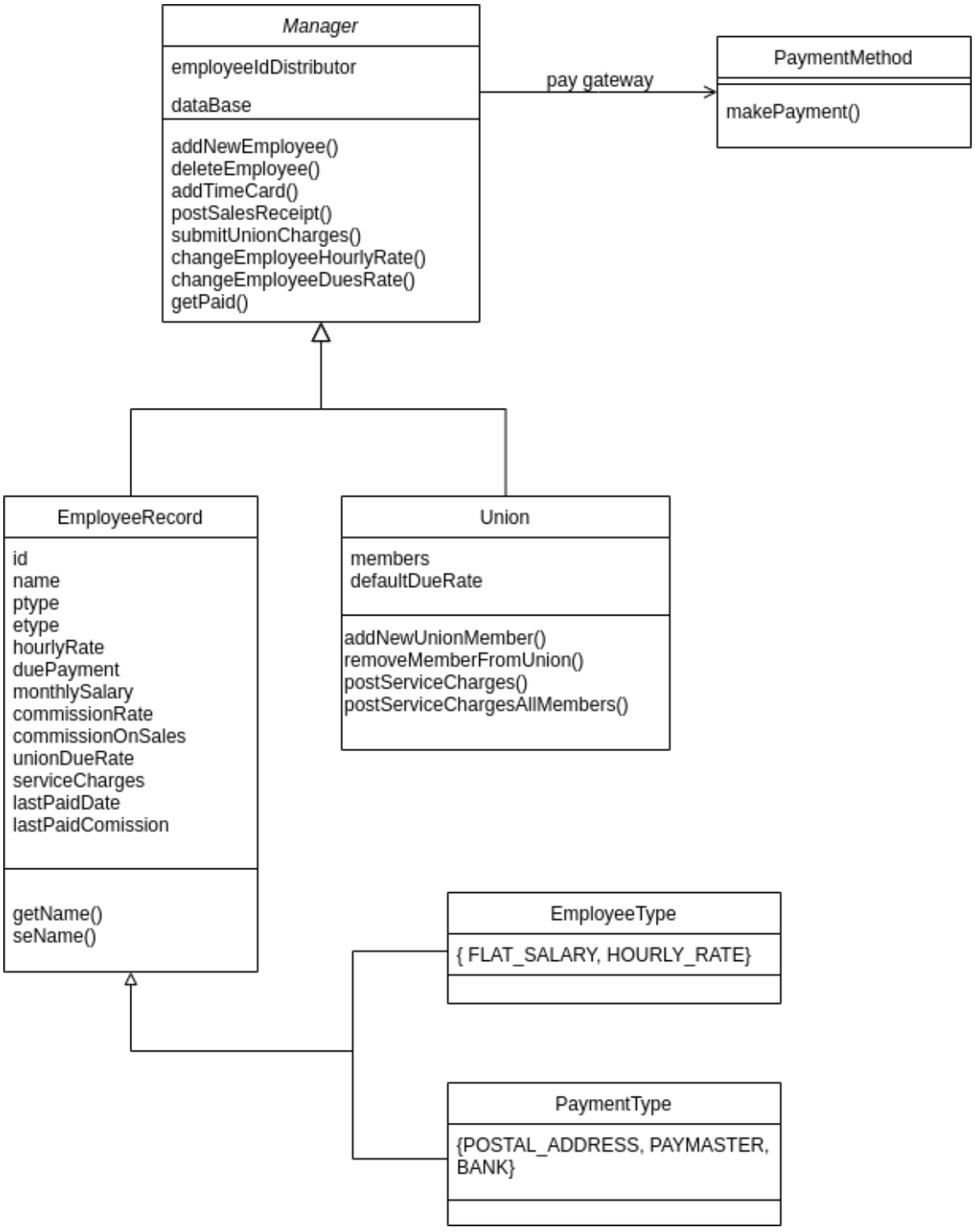
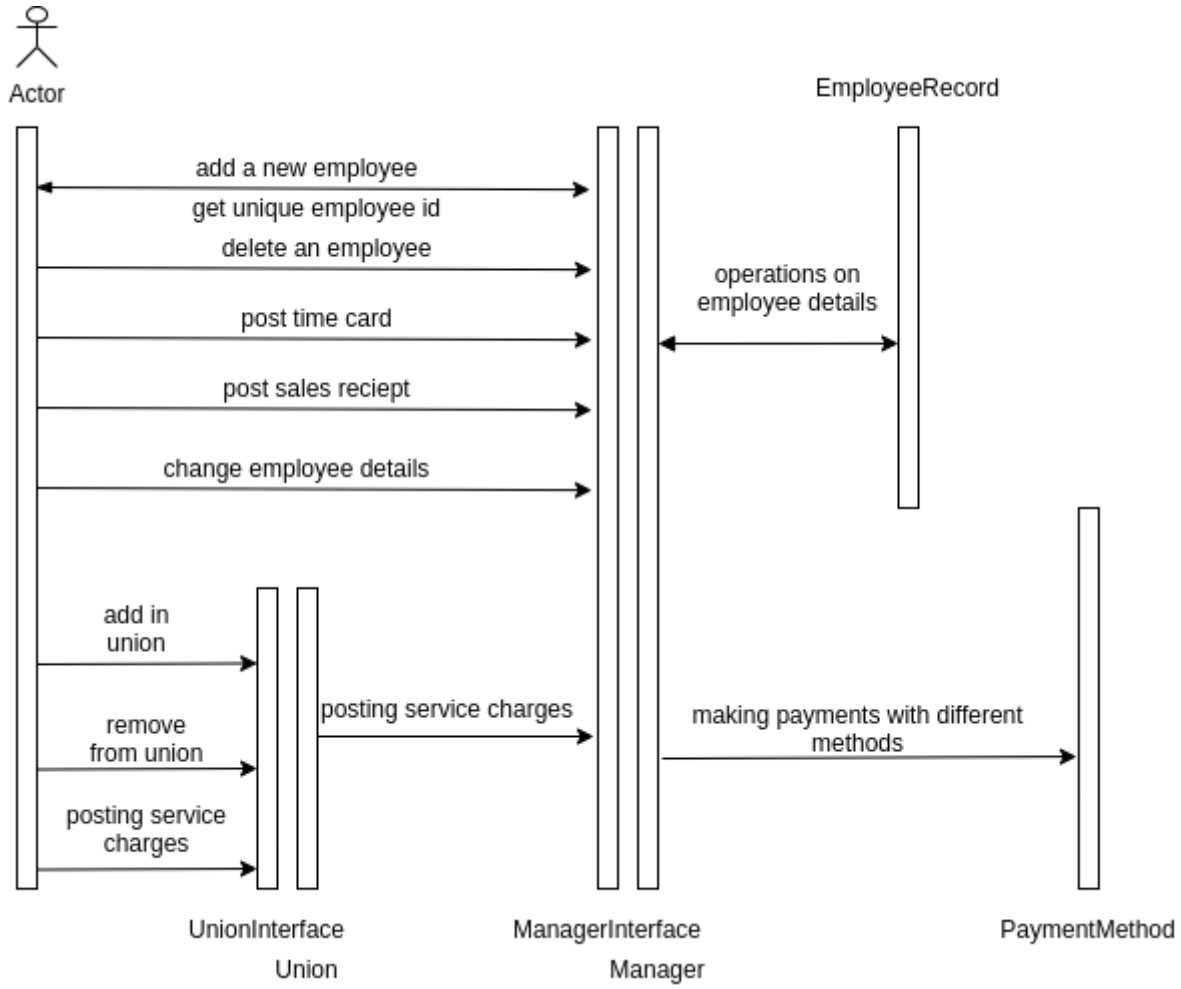
Required Diagrams

Relationship Diagram in UML





Required Documentation

Design Objectives

Objective: To design the architecture capable of handling:

- Employees that are paid a flat salary and the employees that are paid by the hour, inclusive of their individual payment dates.
- Salaried Employees must have a way of recieving commission as per their sale (with predecided and updateable rates.), with independent payment dates.
- Payment gateway inclusive of more than one types with which Employee can demand its payment to be made (where employee is easily able to change the methodology.)
- Union creation, registering and deregistering of a member. Such union would indepedently be able to function and charge their

members a fixed due rate (per weekly) and variable service charges as need be. These would be cut from the next payable amount of the concerned employee.

Design Approaches

The approach that I am going to withhold would be in close proximity of efficiency for the developers and simplicity for users but more than one approaches are possible.

- For easy access to robust functions, a `Manager` class would be handling the functions as: `addNewEmployee()`, `deleteEmployee()`, `addTimeCard()` etc. This would be scalable will integrating databases, as the `Manager` will held the connection to the private stream from database, providing its APIs to such functions.
- An overall approach hence becomes:
    - `Manager`: to manage all the required reciever end of activities possibly performed by employee (like changing details, or posting sales record.)
    - `EmployeeRecord`: to record all states of variables for a particular employee (analgous to a row in a database table —scalability.)

More detailed description in design choices.

Design Role, Responsibilities, Choices and Reasoning.

- Treating `EmployeeRecord` as analogous to a row of database table:
    - This would enable us to reduce overhead functionalities in record class. And so very easily manage in the `Manager` class with or without dealing with a database.
    - In the case of parallelism, the locking of certain shared resources could be handled from `Manager` class with relying on the individual record to hold a lock—a bit far-fetched, but certainly advantageous.
    - All variables can be manipulated in a single object.—It gives a coding and clarity advantage. Whenever developer wants to introduce a new feature for employee record, the `EmployeeRecord` can be utilized.
- An enum `EmployeeType` would enable us to differentiate the type of employee from `FLAT_SALARY` to `HOURLY_RATE`:
    - *Reason* : Easily scalable—any other type of employee would be efficiently inserted into enum and used by

implementing its behaviour in `Manager`. Using `EmployeeType` of check is also secure with the unwanted manipulation of inter-variables. It would make sure either flat salary variables have been used or hourly rate ones.

- With similar reasoning as above, an enum `PaymentType` would enable us to differentiate betwen the type of payment method available to use. — `{POSTAL_ADDRESS, PAYMASTER, BANK }`
- In lieu of `PaymentType`, a dedicated class architecture `PaymentMethod` will be highly scalable in writing methods for independent gateways.
    - *Reason:* `PaymentMethod.makePayment(empId, paymentType, amount)` —this function would pay the `amount` utlitizing `paymentType` methodology with the secure exchange of credentials of OAuth for employee `empId`. A design of payment gateway being architecturally independent from main class is required in robustness and usability. Say later a new service from `PayTM` becomes available, one can add that method in this class and its behaviour can be used then.
- Architecture of `Union` would be capable of function on its own. It should be able to post different types of service charges to its memeber, welcome a new memeber, remove a memeber (thereby removing its all due rates and charges):
    - Such a design had been implmented with a separate class `Union` and its interface `UnionInterface`.
    - Design Choice: `Union` class would only be able to work under a `Manager` instance. This would ensure that `Union` is not charging its members unnecessarily. Any charges by `Union` should not go unmoderated, otherwise it would create a security flaw with the updatation of employee records if accidentally `Union` asked a large sum from its member. An instance of `Manager` will moderate with required designs.

- `lastPaidDate` for a newly joined employee is premised to be last Friday for employees working on per hour basis.
    - *Reason* : If a person joins any other day besides Friday, say Saturday, paying him the next Friday assures a good behavioral attitude to the new employee. And besides, it paid next to next Friday, would cause a possible situation of `lastPaidDate` initially being after today—requiring developer to write code that would be check-bounded of `lastPaidDate` exceeding today. Hence it is assumed to be last Friday. If a person joins on Friday itself, assuming `lastPaidDate` as last Friday, would enable him to earn

payment by reported hours for today only.—Coding and behavioural advantage.

- Similar argument for choosing the `lastPaidCommission` to be the previous to previous Friday. (because it should be paid every other Friday.)
- A public class `Date` with static methods to manipulate the date as per need. For example: `getLastFriday()`, `getLastToLastFriday()`, or `lastWorkingDayOfMonth` would be helpful over the classes.—Reasoning with coding advantages.

Future Improvements

- Integration of a database for secure storage of data even after a shutdown of running machines.
- Moderation of union charges and external updates by manager more exhaustively.
- Better design of payment gateways in secure and encrypted format.

Design Challenges, Alternative Desgin

- A good alternative design could be the separate of employees with flat salary and hourly rate, with two individual classes extending a super class employee.—This will enable a hierarchy of instance variables that could be used with appropriate access modifiers. This is better in the sense that only those instance variables will be used which would be needed, but it would increase the complexity of the employee lists.—Pros and cons. One has to think more on this.

Submitted by - Ankit Solanki