerView` would have the following functions used to implement the filter functions
`getSelectedCategories()` - user selected categories
`getMinFilterAmount()` - min amount set by user
`getMaxFilterAmount()` - max amount set by user
`getStartDateFilter()` - start date set by user
`getEndDateFilter()` - end date set by user