Introduction:

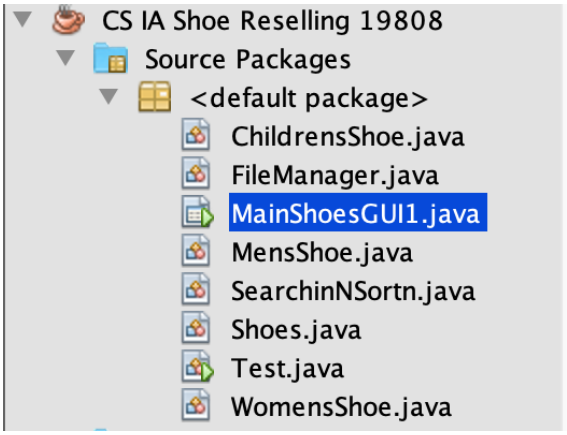
The product is a NetBeans IDE program that utilizes Object Oriented Programming. A program that is meant to keep track of bought shoes, and its profit after reselling. It accepts information from the user based on what they input about the product. It can search and sort through the data that is either in the users inventory, or their previous transactions. It also includes the option for potential versatility, that can be further diversified if required by the user.

Summary of Programming Techniques:

- Switch case
- Parameter passing
- For loop
- Nested loops
- Method returning a value
- User defined objects made from an OOP "template" class
- Encapsulation of private methods that work on public attribute of a "template" class
- Making a List of objects
- Simple and compound selection (if/else)
- Sorting (bubble sort etc.), and in particular sorting an array of objects based on one key attribute
- Searching (linear search, binary search…)
- Saving to a file
- Opening a file to a table
- Error handling (for example catching a divide by 0 error, or a null pointer while using an array of object…)
- Option pane generation for communicating with the user
- GUI tabs
- Use of a flag value (such as -999, or "not set yet")
- Overloaded constructors, which work differently depending on the parameters sent

- Inheritance between a super class and a sub class (Shoe with different genders and Sorting/Searching)
- Polymorphism (Shoe to MensShoe, WomensShoe, ChildrensShoe)
- Linked List
- Not used:
    - Due to the fact that I believed recursion would not have been as necessary or efficient
    - Array or ArrayList was not utilized, due to the fact that the data being worked with is dynamically and changes size, so to be less memory and efficiency consuming lists were used

Structure of Program



Relationships

- Is a (Inheritance)
- Has (Aggregation)
- Uses a (Dependency)
- MainGUI1 has a shoe
- FileManager has shoes
- ChildrensShoe is a Shoe
- WomensShoe is a Shoe
- MensShoe is a Shoe
- MainGUI1 uses SearchinNSortn
- MainGUI1 uses FileManager

Inheritance

I used inheritance to further differentiate a shoe into the specifics that the shoe of each gender has. I did not necessarily have to utilize it as much since my client did not

desire much specification between the shoes he conducts his business with. Although, I had these attributes created for future potential.

```
22        super(name, size, buyingPrice, sellingPrice, shoeDateBoughtDay,
shoeDateBoughtMonth, shoeDateBoughtYear, datebought, shoeDateSoldDay,
```

Polymorphism

Polymorphism was utilized due to the fact that the method for shoes was further diversified into the shoes for each individual gender.

```
20    public MensShoe(String name, double size, double buyingPrice, double
sellingPrice, int shoeDateBoughtDay, int shoeDateBoughtMonth, int
shoeDateBoughtYear, String datebought, int shoeDateSoldDay,

21            int shoeDateSoldMonth, int shoeDateSoldYear, String collaboration, String
brand, double profitability, String gender, double mensSize){

22        super(name, size, buyingPrice, sellingPrice, shoeDateBoughtDay,
shoeDateBoughtMonth, shoeDateBoughtYear, datebought, shoeDateSoldDay,

23            shoeDateSoldMonth, shoeDateSoldYear, collaboration, brand, profitability,
gender);

24        this.mensSize = mensSize;

25    }
```

Encapsulation

There were a plethora of variables/methods that were encapsulated within the classes that were specific to that class, which allowed for not only the ability to share methods and variables through public classes, but the private variables significantly reduced the error handling while going through the code.

```
91 private File getFile(String targetDirectory, int index) throws Exception {

 92        Scanner scanner;

 93        if(targetDirectory.equals("inventory")) {

 94            if(index > directories[0].listFiles().length-1) {

 95                throw new Exception(index+" is too big!");

 96            } else {

 97                return directories[0].listFiles()[index+1];
```

```
98          }
99
100     } else if(targetDirectory.equals("transaction")) {
101         if(index > directories[1].listFiles().length-1) {
102             throw new Exception(index+" is too big!");
103         } else {
104             return directories[1].listFiles()[index+1];
105         }
106     }
107     return null;
108 }
```

Aggregation

The MainGUI1 and FileManager have a shoe because that is what they are either working with for the tables or for saving to the computer.

```
public void addToShoeArray(String targetDirectory, Shoes shoe) throws IOException {
    if(targetDirectory.equals("inventory")) {
        int fileIndex = directories[0].listFiles().length-1;
        File file = new File("inventory/"+fileIndex+".txt");
        file.createNewFile();

        addTextToFile(shoe.getName(), file);
        addTextToFile(shoe.getSize(), file);
        addTextToFile(shoe.getBuyingPrice(), file);
        addTextToFile(shoe.getSellingPrice(), file);
        addTextToFile(shoe.getShoeDateBoughtDay(), file);
        addTextToFile(shoe.getShoeDateBoughtMonth(), file);
        addTextToFile(shoe.getShoeDateBoughtYear(), file);
        addTextToFile(shoe.getBoughtDate(), file);
        addTextToFile(shoe.getShoeDateSoldDay(), file);
        addTextToFile(shoe.getShoeDateSoldMonth(), file);
        addTextToFile(shoe.getShoeDateSoldYear(), file);
        addTextToFile(shoe.getCollaboration(), file);
        addTextToFile(shoe.getBrand(), file);
        addTextToFile(shoe.getProfitability(), file);
        addTextToFile(shoe.getGender(), file);
```

Dependency

The MainGUI1 uses the SearchinNSortn class to search and sort through the data in the tables. This class uses the FileManager in order to save to the computer and keep the data on the device rather than just on the application.

```
SearchinNSortn ss = new SearchinNSortn();
```

```
try {
    switch(sortByInventoryComboBox.getSelectedItem().toString())
    {
        case "Name":
        ss.AlphabetSortName(shoesStored);

        for(int i = 0; i < shoesStored.size(); i++)
        {
            setTableValueInventory(shoesStored.get(i), i);
        }

        break;

        case "Brand":
        ss.AlphabetSortBrand(shoesStored);

        for(int i = 0; i < shoesStored.size(); i++)
        {
            setTableValueInventory(shoesStored.get(i), i);
        }

        break;

        case "Size"  :
        ss.NumSortSize(shoesStored);

        for(int i = 0; i < shoesStored.size(); i++)
        {
            setTableValueInventory(shoesStored.get(i), i);
        }

        break;

        case "Collaboration":
        ss.AlphabetSortCollaboration(shoesStored);

        for(int i = 0; i < shoesStored.size(); i++)
        {
            setTableValueInventory(shoesStored.get(i), i);
        }

        break;
    }
} catch(Exception ignored) {}
```

Classes/Objects/Data Abstraction

The use of objects and classes, provide a level of abstraction that aids in reducing the difficulty to debug, and reuse of code, which aids in code efficiency and modularity.

```
public class Shoes {
    private String name = "Not Set Yet"; //If the name has a number in it, convert it
    private double size = -99999.99999;
    private double buyingPrice = -99999.99999;
    private double sellingPrice = -99999.99999;
    private int shoeDateBoughtDay = -9999999;
    private int shoeDateBoughtMonth = -9999999;
    private int shoeDateBoughtYear = -9999999;

    private String dateBought = "unknown";

    private int shoeDateSoldDay = -9999999;
    private int shoeDateSoldMonth = -9999999;
    private int shoeDateSoldYear = -9999999;
    private String collaboration = "Not Set Yet";
    private String brand = "Not Set Yet";
    private double profitability = -99999.99999;

    private String gender = "NA";

    //Overloaded constructor method
    public Shoes(){

    }

    //Constructor method
    public Shoes(String name, double size, double buyingPrice, double sellingPrice, int shoeDateBoughtDay, int shoeDateBoughtMonth, int shoeDateBou
            int shoeDateSoldMonth, int shoeDateSoldYear, String collaboration, String brand, double profitability, String gender) {
        this.name = name;
        this.size = size;
        this.buyingPrice = buyingPrice;
        this.sellingPrice = sellingPrice;
        this.shoeDateBoughtDay = shoeDateBoughtDay;
        this.shoeDateBoughtMonth = shoeDateBoughtMonth;
        this.shoeDateBoughtYear = shoeDateBoughtYear;
        this.shoeDateSoldDay = shoeDateSoldDay;
        this.shoeDateSoldMonth = shoeDateSoldMonth;
        this.shoeDateSoldYear = shoeDateSoldYear;
        this.collaboration = collaboration;
        this.brand = brand;
        this.profitability = profitability;
        this.gender = gender;

        this.dateBought = shoeDateBoughtDay + "/" + shoeDateBoughtMonth + "/" + shoeDateBoughtYear;
```

```
private void refreshTables() throws Exception {
    DefaultTableModel model1 = (DefaultTableModel)InventoryTable.getModel();//New model for saving the bought shoes
    clearTable(InventoryTable);
    shoesStored = new LinkedList<Shoes>();//LinkedList for the shoes that are bought but not sold
    for(int i = 0; i < fileManager.arrayCount("inventory"); i++) {
        Shoes tempShoe = fileManager.getShoe("inventory", i);
        model1.addRow(new Object[]{tempShoe.getName(), tempShoe.getBrand(), tempShoe.getSize(), tempShoe.getCollaboration(), tempShoe.getGender
        shoesStored.add(tempShoe);//Add the shoe to the Inventory table
    }

    DefaultTableModel model2 = (DefaultTableModel)TransactionTable.getModel();//New model for saving the sold shoes
    clearTable(TransactionTable);
    shoesSold = new LinkedList<Shoes>();//LinkedList for the shoes that are sold
    for(int i = 0; i < fileManager.arrayCount("transaction"); i++) {
        Shoes tempShoe = fileManager.getShoe("transaction", i);
        String soldDate = tempShoe.getShoeDateSoldDay() + "/" + tempShoe.getShoeDateSoldMonth() + "/" + tempShoe.getShoeDateSoldYear();
        model2.addRow(new Object[]{tempShoe.getName(), tempShoe.getBrand(), tempShoe.getSize(), tempShoe.getCollaboration(), tempShoe.getGender
        shoesSold.add(tempShoe);//Add the shoe to the Transaction table
    }
}
```

Data Structures Used

I primarily utilized LinkedList mainly due to the fact that the data that this program is working with requires for data to be added, and deleted, from either in the middle of the list or the end. Since there is no minimum or maximum of the number of shoes that would be stored, the list of data would have to be dynamic, since the amount of data points required to be stored would increase or decrease accordingly to the users preference. Additionally, since the speed of the sorting and searching is not as pivotal to the user, since this program is mainly focused on keeping track of shoes, a LinkedList would not be at much of a disadvantage in comparison to the use of an ArrayList.

```
37 private FileManager fileManager = new FileManager();//Reads and writes data to either save or retreat from the hard drive

38     LinkedList <Shoes> shoesStored;//A dynamic list of all the shoes that have been bought but not sold (inventory)

39     LinkedList <Shoes> shoesSold; //A dynamic list of all the shoes that have been sold (transactions)

40     LinkedList <Shoes> shoesTemp = new LinkedList<Shoes>();//Used when displaying search results and in the process of moving shoes

41

42

43     SearchinNSortn ss = new SearchinNSortn();
```

Main Unique Algorithms

```
private void refreshTables() throws Exception {
    DefaultTableModel model1 = (DefaultTableModel) InventoryTable.getModel();
    clearTable(InventoryTable);
    shoesStored = new LinkedList<>();
    for (int i = 0; i < fileManager.arrayCount("inventory"); i++) {
        Shoes tempShoe = fileManager.getShoe("inventory", i);
        model1.addRow(new Object[]{tempShoe.getName(), tempShoe.getBrand(), tempShoe.getSize(),
            tempShoe.getCollaboration(), tempShoe.getGender(), tempShoe.getBoughtDate()});
        shoesStored.add(tempShoe);
    }

    DefaultTableModel model2 = (DefaultTableModel) TransactionTable.getModel();
    clearTable(TransactionTable);
    shoesSold = new LinkedList<>();
    for (int i = 0; i < fileManager.arrayCount("transaction"); i++) {
        Shoes tempShoe = fileManager.getShoe("transaction", i);
        String soldDate = tempShoe.getShoeDateSoldDay() + "/" + tempShoe.getShoeDateSoldMonth() + "/" + tempShoe.getShoeDateSoldYear()
        model2.addRow(new Object[]{tempShoe.getName(), tempShoe.getBrand(), tempShoe.getSize(),
            tempShoe.getCollaboration(), tempShoe.getGender(), tempShoe.getBoughtDate(), soldDate, tempShoe.getProfitability()});
        shoesSold.add(tempShoe);
    }
}
```

Refresh the Inventory Table

model1 = new Table Model get INVENTORY TABLE model

clear INVENTORY TABLE

new SHOES STORED Linked List

loop i from 0 to number of rows in the INVENTORY TABLE

tempShoe = get shoe of index i from INVENTORY TABLE

Add a row to INVENTORY TABLE with new objects of the shoe name, brand, size,

collaboration, gender, the day bought

Store the shoe of index i in the INVENTORY TABLE

end loop

Refresh the Transaction Table

model2 = new Table Model get TRANSACTION TABLE model

clear TRANSACTION TABLE

new SHOES SOLD Linked List

loop i from 0 to number of rows in the TRANSACTION TABLE

    tempShoe = get shoe of index i from TRANSACTION TABLE

    SOLD DATE = SOLD DAY and SOLD MONTH and SOLD YEAR

    add a row to TRANSACTION TABLE with new objects of the shoe name, brand,

    size, collaboration, gender, the day bought

    Store the shoe of index i in the TRANSACTION TABLE

    end loop

```
if (fieldsNotBlank() || 0 < Integer.parseInt(dateBoughtDayTF.getText()) || 32 > Integer.parseInt(dateBoughtDayTF.getText())
        || 0 < Integer.parseInt(dateBoughtMonthTF.getText()) || 13 > Integer.parseInt(dateBoughtMonthTF.getText())) {
    double d = Double.parseDouble(boughtShoeSizeSpinner.getValue().toString());

    Shoes newShoe = new Shoes(boughtShoeNameTF.getText(), d, (int) boughtShoePriceSpinner.getValue(), 0, Integer.parseInt(dateBought

    try {
        fileManager.addToShoeArray("inventory", newShoe);

    } catch (IOException ex) {
    }
    boughtShoeNameTF.setText("");
    boughtShoeBrandTF.setText("");
    boughtShoeCollaborationTF.setText("");
    dateBoughtDayTF.setText("DD");
    dateBoughtMonthTF.setText("MM");
    dateBoughtYearTF.setText("YYYY");
    boughtShoeGenderComboBox.setSelectedIndex(0);
    boughtShoePriceSpinner.setValue(0);
    boughtShoeSizeSpinner.setValue(0);

    try {
        refreshTables();
    } catch (Exception ex) {
        Logger.getLogger(MainShoesGUI.class.getName()).log(Level.SEVERE, null, ex);
    }

} else {
    JOptionPane.showMessageDialog(null, "Do not leave any fields blank, or input values out of calendar boundaries!");
}
```

Adding a shoe

if fields are not blank or bought day > 0 or bought day < 32 or bought month > 0 or

bought day < 13

d = convert SHOE SIZE to double from inputted spinner value

NEWSHOE = new shoe get name, size, price, day date bought, month date bought,

year date bought, gender, brand, collaboration

try

Make File Manager Add NEWSHOE to INVENTORY TABLE

catch exception

set bought shoe name text field to blank

set bought shoe brand text field to blank

set bought shoe collaboration text field to blank

set bought shoe day date text field to DD

set bought shoe month date text field to MM

set bought shoe year date text field to YYYY

set bought shoe name text field to original combo box index

set bought shoe name text field to original value

set bought shoe name text field to original value

try

Refresh Tables

catch exception

ignore exception

else

Show Error Message

end if

```
private void sellMouseReleased(java.awt.event.MouseEvent evt) {
    // TODO add your handling code here:
    if(InventoryTable.getSelectedRow() >= 0 && 0 < Integer.parseInt(soldDay.getText()) && 32 > Integer.parseInt(soldDay.getText())
            && 0 < Integer.parseInt(soldMonth.getText()) && 13 > Integer.parseInt(soldMonth.getText()))
    {

        int day = Integer.parseInt(soldDay.getText());
        int month = Integer.parseInt(soldMonth.getText());
        int year = Integer.parseInt(soldYear.getText());

        String soldDate = day + "/" + month + "/" + year;
        int index = InventoryTable.getSelectedRow();
        int sellingPrice = Integer.parseInt(FinalSellingPrice.getValue().toString());


        try {
            Shoes temp = fileManager.getShoe("inventory", index);
            temp.setShoeDateSoldDay(day);
            temp.setShoeDateSoldMonth(month);
            temp.setShoeDateSoldYear(year);
            temp.setProfitability(sellingPrice-temp.getBuyingPrice());


            fileManager.moveToTransactions(fileManager.getShoe("inventory", InventoryTable.getSelectedRow()), index);
                refreshTables();
        } catch (Exception ex) {
            Logger.getLogger(MainShoesGUI1.class.getName()).log(Level.SEVERE, null, ex);
        }


        InventoryTable.clearSelection();

    } else {
        JOptionPane.showMessageDialog(null, "Do not leave any fields blank, or input values out of calendar boundaries!");
    }

    soldDay.setText("DD");
    soldMonth.setText("MM");
    soldYear.setText("YYYY");
    FinalSellingPrice.setValue(0);

}
```

Adding to Inventory table

if converted to int bought day > 0 and converted to int bought day < 32 and converted to int bought month > 0 and converted to int bought day < 13

Day = SOLD DAY text converted to int

Month = SOLD MONTH text converted to int

Year = SOLD YEAR text converted to int

SOLD DATE = Day + / + Month + / + Year

INDEX = selected row from INVENTORY TABLE

Selling Price = convert FINAL SELLING PRICE converted to String to int

try

temp = get shoe of index i from INVENTORY TABLE

for temp shoe set sold day as Day

for temp shoe set sold month as Month

for temp shoe set sold year as Year

for temp shoe set profitability as Selling price - get the BUYING PRICE

make FILE MANAGER move the shoe at the SELECTED ROW from INVENTORY TABLE to TRANSACTIONS TABLE

Refresh the tables

catch exception

ignore the error

clear row moved to TRANSACTION TABLE from INVENTORY TABLE

else

Show Error Message

end if

Set SOLD DAY text field blank

Set SOLD MONTH text field blank

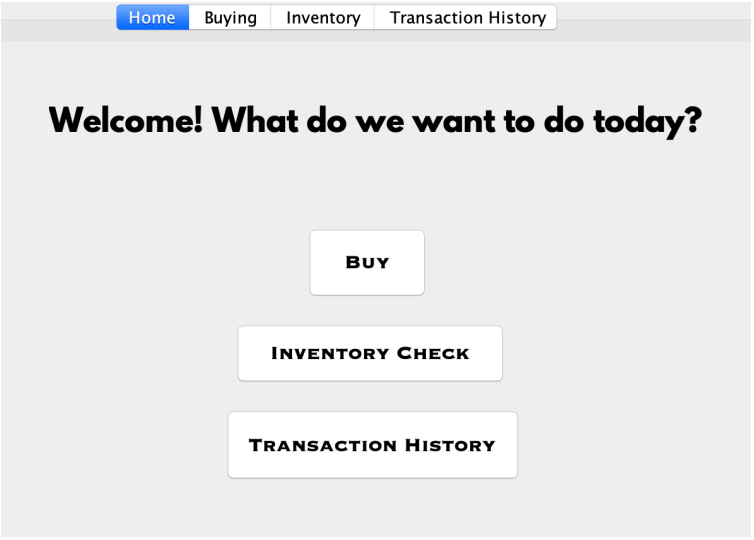
Set SOLD YEAR text field blank

Set SOLD PRICE spinner value to original

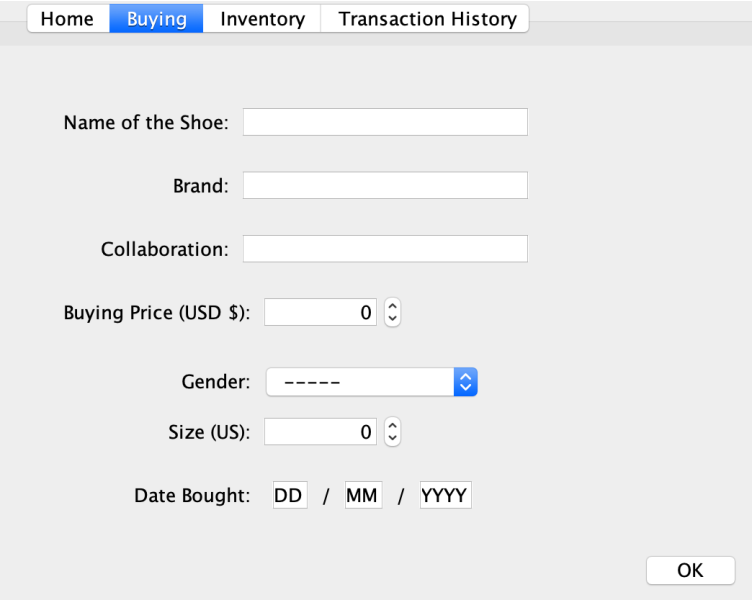
This database is able to save and remove the specific shoes (data) from the device memory in real time, hence providing efficient saving and removing, that is at utmost accuracy with current processes.

Using sequential search, instead of binary search, reduces the need to sort the data, which may not function due to potential errors with sorting with so many data points.

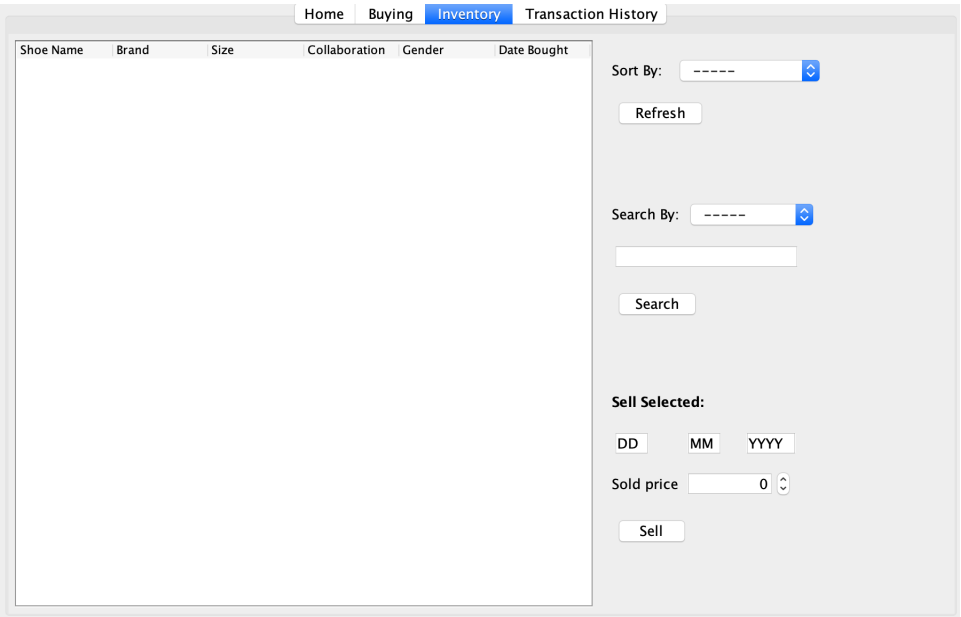
User Interface/GUI Work



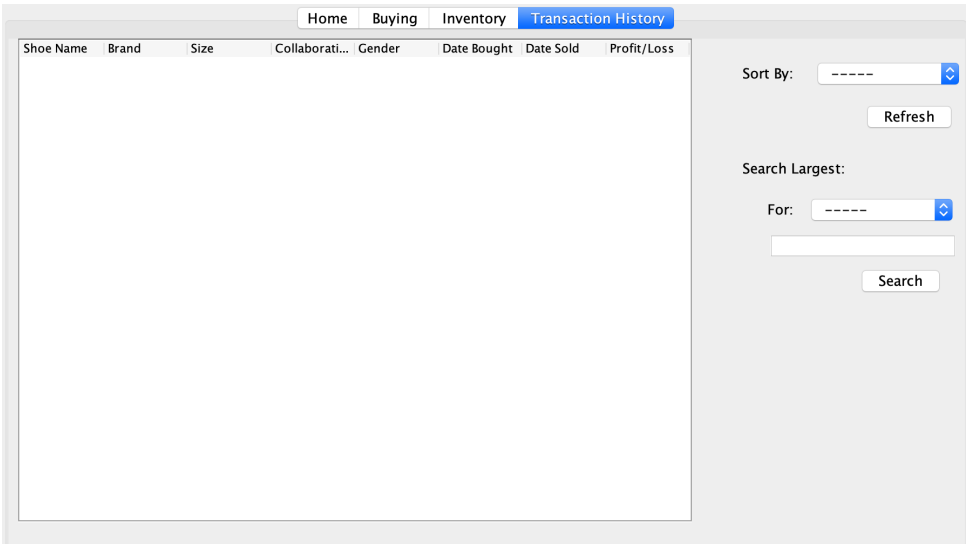
The simplistic design allows for the user to simply click on a button to go to the specific panel.



The labels instruct the user what to put, and includes a confirmation button. The spinner value, and its increments, and the gender combo box limit input errors.

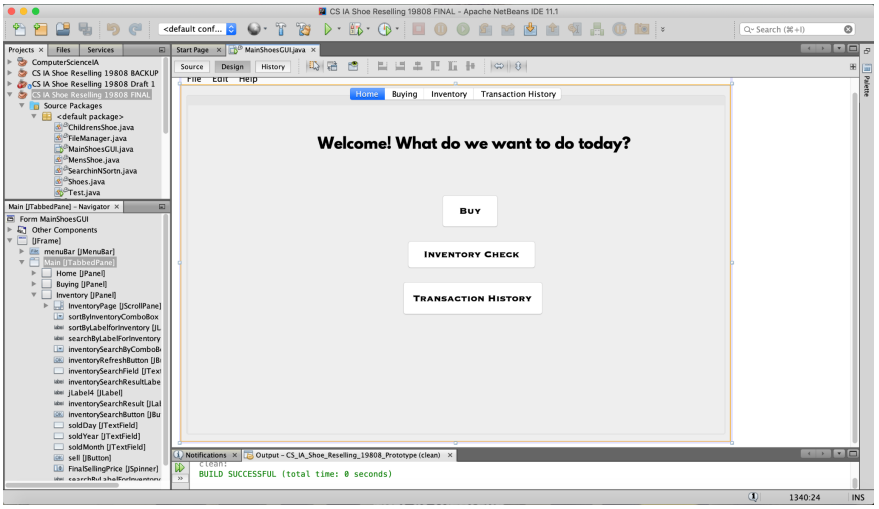


The table is able to allocate the data of each shoe accordingly. The sorting and searching allow for a variety of searching and sorting through the data, but are limited to the options given to them, to reduce the error possibility. Having a similar method to input the data about the selected shoe from the table being sold as the buying process reduces the conflict of understanding.



Having similar methods of sorting and searching processes to the inventory table allows for the ease for the user experience.

Software Tools Used



Utilizing Java NetBeans IDE allows for the ease in combining GUI elements to programming abilities. This has allowed to provide a level of abstraction that creates a sense of ease in programming the necessary actions following each user interaction with the GUI. Using NetBeans also allows for the programming to take advantage of Object Oriented Programming, which allows for not only ease in coding, but also code efficiency and modularity.

Word Count: 801