



잘못된 테스트 사례 학습

Java/Spring 테스트를 추가하고 싶은 개발자들의 오답노트

왜 내가 하는 TDD는 실패하는가?

개요

의존성이 무엇인지 간략히 알아보고, Testability의 개념에 대해 알아봅시다.

제 1장 ━━━━ 의존성

제 2장 ━━━━ 의존성과 테스트

제 3장 ━━━━ Testability

1. 의존성

의존성

의존성에 대해 알아봅니다.

의존성 역전

의존성 역전 원칙에 대해 알아봅니다.

1. 의존성

의존성이란 무엇인가?

“ Dependency (computer science) or coupling, a state in which one object uses a function of another object

Wikipedia contributors, "Dependency," Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/w/index.php?title=Dependency&oldid=1090325642> (accessed January 23, 2023).

 A는 B를 사용하기만 해도 A는 B에 의존한다 할 수 있다.

```
class Chef {  
    public Hamburger makeHamburger() {  
        Bread bread = new Bread();  
        Meat meat = new Meat();  
        Lettuce lettuce = new Lettuce();  
        Source source = new Source();  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```

1. 의존성

의존성이란 무엇인가?

“ Dependency (computer science) or coupling, a state in which one object uses a function of another object

Wikipedia contributors, "Dependency," Wikipedia, The Free Encyclopedia, <https://en.wikipedia.org/w/index.php?title=Dependency&oldid=1090325642> (accessed January 23, 2023).

 A는 B를 사용하기만 해도 A는 B에 의존한다 할 수 있다.

```
class Chef {  
    public Hamburger makeHamburger() {  
        Bread bread = new Bread();  
        Meat meat = new Meat();  
        Lettuce lettuce = new Lettuce();  
        Source source = new Source();  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```

1. 의존성

의존성 주입

“ Depedency Injection 의존성을 약화시키는 테크닉

```
class Chef {  
    public Hamburger makeHamburger() {  
        Bread bread = new Bread();  
        Meat meat = new Meat();  
        Lettuce lettuce = new Lettuce();  
        Source source = new Source();  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```

```
class Chef {  
    public Hamburger makeHamburger(  
        Bread bread,  
        Meat meat,  
        Lettuce lettuce,  
        Source source) {  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```

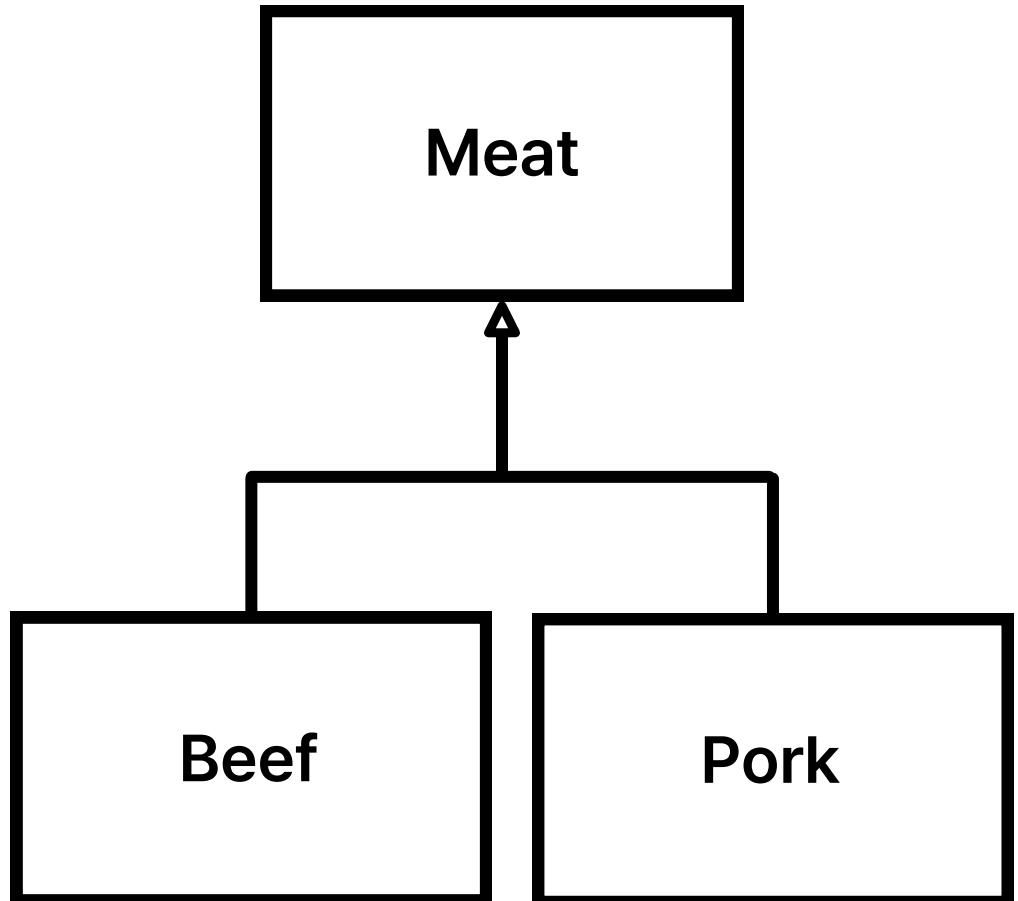
1. 의존성

의존성 주입

“ 인스턴스를 만드는 것보다 의존성 주입을 받는게 좋은 이유
new는 사실상 하드 코딩이다

```
class Chef {  
  
    public Hamburger makeHamburger() {  
        Bread bread = new Bread();  
        Meat meat = new Meat();  
        Lettuce lettuce = new Lettuce();  
        Source source = new Source();  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```

```
class Chef {  
  
    public Hamburger makeHamburger(  
        Bread bread,  
        Meat meat,  
        Lettuce lettuce,  
        Source source) {  
        return Hamburger.builder()  
            .bread(bread)  
            .meat(meat)  
            .lettuce(lettuce)  
            .source(source)  
            .build();  
    }  
}
```



1. 의존성

의존성의 종류

“

Types of coupling [\[edit\]](#)

Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Some types of coupling, in order of highest to lowest coupling, are as follows:

Procedural programming [\[edit\]](#)

A module here refers to a subroutine of any kind, i.e. a set of one or more statements having a name and preferably its own set of variable names.

Content coupling (high)

Content coupling is said to occur when one module uses the code of another module, for instance a branch. This violates [information hiding](#) - a basic software design concept.

Common coupling

Common coupling is said to occur when several modules have access to the same global data. But it can lead to uncontrolled error propagation and unforeseen side-effects when changes are made.

External coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

Control coupling

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp coupling (data-structured coupling)

Stamp coupling occurs when modules share a composite data structure and use only parts of it, possibly different parts (e.g., passing a whole record to a function that needs only one field of it).

In this situation, a modification in a field that a module does not need may lead to changing the way the module reads the record.

Data coupling

Data coupling occurs when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Object-oriented programming [\[edit\]](#)

Subclass coupling

Describes the relationship between a child and its parent. The child is connected to its parent, but the parent is not connected to the child.

Temporal coupling

It is when two actions are bundled together into one module just because they happen to occur at the same time.

In recent work various other coupling concepts have been investigated and used as indicators for different modularization principles used in practice.^[5]

Dynamic coupling [\[edit\]](#)

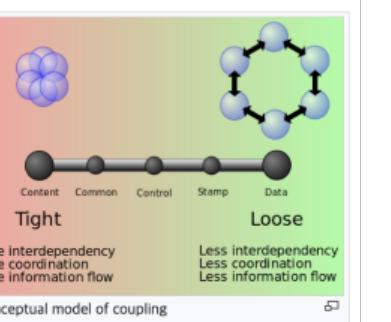
The goal of this type of coupling is to provide a run-time evaluation of a software system. It has been argued that static coupling metrics lose precision when dealing with an intensive use of dynamic binding or inheritance.^[6] In the attempt to solve this issue, dynamic coupling measures have been taken into account.

Semantic coupling [\[edit\]](#)

This kind of coupling considers the conceptual similarities between software entities using, for example, comments and identifiers and relying on techniques such as [latent semantic indexing](#) (LSI).

Logical coupling [\[edit\]](#)

Logical coupling (or evolutionary coupling or change coupling) exploits the release history of a software system to find change patterns among modules or classes: e.g., entities that are likely to be changed together or sequences of changes (a change in a class A is always followed by a change in a class B).



1. 의존성

의존성 역전

“ Dependency Injection과 Dependency Inversion은 다릅니다.

 Dependency Injection은 의존성 주입 (DI)

Dependency Inversion 은 의존성 역전 (SOLID-DIP)

1. 의존성

의존성 역전이란?

- “ 첫째, 상위 모듈은 하위 모듈에 의존해서는 안된다. 상위 모듈과 하위 모듈 모두 추상화에 의존해야 한다.
- 둘째, 추상화는 세부 사항에 의존해서는 안된다. 세부사항이 추상화에 의존해야 한다.

위키백과 기여자, "의존관계 역전 원칙," 위키백과, , https://ko.wikipedia.org/w/index.php?title=의존관계_역전_원칙&oldid=31527077 (2023년 1월 23일에 접근).

그림 a

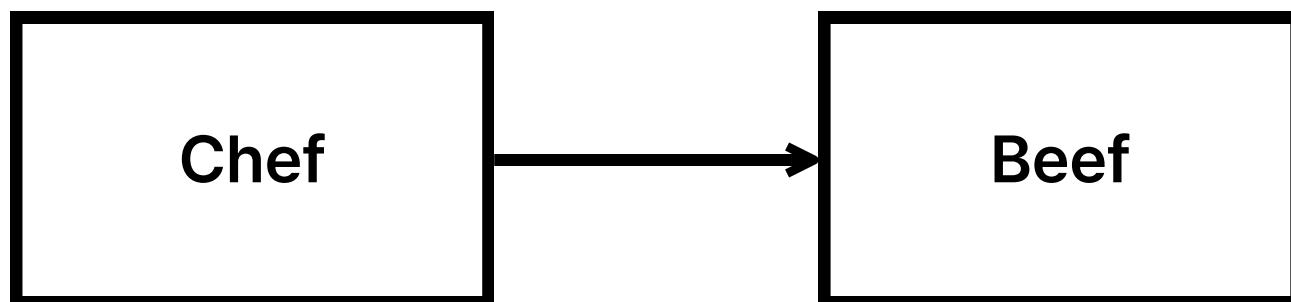
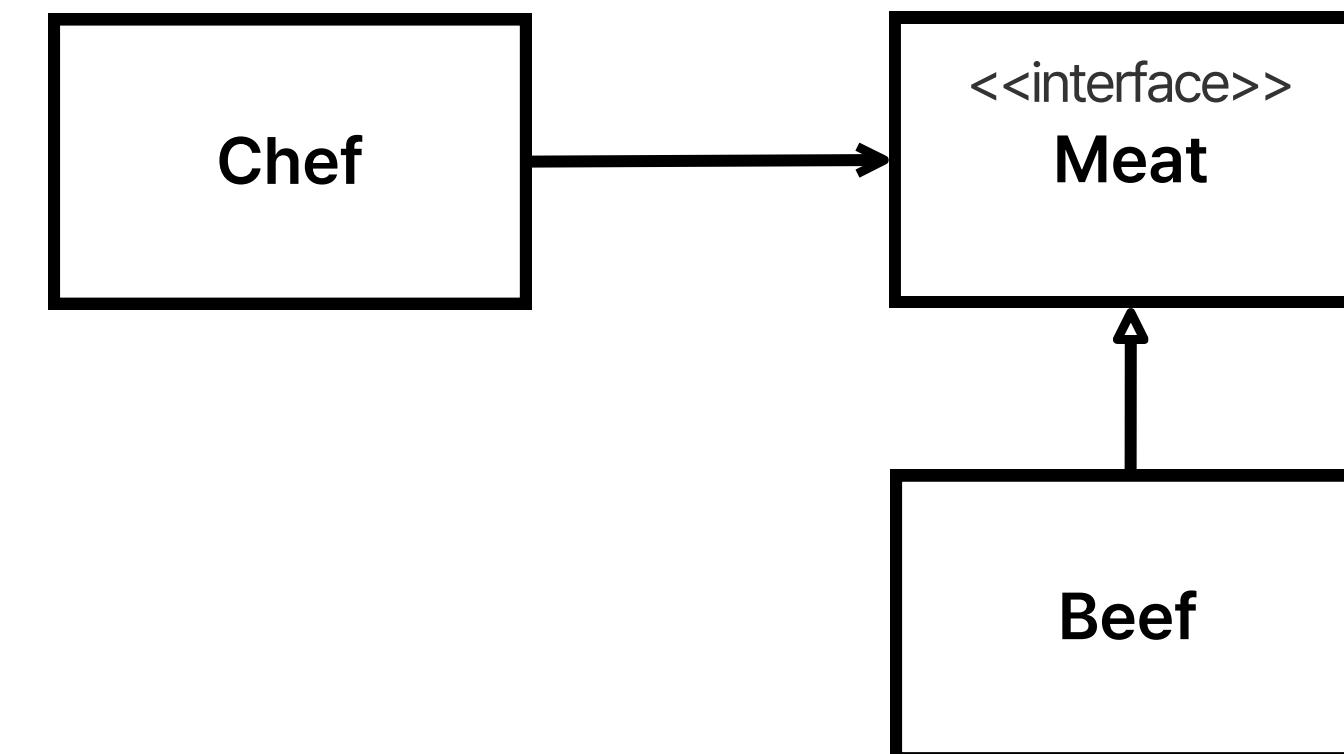


그림 b



1. 의존성

의존성 역전이란?

“ 고수준 정책을 구현하는 코드는 저수준 세부사항을 구현하는 코드에 절대로 의존해서는 안 된다. 대신 세부사항이 정책에 의존해야 한다.

로버트 C. 마틴 저, 클린 아키텍처 소프트웨어 구조와 설계의 원칙, 송준이 옮김, (인사이트(insight), 2019-08-20), 65p

그림 a

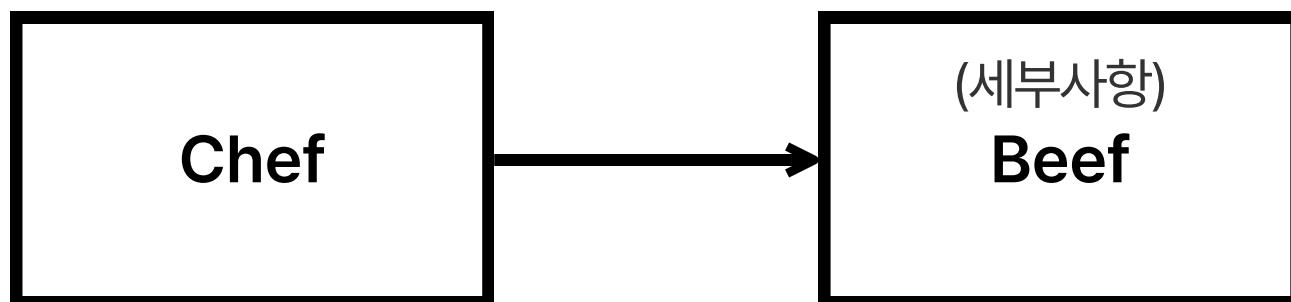
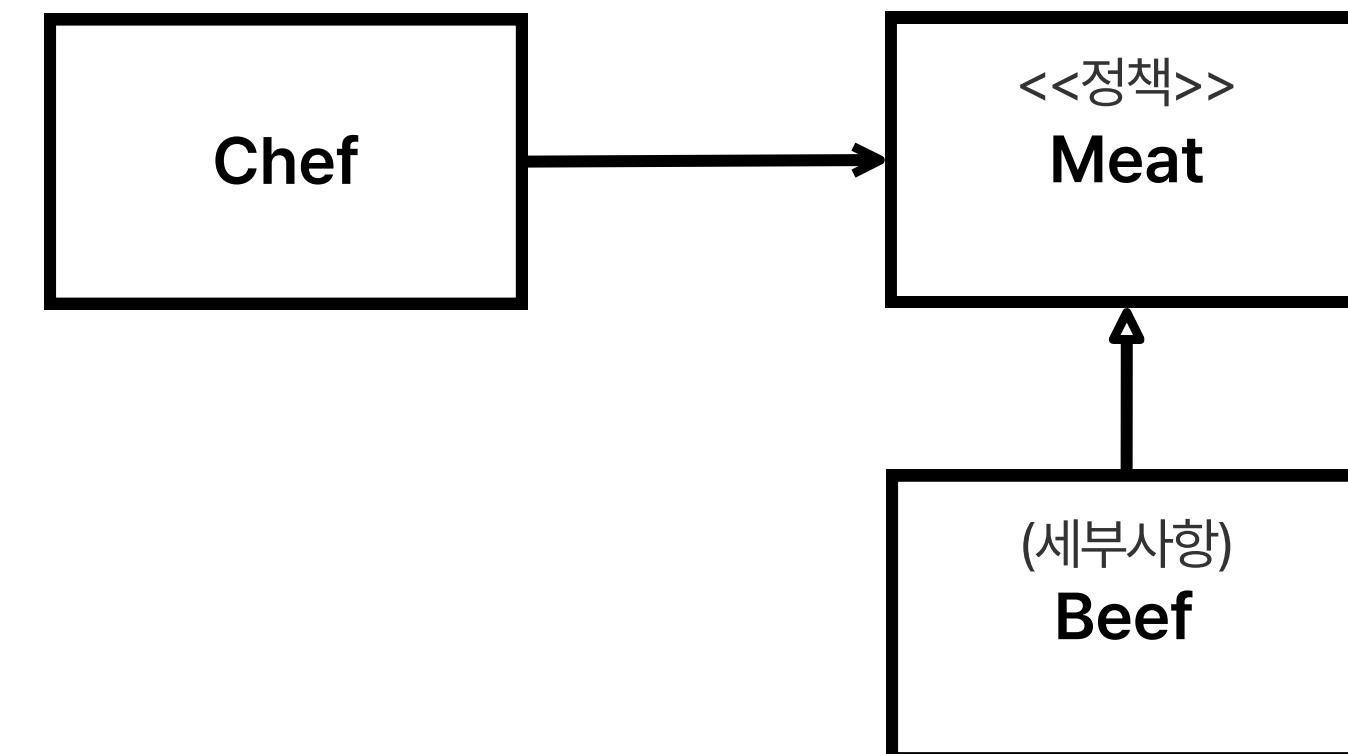


그림 b

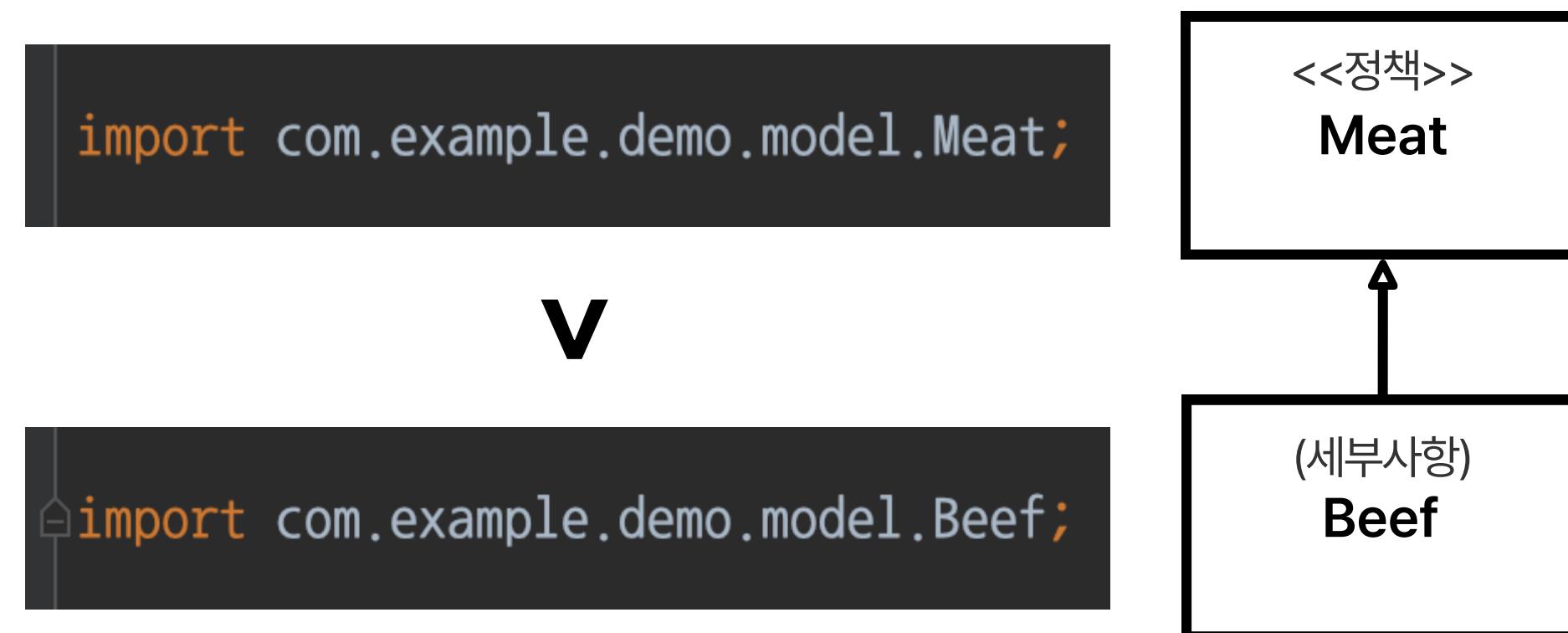


1. 의존성

의존성 역전이란?

- “ 자바와 같은 정적 타입 언어에서 이 말은 use, import, include 구문은 오직 인터페이스나 추상 클래스 같은 추상적인 선언만을 참조해야 한다는 뜻이다.
(중략)
우리가 의존하지 않도록 피하고자 하는 것은 바로 변동성이 큰 구체적인 요소다.

로버트 C. 마틴 저, 클린 아키텍처 소프트웨어 구조와 설계의 원칙, 송준이 옮김, (인사이트(insight), 2019-08-20), 92p

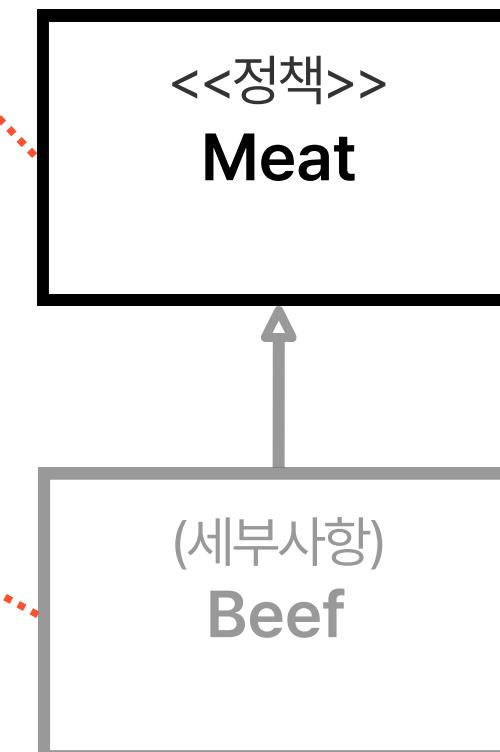


1. 의존성

의존성 역전이란?

“ 자바와 같은 정적 타입 언어에서 이 말은 use, import, include 구문은 오직 인터페이스나 추상 클래스 같은 추상적인 선언만을 참조해야 한다는 뜻이다.
(중략)
우리가 의존하지 않도록 피하고자 하는 것은 바로 변동성이 큰 구체적인 요소다.”

로버트 C. 마틴 저, 클린 아키텍처 소프트웨어 구조와 설계의 원칙, 송준이 옮김, (인사이트(insight), 2019-08-20), 92p



2. 의존성과 테스트

의존성과 테스트

의존성과 테스트의 상관 관계에 대해 알아봅니다

2. 의존성과 테스트

갑자기 의존성은 왜?

“ 테스트를 잘 하려면 의존성 주입과 의존성 역전을 잘 다룰 수 있어야 합니다.

예시 | 마지막 로그인 시간

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login() {  
        // ...  
        this.lastLoginTimestamp = Clock.systemUTC().millis();  
    }  
}
```

내부 로직을 보면 login 은 분명 Clock 에 의존적입니다

```
user.login();
```

그렇지만 외부에서 보면 login이 시간에 의존하고 있는지를 알 수 없습니다



2. 의존성과 테스트

갑자기 의존성은 왜?

“ 테스트를 잘 하려면 의존성 주입과 의존성 역전을 잘 다룰 수 있어야 합니다.

예시 | 마지막 로그인 시간

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login() {  
        // ...  
        this.lastLoginTimestamp = Clock.systemUTC().millis();  
    }  
}
```

```
class UserTest {  
  
    @Test  
    public void login_테스트() {  
        // given  
        User user = new User();  
  
        // when  
        user.login();  
  
        // then  
        assertThat(user.getLastLoginTimestamp()).isEqualTo(???);  
    }  
}
```



대체 어떻게 테스트하죠...?

2. 의존성과 테스트

갑자기 의존성은 왜?

- “ 1. 시간을 의존성 주입으로 해결

예시 | 마지막 로그인 시간

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login(Clock clock) {  
        // ...  
        this.lastLoginTimestamp = clock.millis();  
    }  
}
```

```
class UserTest {  
  
    @Test  
    public void login_테스트() {  
        // given  
        User user = new User();  
        Clock clock = Clock.fixed(Instant.parse("2000-01-01T00:00:00.00Z"), ZonedDateTime.ofInstant(Instant.parse("2000-01-01T00:00:00.00Z"), ZoneId.of("UTC")));  
  
        // when  
        user.login(clock);  
  
        // then  
        assertThat(user.getLastLoginTimestamp()).isEqualTo(946684800000L);  
    }  
}
```



의존성 주입을 사용하면 이 문제를 해결할 수 있습니다!

2. 의존성과 테스트

갑자기 의존성은 왜?

- “ 1. 시간을 의존성 주입으로 해결 (?)

예시 | 마지막 로그인 시간

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login(Clock clock) {  
        // ...  
        this.lastLoginTimestamp = clock.millis();  
    }  
}  
  
class UserService {  
    public void login(User user) {  
        // ...  
        user.login(Clock.systemUTC());  
    }  
}
```

2. 의존성과 테스트

갑자기 의존성은 왜?

- “ 1. 시간을 의존성 주입으로 해결 (?)

예시 | 마지막 로그인 시간

```
class UserService {  
    public void login(User user) {  
        // ...  
        user.login(Clock.systemUTC());  
    }  
}
```

```
class UserServiceTest {  
    @Test  
    public void login_테스트() {  
        // given  
        User user = new User();  
        UserService userService = new UserService();  
  
        // when  
        userService.login(user);  
  
        // then  
        assertThat(user.getLastLoginTimestamp()).isEqualTo(????);  
    }  
}
```



똑같은 문제가 발생하는데요...?

2. 의존성과 테스트

갑자기 의존성은 왜?

“ 폭탄을 넘겨줬을 뿐, 어딘가에선 고정된 값을 넣어줘야 합니다.

예시 | 마지막 로그인 시간

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login(Clock clock) {  
        // ...  
        this.lastLoginTimestamp = clock.millis();  
    }  
}
```

```
class UserService {  
    public void login(User user) {  
        // ...  
        user.login(Clock.systemUTC());  
    }  
}
```

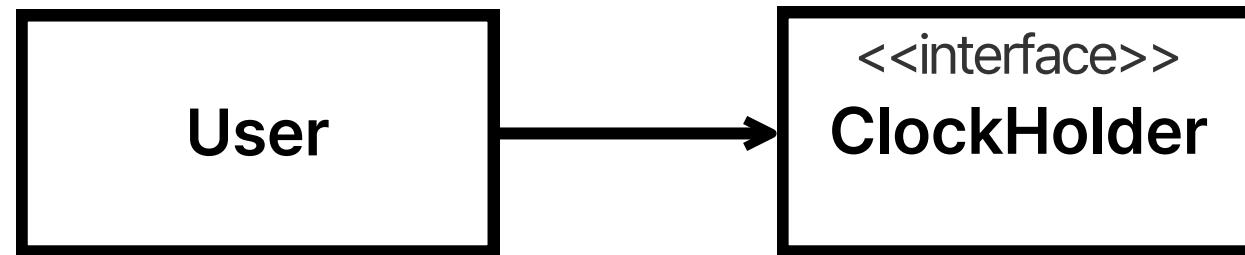


2. 의존성과 테스트

갑자기 의존성은 왜?

“ 2. 의존성 주입 + 의존성 역전으로 해결

```
interface ClockHolder {  
    long getMillis();  
}  
  
@Getter  
class User {  
  
    private long lastLoginTimestamp;  
  
    public void login(ClockHolder clockHolder) {  
        // ...  
        this.lastLoginTimestamp = clockHolder.getMillis();  
    }  
}  
  
@Service  
@RequiredArgsConstructor  
class UserService {  
  
    private final ClockHolder clockHolder;  
  
    public void login(User user) {  
        // ...  
        user.login(clockHolder);  
    }  
}
```

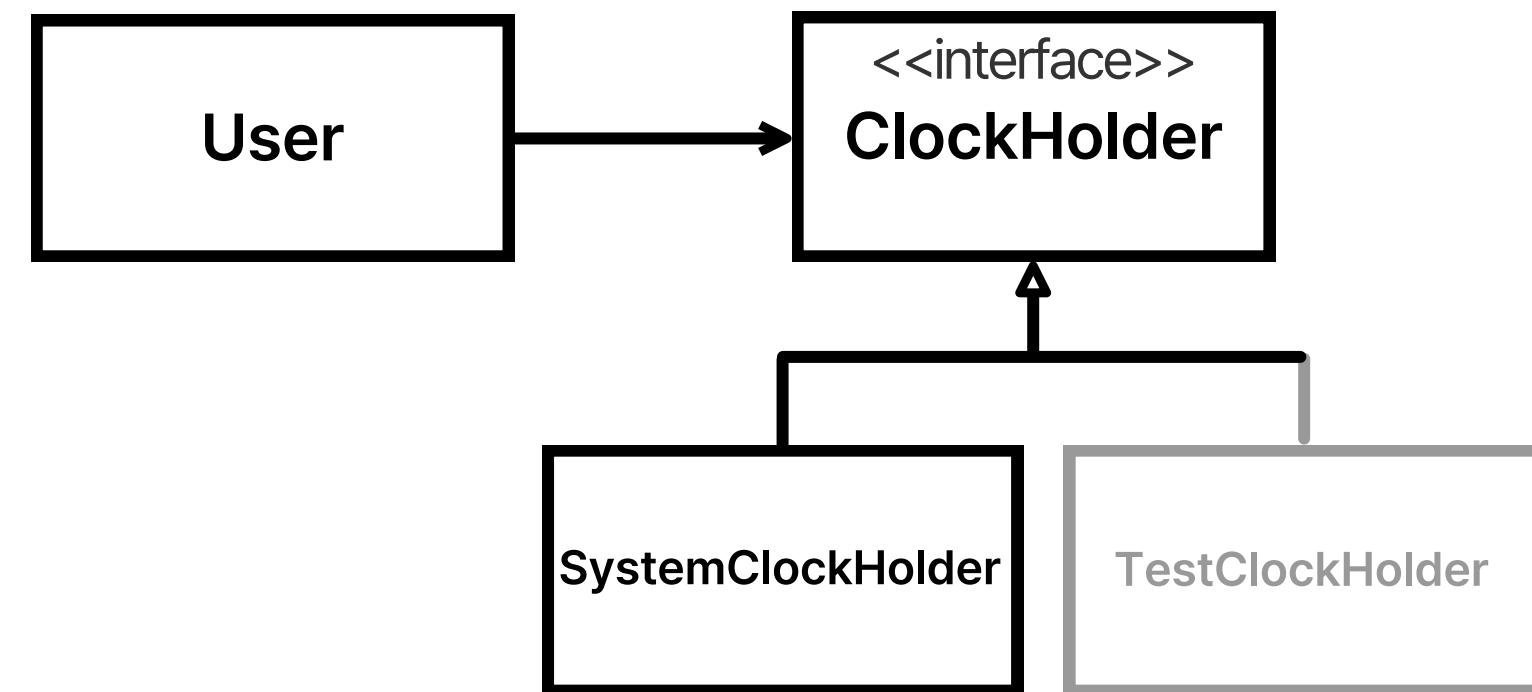


2. 의존성과 테스트

갑자기 의존성은 왜?

- “ 2. 의존성 주입 + 의존성 역전으로 해결

```
interface ClockHolder {  
    long getMillis();  
}  
  
@Getter  
class User {  
  
    private long lastLoginTimestamp;  
  
    public void login(ClockHolder clockHolder) {  
        // ...  
        this.lastLoginTimestamp = clockHolder.getMillis();  
    }  
}  
  
@Service  
@RequiredArgsConstructor  
class UserService {  
  
    private final ClockHolder clockHolder;  
  
    public void login(User user) {  
        // ...  
        user.login(clockHolder);  
    }  
}
```



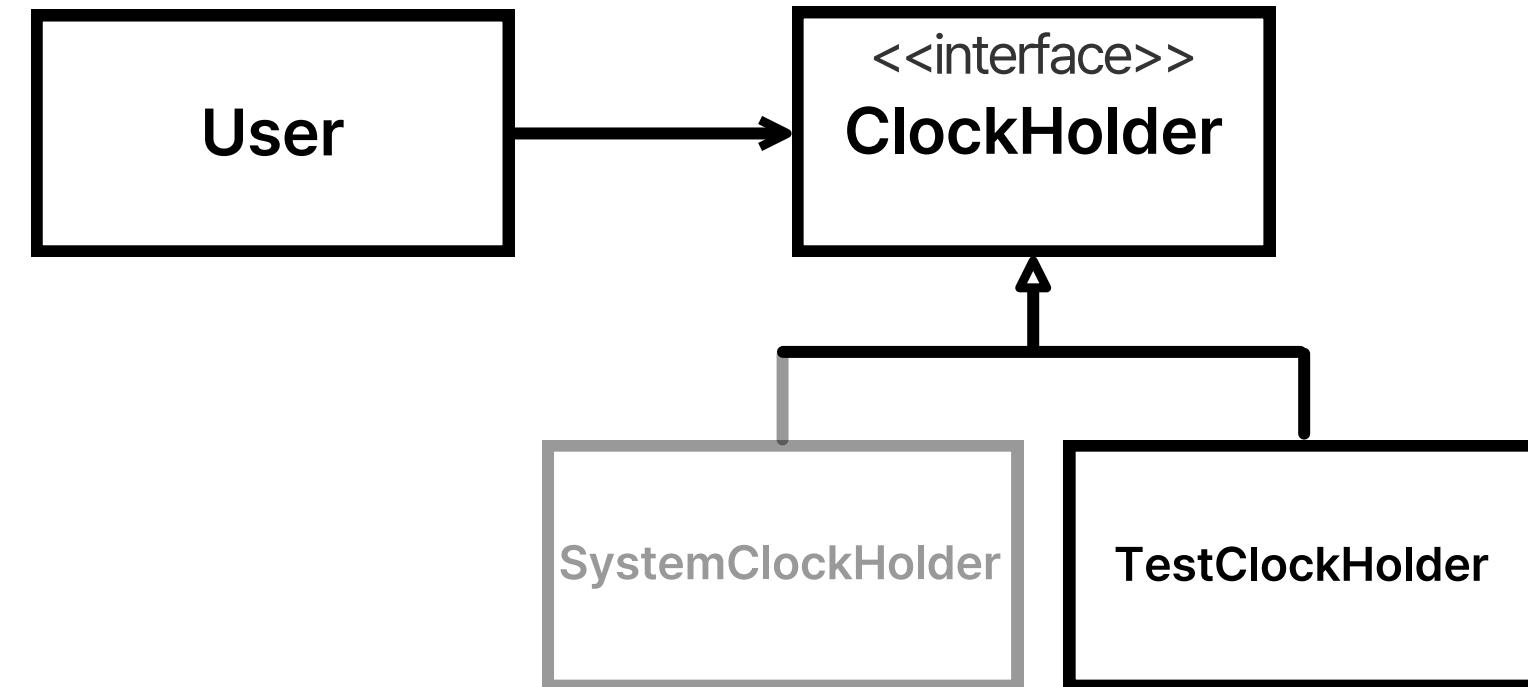
```
@Component  
class SystemClockHolder implements ClockHolder {  
  
    @Override  
    public long getMillis() {  
        return Clock.systemUTC().millis();  
    }  
}
```

2. 의존성과 테스트

갑자기 의존성은 왜?

- “ 2. 의존성 주입 + 의존성 역전으로 해결

```
interface ClockHolder {  
    long getMillis();  
}  
  
@Getter  
class User {  
  
    private long lastLoginTimestamp;  
  
    public void login(ClockHolder clockHolder) {  
        // ...  
        this.lastLoginTimestamp = clockHolder.getMillis();  
    }  
}  
  
@Service  
@RequiredArgsConstructor  
class UserService {  
  
    private final ClockHolder clockHolder;  
  
    public void login(User user) {  
        // ...  
        user.login(clockHolder);  
    }  
}
```



```
@AllArgsConstructor  
class TestClockHolder implements ClockHolder {  
  
    private Clock clock;  
  
    @Override  
    public long getMillis() {  
        return clock.currentTimeMillis();  
    }  
}
```

2. 의존성과 테스트

갑자기 의존성은 왜?

“ 2. 의존성 주입 + 의존성 역전으로 해결

```
class UserServiceTest {  
  
    @Test  
    public void login_테스트() {  
        // given  
        Clock clock = Clock.fixed(Instant.parse("2000-01-01T00:00:00.00Z"), ZoneId.of("UTC"));  
        User user = new User();  
        UserService userService = new UserService(new TestClockHolder(clock));  
  
        // when  
        userService.login(user);  
  
        // then  
        assertThat(user.getLastLoginTimestamp()).isEqualTo(946684800000L);  
    }  
}
```

2. 의존성과 테스트

갑자기 의존성은 왜?

“ 프로덕션 환경에서는 스프링을 쓴다면 알아서 잘 주입 해줄 겁니다.

```
@Getter  
class User {  
  
    private long lastLoginTimestamp;  
  
    public void login(ClockHolder clockHolder) {  
        // ...  
        this.lastLoginTimestamp = clockHolder.getMillis();  
    }  
}  
  
@Service  
@RequiredArgsConstructor  
class UserService {  
  
    private final ClockHolder clockHolder;  
  
    public void login(User user) {  
        // ...  
        user.login(clockHolder);  
    }  
}  
  
@Component  
class SystemClockHolder implements ClockHolder {  
  
    @Override  
    public long getMillis() {  
        return Clock.systemUTC().millis();  
    }  
}
```

2. 의존성과 테스트

SOLID → DIP: 의존성 역전 원칙

“ 대부분의 소프트웨어 문제는 의존성 역전으로 해결이 가능하다.

3. Testability

Testability

Testability에 대해 같이 살펴봅니다

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
class User {  
    private long lastLoginTimestamp;  
  
    public void login() {  
        // ...  
        this.lastLoginTimestamp = Clock.systemUTC().millis();  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
@Getter  
@Builder(access = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class Account {  
  
    private final String username;  
    private final String authToken;  
  
    public static Account create(String username) {  
        return Account.builder()  
            .username(username)  
            .authToken(UUID.randomUUID().toString())  
            .build();  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
@Getter  
@Builder(access = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class Account {  
  
    private final String username;  
    private final String authToken;  
  
    public static Account create(String username) {  
        return Account.  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
    }  
}
```



어...? 권한 인증에 사용되는 토큰은 어떻게 만들어 지는거지?

```
@Getter  
@Builder(access = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class Account {  
  
    private final String username;  
    private final String authToken;  
  
    public static Account create(String username) {  
        return Account.  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
    }  
}
```

```
@Getter  
@Builder(access = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class Account {  
  
    private final String username;  
    private final String authToken;  
  
    public static Account create(String username) {  
        return Account.builder()  
            .username(username)  
            .authToken(UUID.randomUUID().toString())  
            .build();  
    }  
}
```



UUID가 사용된다는 얘기는 처음 듣는데요?

3. Testability

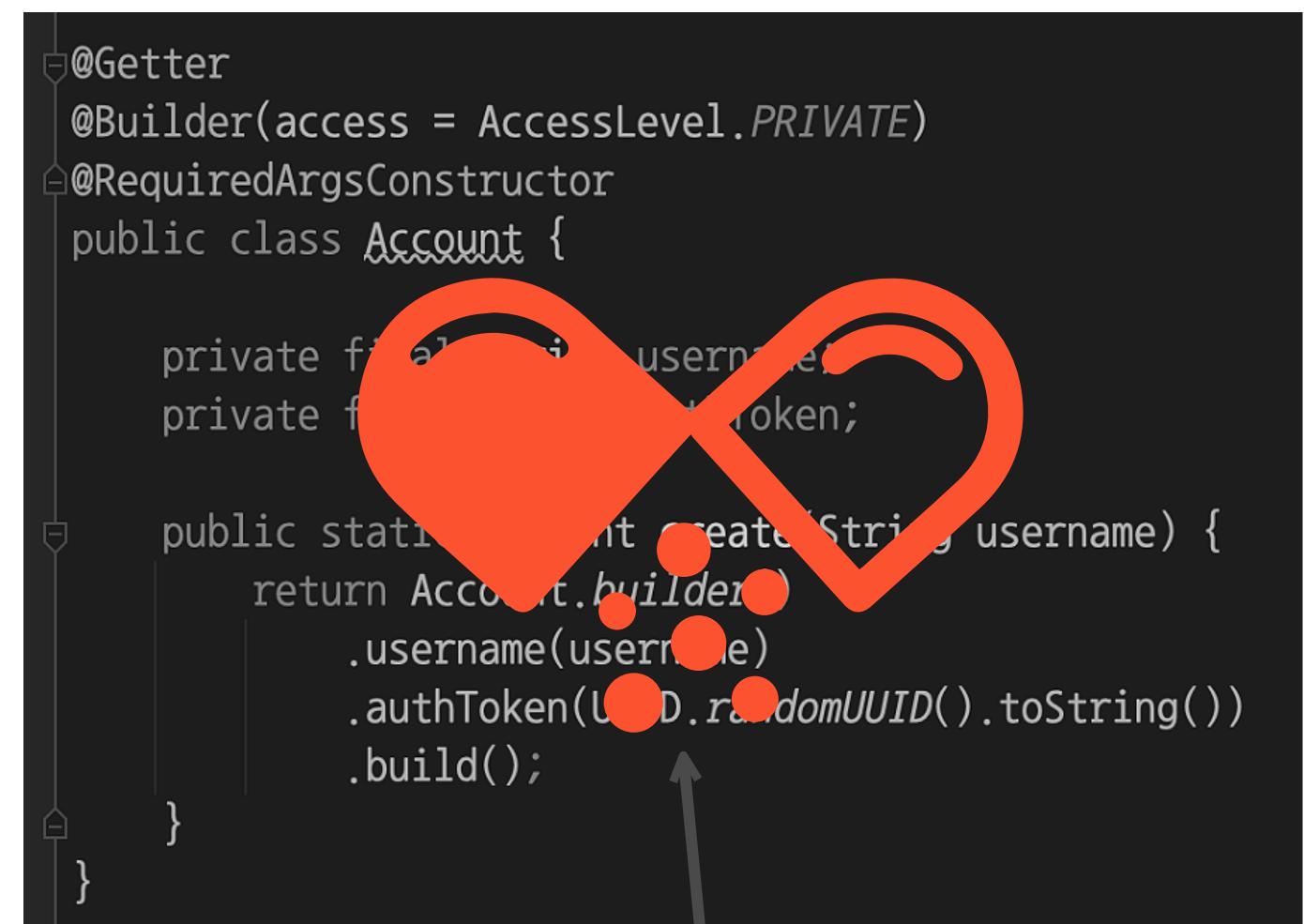
테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
    }  
}
```



```
@Getter  
@Builder(access = AccessLevel.PRIVATE)  
@RequiredArgsConstructor  
public class Account {  
  
    private final String username;  
    private final String authToken;  
  
    public static Account create(String username) {  
        return Account.builder()  
            .username(username)  
            .authToken(UUID.randomUUID().toString())  
            .build();  
    }  
}
```



UUID가 사용된다는 얘기는 처음 듣는데요?

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 의존성

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
        assertThat(account.getAuthToken()).isEqualTo(랜덤_값을_어떻게);  
    }  
}
```



심지어 값을 비교할 방법도 없네...?

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

어거리 테스트

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
        String expectedAuthToken = "550e8400-e29b-41d4-a716-446655440000";  
        PowerMockito.mockStatic(UUID.class);  
        when(UUID.randomUUID()).thenReturn(UUID.fromString(expectedAuthToken));  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
        assertThat(account.getAuthToken()).isEqualTo(expectedAuthToken);  
    }  
}
```



static 메소드를 mock 해줘야겠다

static 메소드는 Mockito로 stub이 불가하니
이걸 가능하게 해주는 PowerMock 라이브러리 추가

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

어거리 테스트

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
        String expectedAuthToken = "550e8400-e29b-41d4-a716-446655440000";  
        PowerMockito.mockStatic(UUID.class);  
        when(UUID.randomUUID()).thenReturn(UUID.fromString(expectedAuthToken));  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
        assertThat(account.getAuthToken()).isEqualTo(expectedAuthToken);  
    }  
}
```



3. Testability

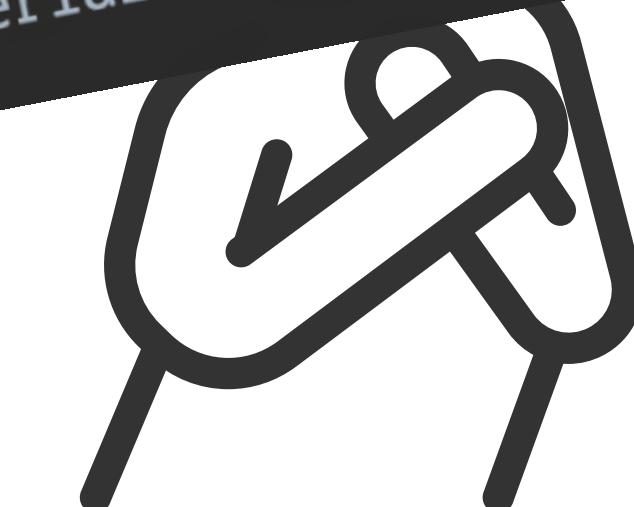
테스트 가능성

“얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

어거지 테스트

호출자는 모르는 입력이 존재한다.

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
        String password = "password";  
        byte[] expectedAuthToken = new byte[16];  
        String expectedBase64AuthToken = Base64.getEncoder().encodeToString(expectedAuthToken);  
  
        // when  
        Account account = Account.create(username, password);  
  
        // then  
        assertEquals("foobar", account.getUsername());  
        assertEquals(expectedBase64AuthToken, account.getAuthToken());  
    }  
}
```



3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

어거리 테스트

뭐 어떻게든 방법을 찾을 수는 있겠지만, 뭔가 잘못됨을 느끼고 설계를 고쳐야 합니다. 어떻게? [의존성 역전으로](#)

```
class AccountTest {  
  
    @Test  
    void create() {  
        // given  
        String username = "foobar";  
        String expectedAuthToken = "550e8400-e29b-41d4-a716-446655440000";  
        PowerMockito.mockStatic(UUID.class);  
        when(UUID.randomUUID()).thenReturn(UUID.fromString(expectedAuthToken));  
  
        // when  
        Account account = Account.create(username);  
  
        // then  
        assertThat(account.getUsername()).isEqualTo("foobar");  
        assertThat(account.getAuthToken()).isEqualTo(expectedAuthToken);  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

하드 코딩

파일이 존재하지 않을 때를 테스트 할 수 없다.

```
public class Example {  
    private static final File FILE = new File( pathname: "data.txt");  
  
    public void processData() {  
        // read from FILE and process data  
    }  
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

외부 시스템

하드 코딩된 외부 시스템과 연동이 되어있는 경우

```
public class Example {
    public void processData(String data) {
        String processedData = data.toUpperCase(); // 데이터 처리
        sendDataOverNetwork(processedData);
    }

    private void sendDataOverNetwork(String data) {
        // HTTP를 이용한 네트워크 요청
    }
}
```

3. Testability

테스트 가능성

“ 얼마나 쉽게 input을 변경하고, output을 쉽게 검증할 수 있는가?

감춰진 결과

외부에서 결과를 볼 수 없는 경우

```
public class Example {  
    public void processData(int[] numbers) {  
        int sum = 0;  
        for (int number : numbers) {  
            sum += number;  
        }  
        System.out.println("Sum: " + sum);  
    }  
}
```

정리

- 의존성과 의존성 주입, 의존성 역전에 대해 살펴봤습니다
- 의존성과 테스트의 상관 관계에 대해 살펴봤습니다
- Testability에 대해 같이 살펴봤습니다

RETURN;