

## 제 2 강 프레임워크 개요

### 1. 객체지향 프로그램 기법에 대한 고찰

#### ① 초 난감 DAO

■ ID, NAME, PASS의 3개 필드를 가지고 있는 MyUser라는 테이블이 있다고 가정해 보자. MyUser 테이블과 연동할 수 있는 UserVO가 있고 아래와 같은 UserDao 클래스가 있다.

```
1 public class UserDao {
2     public void add(UserVO user) throws ClassNotFoundException, SQLException {
3         Class.forName("oracle.jdbc.driver.OracleDriver");
4         Connection c = DriverManager.getConnection(
5             "jdbc:oracle:thin:@localhost:1521/XEPDB1", "mytest", "mytest");
6         PreparedStatement ps = c.prepareStatement(
7             "insert into myuser(id, name, pass) values(?,?,?)");
8         ps.setString(1, user.getId());
9         ps.setString(2, user.getName());
10        ps.setString(3, user.getPass());
11        ps.executeUpdate();
12        ps.close();
13        c.close();
14    }
15    public User get(String id) throws ClassNotFoundException, SQLException {
16        Class.forName("oracle.jdbc.driver.OracleDriver");
17        Connection c = DriverManager.getConnection(
18            "jdbc:oracle:thin:@localhost:1521/XEPDB1", "mytest", "mytest");
19        PreparedStatement ps = c
20            .prepareStatement("select * from myuser where id = ?");
21        ps.setString(1, id);
22        ResultSet rs = ps.executeQuery();
23        rs.next();
24        UserVO user = new UserVO();
25        user.setId(rs.getString("id"));
26        user.setName(rs.getString("name"));
27        user.setPass(rs.getString("pass"));
28        rs.close();
29        ps.close();
30        c.close();
31        return user;
32    }
33 }
```

※ 위 DAO 클래스는 우리가 일반적인 JDBC를 활용할 경우에 많이 볼 수 있는 코드이다. 위 DAO의 문 제점이 무엇인지 생각해 보자.

## ② DAO의 분리

### ■ 첫 번째 접근 : 관심사의 분리

- 객체지향의 세계에서는 모든 것이 변한다. 여기서 변한다는 것은 변수나 오브젝트 필드의 값이 변한다는 게 아니다. 오브젝트에 대한 설계와 이를 구현한 코드가 변한다는 뜻이다.
- 객체지향의 설계와 프로그래밍이 이전의 절차적 프로그래밍 패러다임에 비해 초기에 좀 더 많은 번거로운 작업을 요구하는 이유는 객체지향 기술 자체가 지니는 변화에 효과적으로 대처할 수 있다는 기술적인 특징 때문이다.
- **프로그램의 기법 중에 관심사의 분리(Seperation of Concern)이라는 게 있다.** 이는 관심이 같은 것끼리는 하나의 객체 안으로 또는 친한 객체로 모이게 하고 관심이 다른 것은 가능한 따로 떨어져서 **서로 영향을 주지 않도록 분리하는 것**을 말한다.

### ■ UserDAO의 관심사항

- ① 첫째는 DB와 연결을 위한 커넥션을 어떻게 가져올까라는 관심
- ② 둘째는 사용자 등록을 위해 DB에게 보낼 SQL 문장을 담은 Statement를 만들고 실행하는 것이다.
- ③ 셋째는 작업이 끝나면 사용한 리소스인 Statement와 Connection을 닫아줘서 소중한 공유 리소스를 시스템에게 돌려주는 것이다.

※ 위의 이론을 바탕으로 가장먼저 할 일은 커넥션을 가져오는 것이다. 따라서 커넥션을 가져오는 중복된 코드를 분리하자.

```
1 public void add(UserVO user) throws ClassNotFoundException, SQLException {
2     Connection c = getConnection();
3     ...
4 }
5 public User get(String id) throws ClassNotFoundException, SQLException {
6     Connection c = getConnection();
7     ...
8 }
9 private Connection getConnection() throws ClassNotFoundException, SQLException{
10     Class.forName("oracle.jdbc.driver.OracleDriver");
11     Connection c = DriverManager.getConnection(
12         "jdbc:oracle:thin:@localhost:1521/XEPDB1", "scott", "tiger");
13     return c;
14 }
```

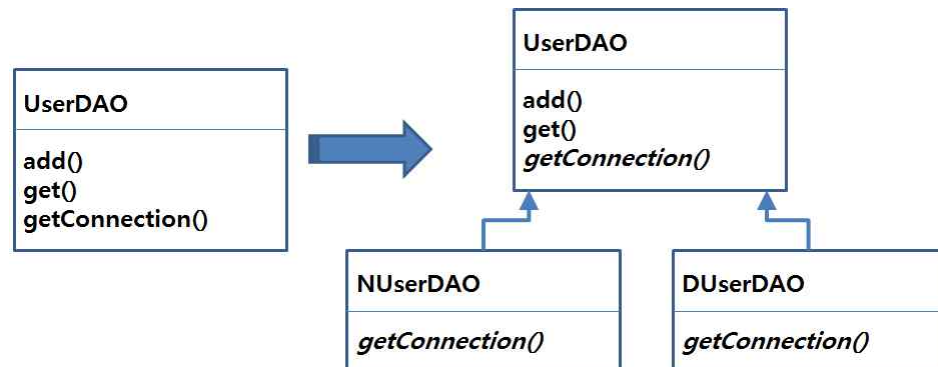
- 위와 같은 작업은 **기능에는 전혀 영향을 주지 않으면서 코드의 구조만 변경**시킨다.
- 이런 작업을 **리팩토링(ReFactoring)**이라고 한다. getConnection()이라고 하는 공통기능을 담당하는 메서드로 중복코드를 뽑아내서 사용하였다. 이와 같은 기법을 리팩토링에서는 **메서드 추출기법**이라고 한다.

## ③ DB 커넥션 만들기의 독립

- 만약 위에서 만든 DAO가 발전을 거듭해서 여러 회사에 구매주문이 들어왔다고 가정하자. 그런데 A사는 Oracle을 사용하고 있고, B사는 mySQL을 사용하고 있다고 하면 어떻게 해야 할까?
- 이럴 때는 기존의 UserDAO를 한 단계 더 분리하면 된다. (상속을 통한 확장)

□ 즉 우리가 만든 UserDao에서 메서드 구현 코드를 제거하고 getConnection()을 추상 메서드로 만들어 놓으면 된다.

※ 아래의 그림과 같이 구조를 변경 후 아래의 코드처럼 각 회사에 판매하고 상속해서 각각 getConnection()을 구현하게 한다.



```

1 public abstract class UserDao {
2     public void add(UserVO user) throws ClassNotFoundException, SQLException {
3         Connection c = getConnection();
4         ...
5     }
6     public User get(String id) throws ClassNotFoundException, SQLException {
7         Connection c = getConnection();
8         ...
9     }
10    public abstract Connection getConnection()
11                                   throws ClassNotFoundException, SQLException;
12 }
13 public class NUserDAO extends UserDao{
14     public Connection getConnection() throws ClassNotFoundException, SQLException{
15         //N사의 DB에 맞는 Connection 생성코드
16     }
17 }
18 public class DUserDAO extends UserDao{
19     public Connection getConnection() throws ClassNotFoundException, SQLException{
20         //D사의 DB에 맞는 Connection 생성코드
21     }
22 }

```

## ■ 참고

④ 디자인 패턴 : 소프트웨어 설계 시 특정 상황에서 자주 만나는 문제를 해결하기 위해 사용할 수 있는 재사용 가능한 솔루션을 말한다.

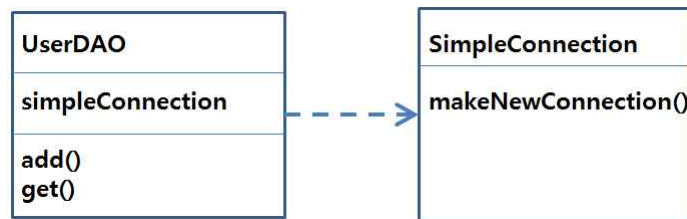
⑤ 템플릿 메서드 패턴 : 상속을 통해 슈퍼클래스의 기능을 확장할 때 사용하는 방법. 슈퍼클래스에서 디폴트 기능을 정의해 두거나 비워두었다가 서브클래스에서 선택적으로 오버라이드 할 수 있게 만들어둔 메서드를 훅(Hook) 메서드라고 한다.

© 팩토리 메서드 패턴 : 상속을 통해 기능을 확장하게 하는 패턴. 슈퍼클래스 코드에서는 서브클래스에서 구현할 메서드를 호출해서 필요한 타입의 오브젝트를 가져와 사용한다. 이때 주로 인터페이스 타입으로 리턴을 한다.

※ 이러한 패턴을 적용했을 때의 단점은 상속을 사용했다는 것이다. 첫째로 자바는 다중상속을 허용하지 않는다. 둘째로 두 클래스간의 관계가 밀접하다는 것이다.

#### ④ DAO의 확장

■ 두 개의 독립된 클래스로 분리

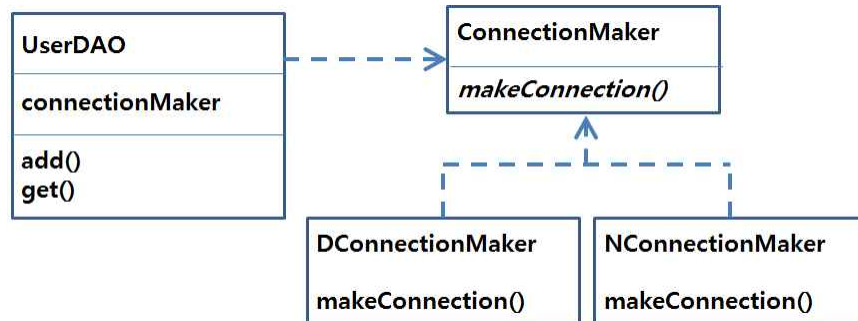


```

1 public class UserDAO {
2     private SimpleConnection simpleConnection;
3     public UserDAO(){
4         simpleConnection = new SimpleConnection();
5     }
6     public void add(UserVO user) throws ClassNotFoundException, SQLException {
7         Connection c = simpleConnection.makeNewConnection();
8         ...
9     }
10    public User get(String id) throws ClassNotFoundException, SQLException {
11        Connection c = simpleConnection.makeNewConnection();
12        ...
13    }
14 }
15 public class SimpleConnection{
16     public Connection makeNewConnection()
17         throws ClassNotFoundException, SQLException{
18         Class.forName("oracle.jdbc.driver.OracleDriver");
19         Connection c = DriverManager.getConnection(
20             "jdbc:oracle:thin:@localhost:1521/XEPDB1", "scott", "tiger");
21         return c;
22     }
23 }
  
```

□ 위와 같이 클래스를 분리한 경우에도 상속과 마찬가지로 자유로운 확장이 가능하려면 두 가지 문제점이 생긴다. 첫째는 SimpleConnection의 메서드문제. 두 번째는 DB커넥션을 제공하는 클래스가 어떤 것인지를 UserDAO가 구체적으로 알고 있어야 한다는 문제이다.

- 이와 같은 문제를 해결하기 위해서 추상적인 느슨한 연결고리를 만들어 주자. **추상화란 어떤 것들의 공통적인 성격을 뽑아내서 이를 따로 분리해내는 작업**이다.



```

1 public interface ConnectionMaker{
2     public abstract Connection makeConnection()
3     throws ClassNotFoundException, SQLException;
4 }
5 public class NUserDAO implements ConnectionMaker{
6     public Connection makeConnection() throws ClassNotFoundException, SQLException{
7         //N사의 DB에 맞는 Connection 생성코드
8     }
9 }
10 public class UserDAO {
11     private ConnectionMaker connectionMaker;
12     public UserDAO(){
13         connectionMaker = new NConnectionMaker();
14     }
15     public void add(UserVO user) throws ClassNotFoundException, SQLException {
16         Connection c = connectionMaker.makeConnection();
17         ...
18     }
19     public User get(String id) throws ClassNotFoundException, SQLException {
20         Connection c = connectionMaker.makeConnection();
21         ...
22     }
23 }
  
```

※ 거의 완벽하나 아직 한 가지 클래스 이름에 오브젝트를 만들지 않고 어떻게 사용하지?

#### ⑤ 관계설정과 책임의 분리

- `new NConnectionMaker()`라는 코드는 매우 짧고 간단하지만 그 자체로 충분한 관심사를 담고 있다. 그것은 바로 UserDAO가 어떤 ConnectionMaker 구현 클래스의 오브젝트를 이용하게 할지를 결정하는 것이다. 다시 말하면 UserDAO와 UserDAO가 사용할 ConnectionMaker의 특정 구현 클래스 사이의 관계를 설정해주는 것에 관한 관심이다.
- 오브젝트 사이에 관계가 만들어 지려면 일단 만들어진 오브젝트가 있어야 하는데 이처럼 직접 생성자를 호출해서 오브젝트를 만드는 방법도 있지만 외부에서 만들어진 것을 가져오는 방법도 있다.

□ 외부에서 만든 오브젝트를 전달받으려면 메서드 파라미터나 생성자 파라미터를 이용하면 된다.

1	public UserDao(ConnectionMaker connectionMaker){
2	this.connectionMaker = connectionMaker;
3	}

※ 이렇게 하면 클래스 사이에 관계가 만들어진 것은 아니고 단지 **오브젝트 사이에 다이내믹한 관계가 만들어지는 것이다.**

#### ⑥ 객체지향 프로그래밍 이론

##### ㉠ 개방 폐쇄 원칙 (OCP : Open-Closed Principle)

- 클래스나 모듈은 확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.
- 인터페이스를 사용해 확장 기능을 정의한 대부분의 API는 바로 이 개방 폐쇄 원칙을 따른다고 볼 수 있다.

##### ㉡ 객체지향 설계의 원칙 (SOLID)

- SRP(The Single Responsibility Principle) : 단일 책임 원칙
- OCP(The Open Closed Principle) : 개방 폐쇄 원칙
- LSP(The Liskov Substitution Principle)) : 리스코프 치환 원칙
  - 참조되는 기반클래스의 함수는 파생클래스 객체의 상세를 알지 않고서도 사용될 수 있어야 한다.
- ISP(The Interface Segregation Principle) : 인터페이스 분리 원칙
- DIP(The Dependency Inversion Principle) : 의존 관계 역전 원칙

##### ㉢ 높은 응집도와 낮은 결합도

- 응집도가 높다는 건 하나의 모듈, 클래스가 하나의 책임 또는 관심사에만 집중되어 있다는 뜻이다.
- 책임과 관심사가 다른 오브젝트 또는 모듈과는 낮은 결합도 즉 느슨하게 연결된 형태를 유지하는 것이 바람직하다.

##### ㉣ 전략패턴

- 전략패턴은 자신의 기능 맥락(Context)에서 필요에 따라 변경이 필요한 알고리즘을 인터페이스를 통해 통째로 외부로 분리시키고 이를 구현한 구체적인 알고리즘 클래스를 필요에 따라 바꿔서 사용할 수 있게 하는 디자인 패턴이다.

※ 결론 : 이러한 이론을 바탕으로 앞에서 우리는 UserDao를 개선했던 것이다.

## 2. 프레임워크의 개념

### ① 프레임워크의 개요

프레임워크의 사전적인 의미는 뼈대 혹은 틀로서 이 의미를 소프트웨어 과정에서 접근하면 아키텍처에 해당하는 골격코드이다. 여기서의 핵심단어는 “아키텍처”와 “골격코드”이다. 애플리케이션을 개발할 때 가장 중요한 것은 전체 애플리케이션의 구조를 결정하는 아키텍처인데 이 아키텍처에 해당하는 골격코드를 프레임워크가 제공한다.

시스템을 개발하는 과정에서 대부분 개발자들은 산출물에 입각해서 개발하므로 아키텍처의 일관성이 잘 유지된다. 하지만 유지보수 과정에서 인력과 시간부족으로 인해 산출물은 무시되고 쉽고, 개발자들의 경험에 의존하여 유지보수가 진행되는 경우가 많다. 이러한 문제를 프레임워크는 근본적으로 해결해 줄 수 있다. 개발자에게 모든 것을 위임하는 것이 아니라 애플리케이션의 기본 아키텍처는 프레임워크가 제공하고, 그 뼈대에 살을 붙이는 작업만 개발자가 하는 것이다.

### ② 프레임워크의 장점

- 빠른 구현 시간
- 쉬운 관리
- 개발자들의 역량 획일화 : 프레임워크를 사용하면 숙련된 개발자와 초급 개발자가 생성한 코드가 비슷해진다.
- 검증된 아키텍처의 재사용과 일관성 유지

### ③ 자바 기반 프레임워크

처리영역	프레임워크	설명
Presentation	Struts	Struts 프레임워크는 UI Layer에 중점을 두고 개발된 MVC 프레임워크이다.
	Spring (MVC)	Struts와 동일하게 MVC 아키텍처를 제공하는 UI Layer 프레임워크이다. 하지만 Struts처럼 독립된 프레임워크는 아니고 Spring 프레임워크에 포함되어 있다.
Business	Spring(IoC, AOP)	Spring은 컨테이너 성격을 가지는 프레임워크이다. Spring의 IoC와 AOP 모듈을 이용하여 Spring 컨테이너에서 동작하는 엔터프라이즈 비즈니스 컴포넌트를 개발할 수 있다.
Persistence	Hibernate or JPA	Hibernate는 완벽한 ORM 프레임워크이다. ORM 프레임워크는 SQL 명령어를 프레임워크가 자체적으로 생성하여 DB연동을 처리한다. JPA는 Hibernate를 비롯한 모든 ORM의 공통 인터페이스를 제공하는 자바 표준 API이다.
	MyBatis	MyBatis 프레임워크는 개발자가 작성한 SQL 명령어와 자바 객체(VO)를 매핑해주는 기능을 제공하며 기존에 사용하던 SQL 명령어를 재사용하여 개발하는 차세대 프로토콜에 유용하게 적용할 수 있다.

#### ④ 스프링 프레임워크의 특징

##### ☐ 경량(Lightweight) 프레임워크

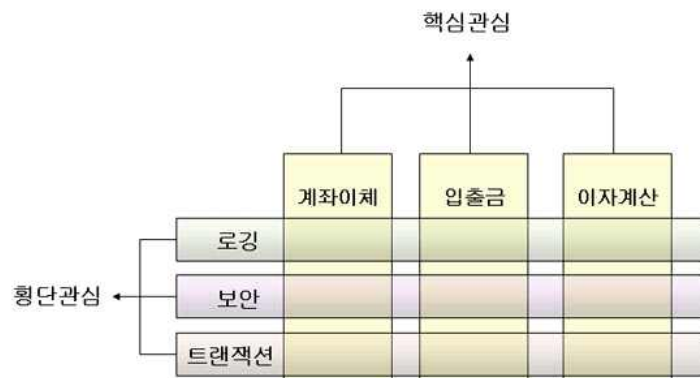
우선 스프링 프레임워크는 크기 측면에서 가볍다. 그리고 스프링 프레임워크는 POJO 형태의 객체를 관리한다.

##### ☐ 제어의 역행(Inversion of Control)

우리가 비즈니스 컴포넌트를 개발할 때 항상 신경 쓰는 것이 낮은 결합도와 높은 응집도이다. 스프링의 제어의 역행을 통해 애플리케이션을 구성하는 객체 간에 느슨한 결합 즉 낮은 결합도를 유지할 수 있다.

##### ☐ 관심지향 프로그래밍(Aspects Oriented Programming, AOP)

공통 기능을 분리함으로써 응집도가 높은 프로그램을 개발할 수 있도록 한다.



##### ☐ 컨테이너(Container)

애플리케이션의 운용에 필요한 객체를 생성하고 객체 간의 의존관계를 관리한다는 점에서 스프링도 일종의 컨테이너라고 할 수 있다.

### 3. IoC(Inversion of Control) 컨테이너

#### ① 컨테이너의 개념

스프링 프레임워크를 이해하는데 가장 중요한 개념이 컨테이너이다. 컨테이너라는 개념은 서블릿이나 EJB 기술에서 이미 사용해왔다. 따라서 간단한 서블릿 프로그램을 통해서 스프링 컨테이너의 동작 방식을 유추해 보자.

■ 실습을 위해 앞에서 작성했던 myfirst 프로젝트에 다음과 같은 예제를 작성해 보자.

☐ src/tommy/spring/servlet/HelloServlet.java

```
1 package tommy.spring.servlet;
2 import java.io.IOException;
3 import javax.servlet.ServletException;
4 import javax.servlet.http.HttpServlet;
5 import javax.servlet.http.HttpServletRequest;
6 import javax.servlet.http.HttpServletResponse;
7 public class HelloServlet extends HttpServlet {
8     private static final long serialVersionUID = 1L;
9     public HelloServlet() {
10         super();
11         System.out.println("HelloServlet 객체 생성");
12     }
```



```

13     protected void doGet(HttpServletRequest request, HttpServletResponse response)
14         throws ServletException, IOException {
15         System.out.println("doGet() 메서드 호출");
16     }
17 }

```

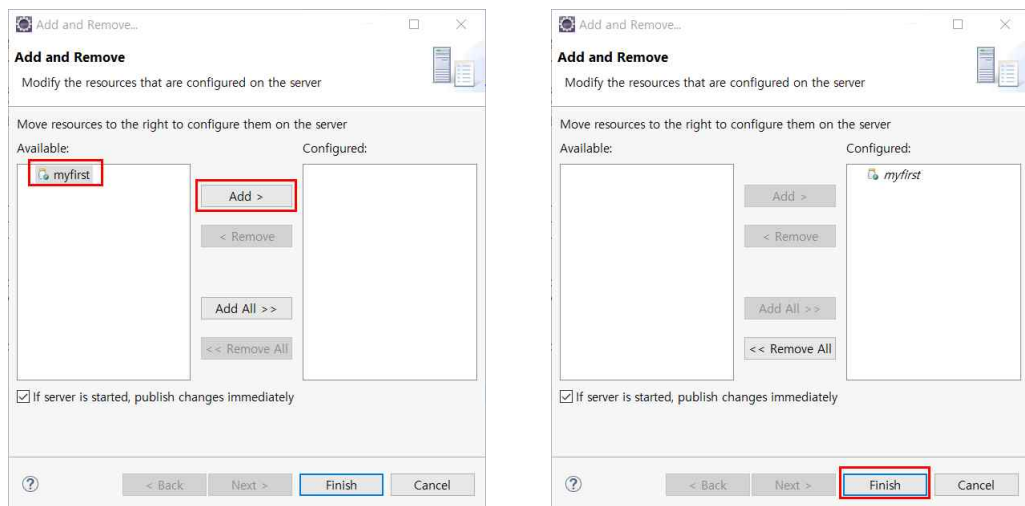
- ☐ web.xml 파일에 아래와 같은 내용을 추가하자.

```

1     <!-- 상단부분 생략 -->
2     </welcome-file-list>
3     <servlet>
4         <servlet-name>hello</servlet-name>
5         <servlet-class>tommy.spring.servlet.HelloServlet</servlet-class>
6     </servlet>
7     <servlet-mapping>
8         <servlet-name>hello</servlet-name>
9         <url-pattern>/hello.do</url-pattern>
10    </servlet-mapping>
11 </web-app>

```

- ☐ 실행을 위하여 하단의 [Servers] 탭에서 server를 선택하고 우 클릭하여 [Add and Remove...] 메뉴를 선택하여 아래와 같이 프로젝트를 서버에 추가해 주자.



- ☐ 서버를 실행한 후 브라우저를 열고 <http://localhost:8080/myfirst/hello.do>라고 입력해서 실행해 보자. 아래와 같은 결과화면을 볼 수 있을 것이다.

```

Markers Properties Servers Data Source Explorer Snippets Console
Tomcat v9.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre1.8.0_221\bin\W
정보: 프로토콜 핸들러 ["ajp-nio-8009"]을(를) 시작합니다.
9월 06, 2019 4:26:39 오전 org.apache.catalina.startup.Catalina start
정보: 서버가 [1,097] 밀리초 내에 시작되었습니다.
HelloServlet 객체 생성
doGet() 메서드 호출

```

- 위의 예제에서 주목할 부분은 서블릿 코드 어디를 살펴봐도 객체를 생성하는 부분이 존재하지 않는다는 것이다. 객체를 생성해야 가지고 있는 doGet() 메서드도 호출할 것인데 아무리 찾아보아도 객체를 생성하는 코드가 존재하지 않는다. 그렇다면 도대체 누가 서블릿 객체를 생성했으며 doGet() 메서드를 호출하였는가? 정답은 바로 서블릿 컨테이너(Tomcat)이다.

■ 참고 : 서블릿 컨테이너의 동작순서

- WEB-INF/web.xml 파일을 로딩하여 구동
- 브라우저로부터 /hello.do 요청을 수신
- tommy.spring.servlet.HelloServlet 클래스를 찾아서 객체를 생성하고 doGet() 메서드 호출
- doGet() 메서드의 실행 결과를 브라우저로 전송

② 제어의 역행 : 결합도와 관련

기존의 자바기반의 애플리케이션을 개발할 때 객체를 생성하고 객체들 사이의 의존관계를 처리하는 것은 전적으로 개발자에게 있었다. 즉 개발자가 어떤 객체를 생성할지 판단하고 객체 간의 의존관계 역시 소스코드로 표현해야 하는 것이다.

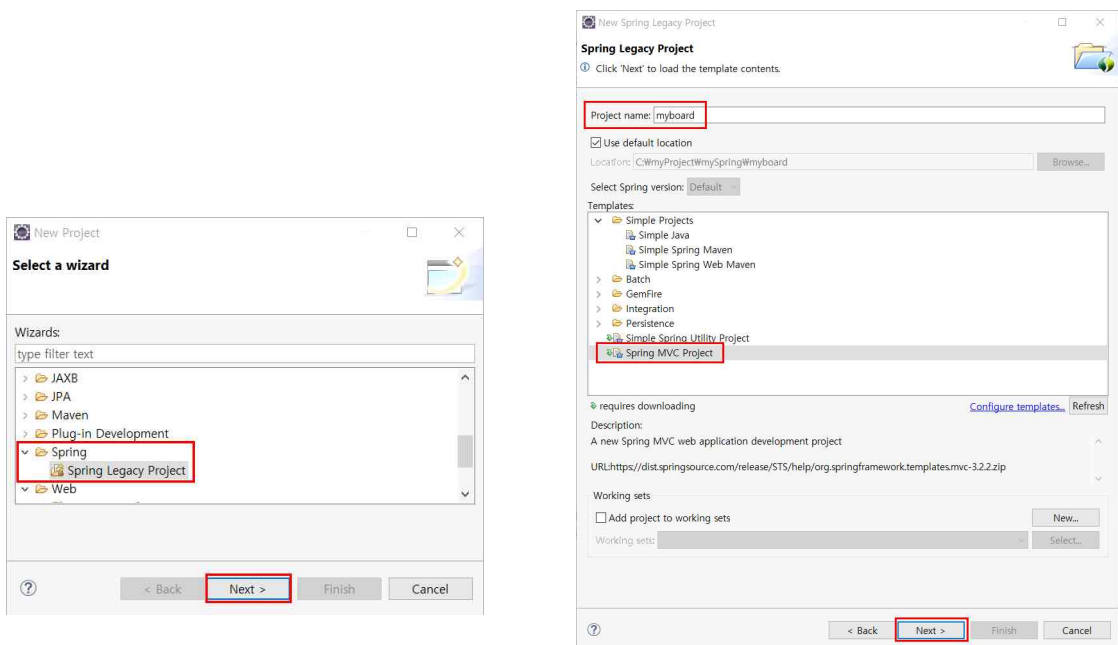
하지만 제어의 역행이라는 것은 이런 일련의 작업들을 소스코드로 처리하지 않고 컨테이너에서 처리하는 것을 의미한다.

따라서 제어의 역행을 이용하면 소스코드에서 객체의 생성과 의존관계에 대한 코드가 사라져서 결과적으로 낮은 결합도의 컴포넌트를 구현할 수 있게 된다.

③ 결합도(Coupling)가 높은 프로그램

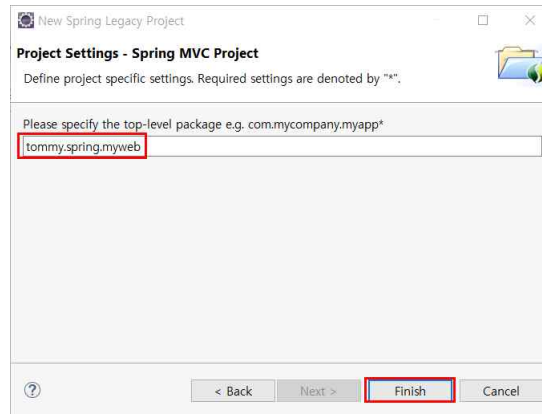
- 결합도란 하나의 클래스가 다른 클래스와 얼마나 많이 연결되어 있는지를 나타낸다. 결합도가 높은 프로그램은 유지보수하기가 어렵다.

- 실습을 위하여 myboard라는 SpringMVC 프로젝트를 생성하자. [File]-[New]-[Spring Legacy Project]를 선택한다. 이어서 나오는 화면에서 “Project Name”에 “myboard”라고 프로젝트명을 입력하고 아래에 “Spring MVC project”를 선택한 후 [Next] 버튼을 클릭한다.





- 최상위 패키지를 지정하는 화면이 나오면 패키지 경로에 최소 세 개 이상의 패키지를 지정하도록 한다. 아래와 같이 “tommy.spring.myweb” 이라고 지정하고 [Finish] 버튼을 클릭한다.

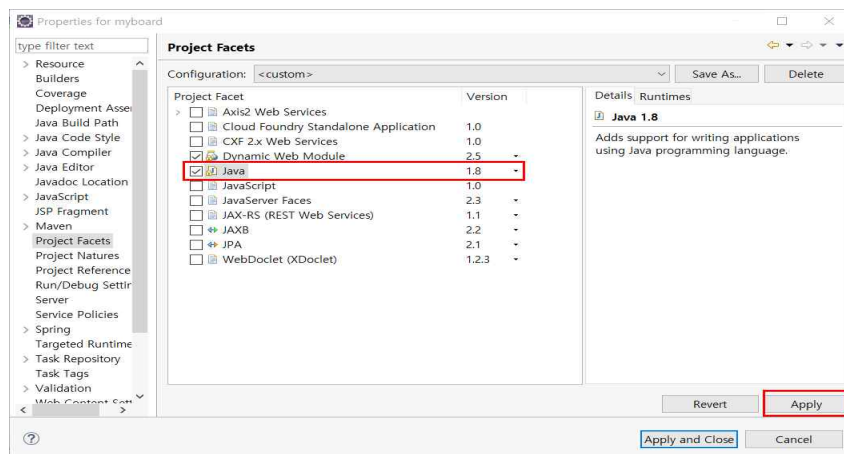


- 참고 : 위에서 입력한 tommy.spring.myweb 패키지에서 마지막 myweb이 Spring MVC에서는 ContextPath가 된다.

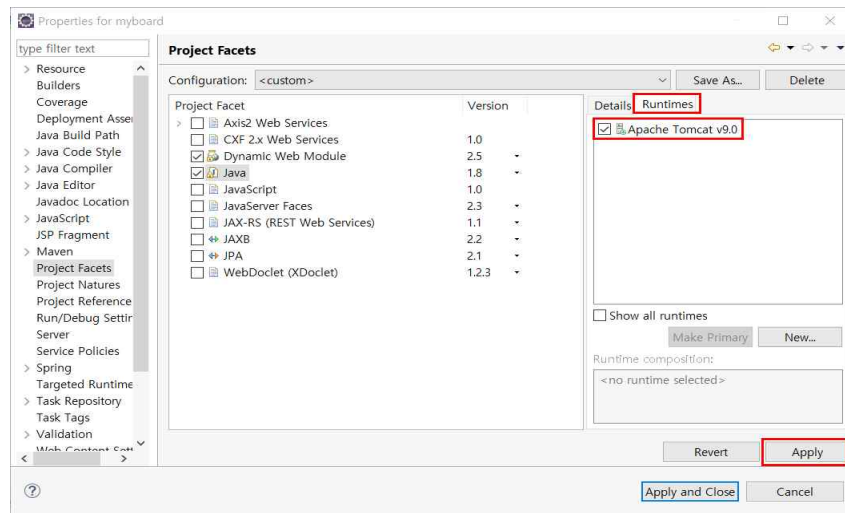
#### □ 프로젝트 설정변경

STS를 이용하여 “Spring MVC Project” 를 생성하면 JRE 버전도 맞지 않고 서버 라이브러리도 등록되어 있지 않으므로 이를 적절하게 변경해야 한다.

- 우선 myboard 프로젝트를 선택하고 우 클릭 하여 제일 마지막에 [Properties]를 선택한다. 왼쪽의 메뉴에서 [Project Facets]를 선택하고 Java 버전을 1.8로 수정한다.



- 그리고 오른쪽에 [Runtimes] 탭을 선택하고 “Apache Tomcat v9.0”을 체크한 후에 [Apply]를 클릭한다.



- 프로젝트가 생성되고 나면 스프링 기반의 웹 프로젝트 수행에 필요한 기본설정 파일들과 소스코드들이 자동으로 생성된다. 실습에 불필요한 파일들을 정리하자.
  - src/main/java 폴더 안에 내용을 모두 삭제한다.
  - src/main/resources 폴더에 log4j.xml 파일을 제외한 나머지는(패키지, 폴더, 파일들) 모두 삭제 하도록 한다.
  - src/main/webapp/WEB-INF 폴더 아래에 spring과 views 폴더도 삭제한다.
  - src/main/webapp/WEB-INF/web.xml 파일을 열어서 <web-app> 루트 엘리먼트를 제외한 나머지 설정은 모두 삭제한다.
  - pom.xml 파일을 열어서 Spring 버전을 최신버전으로 맞춘다.
  - pom.xml 파일의 junit 버전을 4.12로 맞춘다.

- 실습을 위하여 src/main/java 폴더에 tommy.spring.polymorphism.SamsungTV.java를 작성한다.

```

1 package tommy.spring.polymorphism;
2 public class SamsungTV {
3     public void powerOn() {
4         System.out.println("SamsungTV 전원을 켜다.");
5     }
6     public void powerOff() {
7         System.out.println("SamsungTV 전원을 끈다.");
8     }
9     public void volumeUp() {
10        System.out.println("SamsungTV 볼륨을 올린다.");
11    }
12    public void volumeDown() {
13        System.out.println("SamsungTV 볼륨을 내린다.");
14    }
15 }

```

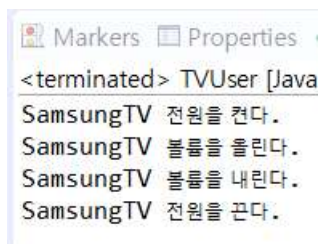
- 같은 폴더에 LgTV 클래스도 작성한다.

```
1 package tommy.spring.polymorphism;
2 public class LgTV {
3     public void turnOn() {
4         System.out.println("LgTV 전원을 켜다.");
5     }
6     public void turnOff() {
7         System.out.println("LgTV 전원을 끈다.");
8     }
9     public void soundUp() {
10        System.out.println("LgTV 볼륨을 올린다.");
11    }
12    public void soundDown() {
13        System.out.println("LgTV 볼륨을 내린다.");
14    }
15 }
```

- SamsungTV와 마찬가지로 LgTV 클래스도 같은 기능을 수행하는 메서드가 있지만 서로 메서드 이름이 다르다. 이제 이 두 TV클래스를 번갈아 사용하는 TVUser 프로그램을 구현해 보자.

```
1 package tommy.spring.polymorphism;
2 public class TVUser {
3     public static void main(String[] args) {
4         SamsungTV tv = new SamsungTV();
5         tv.powerOn();
6         tv.volumeUp();
7         tv.volumeDown();
8         tv.powerOff();
9     }
10 }
```

#### ■ 실행결과

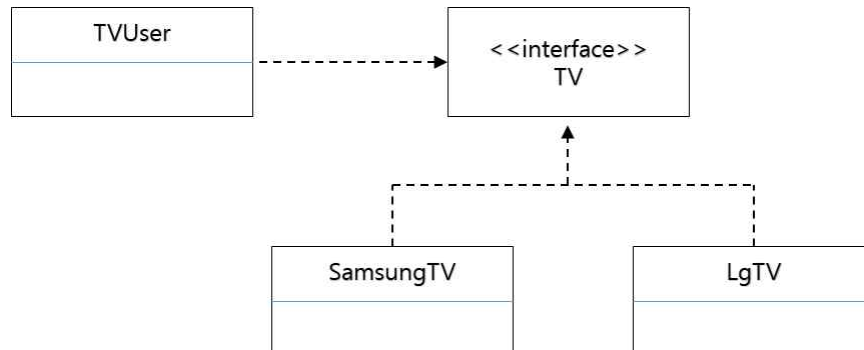


```
Markers Properties
<terminated> TVUser [Java]
SamsungTV 전원을 켜다.
SamsungTV 볼륨을 올린다.
SamsungTV 볼륨을 내린다.
SamsungTV 전원을 끈다.
```

- 이제 SamsungTV를 시청하는 TVUser를 LgTV를 시청하는 프로그램으로 수정해 보자. SamsungTV와 LgTV는 메서드 시그니처가 다르므로 TVUser 클래스 코드 대부분을 수정해야 프로그램을 수정할 수 있다. 이러한 상황이라면 유지보수가 매우 힘들 것이다.

#### ④ 다형성 이용하기

- ☐ 결함도를 낮추기 위해서 다양한 방법을 사용할 수 있지만 객체지향 언어의 핵심 개념인 다형성을 이용해 보자.



- ☐ 위의 구조와 같이 프로그램을 변경하기 위하여 TV라는 인터페이스를 작성한다.

```

1 package tommy.spring.polymorphism;
2 public interface TV {
3     public void powerOn();
4     public void powerOff();
5     public void volumeUp();
6     public void volumeDown();
7 }
    
```

- ☐ 이제 SamsungTV와 LgTV 클래스가 위에서 작성한 TV 인터페이스를 상속받도록 수정하자.

```

1 package tommy.spring.polymorphism;
2 public class SamsungTV implements TV {
3     public void powerOn() {
4         System.out.println("SamsungTV 전원을 켜다.");
5     }
6     public void powerOff() {
7         System.out.println("SamsungTV 전원을 끈다.");
8     }
9     public void volumeUp() {
10        System.out.println("SamsungTV 볼륨을 올린다.");
11    }
12    public void volumeDown() {
13        System.out.println("SamsungTV 볼륨을 내린다.");
14    }
15 }
    
```

```

1 package tommy.spring.polymorphism;
2 public class LgTV implements TV {
3     public void powerOn() {
4         System.out.println("LgTV 전원을 켜다.");
5     }
6 }
    
```

```

5      }
6      public void powerOff() {
7          System.out.println("LgTV 전원을 끈다.");
8      }
9      public void volumeUp() {
10         System.out.println("LgTV 볼륨을 올린다.");
11     }
12     public void volumeDown() {
13         System.out.println("LgTV 볼륨을 내린다.");
14     }
15 }

```

■ 위와 같이 수정함으로써 인터페이스를 이용하여 모든 TV 클래스가 같은 메서드를 가질 수밖에 없도록 강제할 수 있게 되었다.

□ TVUser 클래스를 아래와 같이 변경해 보고 실행해 보자.

```

1 package tommy.spring.polymorphism;
2 public class TVUser {
3     public static void main(String[] args) {
4         TV tv = new SamsungTV(); // new LgTV(); 로 수정해서 실행해 보자.
5         tv.powerOn();
6         tv.volumeUp();
7         tv.volumeDown();
8         tv.powerOff();
9     }
10 }

```

■ 결론 : 이렇게 다형성을 이용하면 TVUser와 같은 클라이언트 프로그램이 여러 개 있더라도 최소한의 수정으로 TV를 교체할 수 있다. 따라서 **유지보수가 좀 더 편해졌다고 할 수 있다.**

## ⑤ 디자인 패턴 이용하기

□ 결합도를 낮추기 위한 다른 방법으로 디자인 패턴을 이용하는 방법이 있다. TV를 교체할 때 클라이언트 소스코드를 수정하지 않고 교체할 수 있다면 유지보수는 더욱 편리할 것이다. 이를 위해서 Factory 패턴을 적용해야 하는데 Factory 패턴은 클라이언트에서 사용할 객체 생성을 캡슐화하여 TVUser와 TV사이를 느슨한 결합상태로 만들어 준다.

□ Factory 개념을 적용하여 BeanFactory 클래스를 아래와 같이 추가한다.

```

1 package tommy.spring.polymorphism;
2 public class BeanFactory {
3     public Object getBean(String beanName) {
4         if (beanName.equals("samsung")) {
5             return new SamsungTV();
6         } else if (beanName.equals("lg")) {
7             return new LgTV();
8         }
9     }
10 }

```

```

9      return null;
10     }
11 }

```

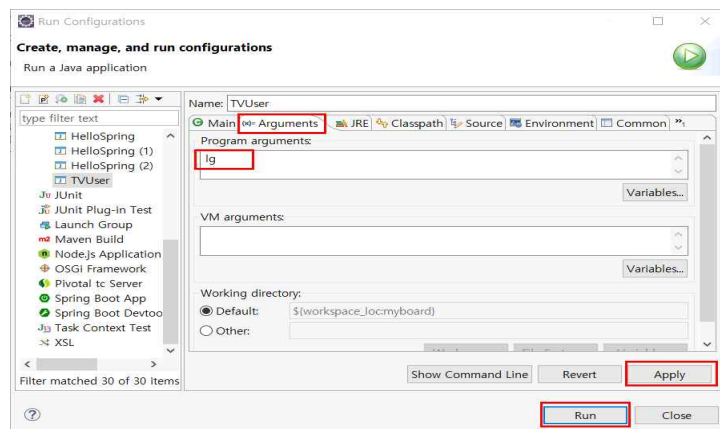
□ 이제 TVUser 클래스를 아래와 같이 변경해 보자.

```

1 package tommy.spring.polymorphism;
2 public class TVUser {
3     public static void main(String[] args) {
4         BeanFactory factory = new BeanFactory();
5         TV tv = (TV) factory.getBean(args[0]);
6         tv.powerOn();
7         tv.volumeUp();
8         tv.volumeDown();
9         tv.powerOff();
10    }
11 }

```

□ 실행 시 [Run As]-[Run Configuration]을 선택하여 [Arguments] 탭을 선택하고 “lg” 나 “samsung” 을 입력하고 [Run] 버튼을 클릭해 보자.



■ 결론 : 이제 실행되는 TV를 변경하고 싶을 때는 매개변수만 수정하여 실행하면 된다. 결국 클라이언트 소스코드를 수정하지 않고도 실행되는 객체(TV)를 변경할 수 있게 되었다.

```

Markers Properties Services
<terminated> TVUser [Java Applic
LgTV 전원을 켜다.
LgTV 볼륨을 올린다.
LgTV 볼륨을 내린다.
LgTV 전원을 끈다.

```

```

Markers Properties Services
<terminated> TVUser [Java
SamsungTV 전원을 켜다.
SamsungTV 볼륨을 올린다.
SamsungTV 볼륨을 내린다.
SamsungTV 전원을 끈다.

```