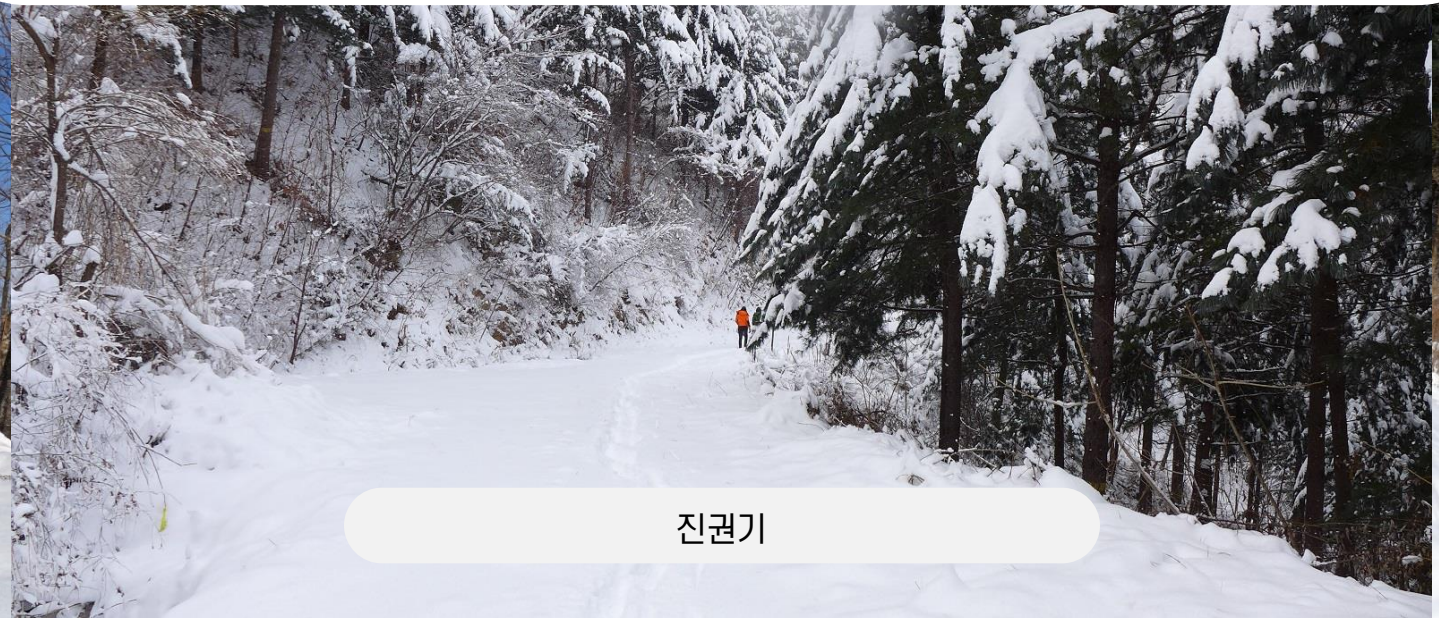



Spring Framework Basic


진권기

Spring의 이해

- ✓ Spring은 Java Enterprise Application 개발에 사용되는 애플리케이션 프레임워크 입니다.
- ✓ 애플리케이션 프레임워크는 애플리케이션 개발을 빠르고 효율적으로 할 수 있도록 해당 애플리케이션의 바탕이 되는 틀과 공통 프로그래밍 모델, 기술 API등을 제공합니다.
- ✓ Spring은 Spring Container 또는 Application Context라 불리는 Spring Runtime Engine을 제공합니다.
- ✓ Spring Container는 설정 정보를 참고로 해서 애플리케이션을 구성하는 객체를 생성하고 관리합니다.
- ✓ Spring Container는 독립적으로 동작할 수도 있지만 보통 Web Module에서 동작하는 서비스나 서블릿으로 등록해서 사용합니다.

Spring의 이해

- ✓ Spring은 서버 측 개발에만 유용한 것이 아니라 간소함, 테스트의 용이함, 낮은 결합도라는 견지에서 모든 애플리케이션에 Spring을 적용할 수 있습니다.
- ✓ Spring의 기본 원칙은 "Java의 개발 간소화" 입니다.

- ✓ Spring의 4가지 주요 전략
  - POJO를 이용한 가볍고(lightweight) 비 침투적인(non-invasive) 개발
  - DI와 인터페이스 지향(Interface orientation)을 통한 느슨한 결합도(loose coupling)
  - Aspect와 공통 규약을 통한 선언적 프로그래밍
  - Aspect와 Template을 통한 상투적인 코드의 축소

- ✓ POJO : Plain Old Java Object
- ✓ DI : Dependency Injection

- ✓ Spring은 Java를 기반으로 한 기술이다. Spring이 Java에서 가장 중요하게 가치를 두는 것은 객체지향 프로그래밍이 가능한 언어라는 것이다.

Spring의 이해

✓ Spring의 주요 모듈들은 다음과 같습니다.

✓ Spring Framework
- 스프링을 이용해서 애플리케이션을 개발할 때 기반이 되는 프레임워크 입니다. 스프링의 핵심 기능은 DI와 AOP 기능을 제공합니다. 웹 애플리케이션을 개발할 때 사용하는 스프링 MVC, 스프링 ORM 등의 기능도 스프링 프레임워크에 포함되어 있습니다.

✓ Spring Data
- 데이터 연동을 위한 단일 API를 제공하며, 이 API를 기반으로 JPA, MongoDB, Redis등 RDBMS와 NoSQL과의 연동을 적은 양의 코드로 처리할 수 있습니다.

✓ Spring Security
- 인증과 허가에 대한 기반 프레임워크 및 관련 모듈을 제공합니다. 웹 애플리케이션을 위한 보안을 간단한 설정과 약간의 코드 구현으로 처리할 수 있습니다.

✓ Spring Batch
- 배치 처리를 위한 기반 프레임워크를 제공해줍니다. 데이터 처리, 흐름제어, 실패 재처리 등 배치 처리 애플리케이션이 필요로 하는 기능을 기본으로 제공합니다.

✓ Spring Intigration
- 시스템 간의 연동을 위한 메시징 프레임워크를 제공합니다.

✓ Spring Social
- 트위터, 페이스북 등 소셜 네트워크 연동을 위한 기능을 제공합니다.

Spring의 이해

✓ Java Bean 객체는 다음과 같은 형태를 갖는 객체를 뜻합니다.

✓ Default Constructor

- Java Bean은 파라미터가 없는 디폴트 생성자를 갖고 있어야 합니다.
- 프레임워크에서는 자바의 reflection을 통해 객체를 생성하기 때문입니다.

✓ Property

- Java Bean이 노출하는 이름을 가진 속성을 Property라 합니다.
- Property는 set/get 메소드를 통해 값을 수정 또는 조회 할 수 있습니다.

✓ 리플렉션(reflection)

- 대부분의 객체지향 프로그래밍 언어에서는 객체가 속한 타입을 추적할 수 있도록 객체들에 대한 RTTI(Runtime Type Identification)를 관리합니다. Java 역시 RTTI를 사용하는 언어 중의 하나이며, JVM은 런타임 타입정보를 통해 정확한 메소드를 찾아내어 실행합니다.
- Spring Framework의 bean 설정에 정의한 클래스 정보나 MyBatis의 resultMap에 정의한 객체의 프로퍼티 정보가 실제로 동작하기 위해서는 리플랙션의 도움을 받아야 합니다.

```
public class User {
   private String id;
   private String name;
   private String pw;

   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 getPw() {
      return pw;
   }
   public void setPw(String pw) {
      this.pw = pw;
   }
}
```

Spring의 이해

✓ UserDAO 클래스(사용자 정보를 DB에 넣고 관리할 수 있는 DAO 클래스)

```
public class UserDAO {

   public void add(User user) throws ClassNotFoundException, SQLException{
      Class.forName("com.mysql.jdbc.Driver");
      Connection c = DriverManager.getConnection("jdbc.....");

      PreparedStatement ps = c.prepareStatement("insert into.....");
      ps.setString(1, user.getId());
      ps.setString(2, user.getName());
      ps.setString(3, user.getPw());

      ps.executeUpdate();
      ps.close();
      c.close();
   }

이어서…
```

Spring의 이해

```
public User get(String id) throws ClassNotFoundException, SQLException{
   Class.forName("com.mysql.jdbc.Driver");
   Connection c = DriverManager.getConnection("jdbc.....");

   PreparedStatement ps = c.prepareStatement("select * from.....");
   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.setPw(rs.getString("pw"));

   rs.close();
   ps.close();
   c.close();
   return user;
}
```

Spring의 이해

[질문]

✓ User 클래스와 UserDAO 클래스를 이용해 DB에 데이터를 삽입하고 꺼내 오는 것을 main() 메소드를 이용해 테스트를 진행하여 이상 없이 동작하는 것을 확인했다고 한다면 이 두 개의 클래스는 잘 만들어진 클래스일까?

✓ 소프트웨어 개발에서 가장 중요한 것은 내가 작성한 현재의 코드는 언제든지 변경된다는 것입니다. 그것은 사용자의 요구사항을 통해서 일수도 있고, 프로그램의 효율성 때문이기도 합니다.

✓ 개발자가 객체를 설계할 때 가장 먼저 고려해야 할 것은 바로 미래의 변화를 어떻게 대비할 것인가 입니다.

✓ 객체지향 설계와 프로그래밍이 이전의 절차적인 프로그래밍 패러다임에 비해 초기에 좀 더 많은… 번거로운 작업을 요구하는 이유는 객체지향 기술 자체가 지니는, 변화에 효과적으로 대처할 수 있다는 기술적인 특징 때문입니다.

Spring의 이해

✓ 일반적으로 프로그램에 대한 변화는 집중된 한 가지 관심에 대해 일어나지만 그에 따른 작업은 한곳에 집중되지 않는 경우가 많습니다.

(예시)
- DB 접속 암호를 변경하려고 DAO 클래스 수백 개를 모두 수정해야 한다면?

- 트랜잭션 기술을 다른 것으로 변경했다고 비즈니스 로직이 담긴 코드의 구조를 모두 변경해야 한다면?

- 다른 개발자가 개발한 코드에 변경이 일어날 때마다 내가 만든 클래스도 함께 수정을 해줘야 한다면?

✓ 이를 해결하기 위해 우선적으로 해야 할 것은 관심사의 분리(Separation of Concerns)입니다.
관심이 같은 것끼리는 하나의 객체 안으로 또는 친한 객체로 모이게 하고, 관심이 다른 것은 가능한 따로 떨어져서 서로 영향을 주지 않도록 분리하는 것입니다.

Spring의 이해

✓ UserDAO 클래스의 관심사항

- DB와 연결을 위한 Connection을 어떻게 가져올 것인가?
  어떤 Database를 쓰고, 어떤 Driver를 쓰고, Connection 생성하는 방법은 어떤 것을 쓸 것인가? 등…

- 사용자 등록을 위해 DB에 보낼 SQL 문장을 담을 Statement를 만들고 실행하는 것.
  파라미터로 넘어온 사용자 정보를 Statement에 바인딩시키고, Statement에 담겨진 SQL을 DB를 통해 실행 시키는 방법

- 작업이 끝나면 사용한 리소스들은 어떻게 반납할 것인가?

Spring의 이해

✓ 첫번째 변화

UserDAO 클래스의 개선(리팩토링) 첫 번째 중복 코드의 메소드 추출

```
public class UserDAO {
   public void add(User user) throws ClassNotFoundException, SQLException{
      Connection c = getConnection();
      …
   }
   public User get(String id) throws ClassNotFoundException, SQLException{
      Connection c = getConnection();
      …
   }

   private Connection getConnection() throws SQLException, ClassNotFoundException{
      Class.forName("com.mysql.jdbc.Driver");
      Connection c = DriverManager.getConnection("jdbc.....");
      return c;
   }
}
```

✓ 만일 UserDAO 클래스의 메소드가 1000개이고 DB연결과 관련된 부분의 변경이 일어날 경우 이제는 getConnection() 메소드를 수정하는 것으로 문제를 해결할 수 있습니다.

Spring의 이해

✓ 기능이 추가되거나 바뀐 것은 없지만 UserDAO는 이전보다 훨씬 깔끔해졌고 미래의 변화에 좀 더 손쉽게 대응할 수 있는 코드가 됐다. 이런 작업을 리팩토링(refactoring)이라고 합니다.

또한, getConnection()이라고 하는 공통의 기능을 담당하는 메소드로 중복된 코드를 뽑아내는 것을 리팩토링에서는 메소드 추출(extract method)기법 이라고 합니다.

리팩토링은 객체지향 개발자라면 반드시 익혀야 하는 기법입니다.

Spring의 이해

- ✓ 두 번째 변화: DB 커넥션 만들기의 독립
  만일 UserDAO 클래스에 새로운 기술이 적용되어 훌륭한 클래스로 발전하고 많은 회사에서 UserDAO 클래스의 구매를 원할 경우에 우리는 어떻게 이 클래스를 배포할 것인가?

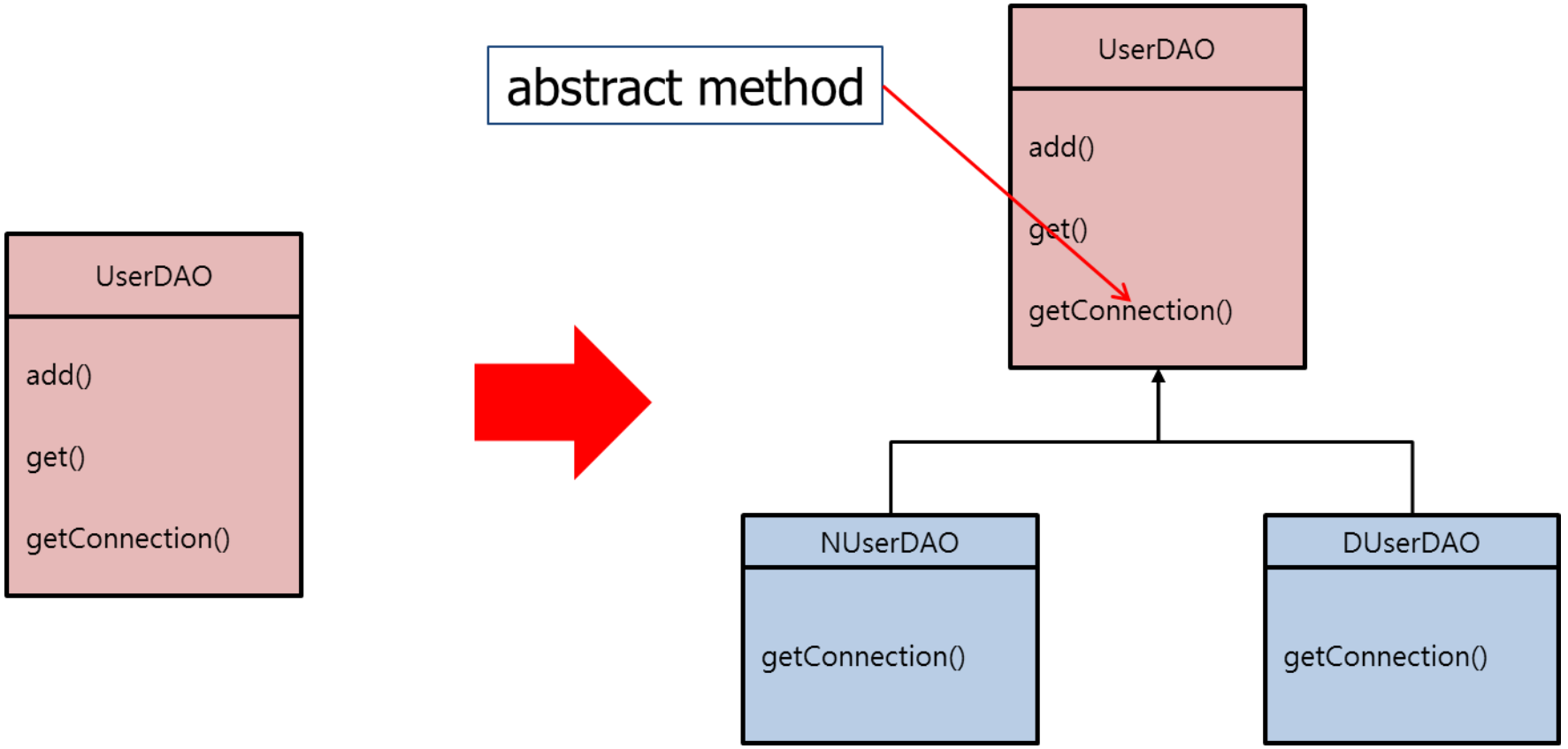
- ✓ 문제점 1 : UserDAO 클래스에는 우리 회사만의 신기술이 적용되어 소스 코드를 제공할 수 없습니다.

- ✓ 문제점 2 : 구매를 원하는 회사들은 각기 다른 DB 시스템을 사용하기 때문에 Connection을 얻어오는 방법을 각 회사들이 능동적으로 변경할 수 있도록 하여야 합니다.

- ✓ UserDAO 클래스의 소스를 제공하지 않고 고객 스스로 원하는 DB Connection 생성 방식을 사용해가면서 UserDAO를 사용하게 할 수 있을까?

Spring의 이해

✓ 상속을 통한 확장



Spring의 이해

✓ UserDAO 클래스

```
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();
      …
      return new User();
   }

   public abstract Connection getConnection() throws SQLException, ClassNotFoundException;
}
```

Spring의 이해

✓ N, D사 UserDAO 클래스

```
public class NUserDAO extends UserDAO{

   @Override
   public Connection getConnection() throws SQLException, ClassNotFoundException {
      //N사 DB connection 생성 코드
   }
}
```

```
public class DUserDAO extends UserDAO{

   @Override
   public Connection getConnection() throws SQLException, ClassNotFoundException {
      //D사 DB connection 생성 코드
   }
}
```

Spring의 이해

✓ 상속을 통한 확장을 통해 두 개의 관심이 독립적으로 분리되면서 변경 작업이 한층 용이해 졌습니다.
이제는 UserDAO 클래스의 코드는 한 줄도 수정할 필요 없이 DB연결 기능을 새롭게 정의한 클래스를 만들 수 있습니다.
새로운 DB 연결 방법을 적용해야 할 때는 UserDAO 상속을 통해 확장해 주기만 하면 됩니다.

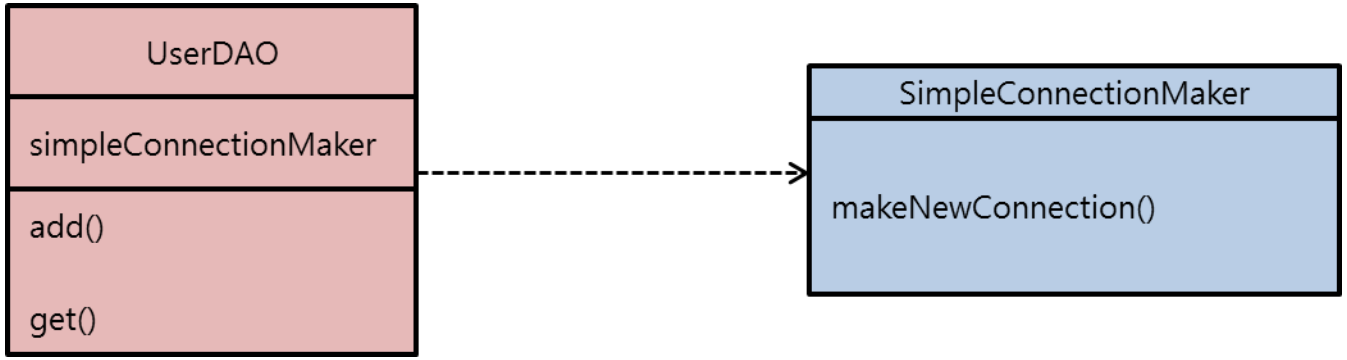
✓ 슈퍼 클래스에 기본적인 로직의 흐름을 만들고, 그 기능의 일부를 추상 메소드나 오버라이딩이 가능한 protected 메소드 등으로 만든 뒤 서브클래스에서 이런 메소드를 필요에 맞게 구현해서 사용하도록 하는 방법을 디자인 패턴에서 템플릿 메소드 패턴(template method pattern)이라고 하며 Spring에서 자주 사용되는 디자인 패턴입니다.

✓ 서브 클래스에서 구체적인 오브젝트 생성 방법을 결정하게 하는 것을 팩토리 메소드 패턴(factory method pattern)이라고 하기도 합니다.

Spring의 이해

✓ 세 번째 변화 : 클래스의 분리
두 개의 관심사를 본격적으로 독립시키면서 동시에 손쉽게 확장할 수 있는 방법

✓ 첫 번째는 독립된 메소드를 만들어서 분리했고, 두 번째는 상,하위 클래스로 분리했지만 이번에는 상속 관계도 아닌 완전히 독립적인 클래스로 만들어 분리합니다.



Spring의 이해

✓ UserDAO 클래스

```
public class UserDAO {
   private SimpleConnectionMaker simpleConnectionMaker;

   public UserDAO(){
      simpleConnectionMaker = new SimpleConnectionMaker();
   }

   public void add(User user) throws ClassNotFoundException, SQLException{
      Connection c = simpleConnectionMaker.makerNewConnection();
      …
   }
   public User get(String id) throws ClassNotFoundException, SQLException{
      Connection c = simpleConnectionMaker.makerNewConnection();
      …
   }
}
```

Spring의 이해

✓ SimpleConnectionMaker 클래스

```
public class SimpleConnectionMaker{
   public Connection makerNewConnection() throws ClassNotFoundException, SQLException{
      Class.forName("com.mysql.jdbc.Driver");
      Connection c = DriverManager.getConnection("jdbc.....");
      return c;
   }
}
```

✓ 상속은 몇 가지의 문제점을 가지고 있습니다. 그 문제점은 상위 클래스와 하위 클래스가 나름의 결합도를 갖는다는 것입니다. 서브 클래스는 슈퍼 클래스의 기능을 직접 사용할 수 있기 때문에 슈퍼 클래스의 변경이 일어나면 서브 클래스의 수정도 이루어져야 합니다. 반대로 그런 변화에 따른 불편을 주지 않기 위해 슈퍼 클래스가 더 이상 변화하지 않도록 제약을 가해야 하는 경우가 생길 수도 있습니다.

✓ 이 문제점을 해결하기 위해 상속이 아닌 완전히 다른 클래스로의 분리를 이루었지만 여기에도 문제점은 존재합니다.

Spring의 이해

✓ UserDAO와 SimpleConnectionMaker 클래스로 분리를 하게 되면 이전에 N사와 D사에 제공했던 것처럼 DB Connection 기능을 확장해서 사용하게 했던 것을 다시 할 수 없게 됩니다.

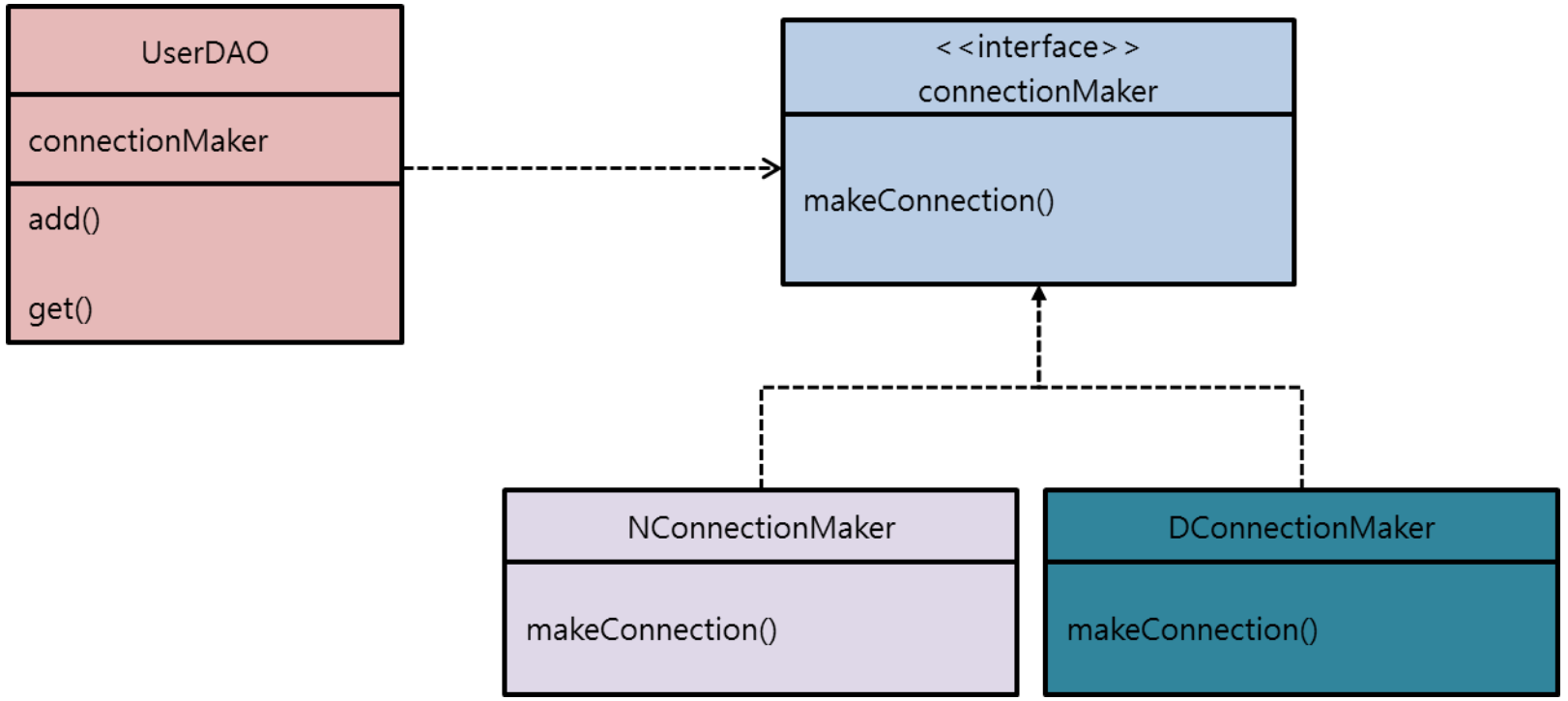
원인은 UserDAO가 SimpleConnectionMaker라는 특정 클래스에 종속되어 있기 때문에 상속을 사용했을 때처럼 UserDAO의 코드 수정 없이 DB Connection 생성 기능을 변경할 방법이 없습니다.

✓ 이 문제의 가장 좋은 해결책은 두 개의 클래스가 서로 긴밀하게 연결되어 있지 않도록 중간에 추상적인 느슨한 연결고리를 만들어 주는 것인데 이 때 사용되는 것이 interface 입니다.

✓ 추상화란 어떤 것들의 공통적인 성격을 뽑아내어 이를 따로 분리해 내는 작업이며 Java가 추상화를 위해 제공하는 가장 유용한 도구가 바로 interface 입니다.

Spring의 이해

✓ 인터페이스의 도입



Spring의 이해

✓ ConnectionMaker 인터페이스

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

✓ DConnectionMaker 클래스

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

Spring의 이해

✓ UserDAO 클래스

```
public class UserDAO {
   private ConnectionMaker connectionMaker;
   public UserDAO(){
      connectionMaker = new DConnectionMaker();
   }
   public void add(User user) throws ClassNotFoundException, SQLException{
      Connection c = connectionMaker.makeConnection();
      …
   }
   public User get(String id) throws ClassNotFoundException, SQLException{
      Connection c = connectionMaker.makeConnection();
      …
   }
}
```

✓ 인터페이스를 이용하여 관심사를 분리하고자 하는 클래스간의 추상적인 느슨한 결합을 이루었지만 여기에서도 문제점은 존재 합니다. 그 문제점은?

Spring의 이해

✓ 관계설정 책임의 분리

UserDAO와 ConnectionMaker라는 두개의 관심을 인터페이스를 사용하면서까지 완벽하게 분리했지만, 왜 UserDAO가 인터페이스뿐 아니라 구체적인 클래스까지 알아야 한다는 문제가 발생하는 것일까?

여전히 UserDAO에는 어떤 ConnectionMaker 구현 클래스를 사용할지를 결정하는 코드가 남아 있습니다. 이 때문에 인터페이스를 이용한 분리에도 불구하고 여전히 UserDAO 변경 없이는 DB 커넥션 기능의 확장이 자유롭지 못합니다.

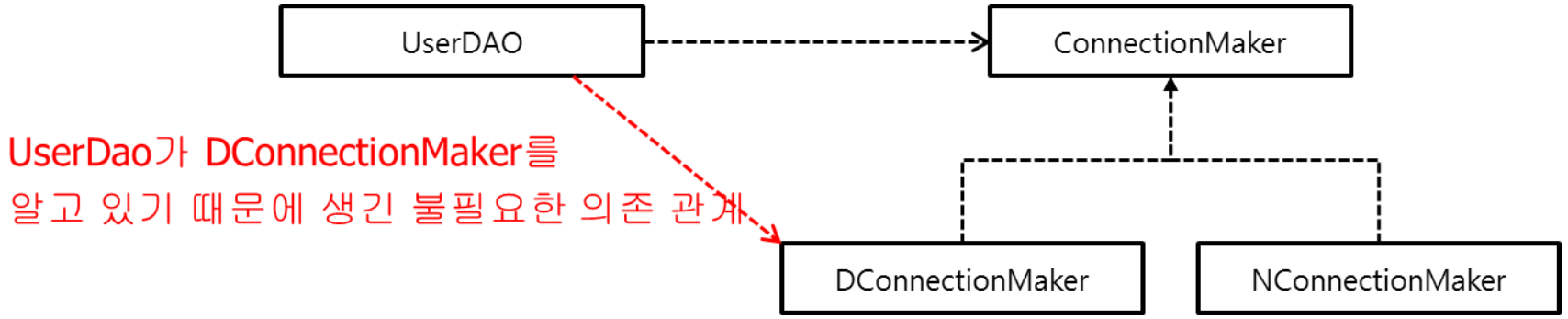
그 이유는 UserDAO 안에 분리되지 않은, 또 다른 관심사항이 존재하고 있기 때문입니다.

이 관심사항은 UserDAO와 UserDAO가 사용할 ConnectionMaker의 특정 구현 클래스 사이의 관계를 설정해주는 것에 대한 관심사 입니다.

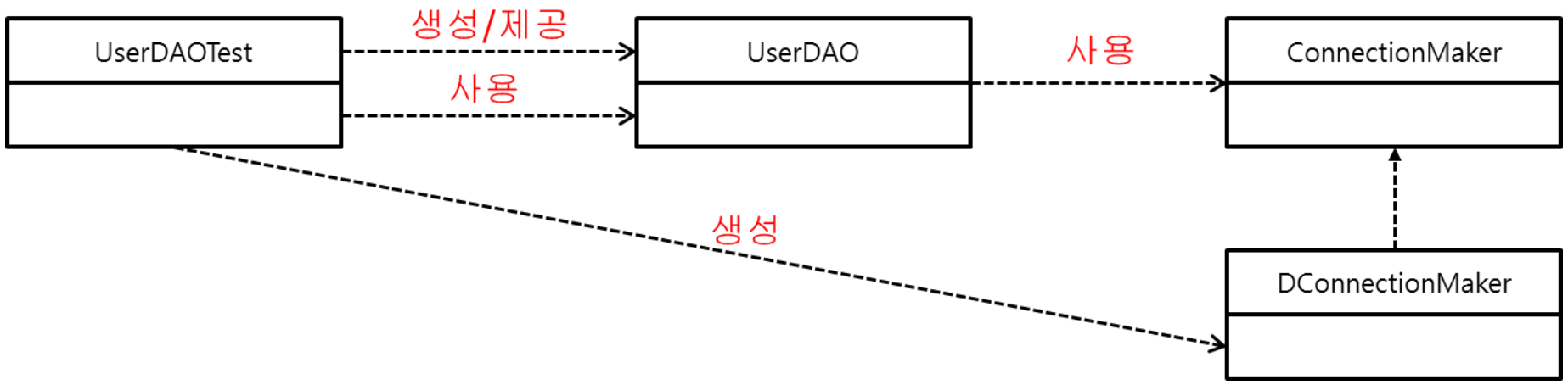
이 관심사를 담은 코드를 UserDAO에서 분리하지 않으면 UserDAO는 결코 독립적으로 확장 가능한 클래스가 될 수 없습니다.

Spring의 이해

✓ 문제점



✓ 문제점 해결 방법



Spring의 이해

✓ 수정된 UserDAO 클래스

```
public class UserDAO {
   private ConnectionMaker connectionMaker;
   public UserDAO(ConnectionMaker connectionMaker){
      this.connectionMaker = connectionMaker;
   }
   public void add(User user) throws ClassNotFoundException, SQLException{
      Connection c = connectionMaker.makeConnection();
      …
   }
   public User get(String id) throws ClassNotFoundException, SQLException{
      Connection c = connectionMaker.makeConnection();
      …
   }
}
```

Spring의 이해

✓ UserDAOTest 클래스

```
public class UserDAOTest {
   public static void main(String[] args) {
      ConnectionMaker connectionMaker = new DConnectionMaker();
    //UserDAO가 사용할 ConnectionMaker 구현 클래스를 결정하고 Object를 만든다.

      UserDAO dao = new UserDAO(connectionMaker);
    //1. UserDAO 생성
    //2. 사용할 ConnectionMaker 타입의 Object 제공
    //   결국 두 Object 사이의 의존관계 설정 효과
   }
}
```

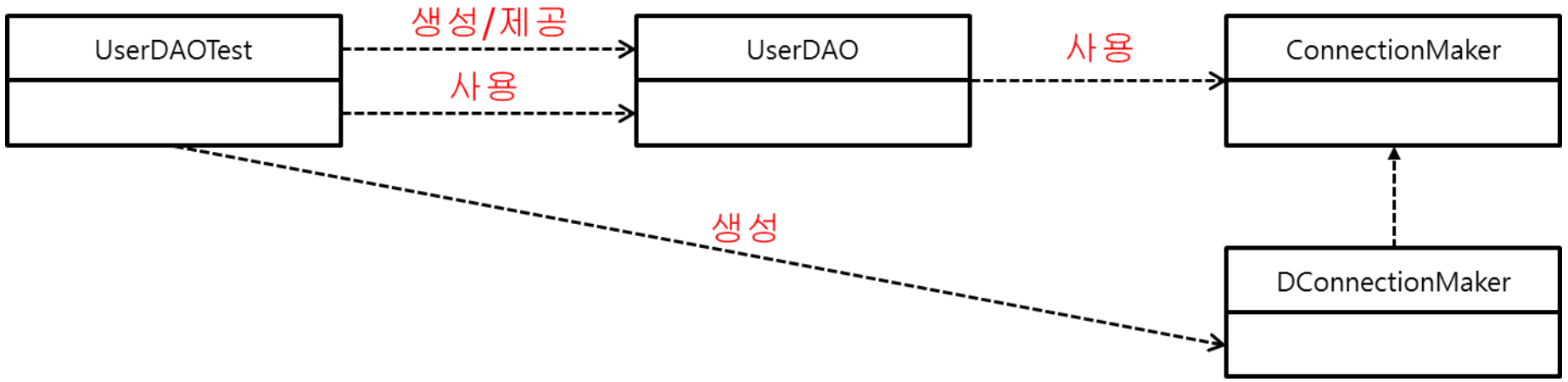
✓ DConnectionMaker 클래스

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

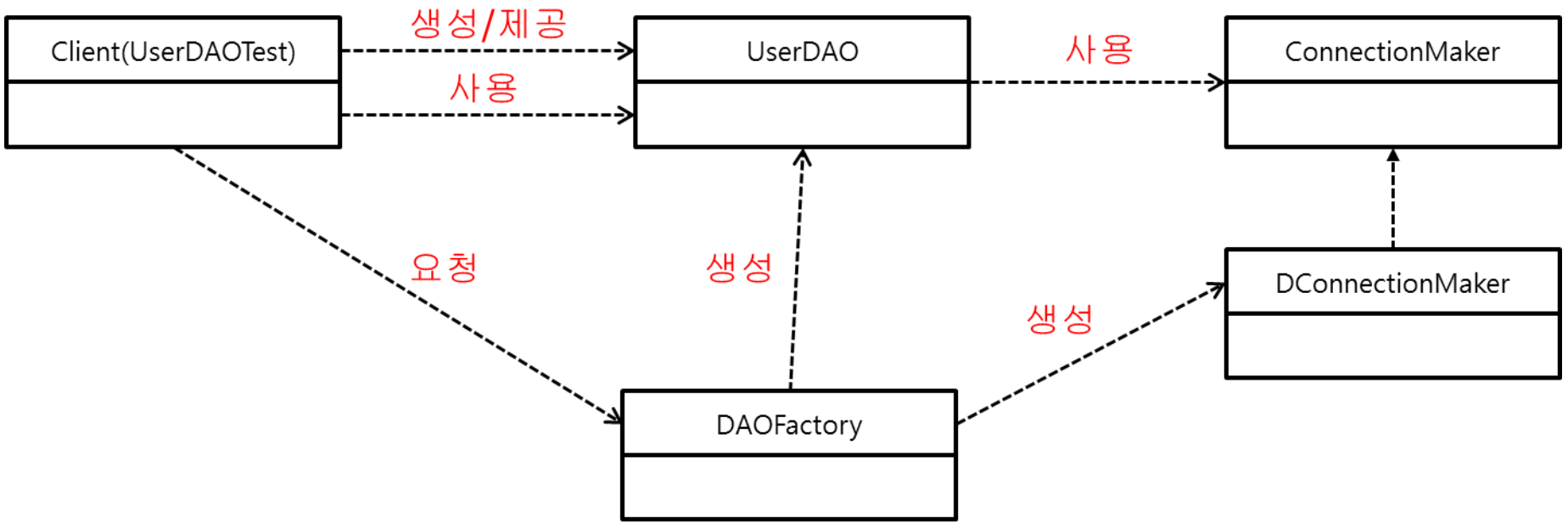
Spring의 이해

✓ 팩토리(Factory)
객체의 생성 방법을 결정하고 그렇게 만들어진 객체를 반환하는 것을 말합니다.

✓ 객체를 생성하는 쪽과 생성된 인스턴스 객체를 사용하는 쪽의 역할과 책임을 분리하려는 목적으로 사용합니다.



Spring의 이해


Client(UserDAOTest)
생성/제공
사용
UserDAO
사용
ConnectionMaker
요청
생성
생성
DConnectionMaker
DAOFactory

Spring의 이해

✓ DAOFactory 클래스

```
public class DAOFactory {

   public UserDAO userDAO(){
      ConnectionMaker connectionMaker = new DConnectionMaker();
      UserDAO userDAO = new UserDAO(connectionMaker);
      return userDAO;
   }
}
```

✓ UserTest 클래스

```
public class UserDAOTest {
   public static void main(String[] args) {
      UserDAO dao = new DAOFactory().userDAO();
   }
}
```

Spring의 이해

- ✓ UserDAO와 ConnectionMaker는 각각 Application의 핵심적인 데이터 로직과 기술 로직을 담당하고, DAOFactory는 이런 Application의 Object를 구성하고 그 관계를 정의하는 책임을 맡고 있습니다.

- ✓ UserDAO와 ConnectionMaker 클래스가 실직적인 로직을 담당하는 컴포넌트라면 DAOFactory는 Application을 구성하는 컴포넌트의 구조와 관계를 정의하는 설계도 같은 역할을 한다고 할 수 있습니다.

- ✓ DAOFactory를 분리했을 때 얻을 수 있는 장점은 애플리케이션의 컴포넌트 역할을 하는 객체와 애플리케이션의 구조를 결정하는 객체를 분리했다는 것에 가장 큰 의미가 있습니다.

- ✓ 객체 팩토리의 활용
  UserDAO 이외에 좀더 다양한 DAO 클래스가 새로 생긴다면? (AccountDAO, MessageDAO…)

Spring의 이해

```
public class DAOFactory {
   public UserDAO userDAO(){
      return new UserDAO(connectionMaker());
   }
   public AccountDAO accountDAO(){
      return new AccountDAO(connectionMaker());
   }
   public MessageDAO messageDAO(){
      return new MessageDAO(connectionMaker());
   }
   public ConnectionMaker connectionMaker(){
      return new DConnectionMaker();
   }
}
```

Spring IoC(Inversion of Control)

✓ IoC(제어의 역전)

✓ 프로그램의 제어 흐름 구조가 뒤바뀌는 것을 뜻합니다.

일반 적인 객체지향 프로그램에서는 모든 객체가 능동적으로 자신이 사용할 클래스를 결정하고, 언제 어떻게 그 객체를 만들지를 스스로 관리합니다. 모든 종류의 작업을 사용하는 쪽에서 제어하는 구조인 것입니다.

제어의 역전이란 이런 흐름의 개념을 거꾸로 뒤집는 것을 말합니다. 제어의 역전에서는 객체가 자신이 사용할 객체를 스스로 선택하지 않고, 생성하지도 않습니다. 또 자신도 어떻게 만들어지고 어디서 사용되는지 알 수 없습니다.

모든 제어 권한을 자신이 아닌 다른 대상에게 위임합니다.

Spring IoC(Inversion of Control)

[Spring IoC]

✓ Bean
Spring이 IoC 방식으로 관리하는 객체를 뜻합니다. Spring이 직접 그 생성과 제어를 담당하는 모든 객체를 뜻합니다.

✓ Bean Factory
Spring의 IoC를 담당하는 핵심 컨테이너입니다.
Bean을 등록하고, 생성하고, 조회하고 돌려주고, 그 외에 부가적인 Bean을 관리하는 기능을 담당합니다.

✓ Application Context
Bean Factory를 확장한 IoC 컨테이너입니다. Bean을 등록하고 관리하는 기본적인 기능은 Bean Factory와 동일합니다
여기에 Spring이 제공하는 각종 부가 서비스를 추가로 제공합니다.

Spring IoC(Inversion of Control)

✓ DAOFactory를 Spring의 Bean Factory가 사용할 수 있는 설정 정보로 변환하는 방법

```
// Application Context 또는 Bean Factpry가 사용할 설정 정보라는 표시
@Configuration
public class DAOFactory {
   @Bean // Object 생성을 담당하는 IoC용 메소드라는 표시
   public UserDao userDao(){
      return new UserDAO(connectionMaker());
   }
   @Bean
   public ConnectionMaker connectionMaker(){
      return new DConnectionMaker();
   }
}
```

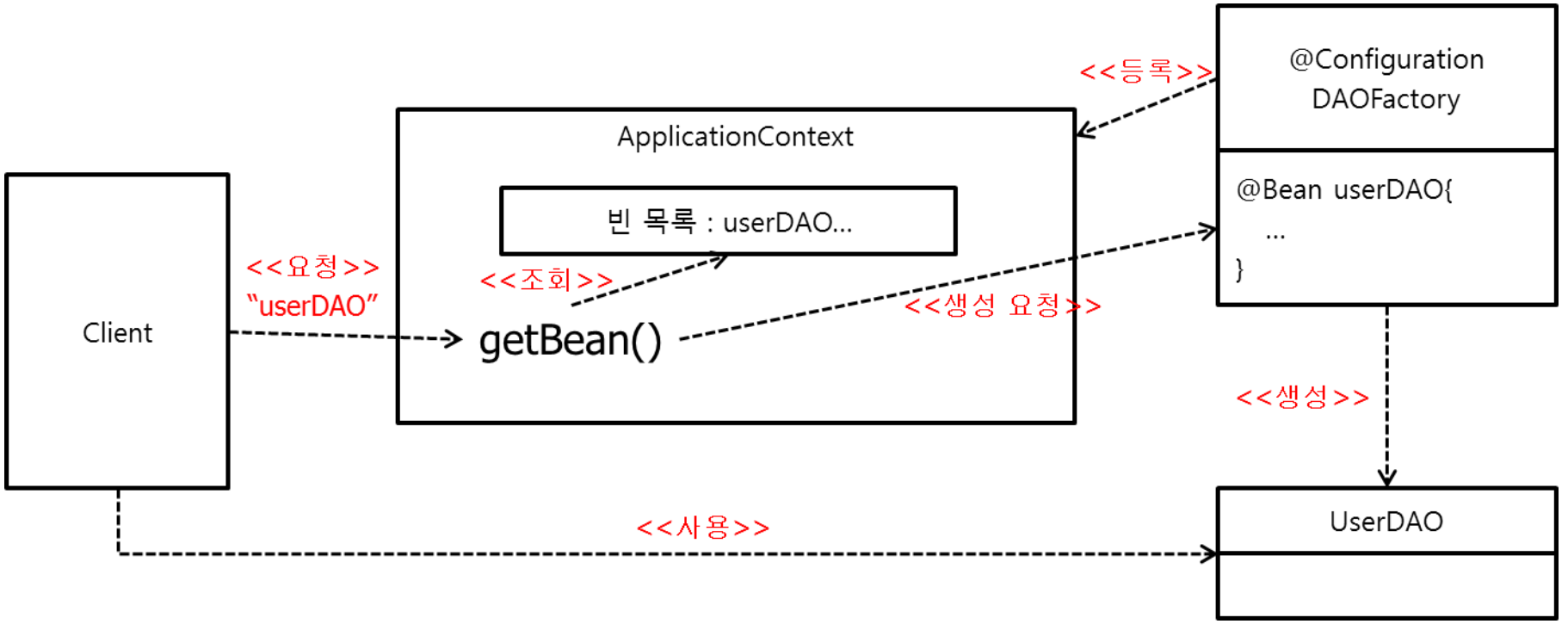
Spring IoC(Inversion of Control)

✓ Application Context를 적용한 UserDAOTest 클래스

```
public class UserDAOTest {
   public static void main(String[] args) {
      ApplicationContext context = new AnnotationConfigApplicationContext(DAOFactory.class);
      UserDAO dap = context.getBean("userDAO", UserDAO.class);
   }
}
```

Spring IoC(Inversion of Control)

✓ Application Context의 동작 방식



Spring IoC(Inversion of Control)

✓ ApplicationContext를 사용했을 때의 장점은 다음과 같습니다.

(1) 클라이언트는 구체적인 팩토리 클래스를 알 필요가 없습니다.
애플리케이션이 커지면 DAOFactory처럼 IoC를 적용한 객체는 계속 추가될 것이고 클라이어트 입장에서 필요한 객체를 얻기 위해서는 어떤 팩토리 클래스를 사용해야 할 지 알아야 합니다. 그리고 필요할 때마다 팩토리를 생성해야 하는 번거로움이 존재합니다.
ApplicationContext를 사용하면 이와 같은 작업이 필요 없고 XML 형태의 IoC 설정 정보의 생성 또한 가능합니다.

(2) ApplicationContext는 종합 IoC 서비스를 제공합니다.
ApplicationContext는 단순히 객체의의 생성과 관계 설정 이외에도 객체의 생성 방식, 시점과 전략을 다르게 할 수 있고, 자동 생성, Clean-up등 다양한 기능을 제공합니다.

(3) Bean을 검색하는 다양한 방법을 제공합니다.
ApplicationContext의 getBean() 메소드는 Bean의 이름으로 검색하고 이외에도 타입을 이용한 Bean의 검색이나 Annotation을 통해 Bean을 검색 할 수도 있습니다.

Spring IoC(Inversion of Control)

✓ DAOFactory의 직접 사용과 Spring의 ApplicationContext를 통한 IoC 적용의 또 다른 차이점은 싱글톤 레지스트리 개념에 있습니다.

```
ApplicationContext context = new AnnotationConfigApplicationContext
                                                    (DAOFactory.class);
UserDAO dao1 = context.getBean("userDAO", UserDAO.class);
UserDAO dao2 = context.getBean("userDAO", UserDAO.class);
System.out.println(dao1);
System.out.println(dao2);
```

✓ Spring이 주로 적용되는 대상은 Java Enterprise 기술을 사용하는 서버 환경이 됩니다. 따라서 서버 환경에 대한 고려가 필요한데 일반적으로 서버 환경에서는 동시 사용자가 많기 때문에 각 사용자가 특정 서비스를 사용할 때마다 객체를 생성할 수는 없습니다.(Servlet을 생각해보면…) 따라서 Spring의 ApplicationContext도 특별한 설정이 없는 경우 Bean 객체를 싱글톤 레지스트리를 적용한 단일 객체로 제공합니다.

Spring IoC(Inversion of Control)

✓ Singleton Pattern
Singleton Pattern이란 객체의 생성자를 private으로 하여 외부에서의 객체생성을 제한합니다.

✓ Singleton Pattern의 단점

(1) private 생성자를 갖고 있기 때문에 상속할 수 없습니다.

(2) Singleton은 테스트의 제한이 있습니다.

(3) 서버 환경에서는 Singleton이 하나만 만들어지는 것을 보장할 수 없습니다.

(4) Singleton의 사용은 전역 상태를 만들 수 있습니다.

✓ Spring에서는 Singleton registry를 제공하기 때문에 Singleton이 갖는 단점이 존재하지 않습니다.

✓

✓ Spring에서 사용되는 모든 Bean 객체는 일반 클래스와 동일한 형태로 만드는 것이 가능하고 ApplicationContext는 이러한 Bean 객체들을 단일한 객체로 생성하여 제공합니다.

Spring IoC(Inversion of Control)

✓ Bean 객체의 생성시 주의할 점
Spring에서의 Bean 객체들은 특별한 조건이 없을 경우 단일 객체로 생성되기 때문에 Thread 상황에서의 데이터 사용에 대한 주의를 기울여야 합니다.
즉, Bean 객체의 필드로 자주 변경되는 값을 지정하면 안됩니다.

✓ Spring Bean의 Scope
Spring은 경우에 따라 다양한 Scope(생성되고, 사용되고, 적용되는 범위)를 갖습니다.

- Singleton Scope
  대부분의 Bean 객체가 갖는 Scope이며, 강제로 제거하지 않는 한 컨테이너에 존재합니다.
- Prototype Scope : 요청할 때마다 매번 새로운 객체 생성합니다.
- Request Scope : Http 요청이 있을 때마다 생성합니다.
- Session Scope : Web의 Session과 유사한 Scope를 갖습니다.

Spring DI

✓ DI : Dependency Injection(의존 관계 주입)
의존성(Dependency)이란 특정 클래스가 자신의 업무를 수행하기 위해 필연적으로 맺게 되는 다른 클래스와의 관계를 뜻합니다.

주입(Injection)이란 다른 클래스와의 관계를 나 자신이 주도적으로 맺는 것(해당 객체의 생성이나 참조)이 아니라 외부로부터 관계 설정이 맺어 지는 것을 뜻합니다.

✓ Spring 컨테이너의 역할은 특정 클래스가 필요로 하는 값이나 인스턴스를 생성, 취득하고, 관계를 설정하는 것입니다. 이를 통해 관계 설정을 위한 코드를 생략할 수 있고 클래스간의 관계를 보다 느슨하게 할 수 있습니다.

Spring DI

✓ Spring DI의 다양한 방법

- 생성자를 통한 주입
- 설정 메소드를 통한 주입
- Look-up 메소드를 이용한 주입

✓ 생성자를 통한 주입과 설정 메소드를 통한 주입

```
public class A {
   private B b;
   public A(B b){
      this.b = b;
   }
}
```

```
public class A {
   private B b;
   public setB(B b){
      this.b = b;
   }
}
```

Spring DI

✓ BeanFactory Class
Bean의 생성과 설정, 관리 등의 역할을 담당하고 있는 클래스입니다.

✓ **주요 메소드**

- boolean containsBean(String name) : 인수에 지정된 이름의 Bean이 정의되어 있는지 여부를 반환합니다.
- String [] getAliases(String name) : Bean 이름에 별칭이 정의되어 있는 경우 해당 별칭을 반환합니다.
- Object getBean(String name) : 인수에 지정된 이름의 Bean 인스턴스를 생성해서 반환합니다.
- <T> T getBean(String name, Class<T> requiredType) : 인수에 지정된 이름의 Bean 인스턴스를 생성해서 반환합니다.
- Class<?> getType(String name) : 인수에 지정된 이름의 Bean의 타입을 반환합니다.
- boolean isSingleton(String name) : Bean이 Singleton인지 여부를 반환합니다.

Spring DI

✓ 설정파일(bean.xml)의 name-space 종류

name space	목적
aop	Aspect 선언을 위한 요소와 @AspextJ 어노테이션이 적용된 클래스를 자동으로 스프링 Aspect로 프록시하는 요소를 제공합니다.
beans	핵심 원시 스프링 네임스페이스로, 빈의 선언과 연결 방법을 정의할 수 있도록 합니다.
context	스프링에 의해 직접 관리되지 않는 객체의 주입과 빈을 오토디텍트(autodetect)하고 오토와이어링(autowiring)하는 기능을 포함하여 스프링 애플리케이션 컨텍스트를 설정하기 위한 요소를 제공합니다.
jee	JNDI와 EJB등 Java EE API와 통합을 제공합니다.
jms	메시지 드리블 POJO를 선언하기 위한 설정 요소를 제공합니다.
lang	Groovy, Jruby, 또는 BeanShell 스크립트로 구현되는 빈의 선언을 가능하게 합니다.
mvc	어노테이션 지향 컨트롤러, 뷰 컨트롤러, 인터셉터 등의 스프링 MVC 기능을 가능하게 합니다.
oxm	스프링의 객체 대 XML 매핑 구조의 설정을 지원합니다.
tx	선언적 트랜젝션 설정을 제공합니다.
util	유틸리티 요소의 다양한 선택, 컬렉션을 빈으로 선언하는 긴으과 프로퍼티 대치 요소에 대한 지원을 포함합니다.

Spring DI

✓ 설정 파일(bean.xml)의 요소
Spring에서는 Bean을 관리하기 위하 XML을 정의하게 되는데 요소의 속성은 다음과 같습니다.

속성	의미	기본값
id	Bean에 붙이는 식별자(고유값)	-
name	id에 대한 별칭. 복수의 정의 가능	-
class	Bean 클래스 이름. 완전한 형태의 클래스 이름으로 기술합니다.	-
parent	Bean 정의를 상속하는 경우 지정하는 새로운 Bean의 id	-
abstract	Bean 클래스가 추상 클래스인지 여부	false
singleton	Bean이 singleton으로 관리 되는지의 여부	true
lazy-init	Bean의 로딩을 지연시킬지 여부	default
autowire	오토와이어 설정	Default
dependency-check	의존 관계 확인 방법	default
depends-on	해당 Bean이 의존한 Bean의 이름. 먼저 초기화 되는 것이 보장됩니다.	-
init-method	Bean 초기화에 사용될 메소드 지정	-
destroy-method	Bean 컨테이너 종료시에 사용될 메소드 지정	-

Spring DI

✓ bean.xml의 <constructor-arg>와 <property> 요소의 속성

- index : 생성자의 몇 번째 인수에 값을 넘길 것인가를 지정합니다.
- type : 생성자의 어떤 데이터 타입인 인수에 값을 넘길 것인가를 지정합니다.
- ref : 자식 요소<ref bean="Bean 이름"/>대신 사용할 수 있습니다.
- value : 자식 요소 <value>값</value> 대신 사용할 수 있습니다.

<property> 요소는 ref와 value 속성만을 가집니다.

✓ Bean을 담는 그릇, 컨테이너
Spring Application에서는 Spring Container에서 객체가 태어나고, 자라고, 소멸합니다. Spring Container는 객체를 생성하고, 서로 관계를 맺어주고, 이들의 전체 생명주기를 관리합니다.

✓ Spring Container는 DI를 이용해서 Application을 구성하는 컴포넌트를 관리하며, 협력 컴포넌트 간 연관관계의 형성도 여기에서 이루어집니다.

Spring DI

✓ Spring에는 여러 Container 구현체가 존재하며 크게 두 가지로 분류됩니다.

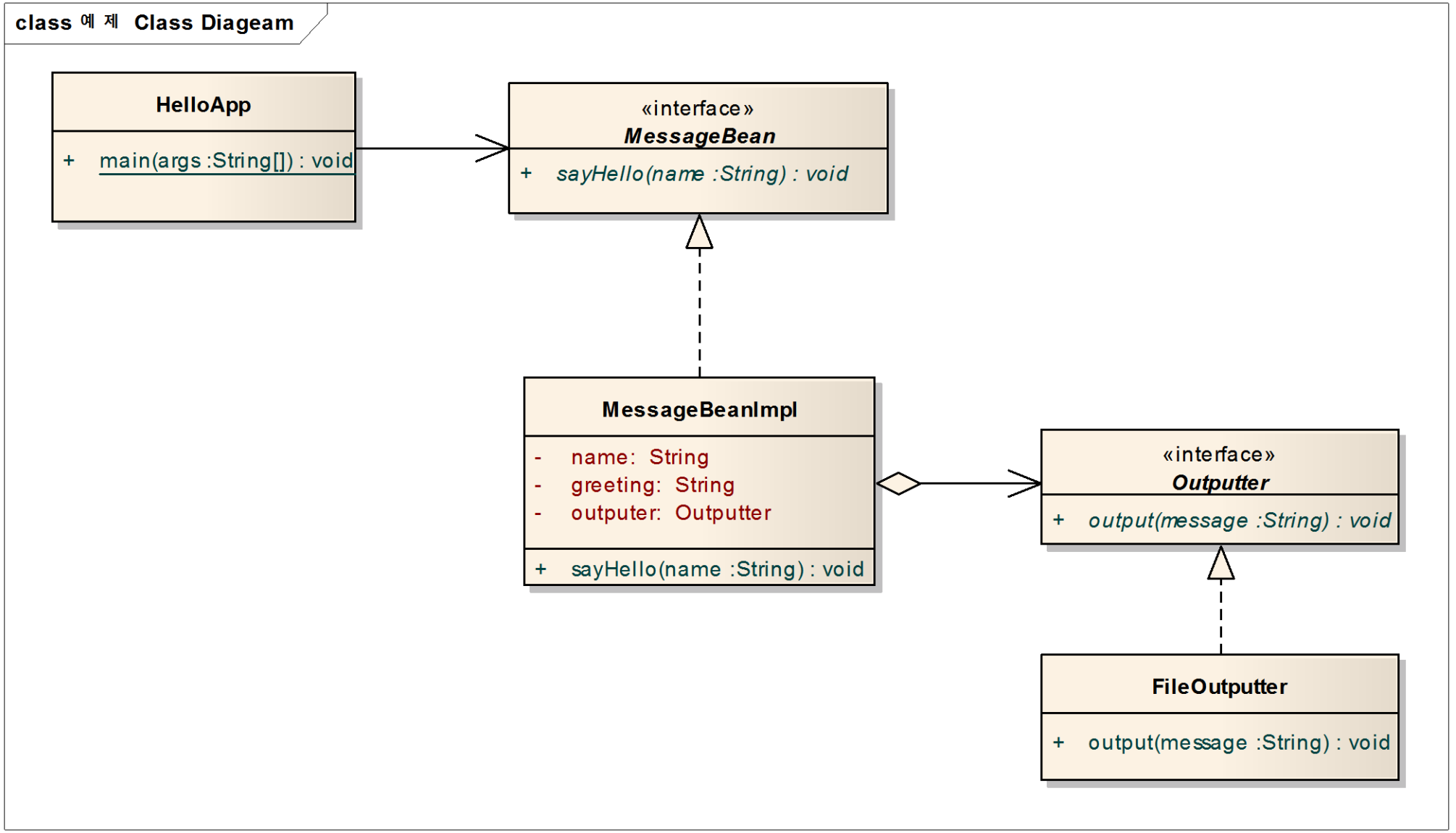
(1) BeanFactory : DI에 대한 기본적인 지원을 제공하는 가장 단순한 컨테이너

(2) ApplicationContext : BeanFactory를 확장한 Container. BeanFactory의 경우 저수준의 기능을 제공하고 있기 때문에 ApplicationContext의 사용을 선호합니다.

✓ ApplicationContext 컨테이너의 종류

- ClassPathXmlApplicationContext : 클래스 패스에 위치한 XML 파일에서 컨텐스트 정의 내용을 로드합니다.
- FileSystemXmlApplicationContext : 파일 경로로 지정된 XML 파일에서 컨텍스트 정의 내용을 로드합니다.
- XmlWebApplicationContext : Web Application에 포함된 XML 파일에서 컨텍스트 정의 내용을 로드합니다.

Spring DI


class 예 제 Class Diageam
HelloApp
+ main(args :String[]) : void
«interface»
MessageBean
+ sayHello(name :String) : void
MessageBeanImpl
- name: String
- greeting: String
- outputer: Outputter
+ sayHello(name :String) : void
«interface»
Outputter
+ output(message :String) : void
FileOutputter
+ output(message :String) : void