

Single Responsibility Principle

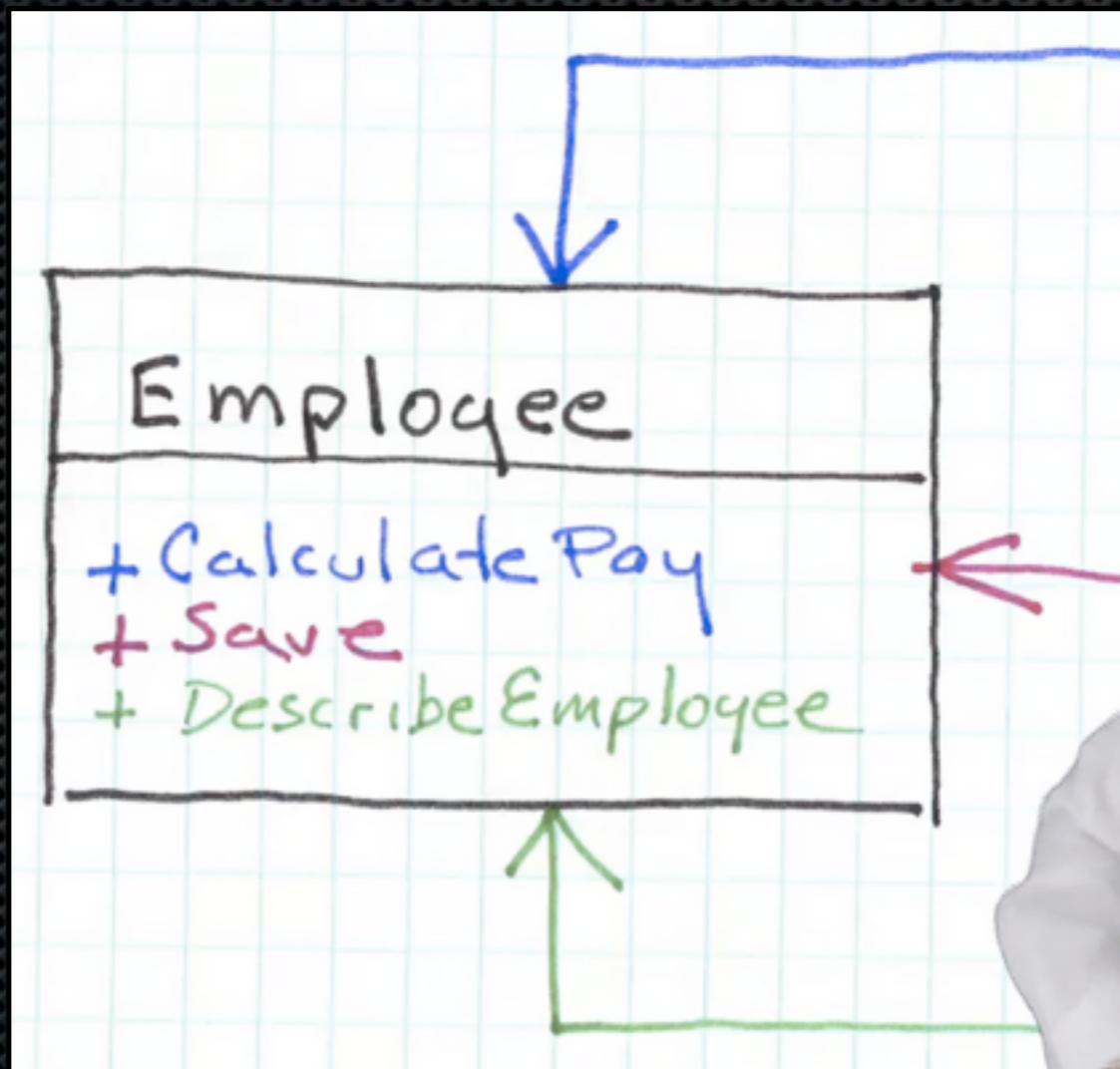
Daum Corp.
백명석

발표목차

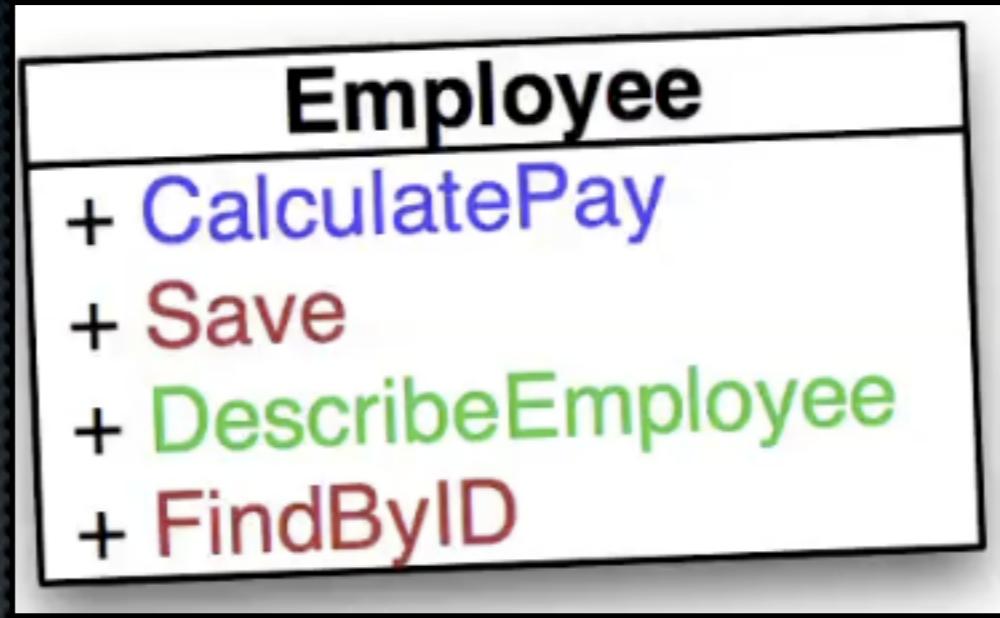
- Responsibility
 - Users
 - Roles
 - Collision
 - Fan Out
 - Collocation is Coupling
- SRP
- Solution
- Case Study
- Faking It

Responsibility

- 클래스는 하나의 책임을 가져야 한다.
- 책임은 무엇인가 ?



Responsibility



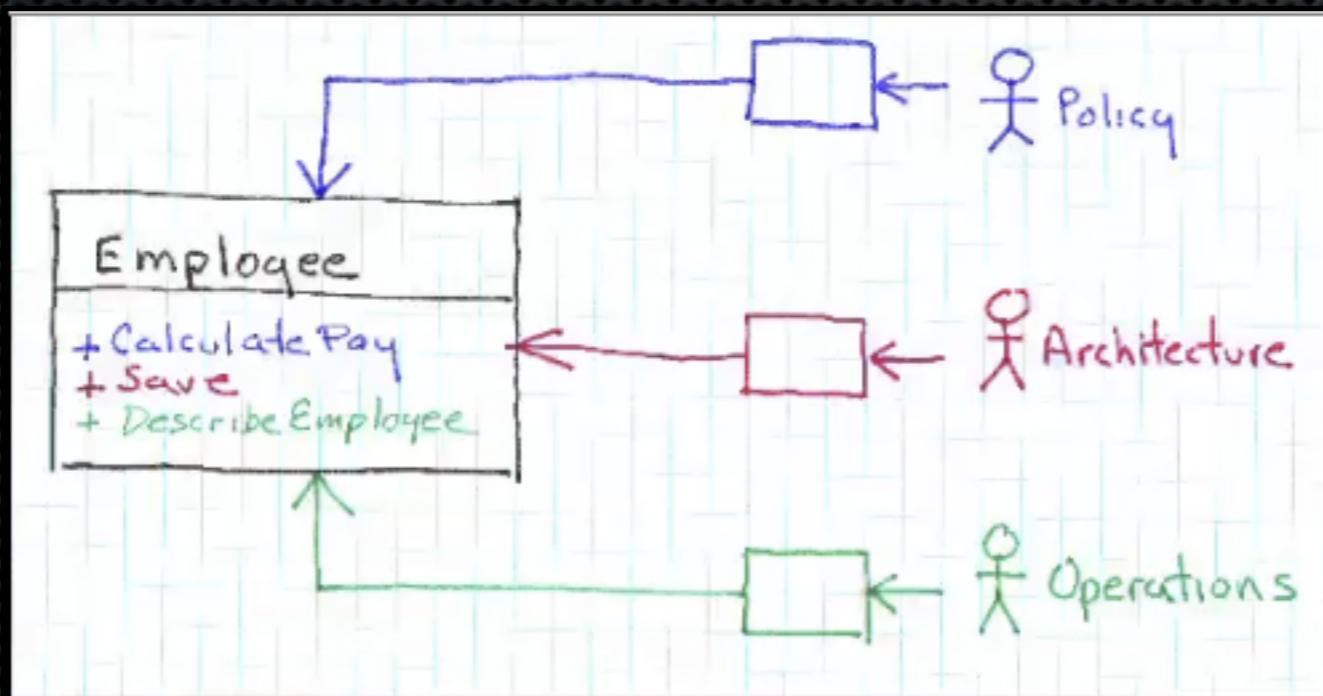
- Save/FindByID - 같은 부류
- 같은 책임을 갖는 기능 - CalculatePay
 - CalculateDeduction, CalculateSalary
 - 추가를 해도 책임의 수는 변하지 않음
- 부류
 - 메소드의 client에 의해 결정
 - 누가 해당 메소드의 변경을 유발하는 사용자인가

It's About Users

- SRP는 사용자에 관한 것
- 책임
 - SW의 변경을 요청하는 특정 사용자들에 대해 클래스/함수가 갖는 것
 - “**변경의 근원**”으로 볼 수 있음

It's About Users

- Employee 클래스의 변경을 요구하는 사용자들
 - Actor: 서로 다른 Needs, Expectation을 가짐



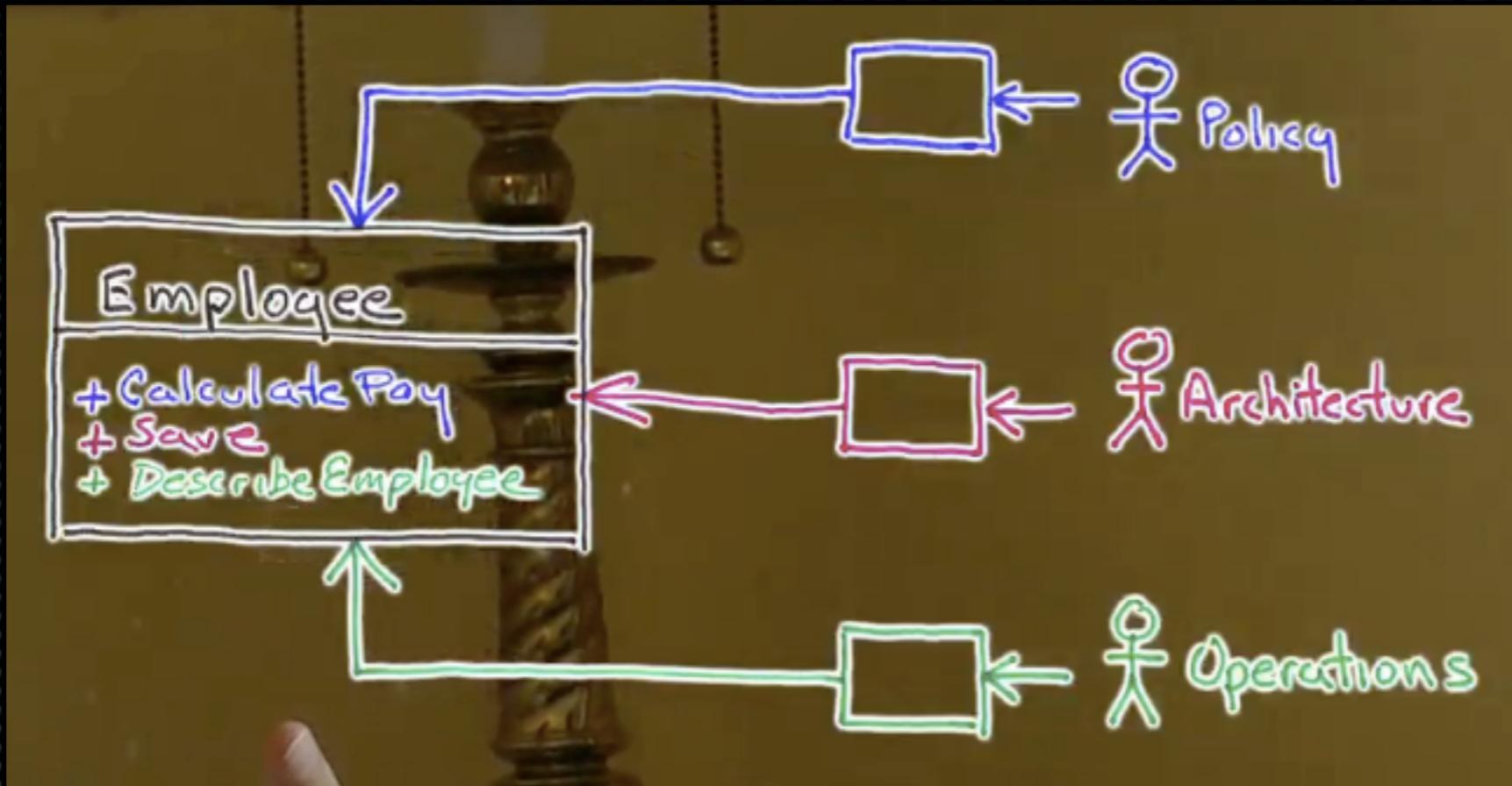
It's About Roles

- User들을 그들이 수행하는 Role에 따라 나눠야
- User가 특정 Role을 수행할 때 Actor라고 부른다.
- Responsibilities are tie to actors not to individuals
- Employee 클래스에는 3개의 액터가 있다
 - Policy, Architect, Operations

Reprise

- Responsibility
 - a family of functions that serves one particular actor
- Actor의 요구사항 변경이 일련의 함수들의 변경의 근원이 된다.

Collision

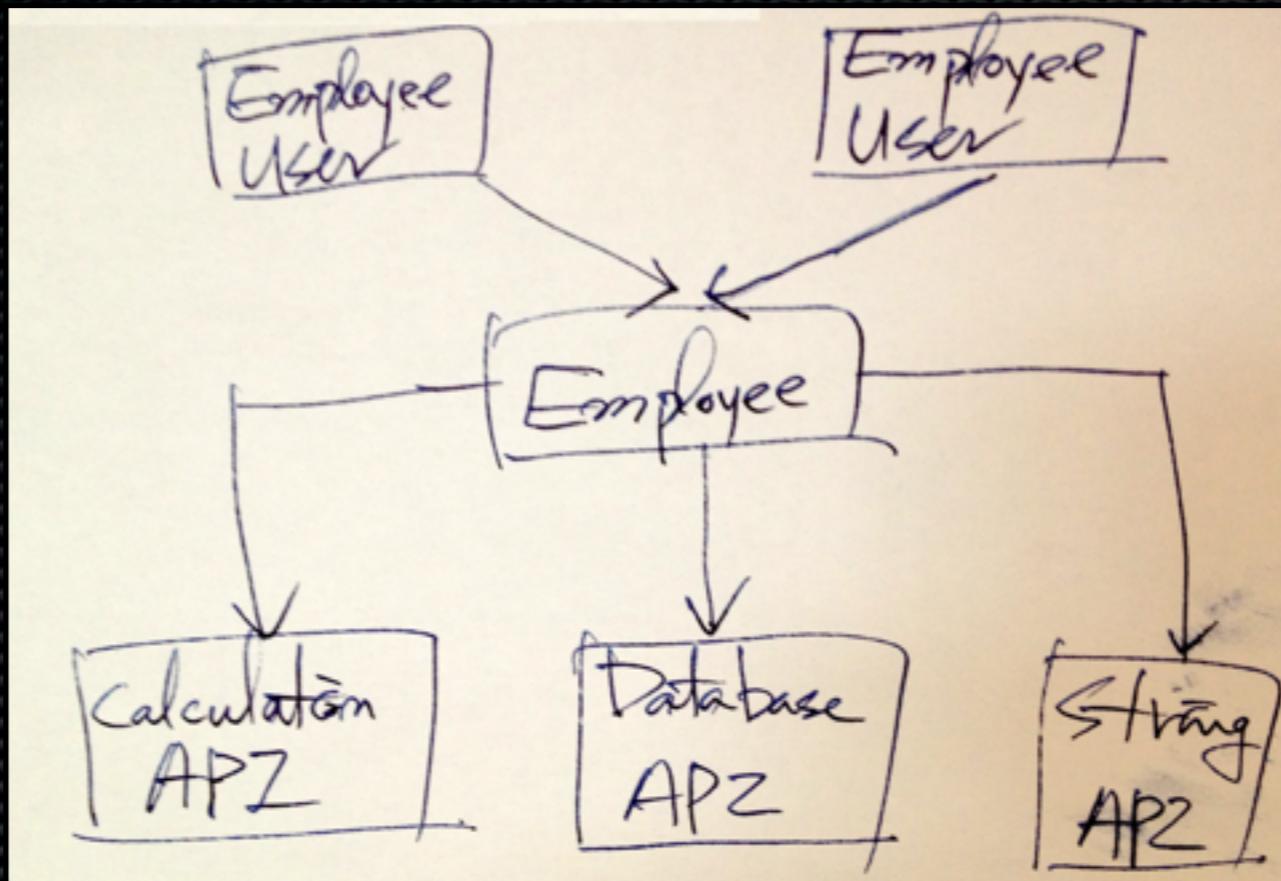


- Primary Value 저하
 - Policy Actors: business rule의 변경을 필요로 함.
 - Architecture Actors: DB Schema 변경을 필요로 함.
 - Operations Actors: Business rule 변경을 원한다.
 - 동일 모듈의 변경. Merge 충돌. Source Repo. 충돌

Fan Out

- Employee Class knows too much
 - business rules
 - DB
 - Reports, formatting
- 많은 책임을 갖는다.
- 각각의 책임을 다른 클래스들을 사용하도록 변경

Fan Out



- Employee 클래스에 거대한 Fan Out이 존재
- 변경에 민감. Employee User에는 더
- Fan Out을 제한하는 것이 바람직
 - 책임을 최소화

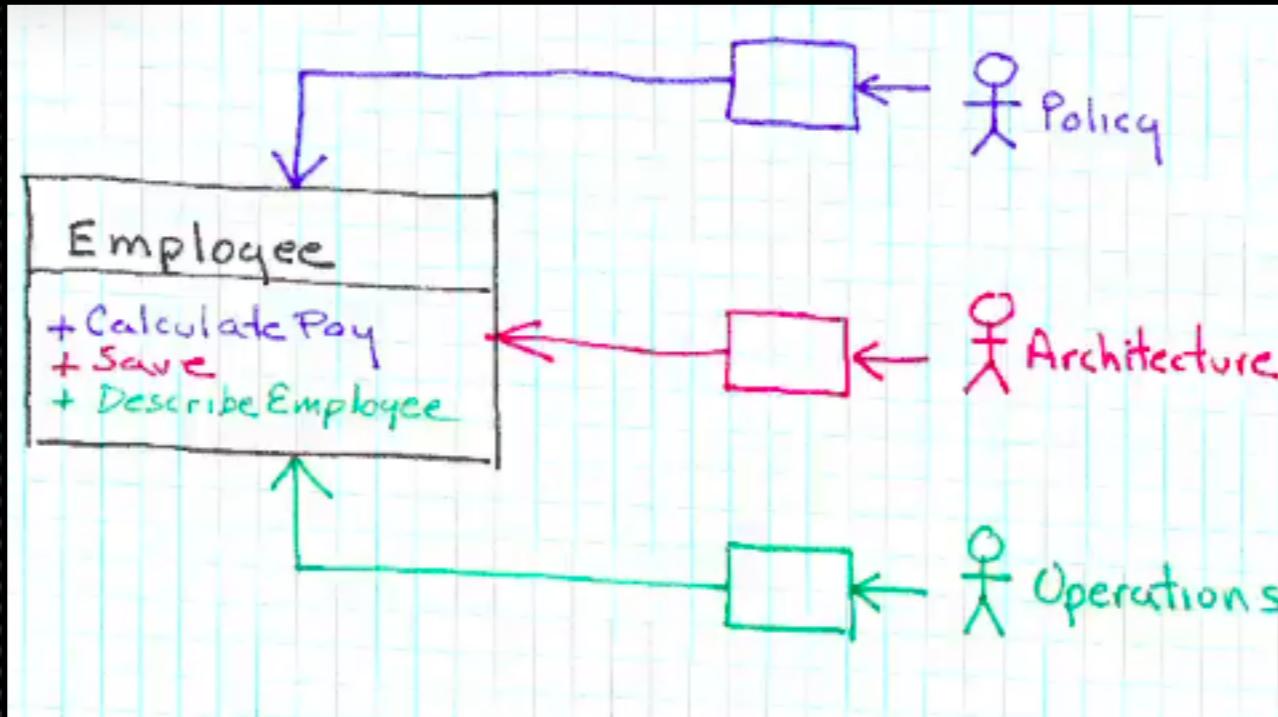
Collocation is Coupling

- Operations Actor가 새로운 리포트 기능을 필요
 - 새로운 리포트 기능도 Employee 클래스에 추가
 - 기존 책임(Policy, Architecture)에는 변경이 없음에도 새로운 리포트가 추가되어 Employee 클래스가 변경
 - 새로운 리포트 기능이 Employee 클래스에 추가되면 이 기능을 필요로 하지 않는 Employee 클래스를 사용하는 모든 클래스들이 다시 컴파일/배포되어야
 - 모든 액터들이 영향을 받게 된다.
 - “Collocation of responsibilities couples the actors”

SRP

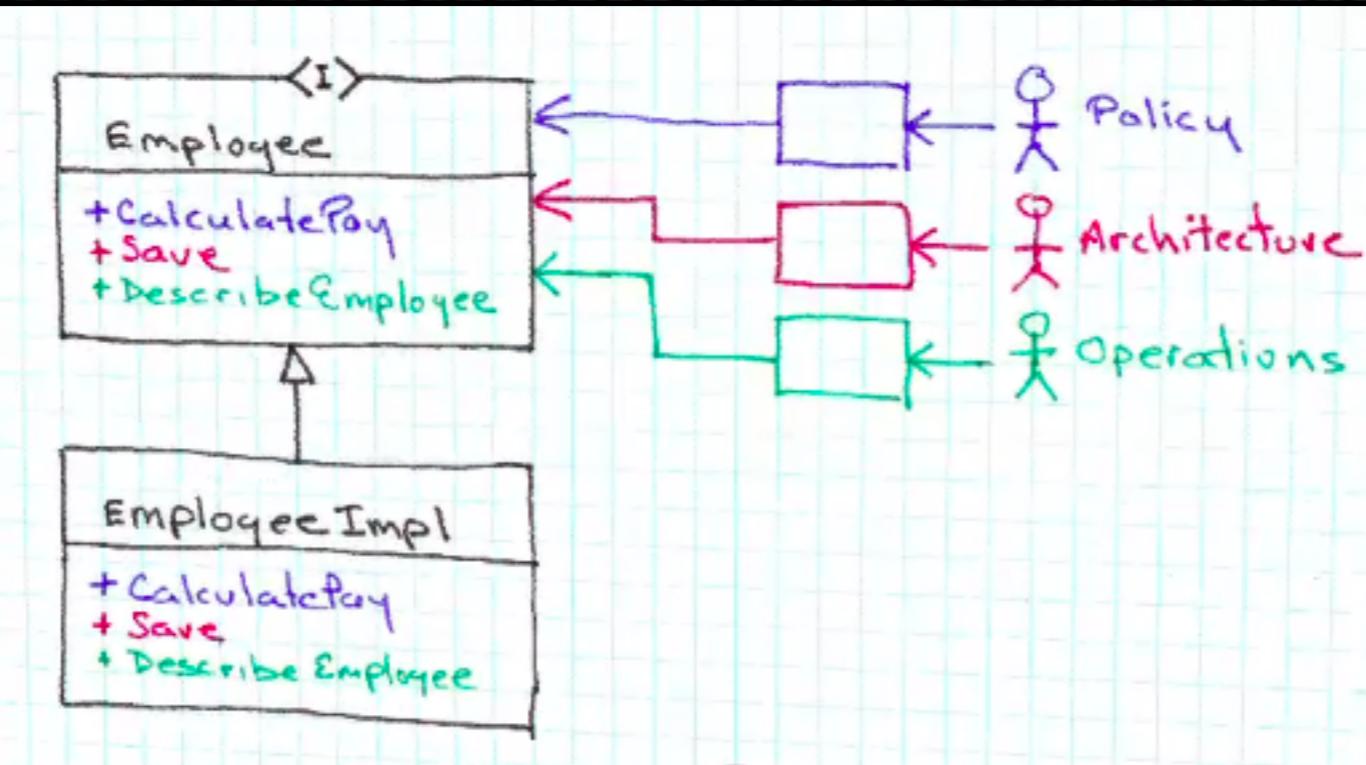
- Module should have one and only one reason to change.
- One and only one responsibility
 - 동일한 이유로 변경되어야 하는 것들은 동일 모듈에,
 - 다른 이유로 변경되어야 하는 것들은 다른 모듈에
- 시스템 설계
 - Actor 파악에 주의해야 함
 - Actor들을 serve하는 책임들을 식별
 - 책임을 모듈에 할당
 - 각 모듈이 반드시 하나의 책임을 갖도록 유지하면서.

Solutions



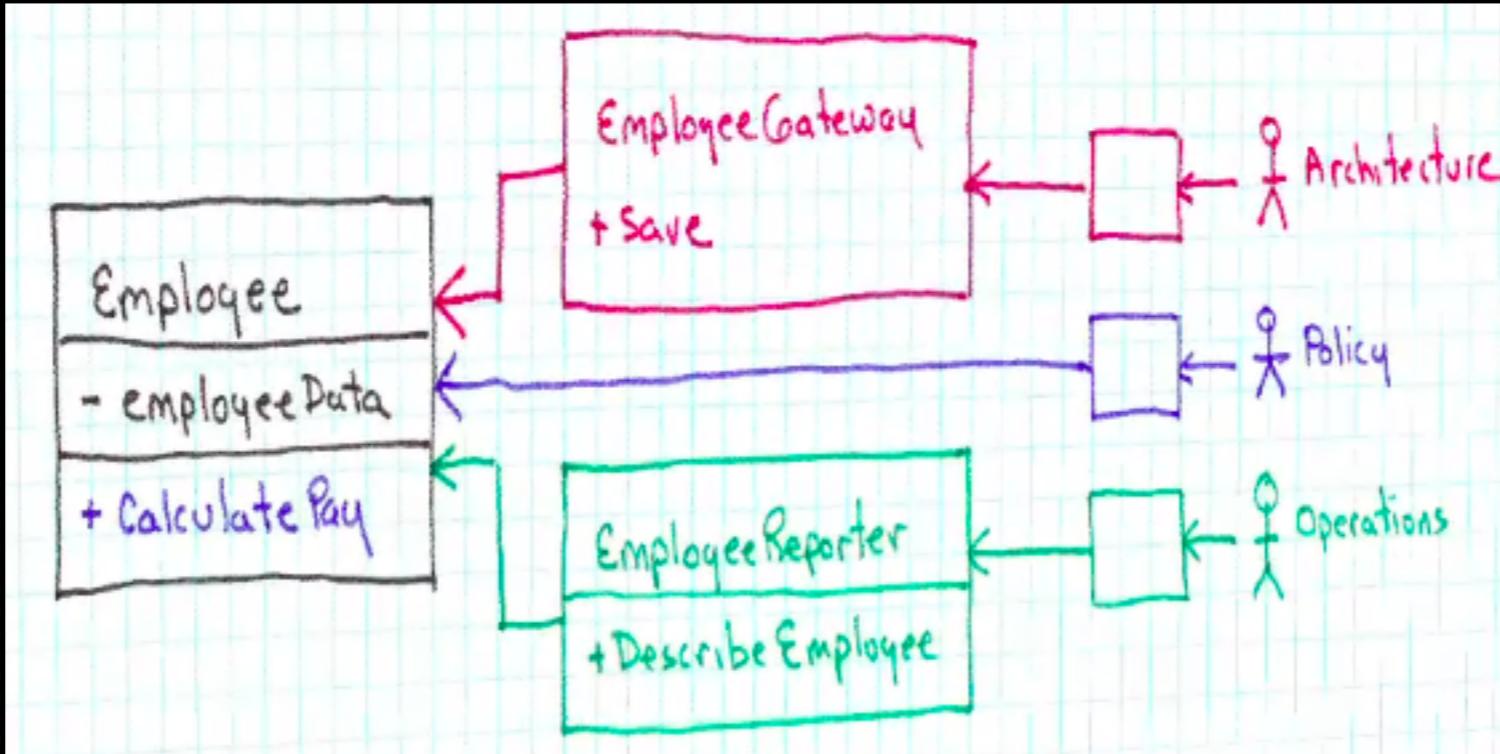
- 3개의 Actor, 3개의 Responsibility이 하나의 클래스
- 어떻게 Responsibility를 분리하나 ?

Inverted Dependencies



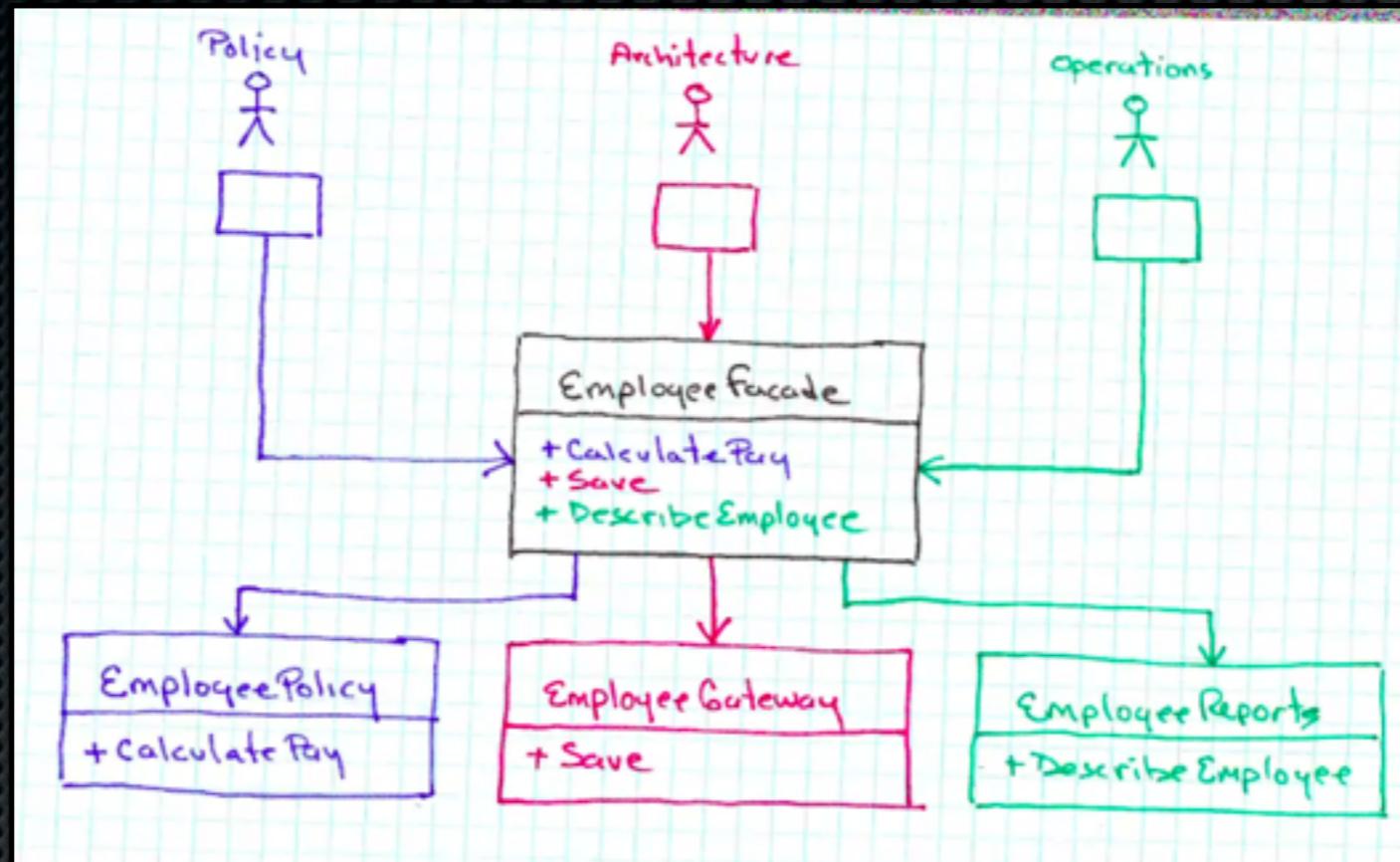
- OOP에서 이런 의존성을 다루는 전략
 - 클래스를 인터페이스와 클래스로 분리
- Actor를 클래스에서 Decouple
 - 모든 Actor들이 하나의 인터페이스에 coupled
 - 하나의 클래스에 구현되어 구현도 coupled

Extract Classes



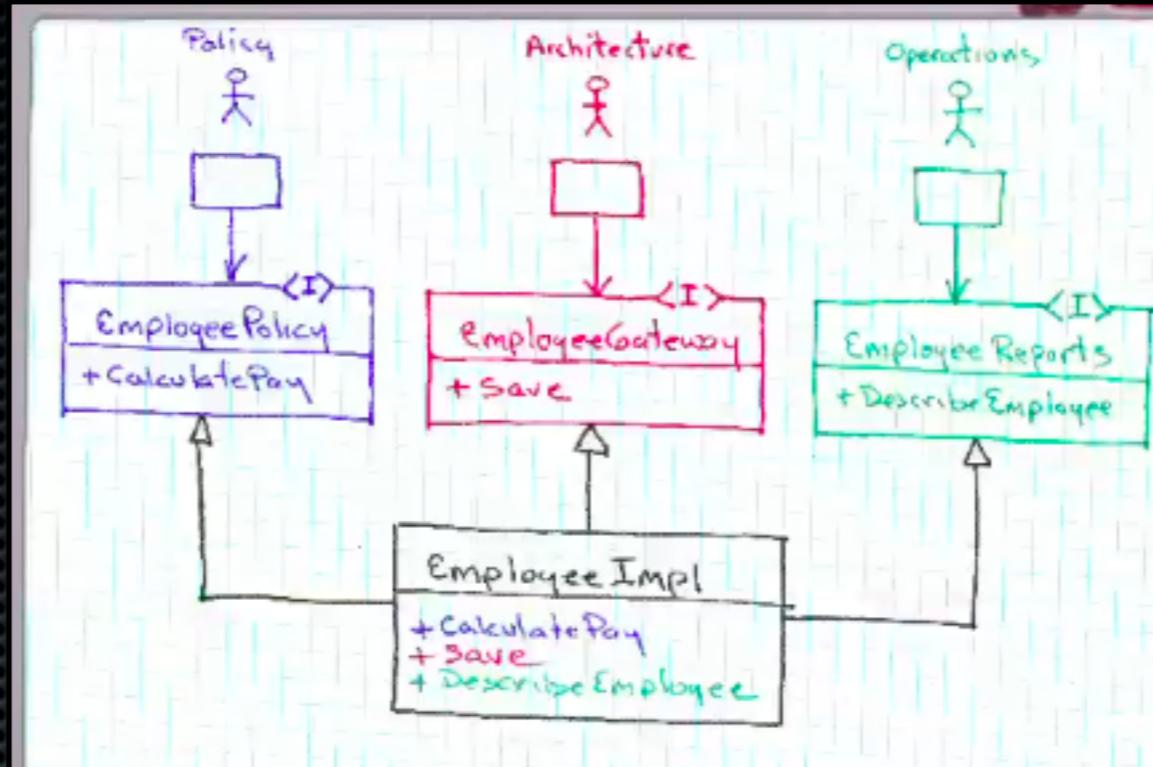
- 3개의 책임을 분리하는 방법: 3개의 클래스로 분리
- 결과
 - Actor들은 분리된 3개의 클래스에 의존
 - 3개의 책임에 대한 구현은 분리
 - 하나의 책임의 변경에 다른 책임에 영향 안미침
- 문제점
 - transitive dependency(EmployeeGateway/EmployeeReporter -> Employee)
 - Employee의 개념이 3개의 조각으로 분리

Facade - 어디에 구현이 있는지 찾기 쉽게



- Put all 3 function families into a facade class
- Facade delegates to the 3 different implementations.
- 장점
 - 어디에 구현되었는지 찾기 쉽다.
- 단점
 - Actor들은 여전히 Coupled

Interface Segregation

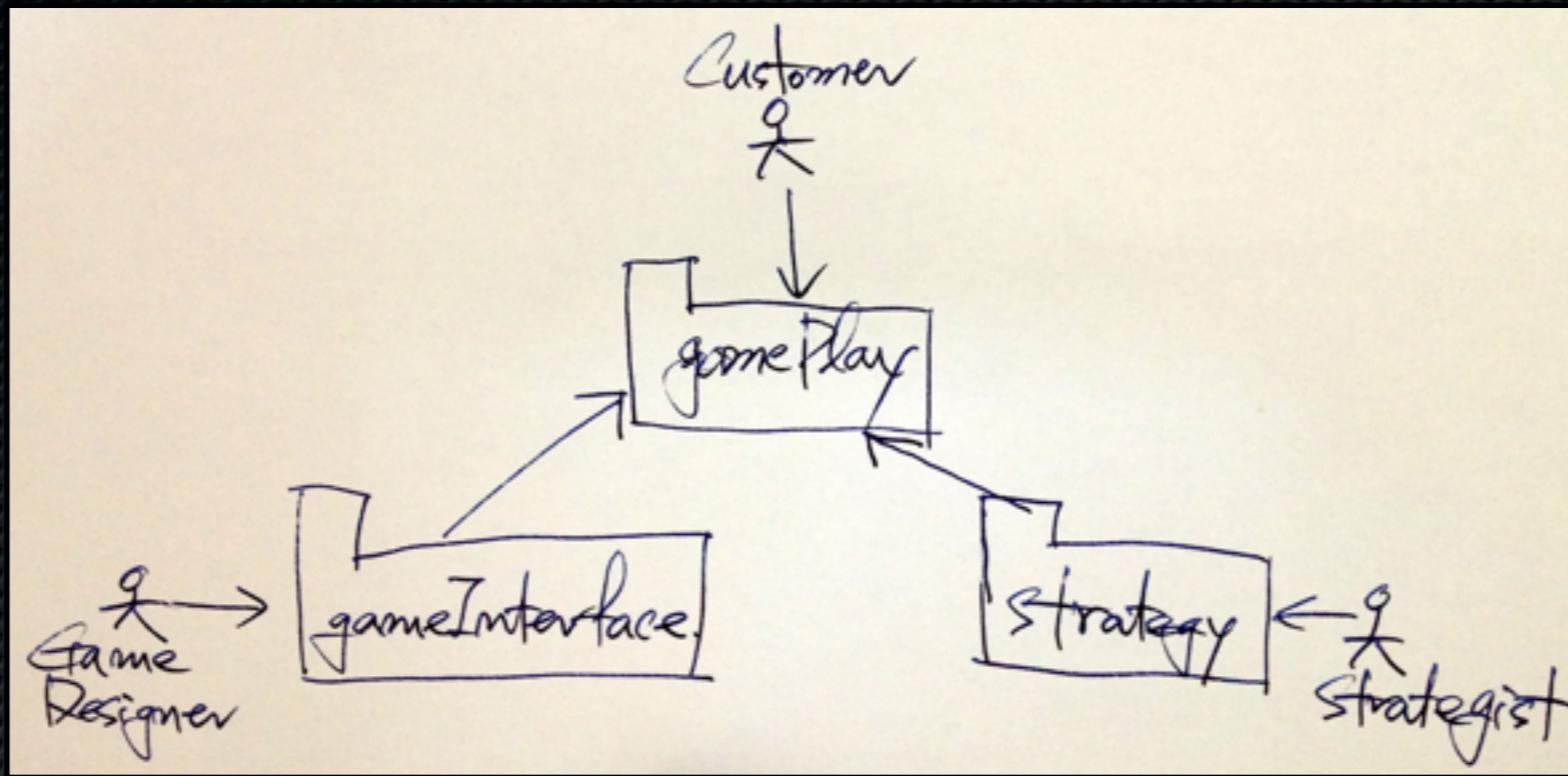


- Create 3 interfaces for each responsibility
- 3개의 인터페이스를 하나의 클래스로 구현
- 장점
 - Actor들은 완전히 decoupled
- 단점
 - 어디에 구현되었는지 찾기 어렵다.
 - 하나의 클래스에 구현되어 구현은 coupled

Case Study

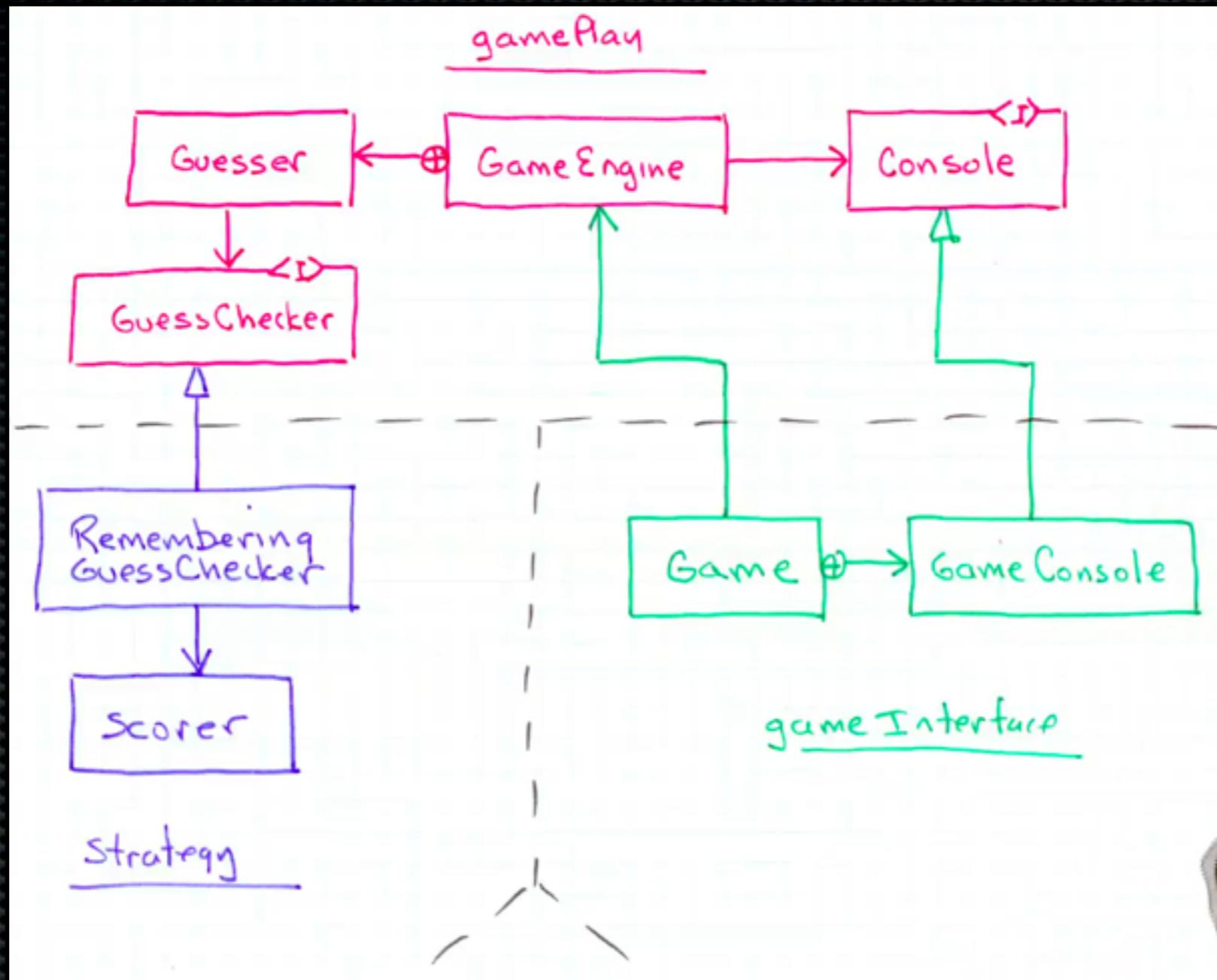
- 무엇인가를 변경할 때는 다른 Actor들에 영향을 주지 않고 변경해야 한다.

Case Study - Architecture



- Customer Actor를 위한 Responsibility가 어플리케이션 아키텍처의 중심
- 각 패키지는 각 액터들을 위한 책임을 구현
- 패키지 간의 의존성 방향에 주의
- 이 설계는 좋은 아키텍처
 - 어플리케이션이 중앙에, 다른 책임들은 어플리케이션에 plug into

Case Study - Design



- 각 클래스는 반드시 하나의 액터만을 위한 기능을 제공
 - 하나의 클래스는 반드시 하나의 책임만을 가짐

Faking It

- Waterfall 순서로 한 것 같은가?
 - 액터 → 패키지 디자인 → 클래스 디자인 → 코드
- step-by-step으로 이 복잡한 것들을 찾아 낸 것 같은가?

Faking It

- 실제로 한 것은
 - 제일 먼저 테스트를 작성하고 통과하도록 했다.
 - 스코어를 계산하는 동작하는 함수를 하나 만들고,
 - 추측하는 로직을 위한 동작하는 함수를 하나 만들고,
 - 설계가 드러날 때까지 이 함수 저 함수를 리팩토링했다.
 - 동작하는 전체 게임을 얻을때까지 모든 동작들이 테스트에 성공하도록 설계를 적용했다.
 - 그리고 아키텍처를 살폈다. 테스트가 당신들에게 보여준 설계의 80%를 유도했다.
 - 그리고 테스트가 3개의 책임을 식별하는 것을 도왔다.
 - unit test가 확보된 후에 무차별적인 리팩토링을 수행한다. 디자인을 향상 시키기 위해서.
 - 이런 후에만 3개의 액터들이 식별된다. 그러면 클래스들을 3개의 패키지로 분리시킨다.
 - 마지막으로 이쁜 다이어그램들을 그린다. 이쁜 다이어그램을 그리기 가장 좋은 때는 완료된 후이다.