Junit and assertJ – 실용적인 테스트 가이드

1. Junit and assertJ

둘 다 테스트 용도로 사용하는 라이브러리. Junit은 JAVA 프로젝트 테스트에 필요한 기본 문법을 제공하고, assertJ는 Junit을 좀 더 편리하게 사용하기 위한 다양한 API를 제공한다.

2. 기본 문법

(1) assertThat

특정 객체의 값이 서로 일치하는지 비교할 수 있다.

```
assertThat(cafeKiosk.getBeverages().get(0)).isEqualTo(americano);
assertThat(cafeKiosk.getBeverages().get(1)).isEqualTo(americano);
```

(2) assertThatThrownBy

예외가 던져졌는지 검사할 수 있다.

```
assertThatThrownBy(()-> cafeKiosk.add(americano, 0))
        .isInstanceOf(IllegalArgumentException.class)
        .hasMessage("음료는 1잔 이상 주문할 수 있습니다.");
```

isInstanceOf 를 사용하면 어떤 종류의 예외가 던져졌는지 검사할 수 잇다.

hasMessage를 사용하면 예외에 어떤 문구가 포함되어 있는지 검사할 수 있다.

3. 테스트 기본

(1) 해피 케이스와 예외 케이스에 관해 각각 테스트를 작성한다.

(2) 경계값을 테스트하는 것이 중요하다.

(3) 테스트하기 어려운 영역도 존재한다. (날짜, 시간 등 가변적인 input, DB에 데이터 넣기 등 직접적으로 외부에 output 등)

4. 테스트 주도 개발(TDD)

- 프로덕션 코드보다 테스트 코드를 먼저 작성해서 테스트 코드가 프로덕션 코드를 이끄는 방식

실패하는 테스트 작성 -> 테스트를 통과하기 위한 최소한의 코딩 -> 리팩토링을 하면서 구현코드를 개선함.

5. 테스트 이름 정하기

- @DisplayName을 사용하여 테스트 이름을 지정할 수 있다. 테스트 이름을 섬세하게 정하자. 명사의 나열보다는 문장으로 표현하는 것이 좋다.
- 도메인 용어를 사용하여 좀 더 specific한 표현을 문장으로 쓰도록 한다. 메서드 자체 관점 보다는 도메인(비즈니스적) 용어를 사용하여 문장을 쓴다.

6. BDD 스타일로 작성하기

- TDD에서 파생된 개발 방법
- 함수 단위로 테스트하지 않고, 시나리오 기반한 테스트케이스에 집중하여 테스트한다.
- 개발자가 아닌 사람이 봐도 이해할 수 있을 정도의 추상화 수준을 권장함.

(1) Given: 시나리오 진행에 필요한 모든 준비 과정 (환경)

(2) When: 시나리오 행동 진행 (행동)

(3) Then: 시나리오 진행에 대한 결과 검증 (결과)

** 참고 **

intellij의 settins의 live templates으로 들어가면 java 클릭하고 + 눌러서 새로운 템플릿을 만든다. 다만, 애노테이션은 라이브러리의 풀네임을 적어줘야 한다.

7. Test annotations

- 스프링 빈을 가져와서 테스트에 활용하려면 이 annotation들을 사용해야 한다. 이것들은 메서드가 아니라 클래스 앞에 붙여야 한다.
- 각 domain 객체에 대한 test는 메서드에 @Test만 붙이는 걸로 충분하다.

1) @SpringBootTest: Spring 서버를 띄워서 테스트 진행함

2) @DataJpaTest: 스프링 부트 테스트보다 가볍다. JPA 관련 bean 들만 띄워서 테스트 진행. persistence layer 테스트할 때만 사용하자.

8. Persistence Layer 테스트

(1) layer 역할: Data를 읽고 쓰는 것에만 집중한다.

(2) 테스트 문법

1) .extracting() 안에 필드명을 쓰면 필드명에 해당하는 데이터를 추출할 수 있다.

2) .containsExactlyInAnyOrder() 안에 값을 쓰면 순서에 상관 없이 데이터가 있는지 확인해 준다. 이 메서드를 쓰려면 tuple()을 사용하여 넣어줘야 한다.

```
@DisplayName("원하는 판매상태를 가진 상품들을 조회한다.")
@Test
void findAllBySellingStatusIn() {
    // given
    Product product1 = Product.builder()
            .productNumber("001")
            .type(HANDMADE)
            .sellingStatus(SELLING)
            .name("아메리카노")
            .price(4000)
            .build();
    Product product2 = Product.builder()
            .productNumber("002")
            .type(HANDMADE)
            .sellingStatus(HOLD)
            .name("카페라떼")
            .price(4500)
            .build();

    productRepository.saveAll(List.of(product1, product2));

    // when
    List<Product> products =
productRepository.findAllBySellingStatusIn(List.of(SELLING, HOLD));

    // then
    assertThat(products).hasSize(2)
            .extracting("productNumber", "name", "sellingStatus")
            .containsExactlyInAnyOrder(
                    tuple("001", "아메리카노", SELLING),
                    tuple("002", "카페라떼", HOLD)
            );
}
```

9. Business Layer 테스트

(1) layer 역할: 비즈니스 로직을 전개시킨다.

(2) 테스트 작성법: Business layer와 persistence layer를 통합한 테스트를 작성한다.

(3) 테스트 문법

1) .isNotNull 은 해당 객체가 Null이 아닐 경우 통과가 된다.

2) .isEqualByComparingTo 는 enum 값을 비교해주는 메서드

(4) 테스트 예시

- repository와 service 모두 빈으로 가져와 테스트한다.

```
@Autowired
private ProductRepository productRepository;
@Autowired
private OrderService orderService;
@DisplayName("주문번호 리스트를 받아 주문을 생성한다.")
@Test
void createOrder() {
    // given
    Product product1 = createProduct(HANDMADE, "001", 1000);
    Product product2 = createProduct(HANDMADE, "002", 3000);
    productRepository.saveAll(List.of(product1, product2, product3));

    OrderCreateRequest request = OrderCreateRequest.builder()
            .productNumbers(List.of("001", "002"))
            .build();

    // when
    OrderResponse orderResponse = orderService.createOrder(request,
registeredDateTime);

    // then
    assertThat(orderResponse.getId()).isNotNull();
    assertThat(orderResponse.getProducts()).hasSize(2)
            .extracting("productNumber", "price")
            .containsExactlyInAnyOrder(
                    tuple("001", 1000),
                    tuple("002", 3000)
            );
}
```

(5) 도메인 객체 테스트

- 빈 주입 필요가 없는 테스트
- 객체 기능만 테스트하면 된다.

```
@DisplayName("주문 생성 시 주문 상태는 INIT 이다.")
@Test
void init() {
    // given
    List<Product> products = List.of(
            createProduct("001", 1000),
            createProduct("002", 2000)
    );

    // when
    Order order = Order.create(products, LocalDateTime.now());

    // then

assertThat(order.getOrderStatus()).isEqualByComparingTo(OrderStatus.INIT);
}
```

(6) 테스트 관련 데이터 지우기

- 복수 개의 테스트를 돌릴 때 데이터를 중복해서 쓰기가 일어날 수도 있다.
- 각각의 테스트는 서로에게 영향을 주면 안 된다. 그러므로 이전 테스트의 데이터를 지워줘야 한다.
- @AfterEach 를 사용하여 매 test 마다 생성한 데이터를 지운다.

```
@AfterEach
void tearDown() {
    orderProductRepository.deleteAllInBatch();
    productRepository.deleteAllInBatch();
    orderRepository.deleteAllInBatch();
    stockRepository.deleteAllInBatch();
}
```

10. Mockito

@SprintBootTest는 통합 테스트이다. 스프링 부트를 띄워서 테스트하므로 테스트에서 만들어진 객체가 DB에 저장되는 등 프로덕션 코드에 영향을 줄 수 있다. DB에 쓰레기 값이 만들어질 수 있다. 이를 방지하기 위해서는 통합테스트가 아닌 유닛테스트만 진행해야 하고, 그러기 위해서는 Mock을 사용해야 한다.

Mockito는 Spring boot stater dependency를 등록하면 자동으로 따라온다.

```
List mockedList = mock(List.class);
```

이렇게 mock으로 감싸면 해당 클래스를 테스트할 수 있다.

(1) @MockBean

이건 예외적으로 스프링을 띄워야 동작하는 객체를 만들어주는 애노테이션이다

스프링 컨테이너에 Mock으로 만든 객체를 넣어주는 역할을 한다. 가짜 Bean을 넣어주는 것이다. 이렇게 가짜 Bean을 넣어주면 SpringBoot를 사용하여 Bean을 주입하지 않아도 MockBean을 사용하여 테스트를 동작시킬 수 있다.

하지만 이 MockBean은 결국 스프링 컨텍스트가 동작해야 사용할 수 있다. 스프링 컨텍스트에 있는 Bean을 갈아 끼우겠다는 의미. 하지만 단위테스트에서는 이걸 사용할 수 없음

아래부터는 진짜로 순수 mockito 만을 사용하여 유닛 테스트를 작성하는 방법

(2) Mockito 객체

Mockito.mock(클래스명.class)을 하면 mock 객체가 만들어진다.

```
MemberRepository memberRepository = Mockito.mock(MemberRepository.class);
AddressService addressService = Mockito.mock(AddressService.class);
MemberService memberService = new MemberService(memberRepository,
addressService);
```

(3) @Mock

Mock 객체를 property에서 만드는 방법

우선 @ExtendWith(MockitoExtension.class) 를 테스트 클래스명 위에 달아줘야 한다.

```
@ExtendWith(MockitoExtension.class)
```

속성에 mock으로 등록할 것들을 작성하면 됨

```
@Mock
private MemberRepository memberRepository;
@Mock
private AddressService addressService;
```

(4) @InjectMocks

```
MemberService memberService = new MemberService(memberRepository,
addressService);
```

이 문장도 대체할 수 있다. @InjectMocks를 사용하면 Mock이 대신 만들어준다.

```
@InjectMocks
private MemberService memberService;
```

(5) stubbing

특정 메소드의 리턴 값을 강제적으로 지정해주는 방법.

when(객체명.메소드명).thenReturn(리턴값);

이 형식으로 작성하면 해당 메서드가 작동되었을 때 리턴값이 강제적으로 정해진다.

```
Mockito.when(memberRepository.findById("123").get()).thenReturn(mockMember)
;
```

(6) 검증하기

Mocito의 verify 메소드를 사용하면 특정 메소드가 몇 번 실행됐는지 횟수를 검증할 수 있다.

```
Mockito.verify(memberRepository,
Mockito.times(1)).make(Mockito.any(Member.class));
```

위 코드는 memberRepositoy의 make() 메서드가 1번 실행되었는지 검증하는 코드다.

(7) @Spy

Spy의 경우는 mock과는 달리 실제 객체를 활용해서 동작시키는 것. 일부만 실제 객체의 기능을 사용하고 일부는 기능을 mock하고 싶을 때 사용한다. 자주 사용되지는 않음.

11. BDD Mockito

BDD관점에서 given의 포지션에 when이라는 문법이 있는 것이 어색해 보여서 나온 문법이다.

stubbing이라는 것 자체가 테스트 대상을 위한 준비 과정이므로 given에 속해 있기 때문.

그냥 기능은 mock stubbing의 when과 같다. 다만 표현 이름만 바뀐 것이다.

```
BDDMockito.given(memberRepository.findById("123").get()).willReturn(mockMember);
```

BDDMockito 객체의 given() 메서드를 사용하고, 리턴 지정은 .willReturn()을 사용하면 된다.