DI(Dependency Injection)와 서비스 로케이터

소프트웨어의 두 개의 영역

1. 어플리케이션 영역: 고수준 정책, 저수준 구현을 포함
2. 메인 영역: 어플리케이션이 동작하도록 각 객체들을 연결

DI(Dependency Injection): 메인 영역에서 객체를 연결하기 위해 사용되는 방법

메인(main) 영역

* 어플리케이션 영역에서 사용될 객체를 생성한다.
* 각 객체 간의 의존 관계를 설정한다.
* 어플리케이션을 실행한다.

어플리케이션 영역의 객체 초기화, 의존 처리, 실행을 담당한다.

모든 의존은 메인 영역에서 어플리케이션 영역으로 향한다.

메인 영역을 변경한다고 해도 어플리케이션 영역은 변경되지 않는다는 것을 뜻한다.

서비스 로케이터(Service Locator): 사용할 객체를 제공하는 책임을 갖는 객체

* ‘로케이터’를 통해서 필요로 하는 객체를 직접 찾는 방식
* 몇 가지 단점이 존재해서 서비스 ‘로케이터’를 사용하는 것 보다는 외부에서 사용할 객체를 주입해 주는 DI 방식을 사용하는 것이 일반적

서비스 로케이터 단점

* 사용할 객체를 직접 생성할 경우, 콘크리트 클래스에 대한 의존이 발생하게 된다.
* 콘크리트 클래스를 직접 사용해서 객체를 생성하게 되면 의존 역전 원칙을 위반하게 되며, 결과적으로는 확장 폐쇄 원칙을 위반하게 된다.
* 변화에 경직된 유연하지 못한 코드를 만들게 된다.

DI: 필요한 객체를 직접 생성하거나 찾지 않고 외부에서 넣어주는 방식

클래스에서 사용할 객체를 전달받을 수 있는 생성자를 추가하는 것으로 DI를 적용할 수 있게 된다.

누군가 외부에서 의존하는 객체를 넣어 주기 때문에, 이런 방식을 의존(dependency) 주입(injection)이라고 부른다.

DI를 통해서 의존 객체를 관리할 때에는 객체를 생성하고 각 객체들을 의존 관계에 따라 연결해 주는 조립 기능이 필요하다.

조립기를 별도로 분리하면 향후에 조립기 구현 변경의 유연함을 얻을 수 있음.

스프링 프레임워크가 바로 객체를 생성하고 조립해 주는 기능을 제공하는 DI 프레임워크이다.

DI 적용 방식

* 생성자 방식: 생성자를 통해서 의존 객체를 전달받는 방식, 생성자를 통해 전달받은 객체를 필드에 보관한 뒤, 메서드에서 사용
* 설정 메서드 방식: 메서드를 이용해서 의존 객체를 전달받는 방식, 파라미터로 전달받은 의존 객체를 필드에 보관하며, 다른 메서드에서는 필드를 사용해서 의존 객체의 기능을 실행.

생성자 방식이나 설정 메서드 방식을 이용해서 의존 객체를 주입할 수 있게 되었다면, 조립기는 생성자와 설정 메서드를 이용해서 의존 객체를 전달하게 된다.

생성자 방식과 설정 메서드 방식 중 아무거나 사용해도 상관없음. 다만 생성자 방식은 객체를 생성하는 시점에 필요한 모든 의존 객체를 준비할 수 있고, 객체를 생성하는 시점에 의존 객체가 정상인지 확인할 수 있어서 선호도가 높다.

생성자 방식을 사용하려면 의존 객체가 먼저 생성되어 있어야 하므로, 의존 객체를 먼저 생성할 수 없다면 생성자 방식을 사용할 수 없게 된다.

설정 메서드 방식은 객체를 생성한 이후에 의존 객체를 설정할 수 있기 때문에, 어떤 이유로 인해 의존할 객체가 나중에 생성된다면 설정 메서드 방식을 사용해야 한다. 다만 ‘NullPointerException’ 조심. 또한 의존할 객체가 많을 경우 이름을 통해서 어떤 의존 객체가 설정되는지 쉽게 알 수 있어서, 코드 가독성을 높여주는 효과가 있다.

DI를 적용하면, Mock 객체를 이용해서 테스트를 하기 때문에 기존 다른 코드를 변경할 필요가 없게 된다.

프로그램 개발 환경이나 사용하는 프레임워크의 제약으로 인해 DI 패턴을 사용할 수 없는 경우 (안드로이드)에는 다양한 방법 중 ‘서비스 로케이터’가 있다.

서비스 로케이터는 어플리케이션에서 필요로 하는 객체를 제공하는 책임을 갖는다.

서비스 로케이터는 의존 대상이 되는 객체 별로 제공 메서드를 정의한다.

의존 객체가 필요한 코드에서는 서비스 로케이터가 제공하는 메서드를 이용해서 필요한 객체를 구한 뒤 알맞은 기능을 실행한다.

서비스 로케이터가 올바르게 동작하려면 서비스 로케이터 스스로 어떤 객체를 제공해야 할지를 알아야 한다.

서비스 로케이터는 어플리케이션 영역에 위치한다. 메인 영역에서는 서비스 로케이터가 제공할 객체를 생성하고, 이 객체를 이용해서 서비스 로케이터를 초기화 해준다.

서비스 로케이터 구현 방법

* 객체 등록 방식의 구현
* 상속을 통한 구현

객체 등록 방식의 구현

* 서비스 로케이터를 생성할 때 사용할 객체를 전달
* 서비스 로케이터 인스턴스를 지정하고 참조하기 위한 static 메서드를 제공
* 서비스 로케이터가 제공할 객체 종류가 많을 경우 객체마다 별도의 등록 메서드를 제공하는 방식을 취해서 서비스 로케이터 초기화 부분의 가독성을 높일 수 있다.
* 객체를 등록하는 인터페이스가 노출되어 있기 때문에 어플리케이션 영역에서 얼마든지 의존 객체를 바꿀 수 있다.

상속을 통한 구현

* 객체를 구하는 추상 메서드를 제공하는 상위 타입 구현
* 상위 타입을 상속받은 하위 타입에서 사용할 객체 설정
* 추상 클래스라는 것은 이 클래스를 상속받아 추상 메서드의 구현을 제공하는 클래스가 필요하다는 뜻. 이 클래스는 어플리케이션에서 실제로 필요로 하는 의존 객체를 생성.
* 추상 클래스를 상속받은 클래스는 메인 영역에 위치한다. 이렇게 함으로써 의존 객체를 교체해야 할 경우 어플리케이션 영역의 코드 수정 없이 메인 영역의 코드만 수정한다.
* 추상 클래스의 구현을 제공하는 클래스를 만들었다면, 메인 영역에서 이 클래스의 객체를 생성해 주기만 하면 된다.

서비스 로케이터의 단점

* 인터페이스 분리 원칙 위반
* 원하지 않는 의존을 받을 수 있다.
* 의존 객체마다 서비스 로케이터를 작성해 주어야 한다. 문제는 클래스를 중복해서 만드는 문제를 야기할 수 있다. (타입만 다르고 구조가 완전히 같은 로케이터 클래스들)
* 자바 제네릭 기반을 이용해서 구현하면 중복된 코드를 피하면서 인터페이스를 분리한 것과 같은 효과를 낼 수 있다.
* 가장 큰 단점은 동일 타입의 객체가 다수 필요할 경우, 각 객체 별로 제공 메서드를 만들어주어야 한다는 점.

서비스 로케이터는 DI에 변경의 유연함을 떨어뜨리는 문제를 갖고 있기 때문에, 부득이한 상황이 아니라면 서비스 로케이터보다는 DI를 사용하자.