

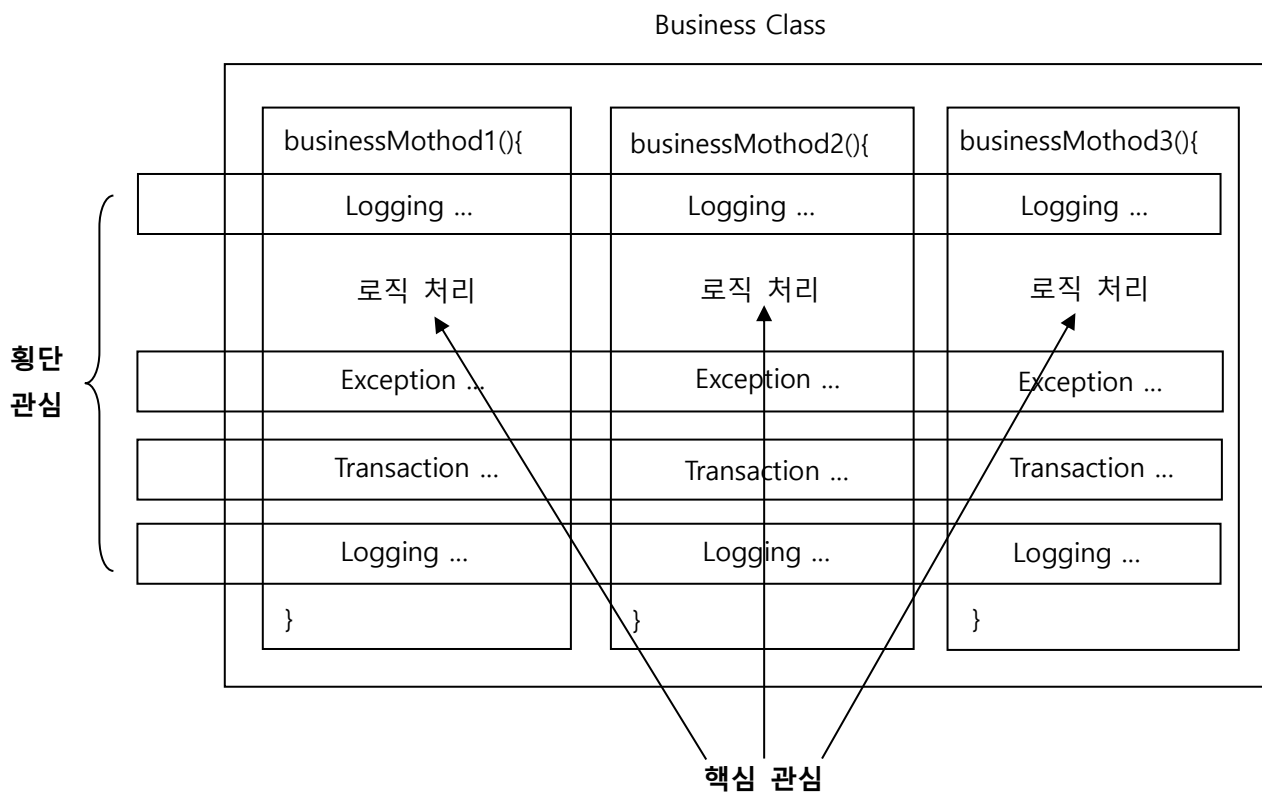
11. 스프링 AOP

비즈니스 컴포넌트 개발에서 가장 중요한 두 가지 원칙은 낮은 결합도와 높은 응집도를 유지하는 것이다. 스프링의 의존성 주입(Dependency Injection)을 이용하면 비즈니스 컴포넌트를 구성하는 객체들의 결합도를 떨어뜨릴 수 있어서 의존관계를 쉽게 변경할 수 있다. 스프링의 IoC가 결합도와 관련된 기능이라면, **AOP(Asspect Oriented Programming)**는 **응집도**와 관련된 기능이라 할 수 있다.

11.1 AOP 이해

엔터프라이즈 애플리케이션의 메소드들은 대부분 복잡한 코드들로 구성되어 있다. 이 중에서 핵심 비즈니스 로직은 적은 부분이고 주로 로깅이나 예외, 트랜잭션 처리 같은 부가적인 코드가 대부분이다. 이 부가적인 코드도 엔터프라이즈 애플리케이션의 비즈니스 로직이기 때문에 중요하고 매번 반복되어 코드 분석과 유지보수를 어렵게 만든다. AOP는 이러한 부가적인 공통 코드들을 효율적으로 관리하는데 주목한다,

AOP를 이해하는 데에 가장 중요한 핵심 개념이 바로 관심 분리(Separation of Concerns)이다. AOP에서는 메소드마다 공통으로 등장하는 로깅이나 예외, 트랜잭션 처리 같은 코드들을 횡단 관심(Crosscutting Concerns)이라고 한다. 이에 비해 사용자의 요청에 따라 실제로 수행되는 핵심 비즈니스 로직을 핵심 관심(Core Concerns)이라고 한다.



이 두 관심을 완벽하게 분리할 수 있다면, 구현하는 메소드에는 실제 비즈니스 로직만으로 구성할 수 있으므로 더욱 간결하고 응집도 높은 코드를 유지할 수 있다. 문제는 기존의 OOP 언어에서는 횡단 관심에 해당하는 공통 코드를 완벽하게 독립적인 모듈로 분리하기가 어렵다.

기존 OOP에서 관심 분리가 어려운지 확인하고 스프링의 AOP가 관심 분리를 해결하는 과정을 살펴본다.

BoardService 컴포넌트의 모든 비즈니스 메소드가 실행되기 직전에 공통으로 처리할 로직을 LogAdvice 클래스에 printLog() 메소드로 구현한다.

LogAdvice.java

```
package com.spring.common;

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

LogAdvice 클래스의 printLog() 메소드를 BoardService 컴포넌트에서 사용할 수 있도록 BoardServiceImpl 클래스를 수정한다.

BoardServiceImpl.java

```
package com.spring.board.impl;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.spring.board.BoardService;
import com.spring.board.BoardVO;
import com.spring.board.dao.BoardDAO;
import com.spring.common.LogAdvice;

@Service("boardService")
public class BoardServiceImpl implements BoardService {
    @Autowired
    private BoardDAO boardDAO;
    private LogAdvice log;

    public BoardServiceImpl() {
        log = new LogAdvice();
    }

    @Override
    public void insertBoard(BoardVO vo) {
        log.printLog();
        boardDAO.insertBoard(vo);
    }

    @Override
    public void updateBoard(BoardVO vo) {
        log.printLog();
        boardDAO.updateBoard(vo);
    }
}
```

```

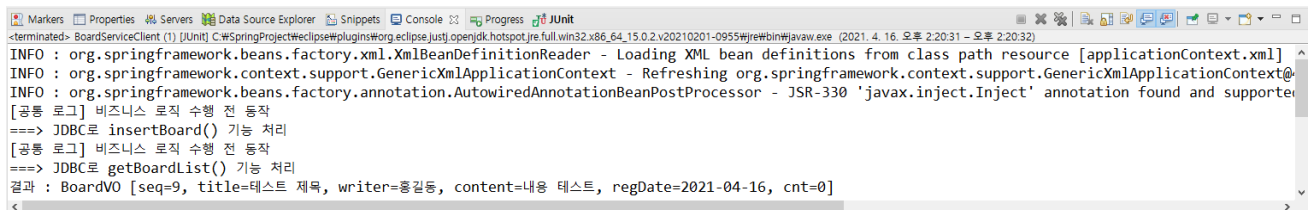
@Override
public void deleteBoard(BoardVO vo) {
    log.printLog();
    boardDAO.deleteBoard(vo);
}

@Override
public BoardVO getBoard(BoardVO vo) {
    log.printLog();
    return boardDAO.getBoard(vo);
}

@Override
public List<BoardVO> getBoardList(BoardVO vo) {
    log.printLog();
    return boardDAO.getBoardList(vo);
}
}

```

BoardServiceImpl 객체가 생성될 때 생성자에서 LogAdvice 객체도 같이 생성한다. 그리고 각 비즈니스메소드에서 비즈니스 로직을 수행하기 전에 LogAdvice의 printLog() 메소드를 호출하기만 하면 된다. 이후에 공통 기능을 수정할 때는 LogAdvice 클래스의 printLog() 메소드만 수정하면 된다. BoardServiceClient를 실행하면 다음과 같은 결과를 볼 수 있다.



The screenshot shows the Eclipse IDE console with the following output:

```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@...
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and supported
[공통 로그] 비즈니스 로직 수행 전 동작
==> JDBC로 insertBoard() 기능 처리
[공통 로그] 비즈니스 로직 수행 전 동작
==> JDBC로 getBoardList() 기능 처리
결과 : BoardVO [seq=9, title=테스트 제목, writer=홍길동, content=내용 테스트, regDate=2021-04-16, cnt=0]

```

기존에 사용하던 LogAdvice 클래스의 성능이 떨어져서 이를 대체할 Log4jAdvice 클래스를 만든다. 메소드 이름도 printLogging()으로 변경한다.

Log4jAdvice.java

```

package com.spring.common;

public class Log4jAdvice {
    public void printLogging(){
        System.out.println("[공통 로그 Log4j] 비즈니스 로직 수행 전 동작");
    }
}

```

BoardServiceImpl 클래스의 모든 메소드는 Log4jAdvice를 이용하도록 수정한다.

BoardServiceImpl.java

```

package com.springexam.biz.board.impl;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.springframework.stereotype.Service;
import com.springexam.biz.board.BoardService;
import com.springexam.biz.board.BoardVO;
import com.springexam.biz.board.dao.BoardDAO;
import com.springexam.biz.common.Log4jAdvice;

@Service("boardService")
public class BoardServiceImpl implements BoardService {

    @Autowired
    private BoardDAO boardDAO;
    private Log4jAdvice log;

    public BoardServiceImpl() {
        log = new Log4jAdvice();
    }

    @Override
    public void insertBoard(BoardVO vo) {
        log.printLogging();
        boardDAO.insertBoard(vo);
    }

    @Override
    public void updateBoard(BoardVO vo) {
        log.printLogging();
        boardDAO.updateBoard(vo);
    }

    @Override
    public void deleteBoard(BoardVO vo) {
        log.printLogging();
        boardDAO.deleteBoard(vo);
    }

    @Override
    public BoardVO getBoard(BoardVO vo) {
        log.printLogging();
        return boardDAO.getBoard(vo);
    }

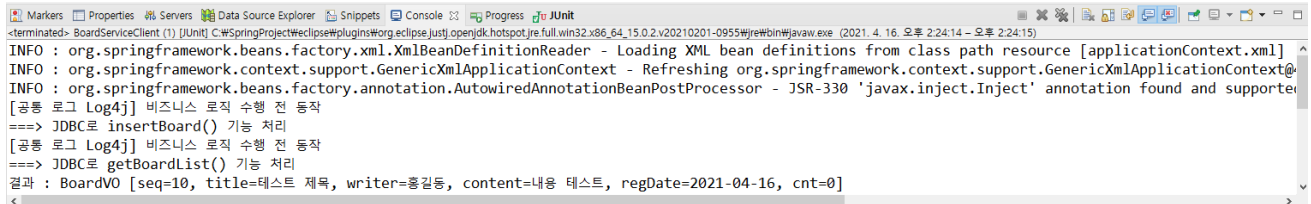
    @Override
    public List<BoardVO> getBoardList(BoardVO vo) {
        log.printLogging();
        return boardDAO.getBoardList(vo);
    }

}

```

결국, Advice 클래스가 LogAdvice에서 Log4jAdvice로 바꾸는 순간 BoardServiceImpl 생성자를 수정해야 한다. 그리고 printLog()가 printLogging() 메소드로 변경되었으므로 printLog()를 호출했던 모든 메소드를 수정해야 한다.

BoardServiceClient를 실행하면 수정내용이 잘 반영된 것을 확인할 수 있다.(Log4jAdvice 객체)



OOP처럼 모듈화가 뛰어난 언어를 사용하여 개발하더라도 공통 모듈에 해당하는 Advice 클래스 객체를 생성하고 공통 메소드를 호출하는 코드가 비즈니스 메소드에 있다면, 핵심 관심(BoardServiceImpl)과 횡단 관심(Log4jAdvice)을 완벽하게 분리할 수는 없다. 하지만 스프링의 AOP는 이런 OOP의 한계를 극복할 수 있도록 도와준다.

11.2 AOP 시작

스프링의 AOP를 이용해서 핵심 관심과 횡단 관심을 분리한다. 그래서 BoardServiceImpl 소스와 무관하게 LogAdvice나 Log4jAdvice 클래스의 메소드를 실행할 수 있다.

(1) 비즈니스 클래스 수정

BoardServiceImpl 클래스를 원래의 상태로 되돌린다.

BoardServiceImpl.java

```
package com.spring.board.impl;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.spring.board.BoardService;
import com.spring.board.BoardVO;
import com.spring.board.dao.BoardDAO;

@Service("boardService")
public class BoardServiceImpl implements BoardService {
    @Autowired
    private BoardDAO boardDAO;

    @Override
    public void insertBoard(BoardVO vo) {
        boardDAO.insertBoard(vo);
    }

    @Override
    public void updateBoard(BoardVO vo) {
```

```

        boardDAO.updateBoard(vo);
    }
    @Override
    public void deleteBoard(BoardVO vo) {
        boardDAO.deleteBoard(vo);
    }
    @Override
    public BoardVO getBoard(BoardVO vo) {
        return boardDAO.getBoard(vo);
    }
    @Override
    public List<BoardVO> getBoardList(BoardVO vo) {
        return boardDAO.getBoardList(vo);
    }
}

```

BoardServiceImpl 클래스는 LogAdvice 클래스와 Log4jAdvice 클래스와 아무런 상관이 없는 클래스가 되었다.

(2) AOP 라이브러리 추가

AOP를 적용하기 위해서 우선 pom.xml 파일에 AOP 관련 라이브러리를 추가한다.

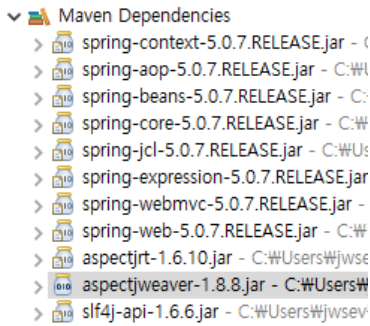
pom.xml

```

생략
<!-- AspectJ -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${org.aspectj-version}</version>
</dependency>
<!-- AOP 관련 라이브러리 추가 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.8</version>
</dependency>
<!-- Logging -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>${org.slf4j-version}</version>
</dependency>

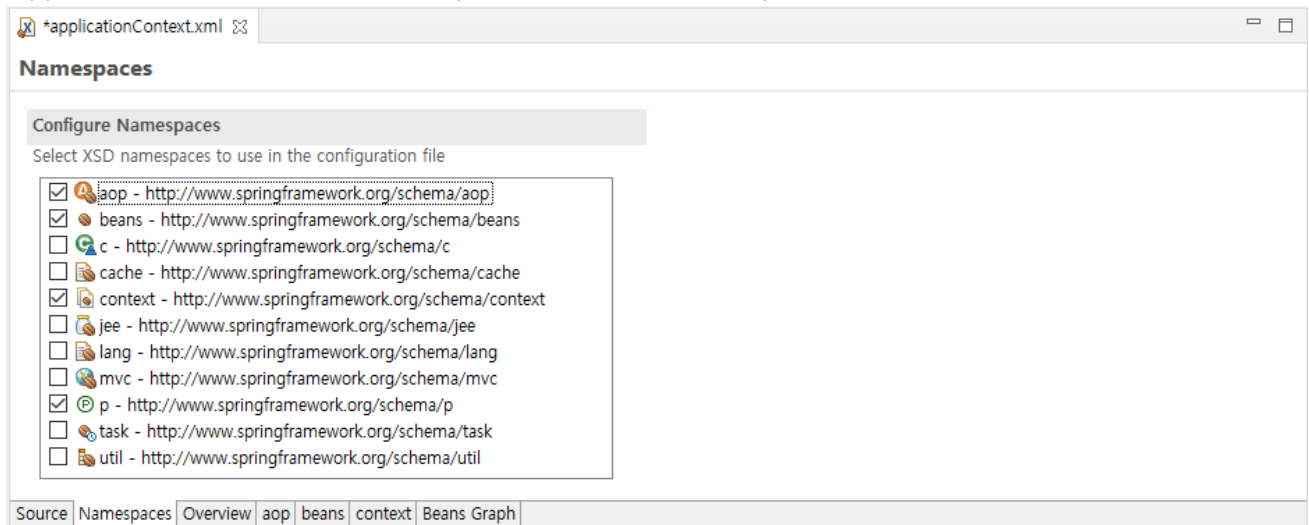
```

생략



(3) 네임스페이스 추가 및 AOP 설정

AOP 설정을 추가하려면 AOP에서 제공하는 엘리먼트들을 사용해야 한다. 따라서 스프링 설정 파일 (applicationContext.xml)에서 [Namespaces] 탭을 클릭하고 aop 네임스페이스를 추가한다.



이전에 작성했던 LogAdvice 클래스를 스프링 설정 파일에 <bean> 등록 후에 AOP 관련 설정을 추가한다. 자세한 설명은 뒤에서 한다.

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-4.3.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">

    <context:component-scan base-package="com.spring">
    </context:component-scan>
```

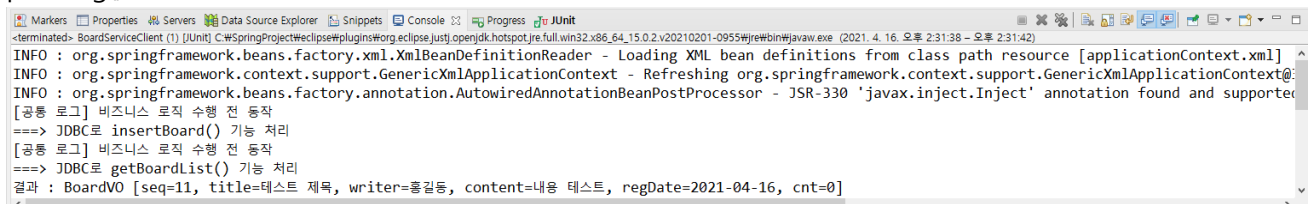
```

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

```

(4) 테스트 및 결과 확인

BoardServiceClient를 실행하여 insertBoard()와 getBoadList() 메소드가 호출될 때 LogAdvice 클래스의 printLog() 메소드가 실행되는지 확인한다.



```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@...
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and supported
[공통 로그] 비즈니스 로직 수행 전 동작
===> JDBC로 insertBoard() 기능 처리
[공통 로그] 비즈니스 로직 수행 후 동작
===> JDBC로 getBoardList() 기능 처리
결과 : BoardVO [seq=11, title=테스트 제목, writer=홍길동, content=내용 테스트, regDate=2021-04-16, cnt=0]

```

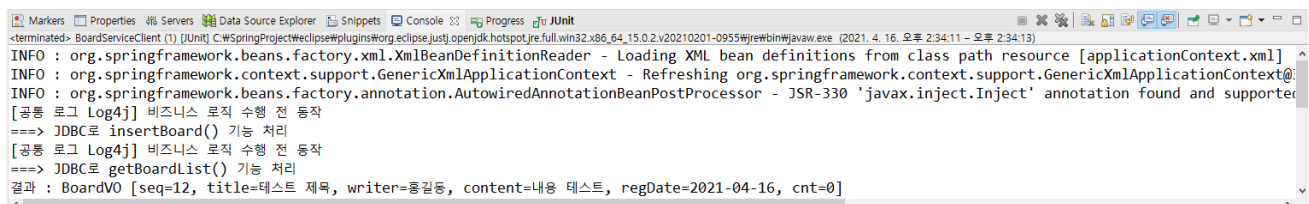
LogAdvice를 Log4jAdvice로 교체하려면 스프링 설정 파일의 AOP 설정 부분만 수정하면 된다.

applicationContext.xml

```

<bean id="log" class="com.spring.common.Log4jAdvice"></bean>
<aop:config>
    <aop:pointcut expression="execution(* com.spring..*Impl.*(..))" id="allPointcut"/>
    <aop:aspect ref="log">
        <aop:before method="printLogging" pointcut-ref="allPointcut"/>
    </aop:aspect>
</aop:config>

```



```

INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource [applicationContext.xml]
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext@...
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found and supported
[공통 로그 Log4j] 비즈니스 로직 수행 전 동작
===> JDBC로 insertBoard() 기능 처리
[공통 로그 Log4j] 비즈니스 로직 수행 후 동작
===> JDBC로 getBoardList() 기능 처리
결과 : BoardVO [seq=12, title=테스트 제목, writer=홍길동, content=내용 테스트, regDate=2021-04-16, cnt=0]

```

스프링의 AOP는 클라이언트가 핵심 관심에 해당하는 비즈니스 메소드를 호출할 때, 횡단 관심에 해당하는 메소드를 적절하게 실행해준다. 이때, 핵심 관심 메소드와 횡단 관심 메소드 사이에서 소스상의 결합은 발생하지 않으며 AOP를 사용하는 주된 목적이다.

12. AOP 용어 및 기본 설정

12.1 AOP 용어

(1) Joinpoint

조인포인트는 클라이언트가 호출하는 모든 비즈니스 메소드로서, BoardServiceImpl이나 UserServiceImpl 클래스의 모든 메소드를 조인포인트라고 생각하면 된다. 조인포인트를 '포인트컷 대상' 또는 '포인트컷 후보'라고 하는데, 이는 조인포인트 중에서 포인트컷이 선택되기 때문이다.

(2) Pointcut

클라이언트가 호출하는 모든 비즈니스 메소드가 조인포인트라면, 포인트컷은 필터링된 조인포인트를 의미한다. 예를 들어 트랜잭션을 처리하는 공통 기능을 만든다면, 횡단 관심 기능은 등록, 수정, 삭제 기능의 비즈니스 메소드에 대해서만 동작하지만, 검색 기능의 메소드에 대해서는 트랜잭션과 무관하므로 동작할 필요가 없다.

수많은 비즈니스 메소드 중에서 원하는 특정 메소드에서만 횡단 관심에 해당하는 공통 기능을 수행시키기 위해서 포인트컷이 필요하다. 포인트컷을 이용하면 메소드가 포함된 클래스와 패키지는 물론이고 메소드 시그니처까지 정확하게 지정할 수 있다.

(3) Advice

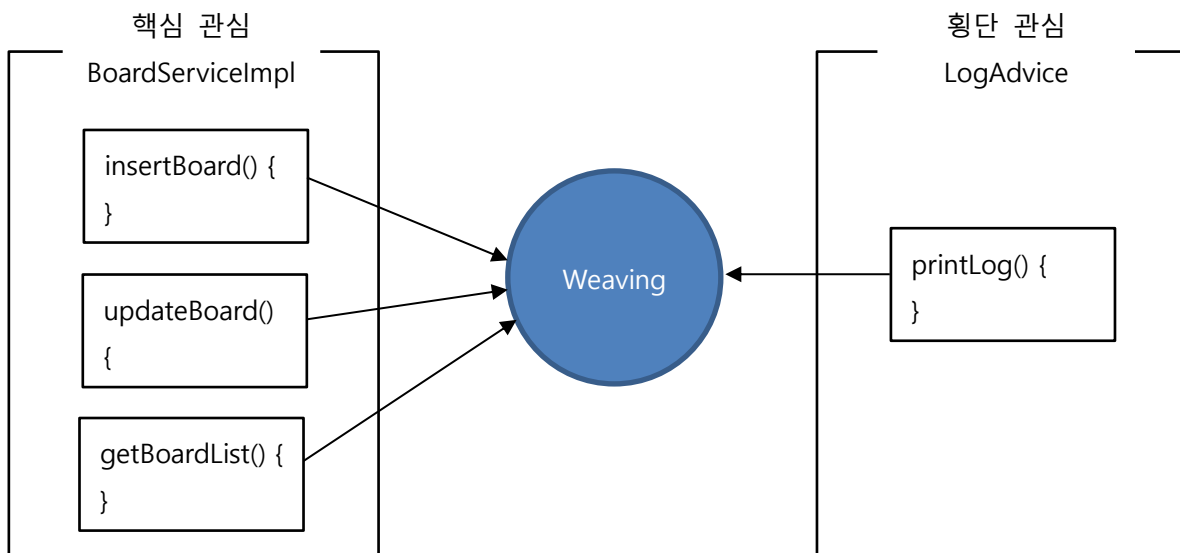
어드바이스는 횡단 관심에 해당하는 공통 기능의 코드를 의미하며, 독립된 클래스의 메소드로 작성한다. 어드바이스로 구현된 메소드가 언제 동작할지 스프링 설정 파일에서 지정할 수 있다.

예를 들어, 트랜잭션 관리 기능의 어드바이스 메소드가 있으면, 이 어드바이스가 비즈니스 로직이 수행되기 전에 동작하는 것은 마무리 의미가 없다. 당연히 비즈니스 로직 수행 후에 트랜잭션을 커밋(commit) 또는 롤백(rollback) 처리하면 된다.

스프링에서 어드바이스의 동작 시점은 'before', 'after', 'after-returning', 'after-throwing', 'around' 등 다섯 가지로 지정할 수 있다.

(4) 위빙(Weaving)

위빙은 포인트컷으로 지정한 핵심 관심 메소드가 호출될 때, 어드바이스에 해당하는 횡단 관심 메소드가 삽입되는 과정을 의미한다. 이 위빙을 통해서 비즈니스 메소드를 수정하지 않고도 횡단 관심에 해당하는 기능을 추가하거나 변경할 수 있다.



위빙을 처리하는 방식은 크게 컴파일타임(Compiletime) 위빙, 로딩타임>Loadingtime) 위빙, 런타임(Runtime) 위빙이 있지만, 스프링에서는 런타임 위빙 방식만 지원한다.

(5) 애스팩트(Aspect) 또는 어드바이저(Advisor)

Aspect Oriented Programming이라는 이름에서 알수 있듯이 AOP의 핵심은 애스팩트이다. 애스팩트는 포인트컷과 어드바이스의 결합으로서, 어떤 포인트컷 메소드에 대해서 어떤 어드바이스 메소드를 실행할 지 결정한다. 이 애스팩트는 설정에 따라 AOP의 동작 방식이 결정되므로 AOP 용어 중 가장 중요한 개념이다. 애스팩트와 어드바이저가 같은 의미의 용어이다.

(6) AOP 용어 종합

사용자는 시스템을 사용하면서 자연스럽게 비즈니스 컴포넌트의 여러 조인 포인트를 호출하게 된다. 이때 특정 포인트컷으로 지정한 메소드가 호출되는 순간, 어드바이스 객체의 어드바이스 메소드가 실행된다. 이 어드바이스 메소드의 동작시점을 5가지로 지정할 수 있으며, 포인트컷으로 지정한 메소드를 호출할 때, 어드바이스 메소드를 삽입하도록 하는 설정을 애스팩트라고 한다. 이 애스팩트(Aspect) 설정에 따라 위빙(weaving)이 처리된다.

12.3 포인트컷 표현식

포인트컷을 이용하면 어드바이스 메소드가 적용될 비즈니스 메소드를 정확하게 필터링할 수 있는데, 이때 다양한 포인트컷 표현식을 사용할 수 있다. 포인트컷 표현식은 메소드처럼 생긴 execution 명시자를 이용하며, execution 명시자 안에 포인트 컷 표현식을 기술한다.

execution(* com.spring. *.Impl . get*(..))
리턴타입 패키지경로 클래스명 메소드명(매개변수)

① 리턴타입 지정

리턴타입 지정에서 가장 기본적인 방법은 '*' 캐릭터를 이용하는 것이다.

표현식	설명
*	모든 리턴타입 허용
void	리턴타입이 void인 메소드 선택
!void	리턴타입이 void가 아닌 메소드 선택

② 패키지 지정

패키지 경로를 지정할 때는 '*', '..' 캐릭터 이용한다.

표현식	설명
com.spring	정확하게 com.spring 패키지만 선택
com.spring..	com.spring 패키지로 시작하는 모든 패키지 선택
com.spring..impl	com.spring 패키지로 시작하면서 마지막 패키지 이름이 Impl로 끝나는 패키지 선택

③ 클래스 지정

클래스 이름을 지정할 때는 '*', '+' 캐릭터 이용한다.

표현식	설명
-----	----

BoardServiceImpl	정확하게 BoardServiceImpl 클래스만 선택
*Impl	클래스 이름이 Impl로 끝나는 클래스만 선택
BoardService+	클래스 이름 뒤에 '+'가 붙으면 해당 클래스로부터 파생된 모든 자식 클래스 선택. 인터페이스 뒤에 '+'가 붙으면 해당 인터페이스를 구현한 모든 클래스 선택

④ 메소드 지정

메소드를 지정할 때는 주로 '*' 캐릭터를 사용하고 매개 변수를 지정할 때는 '..'을 사용한다.

표현식	설명
*(..)	가장 기본 설정으로 모든 메소드 선택
get*(..)	메소드 이름이 get으로 시작하는 모든 메소드 선택

⑤ 매개변수 지정

매개 변수를 지정할 때는 '..', '*' 캐릭터를 상요하거나 정확한 타입을 지정한다.

표현식	설명
(..)	가장 기본 설정으로 '..'은 매개변수의 개수와 타입에 제약이 없음을 의미.
(*)	반드시 1개의 매개변수를 가지는 메소드만 선택
(com.spring.user.UserVO)	매개변수로 UserVO를 가지는 메소드만 선택. 이때 클래스의 패키지 경로가 반드시 포함되어야 한다.
(!com.spring.user.UserVO)	매개변수로 UserVO를 가지지 않는 메소드만 선택.
(Integer, ..)	한 개 이상의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만 선택
(Integer, *)	반드시 두 개의 매개변수를 가지되, 첫 번째 매개변수의 타입이 Integer인 메소드만 선택

13. 어드바이스 동작 시점

어드바이스는 각 조인포인트에 삽입되어 동작할 횡단 관심에 해당하는 공통 기능이며, 동작 시점은 각 AOP 기술마다 다르다. 스프링에서는 다섯 가지의 동작 시점이 있다.

동작 시점	설명
before	비즈니스 메소드 실행 전 동작
after	after Returning : 비즈니스 메소드가 성공적으로 리턴되면 동작 after Throwing : 비즈니스 메소드 실행 중 예외가 발생하면 동작(catch 블록) after : 비즈니스 메소드가 실행된 후 무조건 실행(finally 블록)
around	메소드 호출 자체를 가로로 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있다.

14 JoinPoint와 바인드 변수

횡단 관심에 해당하는 어드바이스 메소드를 의미있게 구현하려면 클라이언트가 호출한 비즈니스 메소드의 정보가 필요하다. 예를 들어, After Throwing 기능의 어드바이스 메소드를 구현한다고 하면, 이때 예외가 발생한 비즈니스 메소드의 이름이 무엇인지, 그 메소드가 속한 클래스와 패키지 정보는 무엇인지 알아야 정확한 예외 처리 로직을 구현할 수 있다. 스프링에서는 이런 다양한 정보들을 이용할 수 있도록 JoinPoint 인터페이스를 제공한다.

15 어노테이션 기반 AOP

15.1 어노테이션 기반 AOP 설정

(1) 어노테이션 사용을 위한 스프링 설정

AOP를 어노테이션으로 설정하려면 먼저 스프링 설정 파일에 <aop:aspectj-autoproxy> 엘리먼트를 선언해야 한다.

applicationContext.xml 수정

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.2.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.2.xsd">

    <context:component-scan base-package="com.spring">
    </context:component-scan>

    <!-- AOP 어노테이션 설정 -->
```

```
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
</beans>
```

<aop:aspectj-autoproxy /> 엘리먼트만 선언하면 스프링 컨테이너는 AOP 관련 어노테이션들을 인식하고 용도에 맞게 처리해준다.

AOP 관련 어노테이션들을 어드바이스 클래스에 설정한다. 그리고 어드바이스 클래스에 선언된 어노테이션들을 스프링 컨테이너가 처리하게 하려면, 반드시 어드바이스 객체가 생성되어 있어야 한다. 따라서 어드바이스 클래스는 반드시 스프링 설정 파일에 <bean>을 등록하거나 **@Service** 어노테이션을 사용하여 컴포넌트가 검색될 수 있도록 해야 한다.

Annotation 설정	@Service public class LogAdvice() { }
XML 설정	<bean id="log" class="com.spring.common.LogAdvice"></bean>

(2) 포인트컷 설정

XML 설정에서 포인트컷을 설정할 때는 <aop:pointcut> 엘리먼트를 사용했다. 그리고 선언된 여러 포인트컷을 식별하기 위한 유일한 아이디를 지정했으며, 이후에 애스팩트 설정에서 특정 포인트컷을 참조할 수 있다.

applicationContext.xml 수정

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

어노테이션 설정으로 포인트컷을 선언할 때는 @Pointcut을 사용하며, 하나의 어드바이스 클래스 안에 여러 개의 포인트컷을 선언할 수 있다. 여러 개의 포인트컷을 식별하기 위한 식별자가 필요한데, 이 때 참조 메소드를 이용한다.

참조 메소드는 메소드 몸체가 비어있는, 즉 구현 로직이 없는 메소드이다. 따라서 어떤 기능 처리를 목적으로 하지 않고 단순히 포인트컷을 식별하는 이름으로만 사용된다.

LogAdvice.java 예시

```
package com.spring.common;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

@Service
```

```

public class LogAdvice {
    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

    @Pointcut("execution(* com.spring..*Impl.get*(..))")
    public void getPointcut(){}
}

```

위 예에서 allPointcut()과 getPointcut() 메소드 위에 각각 @Pointcut을 이용하여 두 개의 포인트컷을 선언한다. 그러면 이 후이 이 포인트컷을 참조할 때 @Pointcut이 붙은 참조 메소드 이름을 이용하여 특정 포인트컷을 지정할 수 있다.

(3) 어드바이스 설정

어드바이스 클래스에서 횡단 관심에 해당하는 어드바이스 메소드가 구현되어 있다. 이 어드바이스 메소드가 언제 동작할지 결정하여 관련된 어노테이션을 메소드 위에 설정하면 된다. 어드바이스 동작 시점은 XML 설정과 마찬가지로 다섯 가지가 제공된다.

이때 반드시 어드바이스 메소드가 결합된 포인트컷을 참조해야 한다. 포인트컷을 참조하는 방법은 어드바이스 어노테이션 뒤에 괄호를 추가하고 포인트컷 참조 메소드를 지정하면 된다.

LogAdvice.java 수정

```

package com.spring.common;

import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

@Service
public class LogAdvice {
    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){ }

    @Before("allPointcut()")
    public void printLog(){
        System.out.println("[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작");
    }
}

```

allPointcut() 참조 메소드로 지정한 비즈니스 메소드가 호출될 때, 어드바이스 메소드인 printLog() 메소드가 Before 형태로 동작하도록 설정한 것이다.

어드바이스 동작 시점과 관련된 어노테이션

어노테이션	설명
@Before	비즈니스 실행 전에 동작
@AfterReturning	비즈니스 메소드가 성공적으로 리턴되면 동작
@AfterThrowing	비즈니스 메소드 실행 중 예외가 발생하면 동작
@After	비즈니스 메소드가 실행된 후 무조건 실행
@Around	호출 자체를 가로채 비즈니스 메소드 실행 전후에 처리할 로직을 삽입할 수 있음

(4) 애스팩트 설정

AOP 설정에서 가장 중요한 애스팩트는 @Aspect를 이용하여 설정한다. 애스팩트는 포인트컷과 어드바이스의 결합이다. 따라서 @Aspect가 설정된 애스팩트 객체에는 반드시 포인트컷과 어드바이스를 결합하는 설정이 있어야 한다.

LogAdvice.java 수정

```
package com.spring.common;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

@Service
@Aspect // Aspect = Pointcut + Advice
public class LogAdvice {

    @Pointcut("execution(* com.springexam.biz..*Impl.*(..))") // 포인트컷
    public void allPointcut(){}

    +

    @Before("allPointcut()") // 어드바이스
    public void printLog(){
        System.out.println("[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작");
    }

}
```

LogAdvice 클래스에 @Aspect가 설정되면 스프링 컨테이너는 LogAdvice 객체를 애스팩트 객체로 인식한다. 그리고 포인트컷 메소드(allPointcut())와 어드바이스 메소드(printLog())가 선언되어 있는데, 이 두 메소드에 설정된 어노테이션에 의해 위빙된다.

allPointcut() 메소드로 지정한 포인트컷 메소드가 호출될 때, printLog()라는 어드바이스 메소드가 실행되도록 설정한 것이다. 그리고 이 printLog() 메소드 위에 @Before가 설정되어 있어 printLog() 메소드는 사전 처리 형태로 동작한다.

15.2 어드바이스 동작 시점

XML 기반으로 설정했던 각 어드바이스들을 어노테이션으로 변경한다.

(1) Before 어드바이스

Before 어드바이스는 비즈니스 메소드가 실행되기 전에 공통으로 처리할 작업을 위해 사용한다.

BeforeAdvice.java 수정

```
package com.spring.common;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

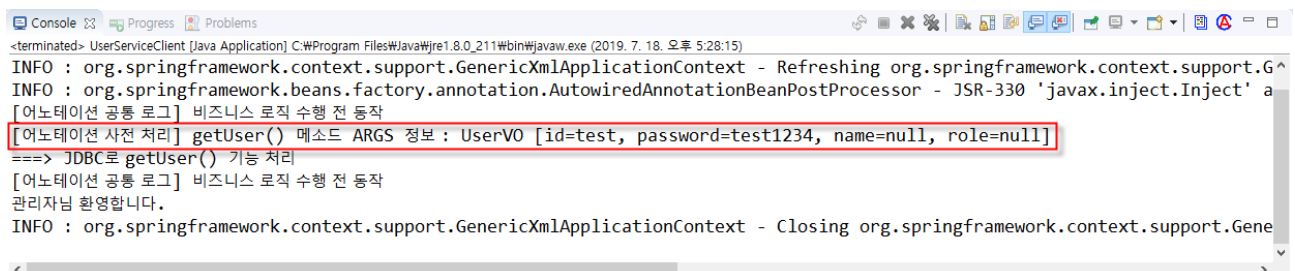
@Service
@Aspect
public class BeforeAdvice {

    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

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

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

UserServiceClient 클래스 실행



```
<terminated> UserServiceClient [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (2019. 7. 18. 오후 5:28:15)
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.G^
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' a
[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작
[어노테이션 사전 처리] getUser() 메소드 ARGS 정보 : UserVO [id=test, password=test1234, name=null, role=null]
====> JDBC로 getUser() 기능 처리
[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작
관리자님 환영합니다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.Gene
```

(2) After Returning 어드바이스

After Returning 어드바이스는 비즈니스 메소드가 리턴한 결과 데이터를 다른 용도로 처리할 때 사용한다.

AfterReturningAdvice.java 수정

```
package com.spring.common;
```



```
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;
import com.spring.user.UserVO;
```

@Service

@Aspect

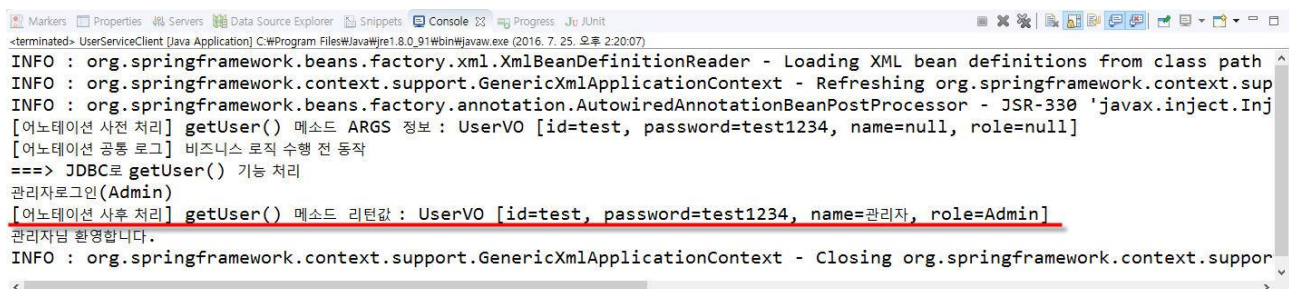
```
public class AfterReturningAdvice {

    @Pointcut("execution(* com.spring..*Impl.get*(..))")
    public void getPointcut(){}

    @AfterReturning(pointcut="getPointcut()", returning="returnObj")
    public void afterLog(JoinPoint jp, Object returnObj){
        String method = jp.getSignature().getName();
        if(returnObj instanceof UserVO){
            UserVO user = (UserVO) returnObj;
            if(user.getRole().equals("Admin")){
                System.out.println(user.getName() + "로그인(Admin)");
            }
        }
        System.out.println("[어노테이션 사후 처리] " + method + "() 메소드 리턴값 : " +
returnObj.toString());
    }
}
```

@AfterReturning은 @Before와 다르게 pointcut 속성을 이용하여 포인트컷을 참조하고 있다. 이는 After Returning 어드바이스가 비즈니스 메소드 수행 결과를 받아내기 위해 바인드 변수를 지정해야 하기 때문이다.

UserServiceClient 클래스 실행



```
<terminated> UserServiceClient [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016. 7. 25. 오후 2:20:07)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.GenericXmlApplicationContext
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' annotation found.
[어노테이션 사전 처리] getUser() 메소드 ARGS 정보 : UserVO [id=test, password=test1234, name=null, role=null]
[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작
===> JDBC로 getUser() 기능 처리
관리자로그인(Admin)
[어노테이션 사후 처리] getUser() 메소드 리턴값 : UserVO [id=test, password=test1234, name=관리자, role=Admin]
관리자님 환영합니다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.GenericXmlApplicationContext
<
```

(3) After Throwing 어드바이스

After Throwing 어드바이스는 비즈니스 메소드 실행 도중에 예외가 발생했을 때, 공통적인 예외 처리 로직을 제공할 목적으로 사용하는 어드바이스이다.

AfterThrowingAdvice.java 에 관련된 어노테이션 추가

```

package com.spring.common;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

@Service
@Aspect
public class AfterThrowingAdvice {

    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

    @AfterThrowing(pointcut="allPointcut()", throwing="exceptObj")
    public void exceptionLog(JoinPoint jp, Exception exceptObj){
        System.out.println("[예외 처리] 비즈니스 로직 수행 중 예외발생");
        String method = jp.getSignature().getName();
        System.out.println(method + "() 메소드 수행 중 예외 발생");
        if(exceptObj instanceof IllegalArgumentException){
            System.out.println("부적합한 값이 입력되었습니다.");
        } else if(exceptObj instanceof NumberFormatException){
            System.out.println("숫자 형식의 값이 아닙니다.");
        } else if(exceptObj instanceof Exception){
            System.out.println("문제가 발생했습니다.");
        }
    }
}

```

exceptionLog() 메소드가 After Throwing 형태로 동작하도록 메소드에 @AfterThrowing 어노테이션을 추가한다. @AfterThrowing도 pointcut 속성을 이용하여 포인트컷을 참조하고 있다. 비즈니스 메소드에서 발생한 예외 객체를 받아내기 위해 바인드 변수를 지정해야 하기 때문이다.

applicationContext.xml 예시

```

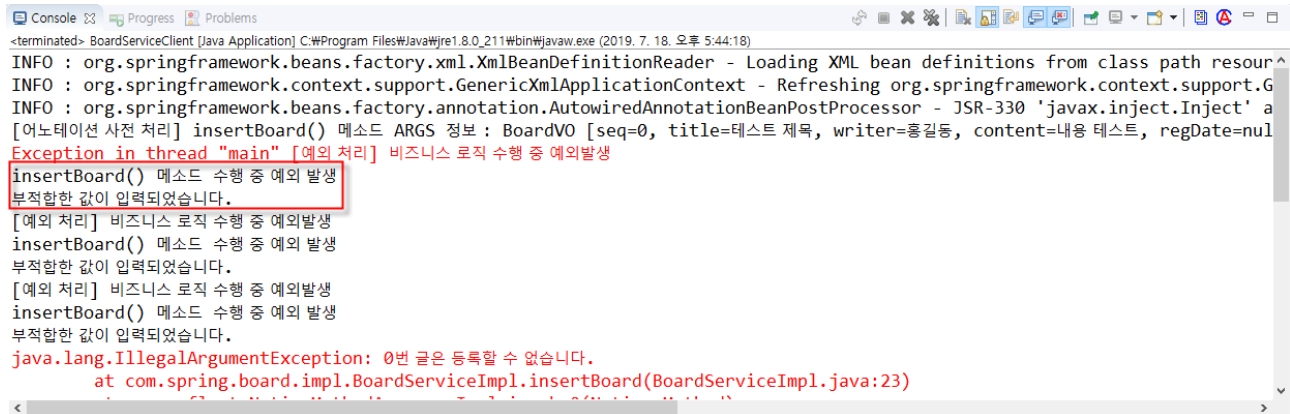
<bean id="afterThrowing" class="com.spring.common.AfterThrowingAdvice" />
<aop:config>
    <aop:pointcut expression="execution(* com.spring..*Impl.*(..))" id="allPointcut"/>
    <aop:aspect ref="afterThrowing">
        <aop:after-throwing method="exceptionLog" pointcut-ref="allPointcut"
throwing="exceptObj"/>
    </aop:aspect>
</aop:config>

```

BoardServiceImpl.java insertBoard() 메소드에서 if문 주석 해제한다.

```
@Override
public void insertBoard(BoardVO vo) {
    if(vo.getSeq() == 0){
        throw new IllegalArgumentException("0번 글은 등록할 수 없습니다.");
    }
    boardDAO.insertBoard(vo);
}
```

BoardServiceClient 클래스 실행하여 AfterThrowingAdvice가 동작하는지 확인한다.



```
<terminated> BoardServiceClient [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (2019. 7. 18. 오후 5:44:18)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.G
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' a
[어노테이션 사전 처리] insertBoard() 메소드 ARGS 정보 : BoardVO [seq=0, title=테스트 제목, writer=홍길동, content=내용 테스트, regDate=null]
Exception in thread "main" [예외 처리] 비즈니스 로직 수행 중 예외발생
insertBoard() 메소드 수행 중 예외 발생
부적절한 값이 입력되었습니다.
[예외 처리] 비즈니스 로직 수행 중 예외발생
insertBoard() 메소드 수행 중 예외 발생
부적절한 값이 입력되었습니다.
[예외 처리] 비즈니스 로직 수행 중 예외발생
insertBoard() 메소드 수행 중 예외 발생
부적절한 값이 입력되었습니다.
java.lang.IllegalArgumentException: 0번 글은 등록할 수 없습니다.
    at com.spring.board.impl.BoardServiceImpl.insertBoard(BoardServiceImpl.java:23)
```

(4) After 어드바이스

After 어드바이스는 예외 발생 여부에 상관없이 무조건 수행되는 어드바이스로서 @After 어노테이션을 사용하여 설정한다.

AfterAdvice.java 수정

```
package com.spring.common;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

@Service
@Aspect
public class AfterAdvice {

    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

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

finallyLog() 메소드가 After Advice 형태로 동작하도록 @After 어노테이션을 선언한다. 그리고 finallyLog() 메소드에 바인드 변수가 없으므로 @After 설정은 @Before처럼 포인트컷 메소드만 참조하면 된다.

BoardServiceClient 클래스 실행

실행 후 BoardServiceImpl.java의 insertBoard() 메소드에서 if문 주석 처리한다.

(5) Around 어드바이스

Around 어드바이스는 하나의 어드바이스로 사전, 사후 처리를 모두 해결하고자 할 때 사용하며 @Around 어노테이션을 상요하여 설정한다.

AroundAdvice.java 수정

```
package com.spring.common;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;
import org.springframework.util.StopWatch;

@Service
@Aspect
public class AroundAdvice {

    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

    @Around("allPointcut()")
    public Object aroundLog(ProceedingJoinPoint pjp) throws Throwable {

        String method = pjp.getSignature().getName();

        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        Object obj = pjp.proceed();
```

```

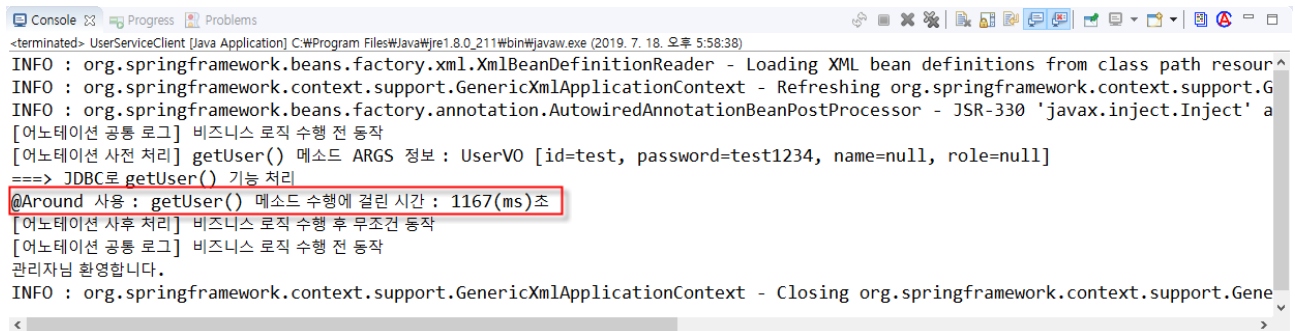
        stopWatch.stop();
        System.out.println("@Around 사용 : " + method + "() 메소드 수행에 걸린 시간 : " +
stopWatch.getTotalTimeMillis() + "(ms)초");

        return obj;
    }
}

```

aroundLog() 메소드에 바인드 변수가 없으므로 포인트컷 메소드만 참조한다. 어드바이스 메소드 중에서 유일하게 Around 어드바이스에서만 JoinPoint가 아닌 ProceedingJoinPoint 객체를 매개변수로 받는다. 그래야 proceed() 메소드를 이용하여 클라이언트가 호출한 비즈니스 메소드를 실행할 수 있기 때문이다.

UserServiceClient 클래스 실행



```

<terminated> UserServiceClient [Java Application] C:\Program Files\Java\jre1.8.0_211\bin\javaw.exe (2019. 7. 18. 오후 5:58:38)
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader - Loading XML bean definitions from class path resource
INFO : org.springframework.context.support.GenericXmlApplicationContext - Refreshing org.springframework.context.support.G
INFO : org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject' a
[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작
[어노테이션 사전 처리] getUser() 메소드 ARGS 정보 : UserVO [id=test, password=test1234, name=null, role=null]
==> JDBC로 getUser() 기능 처리
@Around 사용 : getUser() 메소드 수행에 걸린 시간 : 1167(ms)초
[어노테이션 사후 처리] 비즈니스 로직 수행 후 무조건 동작
[어노테이션 공통 로그] 비즈니스 로직 수행 전 동작
관리자님 환영합니다.
INFO : org.springframework.context.support.GenericXmlApplicationContext - Closing org.springframework.context.support.Gene

```

(6) 외부 Pointcut 참조

XML 설정으로 포인트컷을 관리했을 때는 스프링 설정 파일에 포인트컷을 여러 개 등록했다. 그리고 애스팩트를 설정할 때 pointcut-ref 속성으로 특정 포인트컷을 참조할 수 있었기 때문에 포인트컷을 재사용할 수 있다.

applicationContext.xml 예시

```

<aop:config>
    <aop:pointcut expression="execution(* com.spring..*Impl.*(..))" id="allPointcut" />
    <aop:pointcut expression="execution(* com.spring..*Impl.get*(..))" id="getPointcut" />
    <aop:aspect ref="log">
        <aop:brfore method="printLog" pointcut-ref="allPointcut" />
    </aop:aspect>
</aop:config>

```

어노테이션 설정으로 변경하고부터는 어드바이스 클래스마다 포인트컷 설정이 포함되면서, 비슷하거나 같은 포인트컷이 반복 선언되는 문제가 발생한다. 스프링은 이런 문제를 해결하고자 포인트컷을 외부에 독립된 클래스에 따로 설정하도록 한다.

시스템에서 사용할 모든 포인트컷을 PointcutCommon 클래스에 등록한다.

PointcutCommon.java

```

package com.spring.common;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

```

```

@Aspect
public class PointcutCommon {
    @Pointcut("execution(* com.spring..*Impl.*(..))")
    public void allPointcut(){}

    @Pointcut("execution(* com.spring..*Impl.get*(..))")
    public void getPointcut(){}
}

```

이렇게 정의된 포인트컷을 참조하려면 클래스 이름과 참조 메소드 이름을 조합하여 지정해야 한다. 사전 처리 기능의 BeforeAdvice 클래스를 수정한다.

BeforeAdvice.java 수정

```

package com.spring.common;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;

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

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

```

포인트컷에 대한 소스는 삭제되었고 @Before 어노테이션 PointcutCommon 클래스의 allPointcut() 메소드를 참조하고 있다. 클라이언트 프로그램 실행 결과는 같다.

바인드 변수가 있을 때도 포인트컷 클래스의 메소드를 참조하는 것과 같다.

사후 처리 기능의 AfterReturningAdvice 클래스를 수정한다.

AfterReturningAdvice.java 수정

```

package com.spring.common;

```

```

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Service;
import com.spring.user.UserVO;

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

        if(returnObj instanceof UserVO){
            UserVO user = (UserVO) returnObj;
            if(user.getRole().equals("Admin")){
                System.out.println(user.getName() + "로그인(Admin)");
            }
        }

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

```

UserServiceClient 클래스 실행 결과는 같다.