CS520 Software Engineering
Fall 2023 Heather Conboy HW1

1 Manual Review

1.1 Satisfied Non-functional Requirements

1. Understandability

   - Users and developers can easily see the project and understand what features are there, as well as how to run the code and how it should function.
   - The project structure is well maintained: source code, lib, README and test cases are well separated. `/expense_tracker/README.md` file is clear and easy to follow and provides information to start from the basic installation to run the full project.

2. Testability

   - We find this program is easily testable, meaning the program can be tested and bugs can be located and fixed using these tests.
   - We note that the project is already structured to support testing, as there is already an existing `/test/ExpenseTrackerTest.java` which can be extended with additional tests if desired. Moreover, the build file is already set to run tests with a simple `ant test` command.

1.2 Violated Non-functional Requirements

1. Debuggability

   - The program can be difficult for developers to pinpoint issues and errors in when they encounter unexpected results or crashes.
   - Java version is not clearly mentioned. This will cause a lot problem while installing the java and mess up the environment if wrong version is installed. The correct java version is 21 or above for this code.
   - For instance, the code is entirely lacking in any error handling or logging. In `ExpenseTrackerView.java` in the `getAmountField()` function, the text in the field is assumed to be a double and is parsed as such. If a non-numerical value were to be inputted instead, the program would throw a long error message with no help in finding the issue.
   - The authors of the program made poor design choices in not including any input validation or error handling when writing the code, and also left out any logging in the program.
   - Most obviously, we would recommend adding input validation, error handling, and logging to the code in order to make the code more debuggable. We would also recommend adding more modularity to the code by splitting functions across more classes and files to make it easier to pinpoint issues to certain areas. Each of these could help future developers fix bugs when they encounter them.

2. Readability

- The code is difficult to read, meaning new developers who see the code for the first time find it hard to understand how the code is functioning.
- One obvious thing that stands out in the code is the lack of javadoc comments in all the functions. For instance, in `ExpenseTrackerView.java` none of the functions have javadoc comments or even basic descriptions. Also, the `ExpenseTrackerView.java` is very long compared to the other two files, making it even moer difficult to read. Finally, the authors almost never user `this` when referencing variables, making the readability low for almost no benefit.
- The authors of the code omitted writing javadoc comments in their code, which seems to us like an obvious design flaw. They also chose to put most of the functionality of the code into a single file, `ExpenseTrackerView.java`, when they could have split the functionality more evenly.
- We would recommend adding javadoc comments to the code in order to increase readability. Once again, the authors could also split up their code between files more effectively to make it more readable for future developers.

2 Modularity: MVC Architecture Pattern

2.1 Application UI

1. Component A: Controller. In the top section, the user is prompted to enter an amount and category into the text boxes. This implies the user is interacting with the controller when they enter these numbers to the textbox in the UI.

2. Component B: View, We can view all the entries that we have added using Component A and Component B. Further it's also showing the sum of the added items. This component doesn't have any interactive button or edit options therefore we cannot term as controller.

3. Component C: Controller. In the bottom section, the user is able to press the "Add Transaction" button in order to send a request to the model that adds their entered transaction to the database. This means the user is interacting with the controller.

2.2 Source Code

1. Model: The main variable which serves as the model in the app is `List<Transaction> transaction` in `ExpenseTrackerView`. There are a few methods which serve as the model that I would point to, mainly `refreshTable, refresh`, as they update the view with data internal to the model, and `addTransaction` which updates the internal database and the view at the same time. `addTransaction` interacts with both the view and the controller, meaning it must be associated with the model.

2. One view: It very clear that Component B is used mainly to display user's entry and give the summations of all amount. There neither interactive button or session nor does any critical operation. Some functions from this session are: `ExpenseTrackerView()`. It handles all the view

3. One controller: Component C is made up of a button which serves to interact with the user. It is specified in the source code as the variable `JButton addTransactionBtn`. It can be gotten using the `getAddTransaction` method. I would argue the call to `view.getAddTransactionBtn().addActionListener(...)` would be code associated with the controller in component C, as this part of the code takes input from the user by responding to when the button is clicked.

3 Extensibility: Proposed extension

Our goal is to add to the ExpenseTrackerApp with a filtering feature, which allows the user to enter filters based on category, amount, or date. I would implement this using a few changes:

1. I would create a new class, `TransactionFilter.java` which is an object that has a field which stores the **type** of filter as being "category", "amount", or "date", as well as a field which stores the **value** of the filter. Finally, there would also be a field storing whether we want transactions which are greater than, less than, or equal to the value of the filter.

2. `TransactionFilter.java` would have a method `checkFilter(Transaction t)` which returns `true` if the transaction passes the filter and `false` otherwise.

3. I would add a new button to the UI in the `ExpenseTrackerView` constructor which opens a new window where the user can enter a filter to place on the app. There would be a field for choosing what to filter by, the value to filter, and which type of filter (less than, greater than, equal to) to apply.

4. Once this button is used and a filter is created, I would update a new field in `ExpenseTrackerView` which generates a new `TransactionFilter` object with the given input filter (after some input validation) and store to this.filter.

5. I would update `refreshTable` by adding an if statement to **only count transactions for which** `this.filter.checkFilter` returns true. If `this.filter` is still not set, then I would ignore this if statement and always include all transactions. This if statement would be in the for loop iterating over transaction to get the total cost as well as the loop to add to the model.