Phone Sorting System

Megan Koczur

Problem Statement

A cell-phone repair company collects phones that need to be fixed. Once they repair them, they need an easy and reliable way to sort them so they can be sent out.

Solution

The outside company provides all broken phones to the company that will refurbish them. Once the phones arrive, they are fixed by another group in the warehouse. The company needs to sort the phones by type before returning them to the other company. To do this, an operator is needed. The operator grabs a phone and scans the barcode on it. Once the barcode is scanned, a signal is sent to the LED on the storage system where the phone needs to be placed. Once the operator finds the LED, they place the phone in the box and press the LED to alert the system that the phone has been placed there. The phone count for that box is updated in the phone database as is the location of the phone. If a new box is needed, a label will be printed and the operator needs to place the label on the box. The operator continues to go through this process until a box fills up. Once the box is full, they seal the box, attach the shipping label, and send the full box of refurbished phones to the company.

Design Patterns and Principles

GRASP:

- Low coupling
- High cohesion
- Controller

SOLID

- Single Responsibility Principle
- Open Closed Principle
- Dependency Inversion Principle

GOF:

- Composite role
- Facade role
- Command role

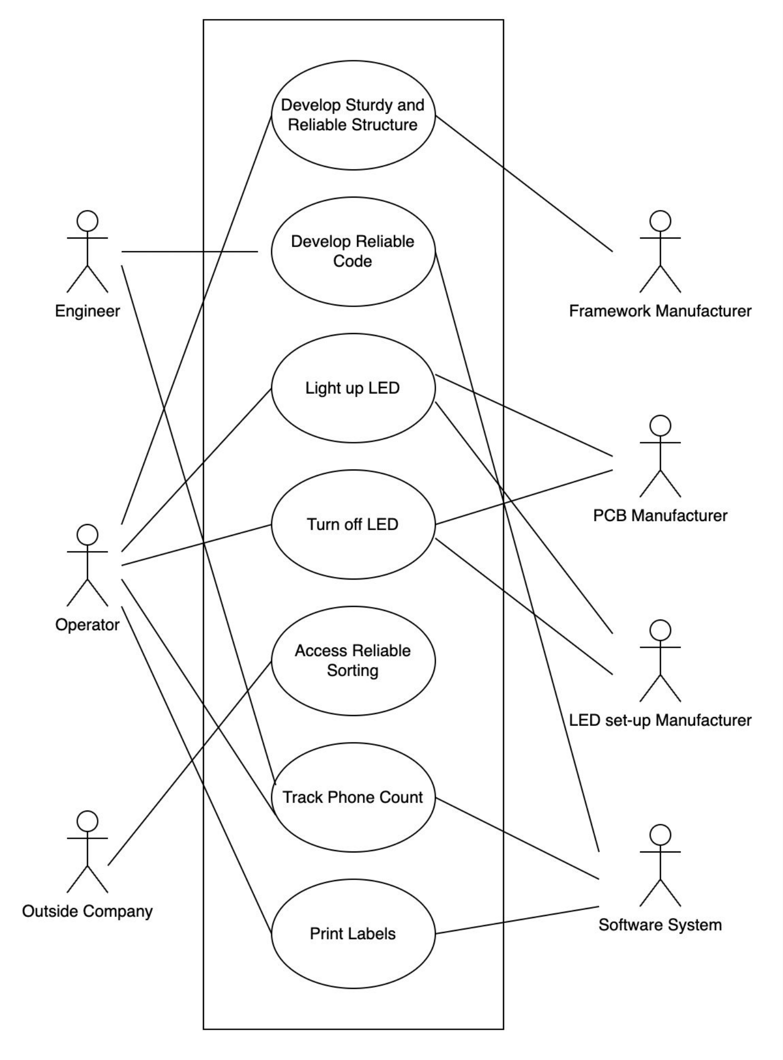
Microservices

- API Gateway
- Authentication

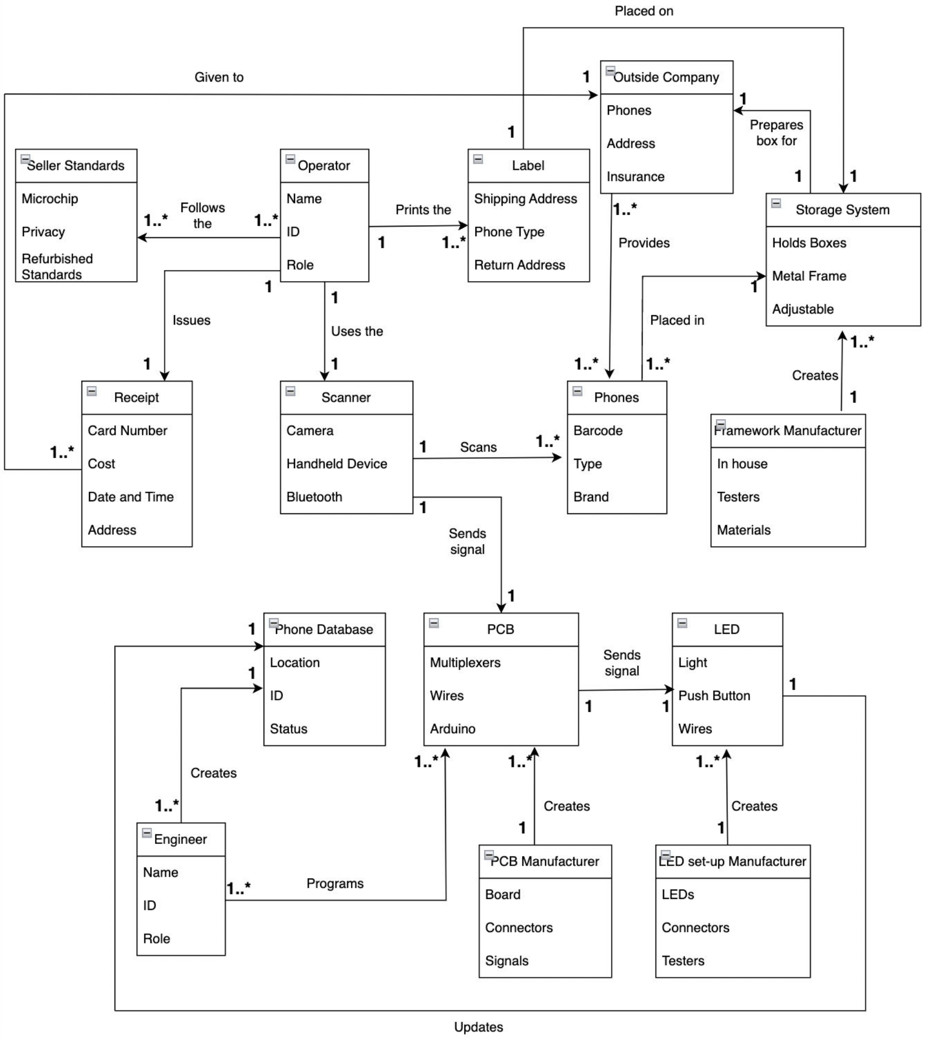
Problem Statement Again

A cell-phone repair company collects phones that need to be fixed. Once they repair them, they need an easy and reliable way to sort them so they can be sent out.

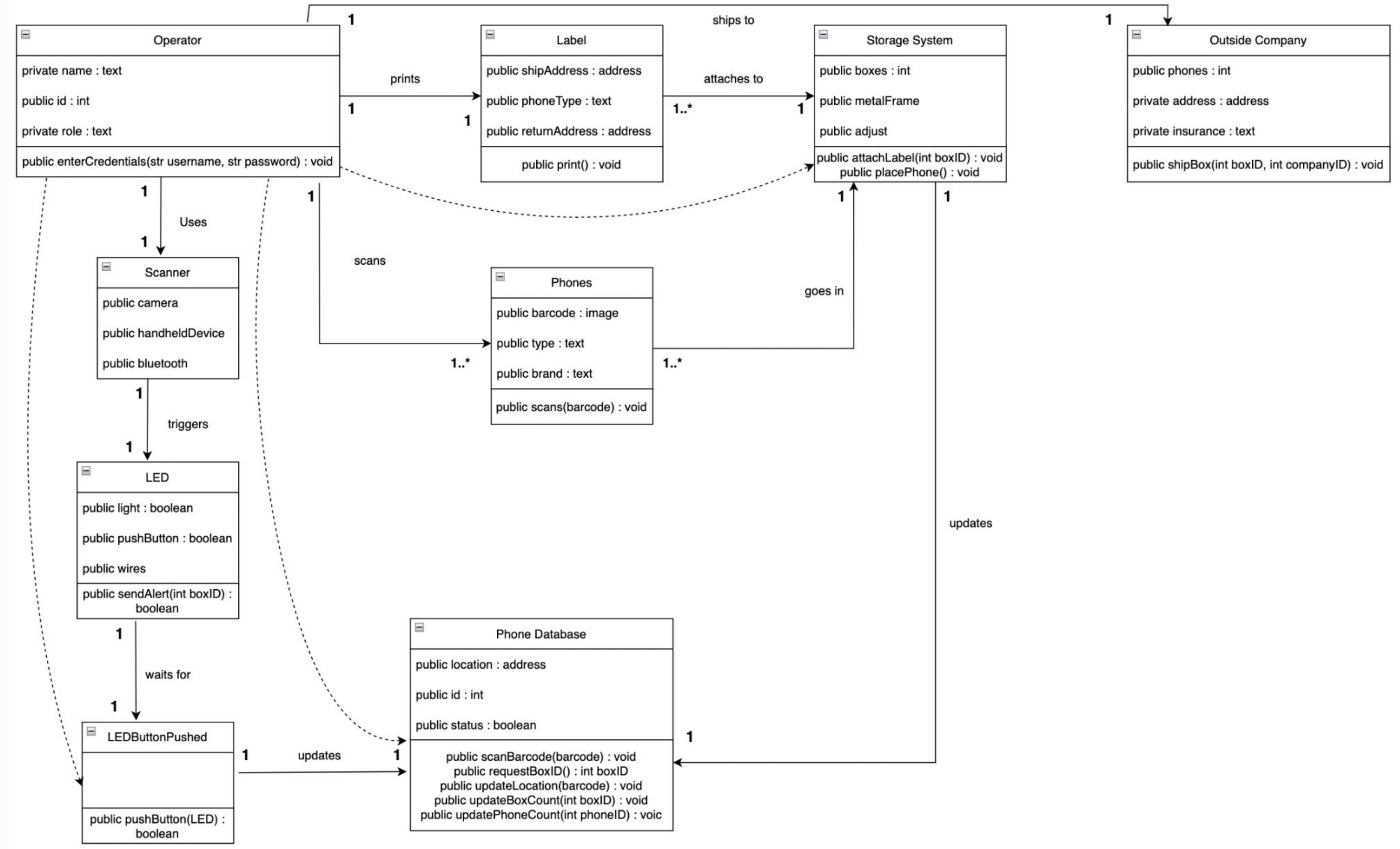
Use Case Diagram



Domain Model



Class Diagram



Operator

private name : text

public id : int

private role : text

public enterCredentials(str username, str password) : void

Label

public shipAddress : address

public phoneType : text

public returnAddress : address

public print() : void

Storage System

public boxes : int

public metalFrame

public adjust

public attachLabel(int boxID) : void
public placePhone() : void

Outside Company

public phones : int

private address : address

private insurance : text

public shipBox(int boxID, int companyID) : void

Scanner

public camera

public handheldDevice

public bluetooth

Phones

public barcode : image

public type : text

public brand : text

public scans(barcode) : void

LED

public light : boolean

public pushButton : boolean

public wires

public sendAlert(int boxID) : boolean

LEDButtonPushed

public pushButton(LED) : boolean

Phone Database

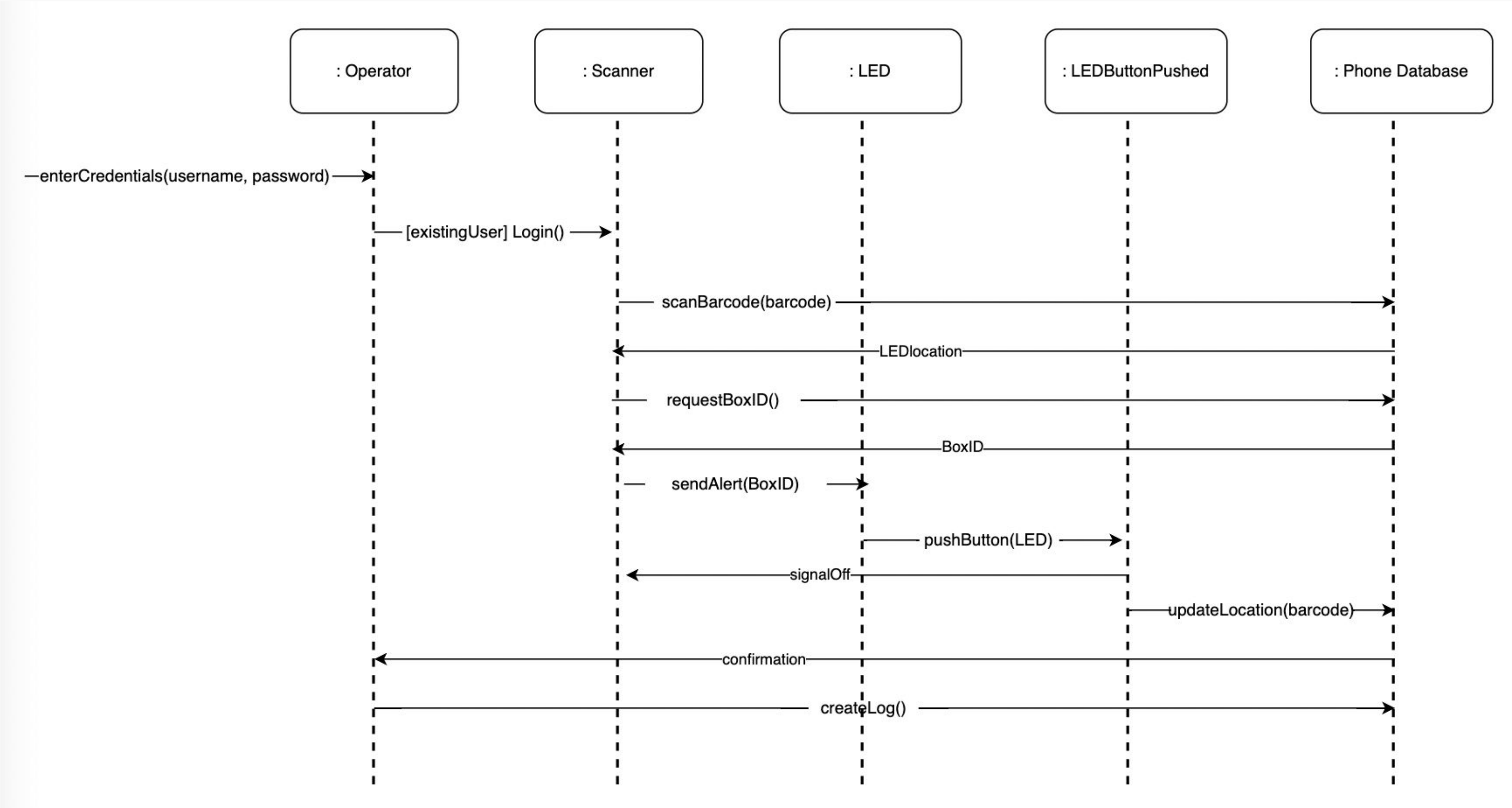
public location : address

public id : int

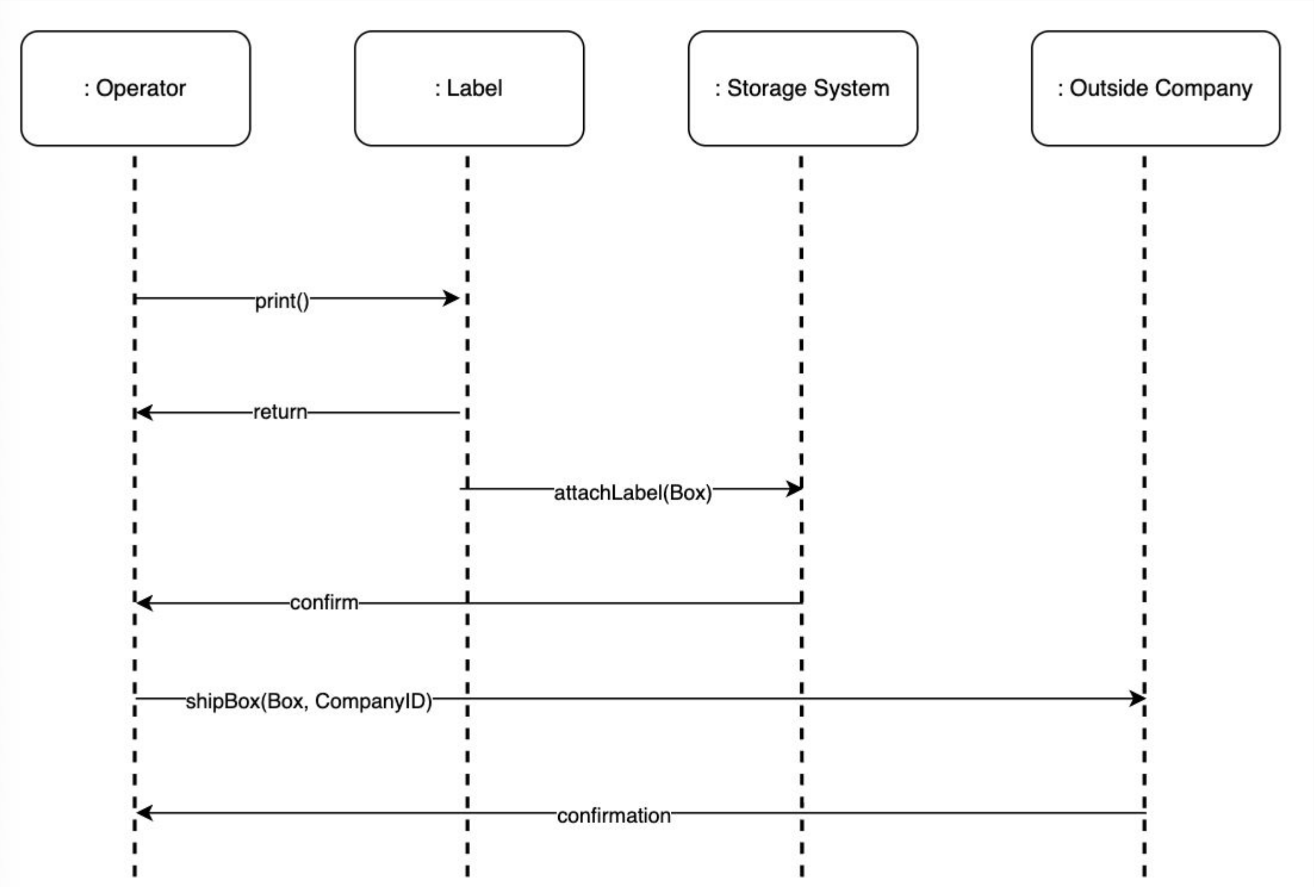
public status : boolean

public scanBarcode(barcode) : void
public requestBoxID() : int boxID
public updateLocation(barcode) : void
public updateBoxCount(int boxID) : void
public updatePhoneCount(int phoneID) : voic

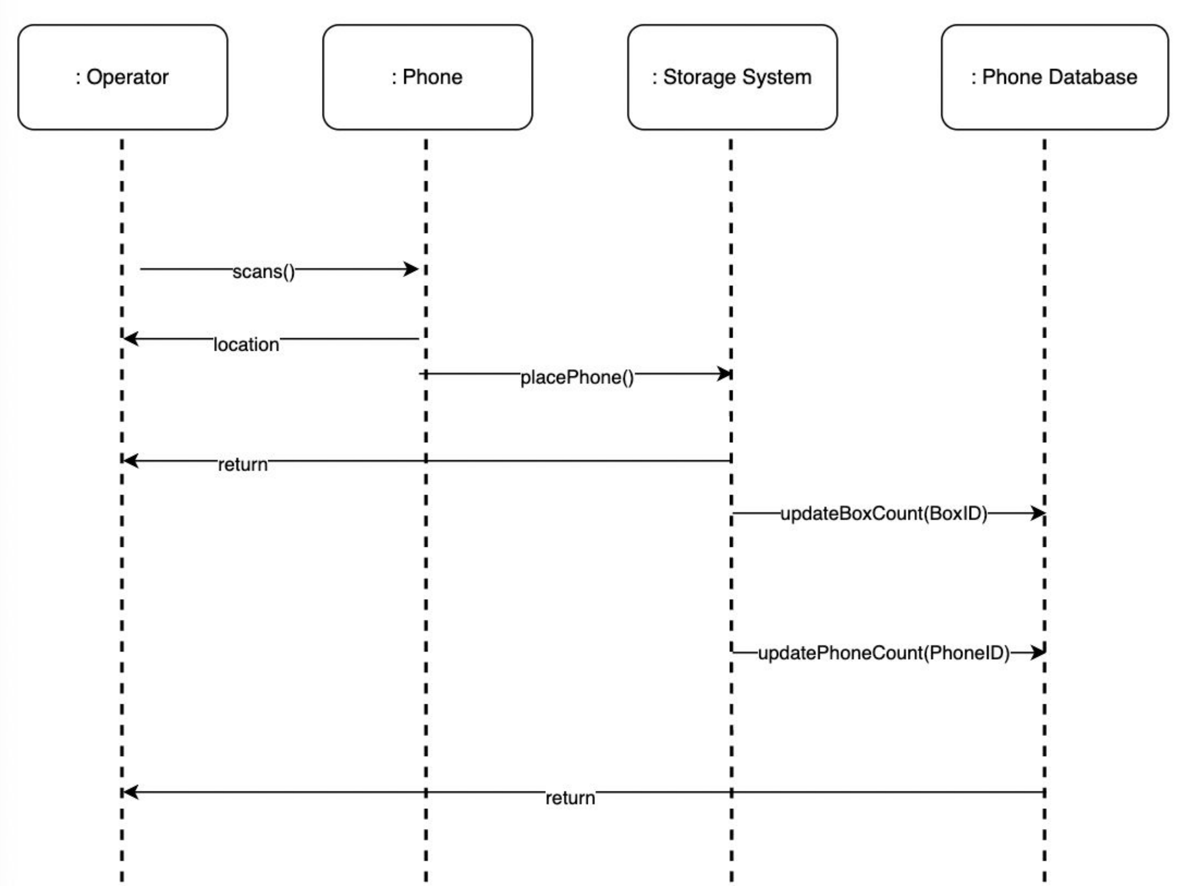
Sequence Diagram 1



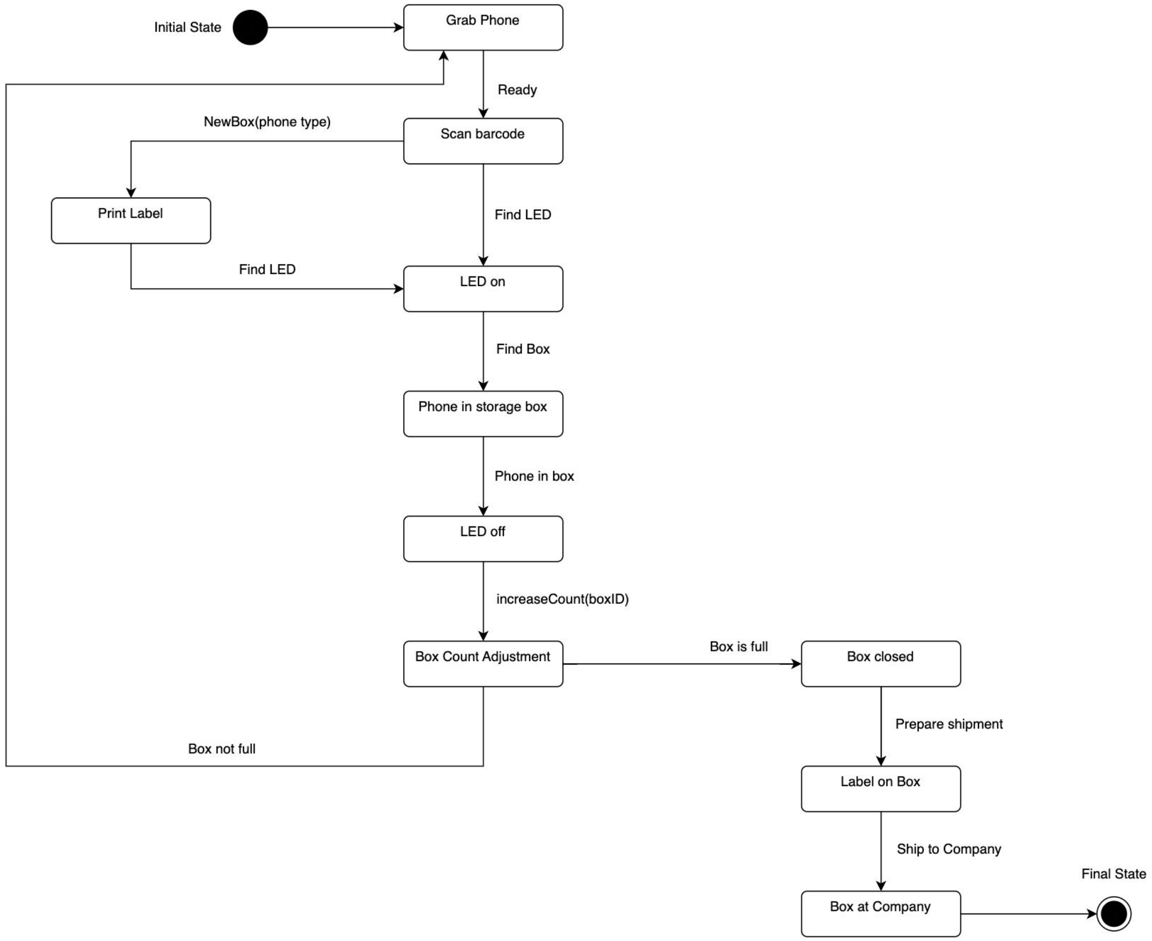
Sequence Diagram 2



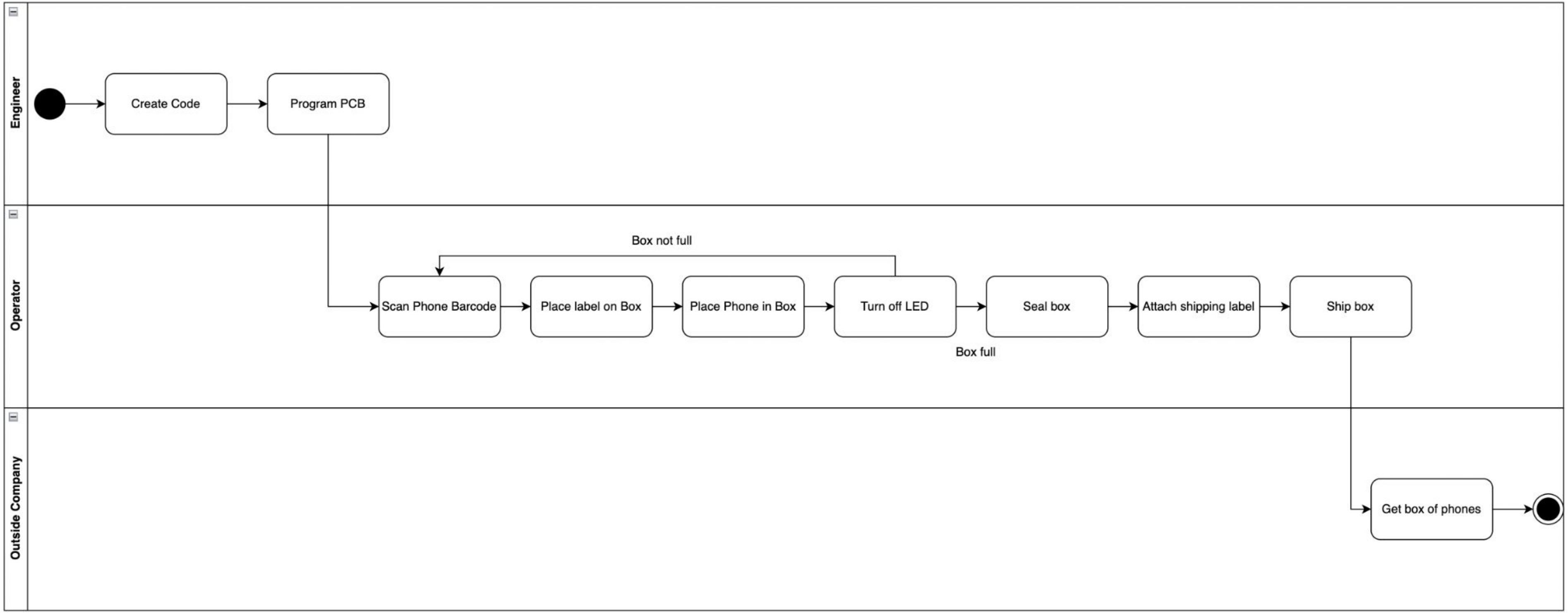
Sequence Diagram 3



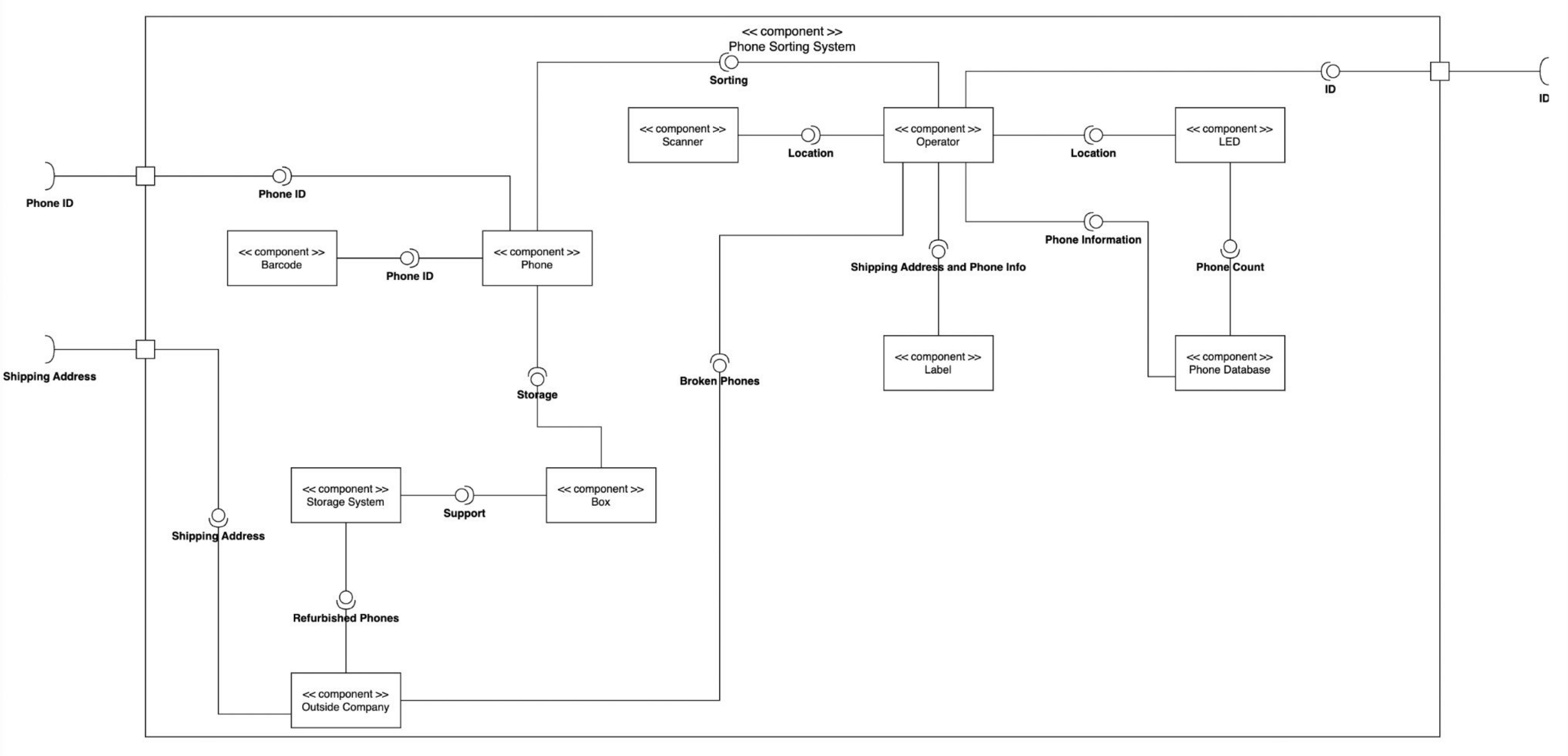
State Diagram



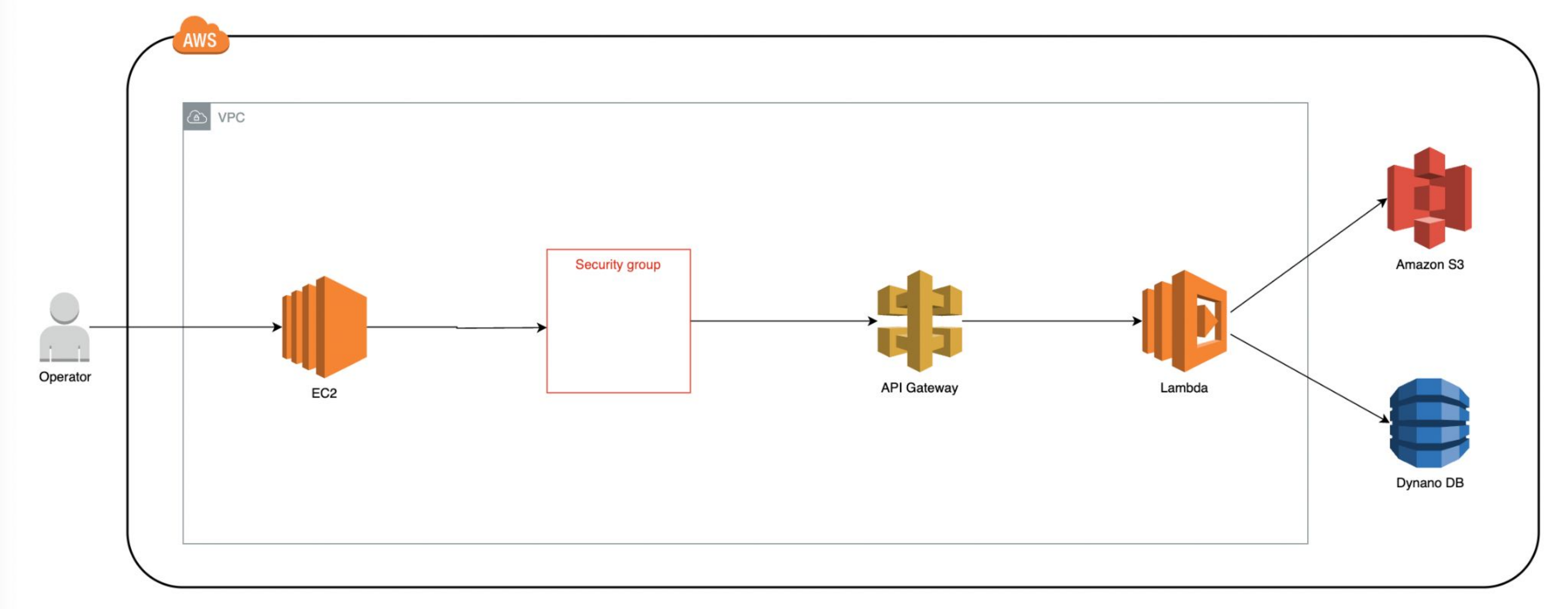
Activity Diagram



Component Diagram



Cloud Deployment Diagram



Architecture Patterns

Model-View-Controller (MVC)

- Model: This layer represents all of the data regarding the phones
- View: This layer deals with the UI that is seen on the scanner
- Controller: This layer is used to connect both the model and the view. The requests are gone through the model (phone database) and can then be displayed to the operator via the view (UI)

Design Principles

GRASP:

Low coupling: I implemented low coupling into my system as there are not too many connections between the classes.

High cohesion: I incorporated high cohesion into the system as most of the methods and attributes in a specific class are related to one another.

Controller: There needs to be logs for when the phones are refurbished and moved around. Instead of placing the responsibility of storing this information on the storage system itself, another class is created to focus solely on these logs.

SOLID:

Single Responsibility Principle is seen as most of the classes are responsible for one task.

Open Closed Principle is incorporated as attributes can easily be added to the classes without making large modifications.

Dependency Inversion Principle is used as the high-level classes are not affected by changes in the low-level ones.

Design Patterns

GOF:

The Operator Class is an example of a composite role as it is holding many components.

The Storage System class is an example of the Facade role as it encompasses a large number of subsystems.

The LEDButtonPushed class is an example of the Command role as this class is used solely to show the execution of the operator pushing the LED.

Microservices:

API Gateway is used for accessing the phone sorting service.

Authentication is used when scanning the phone barcode and allowing access to the system for the operator.

Conclusion

Key Takeaways:

Throughout this class, I learned many things about the software design process that I was either unaware of or have not used in the real world. I have been involved in many projects and only a few of them have incorporated proper planning. It definitely takes a lot of time, time we don’t always have, to work through all of these diagrams but I see how it would make the project development easier if we did. These are definitely things I will try to implement in my life moving forward as I do see the pros in utilizing them.

Acquire Software Design Skills and Better my Career:

I am constantly switching projects and starting new ones so I could see myself utilizing many of these tactics in my career. With new projects, I believe I could begin with creating a couple diagrams and sharing them with my team. The projects get complicated so it would be helpful to have visuals while implementing them. Perhaps we could avoid some of the small issues we run into along the way.