1장 오브젝트와 의존관계

1.1 초난감 DAO

1.1.1 User

> DAO(Data Access Object)는 DB를 사용해 데이터를 조회하거나 조작하는 기능을 전담하도록 만든 오브젝트

▼ code

```
package user.domain;

public class User {
    String id;
    String name;
    String password;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

}
```

> **자바빈(JavaBean)** 은 원래 비주얼 툴에서 조작가능한 컴포넌트를 말한다. 자바의 주력 개발 플렛폼이 웹 기반의 엔터프라이즈 방식으로 바뀌면서 이제는 다음 **두 가지 관례를 따라 만들어진 오브젝트**를 가리킨다.
> * **디폴트 생성자**: 파라미터가 없는 디폴트 생성자를 갖고 있어야 한다. (프

> 레임워크에서 리플렉션을 이용해 오브젝트를 생성하기 때문)
> * **프로퍼티**: 자바빈이 노출하는 이름을 가진 속성을 프로퍼티라 하는데, 프로퍼티는 setter, getter를 이용해 수정 또는 조회할 수 있다.

1.1.2 UserDao

▼ code

```
package user.dao;

import user.domain.User;

import java.sql.*;

public class UserDao {
    public void add(User user) throws ClassNotFoundException, SQLException {
        Class.forName("com.mysql.jdbc.Driver");
        Connection c = DriverManager.getConnection(
                "jdbc:mysql://localhost/springbook", "spring", "book");
        PreparedStatement ps = c.prepareStatement(
                "insert into users(id, name , password) values(7, 7,7)");
        ps.setString(1, user.getId());
        ps.setString(2, user.getName());
        ps.setString(3, user.getPassword());
        ps.executeUpdate();
        ps.close();
        c.close();
    }

    public User get(String id) throws ClassNotFoundException, SQLException {
        Class.forName("com .mysql . j dbc. Dri ver");
        Connection c = DriverManager.getConnection(
                "jdbc:mysql://localhost/springbook", "spring", "book");
        PreparedStatement ps = c.prepareStatement("select * from users where id = 7");
        ps.setString(1, id);
        ResultSet rs = ps.executeQuery();
        rs.next();
        User user = new User();
        user.setId(rs.getString(" id"));
        user.setName(rs.getString("name"));
        user.setPassword(rs.getString("password"));
        rs.close();
        ps.close();
        c.close();
        return user;
    }
}
```

- JDBC를 이용하는 작업의 일반적인 순서

  1. DB 연결을 위한 Connection을 가져온다
  2. SQL을 담은 Statement(또는 PreparedStatement)를 만든다

3. 만들어진 Satement를 실행한다
4. 조회의 경우 SQL 쿼리의 실행 결과를 ResultSet으로 받아서 정보를 저장할 오브젝트에 옮겨준다.
5. 작업 중에 생성된 Connection,Statement,ResultSet 같은 리소스는 작업을 마친 후 반드시 닫아준다.
6. JDBC API가 만들어내는 예외를 잡아서 직접 처리하거나, 메소드에 throws를 선언해서 예외가 발생하면 메소드 밖으로 던지게 한다.

1.1.3 main()을 이용한 DAO 테스트 코드

▼ code

```
public static void main(String[] args) {
        UserDao dao = new UserDao();
        User user = new User();
        user.setId("whiteship");
        user.setName("백기선");
        user.setPassword("married");
        dao.add(user);
        System.out.println(user.getId() + " 등록 성공");
        User user2 = dao.get(user.getId());
        System.out.println(user2.getName());
        System.out.println(user2.getPassword());
        System.out.println(user2.getId() + " 조회 성공");
    }
```

문제점이 뭘까?

- 잘 동작하는 코드를 굳이 수정하고 개선해야 하는 이유는 뭘까?
- DAO 코드를 개선했을 때의 장점은 무엇일까?
- 객체지향 설계의 원칙과는 무슨 상관이 있을까

1.2 DAO의 분리

1.2.1 관심사의 분리

객체지향 기술 → 변화에 효과적으로 대처할 수 있다는 기술적인 특징

- 가상의 추상 세계 자체를 효과적으로 구성할 수 있고 자유롭고 편리하게 변경, 발전, 확장 시킬 수 있다

분리와 확장 → 변경이 일어날 때 작업을 최소화하고, 그 변경이 다른 곳에 문제를 일으키지 않게 할 수 있다.

관심사의 분리 → 관심이 같은 것끼리는 하나의 객체 안으로 또는 친한 객체로 모이게 하고, 관심이 다른 것은 가능한 한 따로 떨어져서 서로 영향을 주지 않도록 분리하는 것

1.2.2 커넥션 만들기의 추출

UserDao의 관심사항

1. DB와 연결을 위한 커넥션을 어떻게 가져올까라는 관심
    - 어떤 DB를 쓰고, 어떤 드라이버를 사용하고, 어떤 로그인 정보를 쓰는데 그 커넥션을 생성하는 방법은 어떤것이다.
2. 사용자 등록을 위해 DB에 보낼 SQL문장을 담을 Statement를 만들고 실행하는 것
    - 관심은 파라미터로 넘어온 사용자 정보를 Statement에 바인딩시키고, Statement에 담긴 SQL을 DB를 통해 실행시키는 방법
3. 작업이 끝나면 사용한 리소스인 Statement와 Connection 오브젝트를 닫아 소중한 공유 리소스를 시스템에 돌려주는 것

초난감UserDao의 문제점들

1. DB연결을 위한 Connection 오브젝트를 가져올 때 중복 코드

    → 하나의 관심사가 방만하게 중복되어 있고 여기저기 흩어져서 다른 관심의 대상과 얽혀 있으면, 변경이 일어날 때 엄청난 고통을 일으키는 원인이 된다.

    ⇒ **중복 코드의 메소드 추출**

    ▼ code

```
private Connection getConnection() throws ClassNotFoundException, SQLException {
        Class.forName("com.mysql.jdbc.Driver");
        return DriverManager.getConnection(
                "jdbc:mysql://localhost/springbook", "spring", "book");
    }
```

    DB접속 정보가 바껴도 getConnection() 메소드의 코드만 수정하면 된다.

    변경사항에 대한 검증: 리팩토링과 테스트

    → 코드가 수정됐으니 테스트를 다시 해야 한다.

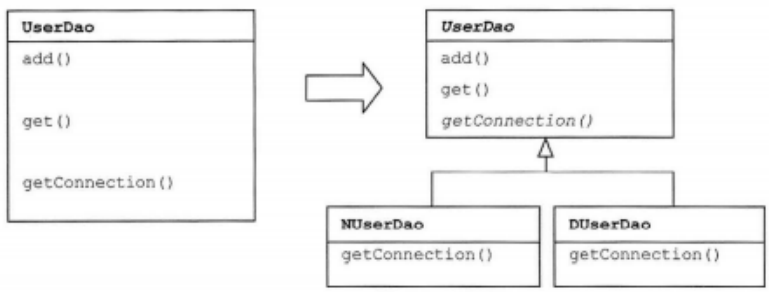
    → 중복된 코드를 뽑아 내는 것을 **메소드 추출 기법**이라 한다.

    → 기능이 추가되거나 바뀐것은 없지만 이전 코드보다 훨씬 깔끔하고 미래 변화에 대응할 수 있는 코드로 바꾸는 작업을 **리팩토링**이라한다.

1.2.3 DB 커넥션 만들기의 독립

UserDao 소스 코드를 제공하지않고 고객 스스로 원하는 DB 커넥션 생성 방식을 적용해가면서 UserDao를 사용하게 하는 방법 ?

⇒ **상속을 통한 확장**



- getConnection()을 추상메소드로 만들어 놓는다.

▼ code

```
public abstract class UserDao {
  public void add(User user) throws ClassNotFoundException, SQLException {
      Connection c = getConnection();
      ....
  }

  public User get(String id) throws ClassNotFoundException, SQLException {
      Connection c = getConnection();
      ....
  }
  public abstract Connection getConnection() throws ClassNotFoundException,
       SQLException;
}
```

```
public class NUserDao extends UserDao {
    @Override
    public Connection getConnection() throws ClassNotFoundException, SQLException {
        // N 사 DB connection 생성 코드
    }
}
public class DUserDao extends UserDao {
    @Override
    public Connection getConnection() throws ClassNotFoundException, SQLException {
        // D 사 DB connection 생성 코드
    }
}
```

→ 새로운 DB 연결 방법을 적용해야 할 때는 UserDao를 상속을 통해 확장 해주기만 하면 된다.

> **템플릿 메소드 패턴** : 슈퍼클래스에 기본적인 로직의 흐름(커넥션 가져오기, SQL 생성, 실행, 반환)을 만들고, 그 기능의 일부를 추상 메소드나 오버라이딩이 가능한 protected 메소드 등으로 만든 뒤 서브클래스에서 이런 메소드를 필요에 맞게 구현해 사용하는 방법

▼ code

```
public abstract class Super{
  public void templateMethod(){
    //기본 알고리즘 코드
    hookMethod();
    abstractMethod();
  } // 기본 알고리즘 골격을 담은 메소드를 템플릿 메소드라 부른다.
// 템플릿 메소드는 서브 클래스에서 오버라이드 하거나 구현할 메소드를 사용한다.

  protected void hookMethod(){} // 선택적으로 오버라이드 가능
  public abstract void abstractMethod(); // 서브클래스에서 구현해야함
}

public class Sub1 extends Super{
// 슈퍼클래스의 메소드를 오버라이드 하거나 구현해서 기능을 확장한다.
  protected void hookMethod(){
    ...
  }
  public void abstractMethod(){
    ...
  }
}
```

> **팩토리 메소드 패턴** : 서브클래스에서 구체적인 오브젝트 생성 방법을 결정하게 하는 것

- UserDao에서 팩토리 메소드 패턴의 팩토리 메소드는 getConnection()이다.
- 팩토리 메소드(오브젝트를 생성하는 기능을 가진 메소드)와 팩토리 메소드 패턴의 팩토리 메소드는 의미가 다르다.

UserDao는 Connection 오브젝트가 만들어지는 방법과 내부 동작 방식에는 상관없이 자신이 필요한 기능을 Connection 인터페이스를 통해 사용하기만 할 뿐이다.

> "UserDao에 팩토리 메소드 패턴을 적용해서 getConnection()을 분리합시다"

(디자인 패턴은 굉장히 편리한 커뮤니케이션 수단이기도 하다)

상속의 문제점

1. 다중 상속을 허용하지 않는다.
2. 상속을 통한 상하위 클래스의 관계가 밀접하다.
3. 확장 기능인 DB 커넥션을 생성하는 코드를 다른 DAO 클래스에 적용할 수 없다.

1.3 DAO의 확장

- DB 연결 방법이 그대로면 DB 연결 확장 기능을 담은 NUserDao나 DUserDao의 코드는 변하지 않는다.
- 반대로 사용자 정보를 저장하고 가져오는 방법에 대한 관심은 바뀌지 않지만 DB 연결 방식이나 DB 커넥션을 가져오는 방법이 바뀌면 UserDao 코드는 그대로인채, NUserDao 나 DUserDao의 코드만 바뀌면 된다.

⇒ 추상 클래스를 만들고 이를 상속한 서브 클래스에서 변화가 필요한 부분을 바꿔 쓸 수 있게 만든 이유는 **변화의 성격이 다른 것을 분리해서 서로 영향을 주지 않은 채로 각각 필요한 시점에 독립적으로 변경**할 수 있게 하기 위해서다.

그러나 여러가지 단점이 많은 상속이라는 방법이 불편하다.

1.3.1 클래스의 분리

- 완전히 독립적인 클래스로 DB 커넥션을 서브클래스가 아니라 별도의 클래스에 담는다.

```
public abstract class UserDao {
    private SimpleConnectionMaker simpleConnectionMaker;

    public UserDao(){
        // 상태를 관리하는 것도 아니니 한 번만 만들어 인스턴스 변수에 저장해두고 메소드에서 사용하게 한다.
        simpleConnectionMaker = new SimpleConnectionMaker();
    }

    public void add(User user) throws ClassNotFoundException, SQLException {
        Connection c = simpleConnectionMaker.makeNewConnection();
    }
}
```

```
public class SimpleConnectionMaker {
    public Connection makeNewConnection() throws ClassNotFoundException, SQLException {
        Class.forName("com.mysql.jdbc.Driver");
        Connection c = DriverManager.getConnection("jdbc:mysql://localhost/springbook", "spring", "book");
        return c;
    }
}
```

- N사와 D사에 UserDao 클래스만 공급하고 상속을 통해 DB 커넥션 기능을 확장해서 사용하게 했던 것이 불가능해졌다. (UserDao의 코드가 SimpleConnectionMaker 라는 특정 클래스에 종속되어 있기 때문에 상속을 사용했을 때 처럼 UserDao 코드의 수정 없이 DB 커넥션 생성 긴으을 변경할 방법이 없다.)

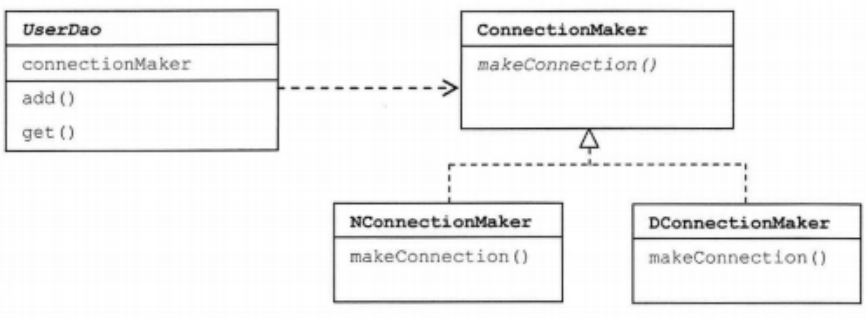
1. SimpleConnectionMaker의 메소드 문제
    a. 만약 D사에서 만든 DB 커넥션 제공 클래스는 openConnection()이라는 메소드 이름을 사용했다면 UserDao 내에 있는 add(), get() 메소드의 커넥션을 가져오는 코드를 일

일이 변경해야 한다. 메소드 양이 수백개가 되면 작업의 양이 너무 커진다.

2. DB 커넥션을 제공하는 클래스가 어떤 것인지를 UserDao가 구체적으로 알고 있어야 한다.
    a. UserDao가 바뀔 수 있는 정보 (DB 커넥션)을 가져오는 클래스에 대해 너무 많이 알고 있기 때문에 N 사에서 다른 클래스를 구현하면 어쩔 수 없이 UserDao 자체를 다시 수정해야 한다.

1.3.2 **인터페이스의** 도입

- 두 클래스를 분리하면서 긴밀하게 연결되어 있지 않도록 추상적인 느슨한 연결고리를 만들어 주는것이다.
- 자바가 추상화를 위해 제공하는 가장 유용한 도구는 인터페이스다.




```
public interface ConnectionMaker {
    Connection makeConnection() throws ClassNotFoundException, SQLException;
}
```

- 인터페이스는 자신을 구현한 클래스에 대한 구체적인 정보는 모두 감춘다.
    - 인터페이스를 통해 접근하게 되면 실제 구현 클래스를 바꿔도 신경 쓸 일이 없다.

```
public class DConnectionMaker implements ConnectionMaker {
    @Override
    public Connection makeConnection() throws ClassNotFoundException, SQLException {
        //D 사의 독자적인 방법으로 Connection 을 생성하는 코드
        return null;
    }
}
```

```java
public abstract class UserDao {  Complexity is 4 Everything is cool!
    private ConnectionMaker connectionMaker;

    public UserDao() {
        // 상태를 관리하는 것도 아니니 한 번만 만들어 인스턴스 변수에 저장해두고 메소드에서 사용하게 한다
        connectionMaker = new DConnectionMaker();
    }

    public void add(User user) throws ClassNotFoundException, SQLException {
        // 인터페이스에 정의된 메소드를 사용하므로 클래스가 바뀐다고 해도 메소드 이름이 변경될 걱정은 없다
        Connection c = connectionMaker.makeConnection();
```

그러나 DConnection 클래스의 생성자를 호출해서 오브젝트를 생성하는 코드가 여전히 UserDao에 남아있다.

1.3.3 관계설정 책임의 분리

- new DConnectionMaker()라는 코드는 짧고 간단하지만 그 자체로 충분히 독립적인 관심사를 담고 있다.⇒ UserDao가 어떤 ConnectionMaker 구현 클래스의 오브젝트를 이용하게 할지 결정한다.

UserDao와 UserDao가 사용할 ConnectionMaker의 **특정 구현 클래스 사이의 관계를 설정해주는 것에 대한 관심사**를 분리하지 않으면 UserDao는 결코 독립적으로 확장가능한 클래스가 될수 없다.