TDD 소개.

백명석 & 최범균 강사

Fast campus

TABLE OF CONTENT.

테스트 주도 개발이란?
& 시연1

테스트 주도 개발
Test-Driven Development란?

테스트로부터 시작하는 개발 방식

- (실패하는) 테스트 코드 작성
- 테스트를 통과시킬 만큼 구현
- 코드 정리(리팩토링)

테스트 주도 개발 Test-Driven Development 란?

TDD 예

“

암호 검사기

- 사용하는 규칙
  - 길이가 8글자 이상
  - 0부터 9사이 숫자 포함
  - 대문자 포함
- 세 규칙을 모두 충족하면 강함
- 2개 규칙을 충족하면 보통
- 1개 이하 규칙을 충족하면 약함

테스트 주도 개발 Test-Driven Development 란?

[코딩 진행] 시연1 - 암호 검사기

시연 진행 순서

1. 테스트 클래스와 메서드 작성, 실행
2. null 입력에 대한 테스트로 시작
   - 테스트 대상 타입 정의,
   - 메서드 정의, 결과 타입 정의
3. 빈 값에 대한 테스트 추가
4. 모든 조건 충족하는 예
5. 두 조건 충족하는 예
6. 한 조건 충족하는 예
7. 아무 조건도 충족하지 않는 예

테스트 주도 개발 Test-Driven Development 란?

TDD 로 진행했더니

테스트가
개발을 주도

지속적인
코드 정리

빠른 피드백

TDD와 설계

TDD와 설계

암호 검사 기능

암호 검사 기능을 실행하려면?

- PasswordChecker? PasswordMeter?
- 객체를 생성해서 메서드 실행? static 메서드 실행
- 파라미터는 문자열? 별도 타입?

검증하려면?

- 리턴 타입은 열거 타입? 단순 숫자?
- 열거 타입이면 이름은 PasswordLevel? PasswordStrength?
  - 열거 타입 값은 ? LEVEL1, LEVEL2, LEVEL3? STRONG, NORMAL, WEAK?

TDD와 설계

TDD는 설계를 지원

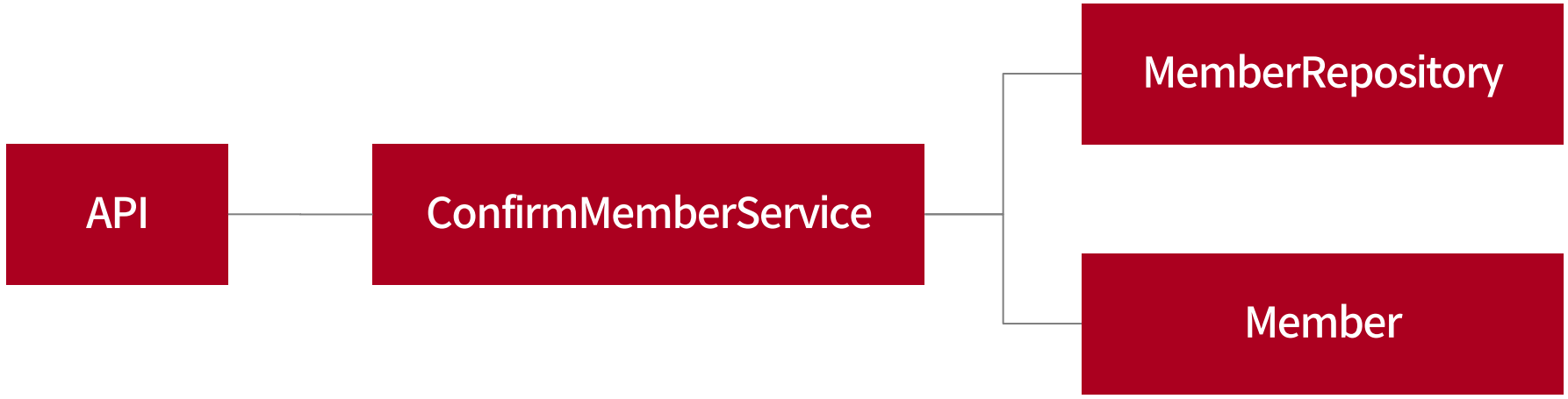
TDD와 설계

조금 더 큰 TDD 예

회원 승인 API

대기 상태의 회원을 승인하면 회원 상태가 활성으로 바뀜

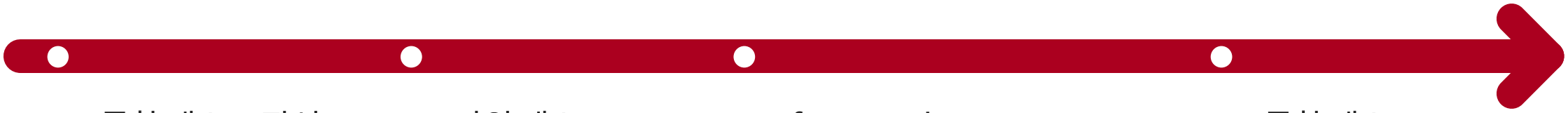
설계 초안



TDD와 설계

[코딩 진행] 시연2 – 회원 승인 기능

시연 진행 순서



API 통합 테스트 작성

- 상황 구성
- 실행 후 데이터 변경 확인
- 일단 API 통합 테스트는 에러 상태로 놔둠

API 단위 테스트

ConfirmMemberService 테스트 작성

- 회원 없는 경우 테스트
- 회원이 있는데 이미 활성 상태인 경우 테스트
  : MemberRepository 메모리 구현 사용
- 대기 상태 회원에 대한 승인 테스트

API 통합 테스트 통과 확인

TDD와 설계

TDD 로 진행했더니

테스트 대상 설계

- API URL
- 클래스, 메서드 명
- 파라미터, 리턴타입, 익셉션 타입 등

연동 대상 도출

- API 연동 대상 클래스
- 리포지토리

테스트 코드 구조

테스트 코드 구조

기능과 상황

주어진 상황에 따라 실행 결과가 달라짐

예: 승인 API 회원이 없는 경우와 존재하는 경우 같은 승인 요청도 응답이 다름

예: 병원 예약

- 평일인 경우와 휴일인 경우 응답이 다름
- 예약 인원을 초과한 경우와 초과하지 않은 경우 응답이 다름

테스트 코드 구조

테스트 구조 : 상황 – 실행 – 결과 검증

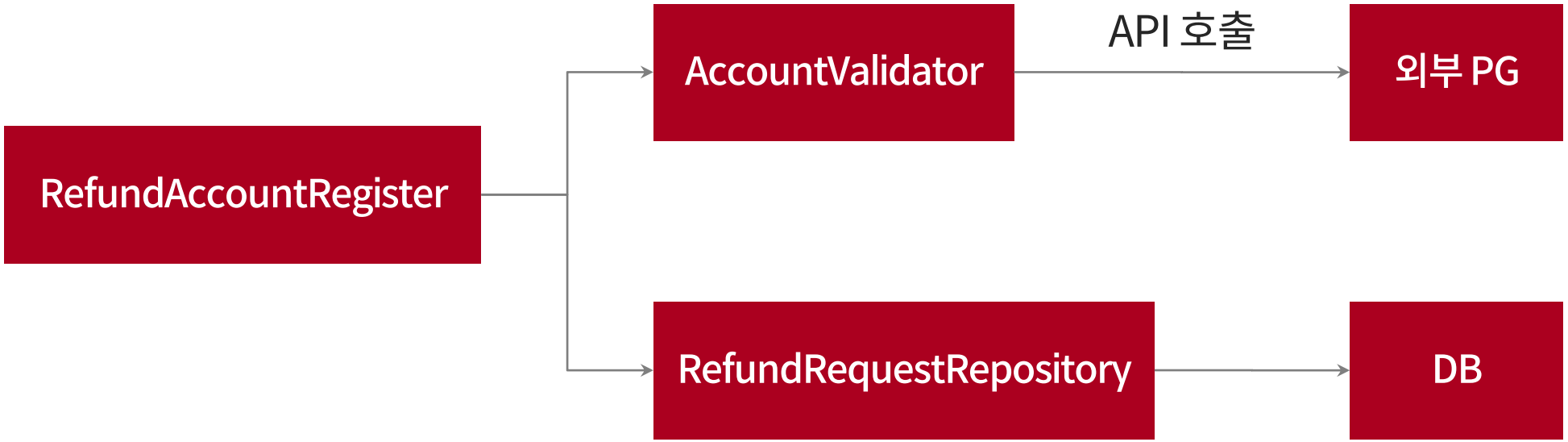
테스트는 상황-실행-결과로 구성

given – when – then

```
@Test
void confirmMember() {
    // 상황: 대기 상태 회원이 존재
    memoryMemberRepository.save(Member.id("id").status(WAITING).builder().build());

    // 실행: 회원을 승인하면
    confirmMemberService.confirm("id");

    // 결과: 회원이 활성 상태가 됨
    Member m = memoryMemberRepository.findById("id");
    assertThat(m.getStatus()).isEqualTo(ACTIVE);
}
```

테스트 코드 구조

외부 상황과 외부 결과



테스트 코드 구조

외부 상황과 외부 결과

- ☑ 테스트가 외부 상태에 의존하는 경우
  파일, DB, REST API, 소켓 서버, …
- ☑ 상황을 만들기 어려울 수 있음
- ☑ 결과를 확인하기 어려울 수 있음
- ☑ 다음 테스트에 영향을 줄 수 있음

대역

대역

테스트 대역(double)

테스트에서 실제 구현 대신에 사용할 대체 구현

예:
계좌번호 검증기

- 실제 구현은 PG가 제공하는 API를 호출하여 계좌 번호 검증
- 대체 구현은 PG와 통신하지 않고 지정한 값을 바로 제공(원하는 행동)

```
public class AccountValidator {
    ...
    public AccountValidity validate(String accountNo) {
        // call api

    }

}
```

```
public class StubAccountValidator extends AccountValidator {
    private AccountVality result;

    public void setResult(AccountVality result) {
        this.result = result;
    }

    @Override
    public AccountValidity validate(String accountNo) {
        return result;
    }
}
```

대역

테스트 대상은 실제 구현 대신 대역 사용

대역을 이용해서 테스트에 필요한 상황/결과를 구성

```
public class RefundAccountRegisterTest {
    private RefundAccountRegister register;
    private StubAccountValidator stuValidator = new StubAccountValidator();

    @BeforeEach
    void setUp() {
        register = new RefundAccountRegister(stuValidator); // 진짜 대신 대역 사용
    }

    @Test
    void invalidAccount() {
        stuValidator.setResult(AccountValidity.INVALID); // 대역에 행동을 지시
        assertThatThrownBy(() -> register.registerRefundAccount("12345", "name", "loginId"))
            .isInstanceOf(InvalidAccountException.class);
    }
}
```

대역

대역 종류

스텁(stub)	구현을 최대한 단순한 것으로 대체
가짜(fake)	기능을 구현해서 진짜와 유사하게 동작(경량 버전)
스파이(spy)	호출된 내역을 기록
모의(mock) 객체	기대한대로 상호작용하는지 행위를 검증 • 보통 모의 객체는 스텁과 스파이 가능

대역

스텁 예

```
public class StubAccountValidator
        extends AccountValidator {
    private AccountVality result;

    public void setResult(AccountVality result) {
        this.result = result;
    }

   @Override
    public AccountValidity validate(String accountNo) {
        return result;
    }
}
```

```
public class RefundAccountRegisterTest {
    private RefundAccountRegister register;
    private StubAccountValidator stuValidator = new StubAccountValidator();

    @BeforeEach
    void setUp() {
        register = new RefundAccountRegister(stuValidator);
    }

    @Test
    void invalidAccount() {
        stuValidator.setResult(AccountValidity.INVALID);
        assertThatThrownBy(() ->
                register.registerRefundAccount("12345", "name", "loginId"))
            .isInstanceOf(InvalidAccountException.class);
    }
}
```

대역

가짜 예

```
public class MemoryMemberRepository implements MemberRepository {
    private Map<String,Member> members = new HashMap<>();

    @Override
    public Member findById(String id) {
        return members.get(id);
    }

    public void save(Member member) {
        member.put(member.getId(), member);
    }
}
```

대역

가짜 예

```
private ConfirmMemberSerivce confirmService;
private MemoryMemberRepository memoryRepository = new MemoryMemberRepository();

@BeforeEach
public void setUp() {
    confirmService = new ConfirmMemberService(memoryRepository);
}

@Test
void confirmMember() {
    // 상황: 대기 상태 회원이 존재
    memoryRepository.save(Member.id("id").status(WAITING).builder().build());
    // 실행: 회원을 승인하면
    confirmService.confirm("id");
    // 결과: 회원이 활성 상태가 됨
    Member m = memoryRepository.findById("id");
    assertThat(m.getStatus()).isEqualTo(ACTIVE);
}
```

대역

모의 예

```
public class AutoDebitRegisterTest {
    private AutoDebitRegister register;
    private CardNumberValidator cardNumberValidator = mock(CardNumberValidator.class);
    private AutoDebitInfoRepository repository = mock(AutoDebitInfoRepository.class);

    @BeforeEach
    void setUp() {
        register = new AutoDebitRegister(cardNumberValidator, repository);
    }

    @Test
    void validCard_Then_Info_Saved() {
        given(cardNumberValidator.validate(anyString())).willReturn(CardValidity.VALID);

        AutoDebitReq req = new AutoDebitReq("user1", "1234123412341234");
        RegisterResult result = register.register(req);

        then(repository).should().save(Mockito.any());
    }
}
```

대역

대역 이점

의존 대상의 진짜 구현 없이 테스트 가능

- 현재 구현 대상에 집중, 병행 개발 가능
- 의존 대상에 대한 상황 지정을 가능하게 함
- 의존 대상에 대한 결과를 확인할 수 있게 함

개발 속도 ↑

- 서버 구동 없이 상당한 기능 검증 가능
- 외부 시스템 연동 없이 주요 로직 검증 가능
- 등등

TDD를 시작할 때 어려운 점

TDD를 시작할 때 어려운 점

업무에 적용

돈 받고 하는 일에 연습없이 TDD 적용하면 안 됨

연습없이 실전도 없음

본인이 망치지 않고 할 수 있는 범위부터 TDD 적용

- 처음부터 모든 걸 TDD로 하겠다는 생각하지 말 것 → 당연히 못 함
- 계산 로직처럼 구조가 단순한 것부터 TDD 시작
- TDD가 익숙해지면 TDD 적용 범위를 넓힐 것

TDD를 시작할 때 어려운 점

테스트 작성 순서

당장 빠르게 구현할 수 있는 것 부터 고민

암호 강도 예: 모든 조건 충족 vs 2개 조건 충족

예외적인 경우 > 정상적인 경우

- 암호강도 검사 예: 입력값이 null인 경우 vs 2개 조건 충족
- 회원 가입 예: 같은 ID 회원이 있는 경우 vs 같은 ID 회원이 없는 경우
- 주문 취소 예: 주문이 이미 취소된 경우 vs 주문이 취소 가능한 경우

예외적인 경우는 코드 구조에 영향을 줌

예외적인 경우를 나중에 하게 될 경우 코드 구조가 복잡해질 수 있음

TDD를 시작할 때 어려운 점

완급 조절

매우 작은 단계씩 점진적으로 진행

- 상수 리턴
- 값 비교
- 구현 일반화

**몇 단계를 한 번에 진행했다가
구현이 막히면
뒤로 돌아와서 천천히 진행**

```
public class PasswordStrengthMeter {
    public PasswordStrength meter(String s) {
        if ("ab12!@A".equals(s))
            return PasswordStrength.NORMAL;
        return PasswordStrength.STRONG;
    }
```

```
public class PasswordStrengthMeter {
    public PasswordStrength meter(String s) {
        if ("ab12!@A".equals(s) || "Ab12!c".equals(s))
            return PasswordStrength.NORMAL;
        return PasswordStrength.STRONG;
    }
```

```
public class PasswordStrengthMeter {
    public PasswordStrength meter(String s) {
        if (s.length() < 8)
            return PasswordStrength.NORMAL;
        return PasswordStrength.STRONG;
    }
```

TDD를 시작할 때 어려운 점

대역 도출 시점

언제 대역을 사용하나?

상황에서
도출

결과 확인
과정에서 도출

기능 구현
과정에서 도출

TDD를 시작할 때 어려운 점

상황 → 의존 도출 → 대역 사용

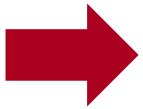
```
@Test
void sameIdExists() {
    동일 ID가 존재하는 상황 필요!

    try {
        svc.register(new RegistReq("id", ...));
        fail();
    } catch(DupIdEx ex) {
    }
}
```

```
@Test
void sameIdExists() {
    memoryRepo.save(new Member("id", ...));

    try {
        svc.register(new RegistReq("id", ...));
        fail();
    } catch(DupIdEx ex) {
    }
}
```

TDD를 시작할 때 어려운 점

결과 검증 → 의존 도출 → 대역 사용

```
@Test
void noSameIdExists() {
    svc.register(new RegistReq("id", ...));

    회원 데이터 생성 확인 필요
}
```



```
@Test
void noSameIdExists() {
    svc.register(new RegistReq("id", ...));

    Member m = memoryRepo.findById("id");
    assertEquals("id", m.getId());
}
```

TDD를 시작할 때 어려운 점

기능 구현 필요 → 의존 도출 → 대역 사용

```
@Test
void pwWeak_Then_Fail() {
    try {
        svc.register(new RegistReq("id", "pw", "ip"));
        fail();
    } catch(WeakPwException ex) {
    }
}
```

```
public void register(RegistReq req) {
    // 암호 등급 검사 기능을 이곳에 구현??
    if ("pw".equals(pw.getPw())) {
        throw new WeakPwException();
    }
}
```

TDD를 시작할 때 어려운 점

기능 구현 필요 → 의존 도출 → 대역 사용

```
PwMeter mockPwMeter = mock(PwMeter.class);

@BeforeEach
void setup() {
    svc = new MemberRegisterSvc(mockPwMeter, repo);
}

@Test
void pwWeak_Then_Fail() {
    given(mockPwMeter.meter("pw")).willReturn(WEAK);
    try {
        svc.register(new RegistReq("id", "pw", "ip"));
        fail();
    } catch(WeakPwException ex) {
    }
}
```

```
private PwMeter pwMeter;
...생성자로 PwMeter 주입

public void register(RegistReq req) {
    if (pwMeter.meter(req.getPw()) == WEAK) {
        throw new WeakPwException();
    }
    ...
}
```

정리

정리

TDD 효과

테스트 코드가 쌓이면 디버깅 시간 감소	테스트 코드가 있으면 문제 범위를 좁혀서 디버깅하는 게 수월
테스트 코드가 쌓이면 코드 변경에 따른 영향 범위 확인 가능	코드를 수정했는데 실패하는 테스트가 발생하면 문제를 빨리 알 수 있음 (회귀 테스트)
코드 구조/설계가 좋아질 가능성이 높아짐	• 테스트가 가능하려면 의존 대상을 대역으로 교체할 수 있어야 함 • 대역으로 교체할 수 있는 구조는 그 만큼 역할별로 분리되어 있을 가능성이 높음

맺음말

“

효과를 얻을 때까지 꾸준한 연마 필요
언젠가 향상된 느낌을 받게 될 것임