섹션 1 강의 내용 요약

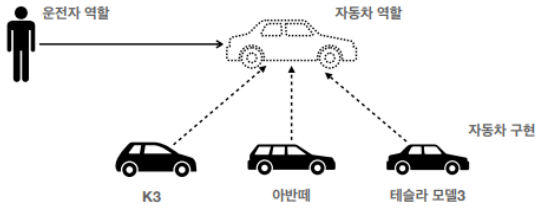
스프링 프레임워크의 핵심 목적: 자바 언어의 핵심인 객체 지향 프로그래밍을 이용해 좋은 애플리케이션을 개발할 수 있게끔 도와주는 것이다.

메모 포함[김1]: 프로그램을 유연하고 변경이 용이하게 (재사용성이 뛰어나게) 만든다.

다형성의 실세계 비유: 역할(인터페이스) / 구현(클래스, 구현 객체) =>

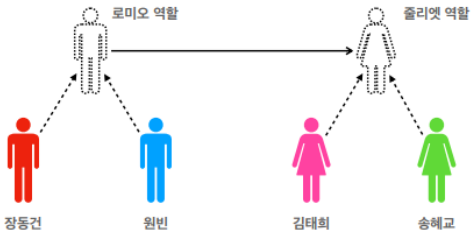
1. 구현이 바뀌어도 역할에 영향을 주지 않기 때문에 그 영향을 생각하지 않고 변경하는 것이 편하다.
2. 클라이언트는 구현 대상의 역할(인터페이스)만 알면 된다.
3. 클라이언트는 구현 대상의 내부 구조를 몰라도 되고, 구조 혹은 대상 자체가 변경되어도 영향을 받지 않는다.

운전자 - 자동차



공연 무대

로미오와 줄리엣 공연



➔ 객체 설계 시 역할(인터페이스)을 먼저 부여하고, 이후에 구현 객체를 만드는 방식이 좋다.

다형성의 본질: 구현 객체 인스턴스를 실행 시점에 유연하게 변경할 수 있다. 클라이언트를 변경하지 않고 서버의 구현 기능을 유연하게 변경할 수 있다.

역할과 구현을 분리하는 것...

- **정리**

1. 다형성을 이용해 실세계의 역할 / 구현이라는 개념을 프로그래밍 할 수 있다.
2. 유연하고 변경이 용이하다.
3. 확장 가능한 설계
4. 클라이언트에 영향을 주지 않는 변경이 가능하다.
5. 인터페이스를 안정적으로 잘 설계해야 한다.

한계

1. 역할 자체에 변경점이 생기면 클라이언트와 서버 모두에 큰 변화가 발생한다. -> 인터페이스를 안정적으로 잘 설계해야 한다.

스프링과 객체 지향

1. 스프링은 다형성을 극대화해서 활용할 수 있게 도와준다.

   ex) 제어 역전(IoC), 의존관계 주입(DI)
2. 구현을 편리하게 변경할 수 있다.

★ SOLID, 좋은 객체 지향 설계의 5가지 원칙

- SRP(Single Responsibility Principle): 단일 책임 원칙

  1. 하나의 클래스는 하나의 책임만 가져야 한다.
  2. 책임을 구분하는 중요한 기준은 변경이다. 변경이 있을 때 파급 효과가 적으면 단일 책임 원칙을 잘 따른 것이라고 판단한다.

     ex) UI 변경, 객체의 생성과 사용을 분리

- OCP(Open/Closed Principle): 개방-폐쇄 원칙

  1. 가장 중요한 원칙이다.
  2. 소프트웨어의 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.
  3. 다형성을 활용하여 구현할 수 있다.

     ex) 역할과 구현의 분리를 통해 기존 코드를 변경하지 않고 확장 (인터페이스의 변경 없이 구현 클래스만 만들어 확장하는 것)

```
public class MemberService {

   private MemberRepository memberRepository = new MemoryMemberRepository();

}
```

```
public class MemberService {

//   private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();


}
```

문제점: 구현 객체를 변경하려면 클라이언트 코드를 변경해야 한다. -> 다형성을 사용했지만 OCP 원칙을 지킬 수 없다. -> 객체를 생성하고 연관 관계를 맺어주는 별도의 assembler, setter가 필요하다.

- **LSP(Liskov Subtitution Principle): 리스코프 치환 원칙**

  1. 프로그램의 객체는 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.
  2. 다형성을 활용함에 있어 하위에 있는 클래스는 인터페이스 규약을 다 지켜야 한다는 의미이다. 구현 객체의 신뢰성 있는 사용을 위해 필요한 기능이다.

  ex) 자동차의 엑셀이라는 기능의 역할이 앞으로 가게끔 정의되어 있는데 구현은 뒤로 가게끔 이루어진 경우 LSP를 위반한 것이다.

- **ISP(Interface Segregation Principle): 인터페이스 분리 원칙**

  1. 특정 클라이언트를 위한 인터페이스 여러 개를 정의하는 것이 범용 인터페이스를 정의하는 것보다 낫다.
  2. 분리하면 인터페이스가 변경되어도 다른 인터페이스의 클라이언트에 영향을 주지 않아 좋다.
  3. 인터페이스가 명확해지고 대체 가능성이 높아진다.

- **DIP(Dependency Inversion Principle): 의존관계 역전 원칙**

  1. 프로그래머는 구체화가 아닌 추상화에 의존해야 한다. 의존성 주입은 이 원칙을 따르는 방법 중 하나이다.
  2. 구현 클래스에 의존하지 말고 인터페이스에 의존해야 한다.
  3. 프로그래밍을 할 때 역할에 의존해야 한다. 구현에 의존하게 되면 변경이 매우 어려워진다. 인터페이스에 의존해야 유연하게 구현을 변경할 수 있다.

```
public class MemberService {

   private MemberRepository memberRepository = new MemoryMemberRepository();

}
```

```
public class MemberService {

//   private MemberRepository memberRepository = new MemoryMemberRepository();
    private MemberRepository memberRepository = new JdbcMemberRepository();


}
```

문제점: MemberService는 인터페이스에 의존하고 있다. 그러나 MemoryMemberRepository, JdbcMemberRepository를 알고 있다. -> 구현 클래스들에도 의존하고 있다. -> DIP를 위반한다. -> 두 구현 클래스는 모르게끔 설계해야 한다.

- **정리**

1. 객체 지향의 핵심은 다형성이다.
2. 다형성만으로는 부품을 갈아 끼우듯 개발할 수 없다.
3. 다형성만 사용하면 구현 객체를 변경할 때 클라이언트의 코드도 함께 변경된다. (클라이언트가 구현 객체에 의존적이다.)
4. 다형성만으로는 OCP(개방-폐쇄 원칙), DIP(의존관계 역전 원칙)을 지킬 수 없다. -> 무언가 더 필요하다. -> 스프링

좋은 객체 지향 설계를 위한 스프링 기술

1. DI(Dependency Injection): 의존 관계, 의존성 주입
2. DI 컨테이너: 자바 객체들을 모아 의존 관계를 연결해 주고 주입해 준다.

➔ 다형성 + OCP + DIP가 가능해진다.

➔ 클라이언트 코드 변경 없이 기능을 확장할 수 있다.

➔ 부품을 교체하듯이 개발할 수 있다.

- **정리**

1. 모든 설계에 있어 역할과 구현을 분리한다.
2. 애플리케이션 설계 역시 공연을 설계하듯이 대본만 만들어두고 배우는 언제든지 유연하게 변경할 수 있도록 만드는 것이 좋은 객체 지향 설계이다.
3. 모든 설계에 인터페이스를 부여하는 것이 이상적인 설계이다. 그러나 인터페이스를 도입하면 추상화라는 비용이 발생한다. -> 기능을 확장할 가능성이 없다면 구현 클래스를 직접 사용하고, 향후에 필요할 때 리팩터링으로 인터페이스를 도입할 수 있다.