토비의 스프링 3.1[ 스프링의 이해와 원리 ]

작성자: 이원태

[개요]

스프링은 객체지향 설계와 구현에 특정한 모델과 기법을 억지로 강요하지 않지만 오브젝트를 어떻게 효과적으로 개선 해나갈 것인가에 대한 기준을 마련해준다.

이를 위해 의도적으로 초난감 dao를 제시함으로써 리펙토링 하고자 한다.

초난감 DAO

개요 : 잘못된 설계를 짚어 나가며 개선한다.

* 장점: 없다.
* 단점: 중복된 코드와 여러가지 관심사가 한 메소드에 존재한다.

[User DAO의 관심사항]

* 1. DB Connection에 대한 관심사
  2. Statement에 담긴 SQL을 실행시키는 관심사
  3. Statement와 Connection 공유 리소스를 반환하는 관심사

[User DAO 중복된 코드]

DB Connection코드가 add()와 get()에 중복되어 있다.

* 해결 방안: 중복된 코드를 메소드로 추출하여 관심사를 분리(Separation of concerns)한다.
  + 관심이 같은 것끼리 하나의 객체로 모아 서로 영향을 주지 않도록 분리한다.
  + 중복된 DB Connection 코드를 getConnection메소드로 추출한다.

관심사를 분리한 DAO

개요 : 변화에 대응하는 수준이 아닌 변화를 반기는 DAO 만든다.

장점 : 변경된 관심사에 대한 코드만 수정하고 다른 관심사에는 영향을 주지 않는다.

단점 : N사와 D사에게 UserDao소스코드를 제공하지 않는다면, 고객이 원하는 DB컨넥션 메소드를 만들 수 없다.

해결방안 : 상속을 통해 서브클래스로 확장한다.

1. UserDAO는 abstract 키워드를 사용하여 추상 클래스로 만든다.
2. 추출된 getConnection 메소드는 구현 코드를 제거한 후 abstract 키워드를 사용하여 추상 메소드로 만든다.
3. 상속 받은 서브클래스는 추상 메소드를 재정의하여 확장 시킨다.

* 템플릿 메소드 패턴 :
  1. 변하지 않는 기능은 슈퍼클래스에 만들어 두고, 자주 변경되며 확장할 기능은 서브클래스에서 만든다.
  2. 슈퍼 클래스의 기능 일부를 추상, protected 메소드로 만든 뒤 서브클래스에서 필요에 맞게 재정의하는 디자인 패턴이다.\*
* 팩토리 메소드 패턴 :
  1. 슈퍼클래스는 객체를 생성하기 위한 인터페이스를 정의하며, 서브클래스는 어떤 클래스의 인스턴스를 만들지 결정한다.
     + UserDAO의 getConnection()는 Connection타입 오브젝트를 생성 한다는 기능을 정의하며 상속받은 서브클래스는 구체적인 오브젝트 생성 방법을 결정한다..\*

상속을 통해 서브클래스로 분리한 DAO

개요 : UserDAO는 클래스 분리 작업으로 상속이 가지는 단점을 해결한다.

장점 : 사용자가 원하는 DB 연결 방식으로 독립적인 메소드를 만들 수 있다.

단점 :

1. 서브 클래스가 상속을 사용하고 있다면 다른 목적의 상속은 적용 할 수 없다.
2. 슈퍼클래스에 변경이 있는 경우 서브클래스도 수정 해야하며 반대로 이런 변경을 주지 않기 위해 슈퍼클래스는 제약을 적용시켜야 한다. -> 두 문장? 또는 제약 설명?
3. 확장된 서브클래스의 메소드는 다른 슈퍼클래스에 적용 할 수 없다.

(다른 슈퍼클래스들이 계속 만들어진다면 서브클래스의 구현 메소드는 슈퍼클래스마다 중복된다.)

해결방안 : 완전히 독립적인 클래스로 분리한다.

1. UserDAO는 abstract를 제거한다.
2. UserDAO의 생성자는 new 키워드를 사용해 SimpleConnectionMaker 클래스의 오브젝트를 만들어두고, 이를 add(), get()에서 사용한다.
3. 독립시킨 SimpleConnectionMaker 클래스는 DB 커넥션 생성 기능을 그 안에 담는다..

클래스 분리한 DAO

개요 : 클래스 분리로 생긴 종속적인 문제는 인터페이스로 해결한다.

장점 : 손쉽게 확장할 수 있고 두 관심사에 대한 결합도가 낮아져 코드 수정이 편리하다..

단점 : UserDAO는 DB커넥션을 가져오는 클래스와 메소드에 종속적이기 때문에 확장이 힘들다.

(UserDao코드 변경 없이는 SimpleConnectionMaker 코드를 변경할 수 없다. )

해결방안 : 인터페이스를 통해 두 개의 클래스가 서로 연결되어 있지 않도록 중간에 추상적인 연결고리를 만들어 준다.

1. ConnectionMaker 인터페이스는 DB커넥션을 가져오는 makerConnection메소드를 정의 한다.
2. ConnectionMaker 인터페이스를 구현한 클래스는 makerConnection 메소드를 작성한다.
3. UserDAO는 ConnectionMaker 인터페이스와 makerConnection메소드를 사용한다.

인터페이스로 구현한 DAO

개요 : UserDAO가 어떤 클래스의 오브젝트를 사용할지 결정하는 코드를 제거해야 한다.

장점 : 구현 클래스의 소스코드가 바뀌어도 인터페이스에서 정의된 메소드 이름은UserDao에서 변경될 일이 없어졌다.

단점 : 어떤 ConnectionMaker 구현 클래스를 사용 할지 결정하는 코드가 남아있어 UserDAO 변경 없이는 확장이 자유롭지 못하다.

해결방안 : 관계설정의 책임을 분리한다.

1. UserDao 생성자는 파라미터로 ConnectionMaker 오브젝트를 받는다.

2. UserDao와 ConnectionMaker의 구현 클래스 사이의 관계를 UserDao의 클라이언트 오브젝트에서 분리한다.

- 클라이언트는 ConnectionMaker의 구현 클래스를 선택하고, 선택한 클래스의 오브젝트는 UserDao와 연결한다.

ConnectionMaker connectionmaker = new DConnectionMaker(); // 구현 클래스 선택 + 오브젝트 생성

userDao dao = new UserDao(connectionMaker); // userDao와 연결

  다형성 ) 해당 클래스가 구현한 인터페이스를 사용 했다면, 그 클래스의 오브젝트를 인터페이스 타입으로 받아서 사용 할 수 있다

관계설정의 책임을 분리한 DAO

개요 : 모든 관심사가 분리되어, UserDao에 대한 변경 없이도 DBConnection을 자유롭게 확장 할 수 있다.

[원칙과 패턴 - 개방 폐쇄 원칙 ( open-closed principle) ]

* > 클래스나 모듈은 확장에는 열려 있어야 하고 변경에는 받혀 있어야 한다.

(Ex: 인터페이스로 구현한 UserDao 는 DB 연결 기능을 확장하는데 열려 있고 이런 변화에 영향을 받지 않으므로 변경에는 닫혀있다.

객체지향 설계 원칙(SOLID) – 로머트 마틴

* SRP : 단일 책임 원칙
  + 작성된 클래스는 하나의 기능만 가지며, 클래스가 제공하는 모든 서비스는 해당? 그 하나의 책임을 수행하는 데 집중되어 있어야 한다 s
  + (extract class- 혼재되어 있는 각각의 책임을 개별 클래스로 분할하여 클래스 당 하나의 책임만 맡도록 한다.)
* OCP : 개방 폐쇄 원칙
  + SW 구성 요소는 확장에는 열려있고, 변경에는 닫혀 있어야 한다.
  + (높은 응집도와 낮은 결합도)
* LSP : 리스코프 지환 원칙
  + 상위형 객체는 프로그램의 속성을 변경하지 않고 하위형 객체로 교체할 수 있다.
* ISP : 인터페이스 분리 원칙
  + 자신이 사용하지 않는 인터페이스는 구현하지 않는다.

(클라이언트가 해당 클래스의 특정 부분만 이용한다면, 이들은 따로 인터페이스로 분리하여 관리한다.)

* DIPs : 의존관계 역전 원칙
  + 상위 모듈은 하위 모듈에 의존해서는 안되며, 둘 다 추상화에 의존해야 한다.

[높은 응집도와 낮은 결합도]

응집도란?

* + 높은 응집도 : 변화가 일어날 때 해당 모듈에서 변하는 부분이 크다.

ex) ConnectionMaker인터페이스를 통해 DB연결 기능을 확장하더라도 다른 클래스에 영향을 주지 않는다.

* + 낮은 결합도 : 책임과 관심사가 다른 모듈과 느슨하게 연결된 형태이다.
    - 하나의 변경이 발생할 때 다른 모듈에게 변경을 요구하는 정도가 낮다.

Ex)

ConnectionMaker 인터페이스를 도입한 경우와ConnectionMaker의 클래스를 결정하는 책임을 DAO의 클라이언트로 분리한 경우

= > ConnectionMaker 를 구현한 클래스가 바뀌더라도 DAO의 코드는 변경될 필요가 없다.

[전략 패턴]

* 자신의 기능 맥락(context)에서 변경이 필요한 알고리즘을 인터페이스를 통해 통째로 외부로 분리시키고, 이를 구현한 구체적인 알고리즘 클래스를 필요에 따라 바꿔서 사용할 수 있게 하는 디자인 패턴이다. -> 원태씨 말로 이해해서 바꾸기?
* 클라리언트(UserDaoTest)는 컨텍스트(UserDao)가 사용할 전략(DConnectionMaker)을 컨텍스트의 생성자 등을 통해 제공한다.

\*DAO(data access object) : DB를 사용하여 데이터를 조회하거나 조작하는 기능을 전담하도록 만든 오브젝트.

\*관심사 : 컴퓨터 프로그램 코드에 영향을 미치는 정보의 집합 -> 이해한대로 바꾸기?

\*리펙토링 : 기존의 코드를 외부의 동작방식에는 변화 없이 내부 구조를 변경해서 재구성하는 작업 또는 기술을 말한다.

* 효과 : 나쁜 냄새(중복 코드) 코드 내부의 설계가 개선되어 유지보수 용이, 코드의 품질 향상

\*디자인 패턴 : SW 설계 시 특정 사항에서 자주 만나는 문제를 해결하기 위한 솔루션

Sw 설계 시에 빈번하게 발생하는 문제를 해결하기 위해 제시된 솔루션?

- 객체지향적인 설계로 해결하기 위해 적용 할 수 있는 확장성 추구 방법은 상속과 오브젝트 합성이 있다.