

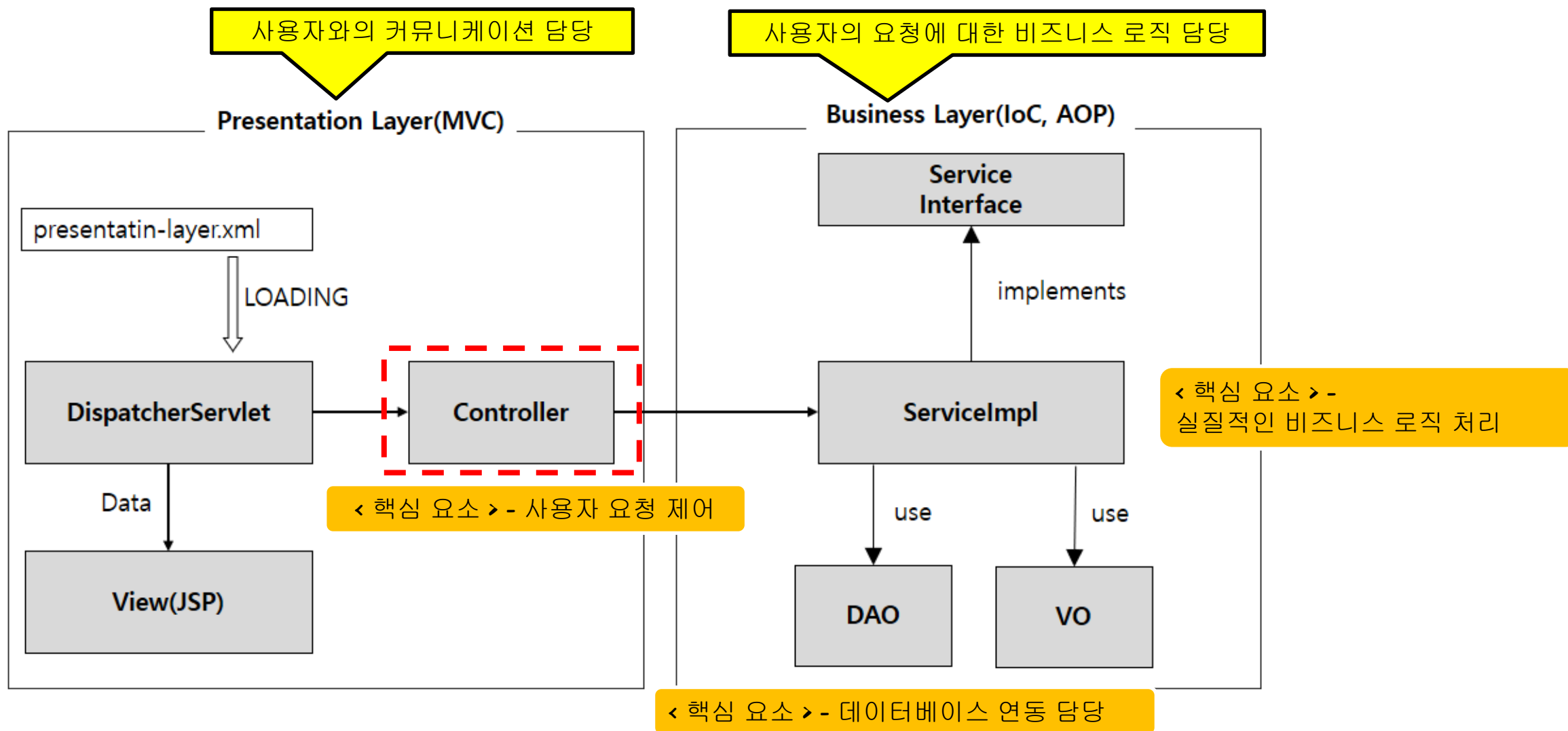
2학기

스프링프레임워크

[교재 Class06 비즈니스 컴포넌트 실습 1] p109~



Layered Architecture (교재 p107)



@Component를 상속하여 수행 역할별 분류

어노테이션	위치	의미
@Service	XXXServiceImpl	비즈니스 로직을 처리하는 Service 클래스
@Repository	XXXDAO	데이터베이스 연동을 처리하는 DAO 클래스
@Controller	XXXController	사용자 요청을 제어하는 Controller 클래스



Spring Layered architecture 구조

1) Presentation Layer

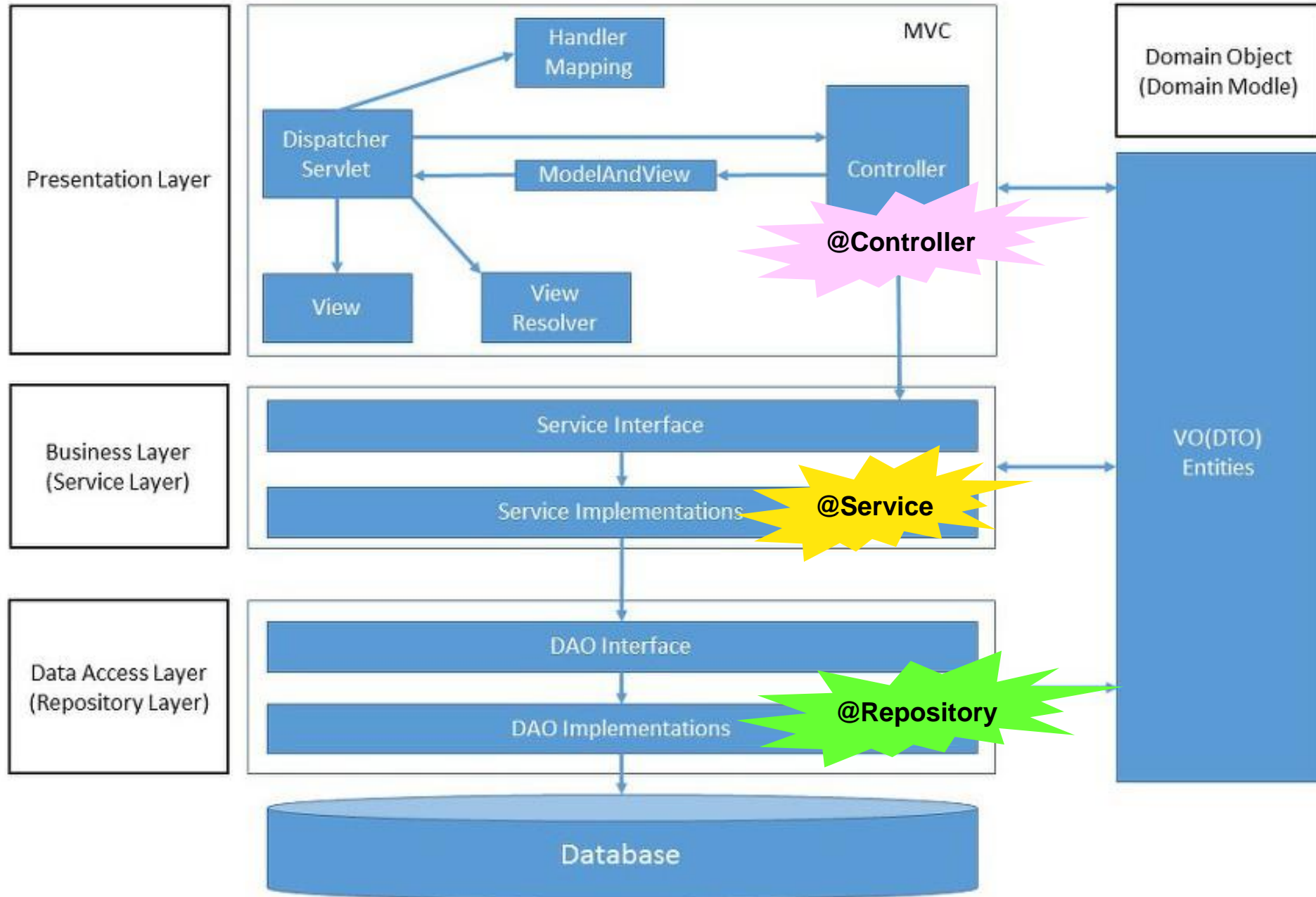
Spring MVC 객체를 말한다.
프론트 컨트롤러
(DispatcherServlet), 컨트롤러,
뷰, 모델이 포함된다.

2) Service Layer(Business Layer)

실제 비즈니스 로직을 수행하는
컴포넌트 컨트롤러(presentation
layer)에서 요청을 보내면
DAO(data access layer)를 이용
해 실제로 비즈니스 로직을 수
행한다.
보통 하나의 비즈니스 로직은
하나의 트랜잭션으로 동작
(ACID 특징 가짐)

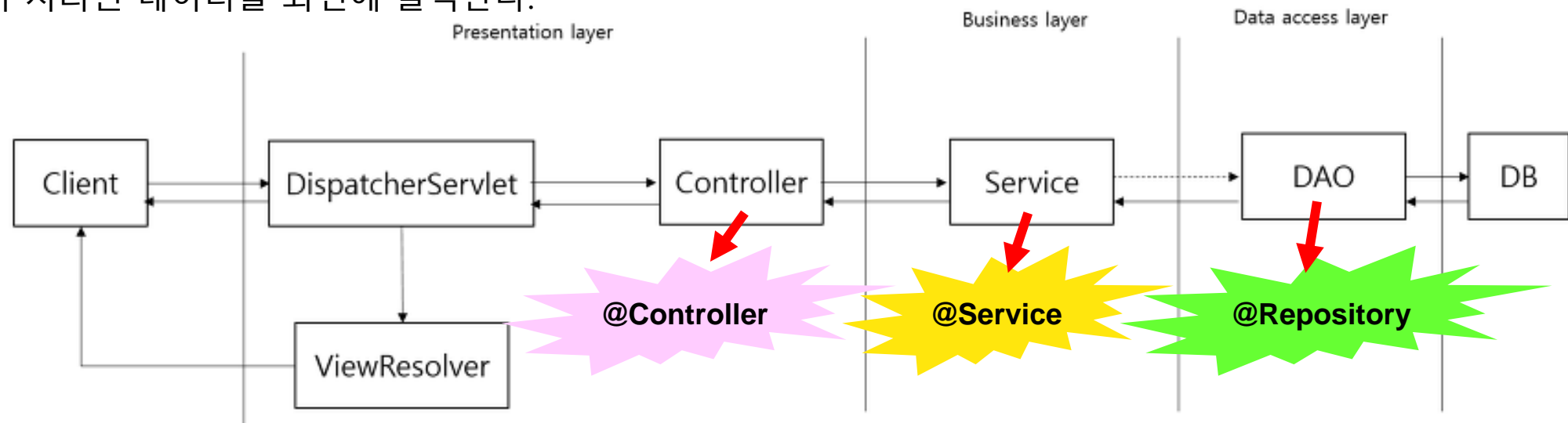
3) Data Access Layer(Repository Layer)

DB에 값을 저장하거나 가져오
기 위해 JDBC, Mybatis, JPA 등
을 사용해 구현한 DAO

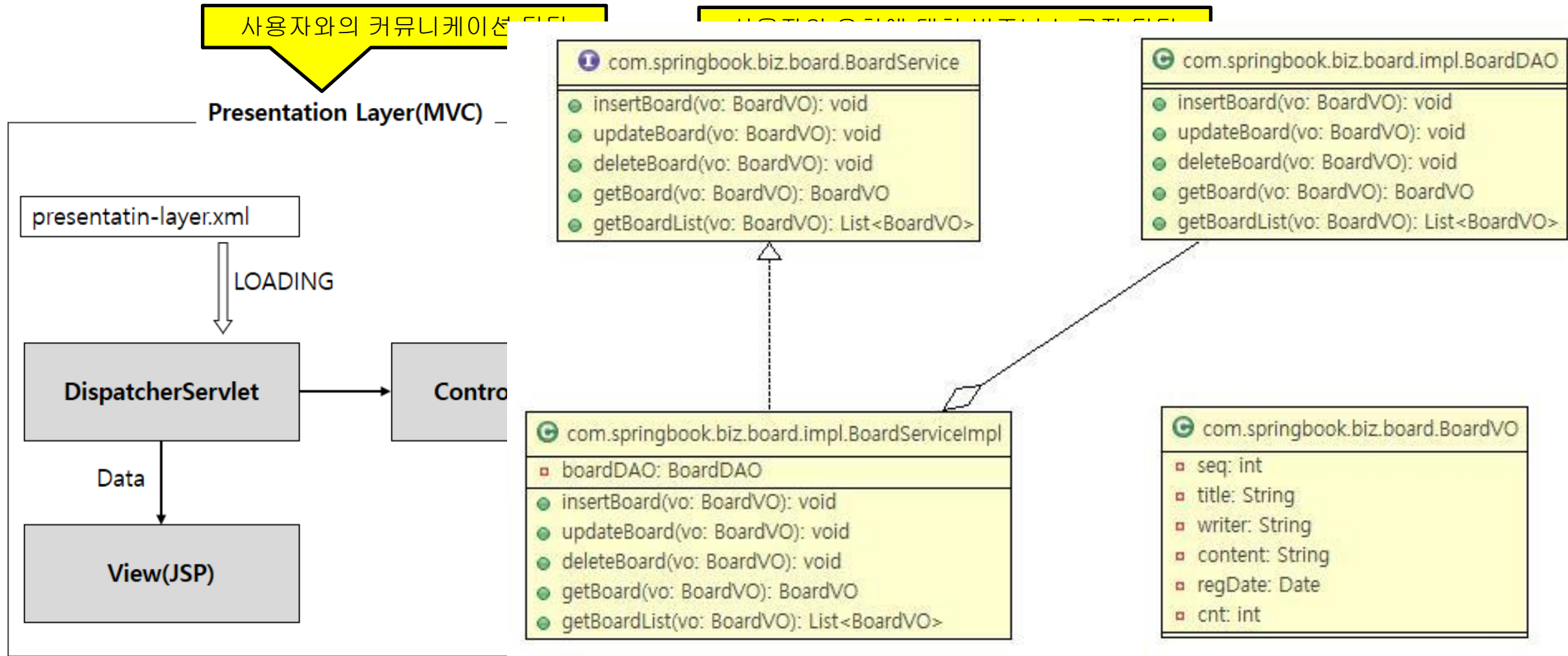


Spring Layered architecture 동작 과정

- ① Client 에서 요청이 들어오면 먼저 Presentation Layer 에서 DispatcherServlet 이 HandlerMapping 을 통해서 Controller 에게 Client 요청이 무엇인지 알리고, Controller 는 Client 요청을 처리를 Business Layer 에게 요구한다. 이 때, Business Layer 에 넘겨줄 데이터(Client 가 요청한 데이터)가 있으면 Domain Object 에 담는다.
- ② Business Layer 는 Presentation Layer 와 Interface 를 통해서 통신하며, Client 요청을 적절히 처리한 후 데이터베이스에 데이터를 저장하거나 데이터를 꺼내기 위해 Data Access Layer 에 요청한다. 이 때, 비즈니스 로직을 수행하기 위해 데이터가 필요하다면 Domain Object 에서 가져오고, Data Access Layer 넘겨줄 데이터가 있으면 Domain Object 에 담는다.
- ③ Data Access Layer 역시 Business Layer 와 Interface 를 통해서 통신하며, Business Layer 의 요청을 처리한다. 이 때, 데이터베이스에 저장하기 위해 필요한 데이터를 Domain Object 에서 가져오고, 데이터베이스에서 데이터를 가져와 반환할 데이터가 있다면 Domain Object 에 저장한다.
- ④ 모든 처리가 끝나면 Controller 는 Client 요청이 처리된 데이터와 사용할 View 정보를 Domain Object 에서 가져와서 ModelAndView에 담는다. 그리고 ModelAndView 객체를 DispatcherServlet 에 넘긴다.
- ⑤ ModelAndView 객체가 DispatcherServlet 에 전달되면, DispatcherServlet 은 ViewResolver 를 통해서 View 를 선택하고 Client 에게 요청이 처리된 데이터를 화면에 출력한다.



비즈니스 컴포넌트 실습 1 (교재 p109) : 게시판 구현



프로젝트명 : **SpringProject4_BizStep1** → 어노테이션 방식으로

1. BoardVO 클래스 작성

- * Oracle Database 설치
- * board테이블 작성 (구조는 다음 슬라이드에..)

2. BoardDAO 클래스 작성

3. BoardService 인터페이스 작성

4. BoardServiceImpl 클래스 작성

5. Spring 설정 파일(applicationContext.xml) 작성

6. BoardServiceClient 작성 및 테스트



비즈니스 컴포넌트 실습 1 : board 테이블 작성

```
create table board(  
    seq number(5) primary key,  
    title varchar(200),  
    writer varchar(30),  
    content varchar(2000),  
    regdate date default sysdate,  
    cnt number(5) default 0  
);
```



UserService 컴퍼넌트 구조 클래스 다이어그램 (교재 p129)

프로젝트명 : **SpringProject4_BizStep2** → DI를 Setter 방식으로

1. UserVO 클래스 작성
 - * users테이블 작성 (구조는 아래..)
2. UserDao 클래스 작성
3. UserService 인터페이스 작성
4. UserServiceImpl 클래스 작성
5. Spring 설정 파일(applicationContext.xml) 작성
6. UserServiceClient 작성 및 테스트

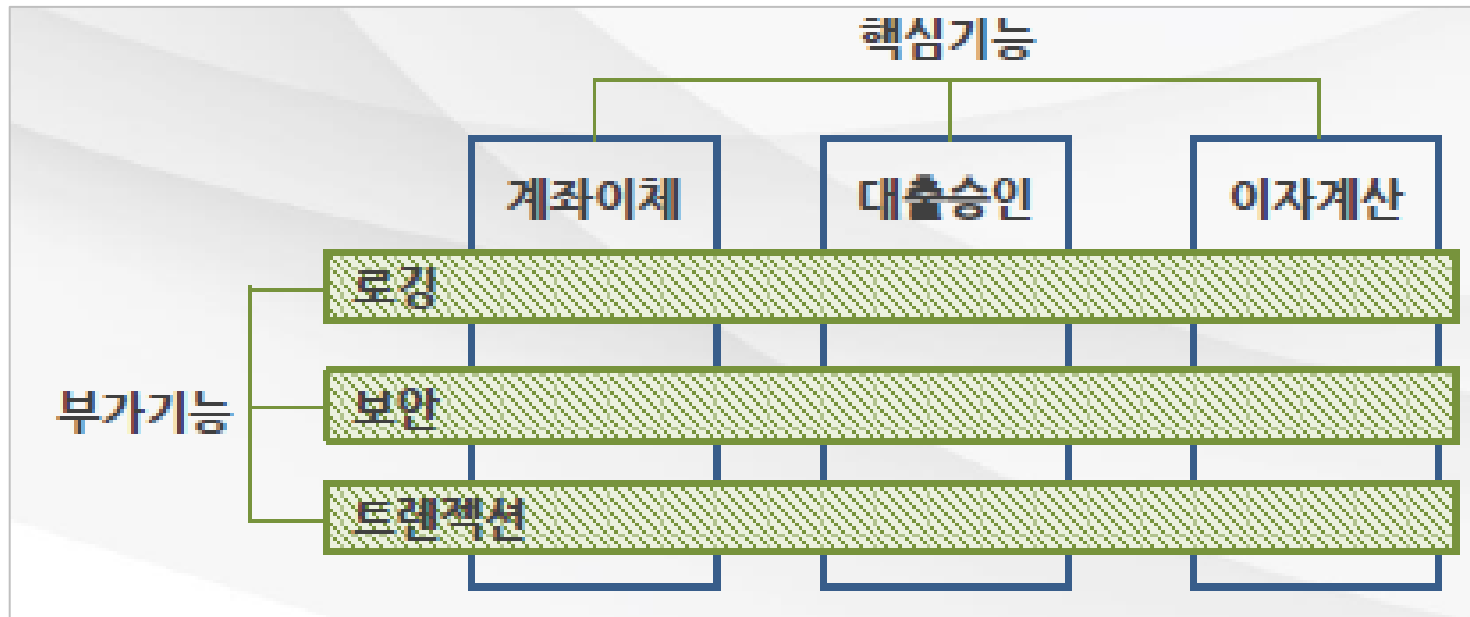
USERS 테이블 작성

```
CREATE TABLE USERS(  
    ID VARCHAR(8) PRIMARY KEY,  
    PASSWORD VARCHAR(8),  
    NAME VARCHAR(20),  
    ROLE VARCHAR(5)  
);
```



AOP(Aspect Oriented Programming)의 개요

- 핵심기능과 부가기능
 - 업무(Biz) 로직을 포함하는 기능을 핵심 기능(Core Concerns)
 - 핵심기능을 도와주면서 공통적이고 반복적으로 처리하는 내용을 부가기능 (Cross-cutting Concerns, 횡단관심사) 이라고 부른다.
 - 프로그램 안에서 횡단 관심사에 해당하는 부분을 분리해서 한 곳으로 모으는 것을 횡단관심사의 분리라 하고, 이를 실현하는 방법을 관점지향 프로그래밍(AOP)이라 함

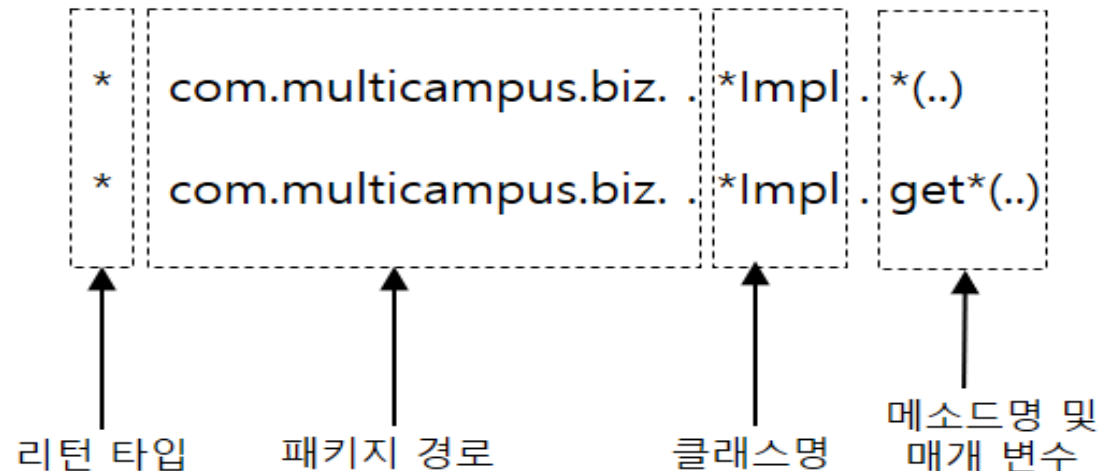


- **조인 포인트(Join Point)**

- 어드바이스(횡단관심사)가 실행될 지점이나 시점을 말함.
- Spring AOP에서는 항상 메서드가 실행되는 시점을 가리킴.
AOP를 적용할 수 있는 위치가 메소드로 한정되기 때문이다.

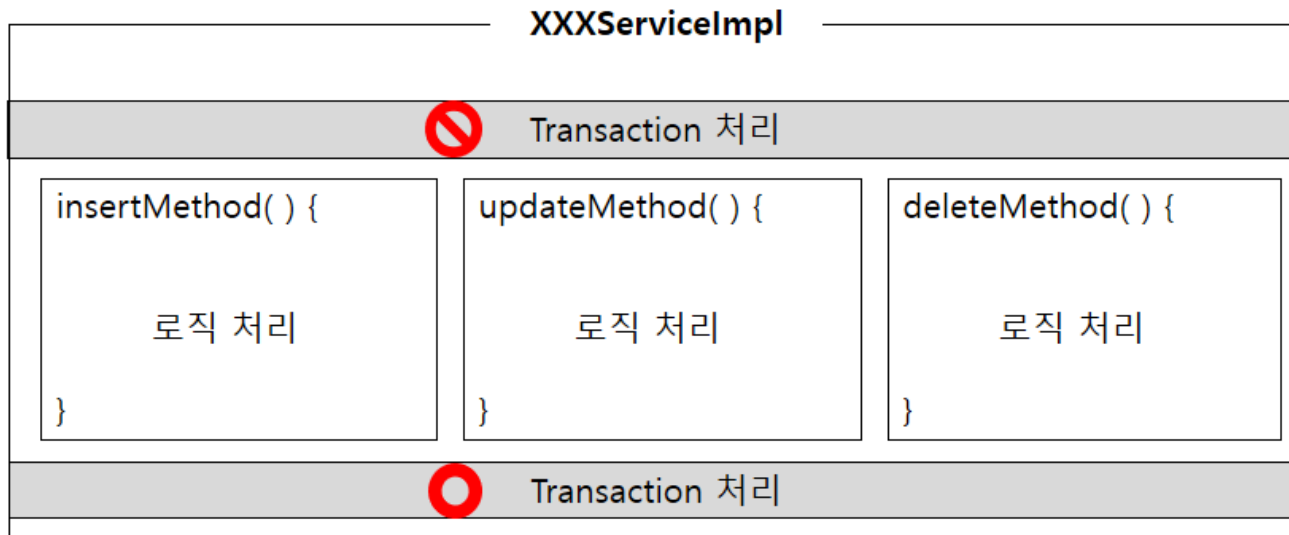
- **포인트 컷(Pointcut)**

- 수많은 조인 포인트 중에서 실제로 어드바이스를 적용할 곳을 선별하기 위한 표현식을 말함.
- 스프링 AOP에서는 포인트 컷을 정의할 때 XML 기반 설정방식으로 빈 정의 파일을 만들거나, 어노테이션 기반 설정방식으로 소스코드에 주석 형태로 정의한다.
- 포인트컷 표현식은 execution으로 시작하고, 메서드의 Signature를 비교하는 방법을 주로 이용한다.



- 어드바이스(Advice)

- 특정 조인 포인트에서 실행되는 코드로 횡단 관심사를 실제로 구현해서 처리하는 부분
- 어드바이스 동작 시점을 5가지로 지정 : before, after, after-returning, after-throwing, around



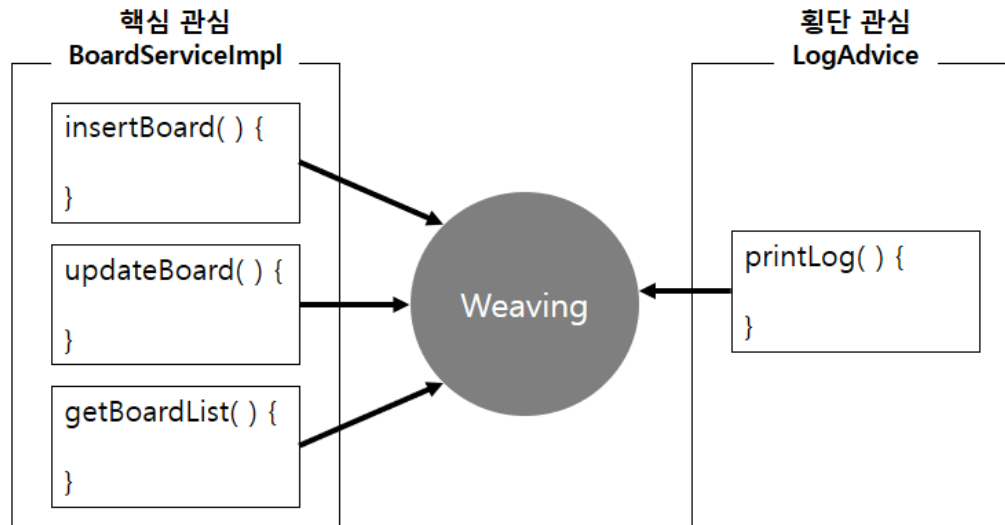
- 애스팩트(Asspect) 또는 어드바이저(Advisor)

- 애스팩트(어드바이저) = 어드바이스 + 포인트컷
- 어떤 포인트컷 메소드에 대해서 어떤 어드바이스 메소드 실행할 지 결정
- `<aop:aspect>` 또는 `<aop:advisor>` (트랜잭션 때 사용)



- 위빙(Weaving)

- 스프링 AOP는 기본적으로 실행 시점에 위빙한다.
- 위빙은 포인트컷에 의해서 결정된 타겟의 조인 포인트에 부가기능(어드바이스)을 삽입하는 과정을 뜻한다.

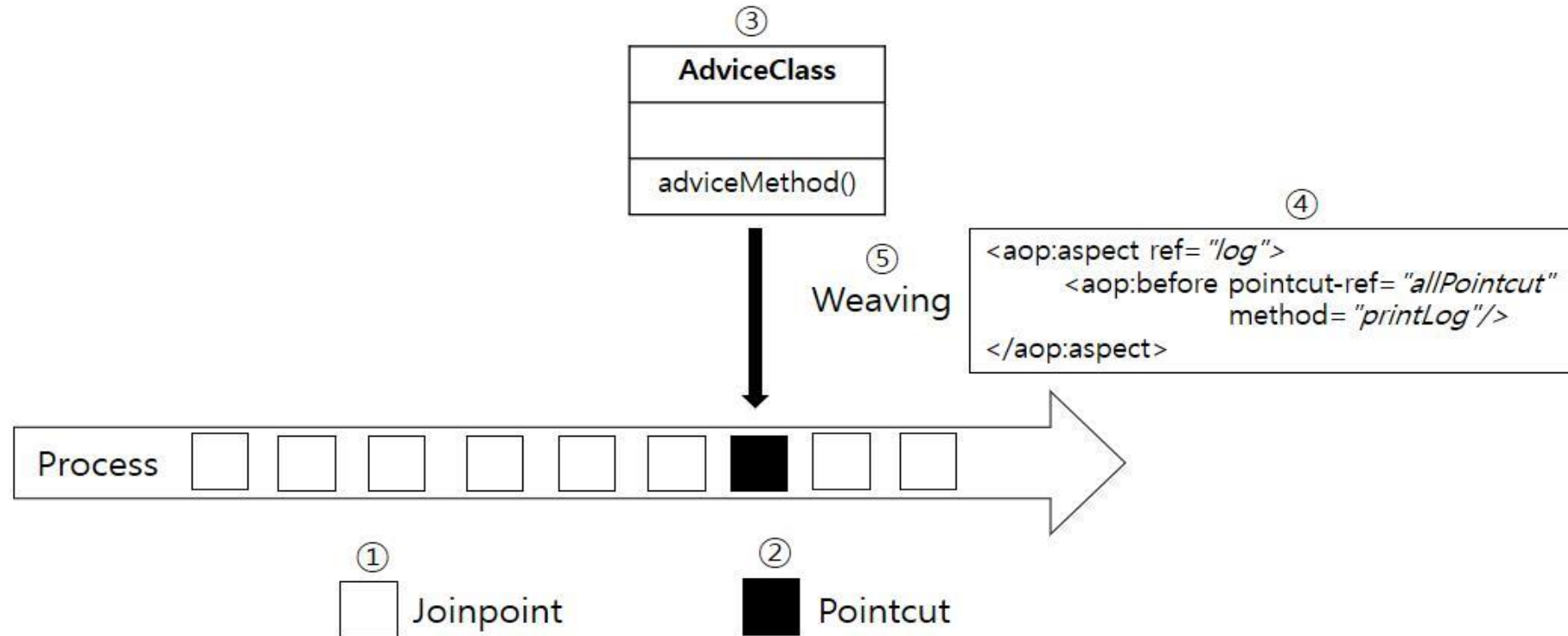


- 타겟(Target)

- 핵심기능을 담고 있는 모듈로, 타겟은 부가기능을 부여할 대상이 된다.



AOP 용어 정리



- ① 사용자가 비즈니스 컴포넌트의 여러 조인포인트 호출
- ② 이때 특정 포인트컷으로 지정한 메소드가 호출되는 순간,
- ③ 어드바이스 객체의 어드바이스 메소드가 실행되는데..
- ④ 위 ③의 실행을 위한 설정 즉, 애스팩트에 의해서 실행이 된다.
- ⑤ 위의 ③④ 실행과정을 위빙이라한다.



● AOP와 관련된 다양한 설정 : XML 방식과 어노테이션 방식

1) XML 기반의 POJO 클래스를 이용한 AOP 구현

- 부가기능을 제공하는 Advice 클래스를 작성한다.
- XML 설정 파일에 <aop:config>를 이용해서 애스펙트를 설정한다. (즉, 어드바이스와 포인트컷을 설정함)

2) @Aspect 어노테이션을 이용한 AOP 구현

- @Aspect 어노테이션을 이용해서 부가기능을 제공하는 Aspect 클래스를 작성한다. 이때 Aspect 클래스는 어드바이스를 구현하는 메서드와 포인트컷을 포함한다.
- XML 설정 파일에 <aop:aspectj-autoproxy />를 설정한다.



Spring AOP 설정 by xml 방식

LogAdvice.java

```
public class LogAdvice {  
    public void printLog() {  
        System.out.println("[공통 로그] 비즈니스 로직 수행 전 동작");  
    }  
}
```

applicationContext.xml

```
<bean id="log" class="com.springbook.biz.common.LogAdvice"></bean>  
  
<aop:config>  
    <aop:pointcut id="allPointcut" expression="execution(* com.springbook.biz.*Impl.*(..))"/>  
  
    <aop:pointcut id="getPointcut" expression="execution(* com.springbook.biz.*Impl.get*(..))"/>  
    <aop:aspect ref="log">  
        <aop:before pointcut-ref="getPointcut" method="printLog"/>  
    </aop:aspect>  
</aop:config>
```

① ② ③ ④



Spring AOP 설정 by xml 방식

```
applicationContext.xml
<beans xmlns="http://www.springframework.org/schema/beans" ...>
    <aop:config>
        <aop:pointcut .../>
        <aop:aspect ...></aop:aspect>
    </aop:config>
</beans>
```

1) < aop:config > 엘리먼트

- AOP설정에서의 루트 엘리먼트 (여러 번 사용가능)
- 하위 엘리먼트로 <aop:pointcut>, <aop:aspect>

2) < aop:pointcut > 엘리먼트

- 포인트컷 지정
- <aop:config> 혹은 <aop:aspect> 의 자식 엘리먼트 사용 가능
(단, <aop:aspect>의 자식으로 설정된 포인트컷은 해당 <aop:aspect>에서만 사용가능)



Spring AOP 설정 by xml 방식

3) < aop:aspect > 엘리먼트

- 핵심관심에 해당하는 포인트컷 메소드와 횡단관심에 해당하는 어드바이스 메소드를 결합하기위해
- 애스팩트를 어떻게 설정하느냐에 따라 위빙 결과가 달라짐
- 어드바이스 객체의 아이디나 메소드 이름을 모르면 사용 불가능

4) < aop:advisor > 엘리먼트

- < aop:aspect > 엘리먼트와 같은 기능
- 트랜잭션 설정같은 몇몇 특수한 경우 < aop:aspect > 엘리먼트 대신에 사용

※ 포인트컷 표현식 (교재 p168~170)

execution(* com.multicampus.biz. *.Impl. *(..) **)**
execution(* com.multicampus.biz. *.Impl. get*(..) **)**

리턴 타입 패키지 경로 클래스명 메소드명 및 매개 변수



어드바이스 동작 시점

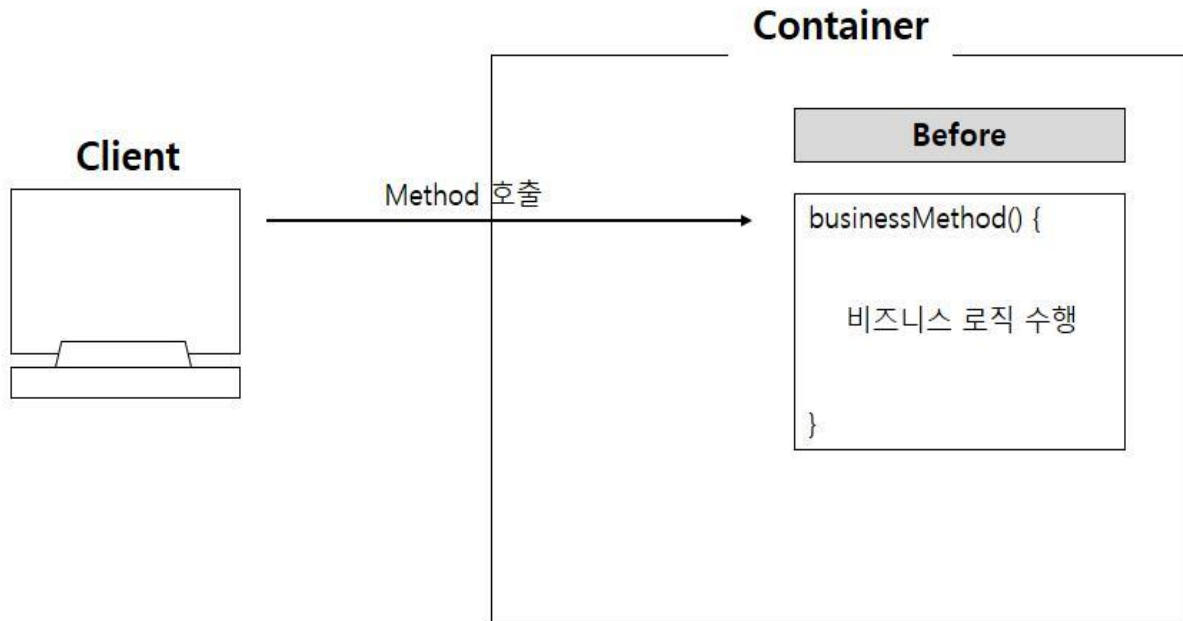
다음 5개의 동작 시점을 제공

동작 시점	설 명
Before	비즈니스 메소드 실행 전 동작
After	<ul style="list-style-type: none">- After Returning: 비즈니스 메소드가 성공적으로 반환되면 동작- After Throwing: 비즈니스 메소드 실행 중 예외가 발생하면 동작 (try~catch 블록에서 catch 블록에 해당)- After: 비즈니스 메소드가 실행된 후, 무조건 실행 (try~catch~finally 블록에서 finally 블록에 해당)
Around	Around는 메소드 호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음



어드바이스 동작 시점

- **Before** 어드바이스 동작 시점

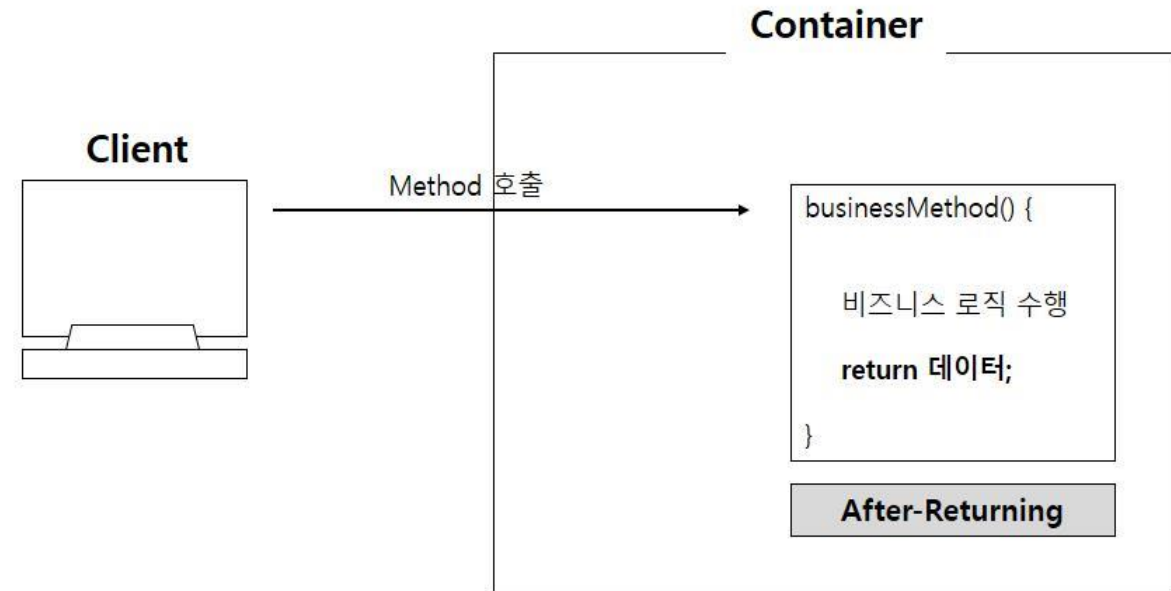


실습예제)

SpringProject5_AOP_step2 :

- BeforeAdvice.class
- BoardServiceClientBefore.class
- applicationContextBefore.xml

- **After Returning** 어드바이스 동작 시점



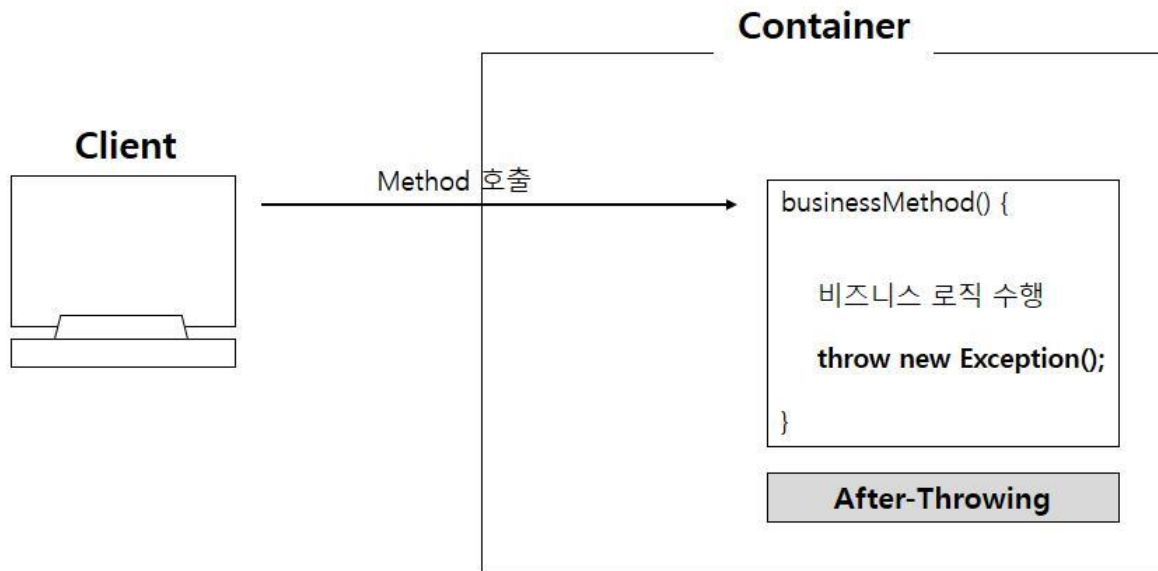
실습예제)

SpringProject5_AOP_step2 :

- AfterReturningAdvice.class
- BoardServiceClientAfterReturning.class
- applicationContextAfterReturning.xml

어드바이스 동작 시점

- **After Throwing** 어드바이스 동작 시점



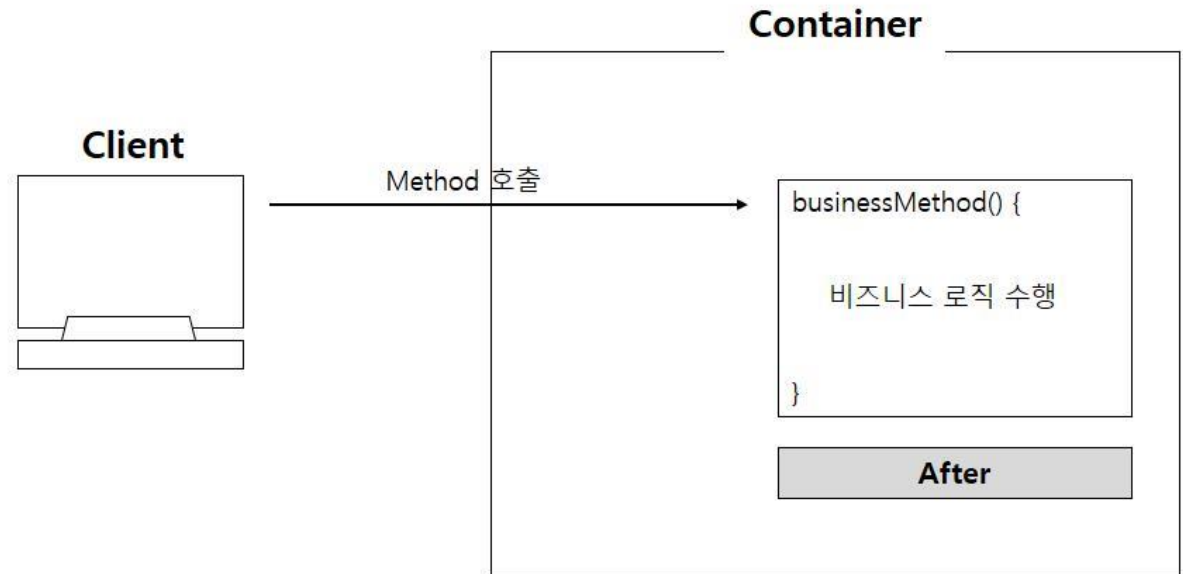
실습예제)

SpringProject5_AOP_step2 :

- AfterThrowingAdvice.class
- BoardServiceClientAfterThrowing.class
- applicationContextAfterThrowing.xml

(BoardServiceImpl.class 에러상황 추가!)

- **After** 어드바이스 동작 시점



실습예제)

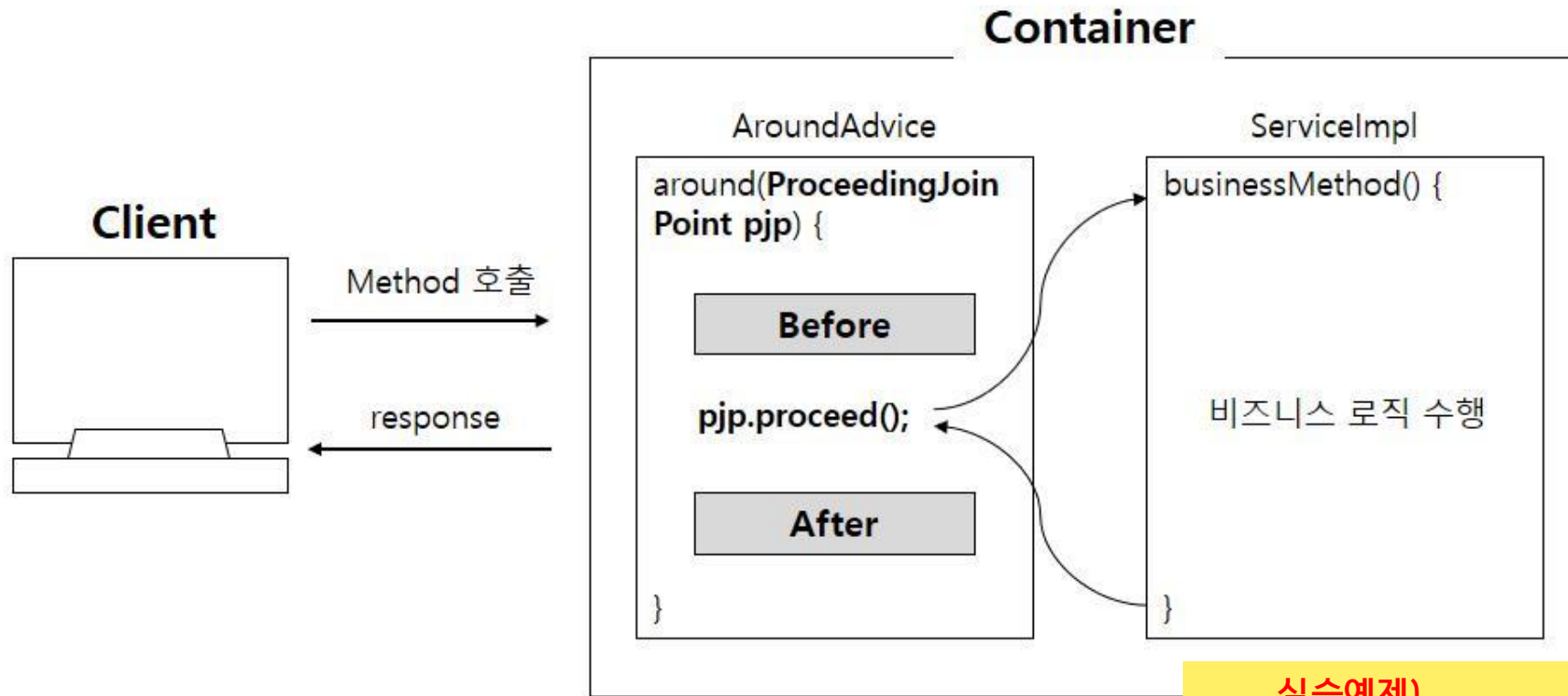
SpringProject5_AOP_step2 :

- AfterAdvice.class
- BoardServiceClientAfter.class
- applicationContextAfter.xml

(BoardServiceImpl.class 에러 有 / 無 에서 실행)

어드바이스 동작 시점

- **Around** 어드바이스 동작 시점



실습예제)

SpringProject5_AOP_step2 :

- AroundAdvice.class
- BoardServiceClientAround.class
- applicationContextAround.xml



JoinPoint와 바인드 변수 p183

- **JointPoint 인터페이스** : 횡단 관심에 해당하는 어드바이스 메소드를 의미있게 구현하기 위해 클라이언트가 호출한 비즈니스 메소드의 다양한 정보가 포함되어 있는 인터페이스
- Before / After Returning / After Throwing / After 어드바이스에서 사용
- JointPoint 메소드

메소드	설 명
Signature getSignature()	<ul style="list-style-type: none">- 클라이언트가 호출한 메소드의 시그니처(반환형, 이름, 매개변수) 정보가 저장된 Signature 객체 반환- Signature 객체의 메소드 :<ul style="list-style-type: none">● String getName() 클라이언트가 호출한 메소드 이름 리턴● String toLongString() 클라이언트가 호출한 메소드의 리턴타입, 이름, 매개변수를 패키지경로까지 리턴● String toShortString() 클라이언트가 호출한 메소드 시그니처를 축약한 문자열로 리턴
Object getTarget()	클라이언트가 호출한 비즈니스 메소드를 포함하는 비즈니스 객체 반환
Object[] getArgs()	클라이언트가 메소드를 호출할 때 넘겨준 인자 목록을 Object 배열로 반환

- **ProceedingJointPoint 인터페이스** :
JointPoint를 상속받은 인터페이스이며, JointPoint메소드+proceed()메소드 추가
- Around 어드바이스에서 사용



JoinPoint와 바인드 변수 - 실습

- **Before** 어드바이스

실습예제)

SpringProject5_AOP_step3 :

- BeforeAdvice.class
- BoardServiceClientBefore.class
- applicationContextBefore.xml

- **After Returning** 어드바이스

실습예제)

SpringProject5_AOP_step3 :

- AfterReturningAdvice.class
- BoardServiceClientAfterReturning.class
- UserServiceClientAfterReturning.class
- applicationContextAfterReturning.xml

Board와
User 두개 다
테스트

- **After Throwing** 어드바이스

실습예제)

SpringProject5_AOP_step3 :

- AfterThrowingAdvice.class
- BoardServiceClientAfterThrowing.class
- applicationContextAfterThrowing.xml

(BoardServiceImpl.class 에러상황 추가!)

- **After** 어드바이스

실습예제)

SpringProject5_AOP_step3 :

- AfterAdvice.class
- BoardServiceClientAfter.class
- applicationContextAfter.xml

(BoardServiceImpl.class 에러 有 / 無 에서 실행)

- **Around** 어드바이스

실습예제)

SpringProject5_AOP_step3 :

- AroundAdvice.class
- BoardServiceClientAround.class
- applicationContextAround.xml



- 먼저 스프링 설정파일에 <aop:aspect-autoproxy> 엘리먼트 선언

```
13 <context:component-scan base-package="com.springbook.biz">
14 </context:component-scan>
15
16 <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
17
18 </beans>
```

- AOP 관련 어노테이션들은 어드바이스 클래스에 설정

< 어노테이션 >

```
6 import org.springframework.stereotype.Service;
7
8 @Service
9 @Aspect
10 public class LogAdvice {
11     @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
12     public void allPointcut(){}
13
14     @Pointcut("execution(* com.springbook.biz..*Impl.get*(..))")
15     public void getPointcut(){}
16
17     @Before("allPointcut()")
18     public void printLog() {
19         System.out.println("[공통 로그] 비즈니스 로직 수행 전 동작");
20     }
21 }
```

< XML >

```
<bean id="log" class="com.springbook.biz.common.Befo
<aop:config>
    <aop:pointcut id="allPointcut" expression="execu
    <aop:pointcut id="getPointcut" expression="execu
    <aop:aspect ref="log">
        <aop:before pointcut-ref="getPointcut" metho
    </aop:aspect>
</aop:config>
```



Spring AOP 설정 by 어노테이션 방식

- 어드바이스 동작시점과 관련된 어노테이션

어노테이션	설 명
@Before	비즈니스 메소드 실행 전에 동작
@AfterReturning	비즈니스 메소드가 성공적으로 반환되면 동작
@AfterThrowing	비즈니스 메소드 실행 중 예외가 발생하면 동작 (마치 try~catch 블록에서 catch 블록에 해당).
@After	비즈니스 메소드가 실행된 후, 무조건 실행 (try~catch~finally 블록에서 finally 블록에 해당)
@Around	호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음



- Before 어드바이스

```
@Service
@Aspect
public class BeforeAdvice {
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
    public void allPointcut() {}

    @Before("allPointcut()")
    public void beforeLog(JoinPoint jp) {
        String method = jp.getSignature().getName();
        Object[] args = jp.getArgs();

        System.out.println("[사전 처리] " + method +
            "() 메소드 ARGS 정보 : " + args[0].toString());
    }
}
```

```
6 import org.springframework.stereotype.Service;
7
8 @Service
9 @Aspect
10 public class LogAdvice {
11     @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
12     public void allPointcut(){}
13
14     @Pointcut("execution(* com.springbook.biz..*Impl.get*(..))")
15     public void getPointcut(){}
16
17     @Before("allPointcut()")
18     public void printLog() {
19         System.out.println("[공통 로그] 비즈니스 로직 수행 전 동작");
20     }
21 }
22
```

```
@Service
@Aspect
public class AfterReturningAdvice {
    @Pointcut("execution(* com.springbook.biz..*Impl.get*(..))")
    public void getPointcut() {}

    @AfterReturning(pointcut="getPointcut()", returning="returnObj")
    public void afterLog(JoinPoint jp, Object returnObj) {
        String method = jp.getSignature().getName();

        System.out.println("[사후 처리] " + method +
            "() 메소드 리턴값 : " + returnObj.toString());
    }
}
```



- **After Throwing** 어드바이스

```
@Service
@Aspect
public class AfterThrowingAdvice {
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
    public void allPointcut() {}

    @AfterThrowing(pointcut="allPointcut()", throwing="exceptObj")
    public void exceptionLog(JoinPoint jp, Exception exceptObj) {
        String method = jp.getSignature().getName();
        System.out.println(method + "() 메소드 수행 중 예외 발생!");

        if(exceptObj instanceof IllegalArgumentException) {
            System.out.println("부적합한 값이 입력되었습니다.");
        }
    }
}
```

- **After** 어드바이스

```
@Service
@Aspect
public class AfterAdvice {
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
    public void allPointcut() {}

    @After("allPointcut()")
    public void finallyLog() {
        System.out.println("[사후 처리] 비즈니스 로직 수행 후 무조건 동작");
    }
}
```

- **Around** 어드바이스

```
@Service
@Aspect
public class AroundAdvice {
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")
    public void allPointcut(){}

    @Around("allPointcut()")
    public Object aroundLog(ProceedingJoinPoint pjp) throws Throwable {
        String method = pjp.getSignature().getName();
        System.out.println(method + "() 메소드 수행에 걸린 시간 : "
            + stopWatch.getTotalTimeMillis() + "(ms)초");
        return obj;
    }
}
```



- 외부 Pointcut 참조

@Aspect

```
public class PointcutCommon {  
    @Pointcut("execution(* com.springbook.biz..*Impl.*(..))")  
    public void allPointcut() {}  
    @Pointcut("execution(* com.springbook.biz..*Impl.get*(..))")  
    public void getPointcut() {}  
}
```

```
@Service  
@Aspect  
public class BeforeAdvice {  
    @Before("PointcutCommon.allPointcut()")  
    public void beforeLog(JoinPoint jp) {  
  
    }  
}
```

