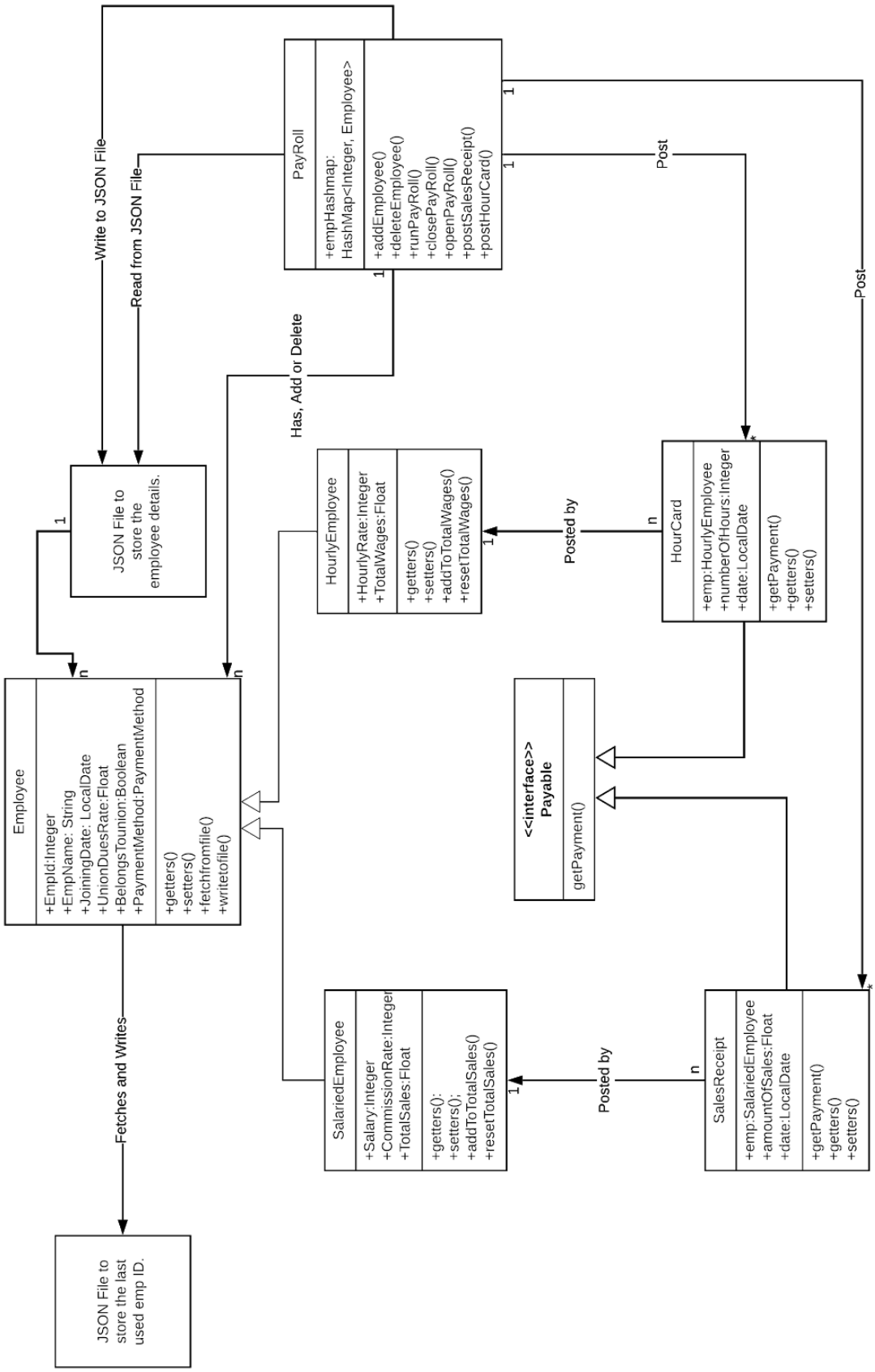
# Design Document

## Flinternship Training Assignment

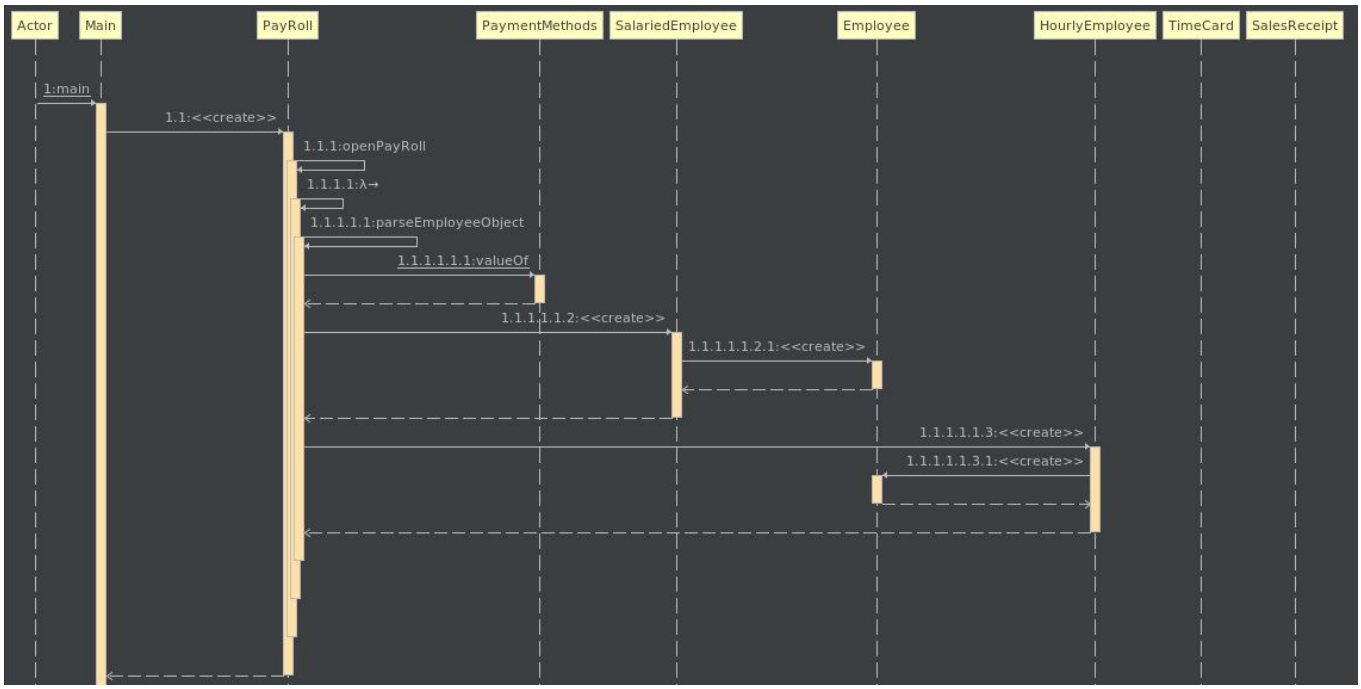
A Naga Neeramitra Reddy
naga.neeramitra@flipkart.com
EmpId: 140726



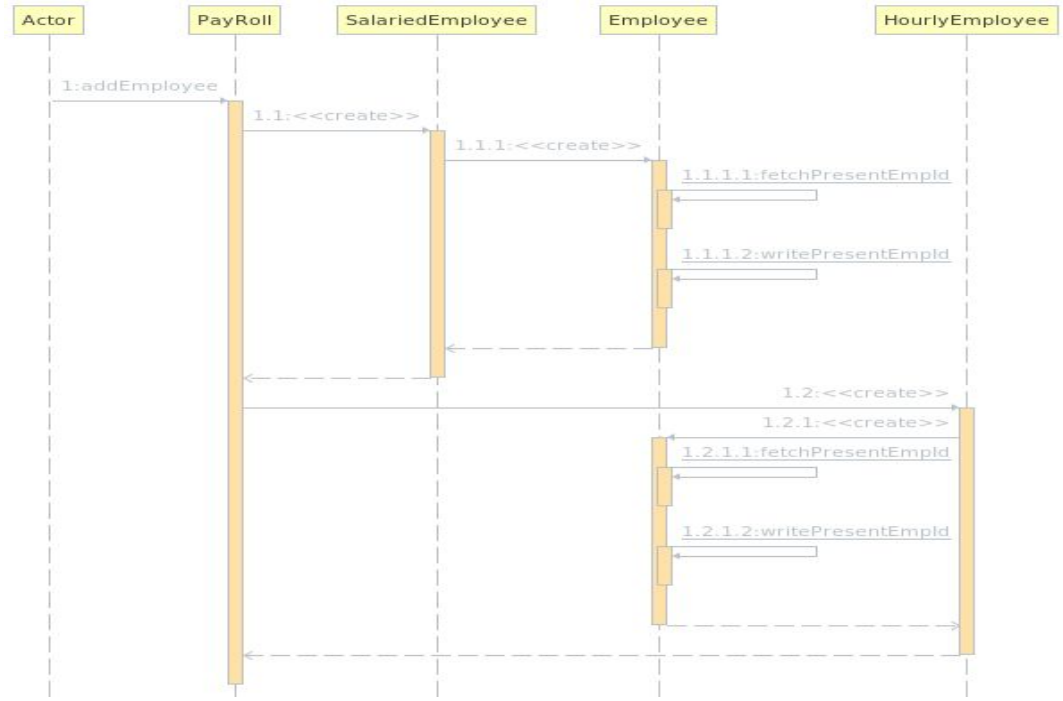

UML Class Diagram

# UML Sequence Diagrams

## Main()

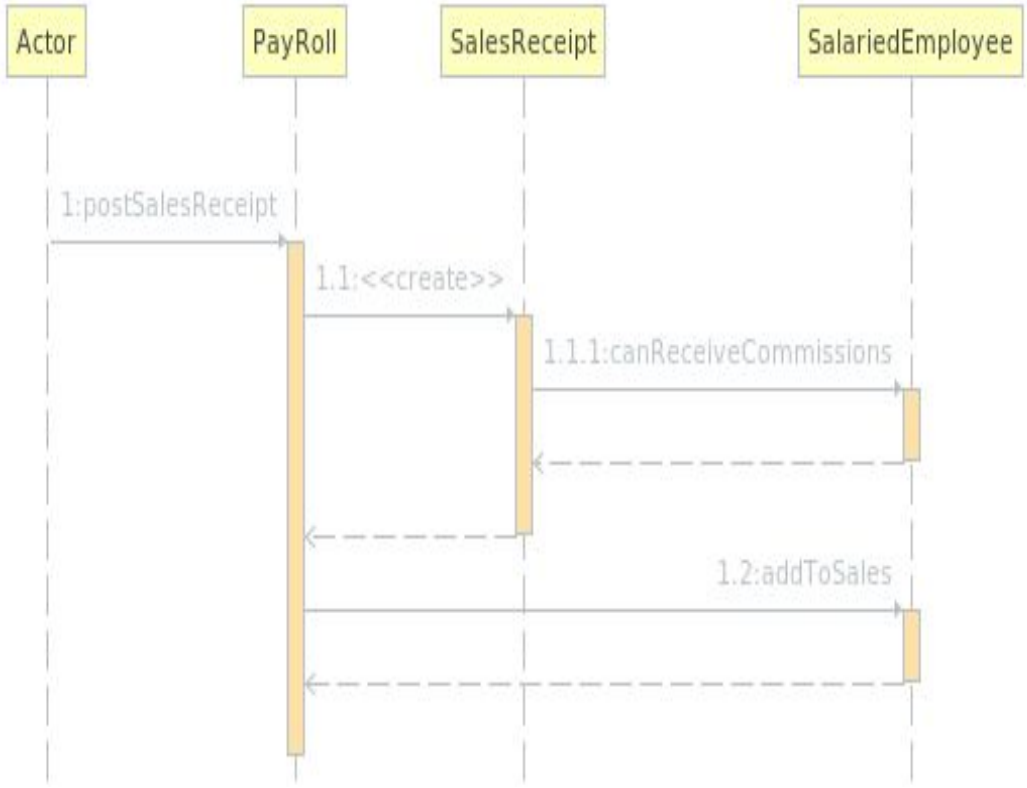



## Add Employee()

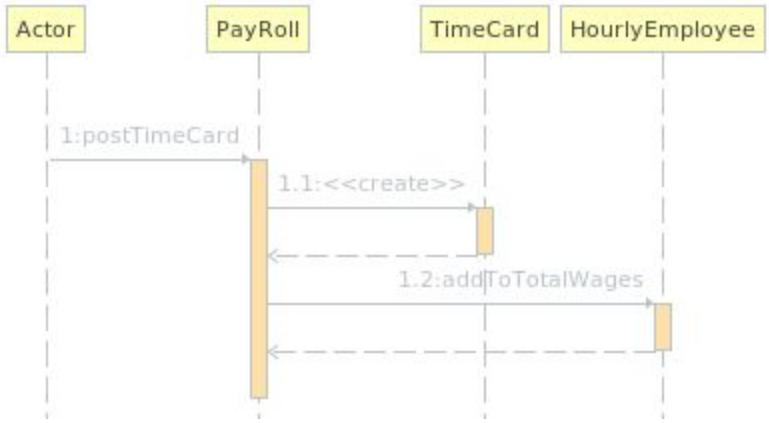


## DeleteEmployee()

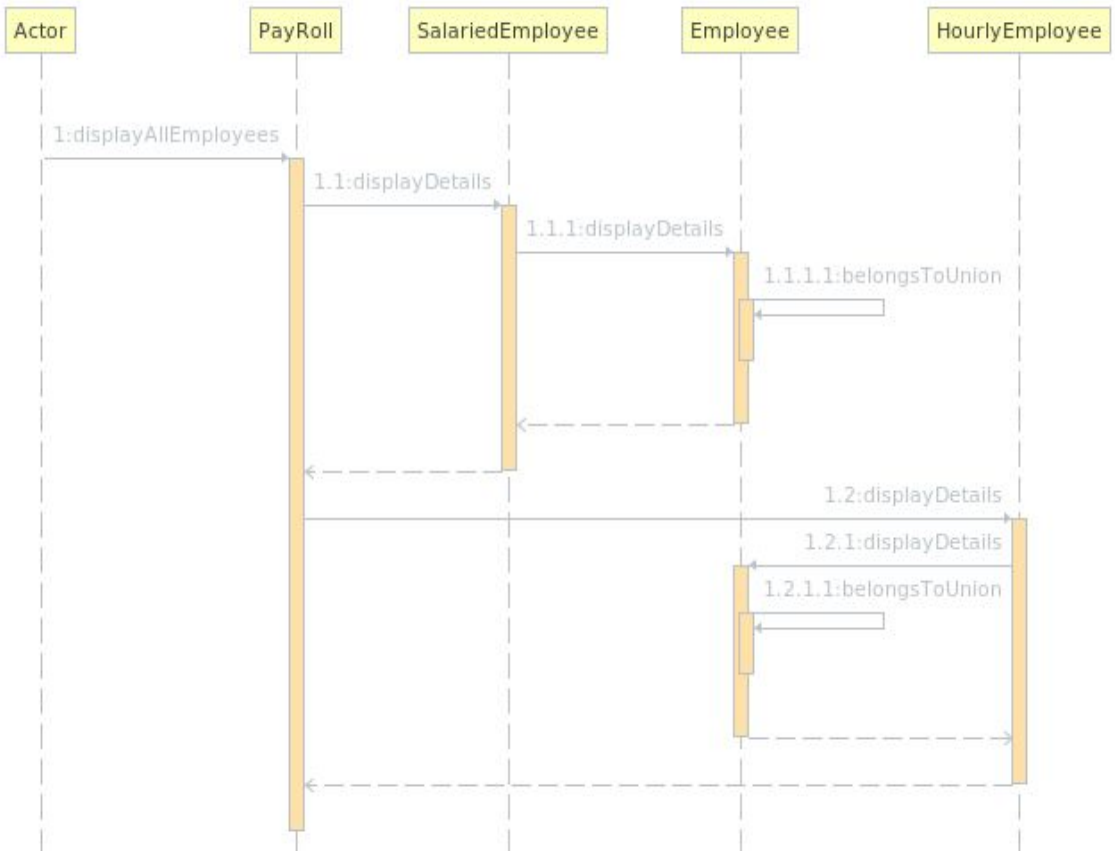

## PostsSalesReceipt()



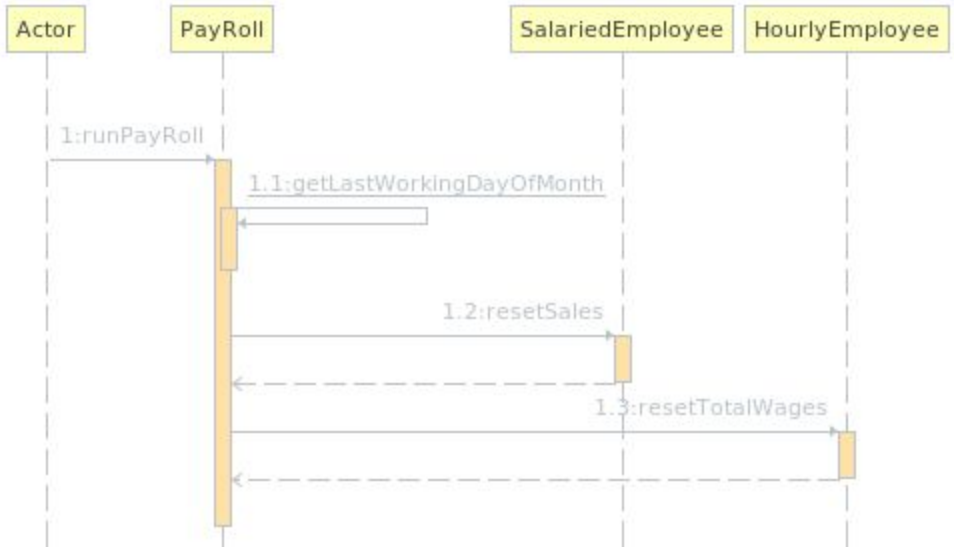
## PostHourCard()




## DisplayAllEmployees()



RunPayRoll()




# Design Objectives:

**Design towards types.**
**Design towards immutability.**
**Design with contracts.**
**Design with conventions rather than configuration.**
**Design towards abstract types.**
**Open to Extension, close to modification.**
**Loose coupling.**
**Encapsulation.**
**Polymorphism.**

# Basic Design Overview:

**Employee Class:** Employee attributes like name, Id etc. along with methods to fetch and write to the ID JSON file.

**The ID JSON File:** The ID JSON file is used to store the value of the static variable ID. This is to assign an unique ID to every new employee.

**Hourly and Salaried Employee (Employee subclasses):** The subclasses of Employee for hourly and salaried employees. Extra member attributes and extra methods. They override display of employee and all their members are final to enforce immutability.

## Payable Interface:

The payable interface has a method getPayment().

## SalesReceipt implements Payable:

This is a class for a sales receipt.

## HourCard implements Payable:

This is a class for an Hour Card.

## PayRoll class:

This is the class for the PayRoll. This class has attributes salariedemployeemap and hourlyemployeemap that are HashMaps that store <key, value> pairs of <empId, SalariedEmp> and <empId, HourlyEmp> respectively. This class has methods to read from and write to the JSON File that store employee details, to run the payroll, to add an employee, to delete an employee and to post sales receipts or hourcards.

## Employee Details JSON:

JSON file to store the employee details.

## Main:

This is the main class that is responsible for calling payroll. This class has an option driven user prompt that helps in running the payroll smoothly.

# Design implemented:

## Design towards Immutability:

This was implemented by making classes, methods and attributes final to prevent inheritance of classes, overriding of functions and changing the values of variables respectively.
Also the use of Enums instead of direct Strings has been done to ensure immutability and avoid hassles.

## Single Responsibility:

Single responsibility means assigning a single responsibility to every class and this has been achieved. I started off by making a rough sketch of all the various interactions and chalked out the necessary classes. No class has more than one responsibility.

## Design towards Concrete Types:

Most of the code is concrete classes and hence design towards abstract types couldn't be implemented.

## Design towards Contracts:

Barely implemented. Only one interface was used to enforce the contract on SalesReceipt and TimeCard that they will provide a getPayment() method.

## Moderate coupling:

The goal was loose coupling but due to a few bad design choices that I realized too late, coupling became moderate. I still strove towards reducing the coupling as much as possible. Satisfactory but could have been done better.

## Polymorphism:

Extensive use of polymorphism through the use of function overloading, function overriding and constructor overloading.

# Possible Design improvements:

## Design towards Abstract Types:

Abstract types could have been used more. The employee class could have been made an abstract class or an Interface with default methods.

## Single responsibility:

Instead of cramming the file interactions into the class files, I could have made file handlers and utility functions to better modularize the code. This I would have done if I had more time.

## Loose Coupling:

Instead of passing other objects to most methods of classes, I could have used handlers to process objects and retrieve the necessary information. Better use of single responsibility would have reduced the coupling. This could have been improved a lot.

## Open to extension, close to modification:

Instead of the use of inheritance, implementations of interfaces would have bettered this principle and provided more flexibility to the code. (As in easy changeability).

## Polymorphism:

Better polymorphism could have been implemented through the use of Variable arguments instead of Constructor and Function overloading.

# Design Contradictions and Design challenges:

The biggest challenge I encountered was designing the PayRoll class. Once I started designing the PayRoll class, a lot of contradictions started arising in the design of the other classes and I had to go back and fix them.

For example, I had to remove a few constructors, add variables and change methods in the Employee class and consequently in it's subclasses. This made me realize that I hadn't implemented open to extension, close to modification at all!

The subclasses Hourly and Salaried employees also turned out to be bad design as I had to use two seperate HashMaps to store the list of employees.

Interacting with the JSON files using json-simple was extremely hard and messy with a lot of type casting and type conversions happening. This could have been done way better using a database or Jackson API for JSONs. This could have been implemented if there was more time.

# Alternative Design:

A better design could have been achievable by following the design improvements listed in the earlier sections.

This design would have an Employable interface instead of Employee. This interface would contain default methods which can be overridden. This is the first step towards **designing by contracts.**

Instead of cramming file handling into the classes, a separate file handler class with utility methods could have been defined and used for cleaner, **less coupled** and more **single responsibility design.**

Instead of having two kinds of Employee subclasses, a single class with an attribute to denote the type along with conditional methods would have been better as the two subclasses weren't that different.

More error checks and better exception handling could have been done.

# Conclusion:

The Design is okay overall for the given time frame but could have been better.
The coding style could have been cleaner.
Documentation in the code base could have been way better (Couldn't do proper documentation due to the lack of time).
Should have spent more time reading the problem statement carefully and should have allocated more time to design instead of jumping into coding straight away.