Team Members:
Grace Crawford
Leslye Morales
William Rittenhouse

Release 3 – Report

In this release our team had to implement two design patterns of our choosing that would work best with our code. We have been working on a code that basically creates a website called *Pet Shop* where a customer can create an account and purchase or return items, and an employee can also create an account and update the inventory. We decided that Null-Object Pattern and Composite Pattern, are two design patterns that would work well with our code.

Null-Object Pattern:

- **Location:** The Null-Object pattern has several files that make up the code. One of the files is AbstractEmployee.java, this file defines our variables for this code. Our other files associated with this pattern are, EmployeeFactor.java, EmployeeTest.java, NotEmployee.java, and RealEmployee.java. To be able to run the code we created an object instance in our Main.java file (lines 34 to 48) from our EmployeeTest.java.

- **Snapshot:**

```
Please enter your employee name:
Leslye
You entered Leslye
You are an employee!
Leslye
Employee list:
Will
Grace
Leslye
```

  - This output shows us how the program will react if an employee name is correctly inputted using Null-Object pattern.

```
Please enter your employee name:
John
You entered John
You are not an employee!
Employee list:
Will
Grace
Leslye
```

  - On the other hand, this shows us what the outcome will be if an employee name is inputted incorrectly using Null-Object pattern.

Team Members:
Grace Crawford
Leslye Morales
William Rittenhouse

- **Brief Description:** We choose to implement this design pattern because it applies to when the user wants to sign in as an employee. The way the code works is if they enter the correct name (“Will”, “Leslye”, or “Grace”) then they are greeted with a nice, successful message. If they enter the wrong name (“John”) then the code goes to the NotEmployee class where they are presented with a bad message. We believed that this was the best design pattern for our program, because it gives a default answer for when the user enters the wrong data.

Composite Pattern:

- **Location:** The Composite Pattern code can be found mainly in our Return.java file where we declared our variables, create our constructor, and methods necessary for our code to run. After that, to be able to execute this code we included the rest of it in our Main.java file (lines 47 to 181) where we included a switch statement to be able to run things as smoothly as possible.

- **Snapshot:**

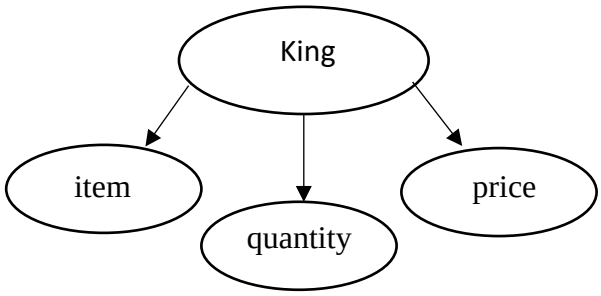
```
How many differnt categorical items are you returning? (Ex: Dog Items)
Please enter a number 1-5
2

What is the category of the item you are returning(Dog, Cat, Fish, Bird,
or Reptile):
Dog
What is the name of the item you are returning:
Dog Bed
What is the quantity you are returning:
1
What is the price of the item you are returning:
15.99

You are in the Dog category

Your item(s) have been returned to inventory.
Here is your receipt.

Receipt:
Item Name: Dog Bed
Quantity: 1
Price: 15.99
```

Team Members:
Grace Crawford
Leslye Morales
William Rittenhouse

```
What is the category of the item you are returning(Dog, Cat, Fish, Bird,
or Reptile):
Fish
What is the name of the item you are returning:
Fish Tank
What is the quantity you are returning:
2
What is the price of the item you are returning:
50.75

You are in the Fish category

Your item(s) have been returned to inventory.
Here is your receipt.

Receipt:
Item Name: Fish Tank
Quantity: 2
Price: 50.75
```

  - This output shows us the outcome of what happens when a user decides they want to return items back to our store.

- **Brief Description:** We decided to use the composite design pattern as well because this pattern focuses on treating a group of objects similarly to a single object, which we felt would work perfectly for our return situation. The way we implemented this into our code was by asking the user how many categorical items they wanted to return, from there they will be ask to input the category, item, quantity, and price. In the example above they inputted that they wanted to return two categorical items, so they had to also input category, item, quantity, and price twice. After they input the information needed, they will be given a receipt for each iteration, this is where we see how our composite pattern works. Since the composite pattern works in the terms of our tree structure this is how it will look:



In our code we created a temporary instance of a class called *King* which became our head node. From there we created a switch case that allowed the users to chose from different item categories. After the customer chooses the category, they get sent to input the item, quantity, and price which becomes our leaves. After everything executes successfully, a receipt will be printed out as shown in the snapshot above.