The Structure of the Product

Figure 1: Represents all the classes used in the application

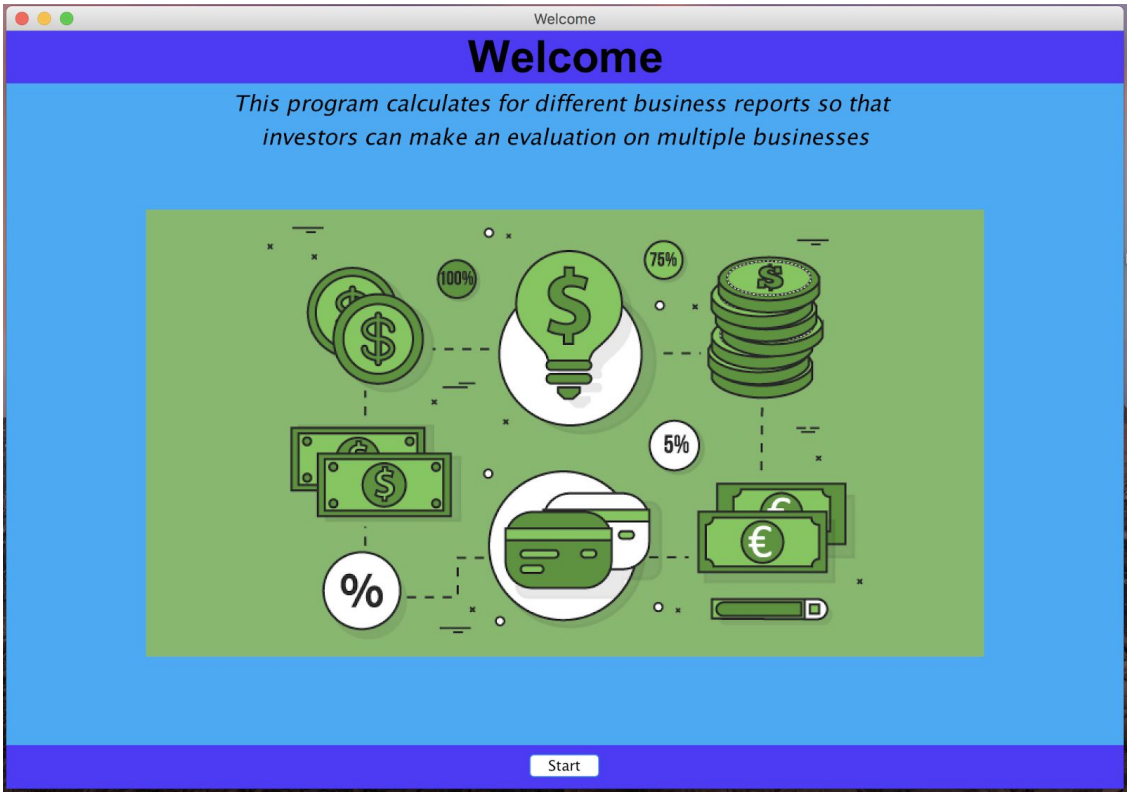

Figure 2: Welcome frame that introduces the user to the application

```
@Override
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();

    if (command.equals("Start"))
    {
        this.dispose();
        LogIn logInObj = new LogIn();
    }
}
```

Figure 2.1: Action performed for the welcome frame and how it navigates to different GUI frames

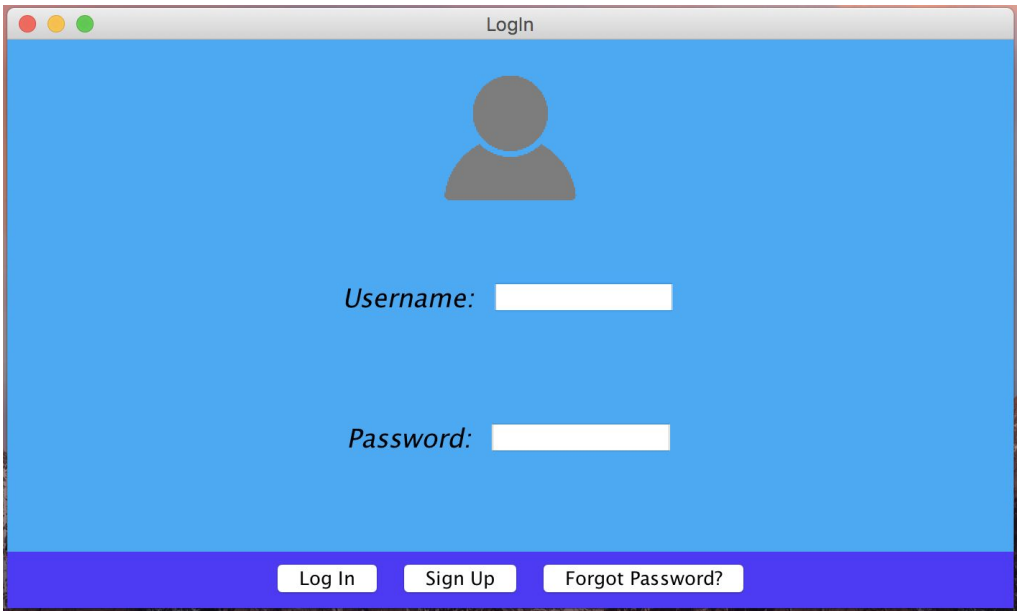

Figure 3: Log In frame that allows for the user to log in with their account onto the application

```
else if (command.equals("Log In"))
{
    //Checks if the log in information is valid
    if (logInObj.checkLogIn(userNameField.getText(),
            convertPassword(passwordField.getPassword())))
    {
        email = logInObj.findEmail(userNameField.getText(),
                convertPassword(passwordField.getPassword()));

        Input inputObj = new Input(email);
        this.dispose();
    }
    else
    {
        Error errorObj = new Error("Error: Invalid Username Or Password");
    }
}
```

Figure 3.1: Action performed for the log in frame and how it navigates to different GUI frames

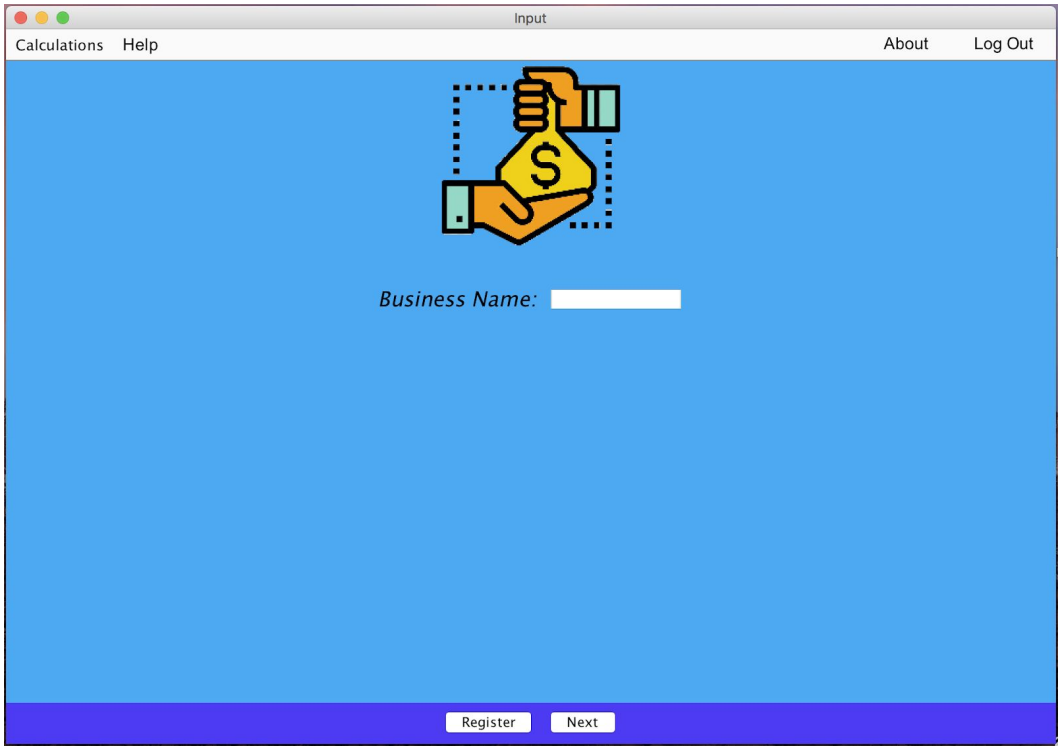

Figure 4: Input frame panel that allows for the user to register and change business information

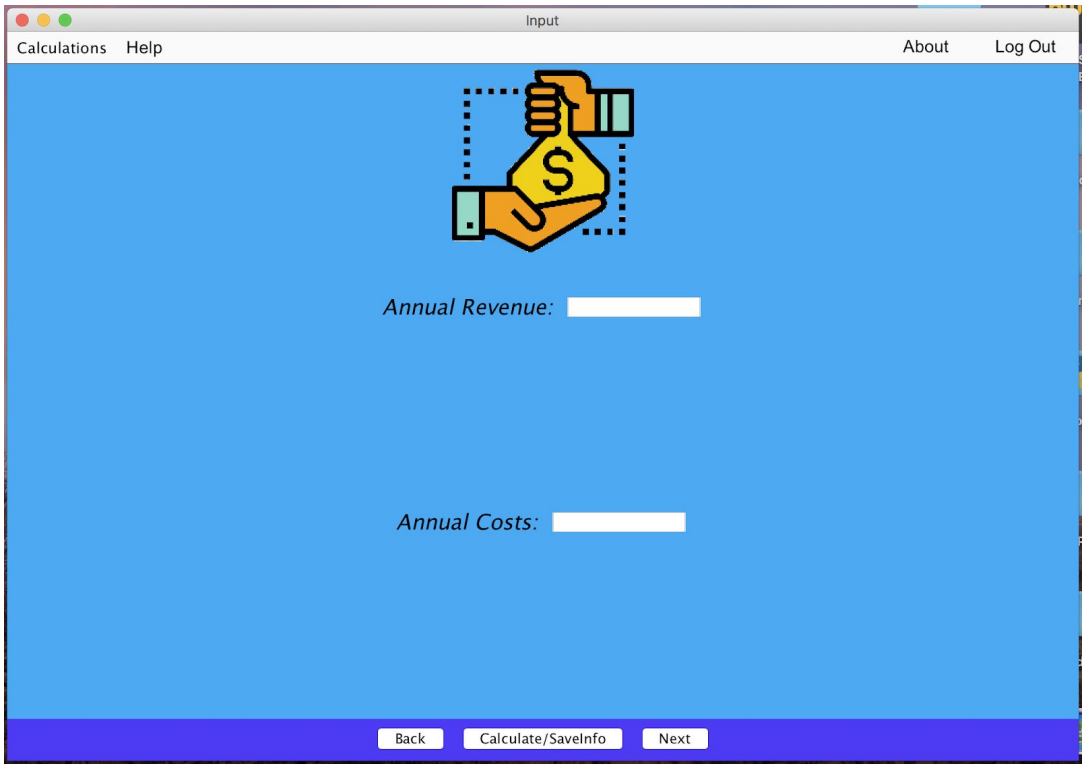

Figure 5: Input frame that allows for the user to enter general information for specified business

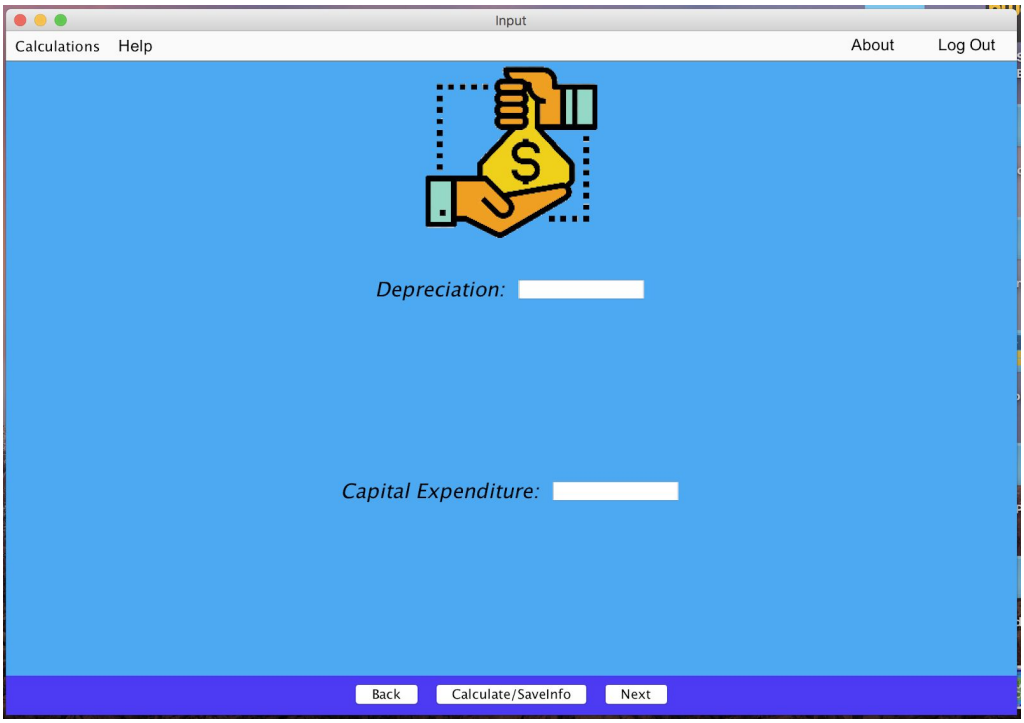

Figure 6: Input frame that allows for the user to enter cash flow information for specified business

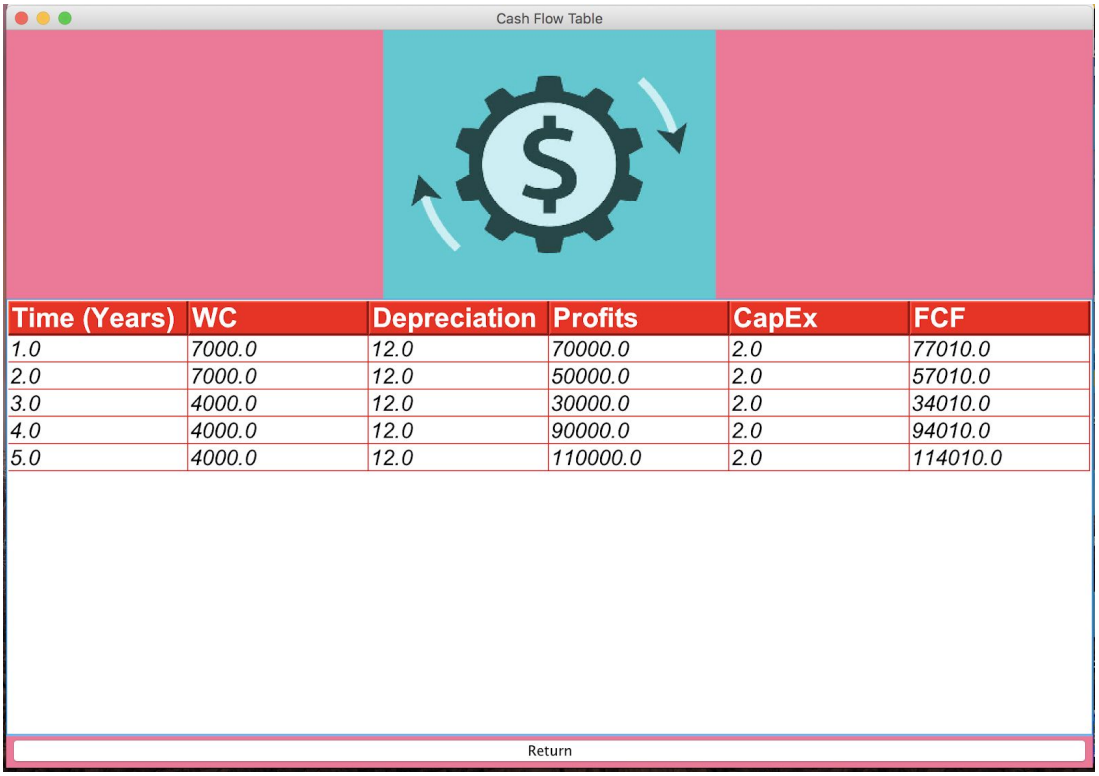

Figure 7: Cash Flow Table that shows the cash flow based on inputed cash flow information

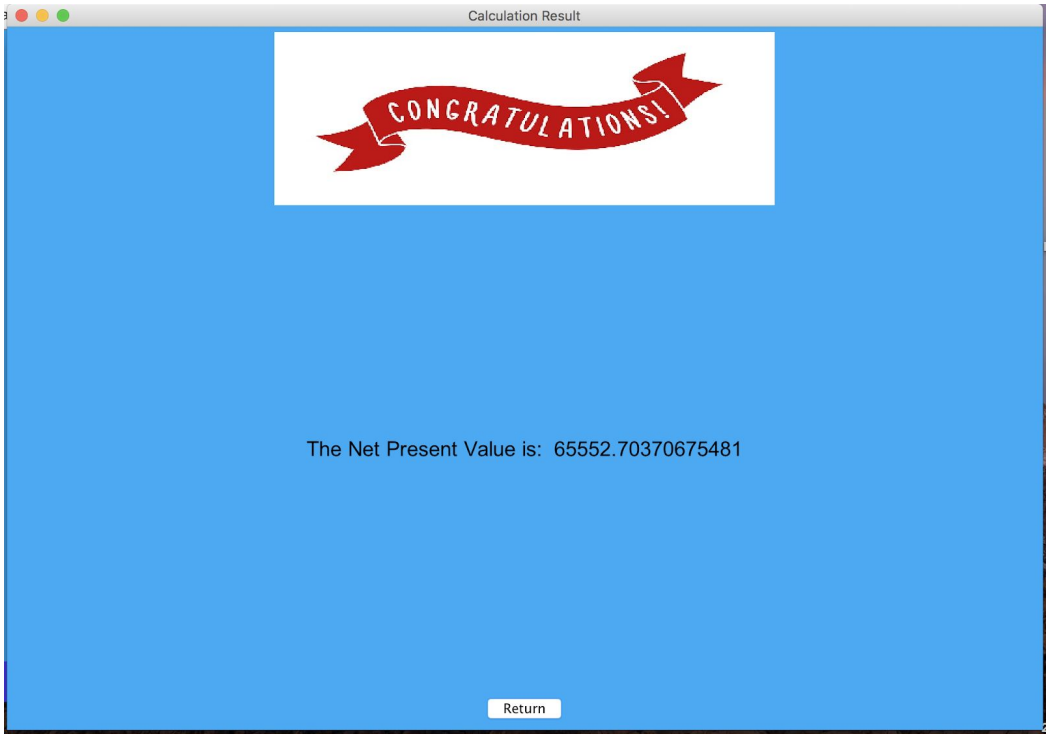

Figure 8: General output that shows general calculated information based on user input

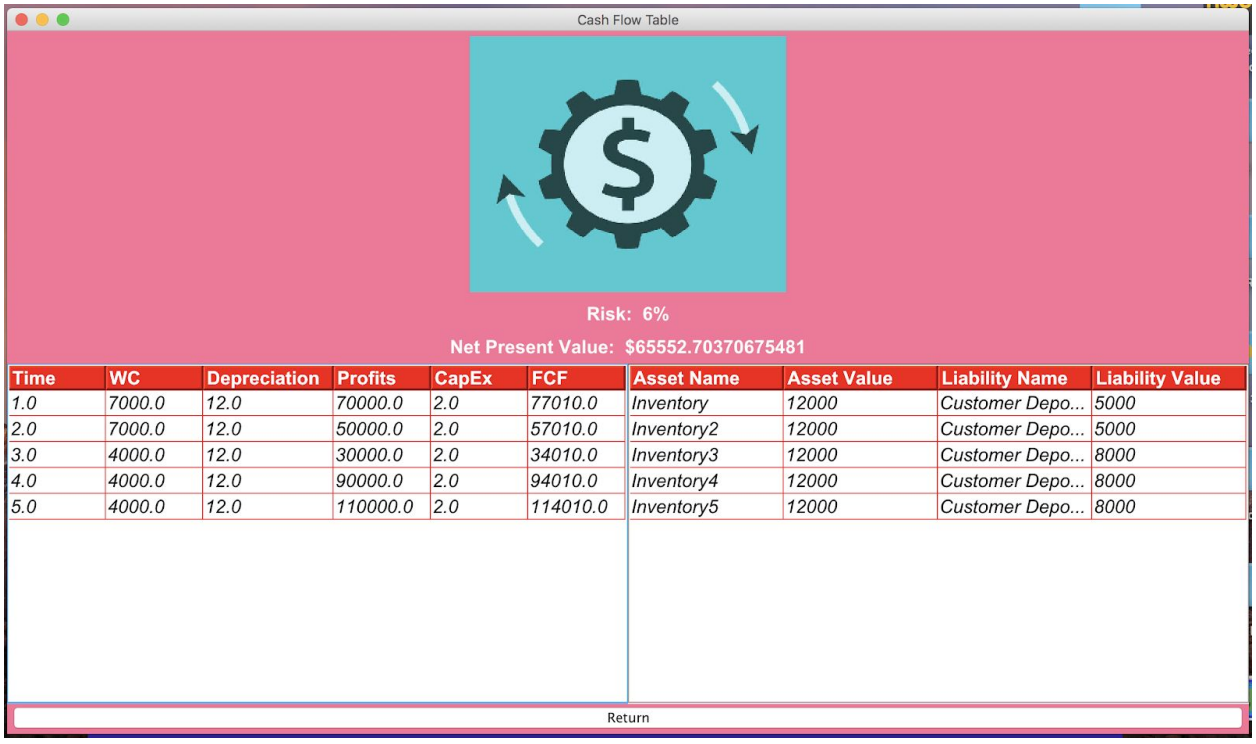

Figure 9: Business report that shows a balance sheet, cash flow data, and other financial information based on specified business

```
//Formats Header
header = cashFlowTable.getTableHeader();
header.setBackground(new Color(230,52,39));
header.setForeground(new Color(255,255,255));
header.setFont(new Font("Arial", Font.BOLD, 25));

//Formats the row
cashFlowTable.setRowHeight(25);

//Formats Column
column = cashFlowTable.getColumnModel().getColumn(0);
column.setPreferredWidth(500);
column = cashFlowTable.getColumnModel().getColumn(1);
column.setPreferredWidth(500);
column = cashFlowTable.getColumnModel().getColumn(2);
column.setPreferredWidth(500);
column = cashFlowTable.getColumnModel().getColumn(3);
column.setPreferredWidth(500);
column = cashFlowTable.getColumnModel().getColumn(4);
column.setPreferredWidth(500);
column = cashFlowTable.getColumnModel().getColumn(5);
column.setPreferredWidth(500);

//Constructs ScrollPane
cashFlowScroll = new JScrollPane();
cashFlowScroll.getViewport().add(cashFlowTable);
cashFlowTable.setFillsViewportHeight(true);
```

Figure 9.1: Example for the creation of a scrollable cash flow table in the Cash Flow Table class

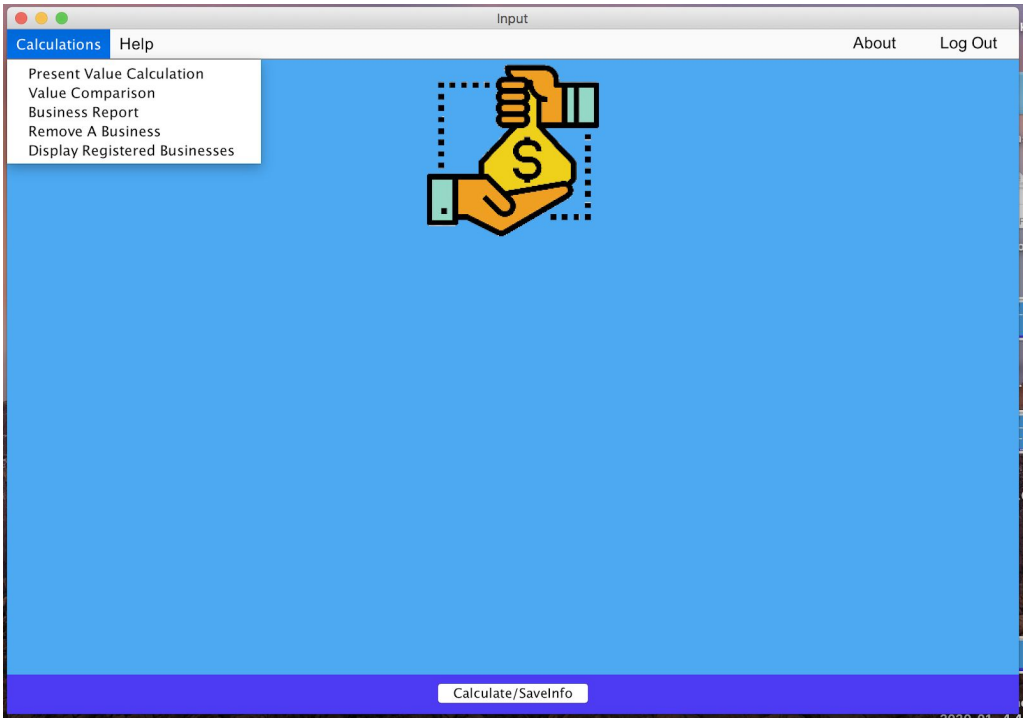

Figure 10: Menu for input class that allows for a user to choose different calculations and reports

```
//Constructs the Menu
mainBar = new JMenuBar();

calculationsMenu = new JMenu("Calculations");
investmentComparisonItem = new JMenuItem("Value Comparison");
investmentComparisonItem.addActionListener(this);
businessReportItem = new JMenuItem("Business Report");
businessReportItem.addActionListener(this);
presentValueItem = new JMenuItem("Present Value Calculation");
presentValueItem.addActionListener(this);
removeBusinessItem = new JMenuItem("Remove A Business");
removeBusinessItem.addActionListener(this);
displayBusinessesItem = new JMenuItem("Display Registered Businesses");
displayBusinessesItem.addActionListener(this);

calculationsMenu.add(presentValueItem);
calculationsMenu.add(investmentComparisonItem);
calculationsMenu.add(businessReportItem);
calculationsMenu.add(removeBusinessItem);
calculationsMenu.add(displayBusinessesItem);

helpMenu = new JMenu("Help");
helpMenu.setFont(new Font("Arial", Font.PLAIN, 16));
generalHelpItem = new JMenuItem("General Help");
generalHelpItem.addActionListener(this);
calculationHelpItem = new JMenuItem("Calculation Help");
calculationHelpItem.addActionListener(this);

helpMenu.add(generalHelpItem);
helpMenu.add(calculationHelpItem);
```

Figure 10.1: Example for creation of calculation and help menu in the input frame

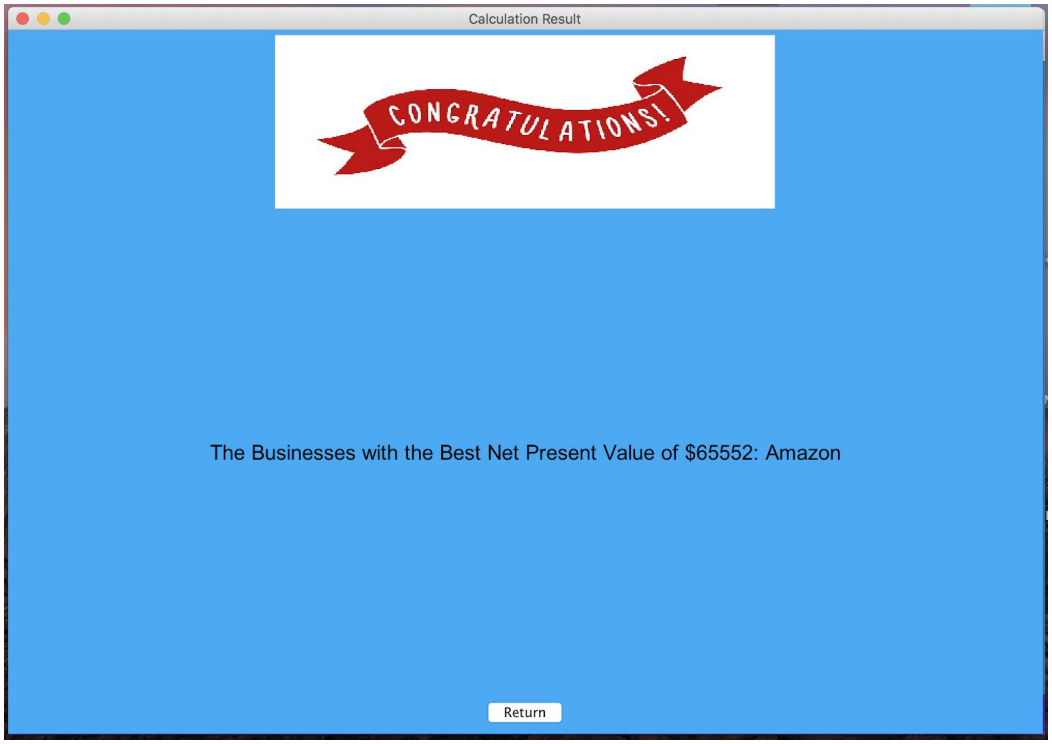

Figure 11: General output that shows the business(es) with the highest net present value

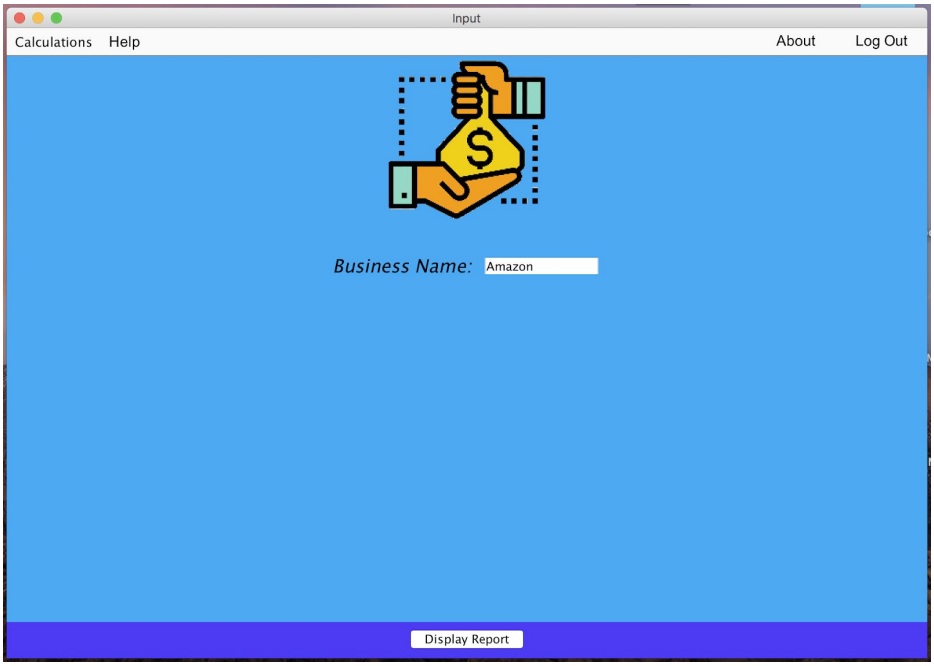

Figure 12: Input frame that allows for a business report to be displayed for a specified business

```
else if (command.equals("Display Report"))
{
    if (registerPanel.isDisplayable())
    {
        businessName = registerField.getText();

        boolean businessExists;

        //Checks if business is registered/exists
        try
        {
            businessExists = checkBusinessName();
        }
        catch (IndexOutOfBoundsException ibe)
        {
            businessExists = false;
        }

        if (businessExists == true)
        {
            cashFlowData.clear();
            balanceSheetData.clear();
            time = 1;

            //Obtain balance and cash flow data
            balanceSheetData = getBalanceData();
            cashFlowData = getFlowData();

            //Displays business report
            businessReportObj = new BusinessReport(cashFlowData, balanceSheetData,
                    Integer.toString(getRisk(businessName)),
                    Double.toString(getPresentValue(businessName)));
        }
        else
        {
            errorObj = new Error("Sorry, that business name does not exist");
        }

    }
```

Figure 12.1: Represents the action performed for displaying a report in the input class

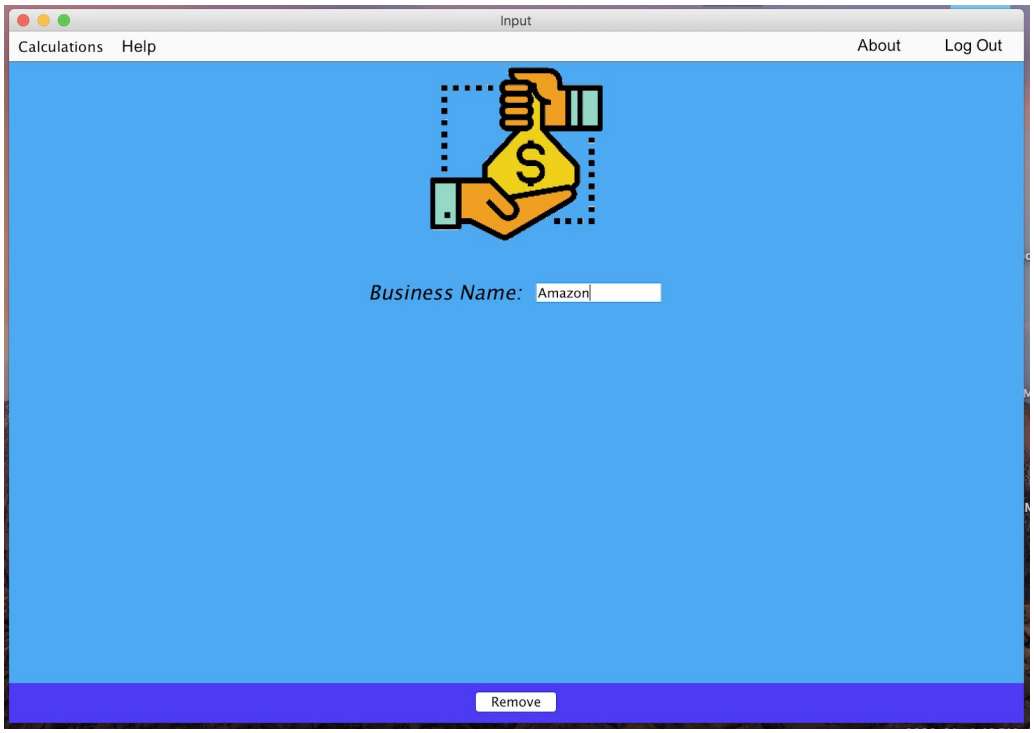

Figure 13: Input frame that allows for a specified business to be removed

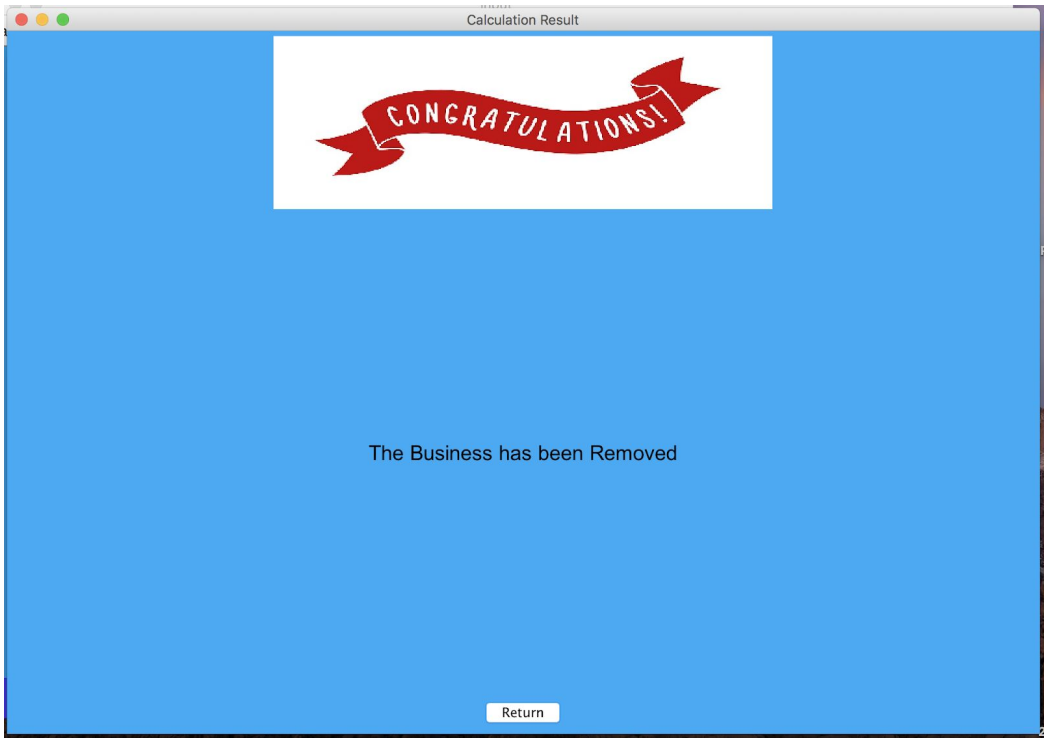

Figure 14: General Output that shows the business has been removed

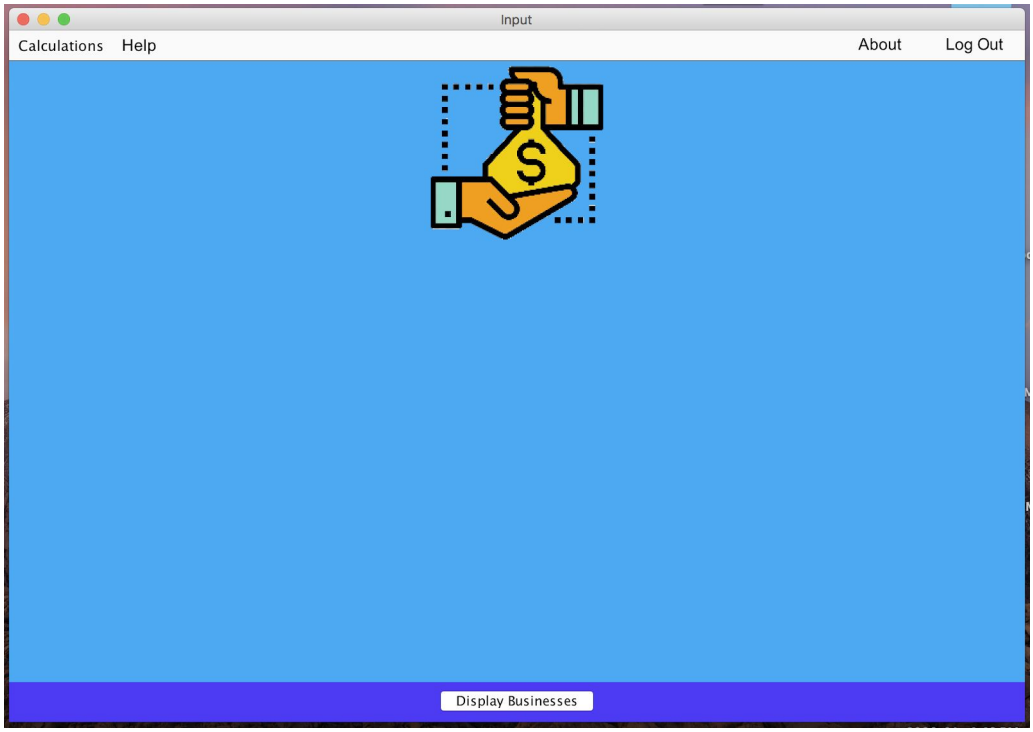

Figure 15: Input frame allows for a user to display the businesses that are registered on their account

```
if (command.equals("Log Out"))
{
    //Opens Log in frame
    LogIn logInObj = new LogIn();
    this.dispose();

    registerField.setText("");
    revenueField.setText("");
    costsField.setText("");
    employeeField.setText("");
    businessYearsField.setText("");
    countriesAmountField.setText("");
    competitorsAmountField.setText("");
    actualEquityField.setText("");
    actualDebtField.setText("");
    costDebtField.setText("");
    assetNameField.setText("");
    assetValueField.setText("");
    liabilityNameField.setText("");
    liabilityValueField.setText("");
    depreciationField.setText("");
    capitalExpenditureField.setText("");
}
else if (command.equals("About"))
{
    //Opens About frame
    About aboutObj = new About();
}
```

Figure 15.1: Action performed for the input frame and how it navigates to the log in and about frame

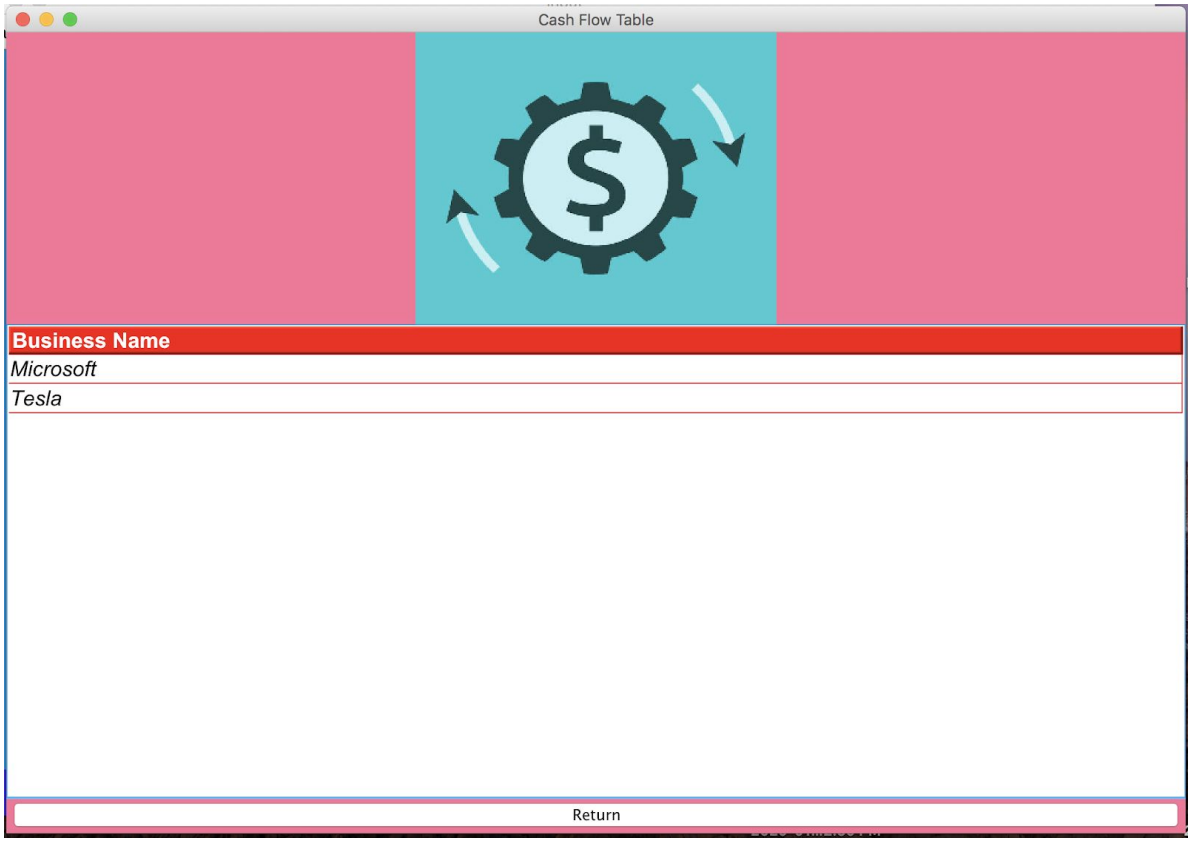

Figure 16: Business report that shows the businesses that are registered on the specific user account in alphabetical order

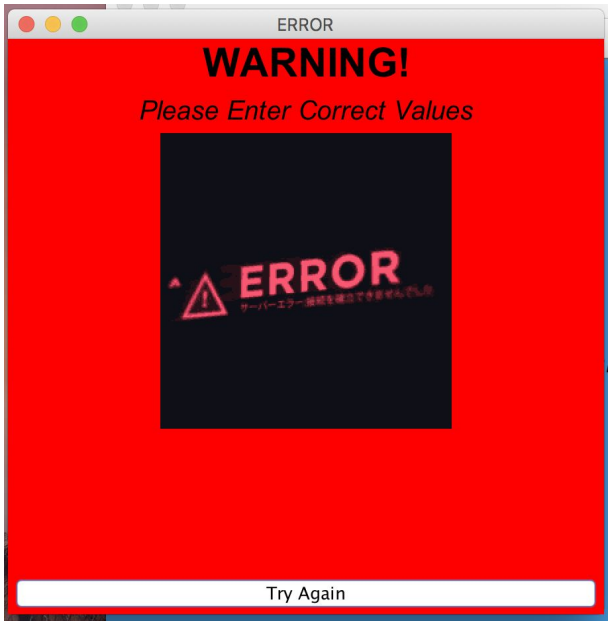

Figure 17: Warning frame that shows a specific error with computation or user input

```
try
{
    //removes data
    removeValueData(registerField.getText());
}
catch(IndexOutOfBoundsException ibe)
{
    errorObj = new Error("The Business Does Not Exist");
}
```

Figure 17.1: Example of exception handling in the input class

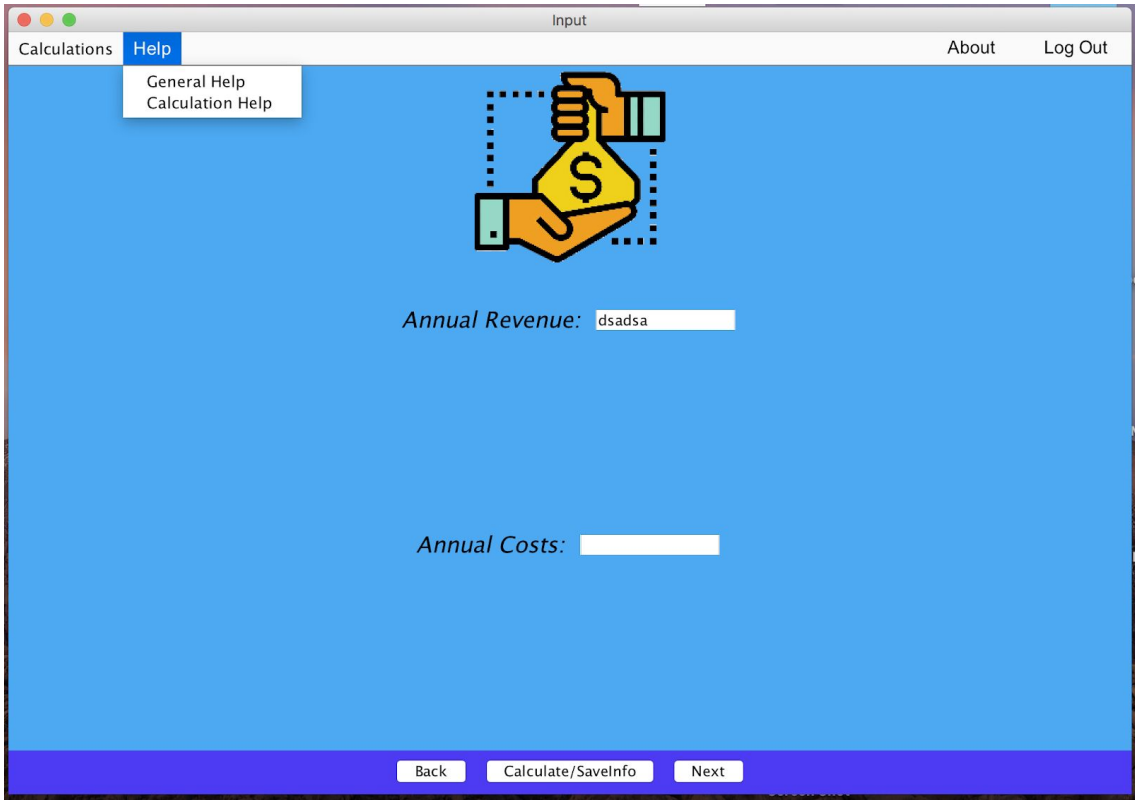

Figure 18: Menu for input class that allows for a user to choose different help frames

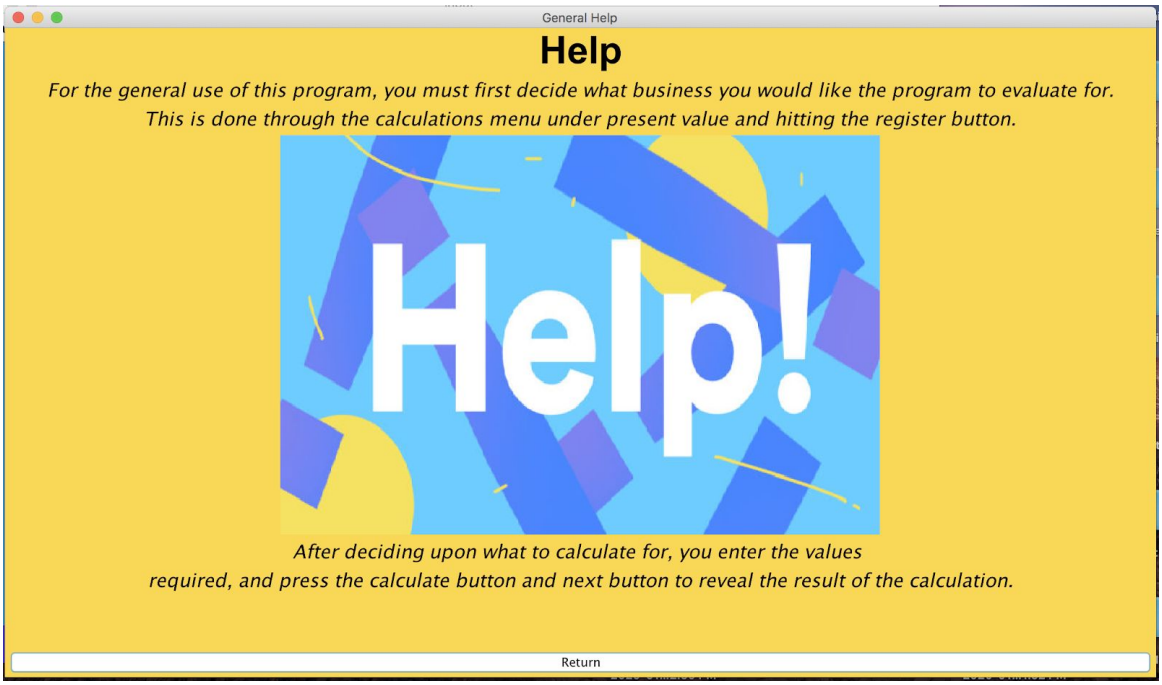

Figure 19: General Help frame that provides general help information for using the application

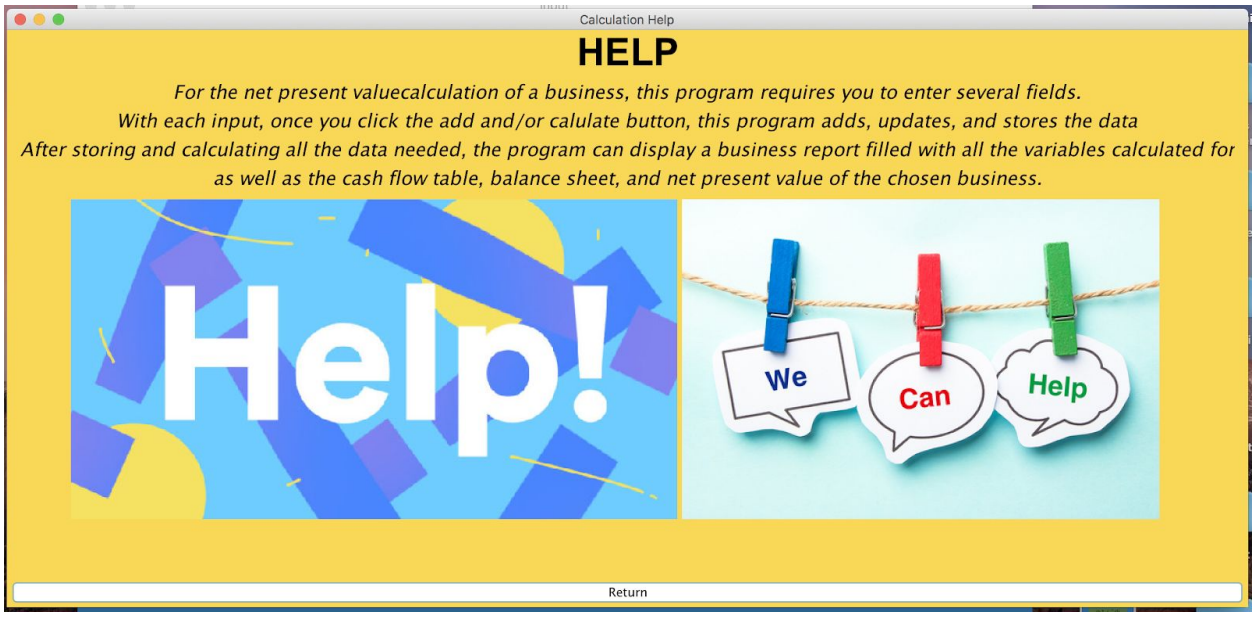

Figure 20: Calculation Help frame that provides specific help information about using the application for calculations

Figure 21: About frame that provides a general summary of what this program does

```
String assetsQuery = "INSERT INTO Assets VALUES (?,?,?,?)";

//Uses query to add data to database
try
{
    PreparedStatement ps = myDbConn.prepareStatement(assetsQuery);
    ps.setString(1, email);
    ps.setString(2, businessName);
    ps.setString(3, assetName);
    ps.setDouble(4, assetValue);
    ps.executeUpdate();
}
catch (SQLException se)
{
      System.out.println("Error inserting data");
    se.printStackTrace(System.err);
}
```

Figure 22: Example of inserting data into the database within the input class

```
/*If the data is the same for the business being removed compared to
the ones not being removed, it does not delete the data*/
if (duplicateRiskData == false)
{
    String riskQuery = "DELETE FROM RiskFactor WHERE EmployeeAmount = ? AND "
            + "BusinessYears = ? AND CountriesAmount = ? AND CompetitorsAmount = ?";

    //Uses query to add data to database
    try
    {
        PreparedStatement ps = myDbConn.prepareStatement(riskQuery);
        ps.setInt(1, employeeAmount);
        ps.setInt(2, businessYears);
        ps.setInt(3, countriesAmount);
        ps.setInt(4, competitorsAmount);
        ps.executeUpdate();
    }
    catch (SQLException se)
    {
          System.out.println("Error inserting data");
        se.printStackTrace(System.err);
    }
}
```

Figure 23: Example of deleting data in the database within the input class

Reasoning for Structure:

The overall use of this structure provides the user with a streamlined way to navigate between frames within the application as evident by the use of the menu bar and several JButtons. In addition to that, it provides business information in a digestible manner through the use of JTables.

Evidence of Algorithmic Thinking

Compare Net Present Values Algorithm

This algorithm obtains all the non time data from different tables in the database. As it obtains the data, it checks whether certain data variables in one table match to that of another table. If they do not, the data is not saved, but if they do, it saves the data into a certain variable. Once all the non time data variables are obtained, the algorithm then obtains the time data from tables in the database. It then checks through the time data variables by going through the maximum number of years of financial information that the user inputted. As it goes through the time data, it checks whether certain data variables in one time table match to that of the saved non time variables as well as other tables depending on the specific table information. If the data matches, the data is saved into a certain variable. That saved data is then used to finally check with the net present value data. If it matches, it adds the data to an array list that holds the current business name as well as the net present value. Once done, it continues doing this until it goes through the maximum number of years/time that the user inputted for the business. After finishing the loop it goes through the present value data arraylist and compares the data to obtain the highest present value. Once obtained, it then compares that highest present value with all the data in the present value arraylist. If the present value data are equal it adds that business with the present value into another arraylist. Once it has gone through all the data, that arraylist is then returned.

```
//Compares net present values of all the businesses
private ArrayList<String> compareValues()
{
    Connection myDbConn = null;

    int employeeAmount = 0;
    int businessYears = 0;
    int countriesAmount = 0;
    int competitorsAmount = 0;

    double actualDebt = 0;
    double actualEquity = 0;

    double equityWeight = 0;
    double debtWeight = 0;

    double equityCost = 0;
    double debtCost = 0;

    int riskFactor = 0;

    double depreciation = 0;
    double capitalExpenditure = 0;
    double revenue = 0;
    double costs = 0;
    double profits = 0;
    double workingCapital = 0;
    double freeCashFlow = 0;
```

Figure 1: Declares variables for the Compare Values method

```
String assetName = "";
String liabilityName = "";

double wacc = 0;

double bestPresentValue = Double.MIN_VALUE;

ArrayList<String> presentValueBusinesses = new ArrayList<>();
ArrayList<String> registeredBusinesses = new ArrayList<>();
ArrayList<String> bestBusinessNames = new ArrayList<>();

String currentBusinessName = "";

try
{
    registeredBusinesses = getBusinessNames();
}
catch(IndexOutOfBoundsException ibe)
{
    return bestBusinessNames;
}

//Creates object of database and connects to database
DatabaseAccess objDb = new DatabaseAccess();
objDb.setDbName("BusinessFinancials");
objDb.setDbConn();
```

Figure 1.1: Obtains business names using another method and declares more variables

```
//Goes through every businesses information
for (int i=0; i<registeredBusinesses.size(); i++)
{
    currentBusinessName = registeredBusinesses.get(i);

    myDbConn = objDb.getDbConn();
    String[] nonTimeHeader =
    {
        "Email", "BusinessName",
        "ActualEquity", "ActualDebt", "DebtCost", "EmployeeAmount",
        "BusinessYears", "CountriesAmount", "CompetitorsAmount"
    };
    Object[][] nonTimeData;

    nonTimeData = objDb.to2dArray(objDb.getData("NonTimeFinancials", nonTimeHeader));
```

Figure 1.2: Creates main loop that goes through the number of registered businesses and then obtains time data

```
/*Goes through specific data and checks if data matches
    of business*/
for (int j = 0; j < nonTimeData.length; j++)
{
    if (email.equalsIgnoreCase(nonTimeData[j][0].toString()) &&
            currentBusinessName.equalsIgnoreCase(nonTimeData[j][1].toString()))
    {
        actualEquity = Double.parseDouble(nonTimeData[j][2].toString());
        actualDebt = Double.parseDouble(nonTimeData[j][3].toString());
        debtCost = Double.parseDouble(nonTimeData[j][4].toString());

        employeeAmount = Integer.parseInt(nonTimeData[j][5].toString());
        businessYears = Integer.parseInt(nonTimeData[j][6].toString());
        countriesAmount = Integer.parseInt(nonTimeData[j][7].toString());
        competitorsAmount = Integer.parseInt(nonTimeData[j][8].toString());
    }
}

String[] riskHeader =
{
    "EmployeeAmount", "BusinessYears",
    "CountriesAmount", "CompetitorsAmount", "RiskFactor"
};
Object[][] riskData;

riskData = objDb.to2dArray(objDb.getData("RiskFactor", riskHeader));
```

Figure 1.3: Creates a loop to go through the time data and obtains the financial values specified. Then, it obtains risk data

```
/*Goes through specific data and checks if data matches
    of business*/
for (int j = 0; j < riskData.length; j++)
{
    if (employeeAmount == Integer.parseInt(riskData[j][0].toString())
            && businessYears == Integer.parseInt(riskData[j][1].toString())
            && countriesAmount == Integer.parseInt(riskData[j][2].toString())
            && competitorsAmount == Integer.parseInt(riskData[j][3].toString()))
    {
        riskFactor = (int) Double.parseDouble(riskData[j][4].toString());
    }
}

String[] weightVariablesHeader =
{
    "ActualDebt", "ActualEquity",
    "DebtWeight", "EquityWeight"
};
Object[][] weightVariablesData;

weightVariablesData = objDb.to2dArray(objDb.getData("WeightVariables", weightVariablesHeader));
```

Figure 1.4: Creates a loop to go through the risk data and obtains the financial values specified. Then, it obtains weight variables data

```
/*Goes through specific data and checks if data matches
    of business*/
for (int j = 0; j < weightVariablesData.length; j++)
{
    if (actualDebt == Double.parseDouble(weightVariablesData[j][0].toString())
            && actualEquity == Double.parseDouble(weightVariablesData[j][1].toString()))
    {
        equityWeight = Double.parseDouble(weightVariablesData[j][3].toString());
        debtWeight = Double.parseDouble(weightVariablesData[j][2].toString());
    }
}

String[] equityCostHeader =
{
    "DebtWeight", "RiskFactor", "EquityCost"
};
Object[][] equityCostData;

equityCostData = objDb.to2dArray(objDb.getData("EquityCost", equityCostHeader));
```

Figure 1.5: Creates a loop to go through the weight variables data and obtains the financial values specified. Then, it obtains equity cost data

```
/*Goes through specific data and checks if data matches
    of business*/
for (int j = 0; j < equityCostData.length; j++)
{
    if (debtWeight == Double.parseDouble(equityCostData[j][0].toString())
            && riskFactor == Double.parseDouble(equityCostData[j][1].toString()))
    {
        equityCost = Double.parseDouble(equityCostData[j][2].toString());
    }
}

String[] waccHeader =
{
    "DebtCost", "EquityCost", "DebtWeight", "EquityWeight", "Wacc"
};
Object[][] waccData;

waccData = objDb.to2dArray(objDb.getData("Wacc", waccHeader));
```

Figure 1.6: Creates a loop to go through the equity cost data and obtains the financial values specified. Then, it obtains wacc (weighted average cost of capital) data

```
/*Goes through specific data and checks if data matches
    of business*/
for (int j = 0; j < waccData.length; j++)
{
    if (debtCost == Double.parseDouble(waccData[j][0].toString())
            && equityCost == Double.parseDouble(waccData[j][1].toString())
            && debtWeight == Double.parseDouble(waccData[j][2].toString())
            && equityWeight == Double.parseDouble(waccData[j][3].toString()))
    {
        wacc = Double.parseDouble(waccData[j][4].toString());
    }
}

String[] timeHeader =
{
    "Email", "BusinessName", "Time", "CapitalExpenditure",
    "Assets", "Liabilities", "Depreciation", "Revenue", "Costs"
};
Object[][] timeData;

timeData = objDb.to2dArray(objDb.getData("TimeFinancials", timeHeader));

String[] profitHeader =
{
    "Time", "Revenue", "Costs", "Profits"
};
Object[][] profitArrayData;

profitArrayData = objDb.to2dArray(objDb.getData("Profits", profitHeader));
```

Figure 1.7: Creates a loop to go through the wacc (weighted average cost of capital) data and obtains the financial values specified. Then, it obtains time and profit data

```
String[] cashFlowHeader =
{
    "Time", "WorkingCapital", "Depreciation",
    "Profits", "CapitalExpenditure", "FreeCashFlow"
};
Object[][] cashFlowData;

cashFlowData = objDb.to2dArray(objDb.getData("FreeCashFlow", cashFlowHeader));

String[] workingCapitalHeader =
{
    "Time", "Assets", "Liabilities", "WorkingCapital"
};
Object[][] workingCapitalData;

workingCapitalData = objDb.to2dArray(objDb.getData("WorkingCapital", workingCapitalHeader));

String[] presentValueHeader =
{
    "Time", "FreeCashFlow", "Wacc", "NetPresentValue"
};
Object[][] presentValueData;

presentValueData = objDb.to2dArray(objDb.getData("NetPresentValue", presentValueHeader));
```

Figure 1.8: Obtains cash flow, working capital, and present value data

```
/*Goes through time specific data*/
for (int j = 1; j <= maxTime; j++)
{
    /*Goes through specific data and checks if data matches
    of business*/
    for (int k = 0; k < timeData.length; k++)
    {
        if (email.equalsIgnoreCase(timeData[k][0].toString())
                && currentBusinessName.equalsIgnoreCase(timeData[k][1].toString())
                && j == Integer.parseInt(timeData[k][2].toString()))
        {
            capitalExpenditure = Double.parseDouble(timeData[k][3].toString());
            assetName = timeData[k][4].toString();
            liabilityName = timeData[k][5].toString();
            depreciation = Double.parseDouble(timeData[k][6].toString());
            revenue = Double.parseDouble(timeData[k][7].toString());
            costs = Double.parseDouble(timeData[k][8].toString());
        }
    }
```

Figure 1.9: Goes through a loop that only ends until it reaches the maximum amount of time. Then goes through the time data and obtains the financial values specified

```
/*Goes through specific data and checks if data matches
of business*/
for (int k = 0; k < profitArrayData.length; k++)
{
    if (j == Integer.parseInt(profitArrayData[k][0].toString())
            && revenue == Double.parseDouble(profitArrayData[k][1].toString())
            && costs == Double.parseDouble(profitArrayData[k][2].toString()))
    {
        profits = Double.parseDouble(profitArrayData[k][3].toString());
    }
}

/*Goes through specific data and checks if data matches
of business*/
for (int k = 0; k < workingCapitalData.length; k++)
{
    if (j == Integer.parseInt(workingCapitalData[k][0].toString())
            && assetName.equalsIgnoreCase(workingCapitalData[k][1].toString())
            && liabilityName.equalsIgnoreCase(workingCapitalData[k][2].toString()))
    {
        workingCapital = Double.parseDouble(workingCapitalData[k][3].toString());
    }
}
```

Figure 1.10: Goes through the profit data and obtains the financial values specified. Then goes through the working capital data and obtains the financial values specified

```
            /*Goes through specific data and checks if data matches
            of business*/
            for (int k = 0; k < cashFlowData.length; k++)
            {
                if (j == Integer.parseInt(cashFlowData[k][0].toString())
                        && workingCapital == Double.parseDouble(cashFlowData[k][1].toString())
                        && depreciation == Double.parseDouble(cashFlowData[k][2].toString())
                        && profits == Double.parseDouble(cashFlowData[k][3].toString())
                        && capitalExpenditure == Double.parseDouble(cashFlowData[k][4].toString()))
                {
                    freeCashFlow = Double.parseDouble(cashFlowData[k][5].toString());
                }
            }

            /*Goes through specific data and checks if data matches
            of business*/
            for (int k = 0; k < presentValueData.length; k++)
            {
                if (j == Integer.parseInt(presentValueData[k][0].toString())
                        && freeCashFlow == Double.parseDouble(presentValueData[k][1].toString())
                        && wacc == Double.parseDouble(presentValueData[k][2].toString()))
                {
                    presentValueBusinesses.add(presentValueData[k][3].toString());
                    presentValueBusinesses.add(currentBusinessName);
                }
            }
        }
    }
```

Figure 1.11: Goes through the cash flow data and obtains the financial values specified. Then goes through the present value data and obtains the financial values specified

```
/*Goes through present value data and obtains the greatest net present value*/
for (int i = 0; i < presentValueBusinesses.size(); i+=2)
{
    if (Double.parseDouble(presentValueBusinesses.get(i)) > bestPresentValue)
    {
        bestPresentValue = Double.parseDouble(presentValueBusinesses.get(i));
    }
}

/*Goes through present value data and obtains the
businesses with the best present value*/
for (int i = 0; i < presentValueBusinesses.size(); i+=2)
{
    if (Double.parseDouble(presentValueBusinesses.get(i)) == bestPresentValue)
    {
        bestBusinessNames.add(presentValueBusinesses.get(i));
        bestBusinessNames.add(presentValueBusinesses.get(i+1));
    }

}

return bestBusinessNames;
```

Figure 1.12: Compares net present values between different businesses and then finds the businesses with the greatest net present value

Security Questions Algorithm

This algorithm obtains the security answer data from the database. It then goes through each row of that data that contains the given email, and sets the question id equal to the one within the data. After obtaining all three question ids, it then obtains the security question data from the database and checks each row of data. If one of the question ids match with the question id within the security question data, it sets the login frame information with the given security question.

```
//Finds security questions of specified account
private void findSecurityQuestions(String email)
{
    Connection myDbConn = null;

    int counter = 0;

    int questionId1 = 0;
    int questionId2 = 0;
    int questionId3 = 0;

    //Creates object of database and connects to database
    DatabaseAccess objDb = new DatabaseAccess();
    objDb.setDbName("BusinessFinancials");
    objDb.setDbConn();
    myDbConn = objDb.getDbConn();
    String[] securityColumnName = {"Email", "SecurityId", "SecurityAnswer"};
    Object[][] data;

    data = objDb.to2dArray(objDb.getData("SecurityAnswers", securityColumnName));
```

Figure 2: Declares variables for the find security questions method

```
//Finds the question ids for the email inputed
for (int i=0; i<data.length; i++)
{
    if (email.equalsIgnoreCase(data[i][0].toString()))
    {
        if (counter == 0)
        {
            questionId1 = Integer.parseInt(data[i][1].toString());
        }
        else if (counter == 1)
        {
            questionId2 = Integer.parseInt(data[i][1].toString());
        }
        else if (counter == 2)
        {
            questionId3 = Integer.parseInt(data[i][1].toString());
        }

        counter++;
    }
}

String[] questionColumnName = {"SecurityId", "SecurityQuestion"};

data = objDb.to2dArray(objDb.getData("SecurityQuestions", questionColumnName));
```

Figure 2.1: Creates a loop that goes through the security answers data and obtains certain question ids based on the information

```
    //Checks if the email the user inputed in is valid
    for (int i=0; i<data.length; i++)
    {
        if (questionId1 == Integer.parseInt(data[i][0].toString()))
        {
            securityQuestion1.setText(data[i][1].toString() + " ");
        }
        if (questionId2 == Integer.parseInt(data[i][0].toString()))
        {
            securityQuestion2.setText(data[i][1].toString() + " ");
        }
        if (questionId3 == Integer.parseInt(data[i][0].toString()))
        {
            securityQuestion3.setText(data[i][1].toString() + " ");
        }
    }

    //Closes connection
    objDb.closeDbConn();
}
```

Figure 2.2: Creates a loop that goes through the security questions data and sets the security question based upon the security question id within the data

Cash Flow Algorithm

This algorithm obtains the time data and cash flow data from the database. It goes through each row of the time data that contains the given email and business name, and then it compares the given row of data with the cash flow data. If the data matches, it adds all the components and information to a separate array list called converted data and then returns it.

```
//Obtains cash flow data
private ArrayList<Double> getFlowData()
{
    int time;
    double workingCapital;
    double depreciation;
    double profits;
    double capitalExpenditure;
    int balanceSheetCounter = 0;

    BusinessFinancials businessObj = new BusinessFinancials();

    Connection myDbConn = null;

    //Creates object of database and connects to database
    DatabaseAccess objDb = new DatabaseAccess();
    objDb.setDbName("BusinessFinancials");
    objDb.setDbConn();
    myDbConn = objDb.getDbConn();

    ArrayList<ArrayList<String>> data;
    Object[][] timeArrayData, profitArrayData;
```

Figure 3: Declares variables for the get flow data method

```
String tableName = "TimeFinancials";
String[] timeFinancialsHeader =
{
    "Email", "BusinessName",
    "Time", "CapitalExpenditure", "Assets", "Liabilities",
    "Depreciation", "Revenue", "Costs"
};

data = objDb.getData(tableName, timeFinancialsHeader);

timeArrayData = objDb.to2dArray(data);

tableName = "FreeCashFlow";
String[] tableHeaders =
{
    "Time", "WorkingCapital", "Depreciation",
    "Profits", "CapitalExpenditure", "FreeCashFlow"
};

ArrayList<Double> convertedData = new ArrayList<>(12);

data = objDb.getData(tableName, tableHeaders);

int columnCount = data.get(0).size();
```

Figure 3.1: Obtains the time financials and free cash flow data and sets variable column count equal to the number of columns within the free cash flow data

```
//Goes through the data
for (int i = 0; i < timeArrayData.length; i++)
{
    //Checks for the data that matches with specified email and business name
    if (email.equalsIgnoreCase(timeArrayData[i][0].toString()))
    {
        if (businessName.equalsIgnoreCase(timeArrayData[i][1].toString()))
        {
            time = Integer.parseInt(timeArrayData[i][2].toString());
            workingCapital = businessObj.calculateWorkingCapital(
                    Double.parseDouble(balanceSheetData.get(balanceSheetCounter * 4 + 1).toString()),
                    Double.parseDouble(balanceSheetData.get(balanceSheetCounter * 4 + 3).toString()));
            depreciation = Double.parseDouble(timeArrayData[i][6].toString());
            profits = businessObj.profitCalculation(
                    Double.parseDouble(timeArrayData[i][7].toString()),
                    Double.parseDouble(timeArrayData[i][8].toString()));
            capitalExpenditure = Double.parseDouble(timeArrayData[i][3].toString());

            //Obtains cash flow data
            for (int j = 0; j < data.size(); j++)
            {
                ArrayList<String> row = data.get(j);

                    for (int k = 0; k < columnCount; k = k + 6)
                    {
                        //Converts the data into an ArrayList<Double>
                        if (time == Integer.valueOf(row.get(k))
                                && workingCapital == Double.parseDouble(row.get(k + 1))
                                && depreciation == Double.parseDouble(row.get(k + 2))
                                && profits == Double.parseDouble(row.get(k + 3))
                                && capitalExpenditure == Double.parseDouble(row.get(k + 4)))
                        {
                            convertedData.add(Double.parseDouble(row.get(k)));
                            convertedData.add(Double.parseDouble(row.get(k + 1)));
                            convertedData.add(Double.parseDouble(row.get(k + 2)));
                            convertedData.add(Double.parseDouble(row.get(k + 3)));
                            convertedData.add(Double.parseDouble(row.get(k + 4)));
                            convertedData.add(Double.parseDouble(row.get(k + 5)));
                        }

                    }
                }

                balanceSheetCounter++;
            }
        }
    }

    return convertedData;
}
```

Figure 3.2: Loops through time data and obtains the free cash flow data that matches with the given time data for a specified email and business name

Techniques Used

Encapsulation

```
//Sets info
public void setNetPresentValue(double pNetPresentValue)
{
    this.netPresentValue = pNetPresentValue;
}
//Sets info
public void setWorkingCapital(double pWorkingCapital)
{
    this.workingCapital = pWorkingCapital;
}
//gets info
public double getWacc()
{
    return this.wacc;
}
//gets info
public double getWeightedEquity()
{
    return this.weightedEquity;
}
//gets info
public double getWeightedDebt()
{
    return this.weightedDebt;
}
```

```
try
{
    businessObj.setNetPresentValue(0);
}
catch (NullPointerException npe)
{

}
```

```
//Adds and calculates for net present values
presentValueData.add(businessObj.presentValueCalculation(time, cashFlowData.get(5), businessObj.getWacc()));
presentValueData.add(businessObj.presentValueCalculation(time + 1, cashFlowData.get(11), businessObj.getWacc()));
presentValueData.add(businessObj.presentValueCalculation(time + 2, cashFlowData.get(17), businessObj.getWacc()));
presentValueData.add(businessObj.presentValueCalculation(time + 3, cashFlowData.get(23), businessObj.getWacc()));
presentValueData.add(businessObj.presentValueCalculation(time + 4, cashFlowData.get(29), businessObj.getWacc()));

generalOutputObj = new GeneralOutput("Net Present Value", businessObj.getNetPresentValue());
```

Figure 1: Represents examples of encapsulation and the use of it within the business financials and input class

Classes, such as the business financials class, implement encapsulation in order to allow for other classes, such as the input class, to access data through public mutators and accessors. This is done in order to create flexibility and maintainability. Flexibility is created as the encapsulated code can be changed and updated easily without having to change it in other classes. Including

this, it is maintainable as the encapsulated class can be changed without having to change other classes utilizing it.

Polymorphism

```
public BusinessFinancials()
{
   wacc = 0;
   weightedEquity = 0;
   weightedDebt = 0;
   equityCost = 0;
   profit = 0;
   freeCashFlow = 0;
   netPresentValue = 0;
   workingCapital = 0;
}
public BusinessFinancials(double pWacc, double pWeightedEquity, double pWeightedDebt, double pEquityCost,
      double pProfit, double pFreeCashFlow, double pNetPresentValue, double pWorkingCapital)
{
   wacc = pWacc;
   weightedEquity = pWeightedEquity;
   weightedDebt = pWeightedDebt;
   equityCost = pEquityCost;
   profit = pProfit;
   freeCashFlow = pFreeCashFlow;
   netPresentValue = pNetPresentValue;
   workingCapital = pWorkingCapital;
}
```

Figure 2: Provides an example of overloading within the business financials class

As evident by the figure above, classes, such as the business financials and logIn manager, implement overloaded methods in order to allow for different classes to refer to them with one common name when utilizing similar, yet slightly different implementations. For example, the class above has two constructors in which an object can be created with default values or with given values. The program also makes use of overriding. This is seen in a majority of the GUI classes as they override a method called action performed in order to execute that specific implementation.

Inheritance

```
class LogIn extends JFrame

super("LogIn");
```

Figure 3: Indicates instances in which the log in frame had utilized inheritance

All the gui classes in this project, such as the Log in frame, use inheritance in order to gain the attributes and methods of a JFrame. This is done in order to allow for code reusability and efficiency as JFrame is apart of Java API and is made by experts in the computer science field. In specific, the log In frame utilizes border layouts and the attributes needed in order to create a frame.

Accessing Database

```
Connection myDbConn = null;

//Creates object of database and connects to database
DatabaseAccess objDb = new DatabaseAccess();
objDb.setDbName("BusinessFinancials");
objDb.setDbConn();
myDbConn = objDb.getDbConn();
String tableName = "Assets";
String[] tableHeaders =
{
    "Email", "BusinessName", "Assets", "AssetValue"
};

ArrayList<ArrayList<String>> data;

data = objDb.getData(tableName, tableHeaders);

ArrayList<Object> assetData = new ArrayList<>(12);

int columnCount = data.get(0).size();

//Obtains asset data
for (int i = 0; i < data.size(); i++)
{
    ArrayList<String> row = data.get(i);

    for (int j = 0; j < columnCount; j = j + 4)
    {
        if (email.equalsIgnoreCase(row.get(j)) &&
                businessName.equalsIgnoreCase(row.get(j + 1)))
        {
            assetData.add(row.get(j + 2));
            assetData.add(row.get(j + 3));
        }
    }
}
```

Figure 4: Depicts an example of the input class accessing data using the database access class

This project can store and maintain data as a result of accessing database. The reason this is important is because it allows a user to store data about a certain business on their account. So, when they exit and then log into the application again, they can view and edit that data.

Iteration

```
//Obtains asset data
for (int i = 0; i < data.size(); i++)
{
    ArrayList<String> row = data.get(i);

    for (int j = 0; j < columnCount; j = j + 4)
    {
        if (email.equalsIgnoreCase(row.get(j)) &&
                businessName.equalsIgnoreCase(row.get(j + 1)))
        {
            assetData.add(row.get(j + 2));
            assetData.add(row.get(j + 3));
        }
    }
}
```

Figure 5: Shows an iteration that traverses through each member of asset data within the Input class

Classes such as the input class utilize iteration in order to go through each member of an array or array list. The reason it is so useful is because it significantly reduces the amount of code. An example is presented in the figure above.

Prototyping

Multiple different prototypes were made and were tested by an advisor and an end user before reaching this final project. This was done in order to better understand the different issues with the product and how it could be improved upon in order to tailor to the client's wants and needs. For example, in the first prototype, the initial business report had lacked crucial information but as my advisor was able to test it, the issue was spotted, and the project was enhanced.

Main Methods

The use of implementing main methods in a majority of the classes within this project allows for alpha testing. Each class can be tested individually and thus, it avoids facing large errors and problems when maintaining and creating this large application. For example, the database access class has a main method in order to check if all the methods for accessing data work correctly before any other class utilizes it.

Abstraction

The use of abstraction in this program provides more efficiency and reduces complexity. This is evident in the computational classes as all but the relevant data about an object of that class is hidden.

Conditional Statements

```
//Checks if register field is empty
if (registerField.getText().equals(""))
{
    errorObj = new Error("Please Enter a Business Name");
}
else
{
    businessName = registerField.getText();

    businessObj = new BusinessFinancials();
    riskObj = new Risk();

    //Displays answer through general output frame
    generalOutputObj = new GeneralOutput("The Business has been Registered");
}
```

Figure 6: Shows an example of a conditional statement used within the Input class

The use of conditional statements within this project is that it allows for executing specific code under certain circumstances. For the example shown above, the program does not begin creating a new business object if the user entered nothing for the business name.

Tools Used

Java Derby

Java derby was used in order to store information within a database. Derby was specifically used as it works well with Java as it is based on it. For example, the database access class obtains data from the database fairly easily.

Java API

Java API was used in order to create the overall GUI and structure of the program as it is efficient and compatible with Java. An example of its use is seen in the Welcome frame where it utilizes JFrame.

Word Count: 1133