Criterion C: Development

The product is a Java program, written in the Eclipse IDE. The program's functionality allows a user to create and register an account to access the program. The user can then select an existing spreadsheet and make modifications or create an entirely new spreadsheet. All inputted data should be saved in text files and the spreadsheet is saved in an Excel file.

The program is executable by clicking the "ExpenseManager.jar" file, (Figure 1) and a login screen will be displayed (Figure 2).

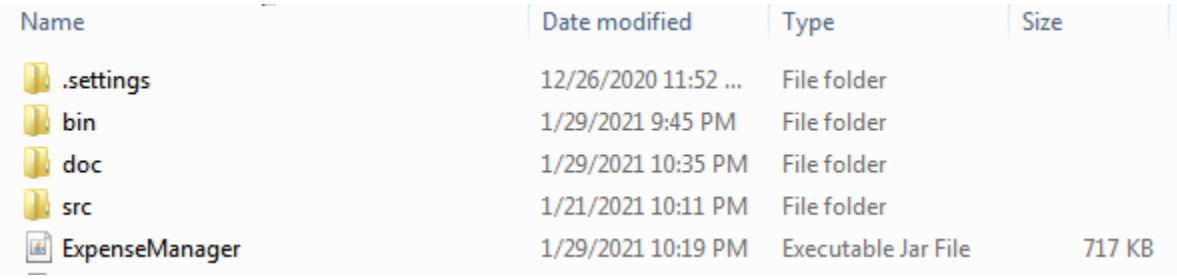


Figure 1: Executable JAR file.

Login Window:

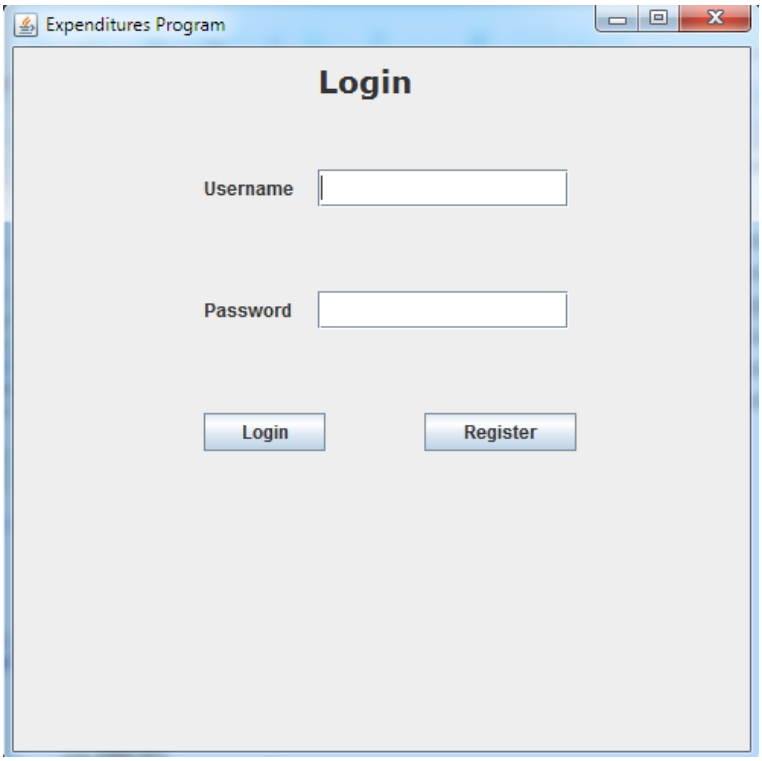


Figure 2: Login Screen

For the graphical user interface, I used javax.swing including JFrames, JPanels, JButtons, JLabels, JTextFields, and JPasswordFields in the entire application because it is easy and efficient (Figure 3).

```
// declaration of instance fields
private JLabel userLabel;
private JLabel passwordLabel;
private JLabel title;
private static JTextField userText;
private JPasswordField passwordText;
private JButton loginButton;
private JButton registerButton;
private static JPanel loginPanel;
```

Figure 3: javax.swing objects in the Login class

Computational Techniques and Algorithms:

1. Encapsulation:

Contrary to my design in Criterion B, my client asked for a more practical interface when adding expenses. Instead of using income and outcome, and the specification of dates, my client proposed that I sort the expense by month, hence I created the Expense class.

I used encapsulation in the Expense class (Figure 4) because this ensured inputted values by the user are valid for the spreadsheet and more than one instance of the Expense can be run at the same time. The private field in the Expense class, including the expenseName, can only be modified by the public void setExpenseName(String name) (Figure 5) method, and accessed through the public double getExpenseName() method (Figure 6). This method was used in the EditExpense class (Figure 7).

```
public class Expense {

    private String expenseName;
    private double januaryExpenses;
    private double februaryExpenses;
    private double marchExpenses;
    private double aprilExpenses;
    private double mayExpenses;
    private double juneExpenses;
    private double julyExpenses;
    private double augustExpenses;
    private double septemberExpenses;
    private double octoberExpenses;
    private double novemberExpenses;
    private double decemberExpenses;
```

Figure 4: Encapsulated fields of Expense class

```
public void setExpenseName(String newName) {
    expenseName = newName;
}
```

Figure 5: Sample of encapsulated modifier method in class Expense

```
public String getExpenseName() {
    return expenseName;
}
```

Figure 6: Sample of encapsulated accessor method in class Expense.

```
} else if (isEditedClicked == false){
    expense.setExpenseName(expenseName.getText());
} else {
```

Figure 7: Use of encapsulated modifier method in class EditExpense

2. LinkedList:

I created a custom singly linked list class, called ExpenseLinkedList (Figure 9), with nodes defined by the class ExpenseNode (Figure 10), which holds objects of type Expense. I used this abstract data type over an array because the number of expenses that can be stored is not constant. When the user clicks “Add an Expense” or “Delete an Expense”, the Spreadsheet class (Figure 8) will be notified and perform its appropriate action by calling methods in the ExpenseLinkedList class. These methods will be explained in Table 1.

```
public class Spreadsheet extends AbstractTableModel {

    // testing with column names and cells
    private String[] months = {"Expense Name", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sept", "Oct", "Nov", "Dec",
            "Yearly Total" };

    private ExpenseLinkedList list;
    private String sheetName;

    /**
     * Constructor
     */
    public Spreadsheet(String name, ExpenseLinkedList list) {
        sheetName = name;
        this.list = list;
    }
```

Figure 8: Spreadsheet class taking in an ExpenseLinkedList as a parameter

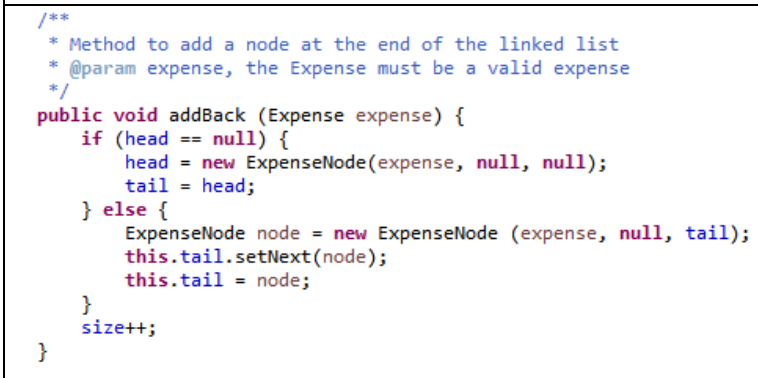
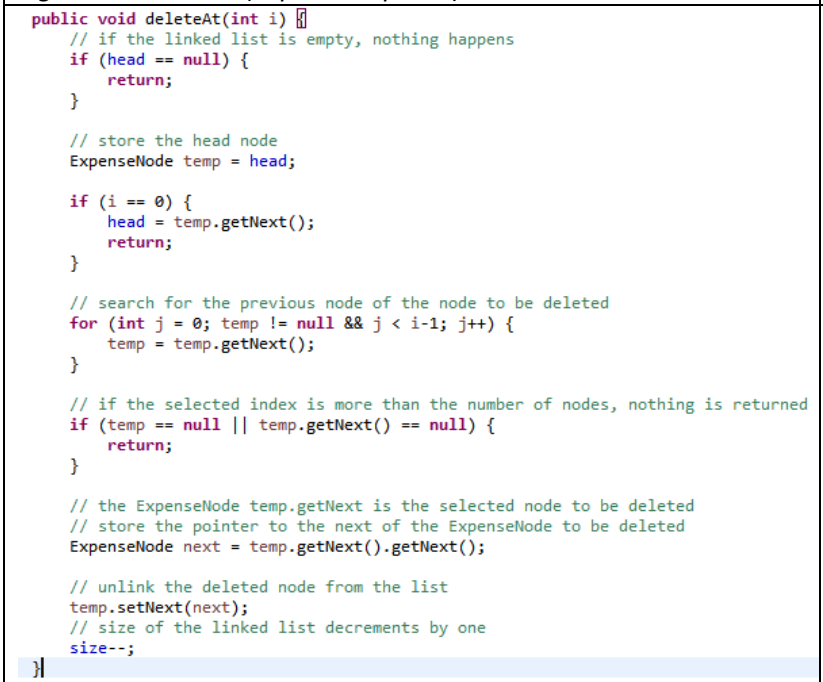
```
public class ExpenseLinkedList {

    private ExpenseNode head;
    private ExpenseNode tail;
    private int size = 0;

    /**
     * Method returns the size of the linked list
     * @return the private variable size
     */
    public int getSize() {
        return size;
    }
```

Figure 9: Encapsulated instance fields of ExpenseNode in ExpenseLinkedList

```java
public class ExpenseNode {

    private ExpenseNode next;
    private ExpenseNode prev;
    private Expense exp;

    public ExpenseNode(Expense exp, ExpenseNode next, ExpenseNode prev) {
        this.exp = exp;
        this.next = next;
        this.prev = prev;
    }

    public void setValue (Expense exp) {
        this.exp = exp;
    }

    public void setNext (ExpenseNode node) {
        next = node;
    }

    public void setPrev (ExpenseNode node) {
        prev = node;
    }

    public Expense getExpense() {
        return exp;
    }

    public ExpenseNode getNext() {
        return next;
    }

    public ExpenseNode getPrev() {
        return prev;
    }

}
```

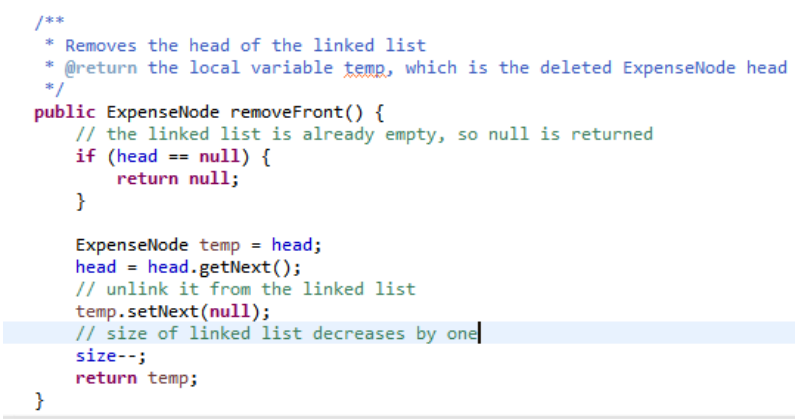
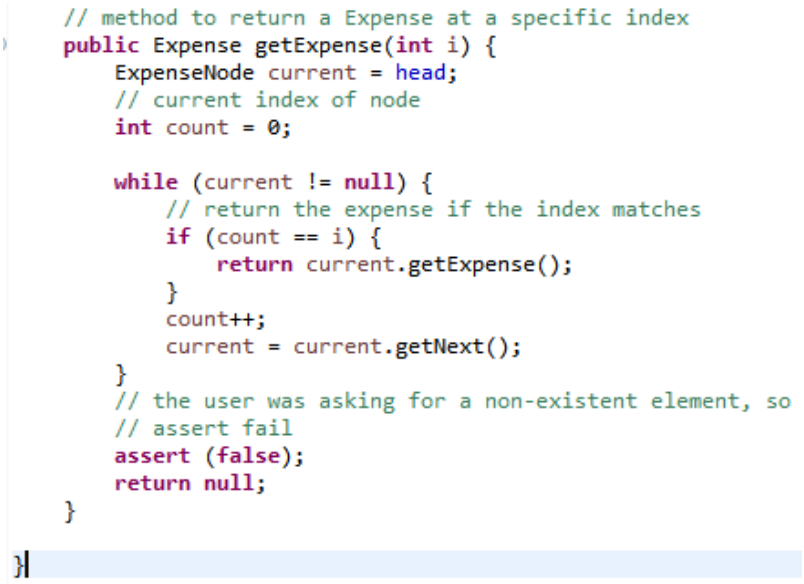
Figure 10: Encapsulated variables and methods of ExpenseNode.

Table 1: Methods of the ExpenseLinkedList class

Method	How it works	Practicality in program
 *Figure 11: addBack(Expense expense) method*	- Expense is added to end of linked list - Node is now "tail" of linked list	The expense is displayed at the bottom of the spreadsheet.
 *Figure 12: deleteAt(int i, Expense expense) method*	- ExpenseNode at user's specified position is deleted from linked list - Linked list must change node in front of it as original node was removed	The user can select any expense and remove them easily. The information of the expense will deleted

```
/**
 * Method to add a node at the end of the linked list
 * @param expense, the Expense must be a valid expense
 */
public void addBack (Expense expense) {
    if (head == null) {
        head = new ExpenseNode(expense, null, null);
        tail = head;
    } else {
        ExpenseNode node = new ExpenseNode (expense, null, tail);
        this.tail.setNext(node);
        this.tail = node;
    }
    size++;
}
```

```
public void deleteAt(int i) {
    // if the linked list is empty, nothing happens
    if (head == null) {
        return;
    }

    // store the head node
    ExpenseNode temp = head;

    if (i == 0) {
        head = temp.getNext();
        return;
    }

    // search for the previous node of the node to be deleted
    for (int j = 0; temp != null && j < i-1; j++) {
        temp = temp.getNext();
    }

    // if the selected index is more than the number of nodes, nothing is returned
    if (temp == null || temp.getNext() == null) {
        return;
    }

    // the ExpenseNode temp.getNext is the selected node to be deleted
    // store the pointer to the next of the ExpenseNode to be deleted
    ExpenseNode next = temp.getNext().getNext();

    // unlink the deleted node from the list
    temp.setNext(next);
    // size of the linked list decrements by one
    size--;
}
```

 *Figure 13: removeFront() method*	- Head of linked list is removed by unlinking current head of linked list - Sets the new head as the next ExpenseNode - Returns original head	If the user selects to remove the first expense in the spreadsheet, the expense will not be stored in the linked list and the row will be removed.
 *Figure 14 The getExpense(int i) method*	- Traverse through linked list until index of ExpenseNode matches with index i - Returns details of expense by calling getExpense() method in ExpenseNode class - Asserts false if caller asks for "non-existent element" (Geeksforgeeks .com, 2020)	If the user selects an expense in the table to edit, the user will be able to see the data stored for that expense in the EditExpense JFrame.

3. Inheritance:

In my Spreadsheet class, I extended Java's built-in AbstractTableModel because I wanted to create my custom JTable. The AbstractTableModel is an "abstract class that provides default implementations for most of the methods in the TableModel interface". (Docs.Oracle.com, 2021). By using the inherited methods of getRowCount(), getColumnCount(), and getValueAt(int row, int column) (Figure 15, 16), I was able to create my own version of a JTable by accessing the number of rows and columns, and retrieving values by using a switch case algorithm to retrieve values for each expense.

```
public int getColumnCount() {
    return months.length;
}

public int getRowCount() {
    return list.getSize();
}
```

Figure 15: Sample inherited methods from the AbstractTableModel class.

```
public Object getValueAt(int rowIndex, int colIndex) {
    Expense expense = list.getExpense(rowIndex);
    if (expense == null) {
        return null;
    }

    switch (colIndex) {
    case 0:
        return expense.getExpenseName();
    case 1:
        return expense.getJanuary();
    case 2:
        return expense.getFebruary();
    case 3:
        return expense.getMarch();
    case 4:
        return expense.getApril();
    case 5:
        return expense.getMay();
    case 6:
        return expense.getJune();
    case 7:
        return expense.getJuly();
    case 8:
        return expense.getAugust();
    case 9:
        return expense.getSeptember();
    case 10:
        return expense.getOctober();
    case 11:
        return expense.getNovember();
    case 12:
        return expense.getDecember();
    case 13:
        return expense.getYearlyTotal();
    default:
        return null;
    }
```

Figure 16: Inherited getValueAt(int rowIndex, int colIndex) from AbstractDataModel class

Although the methods outlined in Figures 15 and 16 are the only three methods required for a “concrete TableModel”, I included other inherited methods such as the setValueAt(Object value, int rowIndex, int colIndex) to modify the contents of an expense (Figure 17) (Docs.Oracle.com, 2021). However, some

methods were created including addExpense(Expense expense) and removeExpense (Figure 18) in the class to suit the success criteria of adding and deleting expenses in the spreadsheet.

To update the contents of the JTable, the inherited methods of fireTableRowsInserted(), fireTableRowsDeleted(), and fireTableDataChanged() are called (Figures 17 and 18)

```
double doubleValue = 0;

if (colIndex == 0) {
    expense.setExpenseName(strValue);
}

try {
    doubleValue = Double.valueOf(strValue);
} catch (NumberFormatException e) {
    e.getMessage();
}

switch (colIndex) {
case 1:
      expense.setJanuary(doubleValue);
case 2:
      expense.setFebruary(doubleValue);
case 3:
      expense.setMarch(doubleValue);
case 4:
      expense.setApril(doubleValue);
case 5:
      expense.setMay(doubleValue);
case 6:
      expense.setJune(doubleValue);
case 7:
      expense.setJuly(doubleValue);
case 8:
      expense.setAugust(doubleValue);
case 9:
      expense.setSeptember(doubleValue);
case 10:
      expense.setOctober(doubleValue);
case 11:
      expense.setNovember(doubleValue);
case 12:
      expense.setDecember(doubleValue);
}
fireTableRowsUpdated(rowIndex, colIndex);
```

Figure 17: Body of setValueAt(Object value, int rowIndex, int colIndex) method in Spreadsheet class

```java
/**
 * Method to remove an expense from the spreadsheet
 * @param i, the index of the row must be specified
 */
public void removeExpense(int i) {
    // if the
    if (i == 0) {
        list.removeFront();
    } else {
        list.deleteAt(i);
    }
    fireTableRowsDeleted(i, i);
    fireTableDataChanged();
}


/**
 * Method to add an expense to the spreadsheet
 * @param expense, a valid Expense must be entered
 */
public void addExpense(Expense expense) {
    int row = list.getSize();
    list.addBack(expense);
    fireTableRowsInserted(row, row);
}
```

Figure 18: removeExpense(int i) and addExpense(Expense expense) methods in the Spreadsheet class

Other inherited methods came from JFrames and JPanels for classes, (e.g. Login and Register respectively), shown in Figure 19. These methods were used to set the basic layout and close operations of the window and panel.

```java
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.RandomAccessFile;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

public class Login extends JFrame {

    // declaration of instance fields
    private JLabel userLabel;
    private JLabel passwordLabel;
    private JLabel title;
    private static JTextField userText;
    private JPasswordField passwordText;
    private JButton loginButton;
    private JButton registerButton;
    private static JPanel loginPanel;

    // constructor
    public Login() {
        // sets the title, size, and close operation of the window
        super("Expenditures Program");
        setSize(500, 500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```java
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

public class Register extends JPanel {

    private static boolean isRegistered = false;

    // text field declaration
    private static JTextField userText;
    private static JPasswordField passwordText;

    // button declaration
    private static JButton register;
    private static JButton back;

    // label declaration
    private static JLabel userLabel;
    private static JLabel passwordLabel;
    private static JLabel title;

    // string declaration
    private String username;
    private String password;


    public Register() {
        super(new BorderLayout());

        // adds title to the middle of the panel
        title = new JLabel("Account Registration");
        title.setBounds(125, 10, 300, 25);
        title.setFont(new Font ("Verdana", 1, 20));
        add(title);
```

Figure 19: Login and Register class that extends JFrame and JPanel respectively, inheriting methods from its superclass.

4. Saving Files

One of the success criteria of the project was to ensure that the data from the table is saved. I created a method to export the JTable as an Excel file (.xlsx). Once the spreadsheet is saved, the user will be notified that it was saved by a pop-up message (Figure 20).

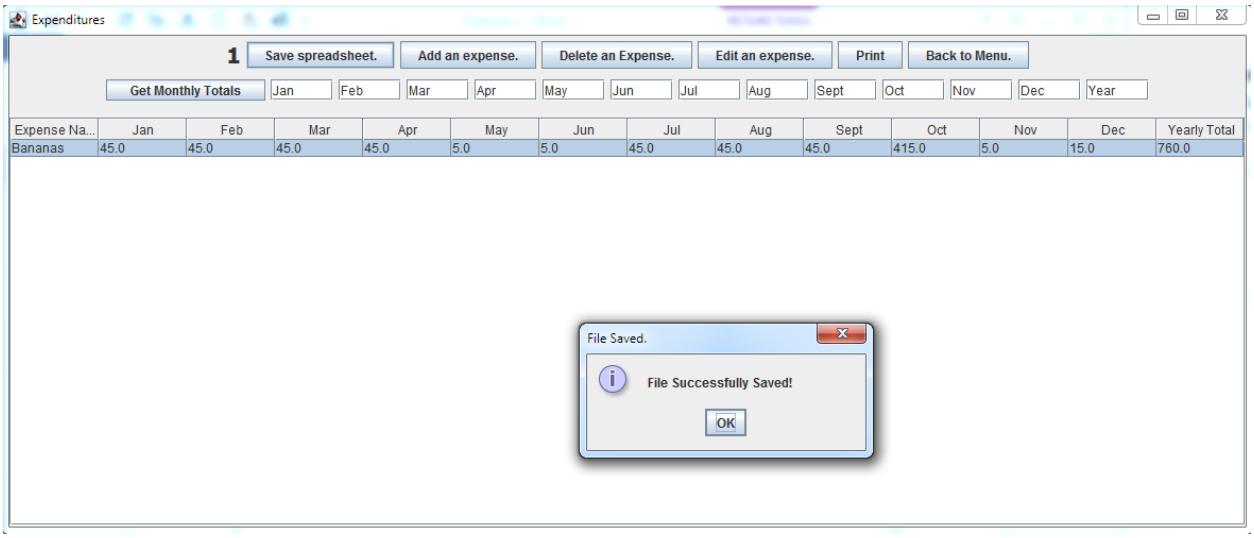


Figure 20: Pop-up message after saving the spreadsheet.

This was created in the DisplaySpreadsheet class by calling the method saveSheet(). It implements a “for loop” to first name the column headers, and then uses a nested “for loop” to save the contents of each expense into a cell in the Excel file (Figure 21).

```
/
 * Method that saves the spreadsheet as an Excel file and a text file
 */
public void saveSheet() {
    // saving a copy as an Excel file
    boolean ifSaved = false;
    try {
        File tableInExcel = new File(spreadsheetName + ".xls");
        TableModel model = spreadsheet.getModel();
        FileWriter excel = new FileWriter(tableInExcel);
        // writes the column names into Excel
        for (int i = 0; i < model.getColumnCount(); i++) {
            excel.write(model.getColumnName(i) + "\t");
        }
        excel.write("\n");
        // writes a new expense on a new line
        for (int i = 0; i < model.getColumnCount(); i++) {
            for (int j = 0; j < model.getColumnCount(); j++) {
                String data = String.valueOf(model.getValueAt(i, j));
                if (data == "null") {
                    data = "";
                }
                excel.write(data + "\t");
            }
            excel.write("\n");
        }
        excel.close();
        ifSaved = true;
    } catch (IOException e) {
        Logger.getLogger(DisplaySpreadsheet.class.getName()).log(Level.SEVERE, null, e);
    }
```

Figure 21: saveSheet() method in DisplaySpreadsheet class

Once the user opens the Excel file, the contents of each expense is displayed in the order the user saved in the program (Figure 22).

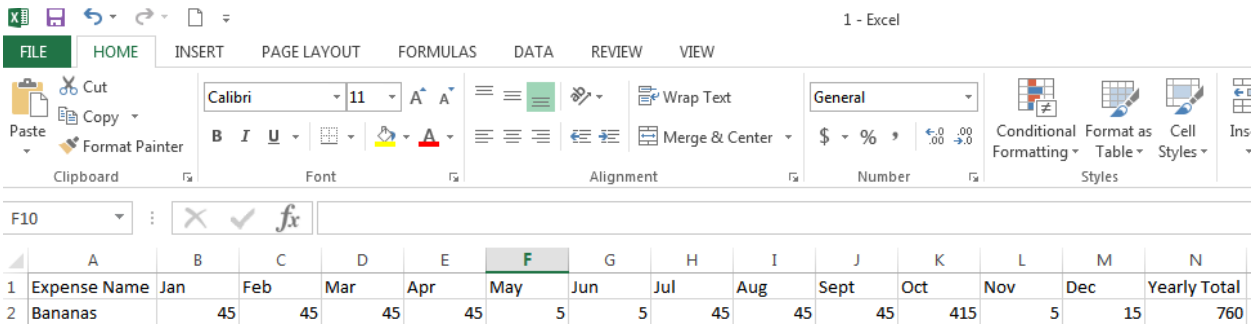


Figure 22: Sample Excel file of saved spreadsheet.

5. Loading Spreadsheets:

Not only does the saveSheet() method save the user's spreadsheet as an .xlsx file, a text file is created with the table contents. I chose to create a separate text file because it would be easier to load the spreadsheet back into the table, meeting another point in the project's success criteria. A FileWriter and a BufferedWriter is used to write down the contents of the spreadsheet, with an asterisk separating the value of each cell (Figure 23). This code was adapted from the Stack Overflow community.

```
// saving a copy as text file, easier to load the spreadsheet
String filePath = spreadsheetName + ".txt";
File file = new File(filePath);
try {
    FileWriter fw = new FileWriter(file);
    BufferedWriter bw = new BufferedWriter(fw);
    for (int i = 0; i < spreadsheet.getRowCount(); i++) {
        for (int j = 0; j < spreadsheet.getColumnCount(); j++) {
            bw.write(String.valueOf(spreadsheet.getValueAt(i, j)));
            // separate each entry with an asterisk
            bw.write("*");
        }
        // new line for a new expense
        bw.newLine();
    }
    bw.close();
    fw.close();
    ifSaved = true;
} catch (IOException e) {
    Logger.getLogger(DisplaySpreadsheet.class.getName()).log(Level.SEVERE, null, e);
}
```

Figure 23: Converting the spreadsheet into a text file

Next, the name of the spreadsheet is stored in separate text file called listOfSpreadsheets.txt. This contains the names of all existing spreadsheets with no overlap in names to avoid confusion in loading (Figure 24).

```
/**
 * Method that stores the name of the spreadsheet being used
 */
public static void storeSpreadsheetNames() {
    boolean alreadyExists = false;
    try {
        // file name is specified
        File file = new File ("listOfSpreadsheets.txt");
        // create the file if the file is not already created
        if (!file.exists()) {
            file.createNewFile();
        }

        Scanner scanner = new Scanner (file);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            // if the spreadsheet name already exists in the file, this text file is not changed
            if (line.equals(spreadsheetName)) {
                alreadyExists = true;
            }
        }
        scanner.close();

        if (!alreadyExists) {
            // if the spreadsheet name does not exist, add the name of the spreadsheet to the text file
            FileWriter fw = new FileWriter(file,true);
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);
            pw.println(spreadsheetName);
            pw.close();
        }

    } catch (IOException e) {
        Logger.getLogger(DisplaySpreadsheet.class.getName()).log(Level.SEVERE, null, e);
    }
}
```

Figure 24: storeSpreadsheetsNames() method in the DisplaySpreadsheet class stores the name of every spreadsheet created on a separate line

This text file will be parsed through during in the SpreadsheetSelect class and initialize a String array containing these names (Figure 25).

```
private void addButtonsSpreadsheets() {
    // local variable that counts the number of lines in the listOfSpreadsheets text file
    int counter = 0;
    File spreadsheetNames = new File ("listOfSpreadsheets.txt");
    try {
        FileReader fr = new FileReader(spreadsheetNames);
        BufferedReader br = new BufferedReader(fr);
        String line = br.readLine();
        while (line != null) {
            // determines the number of spreadsheet names in the text file
            counter++;
            line = br.readLine();
        }
        br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }

    // creates new array to store the spreadsheet names in the program
    array = new String[counter];
    // reset the counter to initialize the contents in the array
    counter = 0;
    // reads the file again
    File spreadsheetNames2 = new File ("listOfSpreadsheets.txt");
    try {
        FileReader fr = new FileReader(spreadsheetNames2);
        BufferedReader br = new BufferedReader(fr);
        String line = br.readLine();
        while (line != null) {
            // store the spreadsheet names into an array
            array[counter] = line;
            counter++;
            line = br.readLine();
        }
        br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 25: Private method addButtonsSpreadsheets() in the SpreadsheetSelect class

Each saved spreadsheet will have its own button, and added to the main panel. When clicked, the private method loadSpreadsheet(String name) is called, and takes in the name of the spreadsheet as a parameter. (Figure 26).

```
for(int i = 0; i < array.length; i++) {
    // add a new button for each spreadsheet name
    JButton button = new JButton (array[i]);
    int temp = i;
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            // when button is clicked, load the spreadsheet and dispose the window
            loadSpreadsheet(array[temp]);
            dispose();
        }
    });
gbc.gridy++;
// adds the buttons to the screen
mainPanel.add(button, gbc);
}
```

Figure 26: Adding a button for each spreadsheet in the SpreadsheetSelect class

This calls the saved text file of the spreadsheet, reads the contents of the table through an array, and sets the values by calling on the various modifier methods in the Expense class. Ultimately, a "new" DisplaySpreadsheet() class is created and returns the saved spreadsheet (Figure 27).

```
/**
 * Loads the spreadsheet that the user chooses to click
 * @param sheetName, takes in the name of the spreadsheet
 * On button click, the saved spreadsheet will be displayed
 */
private void loadSpreadsheet(String sheetName) {
    // creates a new spreadsheet displaying the saved information
    Spreadsheet sheet = new Spreadsheet(sheetName, new ExpenseLinkedList());
    new DisplaySpreadsheet(sheet);
    String filePath = sheetName + ".txt";

    File file = new File(filePath);

    try {
        FileReader fr = new FileReader(file);
        BufferedReader br = new BufferedReader(fr);
        Object [] lines = br.lines().toArray();

        for (int i = 0; i < lines.length; i++) {
            String [] parts = lines[i].toString().split("\\*");
            Expense loadExpense = new Expense();
            loadExpense.setExpenseName(parts[0]);
            for (int j = 1; j < parts.length; j++) {
                switch (j) {
                case 1:
                    loadExpense.setJanuary(Double.parseDouble(parts[j]));
                case 2:
                    loadExpense.setFebruary(Double.parseDouble(parts[j]));
                case 3:
                    loadExpense.setMarch(Double.parseDouble(parts[j]));
                case 4:
                    loadExpense.setApril(Double.parseDouble(parts[j]));
                case 5:
                    loadExpense.setMay(Double.parseDouble(parts[j]));
```

Figure 27: Sample of method loadSpreadsheet(String name) in the SpreadsheetSelect class

6. Basic Encryption

When the user registers for an account, their credentials are saved as a text file and the password is encrypted using the Caesar cipher to promote security. When the user logs in, the saved password is decrypted and compared to the user input to determine the user's correct credentials (Figure 28).

```
// method to create an account for the user
// account for error when nothing is entered and button is clicked
public void createAccount() {
    // gets the user input in the username and password text fields
    String username = userText.getText();
    String password = passwordText.getText();
    String modifiedPassword = "";

    //String savedCredentials = "";
    try {
        // creates a new text file storing the user's credentials
        RandomAccessFile credentials = new RandomAccessFile("users.txt", "rw");
        for (int i = 0; i < password.length(); ++i) {
            // encodes the password using the Caesar cipher by shifting characters of password back three letters
            int k = password.charAt(i) - '\u0003';
            modifiedPassword += (char)k;
        }
        credentials.writeBytes(username + "\\" + modifiedPassword + "\n");
        credentials.writeBytes("");
    } catch (IOException e) {
        System.out.println("Error with saving file.");
        isRegistered = false;
    }

/**
 * Method that determines whether the user's username and password match
 * @return true if the username and password match the credentials saved in the text file, otherwise return false
 */
public boolean checkCredentials() {
    boolean loggedIn = false;
    try {
        RandomAccessFile credentials = new RandomAccessFile("users.txt", "r");

        String readFile = credentials.readLine();
        String name = readFile.substring(0, readFile.indexOf("\\"));
        String savedPassword = readFile.substring(readFile.indexOf("\\") + 1);

        String password = "";

        //
        for (int i = 0; i < savedPassword.length(); ++i) {
            int j = savedPassword.charAt(i) + '\u0003';
            password += (char)j;
        }

        if (name.equals(userText.getText()) && password.equals(passwordText.getText())) {
            loggedIn = true;
        } else {
            loggedIn = false;
        }
        credentials.close();
    } catch (IOException e) {
        e.getMessage();
    }
    return loggedIn;
}
```

Figure 28: createAccount() method from the Register class and checkCredentials() method from the Login class

7. Print

Upon further discussion with my client, a suggestion was to include an option to print the spreadsheet. Therefore, I created the print button where the program will try to print the current list of expenses on button click. The code below was adapted from Docs.oracle.com. (Figure 29).

```
private void print() {
    MessageFormat header = new MessageFormat("Print Expenses");
    try {
        boolean complete = spreadsheet.print(JTable.PrintMode.NORMAL, header, null);
        if (complete) {
            JOptionPane.showMessageDialog(null, "Printing successful!");
        }
    } catch (PrinterException e) {
        JOptionPane.showMessageDialog(null, "Could not print.", "Error", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

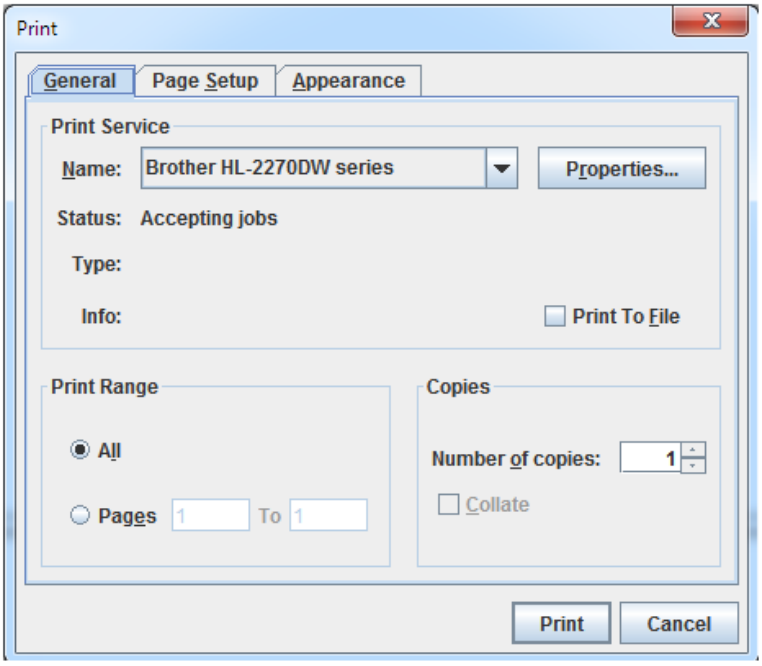


Figure 29: print() method in DisplaySpreadsheet

Changes to Design:

My final program has incorporated many changes compared to its initial design because of feedback from the ICS teacher and my client (see Appendix D). The initial design can be found in Criterion B and Figure 30 represents the new design.

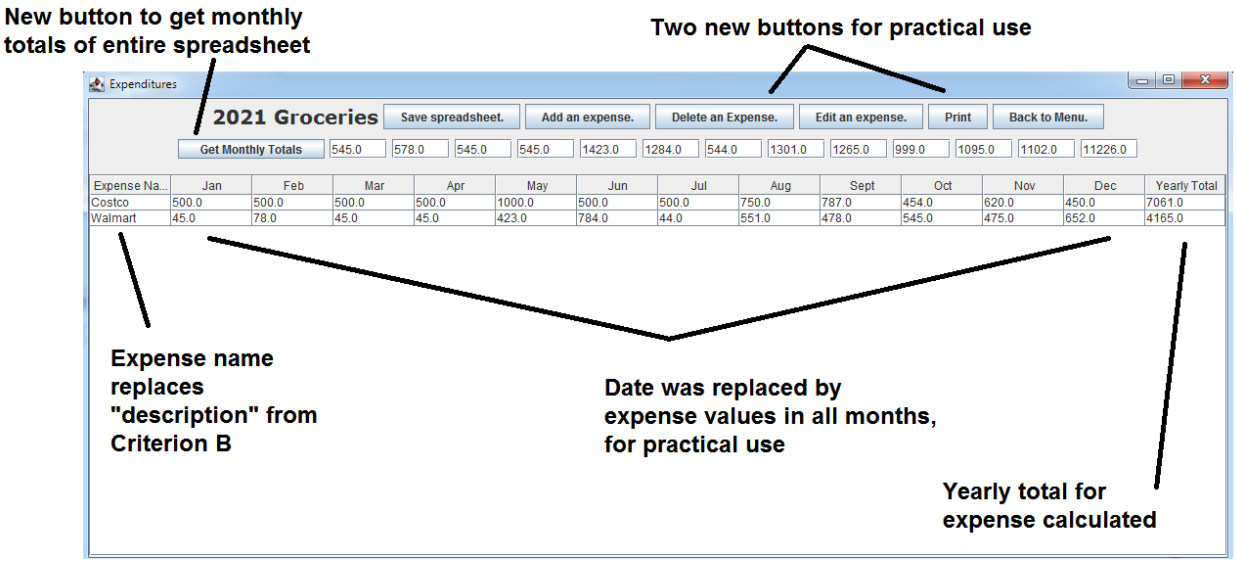


Figure 30: Sample view of spreadsheet program with annotations describing new changes

Total words in Criterion C: 966 words (excluding figure captions, references, and tables)

Works Cited:

Docs.Oracle.Com. "How To Print Tables (The Java™ Tutorials > Creating A GUI With JFC/Swing > Using Other Swing Features) ". Docs.Oracle.Com, 2021, https://docs.oracle.com/javase/tutorial/uiswing/misc/printtable.html. Accessed 29 Jan 2021.

Geeksforgeeks. "Write A Function To Get Nth Node In A Linked List". Geeksforgeeks, 2009, https://www.geeksforgeeks.org/write-a-function-to-get-nth-node-in-a-linked-list/. Accessed 29 Jan 2021.

StackOverflow Community, “Export JTable in Excel file”, Stack Overflow, 2014, https://stackoverflow.com/questions/22560566/export-jtable-in-excel-file. Accessed 30 Jan 2021.