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.

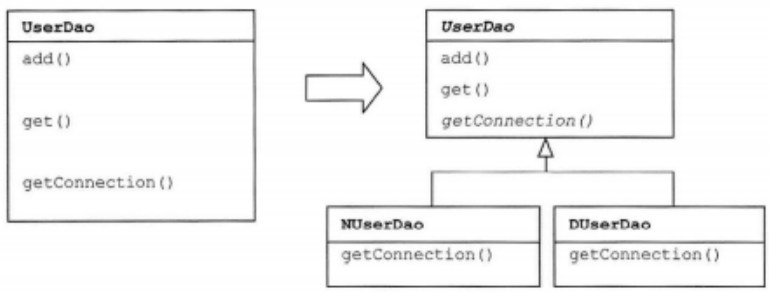
    → 중복된 코드를 뽑아 내는 것을 **메소드 추출 기법**이라 한다.

    → 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.)

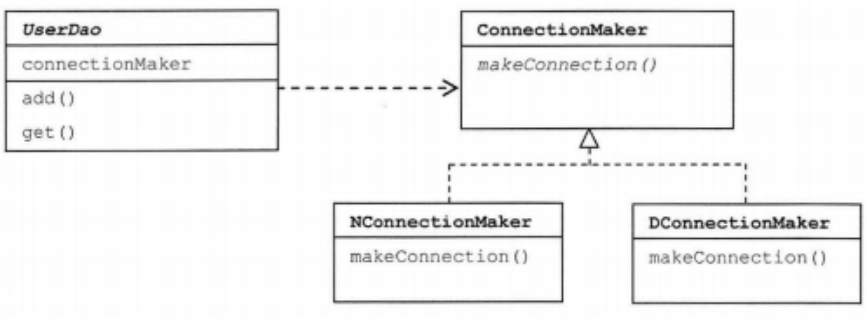
1. SimpleConnectionMaker의 메소드 문제
    a. 만약 D사에서 만든 DB 커넥션 제공 클래스는 openConnection()이라는 메소드 이름을 사용했다면 UserDao 내에 있는 add(), get() 메소드의 커넥션을 가져오는 코드를 일

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.

1.3.4 원칙과 패턴

> 개방 폐쇄 원칙 (OCP, Open-Closed Principle)

- 깔끔한 설계를 위해 적용 가능한 객체지향 설계 원칙 중 하나.
- 클래스나 모듈은 *확장에는 열려 있어야 하고 변경에는 닫혀 있어야* 한다.
- SOLID 원칙
  - SRP Single Responsibility Principle: 단일 책임 원칙
  - OCP Open Closed Principle: 개방 패쇄 원칙
  - LSP Liskov Substitution Principle: 리스코프 치환 원칙
  - ISP Interfase Segregation Principle: 인터페이스 분리 원칙
  - DIP Dependency Inversion Principle: 의존관계 역전 원칙
- 높은 응집도와 낮은 결합도
  - 응집도가 높다: 하나의 모듈, 클래스가 하나의 책임 또는 관심사에만 집중되어 있다.

- 결합도가 낮다: 하나의 변경이 발생할 때 여타 모듈과 객체로 변경에 대한 요구가 전파되지 않는 상태.

- 전략 패턴 (Strategy Pattern)
  - 자신의 기능 맥락(context)에서 필요에 따라 변경이 필요한 알고리즘을 인터페이스를 통해 통째로 외부로 분리시키고, 이를 구현한 구체적인 알고리즘 클래스를 필요에 따라 바꿔서 사용할 수 있게 하는 디자인 패턴
  - UserDAO는 전략 패턴의 컨텍스트에 해당한다. 자신의 기능을 수행하는데 필요한 기능 중에서 변경 가능한, DB 연결 방식이라는 알고리즘을 ConnectionMaker 라느 인터페이스로 정의하고 이를 구현한 클래스, 즉 전략을 바꿔가면서 사용할 수 있게 분리했다.

1.4 제어의 역전(IoC)

1.4.1 오브젝트 팩토리

- 이 클래스의 역할은 객체의 생성 방법을 결정하고 그렇게 만들어진 오브젝트를 리턴한다. 이런 오브젝트를 흔히 팩토리라고 부른다.
- 디자인패턴에서 말하는 추상 팩토리, 팩토리 메소드 패턴과는 다르다.
- 단지 오브젝트를 생성하는 쪽과 생성된 오브젝트를 사용하는 쪽의 역할과 책임을 깔끔하게 분리하려는 목적으로 사용하는 것이다.
- 설계도로서의 팩토리

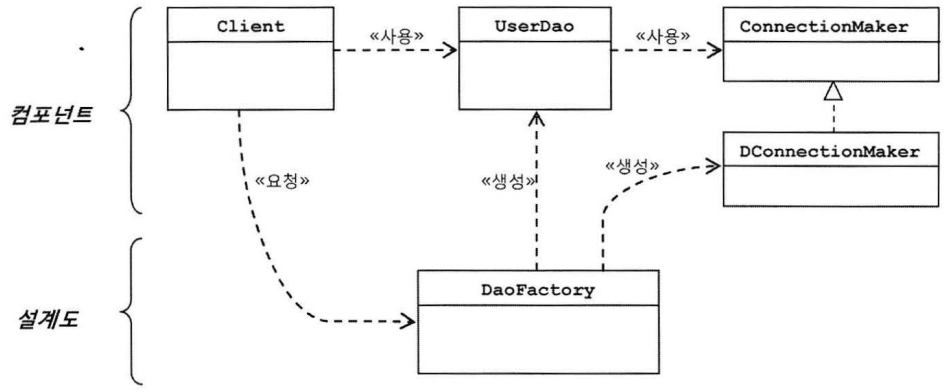


그림 1-8 오브젝트 팩토리를 활용한 구조

  - 애플리케이션을 구성하는 컴포넌트의 구조와 관계를 정의한 설계도 같은 역할을 한다.

1.4.2 오브젝트 팩토리의 활용

```
public AcountDao accountDao(){
  return new AccountDao(new DConnectionMaker());
}
```

```
public MessageDao messageDao(){
  return new MessageDao(new DConnectionMaker());
}
```

```
public ConnectionMaker connectionMaker(){
        return new DConnectionMaker(); // 분리해서 중복을 제거한 ConnectionMaker 타입 오브젝트 생성 코드
    }
```

1.4.3 제어권의 이전을 통한 제어관계 역전

- 모든 오브젝트가 능동적으로 자신이 사용할 클래스를 결정하고 언제 어떻게 그 오브젝트를 만들지 스스로 관장한다. 모든 종류의 작업을 사용하는 쪽에서 제어하는 구조다.
- 제어의 역전이란 이런 제어 흐름의 개념을 뒤집는 것이다. 제어의 역전에서 오브젝트가 자신이 사용할 오브젝트를 스스로 선택, 생성 하지 않는다. 어디서 만들어지고 어디서 사용되는지 모른다.
- 라이브러리와 프레임워크의 차이
  - 라이브러리 : 사용하는 애플리케이션 코드는 애플리케이션 흐름을 직접 제어한다. 단지 동작중에 필요한 기능이 있을 때 능동적으로 라이브러리를 사용할 뿐이다.
  - 프레임워크 : 애플리케이션 코드가 프레임워크게 의해 사용된다. 프레임워크 위에 개발한 클래스를 등록해 두고 프레임워크가 흐름을 주도하는 중에 개발자가 만든 애플리케이션 코드를 사용하도록 만드는 방식.
- UserDao 와 ConnectionMaker의 구현체를 생성하는 책임을 DaoFactory가 맡고 있는 데 이것이 제어의 역전이다.

1.5 스프링의 IoC

1.5.1 오브젝트 팩토리를 이용한 스프링 IoC

- 애플리케이션 컨텍스트와 설정정보
  - 스프링에서는 스프링이 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트를 빈(Bean)이라고 부른다. 자바빈, 엔터프라이즈 자바빈에서 말하는 빈과 비슷한 오브젝트 단위의 애플리케이션 컴포넌트를 말한다.
  - 빈은 스프링 컨테이너가 생성과 관계 설정, 사용 등을 제어해주는 제어의 역전이 적용된 오브젝트를 가리키는 말이다.
  - 스프링에서는 빈의 생성과 관계 설정 같은 제어를 담당하는 IoC 오브젝트를 `빈 팩토리` 라고 한다. → 더 확장되어 `애플리케이션 컨텍스트` 라고 부른다. 애플리케이션 컨텍스트는 IoC 방식을 따라 만들어진 빈 팩토리다! (빈 팩토리 = 애플리케이션 컨텍스트)
  - 컴포넌트와 설계도 역할을 하는 팩토리로 구분한 것처럼 설계도라는 게 애플리케이션 컨텍스트의 설정 정보를 말하고, 애플리케이션도 애플리케이션 컨텍스와 그 설정 정보를 따라서

만들어지고 구성된다!

- DaoFactory를 사용하는 애플리케이션 컨텍스트
  - `@Configuration` 어노테이션을 추가해 스프링이 빈 팩토리를 위한 오브젝트 설정을 담당하는 클래스라고 인식할 수 있도록 만든다.
  - 오브젝트를 만들어 주는 메서드에 `@Bean` 이라는 어노테이션을 추가 한다.

```
public class UserDaoTest {
    public static void main(String[] args) throws ClassNotFoundException, SQLException {
//        UserDao dao = new DaoFactory().userDao();
        ApplicationContext applicationContext
                = new AnnotationConfigApplicationContext(DaoFactory.class);
        UserDao userDao = applicationContext.getBean( name: "userDao", UserDao.class);
    }
}   SONG22, 9 minutes ago • 토비의 스프링 1.4까지의 내용
```

빈의 이름으로 가져오는 이유는 UserDao를 생성하는 방식이나 구성을 다르게한 메소드를 추가할 수 있기 때문이다.

1.5.2 애플리케이션 컨텍스트의 동작 방식

- 오브젝트 팩토리 ⇒ 애플리케이션 컨텍스트 = IoC 컨텍스트 = 스프링 컨테이너 = 빈 팩토리
- 앞의 DaoFactory와 달리 직접 오브젝트를 생성하고 관계를 맺어주는 코드가 없고, 그런 생성 정보와 연관관계 정보를 별도의 설정 정보를 통해 얻는다. 또는 외부의 오브젝트 팩토리에 그 작업을 위임하고 그 결과를 가져다 사용하기도 한다.

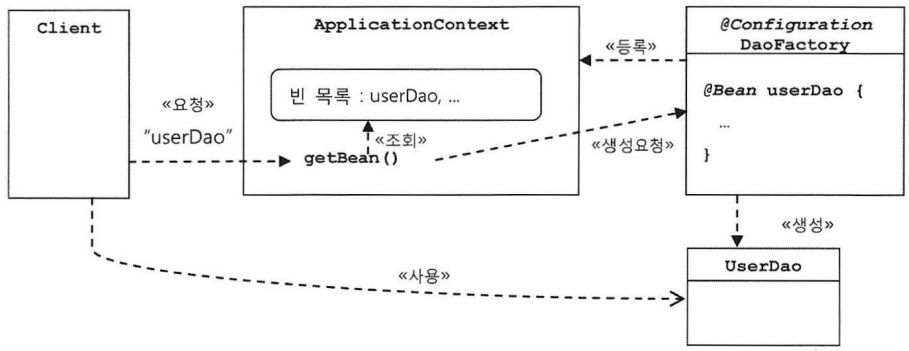


그림 1-9 애플리케이션 컨텍스트가 동작하는 방식

- 클라이언트는 구체적인 팩토리 클래스를 알 필요가 없다.
- 애플리케이션 컨텍스트는 종합 IoC 서비스를 제공해준다.
- 애플리케이션 컨텍스트는 빈을 검색하는 다양한 방법을 제공한다.

1.5.3 스프링 IoC의 용어 정리

- 빈: 또는 빈오브젝트: 스프링이 IoC방식으로 관리하는 오브젝트
- 빈 팩토리: 스프링의 IoC를 담당하는 핵심 컨테이너, 보통 이를 확장한 애플리케이션 컨텍스트를 이용한다.
- 애플리케이션 컨텍스트: 빈 팩토리를 확장한 IoC 컨테이너, 스프링이 제공하는 각종 부가 서비스를 추가로 제공한다. BeanFactory를 상속한다.
- 설정정보/설정 메타 정보: 애플리케이션 컨텍스트가 IoC를 적용하기 위해 사용하는 메타 정보
- 컨테이너 또는 IoC 컨테이너: IoC 방식으로 빈을 관리한다는 의미.
- 스프링 프레임워크: IoC 컨테이너, 애플리케이션 컨텍스트를 포함해 스프링이 제공하는 모든 기능을 통틀어 말할 때 주로 사용한다.

1.6 싱글톤 레지스트리와 오브젝트 스코프

- 오브젝트의 동일성과 동등성
  - 동일성(indentity):
    - == 사용, 두 오브젝트가 완전히 같을 때→ 사실상 하나의 오브젝트만 존재
  - 동등성(equality):
    - equals()사용, 동일한 정보를 담고 있을 때→ 두개의 레퍼런스 변수가 존재
  - Object의 euqals()는 두 오브젝트의 동일성을 비교해서 결과를 돌려준다 → 동일 오브젝트여야만 동등한 오브젝트라 여긴다.
- 위에서 만들었던 오브젝트 팩토리(DaoFactory)와 스프링의 애플리케이션 컨텍스트의 동작방식에는 무엇인가 차이점이 있다.

1.6.1 싱글톤 레지스트리로서의 애플리케이션 컨텍스트

- 서버 애플리케이션과 싱글톤
  - ***왜 스프링은 싱글톤으로 빈을 만드는 것일까***?

    → 스프링이 주로 적용되는 대상이 자바 엔터프라이즈 기술을 사용하는 서버환경이기 때문이다.

    매번 클라이언트에서 요청이 올 때 마다 각 로직을 담당하는 오브젝트를 새로 만든다 가정했을 때, 아무리 자바 GC 성능이 좋다고 하더라도 부하가 걸리면 서버가 감당하기 힘들다. 때문에 엔터프라이즈 분야에서는 서비스 오브젝트라는 개념을 일찍부터 사용해왔다.

    서블릿은 자바 엔터프라이즈 기술의 가장 기본이 되는 서비스 오브젝트다. 서블릿은 대부분 멀티 스레드 환경에서 싱글톤으로 동작한다. 서블릿 클래스당 하나의 오브젝트만 만들어두고, 사용자의 요청을 담당하는 여러 스레드에서 하나의 오브젝트를 공유해 동시에 사용한다.

이렇게 애플리케이션 안에 제한된 수, 대개 한 개의 오브젝트만 만들어서 사용하는 것이 싱글턴 패턴의 원리다.

- **싱글톤 패턴**
  - 디자인 패턴 중에 가장 자주 활용되는 패턴이기도 하지만 가장 많은 비판을 받는 패턴이기도 하다.
  - 싱글톤 패턴은 어떤 클래스를 애플리케이션 내에서 제한된 인스턴스 개수, 이름처럼 주로 하나만 존재하도록 강제하는 패턴이다. 이렇게 하나만 만들어지는 클래스의 오브젝트는 애플리케이션내에서 **전역적으로 접근**이 가능하다.
  - 단일 오브젝트만 존재해야 하고, 이를 애플리케이션의 **여러 곳에서 공유**하는 경우에 주로 사용한다.
- 싱글톤 패턴의 한계

▼ 싱글톤 구현 방법

1. 클래스 밖에서는 오브젝트를 생성하지 못하도록 생성자를 `private`으로 만든다.
2. 생성된 싱글톤 오브젝트를 저장할 수 있는 자신과 같은 타입의 `static` 필드를 정의한다.
3. `static` 팩토리 메소드인 `getInstance()`를 만들고 이 메소드가 최초로 호출되는 시점에 한번만 오브젝트가 만들어지게 한다. 셍성된 오브젝트는 스태틱 필드에 저장한다. 또는 스태틱 필드의 초기값으로 오브젝트를 미리 만들어둘 수도 있다.
4. 한번 오브젝트가 만들어지고 난 후에는 `getInstance()`메소드를 통해 이미 만들어져 스태틱 필드에 저장해둔 오브젝트를 넘겨준다.

```
public class UserDao {  Complexity is 6 It's time to do something...
    private static UserDao INSTANCE;

    private ConnectionMaker connectionMaker;

    public UserDao(ConnectionMaker connectionMaker) {
        // 상태를 관리하는 것이 아니니 한번만 만들어 인스턴스 변수에 저장해두
//          dConnectionMaker = new DConnectionMaker();
        this.connectionMaker = connectionMaker;
    }

    public static synchronized UserDao getInstance(){  Comple
        if(INSTANCE == null) {
            INSTANCE = new UserDao(????);
        }
        return INSTANCE;
    }
```

ConnectionMaker를 넣어주는게 불가능해졌다.

  - 싱글톤 패턴의 문제점
    - `private` 생성자를 갖고 있기 때문에 **상속할 수 없다**.

        - 싱글톤은 **테스트하기가 힘들다**.
        - 서버환경에서는 싱글톤이 **하나만 만들어지는 것을 보장하지 못한다**.
        - 싱글톤의 사용은 *전역 상태를* 만들수 있기 때문에 바람직하지 못하다.
- 싱글톤 레지스트리

  자바의 기본적인 싱글톤 패턴의 구현방식은 여러가지 단점이 있기때문에, 스프링은 직접 싱글톤 형태의 오브젝트를 만들고 관리하는 기능을 제공한다.

    - 스프링 컨테이너는 싱글톤을 생성하고 관리하고 공급하는 싱글톤 관리 컨테이너기도 하다.
    - 스태틱 메소드와 Private 생성자를 사용해야 하는 비정상적인 클래스가 아닌 **평범한 자바 클래스를 싱글톤으로 활용하게 해준다.**
    - 오브젝트 생성에 관한 모든 권한은 IoC 기능을 제공하는 애플리케이션 컨텍스트에게 있기 때문에 싱글톤 방식으로 만들어져 관리되게 할 수 있다.
    - public 생성자, 테스트를 위한 mock 오브젝트로 대체, 생성자 파라미터를 이용해 사용할 오브젝트를 넣어주게 할 수도 있다. 스프링이 지지하는 객체지향적인 설계 방식과 원칙, 디자인 패턴에 제약이 없다.
    - 스프링은 **IoC 컨테이너**일 뿐만 아니라, 고전적인 싱글톤 패턴을 대신해 **싱글톤을 만들고 관리해주는 싱글톤 레지스트리**다.

1.6.2 싱글톤과 오브젝트의 상태

싱글톤은 멀티스레드 환경이면 여러스레드가 동시에 접근해 사용할수 있으므로 상태 관리에 주의를 기울여야 한다. → 기본적으로 싱글톤이 *멀티스레드 환경에서 서비스 형태의 오브젝트로 사용*되는 경우 상태 정보를 내부에 갖고 있지 않은 **무상태 방식**으로 만들어야 한다.

- 스프링의 싱글톤 빈으로 사용되는 클래스를 만들 때는 기존의 UserDao 처럼 개별적으로 바뀌는 정보는 로컬 변수로 정의하거나, 파라미터로 주고 받으면서 사용하게 해야 한다.
- 인터페이스 변수는 사용해도 상관없다. 읽기 전용의 정보이기 때문이다.
    - 자신이 사용하는 다른 싱글톤 빈을 저장하려는 용도라면 인스턴스 변수를 사용해도 좋다.
    - 스프링이 한번 초기화해주고 나면 이후에 수정되지 않기 때문에 멀티스레드 환경에서 사용해도 아무 문제가 없다.

1.6.3 스프링 빈의 스코프

- 스프링이 관리하는 오브젝트(빈)이 생성되고 존재하고 적용되는 범위를 빈의 스코프 라고 한다.
- 스프링 빈의 기본 스코프는 싱글톤이다.
- 싱글톤 스코프는 컨테이너 내에 한 개의 오브젝트만 만들어져서, 강제로 제거하지 않는 한 스프링 컨테이너가 존재하는 동안 계속 유지된다.
- 경우에 따라 싱글톤 외의 스코프를 가질 수 있는데 대표적으로 prototype 스코프가 있다.

- request scope
- session scope 등이 있다.

1.7 의존관계 주입(DI)

1.7.1 제어의 역전(IoC)과 의존관계 주입

- 스프링이 제공하는 IoC 방식의 핵심을 짚어주는 의존관계 주입이라는, 의도가 명확히 드러나는 이름을 사용했다.
- DI 컨테이너로 불린다

1.7.2 런타임 의존관계 설정

- 의존관계
  - 두 개의 클래스 사이에 방향성이 있을 때

그림 1-10 클래스의 의존관계 다이어그램

A 가 B에 의존한다.

B가 변하면 → A 에 영향을 준다.

- UserDao의 의존관계
  - 인터페이스를 통해 의존관계를 제한해주면 그만큼 변경에서 자유로워진다.
  - 모델이나 코드에서 클래스와 인터페이스를 통해 드러나는 의존관계 말고 런타임시에 오브젝트 사이에 만들어지는 의존관계도 있다 → 런타임 의존관계 또는 오브젝트 의존관계
  - 의존관계 주입은 구체적인 의존 오브젝트와 그것을 사용할 주체, 보통 클라이언트라고 부르는 오브젝트를 런타임 시에 연결해주는 작업을 말한다.
  - 의존관계 주입은 다음 3가지 조건을 충족하는 작업이다.
    1. 클래스 모델이나 코드에는 런타임 시점의 의존관계가 드러나지 않는다. 그러기 위해 인터페이스에만 의존하고 있어야 한다.
    2. 런타임 시점의 의존관계는 **컨테이너나 팩토리 같은 제 3의 존재가** 결정한다.
    3. 의존관계는 사용할 오브젝트에 대한 레퍼런스를 외부에서 제공해줌으로써 만들어진다.
- UserDao의 의존관계 주입
  - DaoFactory 는 두 오브젝트 사이의 런타임 의존관계를 설정해주는 의존관계 주입 작업을 주도하는 존재이며, 동시에 IoC방식으로 오브젝트의 생성과 초기화, 제공 등의 작업을 수

행하는 컨테이너다. ⇒ DI 컨테이너다.

- DI 컨테이너에 의해 런타임 시에 의존 오브젝트를 사용할 수 있도록 그 레퍼런스를 전달받는 과정이 마치 메소드(생성자)를 통해 DI 컨테이너가 UserDao에게 주입해주는 것과 같다고 해서 `의존관계 주입` 이라고 부른다.
- **DI 는 자신이 사용할 오브젝트에 대한 선택과 생성 제어권을 외부로 넘기고 자신은 수동적으로 주입받은 오브젝트를 사용한다**는 점에서 IoC 개념과 잘 맞는다.

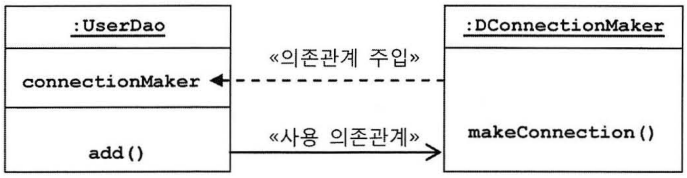


그림 1-13 런타임 시의 의존관계 주입과 사용 의존관계

1.7.3 의존관계 검색과 주입

- 의존관계 검색
  - 의존관계를 맺는 방법이 외부로부터의 주입이 아니라 스스로 검색을 이용하기 때문에 의존관계 검색이라고 불리는 것도 있다.
  - 자신이 필요로 하는 의존 오브젝트를 능동적으로 찾는다.
  - 자신이 어떤 클래스의 오브젝트를 이용할지 결정하지는 않는다.
  - 런타임 시 의존관계를 맺을 오브젝트를 결정하는 것과 오브젝트의 생성 작업은 외부 컨테이너에게 IoC로 맡기지만, 이를 가져올 때는 메소드나 생성자를 통한 주입 대신 스스로 컨테이너에게 요청하는 방법을 사용한다.
  - 스프링 IoC컨테이너(=애플리케이션 컨텍스트)는 `getBean()`이라는 메소드가 의존관계 검색으로 사용된다.
- 의존관계 검색 vs 의존관계 주입
  - 의존관계 주입이 훨씬 단순하고 깔끔하다.
  - 의존관계 검색은 코드 안에 오브젝트 팩토리 클래스나 스프링 API가 나타난다. → 애플리케이션 컴포넌트가 컨테이너와 같이 성격이 다른 오브젝트에 의존하게 되는 것이므로 바람직하지 않다.
  - 의존관계 검색 방식에서는 **검색하는 오브젝트는 자신이 스프링의 빈일 필요가 없다**.
  - 의존관계 주입에서는 UserDao와 ConnectionMaker 사이에 DI가 적용되려면 UserDao도 **반드시 컨테이너가 만드는 빈 오브젝트**여야 한다.
- 의존관계 검색 방식을 사용할 때도 있다.

    - 사용자의 요청을 받을 때마다 main메서드의 역할을 하는 서블릿에서 스프링 컨테이너에 담긴 오브젝트를 사용하려면 한 번은 의존관계 검색 방식을 사용해 오브젝트를 가져와야 한다. ⇒ 서블릿은 스프링이 미리 만들어 제공하기 때문에 직접 구현할 필요는 없다.
- DI 받는다
    - 이름 그대로 외부로부터의 주입이다
    - DI에서 말하는 주입은 다이나믹하게 구현 클래스를 결정해서 제공받을 수 있도록 인터페이스 타입의 파라미터를 통해 이뤄져야 한다.

1.7.4 의존관계 주입의 응용

- 기능 구현의 교환
    - 개발 중 로컬 DB → 운영 서버 DB
- 부가 기능 추가
    - 연결 횟수 추가

1.7.5 **메소드**를 이용한 의존관계 주입

- **수정자** 메소드를 이용한 주입
- **일반** 메소드를 이용한 주입

1.8 XML을 이용한 설정

1.9 정리

1. 관심사 분리, 리팩토링
2. 전략 패턴
3. 개방 폐쇄 원칙
4. 낮은 결합도 높은 응집도
5. 제어의 역전 IoC
6. 싱글톤 레지스트리
7. DI 컨테이너, 의존관계 주입/DI
8. 생성자 주입과 수정자 주입
9. XML 설정