▼ **An overview of design pattern - SOLID**

An overview of design pattern - SOLID

먼저 Design Principle인 SOLID에 대해 알아보자.

Design Smells

“Design Smells”는 나쁜 디자인을 나타내는 증상같은 것이고 아래 4가지 종류가 있다.

1. **Rigidity(경직성)**

   시스템이 변경하기 어렵다. 하나의 변경을 위해서 다른 것들을 변경 해야할 때 경직성이 높다. 경직성이 높다면 non-critical한 문제가 발생했을 때 관리자는 개발자에게 수정을 요청하기가 두려워진다.

2. **Fragility(취약성)**

   취약성이 높다면 시스템은 어떤 부분을 수정하였는데 관련이 없는 다른 부분에 영향을 준다. 수정사항이 관련되지 않은 부분에도 영향을 끼치기 때문에 관리하는 비용이 커지며 시스템의 credibility또한 잃는다.

3. **Immobility(부동성)**

   부동성이 높다면 재사용하기 위해 시스템을 분리해서 컴포넌트를 만드는 것이 어렵다.
   주로 개발자가 이전에 구현되었던 모듈과 비슷한 기능을 하는 모듈을 만들려고 할 때 문제점을 발견한다.

4. **Viscosity(점착성)**

   점착성은 디자인 점착성과 환경 점착성으로 나눌 수 있다.
   시스템에 코드를 추가하는 것보다 핵을 추가하는 것이 더 쉽다면 디자인 점착성이 높다고 할 수 있다. 예를 들어 수정이 필요할 때 다양한 방법으로 수정할 수 있을 것이다. 어떤 것은 디자인을 유지하는 것이고 어떤 것은 그렇지 못할 것이다.

   환경 점착성은 개발환경이 느리고 효율적이지 못할 때 나타난다. 예를들면 컴파일 시간이 매우 길다면 큰 규모의 수정이 필요하더라도 개발자는 recompile 시간이 길기 때문에 작은 규모의 수정으로 문제를 해결하려고 할 것이다.

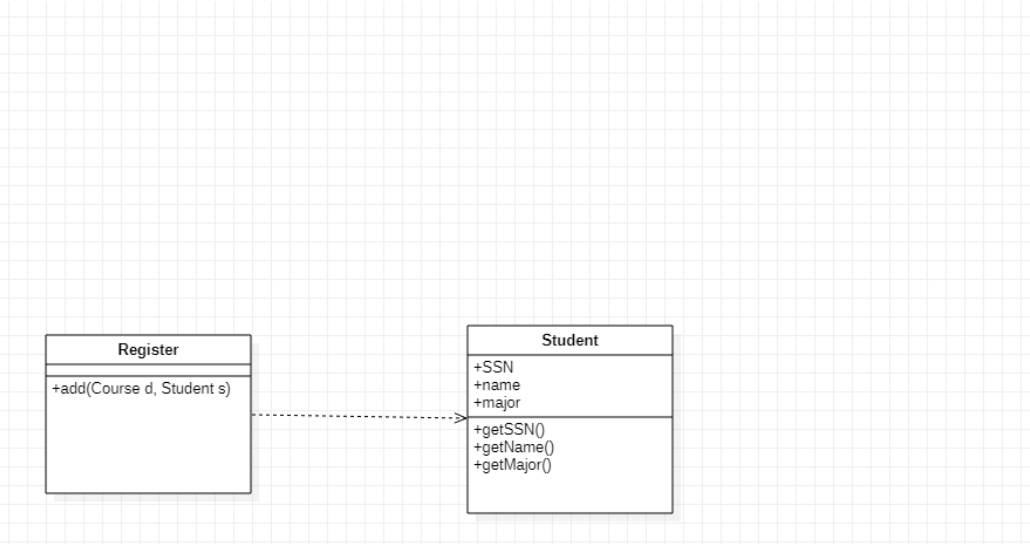
Robert C. Martin’s Software Design Principles (SOLID)

Robert C. Martin은 5가지 Software Design Principle을 정의하였고 앞글자를 따서 SOLID라고 부른다.

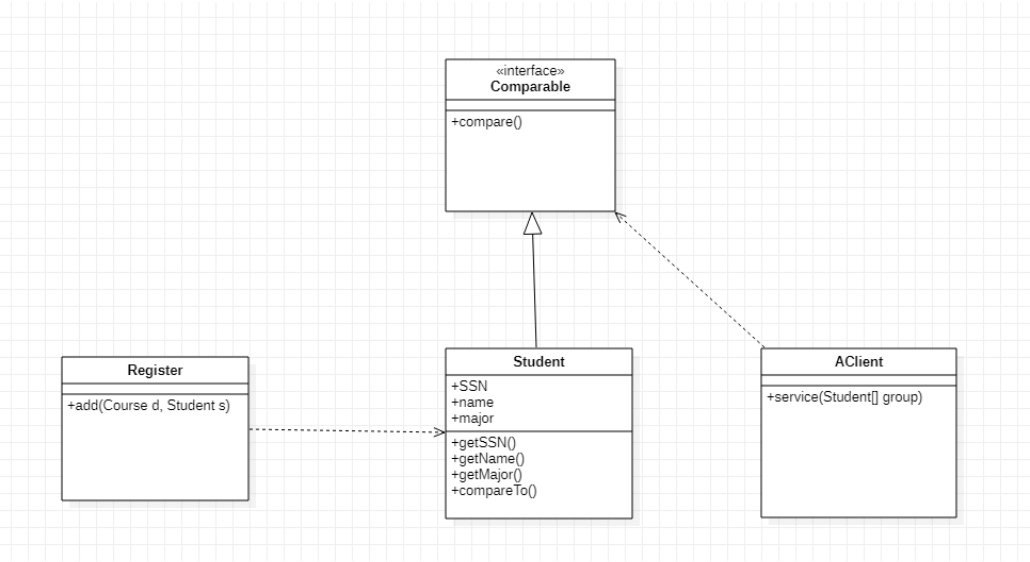
- **Single Responsibility Principle (SRP)**

  “A class should have one, and only one, reason to change”

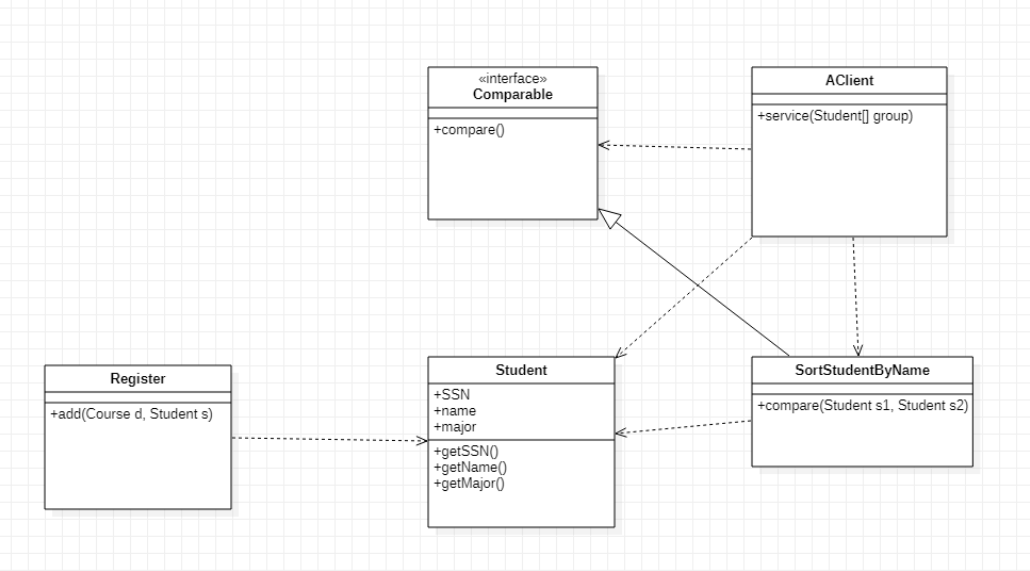
  클래스는 오직 하나의 이유로 수정이 되어야 한다는 것을 의미한다.



Register Class가 Student Class에 의존성을 가지고 있는 모습이다.
만일 어떠한 클래스가 Student를 다양한 방법으로 정렬하고 싶다면 아래와 같이 구현 할 수 있다.



다만 Register Class는 어떠한 변경도 일어나야하지 않아야 하고 Student Class가 바뀌어서 Register Class가 영향을 받는다. 정렬을 위한 변경이 관련 없는 Register Class에 영향을 끼쳤기 때문에 SRP를 위반한다. 이러한 위반을 해결하기위해서는 아래와 같이 구조를 변경하면된다.

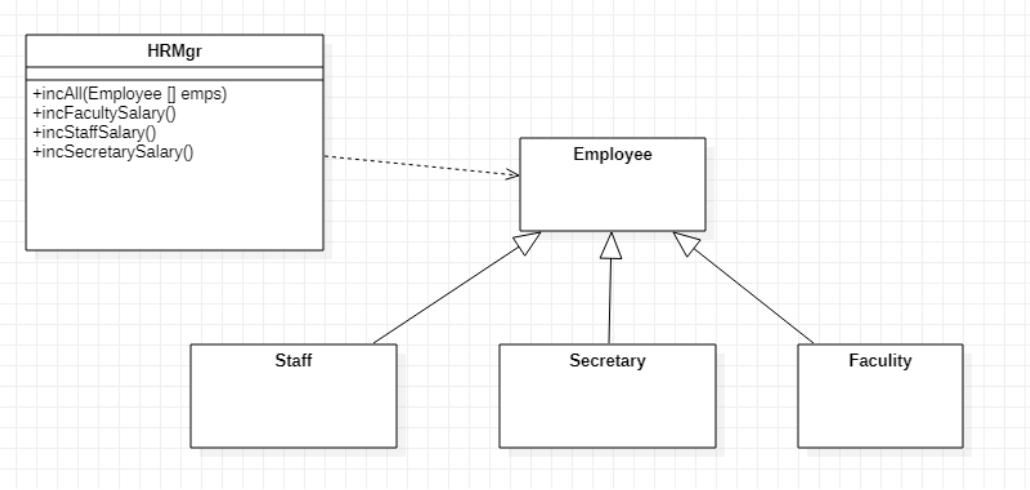


각각의 정렬 방식을 가진 Class를 새로 생성하고 Client는 새로 생긴 Class를 호출한다.

- **Open Closed Principle (OCP)**

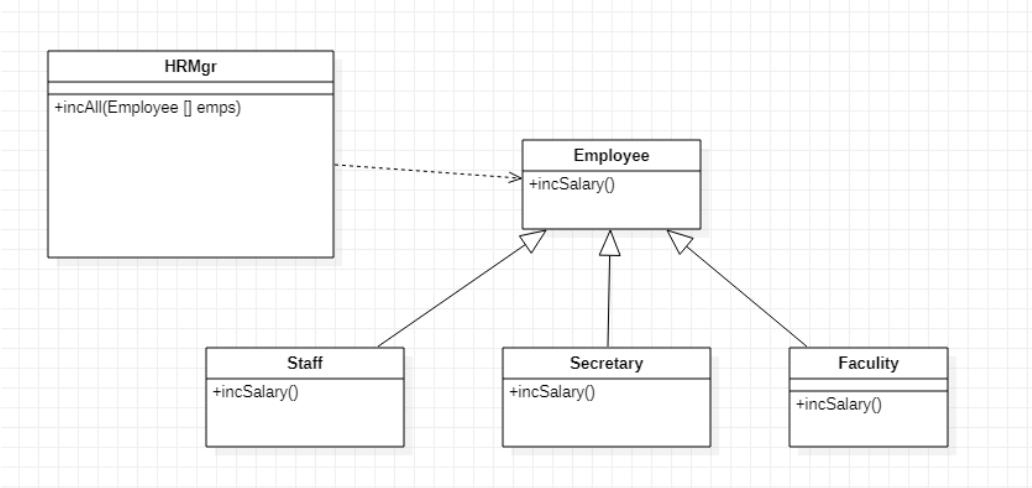
  “Software entities (classes, modules, funnctions, etc..) should be open for extension but closed for modification”

  자신의 확장에는 열려있고 주변의 변화에는 닫혀 있어야 하는 것을 의미한다.



HRMgr Class에 기능이 몰려있는것을 확인할 수 있다.

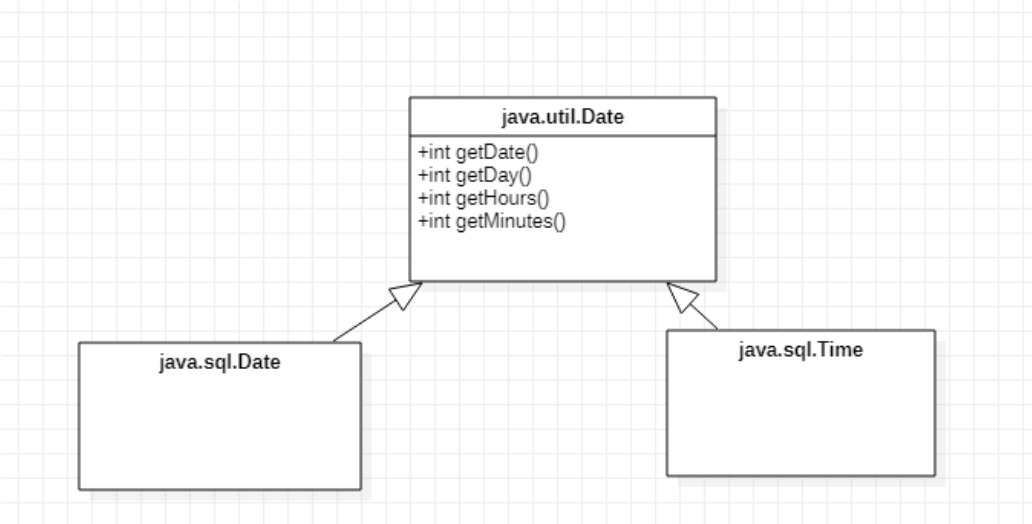
이를 적절하게 분산시킨후 HRMgr Class에서 incAll() 함수를 통해 문제를 해결할 수 있다.
이에 대한 도식화는 아래와 같다.



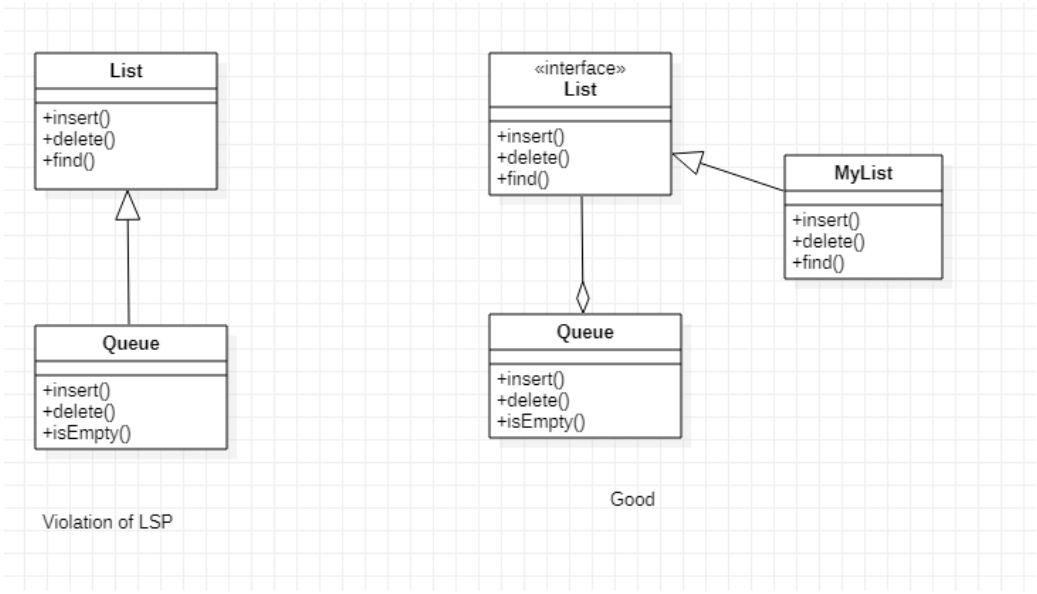
- **Liskov Substitution Principle (LSP)**

  “Subtypes must be substitutable for their base types”

  base Class에서 파생된 클래스는 base Class를 대체해서 사용할 수 있어야 한다.



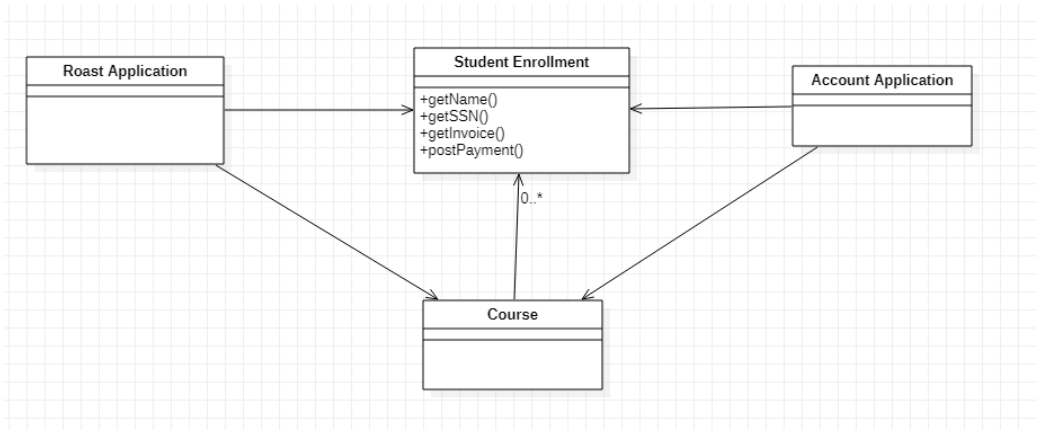
  아래 그림을 보면 List의 Implementation을 재사용할 경우 Inheritance보다 Object Composition을 사용하는 것을 추천한다. 만일 Queue Class가 List Class를 inheritance한다면 LSP를 위반한다.



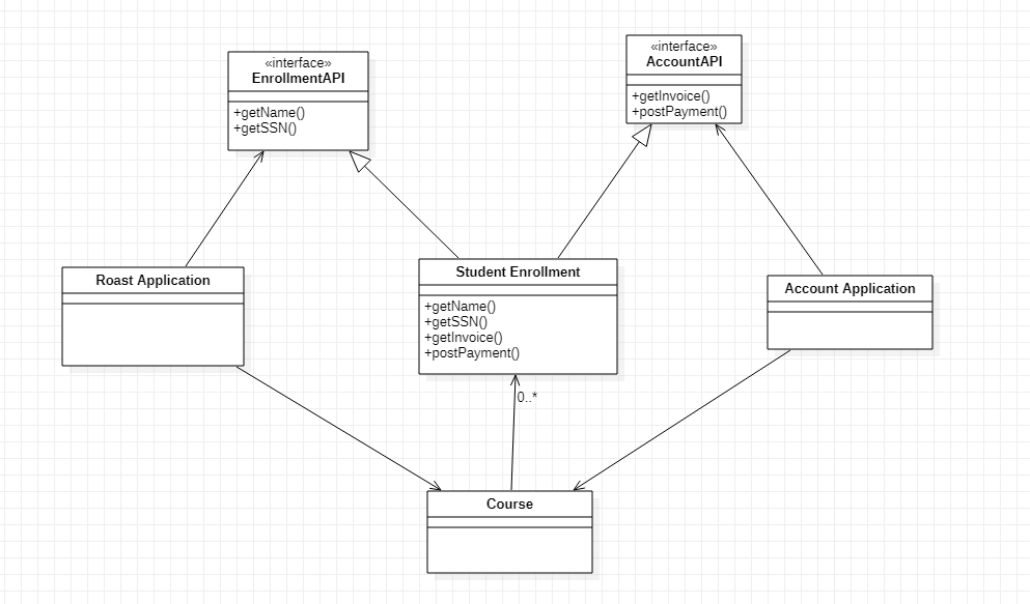
- **Interface Segregation Principle (ISP)**

  “Clients shoud not be forced to depend on methods they do not use”

  사용하지 않는 메소드에 의존하면 안된다.



Roast Application이 getName(), getSSN() 메서드만을 사용하고 Account Application은 getInvoice(), postPayment() 메서드만을 사용한다.

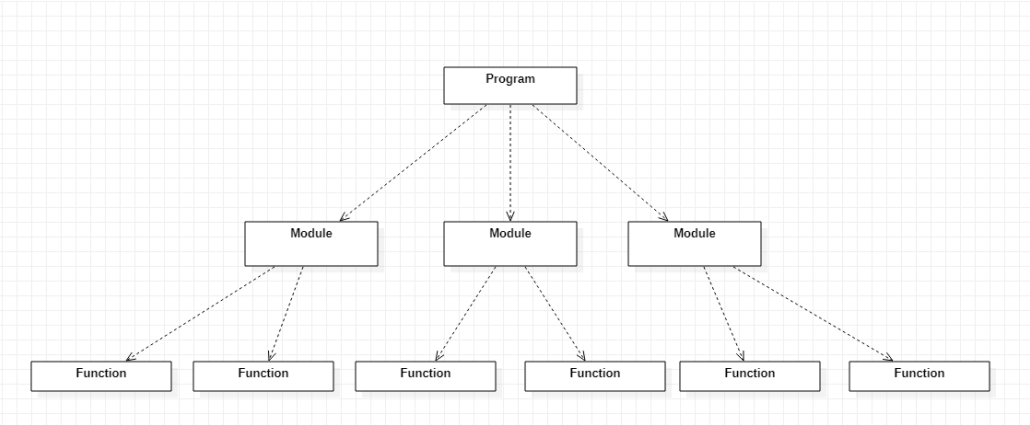


위 그림처럼 다이어그램 클래스에 맞는 interface를 만들어서 제공하면 ISP 문제를 해결할 수 있다.

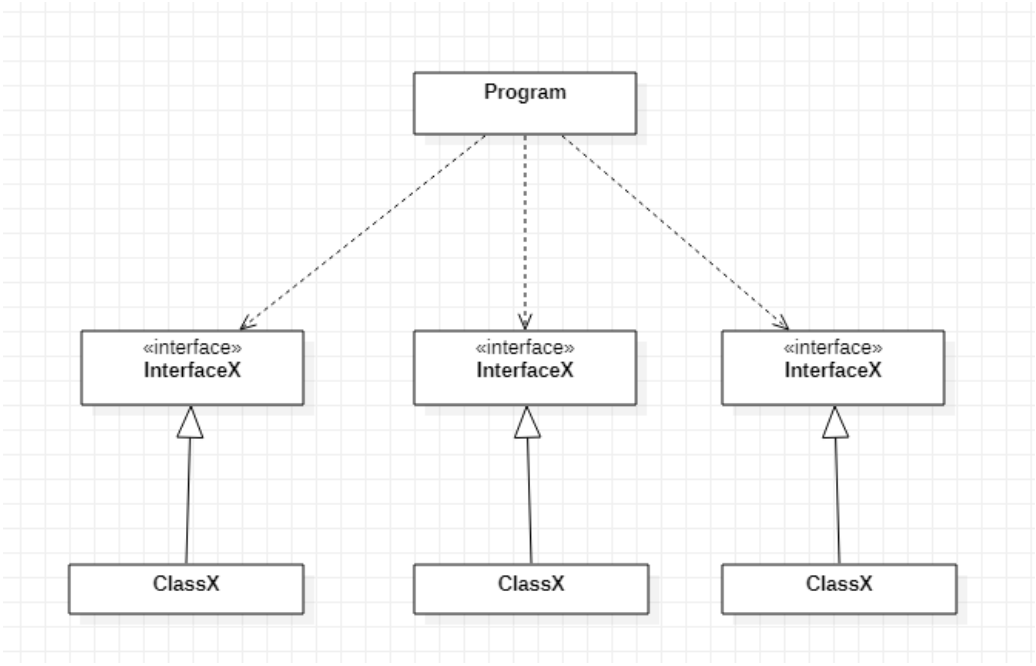
- **Dependency Inversion Principle (DIP)**

  “High-Level modules should not depend on low-level modules, Both should depend on abstractions”

  자신(high level module)보다 변하기 쉬운 모듈(low level module)에 의존해서는 안된다.



Program → Module → Fucntion 순으로 의존성이 구축된다.



Module Class를 인터페이스 클래스로 변경을 한 클래스 다이어그램이다. 의존성이 low level 방향으로 포함관계가 유지되지 않고 의존성이 inversion된 것을 볼 수 있다.

즉, DIP는 의존성을 inversion하는 것 뿐만 아니라 Ownership(소속관계)도 inversion 한다.