토비의 스프링 6 - 이해와 원리 참고 자료

최종 업데이트: 2024/06/23

이 문서는 토비의 스프링 6 - 이해와 원리 강의에 대한 보충 설명, 참고할 링크 등을 제공하기 위해서 작성된 것으로, 강의에 대한 수강생들의 질문이나 추가 설명이 필요한 경우에 계속 업데이트 될 수 있습니다.

01 스프링 개발환경

Spring 6.0.12 또는 그 이후 버전

(SpringBoot 3.1.4+)

2024년 6월 기준

Spring 6.1.8 또는 그 이후 버전

(SpringBoot 3.3.0+)

개발환경 셋업 (Mac)

JDK

JDK 17 이후 버전

다음 중 하나의 방법 선택

- IntelliJ 설치후 SDK - Download JDK
- 공개 JDK 다운로드 후 설치
  - https://aws.amazon.com/corretto/
  - https://adoptium.net/en-GB/temurin/releases/
- SDKMAN (https://sdkman.io)
  - sdk install java [jdk name]
  - sdk java use [jdk name]
  - 프로젝트 폴더의 사용 JDK 저장: sdk env init
  - 프로젝트 폴더의 사용 JDK 복구: sdk env

IDE

- IntelliJ Ultimate
- IntelliJ Community

HTTPie

- https://httpie.io/

HelloSpring 프로젝트 생성

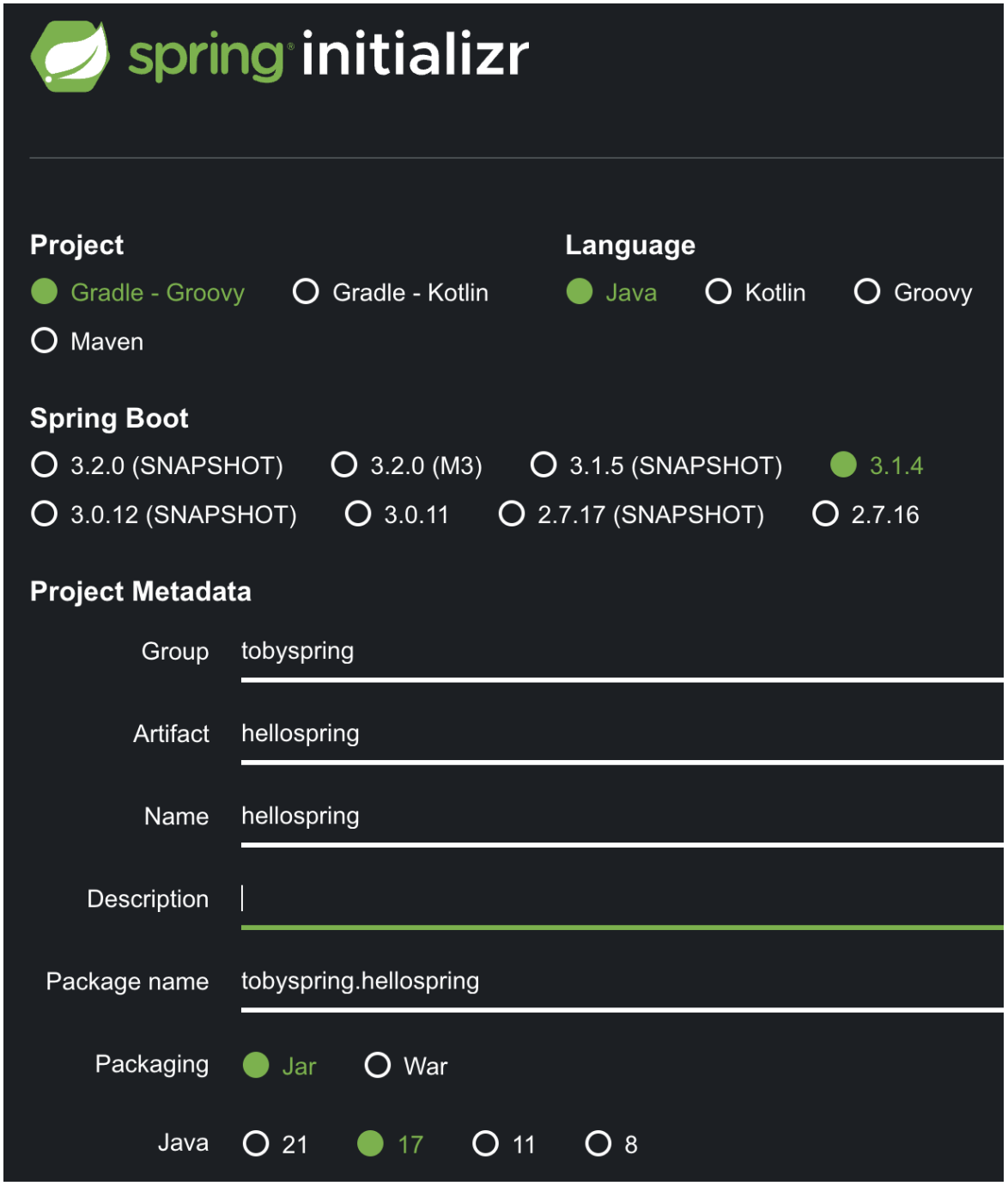
스프링 부트 프로젝트로 생성

스프링 부트의 의존 라이브러리 설정을 이용하기 위함

- IntelliJ Ultimate의 New Project - Spring Initializr
- IntelliJ Community
  - Web Spring Initializr - https://start.spring.io/
    - 다운로드 후 zip 파일 풀고 import
  - Spring Initializr 플러그인

생성 옵션

- Project: Gradle-Groovy (또는 Gradle-Kotlin)
- Language: Java
- Spring Boot: 3.1.4 (혹은 이후 버전)
- Project Metadata
  - Group: tobyspring
  - Artifact: hellospring
  - Name: hellospring
- Packaging: Jar
- Java: 17 (또는 그 이후 버전)



PaymentService 개발

요구사항

- 해외직구를 위한 원화 결제 준비 기능 개발
- 주문번호, 외국 통화 종류, 외국 통화 기준 결제 금액을 전달 받아서 다음의 정보를 더해 Payment를 생성한다
  - 적용 환율
  - 원화 환산 금액
  - 원화 환산 금액 유효시간

- PaymentService.prepare() 메소드로 개발
    - Payment 오브젝트 리턴

개발방법

- 빠르게 완성해서 가장 간단한 방법을 찾는다
- 작성한 코드가 동작하는지 확인하는 방법을 찾는다
- 조금씩 기능을 추가하고 다시 검증한다
- 코드를 한눈에 이해하기 힘들다면 코멘트로 설명을 달아준다

환율 가져오기

- https://open.er-api.com/v6/latest/{기준통화} 이용
    - 이 서비스가 더이상 유지되지 않는 경우 사용할 다른 서비스 URL을 강의자료에서 확인
- JSON 포맷으로 리턴되는 값을 분석해서 원화(KRW) 환율 값을 가져온다
- JSON을 자바 오브젝트로 변환
    - Jackson 프로젝트의 ObjectMapper 사용

개발된 코드 이대로 좋은가?

- 무엇이 문제인지 생각해보자
- 그러면 어떻게 개선할 수 있을까?

02 오브젝트와 의존관계

스프링의 관심은 오브젝트이다.

클래스와 오브젝트는 다르다.

자바의 오브젝트는 클래스의 인스턴스 또는 배열이다.

의존관계(dependency)

의존관계는 반드시 두 개 이상의 대상이 존재하고, 하나가 다른 하나를 사용, 호출, 생성, 인스턴스화, 전송 등을 할 때 의존관계에 있다고 이야기한다.

클래스 사이에 의존관계가 있을 때 의존 대상이 변경되면 이를 사용하는 클래스의 코드도 영향을 받는다.

오브젝트 사이에 의존관계는 실행되는 런타임에 의존관계가 만들어지고 실행 기능에 영향을 받지만 클래스 레벨의 의존관계와는 다를 수 있다.

- https://www.ibm.com/docs/en/dmrt/9.5?topic=diagrams-dependency-relationships

관심사의 분리(Separation of Concerns)

관심사는 동일한 이유로 변경되는 코드의 집합

API를 이용해서 환율정보를 가져오고 JSON을 오브젝트에 매핑하는 관심과 Payment를 준비하는 로직은 관심이 다르다. 변경의 이유와 시점을 살펴보고 이를 분리한다.

메소드 추출(extract method) 리팩터링

- https://refactoring.guru/extract-method

> 💡 **리팩터링**
>
> 리팩터링은 기존의 코드를 외부의 동작방식에는 변화 없이 내부 구조를 변경해서 재구성하는 작업 또는 기술을 말한다. 리팩토링을 하면 코드 내부의 설계가 개선되어 코드를 이해하기가 더 편해지고, 변화에 효율적으로 대응할 수 있다. 결국 생산성은 올라가고, 코드의 품질은 높아지며, 유지보수하기 용이해지고, 견고하면서도 유연한 제품을 개발할 수 있다. 리팩토링이 절실히 필요한 코드의 특징을 나쁜 냄새라고 부르기도 한다. 대표적으로, 중복된 코드는 매우 흔하게 발견되는 나쁜 냄새다. 이런 코드는 적절한 리팩토링 방법을 적용해 나쁜 냄새를 제거해줘야 한다.
>
> 리팩토링은 개발자가 직관적으로 수행할 수 있긴 하지만, 본격적으로 적용하자면 학습과 훈련이 필요하 다. 나쁜 냄새에는 어떤 종류가 있고, 그에 따른 적절한 리팩토링 방법은 무엇인지 알아보고, 충분한 연습 을 해두면 도움이 된다. 리팩토링을 공부할 때는 리팩토링에 관해 체계적으로 잘 정리한 책인 『리팩토링』 (마틴 파울러, 켄트 벡 공저)을 추천한다.

- https://martinfowler.com/books/refactoring.html
- https://refactoring.guru/ko/design-patterns

상속을 통한 확장

Payment의 변경 없이 환율 정보를 가져오는 방법을 확장하게 만들려면 상속을 이용할 수 있다.

어떻게 Payment를 준비할 것이가와 어떻게 환율을 가져올 것인가라는 관심사가 클래스 레벨에서 분리된다.

환율 정보를 담은 오브젝트인 ExRate를 생성하는 책임을 서브 클래스에게 위임하는 방식이다.

객체지향 디자인 패턴에 나오는 팩토리 메소드 패턴을 이용한다.

> 💡 **디자인 패턴**
>
> 디자인 패턴은 소프트웨어 설계 시 특정 상황에서 자주 만나는 문제를 해결하기 위해 사용할 수 있 는 재사용 가능한 솔루션을 말한다. 모든 패턴에는 간결한 이름이 있어서 잘 알려진 패턴을 적용하고 자 할 때 간단히 패턴 이름을 언급하는 것만으로도 설계의 의도와 해결책을 함께 설명할 수 있다는 장 점이 있다. 디자인 패턴은 주로 객체지향 설계에 관한 것이고, 대부분 객체지향적 설계 원칙을 이용 해 문제를 해결한다. 패턴의 설계 구조를 보면 대부분 비슷한데, 그 이유는 객체지향적인 설계로부터 문제를 해결하기 위해 적용할 수 있는 확장성 추구 방법이 대부분 두 가지 구조로 정리되기 때문이다. 하나는 클래스 상속이고 다른 하나는 오브젝트 합성이다. 따라서 패턴의 결과로 나온 코드나 설계 구 조만 보면 대부분 비슷해 보인다. 패턴에서 가장 중요한 것은 각 패턴의 핵심이 담긴 목적 또는 의도 다. 패턴을 적용할 상황, 해결해야 할 문제, 솔루션의 구조와 각 요소의 역할과 함께 핵심 의도가 무엇 인지를 기억해둬야 한다.
>
> 디자인 패턴은 객체지향 언어인 자바를 사용하는 개발자라면 반드시 공부해야 할 주제다. 디자 인 패턴을 최초로 집대성한 책인 『GoF의 디자인 패턴』(에릭 감마 외) 또는 『Head First Design Patterns』(에릭 프리먼)을 추천한다. GoF의 책은 딱딱하고 어렵게 느껴질 수도 있지만, 디자인 패턴 의 가장 중요한 핵심을 잘 설명하는 최고의 책이다. 다른 책을 먼저 공부하더라도 언젠가 한 번은 꼭 읽어보기 바란다.

- https://refactoring.guru/ko/design-patterns

> 💡 **팩토리 메소드 패턴**
>
> 팩토리 메소드 패턴은 상속을 통해 기능을 확장하게 하는 패턴이 다. 슈퍼클래스 코드에서는 서브클래스에서 구현할 메소드를 호출해서 필요 한 타입의 오브젝트를 가져와 사용한다. 이 메소드는 주로 인터페이스 타입으로 오브젝트를 리턴하므 로 서브클래스에서 정확히 어떤 클래스의 오브젝트를 만들어 리턴할지는 슈퍼클래스에서는 알지 못한다. 사실 관심도 없다. 서브클래스는 다양한 방법으로 오브젝트를 생성하는 메소드를 재정의할 수 있다. 이렇게 서브클래스에서 오브젝트 생성 방법과 클래스를 결정할 수 있도록 미리 정의해둔 메소 드를 팩토리 메소드라고 하고, 이 방식을 통해 오브젝트 생성 방법을 나머지 로직, 즉 슈퍼클래스의 기본 코드에서 독립시키는 방법을 팩토리 메소드 패턴이라고 한다.
>
> 자바에서는 종종 오브젝트를 생성하는 기능을 가진 메소드를 일반적으 로 팩토리 메소드라고 부르기도 한다. 이때 말하는 팩토리 메소드와 팩토리 메소드 패턴의 팩토리 메 소드는 의미가 다르므로 혼동하지 않도록 주의해야 한다

- https://refactoring.guru/design-patterns/factory-method

상속을 통해서 관심사가 다른 코드를 분리해내고, 서로 독립적으로 변경 또는 확장할 수 있도록 만드는 것은 간단하면서도 매우 효과적인 방법이다.

하지만 이 방법은 상속을 사용했다는 것이 단점이다. 자바는 다중 상속을 허용하지 않는다 따라서 또 다른 관심사를 분리할 경우 확장을 이용하기가 매우 어렵다. 또, 상속은 상하위 클래스의 관계매 매우 밀접하다. 상위 클래스의 변경에 따라 하위 클래스를 모두 변경해야 하는 상황이 생길 수도 있다. 디자인 패턴에 일부 적용된 특별한 목적을 위해서 사용되는 경우가 아니라면 상속을 통한 확장은 관심사의 분리에 사용하기엔 분명한 한계가 있다.

클래스의 분리

관심사에 따라 클래스를 분리해서 각각 독립적으로 구성할 수 있다.

결과적으로 클래스 레벨에 사용 의존관계가 만들어지기 때문에 강한 코드 수준의 결합이 생긴다. 실제로 사용할 클래스가 변경되면 이를 이용하는 쪽의 코드도 따라서 변경이 되어야 한다. 상속을 통한 확장 처럼 유연한 변경도 불가능해진다.

상속한 것이 아니기 때문에 사용하는 클래스의 메소드 이름과 구조도 제각각일 수 있다. 그래서 클래스가 변경되면 많은 코드가 따라서 변경되어야 한다.

클래스가 다르다는 것을 제외하면 관심사의 분리가 잘 된 방법이 아니다.

인터페이스 도입

독립적인 인터페이스를 정의하고 PaymentService가 사용할 메소드 이름을 정해둔다. 이를 각 클래스가 구현하게 만들면 이를 사용하는 쪽에서 의존하는 클래스가 변경되더라도 사용하는 메소드 이름의 변경이 일어나지 않는다.

하지만, 클래스의 인스턴스를 만드는 생성자를 호출하는 코드에는 클래스 이름이 등장하기 때문에 사용하는 환율 정보를 가져오는 클래스가 변경되면 PaymentService의 코드도 일부분이지만 따라서 변경되어야 한다.

여전히 상속을 통한 확장만큼의 유연성도 가지지 못한다.

관계설정 책임의 분리

앞에서 ExRateProvider 인터페이스를 사용하도록 작성했지만 구현 클래스에 대한 정보를 가지고 있다면 PaymentService 여전히 ExRateProvider를 구현한 특정 클래스에 의존하는 코드가 된다.

자신이 어떤 클래스의 오브젝트를 사용할지를 결정한다면 관계설정 책임을 직접 가지고 있는 것이다.

이 관계설정 책임을 자신을 호출하는 앞의 오브젝트에게 넘길 수 있다. 이렇게 되면 코드 레벨의 의존관계에서 자유로워진다. 이후에는 오직 인터페이스에만 의존하는 코드가 되기 때문에 어떤 구현 클래스의 오브젝트를 사용하게 되더라도 PaymentService의 코드가 변경되지 않는다.

관계설정 책임을 가진 앞의 클래스(Client)는 생성자를 통해서 어떤 클래스의 오브젝트를 사용할지 결정한 것을 전달해주면 된다.

오브젝트 팩토리

Client는 클라이언트로서의 책임과 PaymentService와 ExRateProvider 오브젝트 사이의 관계설정 책임을 두 가지를 가지고 있다. 관심사의 분리가 필요하다.

클라이언트의 관계설정 책임을 가진 코드를 ObjectFactory라는 이름으로 분리한다. ObjectFactory는 사용할 클래스를 선정하고 오브젝트를 만들면서 의존관계가 있다면 이를 생성자에 전달해서 만드는 기능을 담당한다.

원칙과 패턴

객체지향 설계원칙과 객체지향 디자인 패턴

개방-폐쇄 원칙(Open-Closed Principle)

클래스나 모듈은 확장에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.

클래스가 기능을 확장할 때 클래스의 코드는 변경되지 않아야 한다.

- https://blog.cleancoder.com/uncle-bob/2014/05/12/TheOpenClosedPrinciple.html

높은 응집도와 낮은 결합도(High Coherence and low coupling)

응집도가 높다는 것은 하나의 모듈이 하나의 책임 또는 관심사에 집중되어 있다는 뜻. 변화가 일어날 때 해당 모듈에서 변하는 부분이 크다.

책임과 관심사가 다른 모듈과는 낮은 결합도, 즉 느슨하게 연결된 형태를 유지하는 것이 바람직하다.

전략 패턴(Strategy Pattern)

자신의 기능 맥락(Context)에서, 필요에 따라서 변경이 필요한 알고리즘을 인터페이스를 통해 통째로 외부로 분리시키고, 이를 구현한 구체적인 알고리즘 클래스를 필요에 따라서 바꿔서 사용할 수 있게 하는 디자인 패턴.

Collections.sort()는 정렬에 사용할 전략 오브제트를 전달 받아서 사용한다.

- https://refactoring.guru/ko/design-patterns/strategy

제어의 역전(Inversion of Control)

제어권 이전을 통한 제어관계 역전 - 프레임워크의 기본 동작 원리

- https://www.martinfowler.com/bliki/InversionOfControl.html
- https://docs.spring.io/spring-framework/reference/core/beans/introduction.html

스프링 컨테이너와 의존관계 주입(Dependency Injection)

- https://docs.spring.io/spring-framework/reference/core/beans/introduction.html

스프링의 BeanFactory가 앞에서 만든 ObjectFactory가 제공하던 기능을 대체한다. BeanFactory는 ObjectFactory의 구성 정보를 참고해서 동작하게 만든다.

☐ BeanFactory

- https://docs.spring.io/spring-framework/reference/core/beans/beanfactory.html#page-title

스프링 컨테이너는 빈(bean)이라고 불리는 애플리케이션을 구성하는 오브젝트를 관리하는 기능을 담당한다.

☐ Bean

- https://docs.spring.io/spring-framework/reference/core/beans/definition.html

의존관계 주입(Dependency Injection)

IoC는 스프링의 동작원리를 정확하게 설명하기에는 너무 일반적인 프레임워크 동작원리를 설명하는 용어이다.

그래서 스프링과 같이 오브젝트의 의존관계에 대한 책임을 스프링과 같은 외부 오브젝트가 담당하도록 만드는 것을 설명하는, 의존관계 주입(Dependency Injection) 패턴, 줄여서 DI라고 불리는 용어가 마틴 파울러에 의해서 제안되었고 스프링 개발자들 사이에서, 또 이 원칙을 따라서 프레임워크를 만들거나 개발 방식을 셜명하는 다른 언어와 기술에서도 넓게 사용되고 있다.

스프링이 처음 등장했던 시기에는 IoC라는 용어를 주로 사용했기 때문에 이후에 DI를 사용하면서도 IoC라는 용어도 같이 쓰이기도 한다. 스프링은 IoC/DI 컨테이너라는 식으로 설명하는 문서도 많이 있다.

☐ Dependency Injection

- https://martinfowler.com/articles/injection.html

컨테이너

애플리케이션을 구성하는 오브젝트를 만들어서 담아두고 필요할 때 사용하도록 제공하는 기능을 담당하는 것을 컨테이너라고 부른다. 보통 오브젝트를 보관하는 것 뿐 아니라 생명주기(lifecycle)까지 담당한다. 스프링 컨테이너는 빈이라고 부르는 오브젝트를 생성하고 의존관계를 설정하는 것까지 담당한다.

스프링 컨테이너는 빈 오브젝트의 생명주기를 담당하는 기능도 제공한다.

구성정보를 가져오는 다 른 방법

@Configuration, @Bean 애노테이션이 붙은 구성정보 클래스와 메소드를 통해서 만들어질 오브젝트와 의존관계를 정의하는 코드는, 같은 구성정보를 제공 받을 수 있다면 다양한 다른 방법으로 정의할 수 있다.

예전에 많이 사용하던 XML을 이용하는 방법과 @Component 애노테이션이 붙은 클래스를 모두 찾아보는 빈 스캐닝 방식과 생성자 파라미터를 보고 의존 빈 오브젝트를 선택하는데 필요한 타입 정보를 가져오는 방식을 사용할 수도 있다. 빈 정보를 스캐닝에 의해서 동적으로 만들어내는 경우에는 @ComponentScan 애노테이션이 사용된다.

실제로는 빈 스캐닝 방식과 @Configuration/@Bean을 가진 구성정보 클래스 두 가지 방식을 혼합해서 사용한다.

싱글톤 레지스트리(Singleton Registry)

💡 싱글톤 패턴(Singleton Pattern)

싱글톤 패턴은 GoF가 소개한 디자인 패턴 중의 하나다. 디자인 패턴 중에서 가장 자주 활용되는 패턴이기도 하지만 가장 많은 비판을 받는 패턴이기도 하다. 심지어 디자인 책을 쓴 GoF 멤버조차도 싱글톤 패턴은 매우 조심해서 사용해야 하거나 피해야 할 패턴이라고 말하기도 한다.

싱글톤 패턴은 어떤 클래스를 애플리케이션 내에서 제한된 인스턴스 개수, 이름처럼 주로 하나만 존재하도록 강제하는 패턴이다. 이렇게 하나만 만들어지는 클래스의 오브젝트는 애플리케이션 내에서 전역적으로 접근이 가능하다. 단일 오브젝트만 존재해야 하고, 이를 애플리케이션의 여러 곳에서 공유하는 경우에 주로 사용한다.

싱글톤 패턴은 다음과 같은 단점을 가진다.

- private 생성자를 가지고 있기 때문에 상속할 수 없다
- 싱글톤은 테스트하기 힘들다
- 서버 환경에서는 싱글톤이 하나만 만들어지는 것을 보장하지 못한다
- 싱그톤의 사용은 전역 상태를 만들 수 있기 때문에 바람직하지 못하다

스프링은 하나의 오브젝트만 만들어져야 한다는 필요를 충족하면서도 싱글톤 패턴을 사용해서 만들었을 때의 단점을 가지지 않도록 컨테이너 레벨에서 하나의 오브젝트만 만들어지는 것을 보장하는 기능을 제공한다. 이렇게 싱글톤 오브젝트를 만들고 관리하는 스프링 컨테이너를 싱글톤을 등록하고 관리한다는 의미에서 싱글톤 레지스트리라고 부르기도 한다.

스프링의 빈이 생성되고 적용되는 범위를 빈의 스코프(scope)라고 부른다. 스프링은 기본적으로 빈 오브젝트가 싱글톤 스코프를 가지도록 한다. 필요에 따라 여러 개의 빈 오브젝트가 만들어지도록 할 수도 있다.

DI와 디자인 패턴

디자인 패턴을 구분하는 두 가지 방식이 있다.

하나는 사용 목적(purpose)이고 다른 하나는 스코프(scope)이다.

스코프에 의해서 분류하면 클래스 패턴과 오브젝트 패턴으로 나눌 수 있다.

클래스 패턴은 상속(inheritance)을 이용해서 확장성을 가진 패턴으로 만들어지고, 오브젝트 패턴은 합성(composition)을 이용한다. 대부분의 디자인 패턴은 오브젝트 패턴이다. 가능하면 오브젝트 합성을 상속보다 더 선호하라는 디자인 패턴의 기본 객체지향 원리를 따른 것이다.

전략 패턴은 오브젝트 합성을 이용한다.

☐ 데코레이터 패턴(Decorator Pattern)

오브젝트에 부가적인 기능/책임을 동적으로 부여하는 디자인 패턴이다.

- https://refactoring.guru/design-patterns/decorator

의존성 역전 원칙(Dependency Inversion Principle)

1. 상위 수준의 모듈은 하위 수준의 모듈에 의존해서는 안 된다. 둘 모두 추상화에 의존해야 한다.
2. 추상화는 구체적인 사항에 의존해서는 안 된다. 구체적인 사항은 추상화에 의존해야 한다.

DIP는 먼저 인터페이스를 통해서 추상화에 의존하도록 코드를 만들어야 한다.

그리고 인터페이스 소유권의 역전도 필요하다.

☐ Separate Interface 패턴

인터페이스와 그 구현을 별개의 패키지에 위치시키는 패턴이다.

인터페이스를 이를 구현한 클래스와 같은 패키지가 아닌 이 인터페이스를 사용하는 클라이언트와 같은 패키지에 위치하게 한다. 만약 이를 사용하는 클래스가 여럿인 경우에는 별개의 패키지로 인터페이스 구분해둘 수 있다.

- https://martinfowler.com/eaaCatalog/separatedInterface.html

03 테스트

> 💡 테스트를 만들지 않을 거면 스프링을 도대체 뭐하러 쓰는 거죠?

자동으로 수행되는 테스트(Automated Test)

수동으로 개발한 코드를 테스트하는 방법은 번거롭고 활용하는데 한계가 있다.

코드로 만들어져 언제든 실행해서 테스트할 수 있는 자동으로 수행되는 테스트가 필요하다. 이를 통해서 지속적인 개선과 점진적인 개발이 가능해진다.

개발자가 만드는 테스트

테스트를 코드로 만들고 자동으로 수행되는 테스트를 실행해서 작성한 코드에 대한 피드백을 받는다

테스트 작성과 실행이 개발을 하는 과정의 일부가 된다

테스팅 프레임워크를 이용해서 테스트 작성과 실행 과정을 효율적으로 진행할 수 있다

JUnit 테스트 작성

JUnit은 켄트 벡과 에릭 감마가 처음 개발한 가장 대표적인 자동화된 테스트 수행 도구이다. 자바 외의 다른 언어로도 유사하게 개발되어지면서 이를 통틀어 xUnit이라고 부르기도 한다.

☐ JUnit 5

- https://junit.org/junit5/
- https://junit.org/junit5/docs/current/user-guide/

최신 JUnit 버전이며 현재 스프링 프레임워크 자체의 테스트에 사용되고, 스프링을 이용해서 개발하는 프로젝트에서도 가장 많이 사용된다.

PaymentService 테스트

자동화된 테스트는 언제든 실행할 수 있고 항상 동일한 테스트 결과를 얻어야 한다.

때로는 외부 시스템에 대한 테스트, 현재 시간과 같이 코드에서 쉽게 제어할 수 없는 값을 이용하는 테스트를 작성해야 하는데 이런 경우에 일관된 결과를 보장하는 테스트 코드를 작성하기가 쉽지 않다.

☐ System Under Test(SUT)

- http://xunitpatterns.com/SUT.html

테스트와 DI

테스트 대역(Test Double, Imposter), 스텁(Stub), 목(Mock)

- https://martinfowler.com/bliki/TestDouble.html
- https://martinfowler.com/articles/mocksArentStubs.html

스프링 DI를 이용하는 테스트

스프링 컨테이너를 구성하고 여기서 테스트에 필요한 대상과 의존 오브젝트를 설정하거나 테스트에서 참고할 빈 오브젝트를 가져오게 할 수 있다.

스프링은 방대한 양의 테스팅 지원 기술을 제공한다.

- https://docs.spring.io/spring-framework/reference/testing.html

JUnit에서 스프링 컨테이너를 만들어 테스트를 수행할 때 @ExtendWith과 @ContextConfiguration을 이용한다

☐ @ExtendedWith

JUnit 5 테스트 클래스가 스프링 테스팅 기능을 사용하도록 지정한다

아래 @ContextConfiguration과 결합된 합성 애노테이션인 @SpringJUnitConfig을 이용할 수도 있다.

- https://docs.spring.io/spring-framework/reference/testing/annotations/integration-junit-jupiter.html#integration-testing-annotations-junit-jupiter-springjunitconfig

☐ @ContextConfiguration

- https://docs.spring.io/spring-framework/reference/testing/annotations/integration-spring/annotation-contextconfiguration.html#page-title

☐ Autowired

테스트 코드에 @Autowired가 붙은 인스턴스 변수를 선언하며 스프링 테스트에 의해서 인스턴스 변수의 타입과 일치하는 스프링 컨테이너의 빈 오브젝트를 주입해준다. @Autowired 외에도 스프링에서 지원하는 여러가지 종류의 애노테이션을 지원한다.

- https://docs.spring.io/spring-framework/reference/testing/annotations/integration-standard.html#page-title

학습 테스트(Learning Test)

켄트 벡의 테스트주도개발이라는 책에서 소개된 테스트 방법의 한 가지이다. 학습 테스트는 내가 만들지 않은 코드, 라이브러리, API 등에 대한 테스트이다. 학습 테스트의 목적은 사용할 API나 프레임워크의 기능을 테스트로 작성하고 실행해보면서 사용방법을 바르게 이해했는지 확인할 수 있다.

학습 테스트는 켄트 벡의 테스트 주도 개발과 로버트 마틴의 클린 코드에서 소개되었다.

04 템플릿(Template)

템플릿은 고정된 작업 흐름 안에 변경할 수 있는 코드를 콜백 형태로 전달해서 사용할 수 있도록 만들어진 오브젝트를 말한다.

템플릿 메소드 패턴은 상속을 이용해서 고정된 템플릿과 변경 가능한 훅 메소드 등으로 분리하는 패턴이다.

템플릿과 콜백을 사용하는 방식은 프레임워크에 적용되는 제어의 역전으로 설명할 수 있다.

컨텍스트에 전략을 교체해서 쓰는 전략 패턴의 특별한 케이스로 볼 수도 있다.

콜백은 인터페이스로 만들어지고 대부분 하나의 메소드만 가진다.

콜백을 이용해서 확장성을 추구하는 방식은 스프링을 만든 로드 존슨이 쓴 J2EE Design And Development에서 처음 소개되었다. 책에서 예로든 템플릿은 JDBC를 다루는 코드에 적용하도록 만들어진 JdbcTemplate이고, 후에 스프링 프레임워크가 만들어질 때 포함되었다.

☐ 메소드 호출 주입(method call injection)

DI의 일종으로 컨테이너의 구성 정보에 포함되지 않고 메소드 실행 시점에 의존 오브젝트를 파라미터로 주입하는 방식으로 동작한다.

스프링의 룩업 메소드 주입(lookup method injection)과는 다르다. 스프링의 메소드 주입은 런타임 상속을 통해서 메소드의 구현 코드를 직접 주입하는 방식이다.

스프링이 제공하는 템플릿

☐ JdbcTemplate

SQL 쿼리를 수행하거나 등록, 수정, 프로시저 호출을 할 때 사용할 수 있는 템플릿이다.

- https://docs.spring.io/spring-framework/reference/data-access/jdbc/core.html#jdbc-JdbcTemplate

스프링 6에는 JdbcTemplate을 좀 더 모던하게 만든 JdbcClient가 추가되었다.

- https://docs.spring.io/spring-framework/reference/data-access/jdbc/core.html#jdbc-JdbcClient

JdbcTemplate에서 사용하는 RowMapper와 같은 콜백을 사용할 수 있다.

☐ RestTemplate

스프링이 제공하는 가장 오래된 동기 방식의 REST 클라이언트 기술의 하나이다.

- https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-resttemplate

GET과 POST 메소드를 사용하는 간단한 HTTP API를 호출할 때 사용하기에 편리하다. 다양한 HTTP API 기술을 이용하도록 만들 수 있다.

최근에 스프링에 추가된 RestClient을 이용하면 모던한 API 스타일로 된 HTTP API를 호출하는 코드를 만들 수 있다. 여러 가지 콜백 오브젝트를 지원한다.

☐ TransactionTemplate

스프링의 트랜잭션 추상화 기술과 함께 사용 가능한 데이터 트랜잭션 작업용 템플릿이다.

@Transactional이 제공하는 트랜잭션 경계설정 기능을 TransactionTemplate으로도 모두 적용할 수 있다.

JDBC, JPA, MyBatis, Hibernate 등의 다양한 데이터 기술에 모두 사용이 가능하다.

- https://docs.spring.io/spring-framework/reference/data-access/transaction/programmatic.html#tx-prog-template

05 예외(Exception)

예외는 정상적인 흐름의 진행을 어렵게 만드는 이벤트를 말한다.

- https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

1. Error
    a. 시스템에 비정상적인 상황이 발생했을 때 사용되고 주로 JVM에서 발생시킨다
    b. 일반적으로 애플리케이션에서 복구를 기대할 수 없는 종류의 예외의 슈퍼 클래스이다
2. Exception과 체크 예외
    a. Exception은 애플리케이션이 복구를 기대할 수도 있는 모든 예외의 슈퍼 클래스이다
    b. 언체크 예외/런타임 예외가 아닌 애플리케이션에서 발생되는 예외는 체크 예외로 분류된다
    c. 체크 예외는 catch를 사용해서 잡아내든가, 메소드에서 throws를 정의해서 메소드 밖으로 던져야 한다
3. RuntimeException과 언체크 예외
    a. 명시적인 예외처리가 강제되지 않는다
    b. catch/throws를 쓰지 않아도 문제 없이 컴파일된다

JPA를 이용한 Order 저장

자바에서 RDB를 사용하는 모든 기술(JDBC, JPA, MyBatis, ...)은 DB와의 연결 정보를 관리하는 DataSource를 사용한다.

- https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html

☐ EmbeddedDatabaseBuilder

애플리케이션 내장형 DB(H2, HSQL, Derby)를 빠르게 셋업하고 DataSource를 생성하는데 사용하는 빌더이다.

- https://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html

☐ LocalContainerEntityManagerFactoryBean

EntityManagerFactory를 XML 없이 스프링의 빈 설정용 팩토리 메소드에서 간단히 생성하는데 사용된다.

- https://docs.spring.io/spring-framework/reference/data-access/orm/jpa.html#orm-jpa-setup-background

JPA의 EntityManager는 EntityManagerFactory가 있어야 만들 수 있다.

EntityManager와 트랜잭션

코드에 의해서 엔티티 매니저를 생성하고 트랜잭션을 가져와 JPA의 기능을 사용하는 기본 코드는 다음과 같은 구조를 가진다.

- https://docs.jboss.org/hibernate/stable/entitymanager/reference/en/html/transactions.html#transactions-demarcation

```
// Non-managed environment idiom
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if ( tx != null && tx.isActive() ) tx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

Order 리포지토리와 예외

JDBC를 수행중에 발생하는 예외는 SQLException으로 만들어져서 던져진다

☐ SQLExceptiopn

- https://docs.oracle.com/javase/tutorial/jdbc/basics/sqlexception.html

JPA를 이용하는 코드에서 예외가 발생하면 주로 JDBC의 SQLException을 랩핑해서 JPA의 표준 예외로 만들어서 던져진다.

☐ jakarta.persistence.PersistenceException

표준 JPA의 예외 안에 JPA의 구현 기술(Hibernate 등)의 예외가 다시 랩핑되어서 던져진다

DataAccessException 추상화

스프링의 데이터 예외 추상화가 적용되면 JDBC의 SQLException, JPA의 예외 등이 스프링이 정의한 예외 계층구조의 예외로 번역되고, 실제 발생한 예외는 래핑된 형태로 전달된다.

☐ DataAccessException

- https://docs.spring.io/spring-framework/reference/data-access/dao.html#dao-exceptions

06 서비스 추상화

자바 서버 기술(J2Ee/JavaEE)은 여러 계층으로 구분해서 개발하는 레이어 아키텍처를 이용한다.

- https://docs.oracle.com/cd/E19644-01/817-5448/dgdesign.html
- https://openclassrooms.com/en/courses/5684146-create-web-applications-efficiently-with-the-spring-boot-mvc-framework/6156961-organize-your-application-code-in-three-tier-architecture

보통 애플리케이션의 비즈니스 로직/도메인 로직의 코드가 위치하는 계층을 서비스 계층이라고 부른다. 그 외에도 다양한 이름으로 불린다.

☐ Service Layer

- https://martinfowler.com/eaaCatalog/serviceLayer.html

트랜잭션 서비스 추상화

스프링의 트랜잭션 관리 기술의 핵심은 트랜잭션 추상화이다.

- https://docs.spring.io/spring-framework/reference/data-access/transaction/strategies.html#page-title

데이터 액세스 기술에 상관없이 공통적으로 적용되는 트랜잭션 인터페이스인 PlatformTransactionManager이 제공된다.

```
public interface PlatformTransactionManager extends TransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws Transa

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

JDBC 데이터 액세스 기술

JdbcClient를 이용하면 SQL을 사용하는 간단한 코드로 안전하고, 트랜잭션 추상화가 적용된 DB 액세스 코드를 작성할 수 있다.

- https://docs.spring.io/spring-framework/reference/data-access/jdbc/core.html#jdbc-JdbcClient

JDBC를 직접 사용하는 코드를 이용할 때는 JdbcTransactionManager를 트랜잭션 매니저 오브젝트로 등록해서 사용한다.

트랜잭션 프록시

☐ Decorator Pattern

- https://refactoring.guru/design-patterns/decorator

☐ Proxy Pattern

- https://refactoring.guru/design-patterns/proxy

스프링의 트랜잭션 프록시는 ProxyFactoryBean과 AOP 프록시를 이용해서 만들어진다.

- https://docs.spring.io/spring-framework/reference/core/aop-api/pfb.html#aop-pfb-proxy-types