Modularity, Open-Closed Principle, and Encapsulation

Overview

The application is an Expense Tracker, which allows users to track their financial transactions. This document outlines the design principles and patterns used to ensure modularity, encapsulation, and immutability in the application.

Modularity and Open-Closed Principle

Modularity is the degree to which a system's components may be separated and recombined. The Open-Closed Principle (OCP) states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

In the context of the Expense Tracker application, the `Transaction` class and the `TransactionFilterInterface` are designed to be modular and adhere to the OCP. New types of transactions or new filter criteria can be added without modifying the existing code.

Encapsulation and Immutability

Encapsulation is the bundling of data with the methods that operate on that data. Immutability means that an object's state cannot be modified after it is created.

In the Expense Tracker application, the list of transactions is encapsulated within the `ExpenseTrackerView` class. The `getFilter` method returns a filter, not the actual list of transactions, ensuring encapsulation.

To ensure immutability, the `Transaction` class is designed so that once a `Transaction` object is created, its data cannot be modified. This is achieved by making all fields `final` and removing any setter methods.

```
public final class Transaction {

    private final String category;

    private final double amount;

    public Transaction(String category, double amount) {

        this.category = category;

        this.amount = amount;

    }
```

```
    public String getCategory() {
        return category;
    }
    public double getAmount() {
        return amount;
    }
}
```

In this design, the `Transaction` class is immutable. Any attempt to change the category or amount of a transaction will result in a compile-time error.

Conclusion

The design of the Expense Tracker application adheres to the principles of modularity, the Open-Closed Principle, encapsulation, and immutability. This design ensures that the application is flexible, robust, and easy to maintain.

Transaction Filter Interface and Strategy Design Pattern

Overview

The application is an Expense Tracker, which allows users to track their financial transactions. The application provides a feature to filter transactions based on certain criteria. The Transaction Filter Interface and the Strategy Design Pattern are used to implement this feature.

Transaction Filter Interface

The Transaction Filter Interface is an interface that defines a method for filtering transactions. Any class that implements this interface must provide an implementation for this method. The method takes a list of transactions as input and returns a list of transactions that meet the filter criteria.

```
public interface TransactionFilterInterface {
    List<Transaction> filter(List<Transaction> transactions);
}
```

Strategy Design Pattern

The Strategy Design Pattern is used to select the filter criteria at runtime. This pattern involves defining a family of algorithms (in this case, the filter criteria), encapsulating each one, and making them interchangeable. The pattern lets the algorithm vary independently from clients that use it.

In the context of the Expense Tracker application, the Strategy Design Pattern is used to switch between different filter criteria (e.g., Category Filter, Amount Filter) at runtime.

Category Filter

The Category Filter is a class that implements the Transaction Filter Interface. It filters transactions based on their category.

```
public class CategoryFilter implements TransactionFilterInterface {

    private String category;

    public CategoryFilter(String category) {

        this.category = category;

    }

    @Override

    public List<Transaction> filter(List<Transaction> transactions) {

        // Filter logic here

    }

}
```

Amount Filter

The Amount Filter is another class that implements the Transaction Filter Interface. It filters transactions based on their amount.

```
public class AmountFilter implements TransactionFilterInterface {

    private double amount;

    public AmountFilter(double amount) {

        this.amount = amount;

    }
```

```
    @Override
    public List<Transaction> filter(List<Transaction> transactions) {
        // Filter logic here
    }
}
```

Usage

The user can select the filter criteria at runtime. The selected filter is then used to filter the transactions.

```
TransactionFilterInterface filter = new CategoryFilter("Food");
List<Transaction> filteredTransactions = filter.filter(transactions);
```

In this example, the Category Filter is used to filter transactions that belong to the "Food" category. The filtered transactions are then highlighted in the UI.

Conclusion

The Transaction Filter Interface and the Strategy Design Pattern provide a flexible and extensible way to filter transactions in the Expense Tracker application. The design allows for easy addition of new filter criteria in the future.

Undo Functionality Design Document

Overview

The undo functionality will allow users to revert their last action of removing a transaction. This action will be reflected in the Total Cost as well.

Model

The model will need to maintain a stack of transactions that have been removed. Each time a transaction is removed, it should be pushed onto the stack. When an undo action is performed, the model should pop the last transaction from the stack and add it back to the list of transactions.

```
Stack<Transaction> removedTransactions = new Stack<>();
public void removeTransaction(Transaction transaction) {
    transactions.remove(transaction);
```

```
    removedTransactions.push(transaction);
}
public void undoRemoveTransaction() {
    if (!removedTransactions.isEmpty()) {
        Transaction transaction = removedTransactions.pop();
        transactions.add(transaction);
    }
}
```

View

The view will need a new button for the undo action. When this button is clicked, it should trigger the undo action in the controller.

```
JButton undoButton = new JButton("Undo");
undoButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        controller.undoRemoveTransaction();
    }
});
```

Controller

The controller will need a new method to handle the undo action. This method should call the undo method in the model and then refresh the view.

```
public void undoRemoveTransaction() {
    model.undoRemoveTransaction();
    view.refresh();
}
```

Conclusion

This design will allow the user to undo their last remove transaction action. The MVC architecture pattern is maintained, with the model handling the data, the view handling the user interface, and the controller handling the communication between the model and the view.