SOLID Case Study

Daum Corp.
백명석

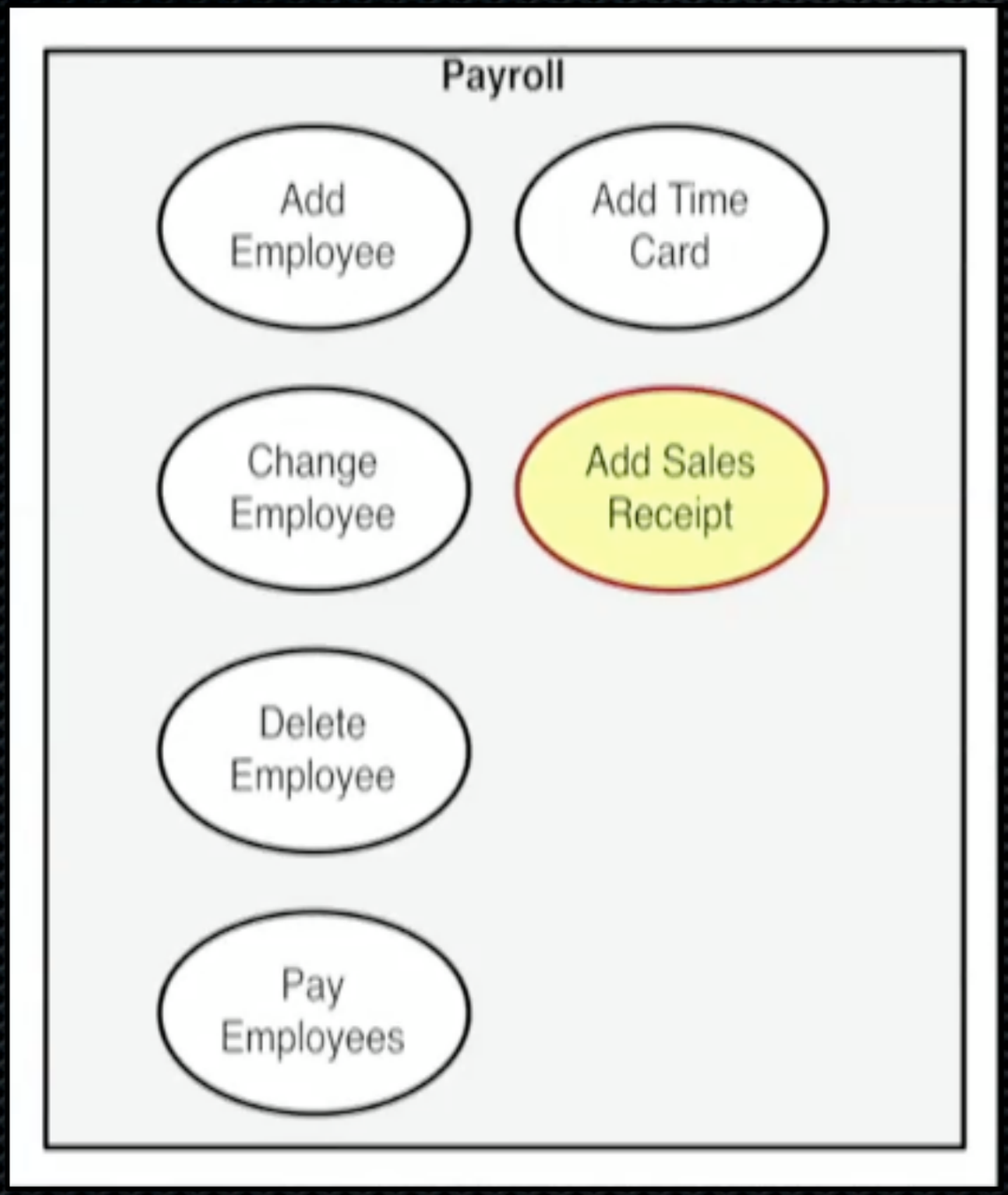
발표목차

- Requirements and Use Cases
  - Use Case List
  - Entity(Data Element) List
- SRP
- OCP
- Example
- 생각해 볼 것들

Requirements and Use Cases

- 요구사항을 분석하면서 Use Case/Entity List 작성
- Use Case List
  - AddEmployee
  - DeleteEmployee
  - ChangeEmployee
  - AddTimeCard
  - PayEmployees
  - AddSalesReceipt
- Entity(Data Element) List
  - Employee
    - CommissionedEmployee
    - HourlyEmployee
    - TimeCard
    - SalesReceipt

Requirements and Use Cases



Data Dictionary
commissioned-employee = base-pay + commission-rate
hourly-employee = hourly-rate
sales-receipt = date + amount-sold
time-card = date + hours-worked

- Data Entity들은 분명히 내부에 데이터를 갖는다.

Requirements and Use Cases

- employee type에 따라 급여 수령 주기가 다르다(BI-WEEKLY, WEEKLY, MONTHLY)

```
Data Dictionary
    commissioned-employee = base-pay + commission-rate + BI-WEEKLY
    hourly-employee = hourly-rate + WEEKLY
    salaried-employee = salary + MONTHLY
    sales-receipt = date + amount-sold
    time-card = date + hours-worked
```

Requirements and Use Cases

- employee에 따라 pay disposition이 다르다.
- 고객의 욕구사항을 들으면서 고객이 사용하는 단어에서 암시하는 데이터를 추출한다.
- Use Case가 복잡해지는 것은 데이터가 변경되기 때문
- Use Case의 오퍼레이션이 변하기 때문은 아님

```
Data Dictionary
    commissioned-employee = base-pay + commission-rate + BI-WEEKLY +
            pay-disposition
    hourly-employee = hourly-rate + WEEKLY + pay-disposition
    salaried-employee = salary + MONTHLY + pay-disposition
    pay-disposition = {mail | PAYMASTER | direct-deposit}
        mail = address
        direct-deposit = account
    sales-receipt = date + amount-sold
    time-card = date + hours-worked
```

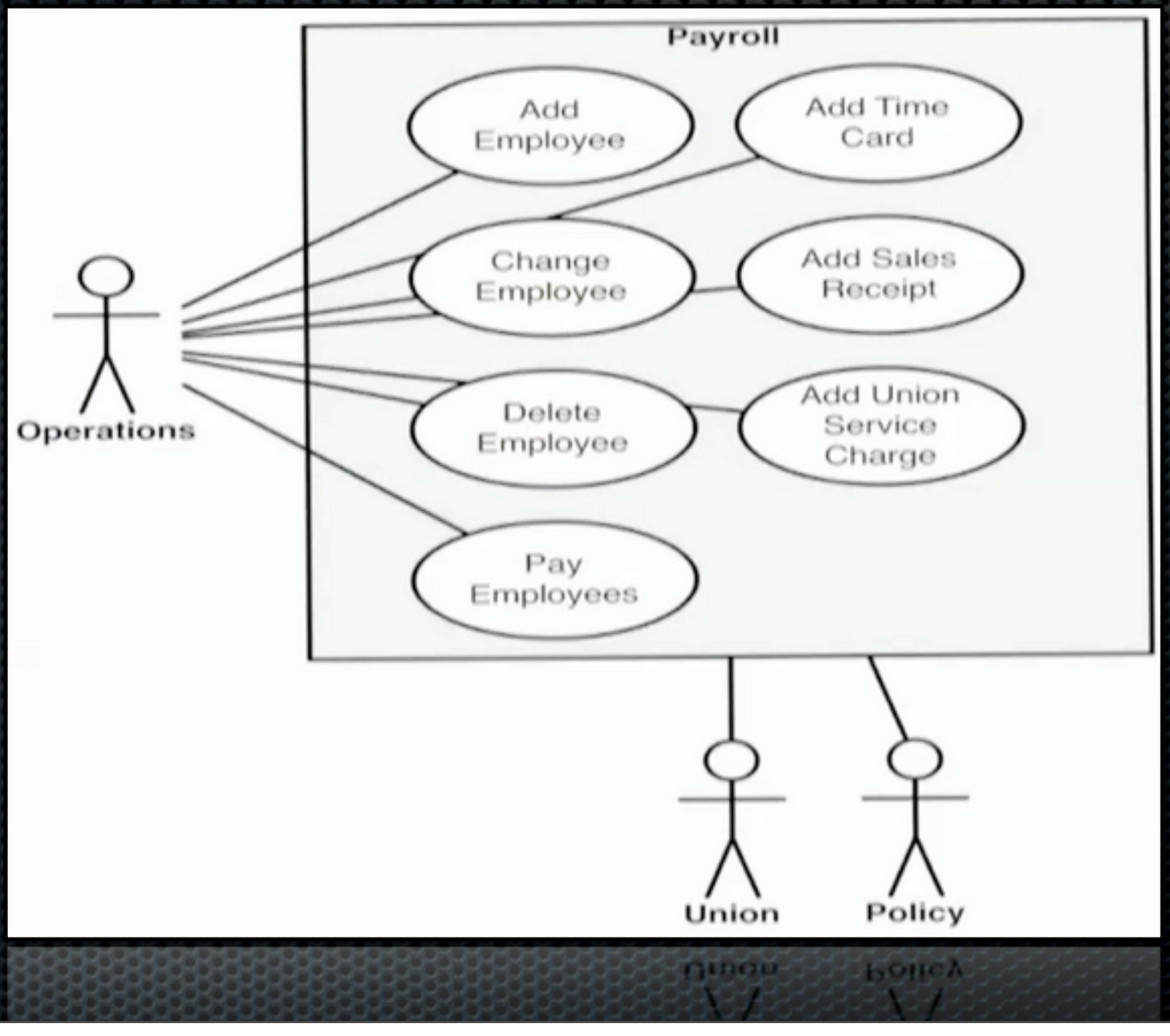
Requirements and Use Cases

- Union Membership 추가
  - –Use Case에 Add Union Service Charge도 추가됨

```
Data Dictionary
    employee = pay-type + pay-disposition + union-membership
        pay-type = {commissioned | hourly | salaried}
            commissioned = base-pay + commission-rate + BI-WEEKLY
            hourly = hourly-rate + WEEKLY
            salaried = salary + MONTHLY
        pay-disposition = {mail | PAYMASTER | direct-deposit}
            mail = address
            direct-deposit = account
    sales-receipt = date + amount-sold
    time-card = date + hours-worked
    union-membership = {MEMBER | NON-MEMBER}
```

The Single Responsibility Principle

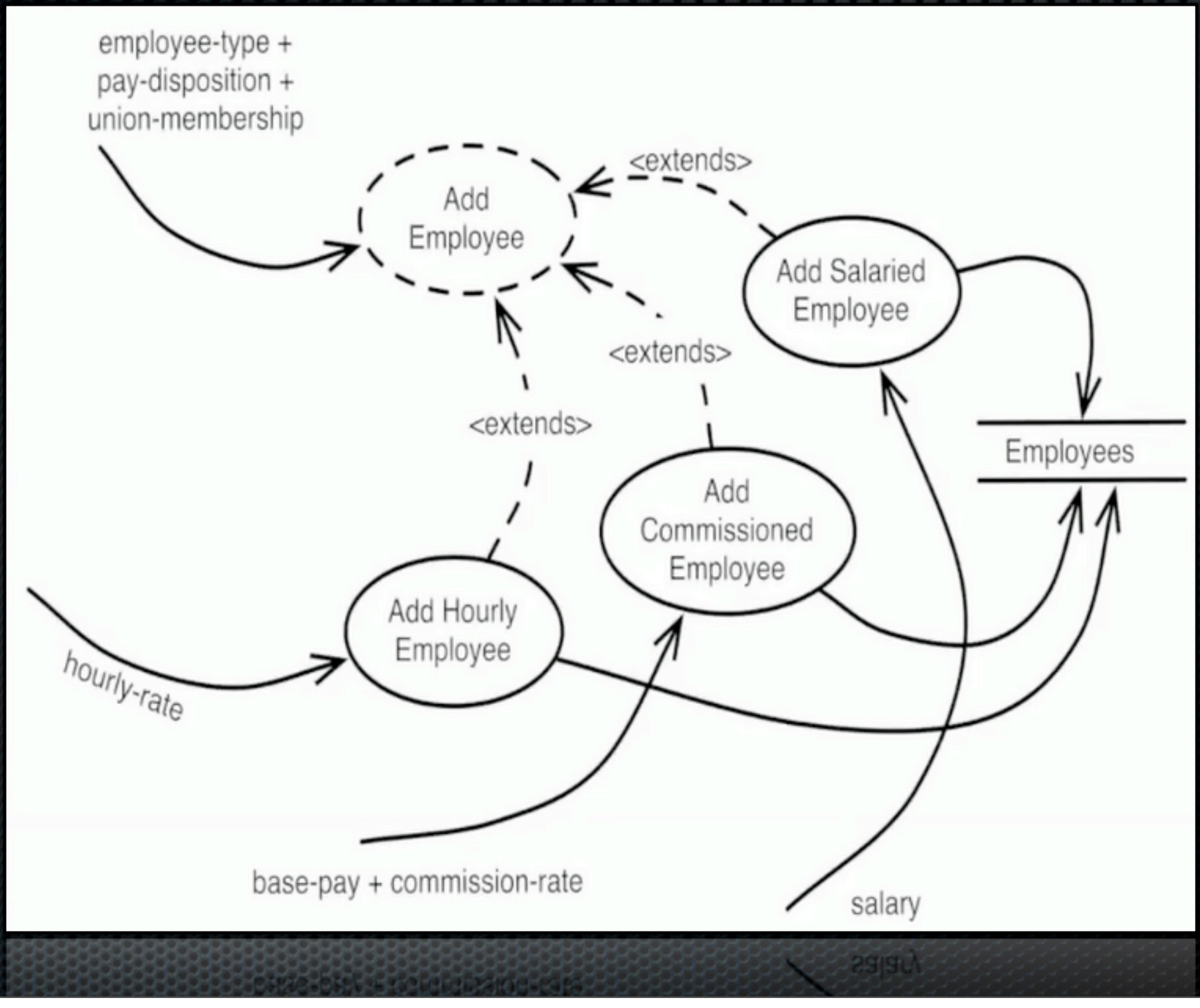
- SRP로 시작해 보자.
- 누가 이 어플리케이션의 액터인가 ?



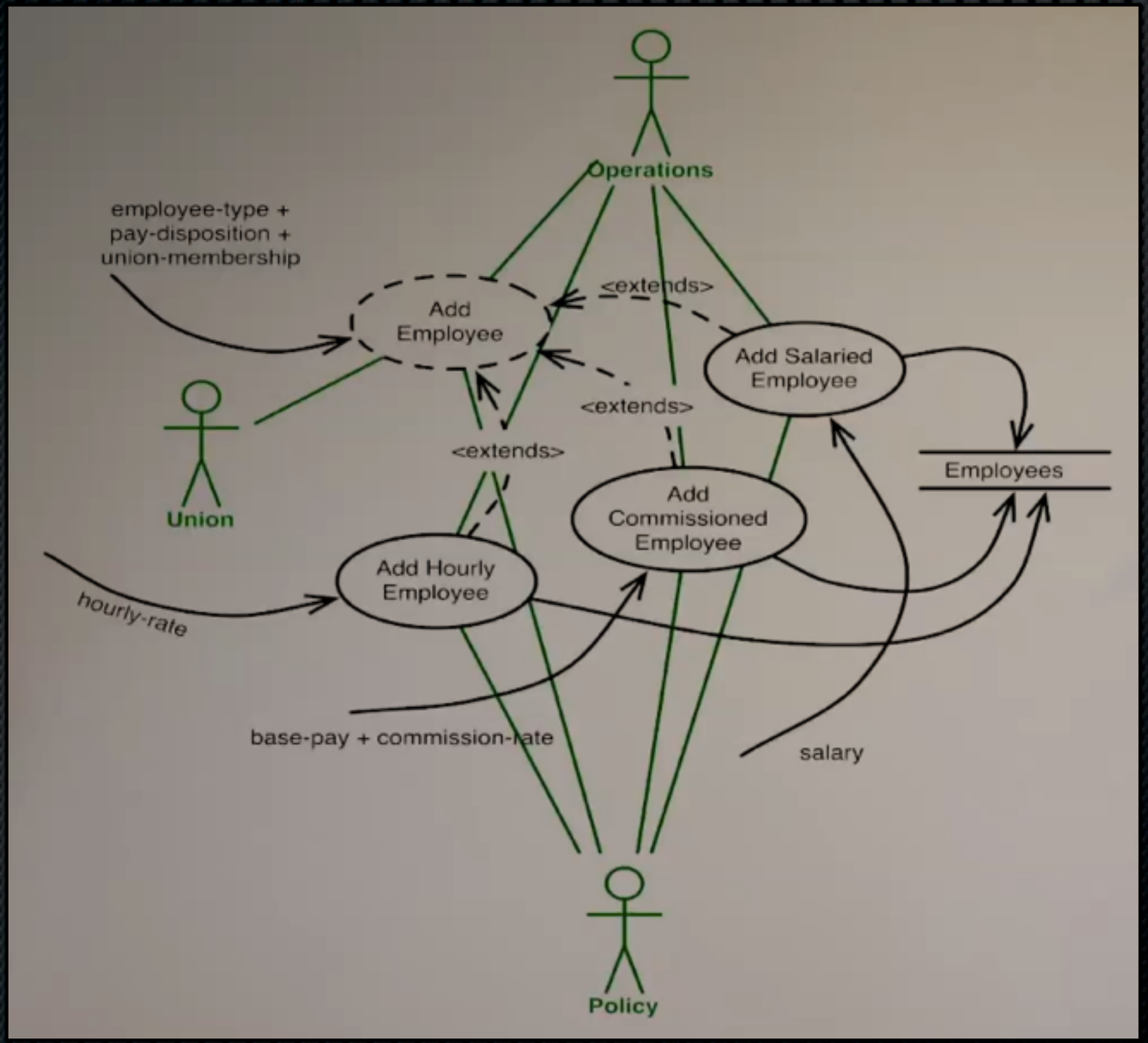
The Single Responsibility Principle

- Our goal here is separate the modules.
- Each module is responsible one and only one actor.
- Tricky한 문제다
  - operations이 사용하는 UI를 가지고 있고
  - 여러 액터들과 연관된 데이터가 존재한다.
- Use Case들을 분해해서 SRP를 준수하는 모듈로 isolate할 수 있는지 살펴보자.

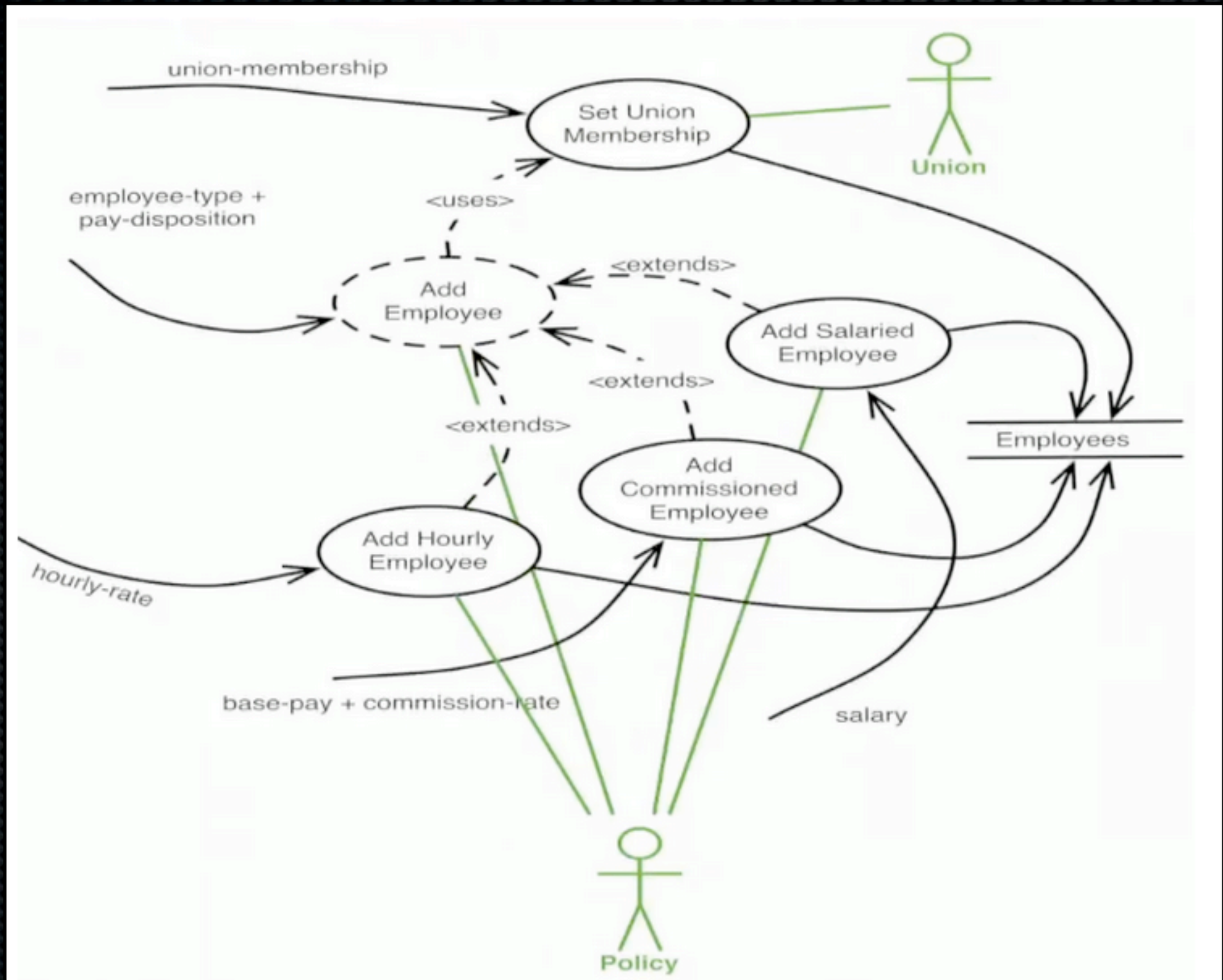
AddEmployee Use Case



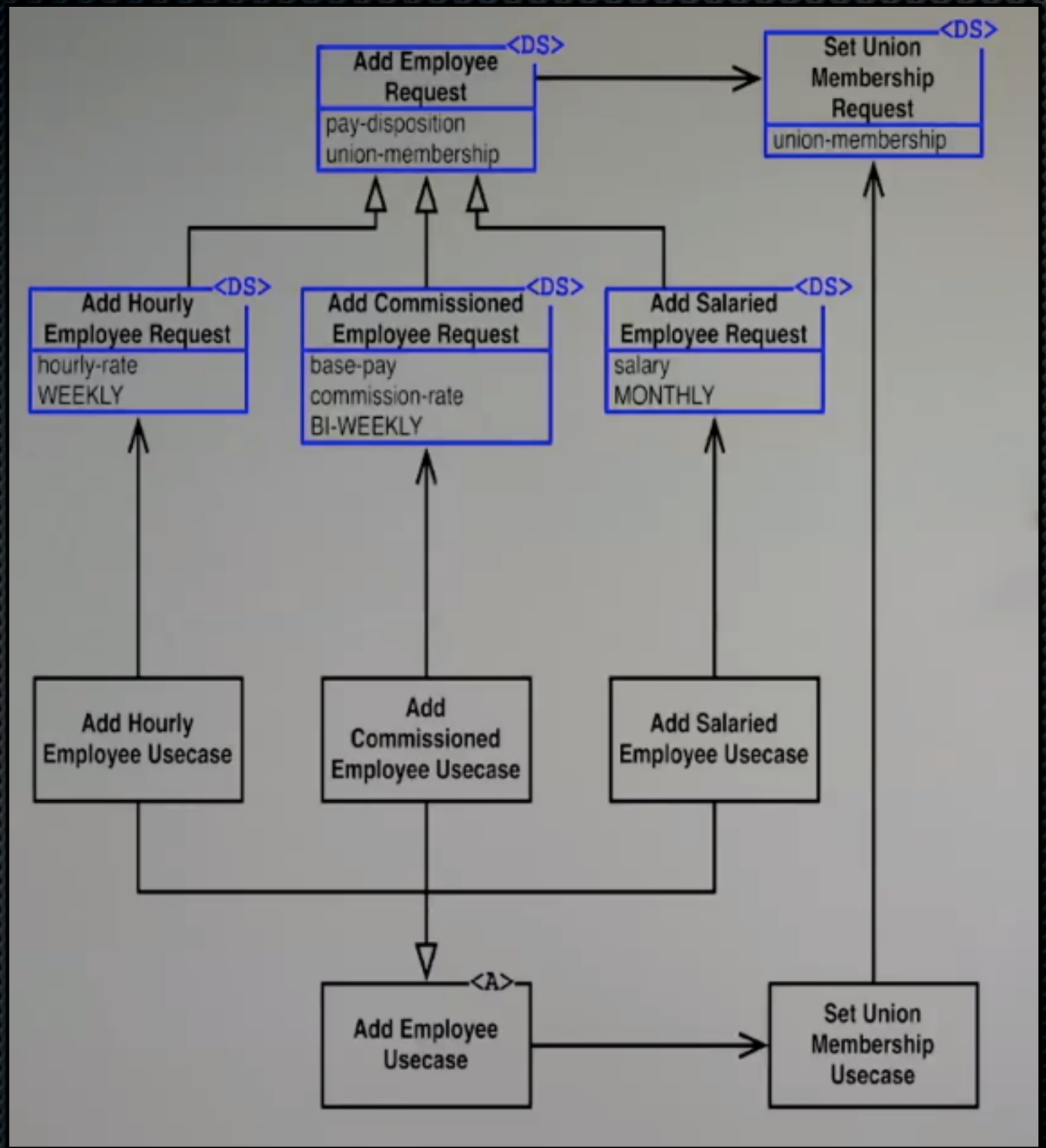
AddEmployee Use Case



AddEmployee Use Case



AddEmployee Use Case



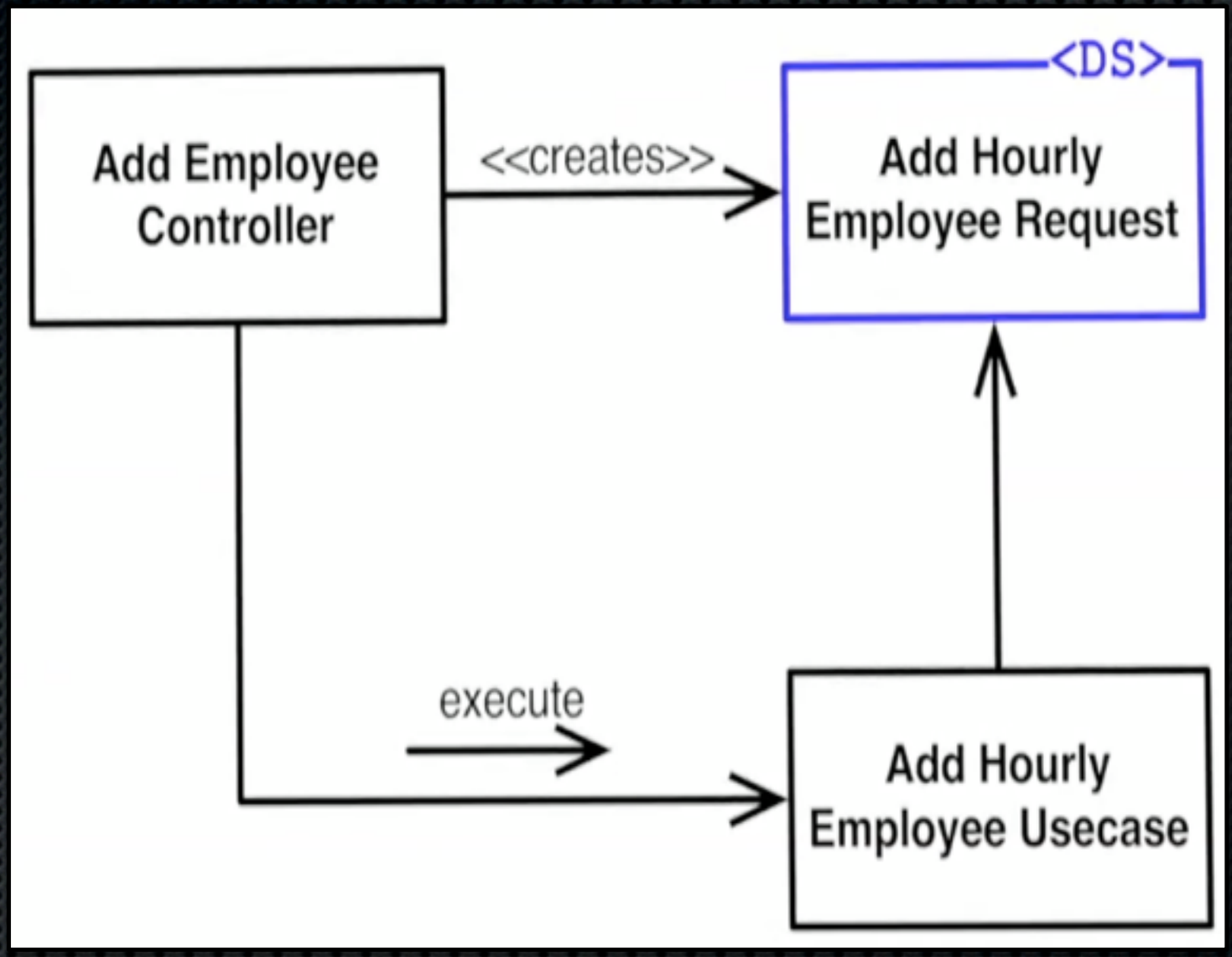
Diagrams and YAGNI

- 다이어그램 작성은 시간 낭비
- 다이어그램을 그리는 이유는 내가 생각하는 프로세스를 설명하기 위함
  - 내가 생각하는 프로세스의 상세함을 다른 사람들에게 전달할 때 작성
- 큰 규모의 프로젝트를 팀원들과 수행한다면 일련의 다이어그램을 화이트보드에 그릴 것
- 다이어그램을 상세하게 작성하는 것이 후에 다른 사람들이 시스템을 이해하는데 도움이 되지 않을까 ?
  - 지속적으로 갱신할 때만 의미를 가짐

SRP의 의미

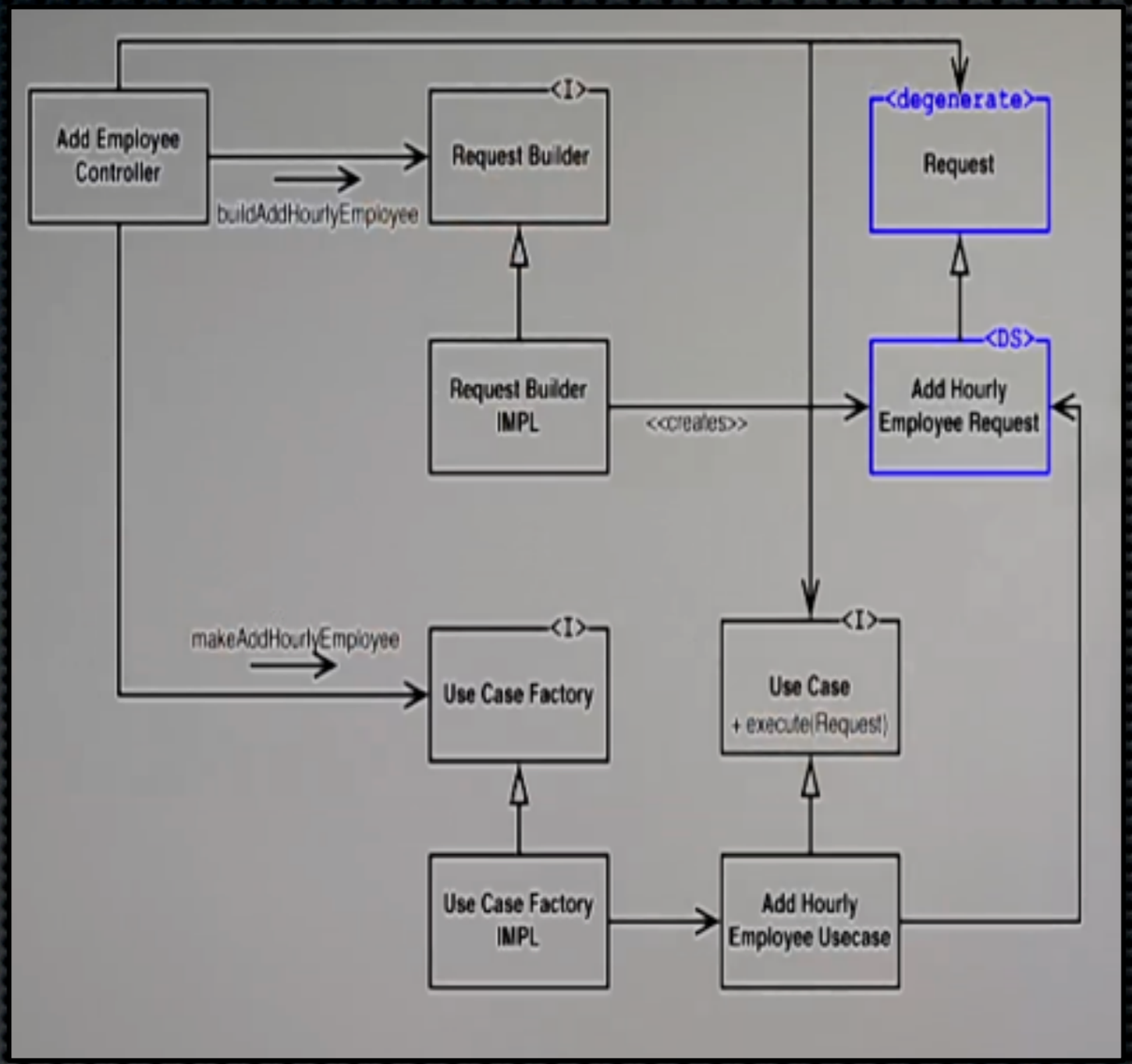
- 한 Module이 하나의 Actor를 위한 Responsibility를 갖도록 분리하는 설계를 하는 것이 의미를 가짐
  - Architectural Framework을 만든 것이다.
- 나는 늘 Actor를 찾고 이에 기반하여 Module을 분리한다.

The Open-Closed Principle



- Controller
  - DS와 UC의 detail에 의존성을 갖는다.
  - DS나 UC의 변화가 생기면 Controller도 재배포되어야

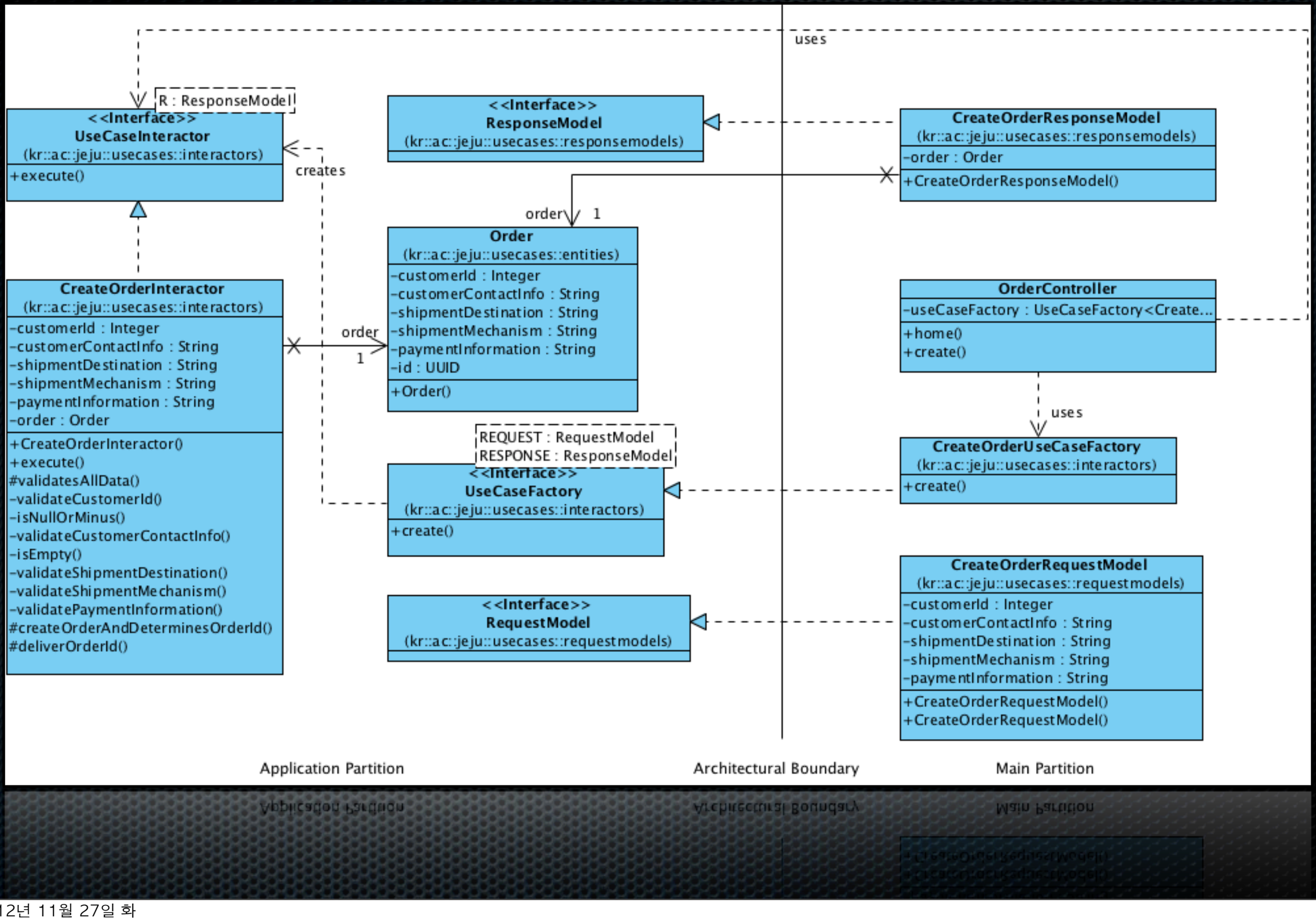
The Open-Closed Principle

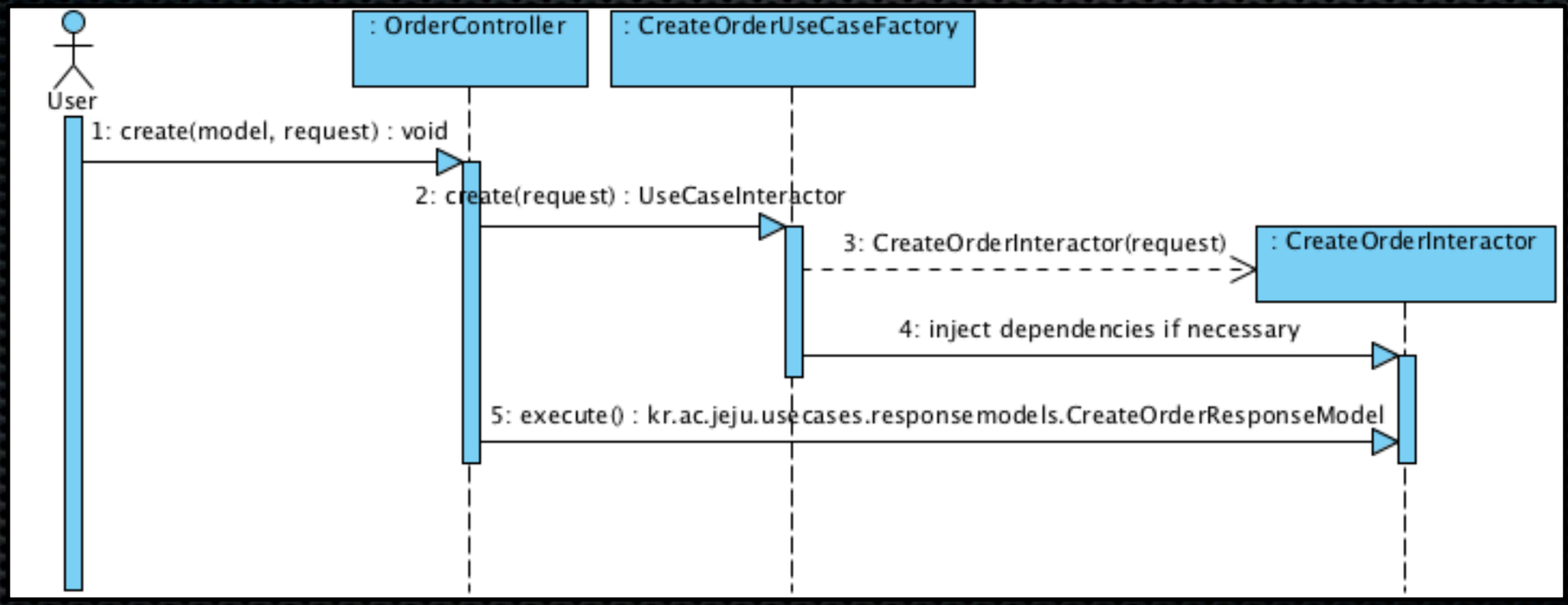


- Solution
  - DS, UC에서 Controller를 decoupling
  - by using Builder, Factory, Interface

Example

- https://github.com/aafwu00/design_by_object/tree/master/cleancode7usecase


uses
R : ResponseModel
<<Interface>>
UseCaseInteractor
(kr::ac::jeju::usecases::interactors)
+execute()
creates
<<Interface>>
ResponseModel
(kr::ac::jeju::usecases::responsemodels)
CreateOrderResponseModel
(kr::ac::jeju::usecases::responsemodels)
-order : Order
+CreateOrderResponseModel()
order
1
Order
(kr::ac::jeju::usecases::entities)
-customerId : Integer
-customerContactInfo : String
-shipmentDestination : String
-shipmentMechanism : String
-paymentInformation : String
-id : UUID
+Order()
CreateOrderInteractor
(kr::ac::jeju::usecases::interactors)
-customerId : Integer
-customerContactInfo : String
-shipmentDestination : String
-shipmentMechanism : String
-paymentInformation : String
-order : Order
+CreateOrderInteractor()
+execute()
#validatesAllData()
-validateCustomerId()
-isNullOrMinus()
-validateCustomerContactInfo()
-isEmpty()
-validateShipmentDestination()
-validateShipmentMechanism()
-validatePaymentInformation()
#createOrderAndDeterminesOrderId()
#deliverOrderId()
order
1
OrderController
-useCaseFactory : UseCaseFactory<Create...
+home()
+create()
uses
REQUEST : RequestModel
RESPONSE : ResponseModel
<<Interface>>
UseCaseFactory
(kr::ac::jeju::usecases::interactors)
+create()
CreateOrderUseCaseFactory
(kr::ac::jeju::usecases::interactors)
+create()
<<Interface>>
RequestModel
(kr::ac::jeju::usecases::requestmodels)
CreateOrderRequestModel
(kr::ac::jeju::usecases::requestmodels)
-customerId : Integer
-customerContactInfo : String
-shipmentDestination : String
-shipmentMechanism : String
-paymentInformation : String
+CreateOrderRequestModel()
+CreateOrderRequestModel()
Application Partition
Architectural Boundary
Main Partition


User
: OrderController
: CreateOrderUseCaseFactory
1: create(model, request) : void
2: create(request) : UseCaseInteractor
3: CreateOrderInteractor(request)
: CreateOrderInteractor
4: inject dependencies if necessary
5: execute() : kr.ac.jeju.usecases.responsemodels.CreateOrderResponseModel

12년 11월 27일 화

생각해 볼 것들

- UseCase Hander
  - Interator
  - Command Pattern vs
  - Domain Object
    - each steps of usecases represented by interface