Criterion C: Development

Data Structure

I chose to use the ArrayList to store the transactions for the program. It allows for an unknown number of transactions. Also, it makes it easy to add, remove, and store data. This makes it a good fit for this program because adding and editing transactions are central functions of this program.

While I was writing my program, I implemented various programming techniques. Below is a list of some of the techniques that I used.

List of Techniques

- Class Decomposition
- Method Decomposition
- Sorting
- Searching
- File I/O
- Tables
- Expense Analysis
- Nested Loops
- Complex Selection
- Flags
- Graphical User Interface

Class Decomposition

I decomposed the budget application into twelve different classes: AddFilters, AdjustBudget, Conversions, CurrencyTableCellRenderer, DisplayFilter, EditTransactions, FileIO, HouseholdBudget, SpendingAnalysis, SpendingData, Transaction, and ViewRemainingData. Decomposing the program into classes simplified the process of designing, writing, and debugging the code. I have included a section of the code from the Transaction class.

```java
public class Transaction implements java.io.Serializable {

    //declare variables
    Category category;
    double price;
    int month;
    int year;
    boolean recurring;
    int everyMonth;

    //copy constructor
    Transaction(Transaction transactionToCopy) {
        category = transactionToCopy.category;
        price = transactionToCopy.price;
        month = transactionToCopy.month;
        year = transactionToCopy.year;
        recurring = transactionToCopy.recurring;
        everyMonth = transactionToCopy.everyMonth;
    }
```

Figure 1 - Class Decomposition

The Transaction class encapsulates the properties of each transaction. This grouping of data allowed me to create the ArrayList of Transactions.

Method Decomposition

Method decomposition allowed me to organize my code based on the different tasks that need to be implemented in the program. The following method returns the ArrayList of transactions.

```java
//reads the current transactions
public static ArrayList<Transaction> readTransactions() {
    ArrayList<Transaction> transactionsToReturn = null;

    try {
        FileInputStream fileInput;
        ObjectInputStream objectInput;

        fileInput = new FileInputStream("/tmp/transactions");
        objectInput = new ObjectInputStream(fileInput);

        transactionsToReturn = (ArrayList<Transaction>) objectInput.readObject();

        objectInput.close();
        fileInput.close();

        //System.out.println("Read data for object array.");

    } catch (IOException error) {
        error.printStackTrace();
        return transactionsToReturn;
    } catch (ClassNotFoundException error) {
        System.out.println("Couldn't find Transaction class.");
        error.printStackTrace();
        return transactionsToReturn;
    }

    return transactionsToReturn;
}
```

Figure 2: Method Decomposition

Sorting

In order to create a clear display of transactions in the table and to allow the user to easily find specific transactions, I sorted the transactions by year and by month so that they are in chronological order. The following methods were used to do this.

```
private void incrementMonth() {
    //increments the month
    month += 1;
    //if the month goes past December, reset to January and increment year
    if (month == 13) {
        month = 1;
        year += 1;
    }
}

private Transaction next() {
    if (month == 1 && year == 2029) {
        return null;
    }

    Transaction transactionToReturn = new Transaction(this);

    //if this transaction isn't recurring, set the next one to be null
    if (!recurring) {
        month = 1;
        year = 2029;
    //otherwise, set month to next calendar month
    } else {
        incrementMonth();
    }

    return transactionToReturn;
}
```

Figure 3 - Sorting

Searching

In order to sort and limit the transactions in the table, I implemented a sorting algorithm to only show certain categories, month, year, or whether it is single or recurring based on the user's selections in four combo boxes. Since there are not very many data items, I chose to use linear search instead of a binary search. Below is the code I used to filter the type of transaction, whether it was single or recurring.

```
//checks whether transaction passes transaction filter
static public boolean passesTransactionFilter(Transaction transaction, DisplayFilter displayFilter) {
    boolean boolToReturn = false;

    switch (displayFilter) {
        case Both:
            boolToReturn = true;
            break;
        case Single:
            if (!transaction.getRecurring()) {
                boolToReturn = true;
            }
            break;
        case Recurring:
            if (transaction.getRecurring()) {
                boolToReturn = true;
            }
            break;
    }

    return boolToReturn;
}
```

Figure 4 - Searching

File I/O

I chose to use serialization to read and write the transactions to disk. This is a good option because the ArrayList class supports serializing. I used the following code to read and write the data.

```
//reads the current transactions
public static ArrayList<Transaction> readTransactions() {
    ArrayList<Transaction> transactionsToReturn = null;

    try {
        FileInputStream fileInput;
        ObjectInputStream objectInput;

        fileInput = new FileInputStream("/tmp/transactions");
        objectInput = new ObjectInputStream(fileInput);

        transactionsToReturn = (ArrayList<Transaction>) objectInput.readObject();

        objectInput.close();
        fileInput.close();

        //System.out.println("Read data for object array.");

    } catch (IOException error) {
        error.printStackTrace();
        return transactionsToReturn;
    } catch (ClassNotFoundException error) {
        System.out.println("Couldn't find Transaction class.");
        error.printStackTrace();
        return transactionsToReturn;
    }

    return transactionsToReturn;
}
```

Figure 5 – Read Transactions

```java
//writes the current transactions
public static void writeTransactions(ArrayList<Transaction>  transactions) {

    try {

        FileOutputStream fileOutput;
        ObjectOutputStream objectOutput;

        fileOutput = new FileOutputStream("/tmp/transactions");
        objectOutput = new ObjectOutputStream(fileOutput);

        objectOutput.writeObject(transactions);

        objectOutput.close();
        fileOutput.close();

    } catch (IOException error) {
        error.printStackTrace();
    }

}
```

Figure 6 – Write Transactions

Table

Since the main purpose of budget program is to keep track of transactions with set properties, I decided that a jTable would be an effective display. Below is the code I used to set up the table model.

```java
public EditTransactions() {

    initComponents();

    //sets table columns
    model = new DefaultTableModel();
                model.addColumn("Month");
                model.addColumn("Year");
                model.addColumn("Amount");
                model.addColumn("Category");
                model.addColumn("Recurring");
                model.addColumn("#/Month");

    //sets model
    jTableTransactions.setModel(model);

    //sets column widths
    jTableTransactions.getColumnModel().getColumn(0).setPreferredWidth(125);
    jTableTransactions.getColumnModel().getColumn(1).setPreferredWidth(65);
    jTableTransactions.getColumnModel().getColumn(2).setPreferredWidth(125);
    jTableTransactions.getColumnModel().getColumn(3).setPreferredWidth(200);
    jTableTransactions.getColumnModel().getColumn(4).setPreferredWidth(100);
    jTableTransactions.getColumnModel().getColumn(5).setPreferredWidth(100);

    //formats the money in the "amount" column of the table
    jTableTransactions.getColumnModel().getColumn(2).setCellRenderer(new CurrencyTableCellRenderer());

    readFromFile();
    updateDisplay();

}
```

Figure 7 – Set Table Model

Expense Analysis

Based on my client's request for the program, I decided to make an ExpenseAnalysis class to display the annual statistics to the user, including the category analysis and price statistics. I used the following if-else statement to calculate the amount of money spent in each category for a selected year and the percent of the total spending for the amount spent in each category. I set the percentages to 0.00 if no transactions have been added since the algorithm for calculating the percentages would divide by zero if the total spending were 0. I decided to create a separate class for the spending data variables so they do not have to be declared as global variables. This improves the code's extensibility.

```
//display category analysis
//total in each category
jLabelBills.setText("$" + String.format("%,.2f", spendingData.costBills));
jLabelKids.setText("$" + String.format("%,.2f", spendingData.costKids));
jLabelDining.setText("$" + String.format("%,.2f", spendingData.costDining));
jLabelGroceries.setText("$" + String.format("%,.2f", spendingData.costGroceries));
jLabelEntertainment.setText("$" + String.format("%,.2f", spendingData.costEntertainment));
jLabelPersonal.setText("$" + String.format("%,.2f", spendingData.costPersonal));
jLabelHealth.setText("$" + String.format("%,.2f", spendingData.costHealth));
jLabelGifts.setText("$" + String.format("%,.2f", spendingData.costGifts));
jLabelInvestments.setText("$" + String.format("%,.2f", spendingData.costInvestments));
jLabelMiscellaneous.setText("$" + String.format("%,.2f", spendingData.costMiscellaneous));

//percentage of total spending for each category
if (spendingData.totalAmountSpent != 0) {
    jLabelBillsPercentage.setText(String.format("%,.2f", (spendingData.costBills/spendingData.totalAmountSpent) * 100) + "%");
    jLabelKidsPercentage.setText(String.format("%,.2f", (spendingData.costKids/spendingData.totalAmountSpent) * 100) + "%");
    jLabelDiningPercentage.setText(String.format("%,.2f", (spendingData.costDining/spendingData.totalAmountSpent) * 100) + "%");
    jLabelGroceriesPercentage.setText(String.format("%,.2f", (spendingData.costGroceries/spendingData.totalAmountSpent) * 100) + "%");
    jLabelEntertainmentPercentage.setText(String.format("%,.2f", (spendingData.costEntertainment/spendingData.totalAmountSpent) * 100) + "%"
    jLabelPersonalPercentage.setText(String.format("%,.2f", (spendingData.costPersonal/spendingData.totalAmountSpent) * 100) + "%");
    jLabelHealthPercentage.setText(String.format("%,.2f", (spendingData.costHealth/spendingData.totalAmountSpent) * 100) + "%");
    jLabelGiftsPercentage.setText(String.format("%,.2f", (spendingData.costGifts/spendingData.totalAmountSpent) * 100) + "%");
    jLabelInvestmentsPercentage.setText(String.format("%,.2f", (spendingData.costInvestments/spendingData.totalAmountSpent) * 100) + "%");
    jLabelMiscellaneousPercentage.setText(String.format("%,.2f", (spendingData.costMiscellaneous/spendingData.totalAmountSpent) * 100) + "%"
} else {
    //set percentage to 0 if no transactions have been added
    jLabelBillsPercentage.setText("0.00%");
    jLabelKidsPercentage.setText("0.00%");
    jLabelDiningPercentage.setText("0.00%");
    jLabelGroceriesPercentage.setText("0.00%");
    jLabelEntertainmentPercentage.setText("0.00%");
    jLabelPersonalPercentage.setText("0.00%");
```

Figure 8 – Expense Analysis

Calculate Remaining Money

In order to allow the user to calculate the amount of money remaining in the budget for a particular month, I read the current budget from FileIO and then subtract the amount spent from a particular month from that budget by multiplying the price of the transaction by how many times it occurs each month. I use the following code to implement this calculator:

```
private void jButtonViewActionPerformed(java.awt.event.ActionEvent evt) {

    int year = Integer.valueOf((String)jComboBoxYear.getSelectedItem());

    String monthAsString = (String)jComboBoxMonth.getSelectedItem();
    int month = Conversions.convertStringToMonth(monthAsString);
    double remainingBudget = FileIO.readCurrentBudget();

    //message box prompting user to set their budget
    if (remainingBudget == 0){
        JOptionPane.showMessageDialog(null, "Please set your monthly budget.");
    }

    //calculate remaining budget
    else {
        ArrayList<Transaction> transactions = FileIO.readTransactions();

        for(Transaction transaction : transactions){
            if (transaction.month == month && transaction.year == year) {
                remainingBudget -= transaction.getPrice() * transaction.getEveryMonth();
            }
        }

        //display to user
        jTextFieldMoneyRemaining.setText("$" + String.format("%,.2f", remainingBudget));

        if (remainingBudget < 0){
            //Notify the user if they are overbudget using pop-up dialog
            JOptionPane.showMessageDialog(null, "You are $" + String.format("%,.2f", -1 * remainingBudget) + " overbudget!!");
        }
    }

}
```

Figure 9 – Calculate Remaining Money

Complex Selection/Flags

In order to give the user the option of either updating one occurrence of a recurring transaction or all future occurrences of the recurring transaction, I used a nested loop. If the user chooses to update all future occurrences, the flag is set to true to update all the future occurrences that stem from the selected transaction.

```
if (transaction.recurring || original.recurring) {
    String[] options = {"This One", "All Future", "Cancel"};
    int selection = JOptionPane.showOptionDialog(null,
            "Do you want to update only this transaction, or all future ones as well?",
            "Update Recurring Transaction",
            JOptionPane.DEFAULT_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,
            options,
            options[0]);

    //if user decides to cancel, add the row back and exit
    if (selection == 2) {
        transactions.add(rowSelected, original);
        return;

    //if user decides to update all future selections, set flag to true
    } else if (selection == 1) {
        updateFuture = true;
    }
}
```

Figure 10 – Complex Selection/Flags

Graphical User Interface

Since the user needs to interact with the different extensively, I tied the program together using a system of buttons, JForms, and jDialogues. I used the jDialogues when another element of the program will not work until the dialog is closed. The following code is an example of when I connected the HouseholdBudget form to the other forms and/or dialogues using an action event on a button in the user interface.

```
    private void jButtonAdjustBudgetActionPerformed(java.awt.event.ActionEvent evt) {

        //creates an instance of the Adjust Budget dialog
        AdjustBudget showDlg = new AdjustBudget(this, true);

        //displays the child dialog
        showDlg.setVisible(true);

    }

    private void jButtonViewMoneyLeftActionPerformed(java.awt.event.ActionEvent evt) {

        //creates an instance of the View Remaining Money dialog
        ViewRemainingMoney showDlg = new ViewRemainingMoney(this, true);

        //displays the child dialog
        showDlg.setVisible(true);

    }

    private void jButtonAddExpensesActionPerformed(java.awt.event.ActionEvent evt) {

        //creates an instance of the Edit Transactions dialog
        EditTransactions transactionFrame = new EditTransactions();

        //displays the child dialog
        transactionFrame.setVisible(true);

    }
```

(768 words)