study 1

객체지향 프로그래밍이란 무엇인가?

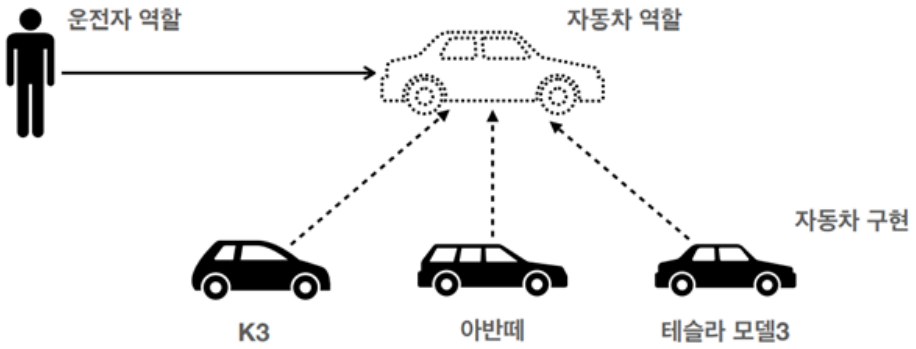
1. 객체 지향 프로그래밍 이란?
    - 객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이다. 각각의 객체는 서로 협력하여 메시지를 주고 받고, 데이터를 처리할 수 있다.
    - 객체 지향 프로그래밍은 프로그램을 **유연**하고 **변경**이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용된다.
    - **"유연하고 변경이 용이"**
        - 레고 블럭 조립하듯이, 컴퓨터 부품 갈아 끼우듯이
        - 어느 한 코드를 변경 & 수정하더라도 다른 코드에 영향을 주지 않는 것

      → 객체 지향 특징 중 "**다형성**"과 연결됨

2. 객체 지향 특징 - 다형성

   1) 다형성 : 여러가지 형태를 가질 수 있는 능력 → **역할**과 **구현**으로 대상을 분리
    - 예 : 운전자 - 자동차

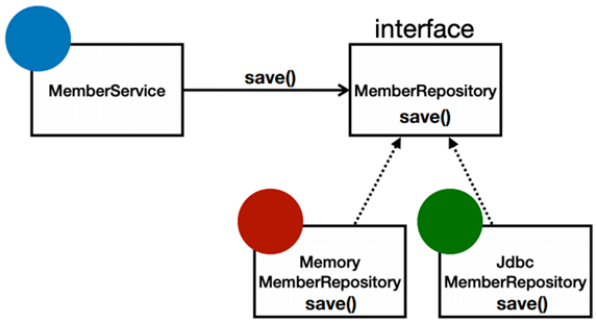
      운전자가 자동차를 사용할 때, 자동차의 구현체가 무엇이든(K3, 아반떼..) 상관 없이 '자동차' 자체를 운전하는 방법만 알면된다.



    - 즉, 클라이언트는 구현 대상의 내부 구조는 알 필요 X

- 클라이언트는 대상의 역할 (인터페이스)만 알면 된다.
- 클라이언트는 구현 대상의 내부 구조가 변경되거나, 대상 자체를 변경해도 영향을 받지 않는다.
- 다시 말해, 다형성을 적용하여 프로그래밍하면, 클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경할 수 있다.

2) 자바 언어의 다형성



여기서 MemberService가 클라이언트

- 객체를 설계할 때 역할과 구현을 명확히 분리
- 따라서 객체를 설계할 때 역할(인터페이스)을 먼저 만들고, 그 역할을 수행하는 구현 객체를 만든다.

```
public class MemberService {
  private MemberRepository memberRepository = new MemoryMemberRepository ();
  memberRepository.add(member);
}
```

```
public class MemberService {
  private MemberRepository memberRepository = new JdbcMemberRepository ();
  memberRepository.add(member);
}
```

MemberService는 MemberRepository가 뭐로 구현되는 상관 없이, MemberRepository 인터페이스면 알면 사용 가능하다. → MemberRepository의 구현체가 뭐인지 알 필요 없다.

3. 좋은 객체 지향 설계 원칙

: 단일 책임 원칙, 개방 폐쇄 원칙, 리스코프 치환 원칙, 인터페이스 분리 원칙, 의존관계 역전 원칙

- **개방-폐쇄 원칙 (OCP) :** 소프트웨어 요소는 **확장에는 열려** 있으나 **변경에는 닫혀** 있어야한다.
- **의존관계 역전 원칙 (DIP) :** 클라이언트는 **추상화 (역할, 인터페이스)에 의존**해야하며, **구체화 (구현체)에 의존해서는 안된다.**

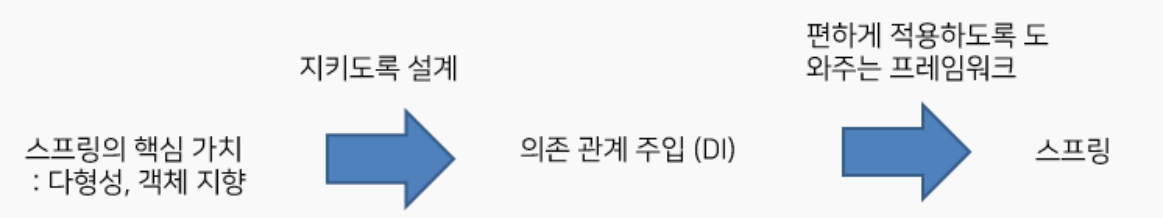
  예) 위 코드에서 다른 Repository를 사용하려면 MemberService(클라이언트) 로직을 수정해야함

  → 이유 : MemberService 클라이언트가 구현 클래스를 직접 선택하기 때문 ⇒ OCP, DIP 위배

→ 해결 : **객체를 생성하고, 연관 관계를 맺어주는 별도의 조립, 설정자가 필요**하다.

⇒ 의존 관계 주입 (DI)

4. 스프링과 DI
   - 스프링은 DI를 통해 다형성, OCP, DIP 원칙을 지키면서 개발하기 쉽도록 프레임워크로 제공



   - 스프링의 핵심 가치는 다형성과 객체 지향을 지키도록 설계하자는 것
   - 이를 위해 의존 관계 주입(DI) 개념 등장
   - 스프링은 DI와 같은 유용한 기술을 편하게 적용하도록 도와줌

RESTful API

- 네트워크 기반 소프트웨어 아키텍처 스타일 → REST
  - REST : Resource Oriented Architecture
  - RESTful API : REST의 기본 원칙을 성실히 지킨 서비스 디자인을 RESTful 하다라고 표현한다.
  - API 설계의 중심에 자원(Resource)이 있고, HTTP Method를 통해 자원을 처리하도록 설계하는 것
- REST 중심 규칙
  - URI는 정보의 자원을 표현해야 한다.
  - 자원에 대한 행위는 HTTP Method (GET, POST, PUT, DELETE)

- 예

  1번 사용자에 대해 정보를 받아야 할때, 아래와 같은 방법은 좋지 않다.

```
GET /users/show/1
```

  이와 같은 URI 방식은 REST를 제대로 적용하지 않았다고 볼 수 있다.

  왜냐하면 자원을 표현해야하는 URI에 `/show/`와 같은 불필요한 표현이 들어있기 때문이다.

  '본다'는 행위는 GET이라는 HTTP Mehod로 충분히 표현할 수 있다. 따라서 다음과 같이 변경해야한다.

```
GET /users/1
```

- 자원은 크게 Collection과 Element로 나누어 표현할 수 있으며, 아래 테이블에 기초한다면 서버 대부분과의 통신 행태를 표현할 수 있다.

RESTful Web Service HTTP methods

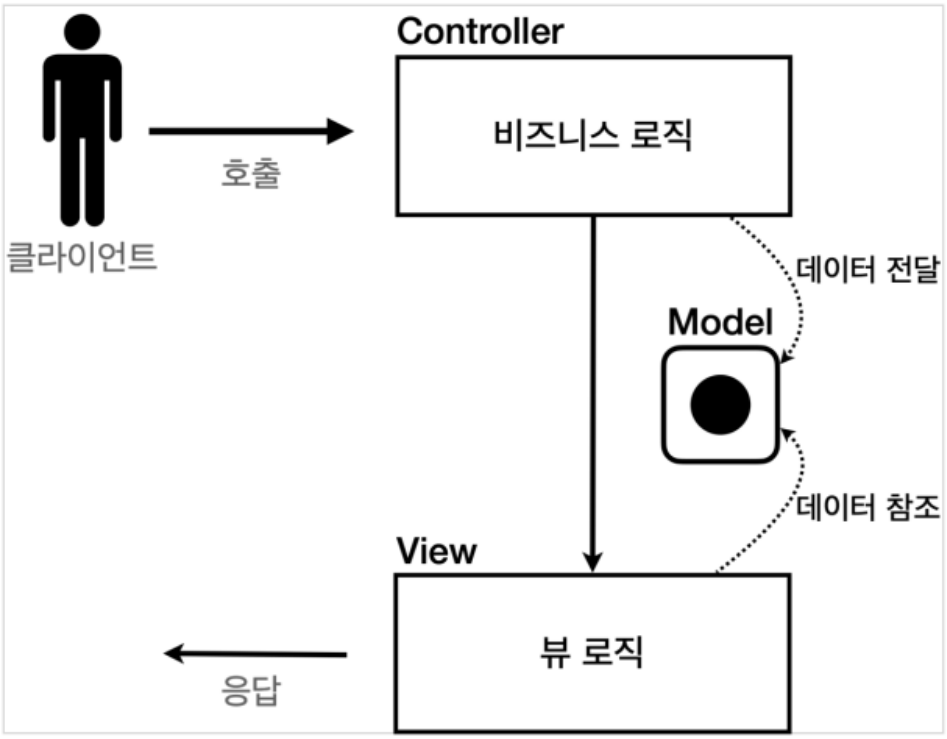
Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://example.com/resources/`	컬렉션에 속한 자원들의 URI나 그 상세사항의 목록을 보여준다.	전체 컬렉션은 다른 컬렉션으로 교체한다.	해당 컬렉션에 속하는 새로운 자원을 생성한다. 자원의 URI는 시스템에 의해 할당된다.	전체 컬렉션을 삭제한다.
Element URI, such as `http://example.com/resources/item17`	요청한 컬렉션 내 자원을 반환한다.	해당 자원을 수정한다.	해당 자원에 귀속되는 새로운 자원을 생성한다.	해당 컬렉션내 자원을 삭제한다.

MVC 패턴이란?

- Model -View -Controller
- 소프트웨어 디자인 패턴
- 하나의 서블릿이나 JSP 만으로 비즈니스 로직과 뷰 렌더링까지 모두 처리하게 되면, 너무 많은 역할을 하게 되고, 결과적으로 유지보수가 어려워짐
- 사용자 인터페이스(UI)와 비즈니스 로직을 분리 → 비즈니스 로직이 변경되더라도 UI에 영향 X
- 컨트롤러
  - HTTP 요청을 받아서 파라미터를 검증하고, 비즈니스 로직을 실행
  - 뷰에 전달할 결과 데이터를 조회해서 모델에 담는다.
- 모델
  - 뷰에 출력할 데이터를 담아둔다. 뷰가 필요한 데이터를 모두 모델에 담아서 전달해주는 덕분에 뷰는 비즈니스 로직이나 데이터 접근을 몰라도 되고, 화면을 렌더링 하는 일에 집중할 수 있다.
- 뷰

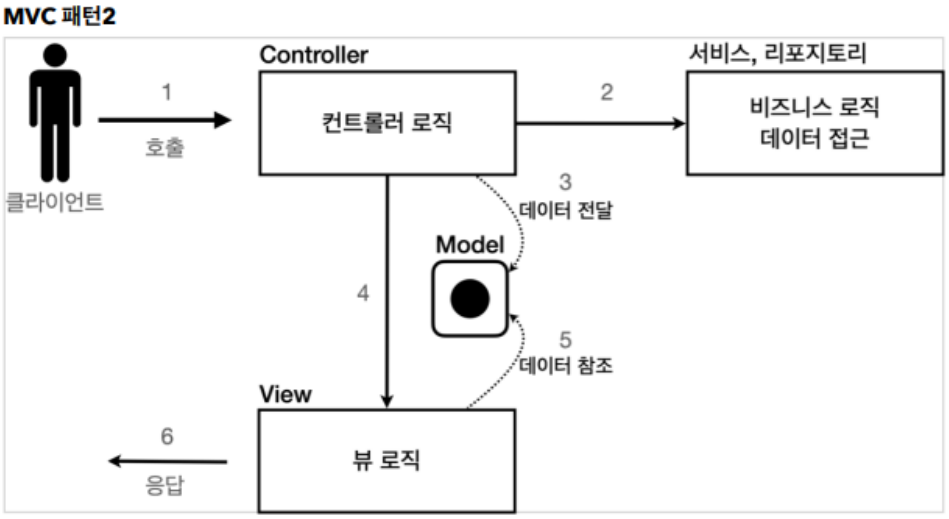
- 모델에 담겨있는 데이터를 사용해서 화면을 그리는 일에 집중한다.

mvc 패턴 1



- 컨트롤러에 비즈니스 로직을 둘 수도 있지만, 그러면 컨트롤러가 너무 많은 역할을 담당하게 됨. 그래서 일반적으로 비즈니스 로직은 서비스라는 별도의 계층에서 처리하도록 한다.

mvc 패턴2



TTD

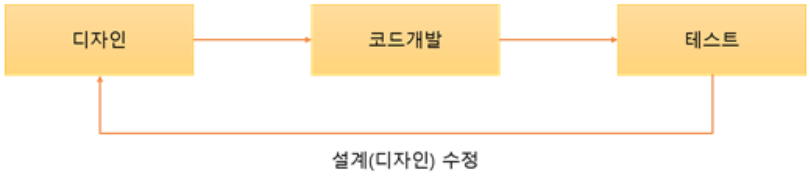
https://github.com/JaeYeopHan/Interview_Question_for_Beginner/tree/master/Development_common_sense#tdd

TDD란?

- Test Driven Development의 약자로 '테스트 주도 개발'이라고 한다.
- 작은 단위의 테스트 케이스를 작성하고 이를 통과하는 코드를 추가하는 단계를 반복하여 구현한다.

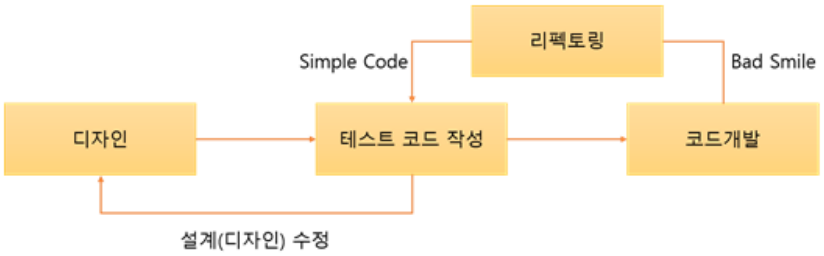
일반 개발 방식과 비교

- 일반 개발 방식



  - 요구사항 분석 → 설계 → 개발 → 테스트 → 배포 형태의 개발 주기를 갖는다.
  - 새로운 기능을 추가할 때마다 전체 코드에 대해 테스트를 해야하므로 유지보수가 어렵다

- TDD 개발 방식

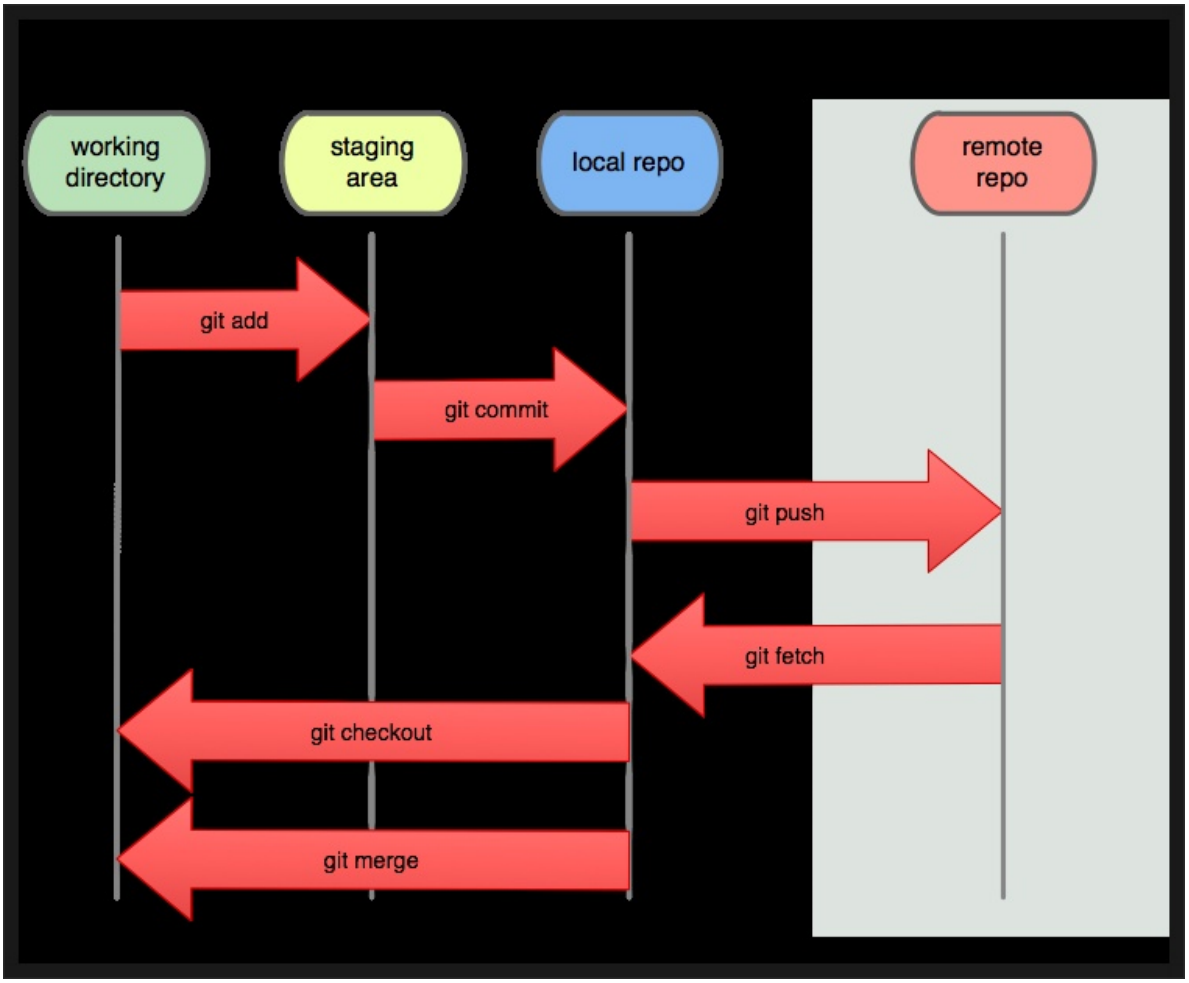


  - 테스트 코드를 작성한 뒤에 실제 코드를 작성한다.
  - 디자인 (설계) 단계에서 프로그래밍 목적을 반드시 미리 정의해야만 하고, 무엇을 테스트 해야할 지 미리 정의 (테스트케이스 작성)해야만 한다.
  - 단위 기능별로 테스트 케이스를 작성하여 개발하므로, 새로운 기능을 추가하기가 용이하다.
  - 단점
    - 생산성 저하 : 테스트 코드를 추가로 작성해야 하므로 생산성이 저하될 수 있다.

- 테스트 코드 작성의 어려움 : 팀원 모두 테스트 코드 작성에 대한 이해가 필요하며, 상황에따라 테스트 코드를 작성하기가 때로는 개발 자체보다 어려울 수 있다.

Git과 Github

- 형상관리 도구(버전 관리 시스템) 중 하나
- SVN : 중앙 서버에 소스코드와 히스토리를 관리
- Git : 소스 코드를 여러 개발 PC와 저장소에 분산해서 관리
  - 소스 코드 사본을 로컬에서 관리하기 때문에 git이 svn보다 훨씬 빠르다 (SVN은 변경 로그 하나 보는 것도 인터넷을 경유해야한다.)
- 브랜치를 통해 각자 병렬로 개발이 가능하고, 이후 merge를 통해 합치는 방식으로 개발 할 수 있다.



- git add : 다음 변경 (commit)을 기록할 때까지 변경분을 모아놓기 위함. 따라서 git commit 명령어를 통해 명시적으로 기록을 남기기 전까지는 아무리 git add 명령어를 많이 실행하도 git 저장소의 변경 이력에는 어떤 영향도 없다.

  → 이 방법으로 원하는 기능만 따로 commit하기 위한 staging 영역에 올려 놓는 것
- git commit : 의미있는 변화를 기록하기 위함. (작업이 완결된 상태)

- git pull : 원격 저장소로부터 필요한 파일을 다운 + 병합
- git fetch : 원격 저장소로부터 필요한 파일을 다운 (병합은 따로 해야함). fetch를 이용해 원래 내용과 바뀐 내용과의 차이를 알 수 있다. 이후 git merge origin/master 하면 git pull 상태와 같아진다. (병합까지 완료)

함수형 프로그래밍

- 명령형 프로그래밍 : 어떻게(How)할 것인지를 설명하는 방식
  - 절차지향 프로그래밍 : 수행되어야 할 순차적인 처리 과정을 포함하는 방식
  - 객체지향 프로그래밍 : 객체들의 집합으로 프로그램의 상호작용을 표현

- 선언형 프로그래밍 : 무엇(What)을 할 것인지를 설명하는 방식
  - 함수형 프로그래밍 : 순수 함수를 조합하여 소프트웨어를 만드느 방식 (클로저, 하스켈, 리스프)

참고)

- RESTful API
  - https://spoqa.github.io/2012/02/27/rest-introduction.html