클린 코드 11장 - 시스템

순수 자바 AOP 프레임워크

순수 자바 관점을 구현하는 스프링 AOP

- 스프링 AOP 내부적으로 프록시를 사용한다.
- 스프링 AOP를 사용하므로서 프록시 코드를 걷어내고 POJO 로직을 작성할 수 있게 된다.
- 스프링은 비지니스 논리를 POJO로 구현한다.

POJO

Plain Old Java Object

> 💡 Java EE 등의 중량 프레임워크들을 사용하게 되면서 해당 프레임워크에 종속된 "무거운" 객체를 만들게 된 것에 반발해서 사용되게 된 용어

pojo는 특정 프레임워크에 의존하지 않는 오래된 순수 자바 객체를 의미한다. 엔터프라이즈 프레임워크, 다른 도메인에도 의존하지 않는다.

따라서 테스트가 개념적으로 더 쉽고 간단하며, 상대적으로 단순하여 사용자 스토리를 올바르게 구현하기 쉽고 미래 스토리에 맞춰 코드를 보수하고 개선하기 편하다.

java의 JMS 기능에 의존하는 POJO가 깨진 코드

```
public class ExampleListener implements MessageListener {

  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      try {
        System.out.println(((TextMessage) message).getText());
      }
      catch (JMSException ex) {
        throw new RuntimeException(ex);
      }
    }
    else {
      throw new IllegalArgumentException("Message must be of type TextMessage");
    }
```

```
  }
}
```

POJO 코드

```
@Component
public class ExampleListener {

  @JmsListener(destination = "myDestination")
  public void processOrder(String message) {
    System.out.println(message);
  }
}
```

필수적인 애플리케이션 기반 구조를 위해 프로그래머는 많은 관심사를 구현한다. 이중에는 영속성, 트랜잭션,
보안, 캐시, 장애조치 등과 같은 **횡단 관심사**도 포함된다. 이때 프레임워크는 사용자가 모르게 프록시나 바이트코드 라이브러리를 사용해 구현한다.

이런 선언들이 요청에 따라 주요 객체를 생성하고 서로 연결하는 등 DI 컨테이너의 구체적인 동작을 제어하게 된다.

```
<beans>
  ...
  <bean id="appDataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"
  p:driverClassName="com.mysql.jdbc.Driver"
  p:url="jdbc:mysql://localhost:3306/mydb"
  p:username="me"/>

  <bean id="bankDataAccessObject"      // DAO
  class="com.example.banking.persistence.BankDataAccessObject"
  p:dataSource-ref="appDataSource"/>

  <bean id="bank"
  class="com.example.banking.model.Bank"
  p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

스프링 xml 설정을 통한 예를 봐보자.

Bank 도메인 객체는 자료 접근자 객체 DAO인 **bankDataAccessObject**로 프록시 되었으며,

bankDataAccessObject는 JDBC 드라이버 자료 소스인 **appDataSource**로 프록시 되었다.

이러한 구조를 통해 클라이언트는 Bank 객체에서 getAccounts()를 호출하여도 실제로는 Bank POJO의 기본동작을 확장한 중첩 DECORATOR 객체 집합의 최외각과 통신하게 된다.

여기에 필요하다면 트랜잭션, 캐싱 등에도 DECORATOR를 추가할 수 있다.

이를 통해 EJB2 시스템이 가졌던 강한 결합 문제가 사라진다. XML이라는 읽기 어려운 문제가 있지만 이런 설정파일에 명시된 정책이, 겉으로 보이지 않지만 자동으로 생성되는 프록시나 관점 논리보다는 단순하다.

EJB3는 **XML 설정 파일**과 **자바5 애노테이션** 기능을 사용해 **횡단 관심사**를 선언적으로 지원하는 스프링 모델을 따르게 되었다.

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Embeddable // An object “inlined” in Bank’s DB row
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }
    @Embedded
    private Address address;
    @OneToMany(
      cascade = CascadeType.ALL, fetch = FetchType.EAGER, mappedBy = "bank"
    )
    private Collection<Account> accounts = new ArrayList<Account>();
    public int getId() {
        return id;
```

```
    }
    public void setId(int id) {
        this.id = id;
    }
    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }
    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = accounts;
    }
}
```

- 일반적인 자바 객체 형태가 되었다.
- 일부 상세한 엔티티 정보는 애노테이션에 포함되어 그대로 남지만, 모든 정보가 애노테이션에 있어 코드 자체는 깔끔하고 깨끗하다.
- 코드를 개선하고 보수하기 쉬워졌다.

AspectJ

- 관심사를 분리하는 가장 강력한 도구이다
- 관점을 분리하는 강력하고 풍부한 도구 집합을 제공한다.
- 새 도구 및 새 언어 문법을 익혀야 한다는 단점이 존재한다.
- 이 책에서 벗어나는 내용이라 skip…

테스트 주도 시스템 아키텍쳐 구축

애플리케이션 도메인 논리를 POJO로 작성할 수 있다면, 코드 수준에서 아키텍쳐 관심사를 분리할 수 있다면
`테스트 주도 아키텍쳐 구축`이 가능해진다.

- 필요할 때 새로운 기술을 채택하여 단순한 아키텍쳐를 복잡한 아키텍쳐로 키워갈 수 있다.
- BDUF(**Big design up front**)를 추구할 필요가 없다. 심지어 더 해롭기까지 하다.
  - 기껏 크게 만들어놓은 노력을 버리지 않으려는 심리로 인해.
  - 미리 큰 그림으로 아키텍쳐를 구축하려는 구조

이를 통해 처음에는 아주 단순하면서도 분리된 아키텍쳐 소프트웨어 프로젝트를 진행하여 결과물을 낸 후, 기반 구조를 추가하며 조금씩 확장하는 방법으로 갈 수 있다.

의사 결정을 최적화

모듈을 나누고 관심사를 분리하므로서 지엽적인 관리와 결정이 가능하다.

- 가장 적합한 사람에게 책임을 맡기는게 좋다.
- 가능한 마지막 순간까지 결정을 미루는 방법이 최선이다.
    - 게으르거나 무책임해서가 아니라! 최대한 정보를 모아 최선의 결정을 내리기 위해서.
- 성급한 결정은 불충분한 지식으로 내린 결정이다.

명백한 가치가 있을 때 표준을 현명하게 사용하라.

- EJB2는 단지 표준이라는 이유로 널리 사용되었다. 가볍고 간단한 설계로 될 프로젝트에서도 사용되었다.
- 아주 과장되게 포장된 표준에 집착하여 고객 가치가 뒷전으로 밀려나는 사례가 있었다.

시스템 도메인 특화 언어가 필요하다.

DSL

Domain specific Language

- Java, Kotlin, Sql, Gradle, JMock, Kotest, Mockk 등등 상황에 맞는 언어를 사용하는 것.
- dsl은 간단한 스크립트 언어나 표준 언어로 구현한 API를 일컫는다.
- dsl로 작성한 코드는 도메인 전문가가 작성한 구조적인 산문처럼 읽힌다.

좋은 DSL은 도메인 개념과 그 개념을 구현한 코드사이에 존재하는 의사소통 간극을 줄여준다.

DSL을 사용하여 고차원 정책에서 저차원 정책까지 모든 추상화 수준과 모든 도메인을 POJO로 표현할 수 있다.

결론

시스템은 깨끗해야 한다. 깨끗하지 못한 아키텍처는 도메인 논리를 흐린다. 도메인 논리가 흐려지면 제품 품질이 떨어진다. 버그가 스며들기 쉬워지며 스토리를 구현하기 어려워진다.

1. POJO를 작성
2. 각 구현 관심사를 분리

위 단계를 통해 모든 추상화 단계에서 의도를 명확히 표현해야 한다.