HW 1

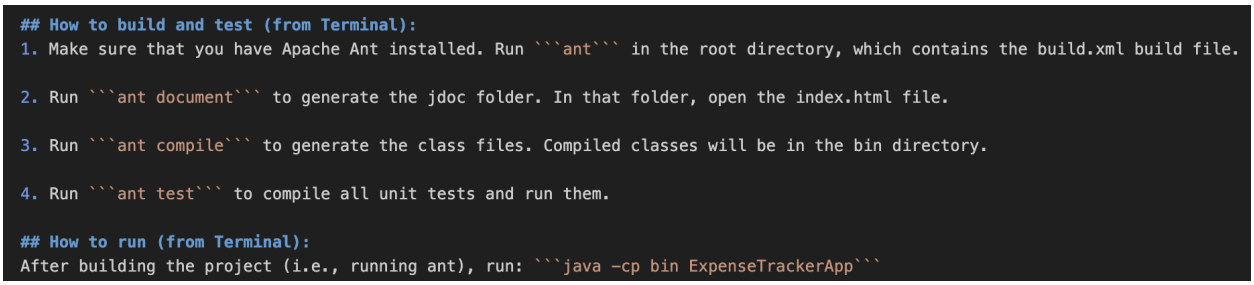
Manual Review

Satisfies

1. **Understandability**

Summary: Understandability is about how easy it is for a developer who is not experienced with the code base to jump right in and be able to understand the application. A codebase that is understandable is easy to get started with, having good documentation describing what the code does.

Example: The application contains a detailed README.md with instructions and details of what the codebase is built with (Java and Apache Ant) and how to compile and run the program as shown below:



```
## How to build and test (from Terminal):
1. Make sure that you have Apache Ant installed. Run ```ant``` in the root directory, which contains the build.xml build file.

2. Run ```ant document``` to generate the jdoc folder. In that folder, open the index.html file.

3. Run ```ant compile``` to generate the class files. Compiled classes will be in the bin directory.

4. Run ```ant test``` to compile all unit tests and run them.

## How to run (from Terminal):
After building the project (i.e., running ant), run: ```java -cp bin ExpenseTrackerApp```
```

Without this piece of documentation, it would have been incredibly unclear how to get started especially from someone not familiar with Java.

Summary:

2. **Testability**

Summary: Testability is about how testing capability and coverage within an application. A testable codebase would have a high test coverage, ensuring app quality and predictability. A poor testable code would be difficult to write tests.

Example: The transaction logic is nicely tied into the Transaction class. This makes it easy to test the transaction logic as you just have to create different instances and test

the expected outcomes from the methods. JUnit is used to test these cases such as the add case as shown below:

```
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());
}
```

Violates

1. **Modularity**

Summary: Modularity is about how divided the codebase is. A modular codebase will have separated pieces of logic with every class trying to do just one thing. A poor modular codebase will have a lot of logic tied in the same place.

Example & Explanation: The logic of the app is hardly split especially with the view. The add transaction, inputs, and table are all contained in the same JFrame class which can be seen below:

```
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);
```

Say there comes a point where more functionality wants to be added such as to the table, the class will quickly expand, juggling a whole swarm of features. This would be difficult to scale as new features have to be built and balanced with all the other view code in the application rather than a small few modifications in just certain views. It also makes it more difficult for more collaborators as they have to constantly worry about merge conflicts since one JFrame represents the whole UI.

Solution: Break up the ExpenseTrackerView JFrame into multiple jframes: one for the table and one for the form with a parent app jframe. These two components are different enough where they can be separated. The table can be set up to receive values from the app jframe and the form can update the values stored in the app frame such as adding an element to the transactions array.

2. **Extensibility**

Summary: Extensibility is about how difficult it is to extend/grow the code base for new features. A codebase with good extensibility can add new features and changes fairly easy while a codebase with poor extensibility would have a difficult time to add new capabilities.

Example & Explanation: The controller logic is largely done by the DefaultTableModel class where has listeners for actions/changes as shown below:

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



ExpenseTrackerView view = new ExpenseTrackerView(tableModel);
```

However, using a default controller for Tables can be a bit limited currently. Say more than the default logic wants to be added such as calculating a total from a table of entries, than the current controller isn't very suitable for adding this feature.

Solution: The solution is as simple as creating a new class perhaps called TransactionTableController and extending the DefaultTableModel to keep the current logic and then adding methods to it for not just more connivence such as adding a transaction but adding more features like directly deleting a transaction or finding the balance summation.

Modularity: MVC architecture pattern

2.1.1) Component A: This component is a bit of both as it displays the view for the form but as well contains the information that will be used by the action listener to update the table with new values.

2.1.2) Component B: This component is largely a view in that it simply displays the list of transaction in a table ui with no direction features for editing or modifying the table presented region. This component is in charge of just displaying the transaction data and is thus a view.

2.1.3) Component C: This component is largely a controller in that it listens for action events of being pressed to then trigger a change in the table, adding a transaction from the form, creating a new row to component B.

2.2.1) Model: The model is the Transaction class which handles storing and managing access to the main business logic data of the application.

```java
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();
  }
```

2.2.2) One view: One view would be Component B which is built with the default JTable class as shown below:

```java
transactionsTable = new JTable(model);
```

2.2.3) One controller: The controller logic is largely contained in the DefaultTableModel and button action listener which contains code to update the data table:

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

Extensibility: Proposed extension

To add filtering for transaction, creating a new class TransactionFilter that accepts an arraylist as a paramter for it's constructor. Then create public methods such as amountGreaterThan(double value). In that method create a duplicate arraylist as to be predictable then modify it according to the filter, in this case remove elements less than or equal to the value parameter then return a new instance of the TransactionFilter that takes in the filtered arraylist. The next step is to add views for managing this filter, in this case we can create an input with label Greater Than with a button to submit filter change. This view can be display below the add transaction form. The submit button of the view would modify the table model to apply new array list of transaction and refresh the table.