

SOLID Case Study

Daum Corp.
백명석

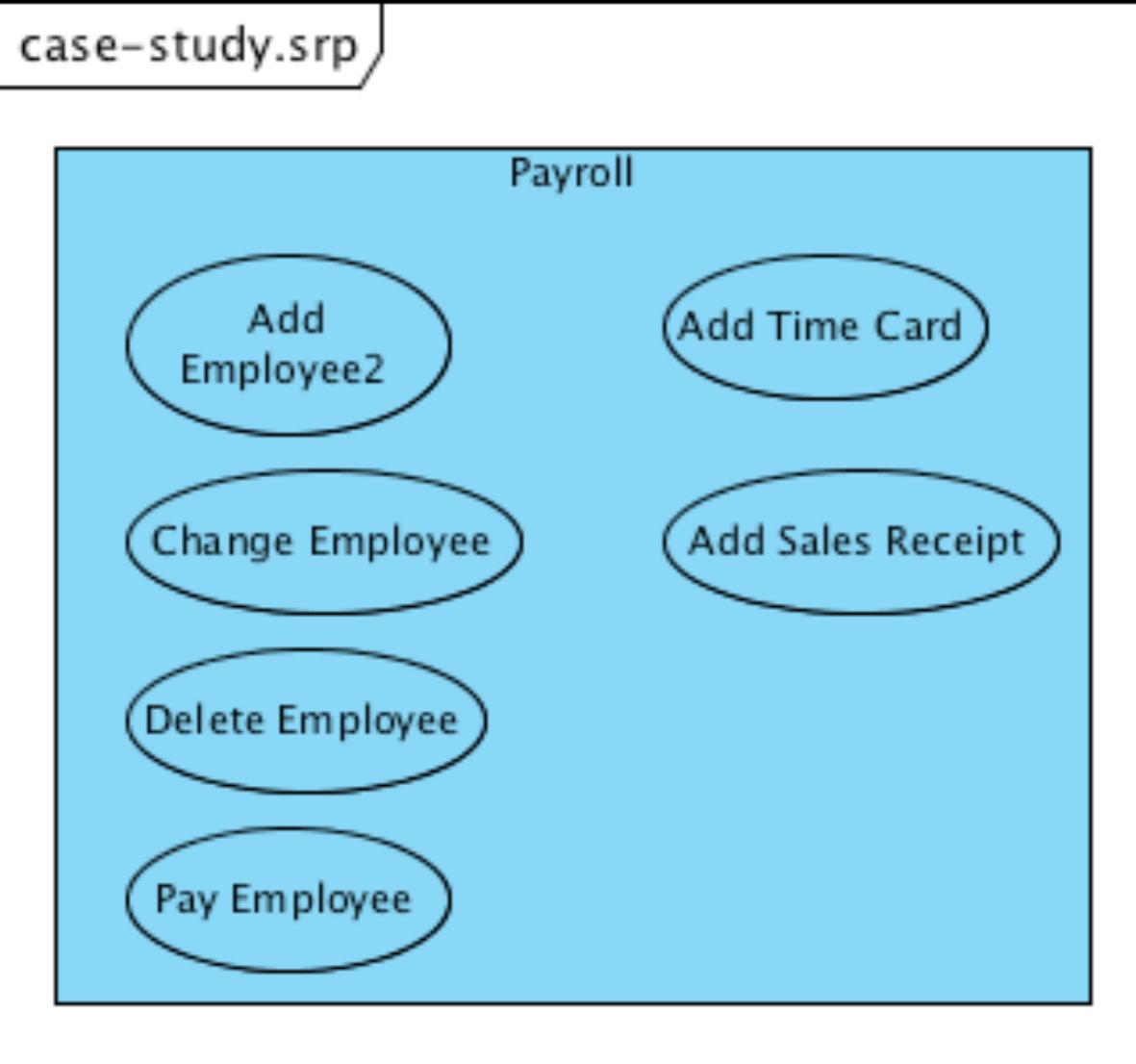
발표목차

- Requirements and Use Cases
 - Use Case List
 - Entity(Data Element) List
- SRP
- OCP
- LSP
- ISP
- DIP
- 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

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

time-card = date + hours-worked

Requirements and Use Cases

- employee type에 따라 pay disposition이 다르다.
- 고객의 요구사항을 들으면서 고객이 사용하는 단어에서 암시하는 데이터를 추출
- 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

time-card = date + hours-worked

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

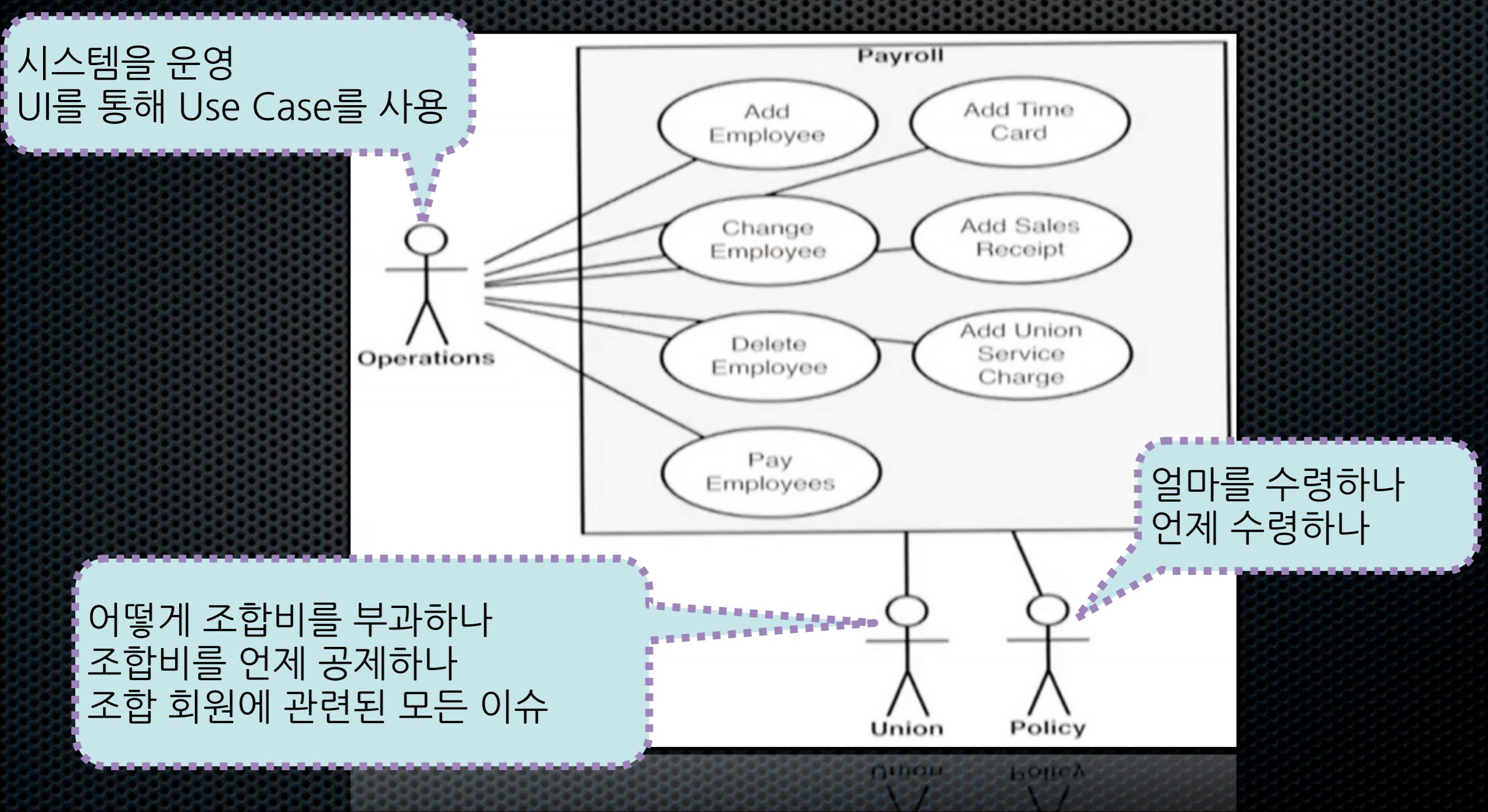
union-membership = {MEMBER | NON-MEMBER}

union-membership = {MEMBER | NON-MEMBER}

union-membership = {MEMBER | NON-MEMBER}

The Single Responsibility Principle

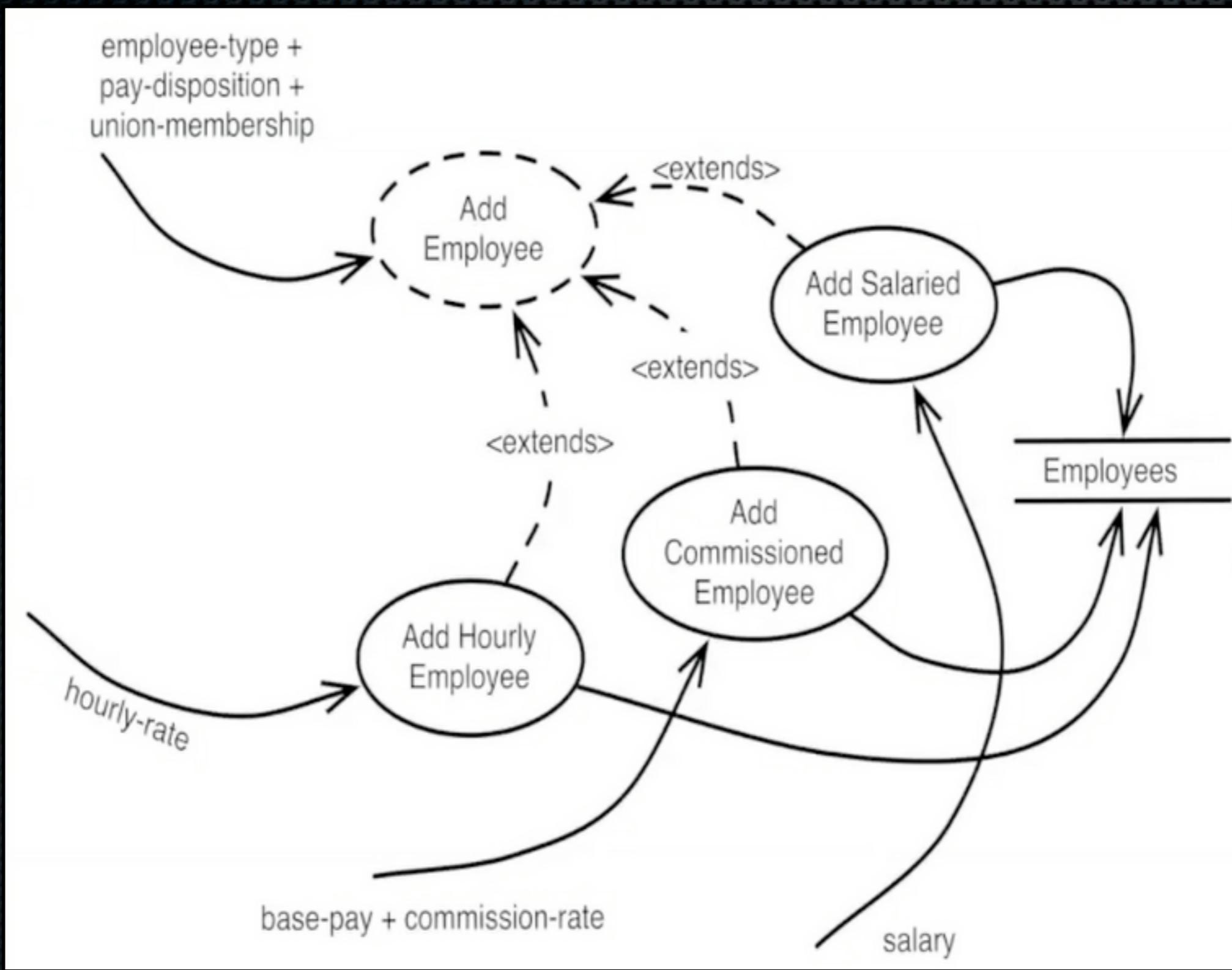
- SRP로 시작
- 누가 어플리케이션의 액터인가 ? 그들의 관심사는 ?



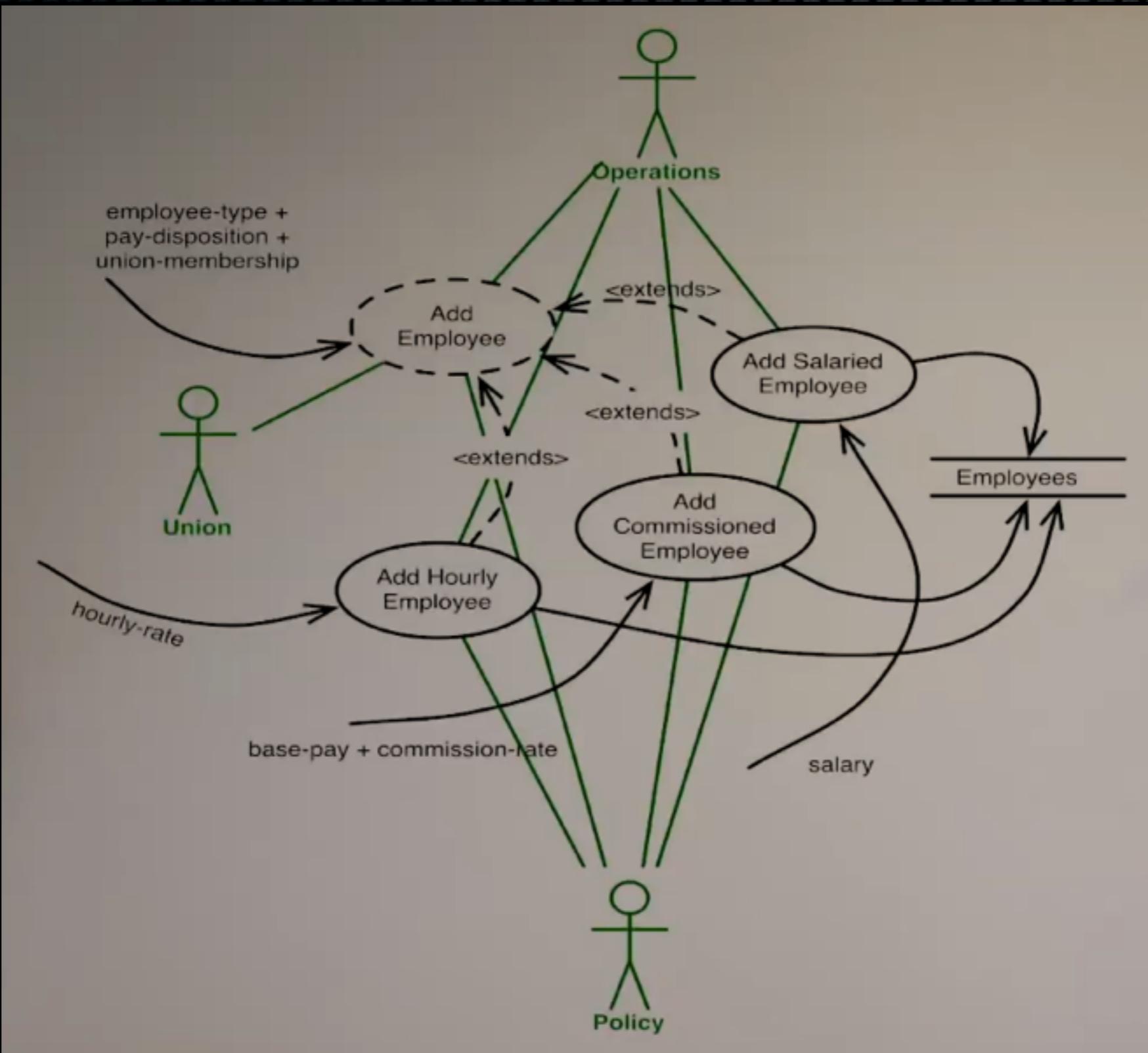
The Single Responsibility Principle

- 이 단계의 목적
 - 모듈들을 분리하는 것
 - 각 모듈은 반드시 하나의 액터만 담당해야 한다.
- 어려운 문제
 - operations은 사용하는 UI를 가지고
 - 여러 액터들과 연관된 데이터가 존재
- Use Case들을 분해해서 SRP를 준수하는 모듈로 isolate

AddEmployee Use Case

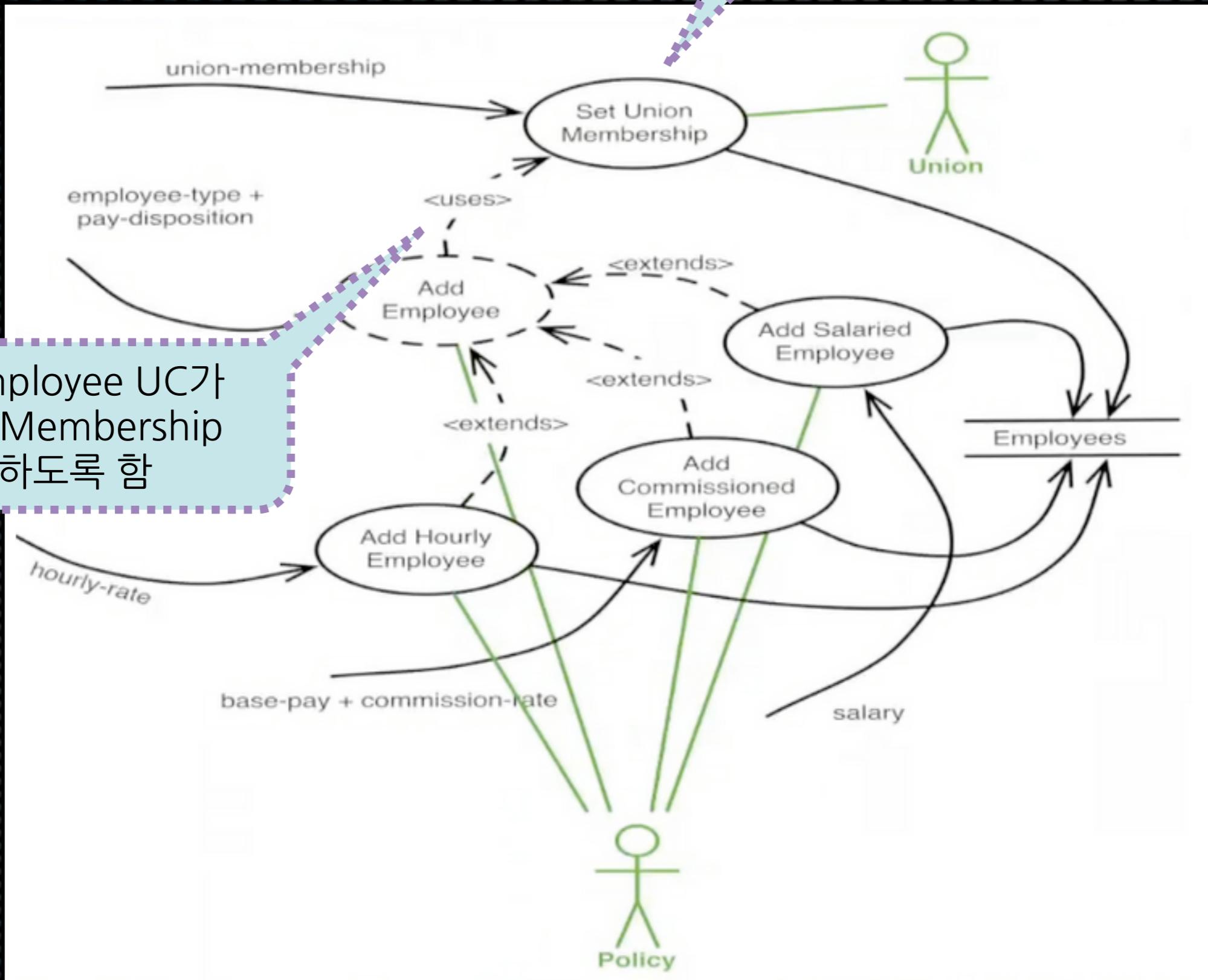


AddEmployee Use Case



AddEmployee Use Case

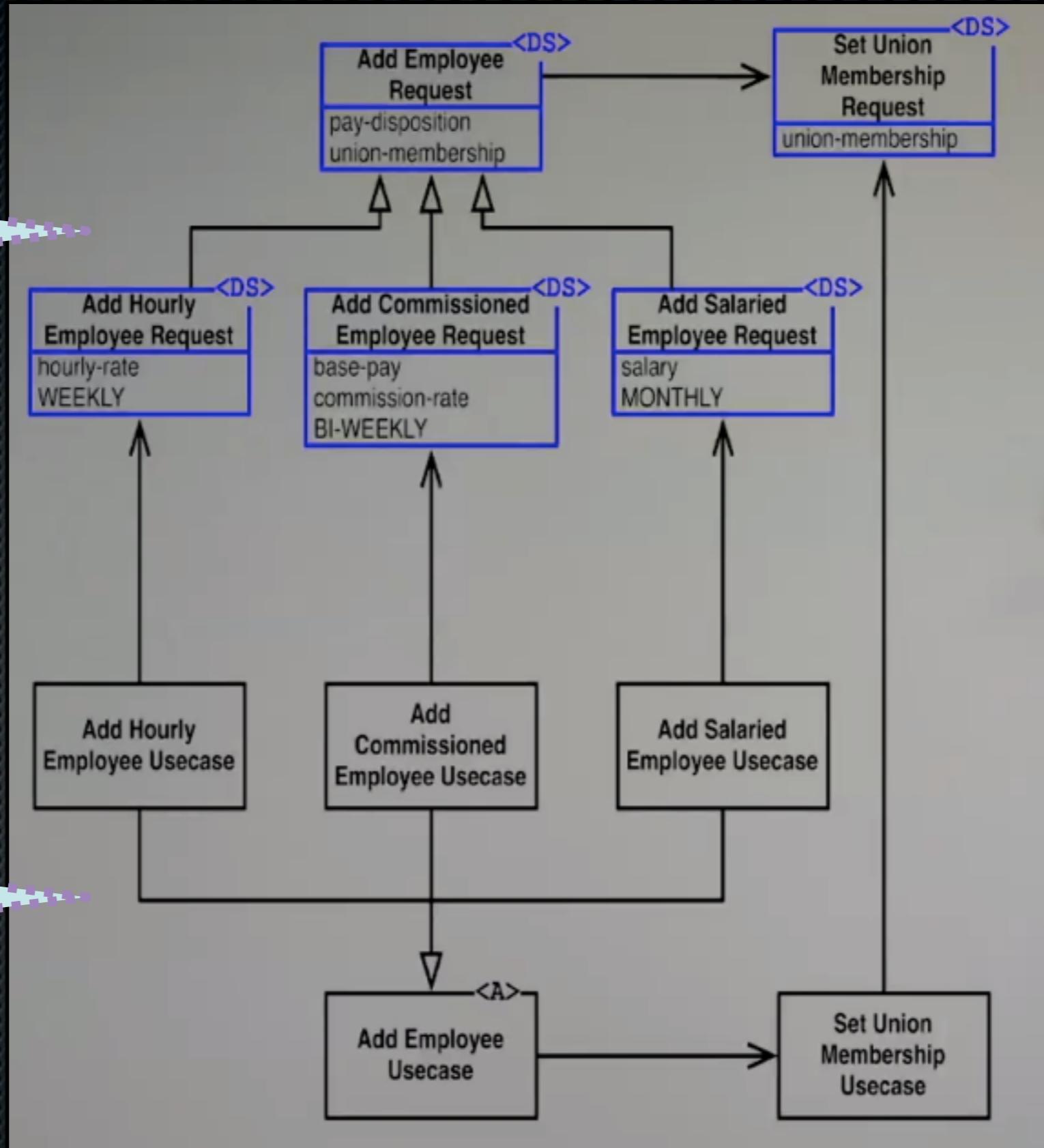
1. Abstract UC에서 새로운 UC를 extracting함



AddEmployee Use Case

UI에 의해 생성되는 Data Structure

UC 오퍼레이션을 구현하는 클래스들



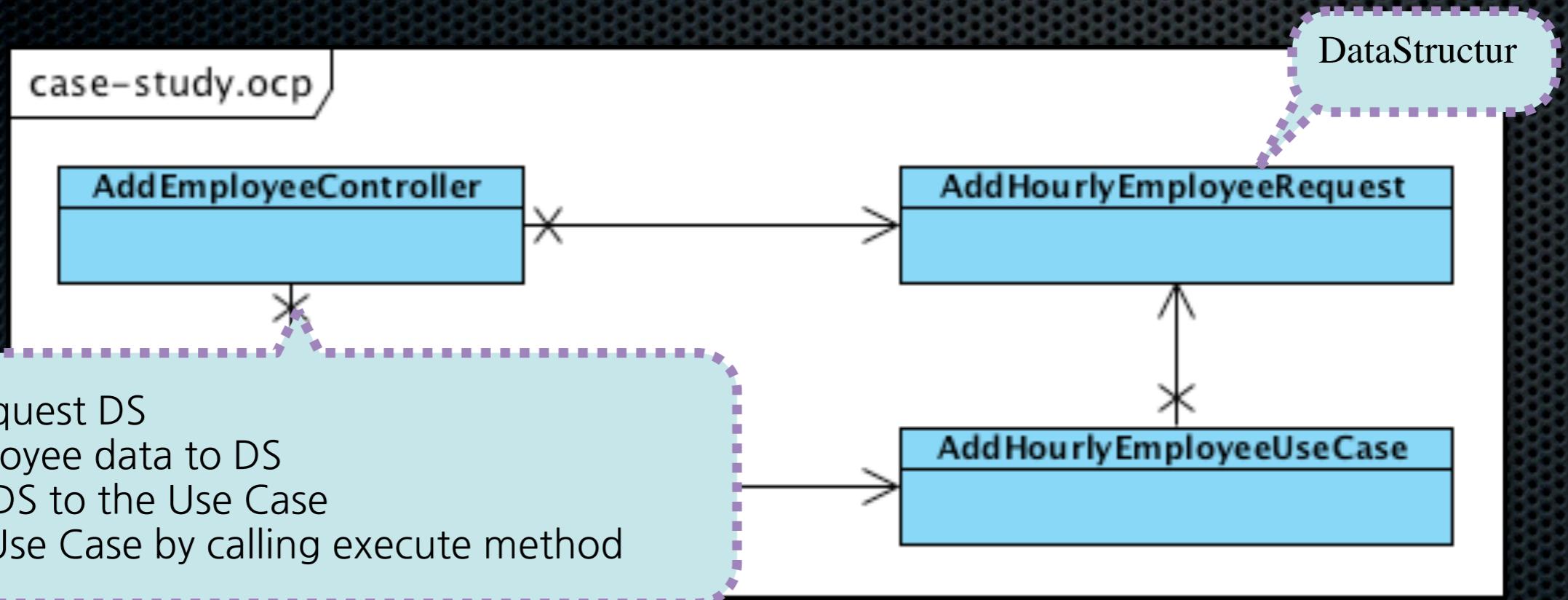
Diagrams and YAGNI

- 다이어그램 작성은 시간 낭비
- 다이어그램 작성 이유 : 생각하는 프로세스 설명
 - 생각하는 프로세스의 상세함을 다른이들에게 전달할 때
- 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

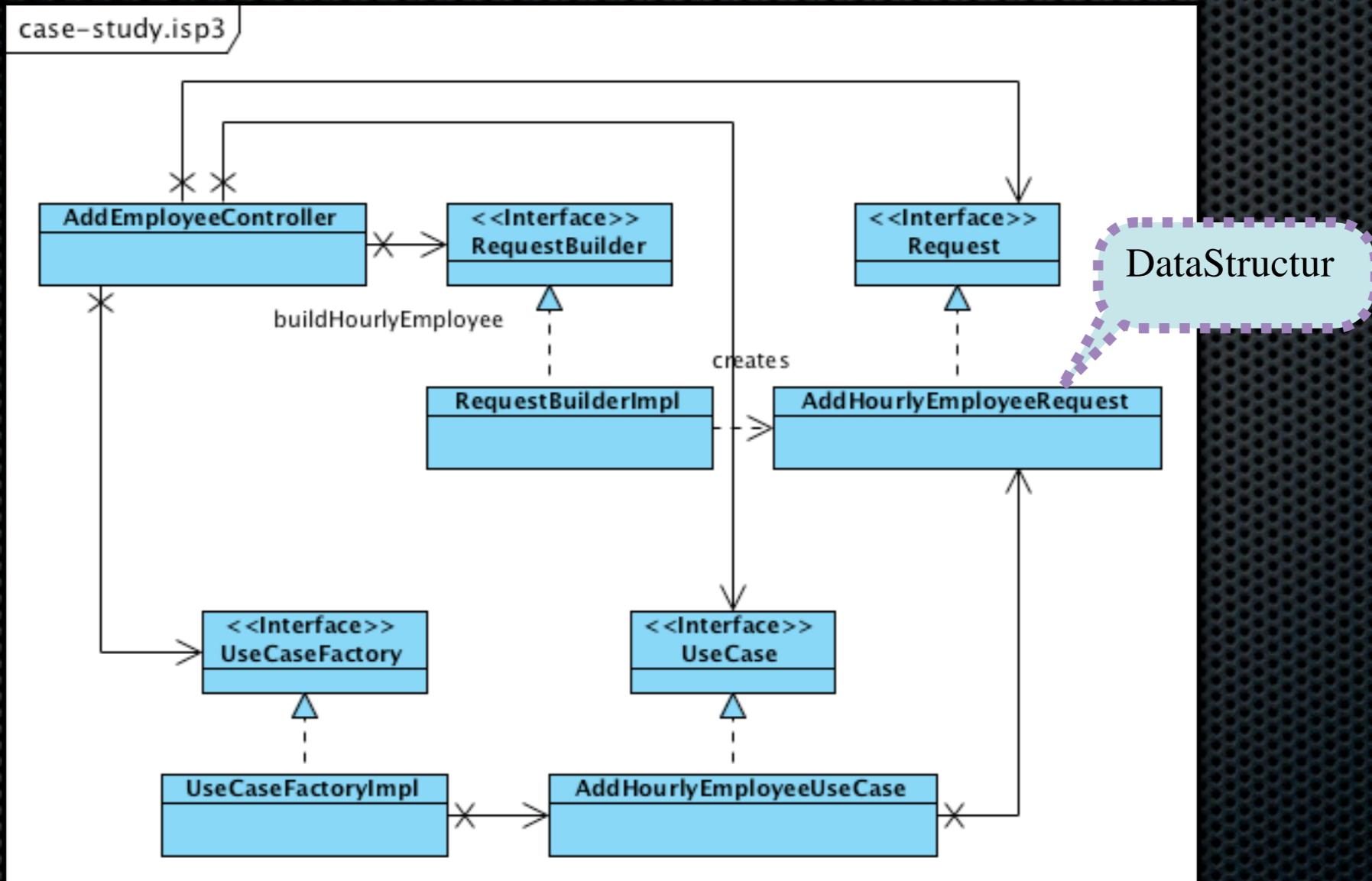
- 한 Module이 하나의 Actor를 위한 Responsibility를 갖도록 분리하는 설계를 하는 것이 의미를 가짐
 - Architectural Framework을 만든 것
- 나는 늘 Actor를 찾고 이에 기반하여 Module을 분리 한다.

The Open-Closed Principle



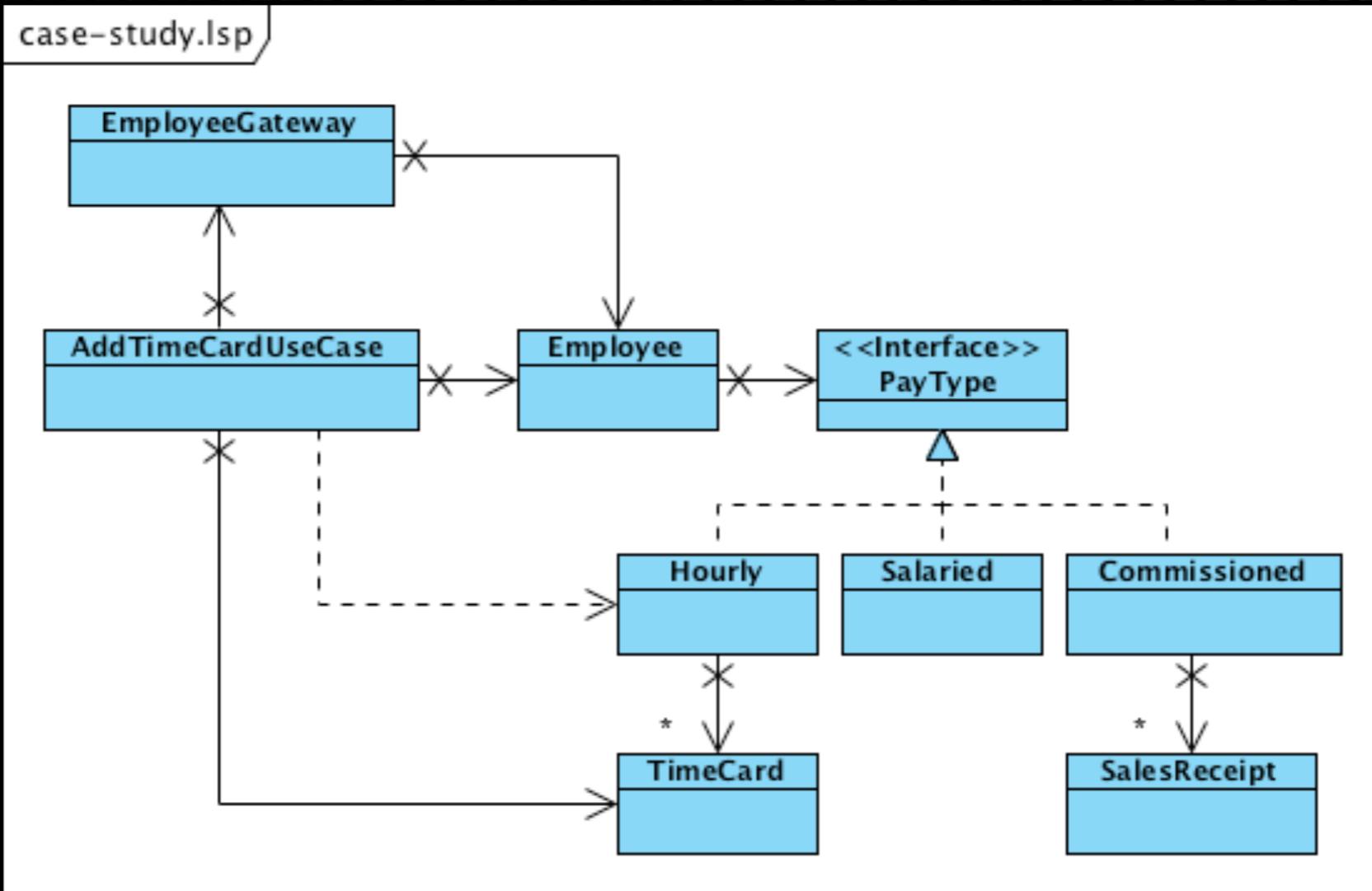
- Controller(UI에 속함)
 - DS와 UC의 detail에 의존성을 갖는다.
 - DS나 UC의 변화가 생기면 Controller도 재배포

The Open-Closed Principle



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

The Liskov Substitution Principle



- AddTimeCardUseCase
 - EmployeeGateway에서 Employee를 fetch
 - TimeCard를 Create
 - TimeCard를 HourlyPayType 객체로 Employee에 추가

The Liskov Substitution Principle

```
public class AddTimeCardUseCase implements UseCase {  
    public void execute(Request request) {  
        AddTimeCardRequest tcReq = (AddTimeCardRequest)request;  
        TimeCard timecard = TimeCard(tcReq.date, tcReq.hours);  
        Employee e = employeeGateway.findEmployee(tcReq.employeeId);  
        e.addTimeCard(timecard);  
    }  
}
```

- 문제점
 - Employee 클래스가 TimeCard를 알아야 함
 - OCP 위반
 - 새로운 지불방법이 생길 때마다 Employee 클래스에 addXXX 메소드가 추가되어야 함.

The Liskov Substitution Principle

- addTimeCard 메소드를 PayType에 추가하면

```
public class AddTimeCardUseCase implements UseCase {  
    public void execute(Request request) {  
        AddTimeCardRequest tcReq = (AddTimeCardRequest) request;  
        TimeCard timecard = TimeCard(tcReq.date, tcReq.hours);  
        Employee e = employeeGateway.findEmployee(tcReq.employeeId);  
        e.getPayType().addTimeCard(timecard);  
    }  
}  
  
public interface PayType {  
    public int calculatePay();  
    public void addTimeCard(TimeCard tc);  
}
```

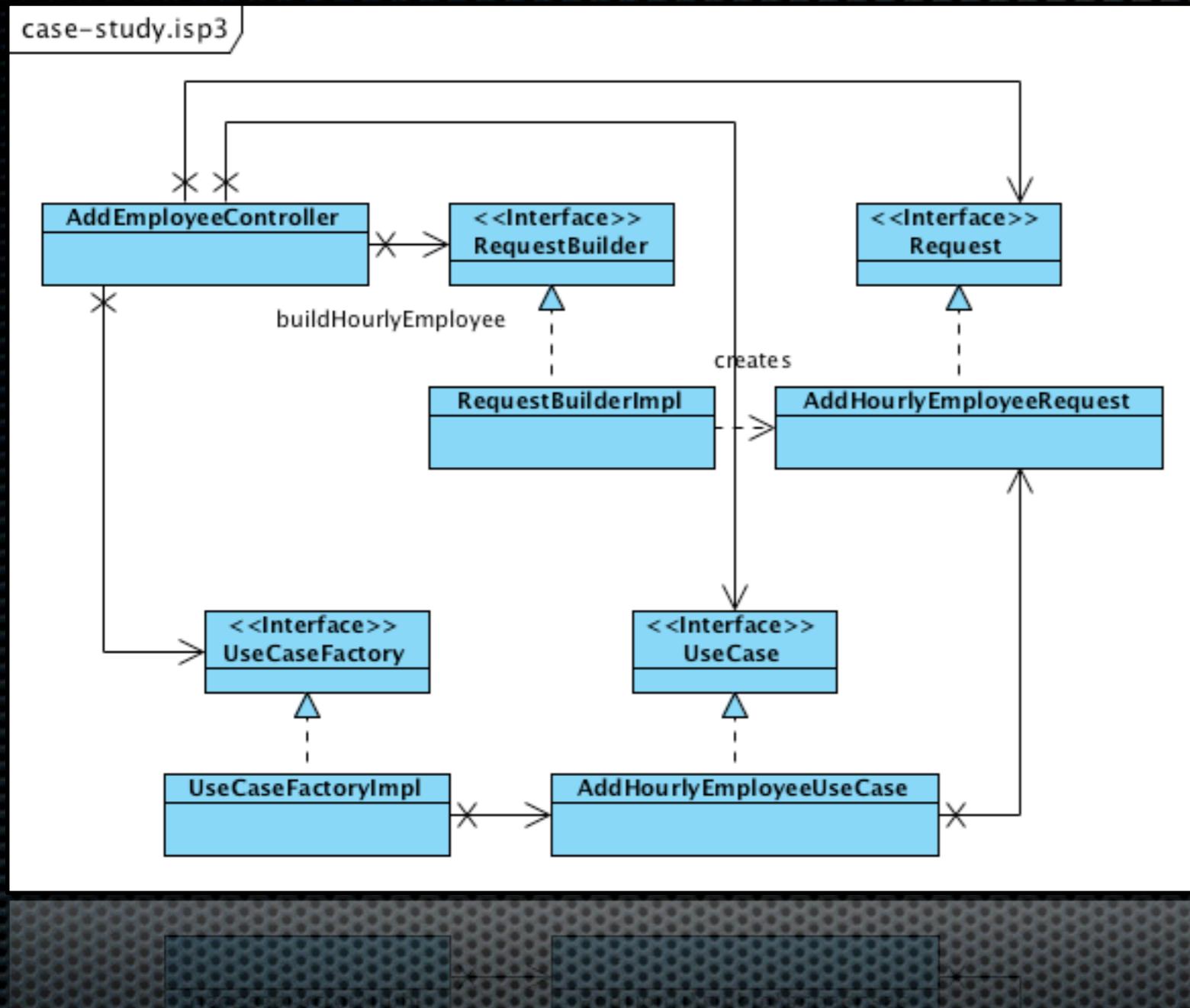
- OCP 위반을 해소
- Hourly는 addTimeCard에 의미있는 구현 제공
- Salary, Commissioned는 ignore하도록
 - 아무것도 안하도록 override하는 것은 LSP 위반
 - Interface에 모든 파생클래스에 적용할 수 없는 메소드를 추가하는 것은 LSP 위반

The Liskov Substitution Principle

```
public class AddTimeCardUseCase implements UseCase {  
    public void execute(Request request) {  
        AddTimeCardRequest tcReq = (AddTimeCardRequest)request;  
        TimeCard timecard = TimeCard(tcReq.date, tcReq.hours);  
        Employee e = employeeGateway.findEmployee(tcReq.employeeId);  
        Hourly hourly = (Hourly)e.getPayType();  
        hourly.addTimeCard(timecard);  
    }  
}
```

- 위 코드의 문제
 - down cast
 - LSP 준수 위해 down cast를 하는 것이 일반적
 - 주어진 파생클래스가 어떤 타입인지 정확히 안다면 down cast로 인한 피해는 없다.

The Interface Segregation Principle

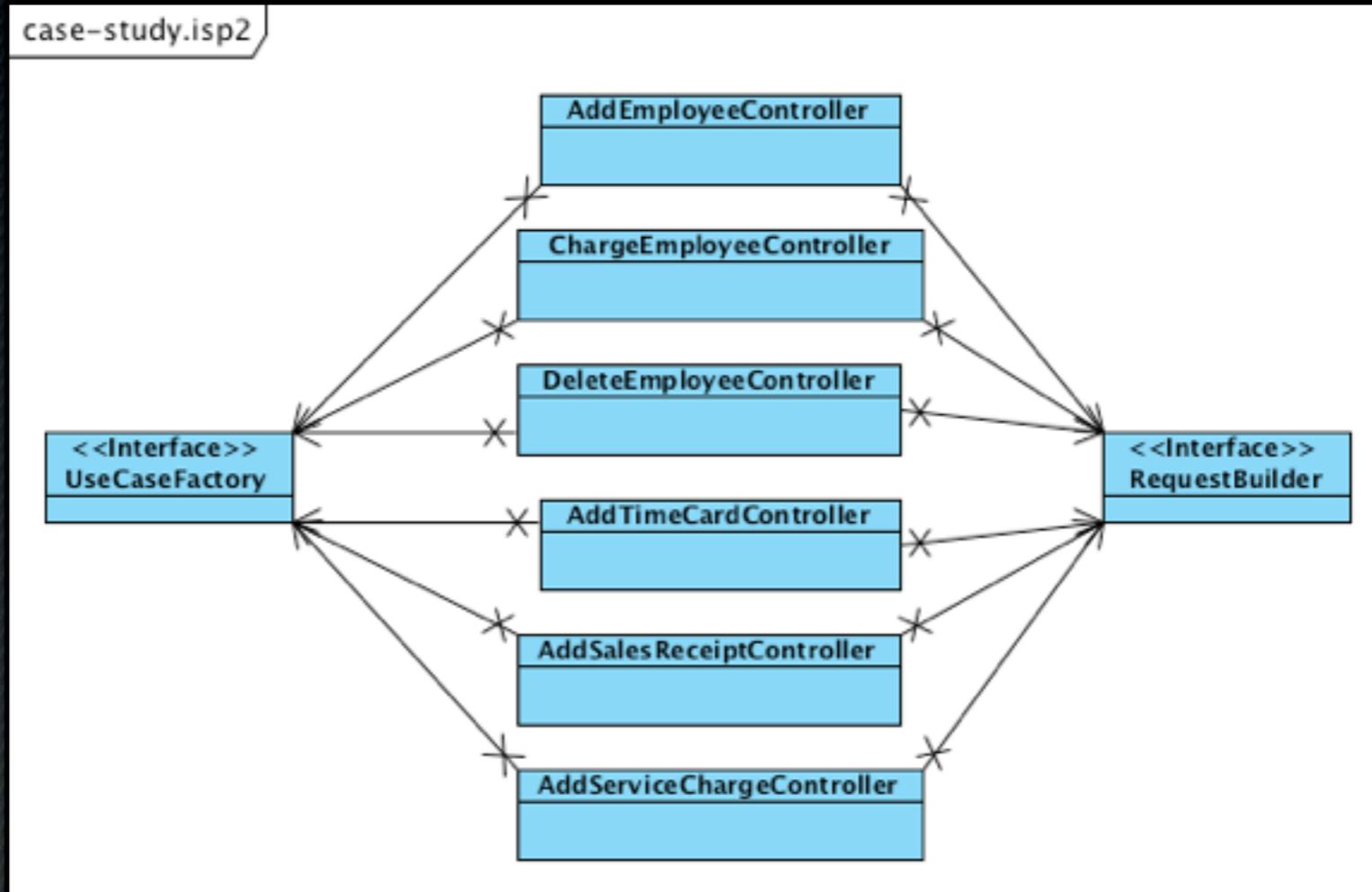


- OCP 준수 but ISP 위반
 - RequestBuilder
 - UseCaseFactory
 - 개별 DS/UseCase 생성을 위한 메소드

AddEmployeeController - It knows too much

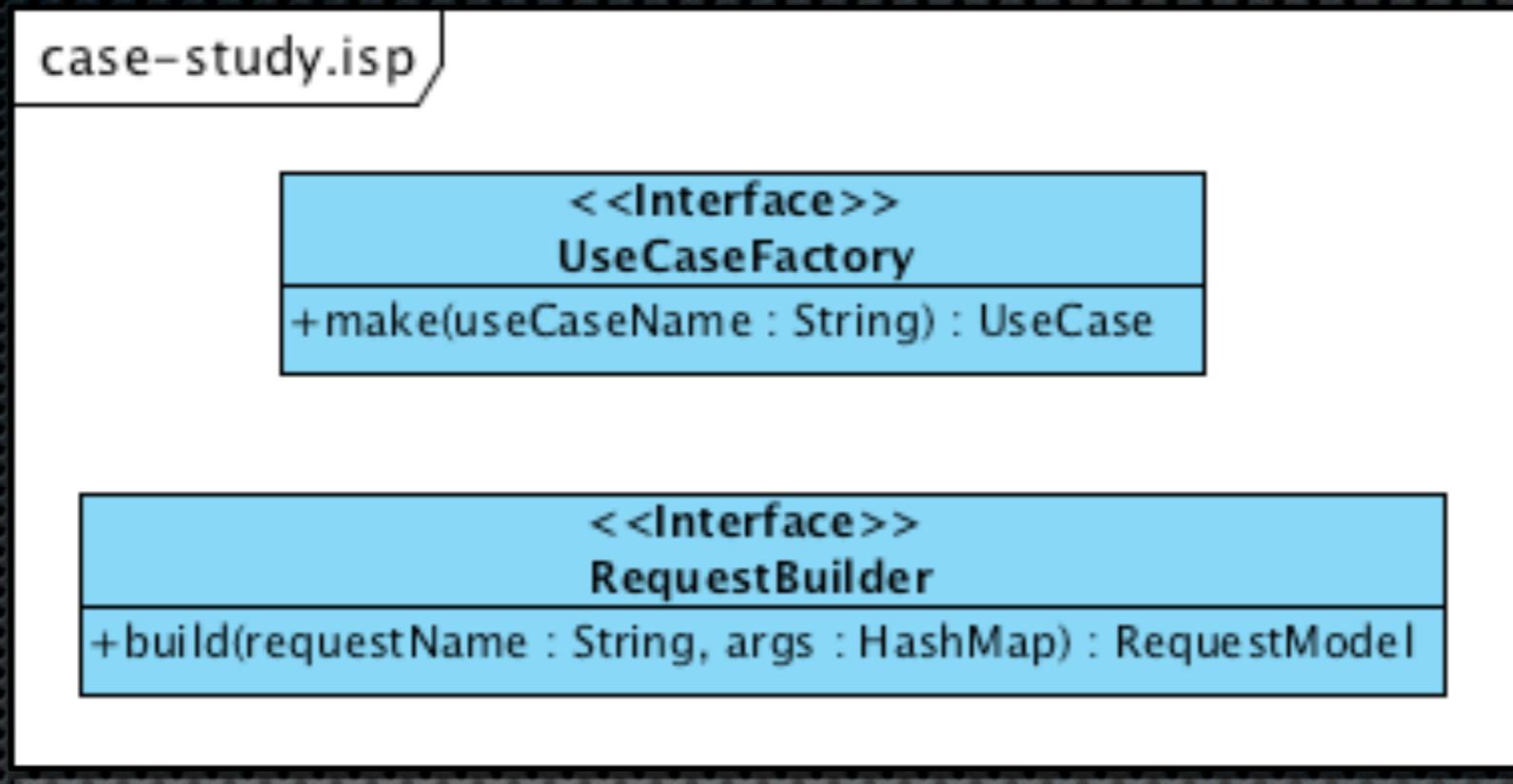
- 자신이 호출하지 않는 메소드에도 의존성을 가짐.

The Interface Segregation Principle



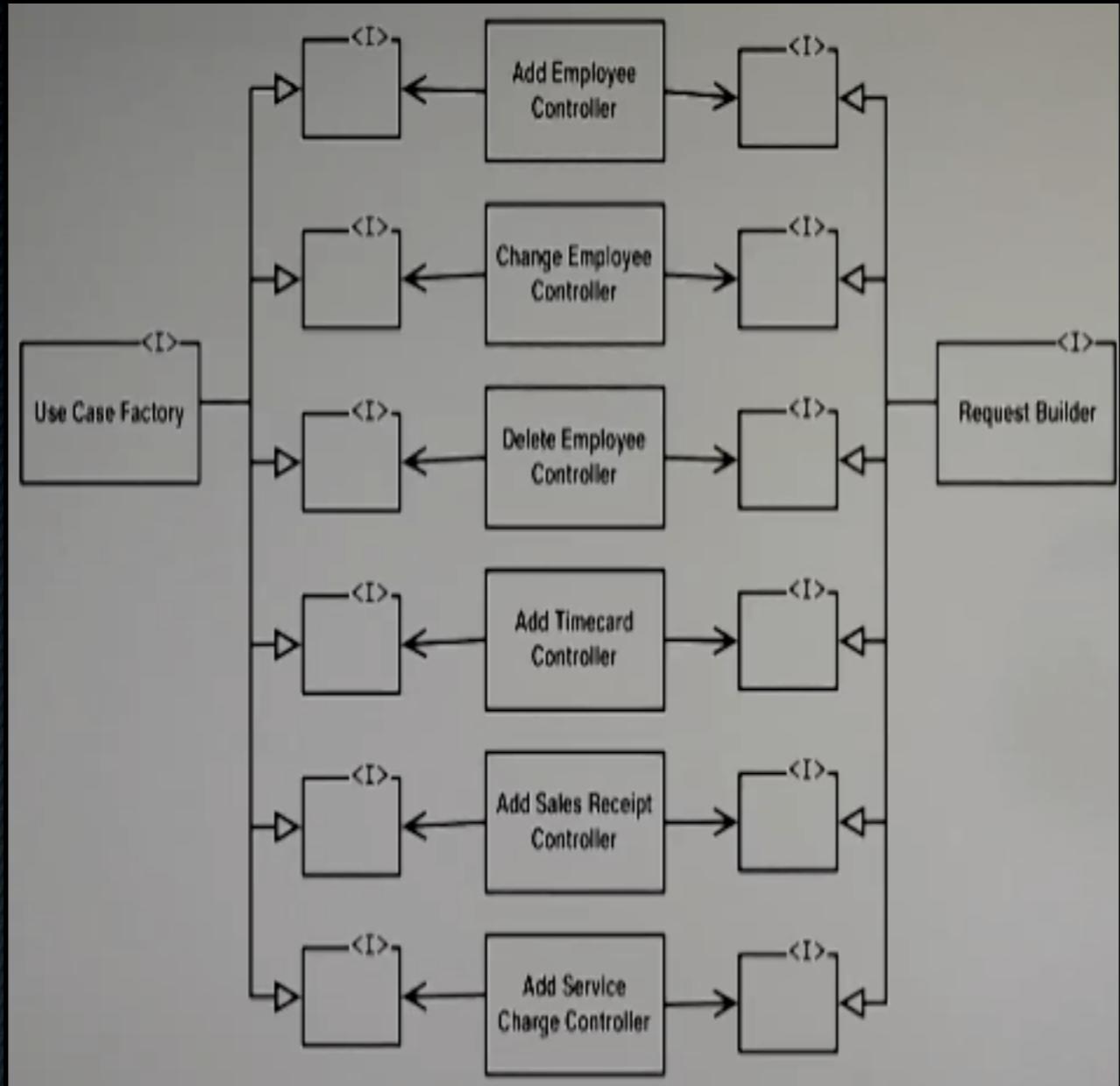
- Controller가 많아지면
 - Fanning이 늘어남
 - Interface의 변경이 생길 때마다 모든 Controller를 recompile/redeploy해야 한다.

The Interface Segregation Principle



- Request Builder와 Use Case Factory에 대한 Dynamic Interface로도 해소 가능
 - Type Safety를 잃음

The Interface Segregation Principle



- 이게 **Messy**해 보이나 ?
- 별로 그렇지 않다.
- 이러한 Interface들은 매우 작고,
- 적은 수의 메소드들만을 갖는다.

- Type Safety 이슈 해소

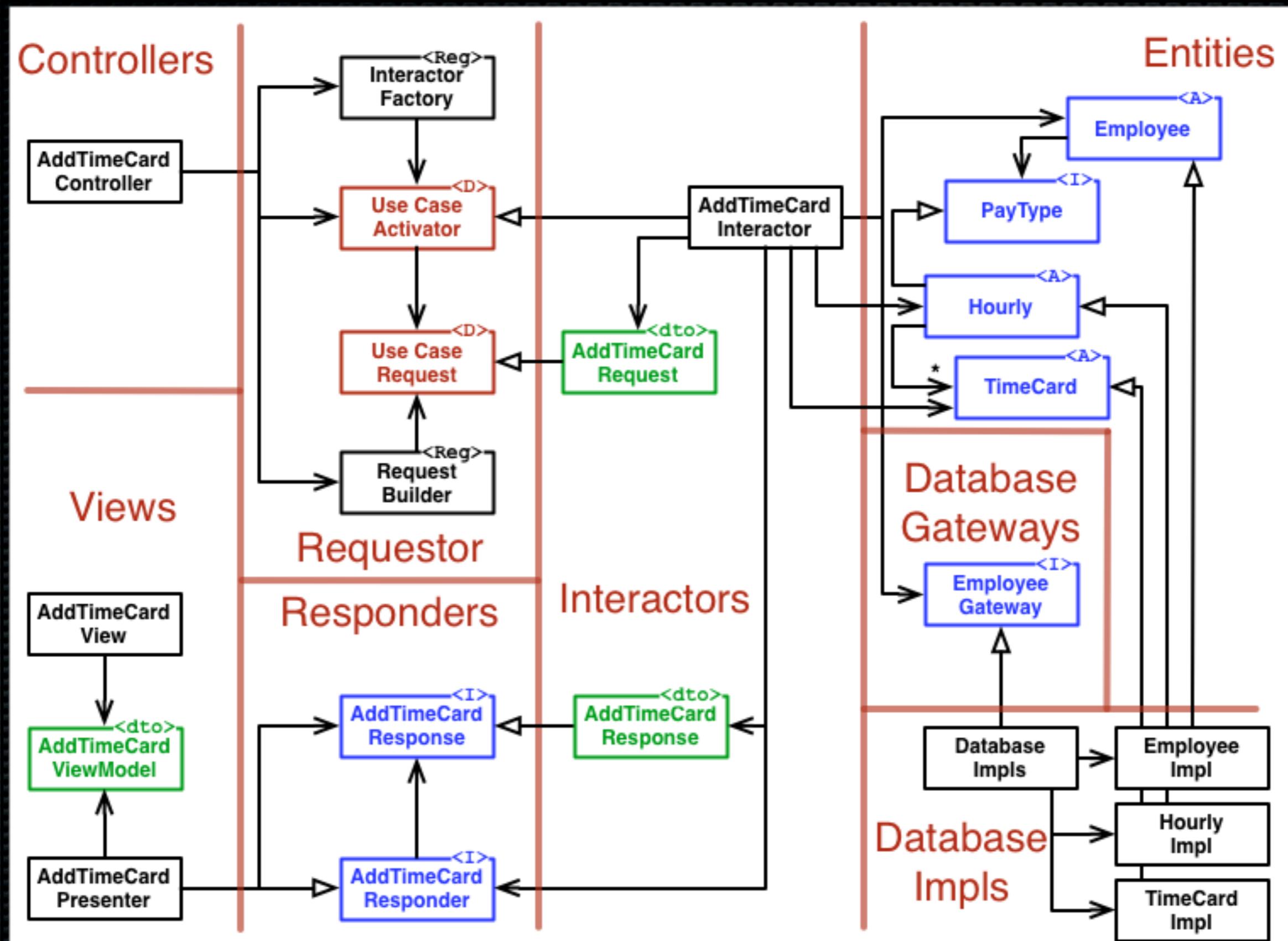
- 특정 Controller가 사용하는 메소드만을 갖는 많은 Interface를 추가

The Dependency Inversion Principle

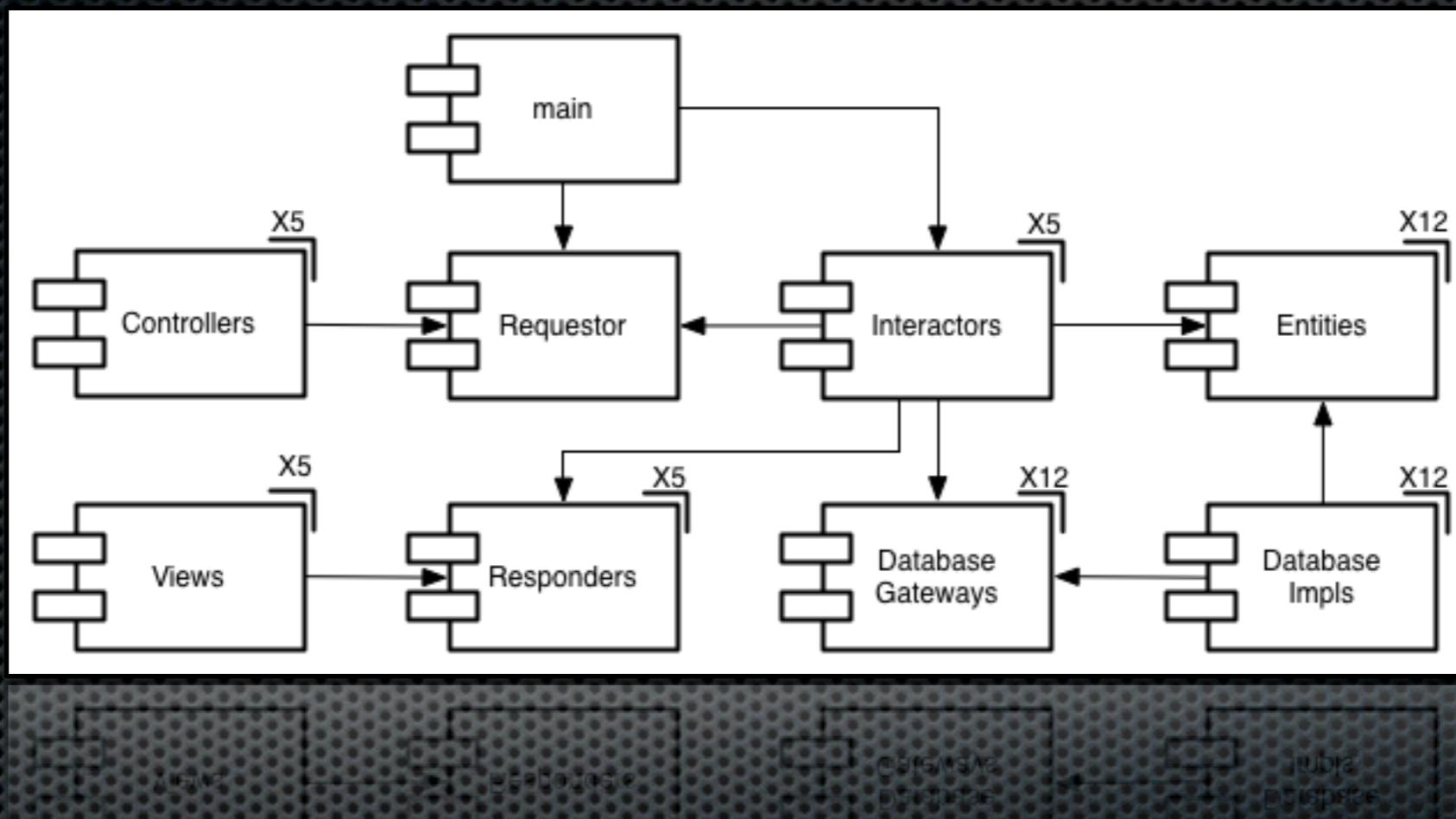
- High Level policy should be independent of low level detail

```
public class PayEmployee implements UseCase {  
    ...  
    public void execute(Request request) {  
        PayEmployeeRequest payReq = (PayEmployeeRequest)request;  
        Date date = payReq.date;  
        List<Employee> employees = employeeGateway.findAll();  
        for(Employee e : employees) {  
            if(e.getPaySchedule().shouldPay(date)) {  
                int pay = e.calculatePay(date);  
                pay -= e.getDues(date);  
                pay -= e.getTotalCharges(date);  
                e.pay(pay);  
            }  
        }  
    }  
}
```

E18. Component Case Study



E18. Component Case Study



More

- Episode 15 - SOLID Components
- Episode 16 - Component Cohesion
- Episode 17 - Component Coupling
- Episode 18 - Component Case Study
- Episode 19 - Advanced TDD 1
- Episode 19 - Advanced TDD 2
- Episode 20 - Clean Tests
- Episode 21 - Test Design Study
- ...