cover story 3

DDD 개발의 찰떡궁합

스프링프레임워크와 DDD

스프링프레임워크(SpringFramework, 이하 스프링)의 첫 번째 메이저 업그레이드 버전인 2.00이 처음 발표되었을 때 개발자들이 가장 관심을 가지고 주목했던 것은 스프링을 통한 DDD(Driven Driven Design)였다. 대부분의 개발자들에게 아직 생소했던 개념이었던 DDD를 과감하게 주요 기능으로 내세우면서 등장한 스프링은 과연 DDD와 어떤 관련이 있는 것일까? 스프링2.x를 통해서 DDD를 적용하는 전략은 어떤 것들이 있는지 살펴보자.

이일민 tobyilee@gmail.com | 오픈소스 기술을 엔터프라이즈 시스템에 적용할 수 있는 전략과 기술연구에 많은 관심을 가지고 있다. 스프링프레임워크와 하이버네이트 등의 오픈소스 프레임워크의 교육, 컨설팅, 기술지원을 제공하고 있는 Epril의 대표 컨설턴트로 활동하고 있다. 한국스프링사용자모임의 기술고문으로도 활동하며 오픈소스 기술의 저변확대를 위해서 노력하고 있다. 오픈소스기술에 대한 정보를 공유하는 toby.epril.com 블로그를 운영하고 있다.

스프링2.0 버전은 2005년 말에 열렸던 TheSpring Experience 컨퍼런스에서 처음 소개가 되었다. 그때까지 1.3버전으로 준비되고 있었던 스프링의 차기 버전이 컨퍼런스 기간에 2.0이라는 메이저버전으로 버전자체의 업그레이드가 일어났다. 그 결정의 배경은 스프링의 다음 버전에 등장하는 기능들의 중요도와 변화의 영향력이 메이저 업그레이드를 하는 것이 바람직할 만큼 중요하다는 공감이 있었기 때문이다. 스프링 2.0에 소개된 새로운 기능에는 XML 설정의 단순화나 스키마의 도입 또 자바5의 새롭게 소개된 언어특성의 적용 등 그동안 지속적으로 요구되어졌기 때문에 당연히 들어갈 것으로 기대했던 내용들이 대거 포함되어 있었다. 그런데 그 중에서 필자의 눈을 사로잡는 생소한 단어가 눈에 띄었는데 그것이 바로 DDD다. TDD는 많이 들어봤어도 DDD는 무엇인가 궁금해하면서 DDD에 대해서 찾아보니 이미 몇 년 전부터 엔터프라이즈 아키텍처 기술에 관한 이야기들이 나올 때마다 자주 등장했던 Eric Evans의 책 제목인 Domain Driven Design의 약자인 DDD였다.

스프링 2.0

AOP, IoC/DI, PSA같은 시대를 앞서나가는 새로운 컨셉트와 기능들을 과감하게 소개하고 이를 현장에 접목시키는데 앞서왔던 스프링이었던지라 많은 스프링 개발자들은 지속적으로 엔터프라이즈 개발에 필요로 한 유용한 기술과 전략이 스프링의 업그

레이드과정을 통해서 소개될 것을 기대해왔다. 하지만 DDD라는 단어와의 만남은 그리 쉽게 예측할 수 없었던 것이라 스프링 커뮤니티와 관련 개발자들 사이에서도 적지 않은 화제를 불러일으키게 되었다. 대다수의 스프링 개발자들은 2.0에 대한 발표소식을 들으며, 함께 소개된 DDD에 대해서도 많은 관심을 가지기 시작했다.

2006년 가을에 스프링 2.0의 정식버전이 릴리즈 되고 나서 열렸던 첫 번째 스프링 컨퍼런스인 TheSpringExperience 2006에서는 컨퍼런스의 네 개 주요트랙중의 하나가 DDD였을 정도였으니 스프링과 DDD는 매우 깊은 관계를 가지고 있음을 짐작할 수 있다. TSE 2006 DDD트랙의 첫 번째 세션의 발표를 바로 Domain Driven Design이라는 책을 통해서 DDD라는 개념을 소개하고 이를 보급시키는데 앞장서고 있는 Eric Evans이 맡았다. Eric은 2007년 유럽에서 열린 스프링원 컨퍼런스에서도 키노트 발표를 담당했을 정도로 스프링과 깊은 관계를 가지고 있다.

DDD란 무엇인가?

이처럼 스프링과 DDD의 깊은 관계가 발생한 이유는 무엇일까? 그것은 스프링이 지금까지 지향해온 핵심가치와 DDD의 그것이 유사하기 때문이라고 생각된다.

스프링의 등장배경과 원리를 소개한 책인 J2EE Development without EJB라는 책의 첫 장에서는 스프링이 지향하는 핵심가치를 소개한다. 그 중 가장 눈에 띄는 것이 있다면 바로 OO(Object Oriented)라는 것이다. 스프링이 추구하는 핵심가치는 바로 객체지향이다. 자바라는 객체지향언어를 사용하는 프레임워크인 스프링이니 당연히 OO적일텐데 왜 그것을 핵심가치라고 하면서 강조하고 등장한 것일까? 그것은 자바의 근본이라고도 할 수 있는 객체지향이 사실은 엔터프라이즈 환경에서 상당히 무시당하고 있다는 안타까운 현실에서 나온 것이다. 스프링은 그것을 사용하는 개발자들에게 자바가 객체지향 언어라는 가장 기초적인 사실을 다시 상기시켜주면서 그 기초에 충실할 때에 가장 복잡하게 생각되는 엔터프라이즈 개발이 사실은 쉽고 단순해진다는 것을 알려주려는 의도를 가지고 만들어 진 것이다.

DDD 또한 그와 유사하다. DDD는 어떤 천재적인 기술자에게서 갑자기 소개된 새로운 개념과 아이디어가 아니다. 소프트웨어 설계라는 것이 존재했던 가장 초기부터 추구하고 가져왔던 가장 기초가 되는 기본에 다시 충실하자는 이야기이다. 그래서 어떤 사람들에게는 매우 진부하고 당연하게 들리기도 한다. Eric Evans의 DDD 책을 읽어본 많은 사람들이 의외로 실망하는 이유가 바로 거기에 있다. 이미 잘 알고 있다고 생각하는 당연한 내용들이 많기 때문이다. 하지만 그것에 충실하는 것이 의외로 현장에서 잘 이루어져 있지 않고, 세세한 전략들이 충분히 제시되고 있지 않다는 것이 현실임을 안다면 소프트웨어 설계의 기본으로 돌아가자는 DDD의 주장이 매우 강력하고 중요한 것이라는 점을 인식할 수 있을 것이다.

스프링프레임워크의 모체인 Interface21의 Ramnivas Laddad가 정리한 DDD의 세 가지 특징을 살펴보자.

첫째는 도메인 그 자체와 도메인 로직에 초점을 맞춘다는 것이다. 일반적으로 많이 사용하는 데이터중심의 접근법을 탈피해서 순수한 도메인의 모델과 로직에 집중하는 것을 말한다.

둘째는 보편적인(ubiquitous) 언어의 사용이다. 도메인 전문가와 소프트웨어 개발자 간의 커뮤니케이션 문제를 없애고 상호가 이해할 수 있고 모든 문서와 코드에 이르기까지 동일한 표현과 단어로 구성된 단일화된 언어체계를 구축해나가는 과정을 말한다. 이로서 분석 작업과 설계 그리고 구현에 이르기까지 통일된 방식으로 커뮤니케이션이 가능해진다.

셋째는 소프트웨어 엔티티와 도메인 컨셉트를 가능한 가장 가까이 일치시키는 것이다. 분석모델과 설계가 다르고 그것과 코드가 다른 구조가 아니라 도메인 모델부터 코드까지 항상 함께 움직이는 구조의 모델을 지향하는 것이 DDD의 핵심원리이다.

물론 DDD는 방법론이 아니다. 따라서 이런 경우 이렇게 하라는 식의 접근법 보다는 원칙과 핵심가치를 설명해주고 그것에 어떻게 집중할 것인가에 주목하게 하는 것이다. 모든 애플리케이션의 핵심은 결국 그 애플리케이션을 사용할 도메인과 그 로직이라고 본다면 소프트웨어 설계와 구현도 그 부분이 핵심이 되는 것이 마땅하다는 것이 DDD의 개념이라고 생각하면 된다.

DDD에서 스프링의 역할

그렇다면 스프링을 이용한 개발과 DDD는 무슨 직접적인 관련이 있는 것일까?

사실 스프링은 설계도구나 방법론이 아니다. 애플리케이션을 개발하는데 있어서 필요로 하는 기본 프레임워크일 뿐이다. 하지만 스프링은 처음부터 단순한 생산성과 품질향상을 위한 애플리케이션 프레임워크일 뿐만 아니라 개발자들에게 개발의 원칙과 지향해야 할 개발 원리를 설명하기 위해서 등장한 것이다. 따라서 스프링을 사용하면 자연스럽게 그 애플리케이션 코드는 스프링이 지향하는 개발원칙을 따라갈 수 있게 된다.

스프링의 핵심 가치는 위에서 언급한 객체지향(OO)과 단순함(Simplicity)이다.

자바는 처음 언어가 소개될 때 객체지향프로그래밍을 지원하는 강력한 언어라는 점을 가장 중요한 특징으로 내세웠다. 하지

만 자바가 본격적으로 사용되는 엔터프라이즈 환경의 코드들을 가만히 살펴보면 과연 객체지향의 특성들이 잘 살아있는지, 그 장점을 잘 활용하고 있는지 생각하면 의문스럽기만 하다.

● Anemic Domain Model

이는 흔히 말하는 빈약한 도메인모델(anemic domain model)라는 말에 잘 나타나 있다. 객체지향에서 말하는 오브젝트는 상태(state)와 행위(behavior)로 구성되어 있어야 한다. 하지만 자바엔터프라이즈 개발에서 흔히 사용되는 방식은 단지 상태 값, 즉 데이터만 가지는 데이터홀더 개념의 단순 오브젝트이다. 이런 오브젝트로 구성된 도메인모델은 객체지향언어와 기술의 장점을 전혀 살릴 수 없는 한계를 가지고 있다. 따라서 구조적으로 불합리한 형태의 코드를 생산하게 한다.

문제는 이런 빈약한 도메인모델과 오브젝트 구조가 거의 엔터프라이즈 자바 개발의 전반에 걸쳐서 사용되고 있다는 점이다. 대부분의 서버사이드 아키텍처라고 제시되는 구조가 빈약한 도메인모델의 사용을 부추기고 있다는 점이 문제다.

빈약한 오브젝트는 자바빈 형태로 나타나는 〈리스트 1〉과 같은 형태를 가진다.

〈리스트 1〉 빈약한 오브젝트(Anemic Object)의 예

```
class Customer {
        Integer customerId
        String name
        String telephone
        String address
        Integer point
        Date lastVisited

        public Integer getCustomerId() {
                return customerId
        }
        public void setCustomerId(Integer customerId) {
                this.customerId = customerId;
        }
        public String getName() {
                return name
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getTelephone() {
                return telephone
        }
        public void setTelephone(String telephone) {
                this.telephone = telephone;
        }
        public String getAddress() {
                return address
        }
        public void setAddress(String address) {
                this.address = address;
        }
        public Integer getPoint() {
                return point
        }
        public void setPoint(Integer point) {
                this.point = point;
        }
        public Date getLastVisited() {
                return lastVisited
        }
        public void setLastVisited(Date lastVisited) {
                this.lastVisited = lastVisited;
        }
}
```

빈약한 오브젝트가 가져오는 문제는 단지 객체지향 기술의 순수주의자들의 주장처럼 모든 객체는 행위가 포함 되야 한다는 정도에 그치는 것이 아니다. 모든 도메인은 행위에 해당하는 로직을 가지고 있고, 그 로직을 어떠한 형태로든 나타내야 하는데 그것이 도메인의 상태 정보만 가지고 있는 빈약한 오브젝트 속에서 모두 빠져나와 다른 형태로 구성된다는 점이다. 대부분의 자바엔터프라이즈 아키텍처가 가지고 있는 이런 구조적인 한계들은 결국 과도한 서비스 레이어의 사용을 부추긴다.

도메인오브젝트의 데이터홀더(data holder)화와 서비스레이어의 비대해짐은 결국 트랜잭션 스크립트 패턴의 형태의 코드를 양산하게 된다. 트랜잭션 스크립트는 각 업무 트랜잭션을 하나의 메소드에 한 번에 구성하는 형태의 패턴이다. 이런 형태의 코드는 결국 객체지향적인 도메인모델의 장점을 살릴 수 없는 결과로 나아가게 된다.

〈리스트 2〉는 트랜잭션 스크립트 스타일의 서비스레이어의 예이다. Customer와 PointRule이라는 두 개의 도메인 오브젝트가 존재하지만 각 도메인에 종속된 로직들이 서비스레이어의 메소드에 산재되어 있는 구조이다. addPoint 메소드는 Customer, PointRule을 받아서 또 다른 서비스 메소드를 사용해서 다 도메인 오브젝트의 로직에 해당하는 코드를 처리한다. 또한 withIn OneMonth 메소드나 isVipCutomer 메소드의 경우도 도메인 오브젝트에 포함 되야 하는 내용을 서비스 레이어에 노출한 형태이다.

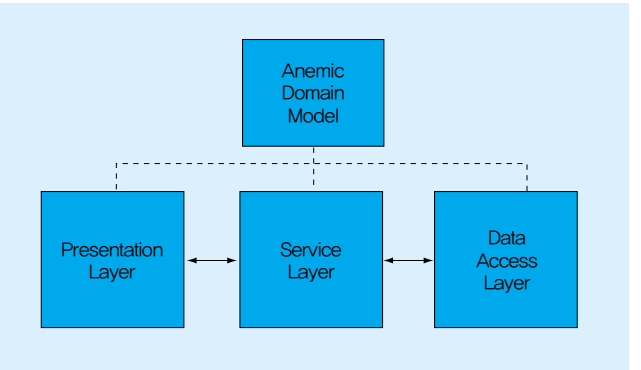
이런 형태의 거대한 서비스 레이어(big service layer) 형태는 객체지향의 설계원칙에 맞지 않을 뿐더러 도메인로직을 여러 곳에 산재하게 만들 뿐더러 코드의 중복과 오브젝트의 재활용성을 극히 떨어뜨리게 한다.

〈리스트 2〉 Big Service Layer의 예

```
class CustomerService {
    CustomerDao customerDao;
    PointRuleDao pointRuleDao;

    public void addPoints() {
        List<Customer> customers =
customerDao.getAllCustomers();
        PointRule pointRule =
pointRuleDao.getCurrentPointRule();

        for(Customer customer : customers) {
            if (withInOneMonth(customers.getLastVisited()))
{
                if (isVipCustomer(customer, pointRule)) {
                    customer.setPoint(customer.getPoint() +
VIP_POINTS);
                }
                else {
                    customer.setPoint(customer.getPoint() +
MEMBER_POINTS);
                }
            }
        }
    }

    boolean withInOneMonth(Date lastVisited) {
        ...
    }

    boolean isVipCustomer(Customer customer, PointRule
pointRule) {
        ...
    }
}
```

데이터중심의 설계와 개발에 익숙한 개발자들은 적절한 도메인모델을 이용해서 개발하는데 경험이 없거나 익숙하지 않기 때문에 이런 방식을 선호하거나 별 문제의식 없이 사용하는 경우가 많다. 또 어떤 기술은 기술자체가 이런 구조적인 한계를 노출하고 있다. 대표적인 것이 EJB의 엔티티빈 기술이다. 스프링을 비롯한 새로운 프레임워크와 기술들이 POJO를 선호하는 이유가 있다면 이 또한 자바의 객체지향적인 특징의 기본으로 돌아갈 수 있는 가장 단순하면서도 이상적이기 때문이다.

〈그림 1〉은 전형적인 3-tier 구조의 J2EE 아키텍처이다. 도메인 로직이 서비스 레이어에 집중되어있고 도메인모델 오브젝트는 단지 DTO와 같은 역할을 하는 데이터홀더로 사용되는 형태이다. 사실 이런 형태가 자바 엔터프라이즈 개발에서 거의 표준과 같이 인식되고 있다는 것은 심각한 문제이다. 심지어는 초기 스프링의 예제나 스프링 개발자들에 의해서 쓰여진 스프링 서적의 샘플 코드도 이런 구조를 그대로 사용했다는 것은 이런 모델이 얼마나 자연스럽게 개발자들에게 받아들여지고 사용되어져 왔는지 짐작하게 해준다.



〈그림 1〉 Anemic domain model + Big service layer 아키텍처

● Rich Domain Model

빈약한 도메인모델의 한계와 문제점을 인식한 개발자들은 점차로 풍성한 도메인 모델(rich domain model)이나 지능적인 도메인 모델(smart domain model)이라고 불리는 형태로의 전환을 시도한다.

지능적 도메인 모델의 특징은 도메인 오브젝트에 단순한 데이터 값의 저장을 위한 getter/setter가 아닌 도메인과 직접 관련이 되어있는 로직을 담았다는 것이다. 제한적이지만 도메인에 극히 종속적인 로직은 도메인 오브젝트에 담았기 때문에 포터블한 도메인 오브젝트로 발전할 뿐더러 상당수의 로직 부분이 서비스 레이어에서 제거 될 수 있게 하는 좋은 효과를 가져왔다.

〈리스트 3〉 풍성한 도메인 오브젝트의 적용 예

```
class Customer {
        Integer customerId
        String name
        String telephone
        String address
        Integer point
        Date lastVisited
        Date registered

        public booean isVipCustomer(PointRule pointRule)
{
                return this.customerLevel >
pointRule.getVipLevel(this.registered)
                        &&
isValidRegiststedCustomer();
        }

        public voiddoVipPointUpgrade(PointRule pointRule)
{
                if (isVipCustomer(pointRule) && ...)
```

```
customer.setPoint(customer.getPoint() +
pointRule.VIP_POINTS);
                }
                else {

customer.setPoint(customer.getPoint() +
pointRule.MEMBER_POINTS);
                }
        }
        ...

}
```

〈리스트 3〉에서는 기존에 서비스 레이어에 있던 도메인 로직이 도메인 오브젝트 안으로 이동한 모습을 볼 수 있다. 이렇게 도메인의 로직이 이동을 하게 되면 서비스 레이어에 중복되고 산재해서 나타나던 도메인 로직이 사라지고 일관성 있게 도메인 오브젝트에 처리를 맞길 수 있는 형태로 발전하게 된다. 〈리스트 4〉는 풍성한 도메인 오브젝트를 사용하는 서비스레이어의 변화된 모습이다.

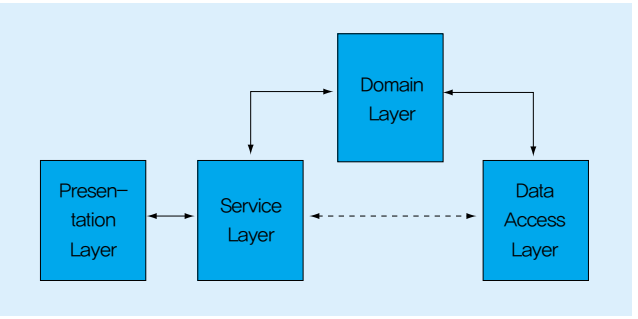
〈리스트 4〉 풍성한 도메인 오브젝트를 사용하는 서비스 레이어의 예

```
class CustomerService {
        CustomerDao customerDao;
        PointRuleDao pointRuleDao;

        public void addPoints() {
                List<Customer> customers =
customerDao.getAllCustomers();
                PointRule pointRule =
pointRuleDao.getCurrentPointRule();

                for(Customer customer : customers) {

customer.doVipPointUpgrade(pointRule);
                }
        }
}
```

● DDD 아키텍처

풍성한 도메인 모델을 사용하는 것만으로도 도메인 모델 중심의 구현과 일치가 어느 정도 가능해졌다. 하지만 DDD에서 지향하는 도메인 레이어라는 개념은 이보다 더 발전된 형태의 아키텍처이다.

〈그림 2〉는 기존의 3-tier 아키텍처와는 다른 형태의 DDD에서 많이 사용하는 구조이다. 기존의 아키텍처와의 가장 큰 변화라면 도메인 레이어의 도입이다. 기존의 도메인 모델/오브젝트는 단지 DTO의 역할을 하거나 데이터 홀더로 사용되어져 왔다면 DDD의 도메인 모델은 독립적인 형태의 레이어를 이루고 있는 것을 볼 수 있다.



〈그림 2〉 DDD아키텍처

도메인 레이어는 비즈니스 애플리케이션의 심장과 같은 역할을 한다. 모든 도메인 정보와 기능과 로직과 룰이 이 레이어에 집중되고 관리된다. 도메인 전문가와 소프트웨어 개발자들이 지속적으로 함께 살펴보며 다듬어 나갈 수 있는 그 도메인모델을 그대로 반영할 수 있는 구조가 만들어지게 되는 것이다.

이 경우 서비스 레이어는 비즈니스 로직을 직접 가지고 있지 않으면서 소프트웨어의 기능들을 적절한 도메인 레이어와의 협력을 통해서 처리할 수 있는 코디네이터 역할을 담당하게 된다. 따라서 서비스 레이어는 매우 얇아지게 된다.

또 한 가지 큰 변화는 저장소(repository) 또는 데이터액세스 레이어와의 연동을 이제는 도메인 레이어가 직접하게 된다는 것이다. 기존의 풍성한 도메인 모델에서는 도메인모델에게 넘겨줄 필요한 도메인 객체의 생성을 서비스레이어가 담당하게 되었다. 따라서 모든 레이어간의 연동이 서비스 레이어에 집중되는 한계를 가질 수밖에 없었다. 하지만 DDD 아키텍처에서는 도메인모델이 자신이 필요로 하는, 퍼시스턴스 관리를 위한 리포지토리 또는 인프라스트럭처 서비스와 직접 연동하는 것이 가능하다.

〈리스트 5〉는 도메인 레이어의 형태로 재구성한 Customer 클래스이다. 풍성한 도메인 모델 구조에서는 서비스 레이어가 데이터 액세스 레이어에 요청을 해서 PointRule를 받아 이를 Customer 오브젝트에 넘겨줘야 했지만, 여기서는 직접 PointRuleRepository에 요청해서 필요한 것을 받아서 사용할 수 있다. 따라서 도메인 오브젝트는 서비스 레이어의 도움이 없이도 독립적으로 도메인 로직을 처리할 수 있는 독립된 레이어 형태로 구성이 가능하게 된 것이다.

〈리스트 5〉 도메인 레이어의 Customer 클래스

```
class Customer {
        PointRuleRepository pointRuleRepository;
```

```
        Integer customerId
        ...
        Date lastVisited
        Date registered

        public booean isVipCustomer(PointRule pointRule)
{
                return this.customerLevel >
pointRule.getVipLevel(this.registered)
                        &&
isValidRegiststedCustomer();
        }

        public voiddoVipPointUpgrade() {

pointRuleRepository.getCurrentPointRule();

                if (isVipCustomer(pointRule))

customer.setPoint(customer.getPoint() + VIP_POINTS);
                }
                else {

customer.setPoint(customer.getPoint() + MEMBER_POINTS);
                }
        }
```

DI/AOP와 DDD

그렇다면 과연 이런 DDD의 아키텍처 구조와 스프링은 무슨 관계가 있다는 것인가? 사실 스프링이 직접적으로 DDD 아키텍처를 강제하거나 그런 구조로 만들어져 있는 것은 아니다. 스프링은 매우 범용적인 자바엔터프라이즈 개발에 사용되어질 수 있도록 만들어진 프레임워크이다. 그러면서도 DDD아키텍처를 적용할 때에 꼭 필요하다고 생각되는 기능을 제공한다.

도메인 모델와 의존삽입(DI)

앞에서와 같은 도메인 레이어를 만들 때 가지는 문제점에 대해서 생각해보자.

첫 번째 문제는 도메인 오브젝트에 리포지토리나 다른 인프라스트럭처 서비스를 어떻게 삽입 할 것인가의 문제가 있다. 스프링이 지원하는 의존삽입(DI)을 사용하면 간단하게 처리할 수 있을 것 같다. 하지만 도메인 오브젝트는 의존삽입을 적용하기에 적절하지 않다.

스프링에서 빈(bean)으로 등록할 수 있는 것과 또는 없는 것, 하지 말아야할 것을 구분할 때 항상 등장하는 예가 바로 도메인 오브젝트다. 도메인 오브젝트가 스프링의 빈으로 등록되지 말아야 하는 첫 번째 이유는 빈약한 도메인 모델처럼 단순한 데이터만을 가지는 형태이기 때문이다. 즉 의존관계가 없다는 것이다. 따라서 굳지 빈으로 등록할 이유가 없다. 그냥 new 키워드를 통해서 생성하면 된다.

두 번째 문제는 도메인 오브젝트의 생성이 애플리케이션 코드에서 직접 일어나지 않고 써드파티(3-rd party) 프레임워크 등에서 만들어지는 경우가 있기 때문이다. 대표적으로 O/R매핑 툴인 하이버네이트를 사용한다면 find나 get 등에 의해서 전달되는 도메인 오브젝트는 모두 하이버네이트 내부에서 직접 생성이 된다. 따라서 그 오브젝트의 라이프사이클을 스프링에 위임할 수 없기 때문에 빈으로 쓰는 것이 부적절하다.

문제는 빈약한 도메인 모델을 사용한다면 상관없지만, DDD에서의 요구되는 도메인 오브젝트를 사용하려면 외부의 서비스나 리포지토리의 삽입이 필수적으로 요구된다. 하지만 위의 두 가지 제약 때문에 따라서 단순히 빈으로 등록해서 사용하는 것은 불가능하다.

바로 이런 문제를 해결할 수 있는 방법으로 등장한 것이 스프링 2.0의 DDD지원기능이다. 엄밀히 말해서 스프링에 DDD기능이란 없다. 단지 위와 같은 특성을 가지고 있는 도메인 모델에 의존삽입이 가능하게 하기 위해서 스프링 2.0에 특별히 도입된 기능이 있을 뿐이다.

- @Configurable

사실 이를 위해서 스프링이 제공하는 기능은 무척 간단하다. @Configurable이라는 새롭게 도입된 어노테이션과 AspectJ의 도움으로 스프링이 직접 생성하지 않는 도메인 오브젝트에도 의존삽입이 가능하게 해주는 것이다. 원리는 간단하다. AspectJ가 지원하는 LTW(Load Time Weaving)기능을 이용해서 오브젝트가 생성되는 시점의 조인 포인트에서 오브젝트에 자동결합(autowiring) 방식으로 의존성을 삽입해주는 것이다.

스프링의 의존삽입 기능은 XML과 같은 형태로 외부에서 정의하는 방법 외에도 프로그래밍 적으로 사용할 수 있는 자동결합 방식도 가능하다. 이를 이용하면 빈의 이름과 일치하는 setter를 찾아서 자동으로 삽입을 시키는 것이 가능하다.

AspectJ 5의 LTW를 이용하면 별도의 AOP를 위한 컴파일 작업 없이도 클래스가 로딩되는 시점을 이용해서 어느 곳에서 생성되는지와 상관없이 자동으로 의존삽입을 시킬 수 있다. 따라서 @Configurable이 적용되어 있는 모든 도메인 오브젝트에 적절한 외부 의존성을 삽입시켜줄 수 있다.

〈리스트 6〉은 @Configurable을 정의한 Customer클래스이다. 사실 스프링 2.0의 DDD 지원기능은 막상 적용하려고 보면 무척 간단하다. 물론 그 내부에서 처리되어지는 것들은 매우 복

잡한 방식으로 이루어지고 있지만 말이다.

〈리스트 6〉 @Configurable이 적용된 Customer클래스

```
@Configurable
class Customer {
        PointRuleRepository pointRuleRepository;

        public void
setPointRuleRepository(PointRuleRepository
pointRuleRepository) {
                this.pointRuleRepository =
pointRuleRepository;
        }
        ...

}
```

● 도메인 모델과 AOP

의존삽입이 가능하다는 것은 모든 빈 형태로 존재하는 각종 서비스들을 다 이용할 수 있다는 것이다. 이 외에도 도메인 모델에 필요로 하는, AOP에서 자주 등장하는 횡단관심(Crosscutting Concerns)도 모두 적용할 수 있다. 트랜잭션 지원이나 로깅, 보안, 트레이싱 같은 것들을 도메인 오브젝트에 적용하는 것도 스프링 빈의 형태로 정의되기 때문에 모두 할 수 있다.

@Configurable을 정의하기 위해서는 스프링 설정파일에 다음과 같이 prototype 형태로 정의하는 것이 필요하다. 빈으로 등록된 이유는 일반적인 형태의 삽입 대상이 되기 위함은 아니다. 따라서 이 도메인 오브젝트 빈을 다른 빈의 의존관계로 만들면 안된다.

```
〈bean class = "com.mycompany.Customer" scope = "prototype"〉
        〈propety ... /〉
〈bean〉
```

빈의 형태로 등록이 되었기 때문에 스프링 AOP의 기능을 모두 적용할 수 있다. 스프링 2.0의 의존삽입과 AOP는 DDD를 본격적으로 구현해서 사용하려면 반드시 필요한 기술이라고 여겨지고 있다. 이제까지 풍성한 도메인 모델 정도의 구현에 만족했던 개발자들에게 도메인 레이어의 구분이라는 본격적인 DDD 구현의 실마리를 찾아준 것이다.

스프링 DDD의 적용전략

스프링이 제공해주는 DI/AOP를 이용하면 DDD의 도메인 레이어 구현을 위해서 필요로 하는 기술적인 요구사항은 충족될 수 있다. 하지만 이것만으로 바로 DDD방식의 구현에 도전하는 개발자들은 다들 한번쯤 좌절을 겪기 마련이다. 기존의 빈약한 도메인 모델구조에서는 경험하지 못했던 아키텍처상의 더 많은 변수들이 있기 때문이다.

첫 번째로 고민해 볼 것은 리포지토리의 사용방식이다.

스프링 시큐리티를 만든 Ben Alex나 스프링웹플로우의 Keith Donald는 하이버네이트와 같은 투명한영속성(transparent persistence) 방식의 ORM 기술을 선호하는 편이다. 그 이유는 투명한 영속성을 사용하는 방식은 명시적으로 리포지토리에 update문을 호출하지 않아도 퍼시스턴스 오브젝트에 발생하는 변화를 감지해서 자동이로 이를 데이터베이스와 싱크를 맞춰주기 때문이다. 또 도메인 오브젝트 그래프를 따라서 데이터를 자동으로 적절한 시점에서 가져오는 것도 가능하다. 따라서 도메인 로직에 빈번하게 리포지토리나 DAO를 호출하는 코드가 등장하지 않는다. 최초로 퍼시스턴스화 할 때와 초기조회, 삭제, 벌크 수정정도만이 사용되어지고 그 외의 퍼시스턴스 코드가 등장하지 않으니 이 방식을 사용한 코드는 순수한 자바 오브젝트만을 사용한 코드처럼 깔끔하고 명확하게 만들어질 수가 있다.

하지만 이 방식의 문제점을 지적하는 사람도 적지 않다. 대표적으로 역시 스프링의 핵심 개발자인 Ramnivas Raddad이다. 그는 하이버네이트나 JPA를 이용해서 투명한 영속성을 사용할 경우, 그 사용이 과도하게 되면 많은 데이터를 처리하는데 불필요한 오버헤드가 발생한다고 생각한다. 따라서 리포지토리의 데이터 로직을 좀 더 강화해서 매우 정교한 형태의 꼭 필요한 처리만 가능하게 하고 벌크처리는 DAO안에서 가능한 간략한 방식으로 일어나는 것이 좋다고 주장한다.

이렇게 될 경우 도메인 로직을 순수한 자바오브젝트로 표현하던 것을 일부 데이터베이스 스크립트 방식으로 만들게 되지만 전체 애플리케이션의 성능을 고려해 본다면 엔터프라이즈 환경에서는 충분히 타협할 수 있는 구조라는 것이다.

순수한 도메인 오브젝트 중심의 로직 구현방식과 데이터베이스와의 협조를 통한 중도적인 접근방법 중 어느 것이 항상 더 낫다고 볼 수는 없을 것 같다. 그럼에도 필자는 투명한 영속성을 지원하는 ORM을 사용하는 것이 DDD에서 가지는 장점이 매우 많다고 생각한다. 어쨌든 좀 더 오브젝트 중심의 코드가 만들어지기 때문이다. 다른 이유로 원천적으로 ORM 툴을 사용할 수 없는 경우가 아니라면, 그렇게 설계된 후에 필요에 따라서 성능을 위해서 개선하는 방식을 취하는 것이 바람직 할 것이다.

두 번째로 고려해야 할 것은 DTO이다.

ROO(Real Object Oriented)라는 애플리케이션 프레임워크를 만든 Ben Alex는 도메인 오브젝트는 도메인 레이어 밖에서는

존재해서는 안 된다고 주장한다. 따라서 그의 프레임워크에서는 도메인 레이어 밖으로 나오는 오브젝트는 DTO 형태로 복제가 되서 나온다. 서비스 레이어와 도메인 레이어 사이에 얇은 DTO를 위한 어셈블리 레이어를 가지고 있다.

또 모든 도메인 오브젝트는 근본적으로 수정가능하지 않아야(immutable) 한다고 주장하는 사람도 있다. 수정이 필요한 경우는 복제를 통해서 도메인 레이어에 요청해야 한다는 것이다.

필자와 같은 개발자들은 DTO를 쓰는 것을 바람직하지 않게 생각한다. 모든 레이어에 존재하는 도메인 모델(domain model everywhere) 방식을 선호한다. 이 경우 DTO를 지지하는 사람들이 문제로 지적하는 도메인 레이어 외부에서 변경이 일어난다는 문제가 발생한다.

예를 들어 프레젠테이션 레이어에서 임의로 도메인 오브젝트의 로직을 호출하거나 변경을 가할 수도 있다는 것이다. 이런 것을 원천적으로 차단하는 방법 중의 하나가 DTO이다. 하지만 도메인 오브젝트를 직접 사용한다고 할지라도 이런 문제를 해결할 방법이 있다.

역시 스프링 2.0에 등장하는 AspectJ를 이용한 SpringAOP를 적용하면 특정 레이어에서 특정 메소드에 접근하는 것을 강제적으로 차단할 수 있다. 이를 통해서 구지 매번 개발자들에게 개발 정책을 교육하고 매번 점검하지 않더라도 이를 편리하게 강제할 수 있다.

DDD의 미래

DDD는 많은 사람들의 주목을 받고 있는 것에 비해서 사실 상당히 인기가 없다. DDD에 대한 말은 많고 많은 개발자들이 이야기하고 있기는 하지만, 정작 이를 현장에 적용했다는 소식은 그다지 들리지 않는다.

이는 DDD에 대한 정확한 이해와 접근방법에 대한 학습이 부족한 원인도 있을 것이다. 하지만 더 큰 이유는 DDD가 단지 개발 기술이 아니기 때문이다. DDD 사실상 소프트웨어 설계에 관한 철학이다. 그 설계와 구현이 일치를 이뤄야 하기 때문에 스프링과 같은 툴들이 이를 지원하고 있을 뿐이다.

따라서 진정한 DDD를 적용하고 그 장점을 충분히 누리려면 사실 많은 연구와 훈련이 필요로 하다. 스프링과 같은 실전 개발을 위한 좋은 툴이 제공되고 있으니 이제 한번쯤 DDD에 대해서 깊이 연구해보고 도전해볼 때가 되지 않았을까?

참고 자료

1. Anemic Domain Model: http://martinfowler.com/bliki/AnemicDomainModel.html
2. Domian Driven Design with DI and AOP: Ramnivas Laddad
3. Spring 2.0 Reference Manual
4. Domain Driven Design: Eric Evans