AOP(Aspect Oriented Programming)

기능을 핵심 비지니스 로직과 공통 모듈로 구분하고, 핵심 로직에 영향을 미치지 않고 사이사이에 공통모듈을 효과적으로 잘 끼워넣도록 하는 개발 방법이다.
공통 모듈(보안 인증, 로깅 같은 요소등)을 만든 후에 코드 밖에서 이 모듈을 비지니스 로직에 삽입하는 것이 바로 AOP 적인개발이다. 코드 밖에서 설정된다는 것이 핵심이다.

AOP가 사용되는 경우

1) 간단한 메소드 성능 검사
   개발 도중 특히 DB에 다량의 데이터를 넣고 빼는 등의 배치 작업에 대하여 시간을 측정해보고 쿼리를 개선하는 작업은 매우 의미가 있다. 이 경우 매번 해당 메소드 처음과 끝에 System.currentTimeMills()를 사용하거나 스프링이 제공하는 StopWatch 코드를 사용하기는 매우 번거롭다.
   이런 경우 해당 작업을 하는 코드를 밖에서 설정하고 해당 부분을 사용하는 편이 편리하다.
2) 트랜잭션 처리
   트랜잭션의 경우 비지니스 로직의 전후에 설정된다. 하지만 매번 사용하는 트랜잭션(try~catch부분)의 코드는 번거롭고, 소스를 더욱 복잡하게 보여준다.
3) 예외 반환
   스프링에는 DataAccessException이라는 매우 잘 정의되어 있는 예외 계층 구조가 있다. 예전 하이버네이트 예외들은 몇 개 없었고 그나마도 Uncatched Exception이 아니였다. 이렇게 구조가 별로 안 좋은 예외들이 발생했을 때, 그걸 잡아서 잘 정의되어 있는 예외 계층 구조로 변환해서 다시 던지는 애스팩트는 제3의 프레임워크를 사용할 때, 본인의 프레임 워크나 애플리케이션에서 별도의 예외 계층 구조로 변환하고 싶을 때 유용하다.
4) 아키텍쳐 검증
5) 기타
   - 하이버네티스와 JDBC를 같이 사용할 경우, DB 동기화 문제 해결
   - 멀티쓰레드 Safety관련하여 작업해야 하는 경우, 메소드들에 일괄적으로 락을 설정하는 애스팩트
   - 데드락 등으로 인한 PessimisticLockingFailureException등의 예외를 만났을 때 재시도하는 애스팩트
   - 로깅, 인증, 권한 등

AOP의 구성요소

- 조인포인트(joinPoint)
  : 횡단 관심 모듈의 기능이 삽입되어 동작할 수 있는 실행 가능한 특정위치
  ex) 메소드가 호출되는 부분 또는 리턴되는 시점, 필드를 액세스하는 부분, 인스턴스가 만들어지는 지점, 예외가 던져지는 시점, 예외 핸들러가 동작하는 위치, 클래스가 초기화되는 곳 등이 대표적인 조인포인트가 될 수 있다. 각각의 조인포인트들은 그 안에 횡단 관심의 기능이 AOP에 의해 자동으로 추가되어져서 동작할 수 있는 후보지가 되는 것이다.

- 포인트컷(pointCut)
  : 어떤 클래스의 어느 조인포인트를 사용할 것인지를 결정하는 선택 기능
  AOP가 항상 모든 모듈의 모든 조인포인트를 사용할 것이 아니기 때문에 필요에 따라 사용해야 할 모듈의 특정 조인포인트를 지정할 필요가 있다. 일종의 조인포인트 선정 룰과 같은 개념이다. AOP에서는 포인트컷을 수행할 수 있는 다양한 접근 방법을 제공한다.
  AspectJ에서는 와일드카드를 ㄹ이용한 메소드 시그니처를 사용한다.

어드바이스(advise) 또는 인터셉터(intercepter)

- 어드바이스 : 각 조인포인트에 삽입되어져 동작할 수 있는 코드
  주로 메소드 단위로 구성된 어드바이스는 포인트컷에 의해 결정된 모듈의 조인포인트에서 호출되어 사용된다. 일반적으로 독립적인 클래스 등으로 구현된 횡단 관심 모듈을 조인포인트의 정보를 참조해서 이용하는 방식으로 작성된다.
- 인터셉터 : 인터셉터 체인 방식의 AOP 툴에서 사용하는 용어로 주로 한 개의 invoke 메소드를 가지는 어드바이스

- 어드바이스(advise)의 종류
  - Before advice : 메서드 실행전에 적용되는 실행
  - After returning advice : 메서드가 정상적으로 실행된 후에 실행 (예외를 던지는 상황은 정상적인 상황에서 제외)
  - After throwing advice : 예외를 발생시킬 때 적용되는 Advice를 정의(catch오 비슷)
  - Around advice : 메서드 호출 이전, 이후 , 예외 발생 등 모든 시점에서 적용 가능한 Advice를 정의

위빙(weaving) 또는 크로스컷팅(crowssCutting)

- 위빙
  : 포인트컷에 의해서 결정된 조인포인트에 지정된 어드바이스를 삽입하는 과정(다른 말로 크로스컷팅)
  위빙은 AOP가 기존의 핵심 관심 모듈의 코드에 전혀 영향을 주지 않으면서 필요한 횡단 관심 기능을 추가할 수 있게 해주는 핵심적인 처리과정이다.
  위빙을 처리하는 방법은 후처리기를 통한 코드생성 기술을 통한 방법부터 특별한 컴파일러 사용하는 것, 이미 생성된 클래스의 정적인 바이트코드의 변환 또는 실행 중 클래스로

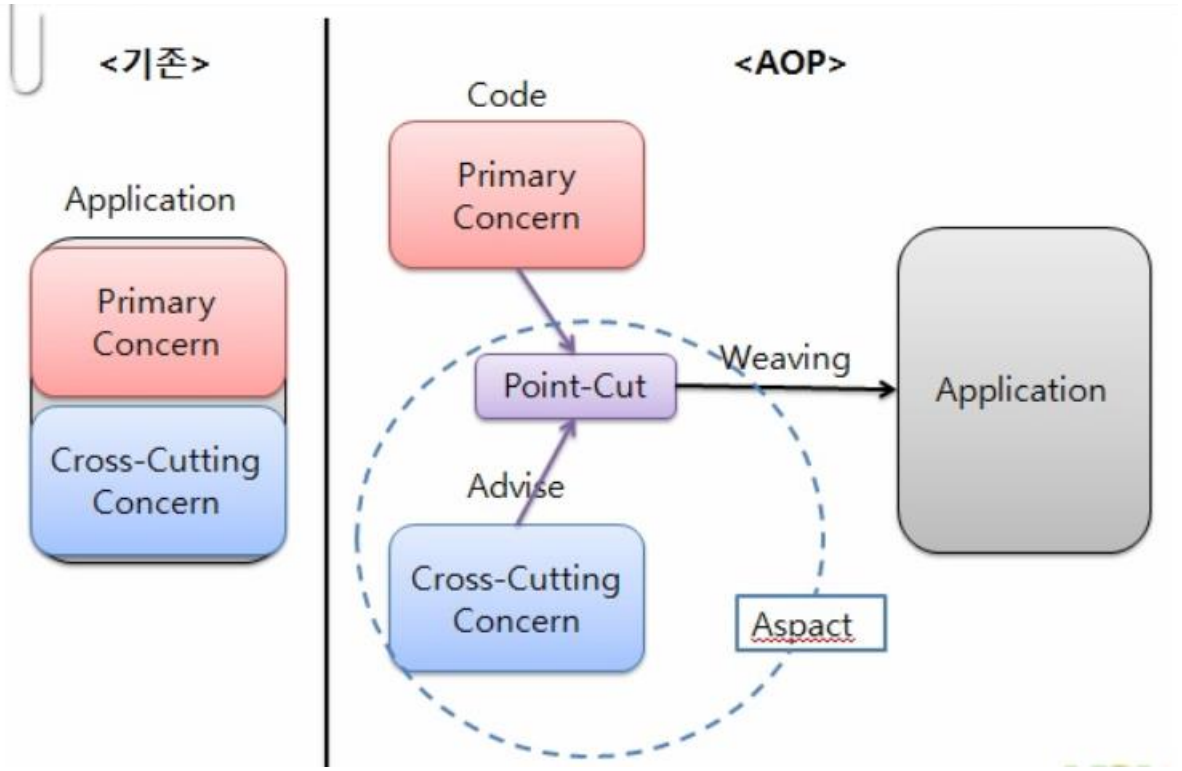
더를 통한 실시간 바이트코드 변환 그리고 다이내믹 프록시를 통한 방법까지 매우 다양하다.

인트로덕션(Introduction) 또는 인터타입 선언

- 인트로덕션 : 정적인 방식의 AOP 기술
  동적인 AOP 방식을 사용하면 코드의 조인포인트에 어드바이스를 적용해서 핵심 관심 콘의 동작방식을 변경할 수 있다.
  인트로덕션은 이에 반해서 기존의 클래스와 인터페이스에 필요한 메소드나 필드를 추가해서 사용할 수 있게 해주는 방법
  OOP에서 말하는 오브젝트의 상속이나 확장과는 다른 방식으로 어드바이스 또는 애스팩트를 이용해서 기존 클래스에 없는 인터페이스등을 다이내믹하게 구현해 줄 수 있다.

애스팩트(aspect) 또는 어드바이저

- 애스팩트
  : 포인트컷(어디에서) + 어드바이스(무엇을 할 것인지) + (필요에 따라 인트로덕션도 포함)
  AspectJ와 같은 자바 언어를 확장한 AOP에서는 마치 자바의 클래스처럼 애스팩트를 코드로 작성할 수 있다. AOP 툴의 종류에 따라서 어드바이스와 포인트컷을 각각 일반 자바 클래스로 작성하고 이를 결합한 어드바이저 클래스를 만들어서 사용하는 방법도 있다.



Primary(Core) Concern : 비지니스 로직을 구현한 부분
Cross-Cutting Concern : 보안, 인증, 로그 등과 같은 부가적인 기능으로서 시스템 전반에 산재되어 사용되는 기능
Code : Primary(Core) concern을 구현해 놓은 코드

기존의 코드는 Primary Concern과 Cross-Cutting Concern이 같이 하나의 프로그램으로 구현되어 졌다. 당연히 비지니스 로직과 상관없는 코드들이 여기저기 산재해 있게 되었기에 가독성과 유지보수성이 좋지 않았다. 하지만 AOP는 Primary Concern과 Cross-Cutting Concern이 별도로 코드로 구현이 되고, 최종적인 프로그램은 이(Code와 Advixe)를 연결해주는 설정 정보인 Point-Cut을 이용하여 Weaving되어 완성하게 되는 것이다.

AOP설정구조

```
<aop:config>
   <aop:aspect> : aspect 설정
    pointcut 설정
   <aop:pointcut id="aa" expression="execution(* part01_xml.ServiceImp.prn1(..))"/>
   <aop:pointcut id="bb" expression="execution(* part01_xml.ServiceImp.prn2(..))"/>
   <aop:pointcut id="cc" expression="execution(* part01_xml.ServiceImp.prn3(..))" />
   <aop:pointcut id="dd" expression="execution(* part01_xml.ServiceImp.prn4(..))" />
   <aop:pointcut id="ee" expression="execution(* part01_xml.ServiceImp.prn5(..))" />

   method 실행 전
   <aop:before method="comm1" pointcut-ref="aa"/>

   method 정상 실행 후
   <aop:after-returning   method="comm3" pointcut-ref="cc" returning="name"/>

   method 예외 발생시
   <aop:after-throwing method="comm4" pointcut-ref="dd" throwing="ex" />

   method 실행 후 (예외 발생 여부 상관없음)
   <aop:after method="comm2" pointcut-ref="bb" />

   모든 시점 적용가능
   <aop:around method="comm5" pointcut-ref="ee"/>
  </aop:aspect>
</aop:config>
```

PSA (Portable Service Abstraction)

포터블 서비스 추상화 (PSA)는 환경과 세부 기술의 변화에 관계없이 일관된 방식으로 기술에 접근할 수 있게 해준다. POJO로 개발된 코드는 특정 환경이나 구현 방식에 종속적이지 않아야 한다. 다시 말해, Spring은 POJO 원칙으로 만들었기 때문에 Spring 패키지 외의 것들을 POJO화 시키기 위해 껍데기를 씌우겠다는 것이다.
각 벤더들이 여러가지 인터페이스로 제공을 하더라도 Spring에서 Adapter pattern을 적용하여 제공하므로, 사용하는 클라이언트에서 공통된 메소드를 호출하는 형태로 구현하면 된다.

대표적인 PSA

1) JUnit : 스프링에서 지원하는 JUnit은 일반적인 JUnit과는 다르다.
2) MyBatis : MyBatis도 일반적인 MyBatis가 있고, Spring에서 지원하는 MyBatis는 다르다
3) JDBC(H2, HSQLDB, Apache Derby, MySQL, PostgreSQL)
4) Data (JDBC, JPA MongoDB, Redis, Gemfire, Solr, Elasticsearch)
5) Social (Facebook, Linkedin, Twitter 로그인)