9장. 애플리케이션 조립하기

그냥 필요할 때 인스턴스화 하면 안될까?
    왜 조립까지 신경써야 하는가?
테스트하기 쉬워졌다 !
설정 컴포넌트
    역할
    단일 책임 원칙을 위반하는 게 아닐까?
조립하기
    1. 평범한 코드로 조립하기
        단점
    2. 스프링의 클래스패스 스캐닝으로 조립하기
        애플리케이션 컨텍스트
        클래스패스 스캐닝
    3. 스프링의 자바 컨피그로 조립하기
결론

그냥 필요할 때 인스턴스화 하면 안될까?

왜 조립까지 신경써야 하는가?

- 코드 의존성이 올바른 방향을 가리키게 하기 위해서
- 모든 의존성은 안쪽(도메인 코드 방향)으로 향해야 한다.
- 도메인 코드가 바깥 계층으로부터 안전하게끔 만들어야 한다.

테스트하기 쉬워졌다 !

클래스가 필요로 하는 모든 객체를 생성자로 전달한다.

실제 객체 대신 목으로 전달할 수 있다.

격리된 단위 테스트를 생성하기 쉽다.

설정 컴포넌트

아키텍처에 대해 중립적, 인스턴스 생성을 위한 모든 클래스에 대한 의존성을 가지는 컴포넌트

- 의존성 규칙을 어기지 않으면서 객체 인스턴스를 생성
- 원의 가장 바깥쪽에 위치

역할

- 웹 어댑터 인스턴스 생성
- HTTP 요청이 실제로 웹 어댑터로 전달되도록 보장
- 유스케이스 인스턴스 생성
- 웹 어댑터에 유스케이스 인스터스 제공
- 영속성 어댑터 인스턴스 생성
- 유스케이스에 영속성 어댑터 인스턴스 제공
- 영속성 어댑터가 실제로 데이터베이스에 접근할 수 있도록 보장
- 설정 파라미터의 소스 접근

단일 책임 원칙을 위반하는 게 아닐까?

맞다. 애플리케이션의 나머지 부분을 깔끔하게 유지하기 위해 어쩔 수 없다.

조립하기

1. 평범한 코드로 조립하기

```
package copyeditor.configuration;

class Application {
    public static void main(String[] args) {

        AccountRepository accountRepository = new AccountRepo
        ActivityRepository activityRepository = new ActivityR
```

```
        AccountPersistenceAdapter accountPersistenceAdapter =
            new AccountPersistenceAdapter(accountRepository,

        SendMoneyUseCase sendMoneyUseCase =
            new SendMoneyUseService(
                accountPersistenceAdapter,
                accountPersistenceAdapter);

        SendMoneyController sendMoneyController =
            new SendMoneyController(sendMoneyUseCase);

        startProcessingWebRequests(sendMoneyController);
    }
}
```

단점

- 완전한 엔터프라이즈 애플리케이션을 실행하려면 위의 코드를 아주 많이 만들어야 한다.
- 패키지 외부에서 인스턴스를 생성하기 때문에 클래스가 전부 public이어야 한다.

2. 스프링의 클래스패스 스캐닝으로 조립하기

애플리케이션 컨텍스트

스프링 프레임워크를 이용해서 애플리케이션을 조립한 결과물

애플리케이션을 구성하는 모든 객체(빈)를 포함

클래스패스 스캐닝

애플리케이션 컨텍스트를 조립하기 위한 방법 중 하나

동작 원리

- 클래스패스 스캐닝으로 클래스패스에서 접근 가능한 모든 클래스를 확인
- `@Component` 애너테이션이 붙은 클래스를 탐색
- 애너테이션이 붙은 각 클래스의 객체를 생성
    - 이 때 클래스는 모든 필드를 인자로 받는 생성자를 가지고 있어야 한다.

- 애플리케이션 컨텍스트에 인스턴스를 추가

단점

- 클래스에 프레임워크에 특화된 애너테이션을 붙여야 한다.
    - 강경 클린 아키텍처파는 이 방식이 특정 프레임워크와 결합되기 때문에 사용하지 말 것을 주장한다.
- 스프링 전문가가 아니라면 원인을 찾는 데 오래 걸리는 숨겨진 사이드 이펙트를 만들어 낼 수도 있다.

3. 스프링의 자바 컨피그로 조립하기

```
@Configuration
@EnableJpaRepositories
class PersistenceAdapterConfiguration {

    @Bean
    AccountPersistenceAdapter accountPersistenceAdapter(
        AccountRepository accountRepository,
        ActivityRepository activityRepository,
        AccountMapper accountMapper) {

        return new AccountPersistenceAdapter(
            accountRepository,
            activityRepository,
            accountMapper);
    }

    @Bean
    AccountMapper accountMapper() {
        return new AccountMapper();
    }
}
```

- `@Configuration` 애너테이션을 통해 스프링의 클래스패스 스캐닝에서 발견해야 할 설정 클래스임을 표시해둔다.
- `@Bean` 애너테이션이 붙은 팩터리 메서드를 통해 빈이 생성된다.

- 팩터리 메서드에 대한 입력으로 수동으로 생성된 객체들을 스프링이 자동으로 제공한다.

단점

- 설정 클래스가 생성하는 빈이 설정 클래스와 같은 패키지에 존재하지 않으면 빈을 public으로 만들어야 한다.

결론

- 클래스패스 스캐닝은 아주 편리하다.
  - 스프링에게 패키지만 알려주면 클래스를 찾아 애플리케이션을 조립한다.
  - 코드의 규모가 커지면 투명성이 낮아지고 테스트에서 애플리케이션 컨텍스트의 일부만 띄우기 어렵다.
- 애플리케이션 조립을 책임지는 전용 설정 컴포넌트를 만들 수도 있다.
  - 서로 다른 모듈로부터 독립되어 코드 상에서 쉽게 옮겨 다닐 수 있다. (응집도 👍🏻)
  - 유지보수가 오래 걸린다.