TDD 이해

목차

1. TDD 기본 이해

2. 테스트 전략

---

1.TDD 기본 이해

1) TDD 목적

코드 품질 향상으로 유지보수 비용 절감

- 설계 품질 향상: 테스트를 먼저 작성하면서 코드 구조와 인터페이스를 먼저 고민

- 회귀 버그 방지: 테스트 자동화로 코드 변경 시 기존 기능의 오작동을 빠르게 감지

- 리팩토링 검증: 코드 개선 후 테스트 코드로 검증할 수 있어 리팩토링에 대한 자신감 확보

- 살아있는 문서: 테스트 코드에 샘플 데이터를 이용한 예시가 있으므로 실제 코드의 동작 방식을 문서화

---

2) 테스트 유형

- 단위 테스트(Unit Test): 외부 기술요소(DB, 웹서버 등)와의 인터페이스 없이 단위 클래스의 퍼블릭 메소드 테스트

- 통합 테스트(Integration Test): 일부 아키텍처 영역에서 외부 기술 요소와 인터페이스까지 테스트

- E2E 테스트(E2E Test): 모든 아키텍처 영역에서 외부 기술 요소와 인터페이스를 테스트

\* 아키텍처 영역: 클래스를 아키텍처적으로 나눈 레이어를 의미함(예: controller, service, domain, repository)

---

3) 테스트 피라미드

- 단위 테스트 70%, 통합 테스트 20%, E2E 테스트 10%의 비율로 권장

- Mike Cohn이 "Succeeding with Agile"에서 처음 제시한 개념

- 단위 테스트에서 E2E 테스트로 가면서 속도는 느려지고 비용은 높아짐

---

4) Red-Green-Refactor 사이클

Red-Green-Refactor는 TDD(Test-Driven Development)를 수행하는 핵심 사이클임

- Red (실패하는 테스트 작성)

- 새로운 기능에 대한 테스트 코드를 먼저 작성

- 아직 구현이 없으므로 테스트는 실패

- 이 단계에서 기능의 인터페이스를 설계

- Green (테스트 통과하는 코드 작성)

- 테스트를 통과하는 최소한의 코드 작성

- 품질보다는 동작에 초점

- Refactor (리팩토링)

- 중복 제거, 가독성 개선

- 테스트는 계속 통과하도록 유지

- 코드 품질 개선

---

2. 테스트 전략

1) 테스트 수행 원칙: FIRST 원칙

- Fast: 테스트는 빠르게 실행되어야 함

- Isolated: 각 테스트는 독립적이어야 함

- Repeatable: 어떤 환경에서도 동일한 결과가 나와야 함

- Self-validating: 테스트는 성공/실패가 명확해야 함

- Timely: 테스트는 실제 코드 작성 전/직후에 작성되어야 함

---

2) 공통 전략: 테스트 코드 작성 관련

- 한 테스트는 한 가지만 테스트

- Given-When-Then 패턴 사용

- Given(준비): 테스트에 필요한 상태와 데이터를 설정

- When(실행): 테스트하려는 동작을 수행

- Then(검증): 기대하는 결과가 나왔는지 확인

- 깨끗한 테스트 코드 작성

- 테스트 의도를 명확히 하는 네이밍

- 테스트 케이스는 시나리오 중심으로 구성

- 공통 설정은 별도 메서드로 분리

- 매직넘버 대신 상수 사용

- 테스트 데이터는 최소한으로 사용

- 경계값 테스트가 중요

- null 값

- 빈 컬렉션

- 최대/최소값

- 0이나 1과 같은 특수값

- 잘못된 포맷의 입력값

---

2) 공통 전략: 테스트 코드 관리 관련

- 비용 효율적인 테스트 전략

- 자주 변경되는 비즈니스 로직에 대한 테스트 강화

- 실제 운영 환경과 유사한 통합 테스트 구성

- 테스트 실행 시간과 리소스 사용량 모니터링

- 지속적인 테스트 개선

- 테스트 커버리지보다 테스트 품질 중시

- 깨진 테스트는 즉시 수정하는 문화 정착

- 테스트 코드도 실제 코드만큼 중요하게 관리

- 팀 협업을 위한 가이드라인 수립

- 테스트 네이밍 컨벤션 수립

- 테스트 데이터 관리 전략 합의

- 테스트 실패 시 대응 프로세스 수립

---

3) 단위 테스트 전략

- 테스트 범위 명확화

- 클래스의 각 public 메소드가 수행하는 단일 책임을 검증

- private 메서드는 public 메서드를 통해 간접적으로 테스트

- 외부 의존성 처리

- DB, 파일, 네트워크 등 외부 시스템은 가짜 객체로 대체(Mocking)

- 테스트 더블(스턴트맨을 Stunt Double이라고 함. 대역으로 이해)은 꼭 필요한 동작만 구현

- Mock: 메소드 호출 여부와 파라미터 검증

- Stub: 반환값의 일치 여부 검증

- Spy: Mocking하지 않고 실제 메소드를 감싸서 호출횟수, 호출순서등 추가 정보 검증

- 격리성 확보

- 테스트 간 상호 영향 없도록 설계: 동일 공유 자원/객체를 사용하지 않게 함

- 테스트 실행 순서와 무관하게 동작

- 가독성과 유지보수성

- 테스트 대상 클래스당 하나의 테스트 클래스

- 테스트 메서드는 한 가지 시나리오만 검증

---

4) 단위 테스트 시 Mocking 전략

- 외부 시스템(DB, 외부 API 등)은 반드시 Mocking

- 같은 레이어의 의존성 있는 클래스는 실제 객체 사용

- 예외적으로 의존 객체가 매우 복잡하거나 무거운 경우 Mocking 고려

\* 참고: 모의 객체 테스트 균형점 찾기

출처: When to mocking by Uncle Bob(https://blog.cleancoder.com/uncle-bob/2014/05/10/WhenToMock.html)

- 모의 객체를 이용 안 하면: 테스트가 오래 걸리고 결과를 신뢰하기 어려우며 인프라에 너무 많은 영향을 받음

- 모의 객체를 지나치게 사용하면: 복잡하고 수정에 영향을 너무 많이 받으며 모의 인터페이스가 폭발적으로 증가

- 균형점 찾기

- 아키텍처적으로 중요한 경계에서만 모의 테스트를 수행하고, 그 경계 안에서는 하지 않는다.

(Mock across architecturally significant boundaries, but not within those boundaries.)

- 여기서 경계란 Controller, Service, Repository, Domain등의 레이어를 의미함

---

5) 통합 테스트 전략

- 웹 서버 인터페이스

- @WebMvcTest, @WebFluxTest 활용

- Controller 계층의 요청/응답 검증

- Service 계층은 Mocking 처리

- Database 인터페이스

- @DataJpaTest 활용

- TestContainer로 실제 DB 엔진 실행

- 외부 서비스 인터페이스

- WireMock 등을 활용한 Mocking

- 실제 API 스펙 기반 테스트

- 테스트 환경 구성

- 테스트용 별도 설정 파일 구성

- 테스트 데이터는 테스트 시작 시 초기화

- @Transactional을 활용한 테스트 격리

- 테스트 간 독립성 보장

---

6) E2E 테스트 전략

- 원칙

- 단위 테스트나 컴포넌트 테스트에서 놓칠 수 있는 시나리오를 찾아내는 것이 목표임

- 조건별 로직이나 분기 상황(edge cases)이 아닌 상위 수준의 일반적인 시나리오만 테스트

- 만약 어떤 시스템 테스트 시나리오가 실패 했는데 단위 테스트나 통합 테스트가 없다면 만들어야 함

- 운영과 동일한 테스트 환경 구성: 웹서버/WAS, DB, 캐시, MQ, 외부시스템

- 테스트 데이터 관리

- 테스트용 마스터 데이터 구성

- 시나리오별 테스트 데이터 세트 준비

- 데이터 초기화 및 정리 자동화

- 테스트 자동화 전략

- UI 테스트: Selenium, Cucumber, Playwright 등 도구 활용

- API 테스트: Rest-Assured, Postman 등 도구 활용

---

7) 테스트 코드 네이밍 컨벤션

- 패키지 네이밍

```

[Syntax]

{프로덕션패키지}.test.{테스트유형}

[Example]

- 단위테스트: com.company.order.test.unit

- 통합테스트: com.company.order.test.integration

- E2E테스트: com.company.order.test.e2e

- 클래스 네이밍

```

[Syntax]

{대상클래스}{테스트유형}Test

[Example]

- 단위테스트: OrderServiceUnitTest

- 통합테스트: OrderServiceIntegrationTest

- E2E테스트: OrderServiceE2ETest

```

- 메소드 네이밍

```

[Syntax]

given{초기상태}\_when{행위}\_then{결과}

[Example]

givenEmptyCart\_whenAddItem\_thenSuccess()

givenInvalidToken\_whenAuthenticate\_thenThrowException()

givenExistingUser\_whenUpdateProfile\_thenProfileUpdated()

```

- 테스트 데이터 네이밍

```

[Syntax]

상수: {상태}\_{대상}

변수: {상태}{대상}

[Example]

// 상수

VALID\_USER\_ID = 1L

EMPTY\_ORDER\_LIST = Collections.emptyList()

// 변수

normalUser = new User(...)

emptyCart = new Cart()

```