7장. 아키텍처 요소 테스트하기

테스트 피라미드

구조

아래로 내려갈수록 테스트의 개수가 많아진다. (=피라미드 형태)

- 시스템 테스트
- 통합 테스트
- 단위 테스트
    - 의존되는 클래스들은 목으로 대체한다.

핵심

- 작은 크기의 테스트들에 대해 높은 커버리지를 유지하자.
    - (= 비용이 적게 드는 테스트를 많이 만들자.)
- 테스트가 비싸질수록 테스트의 커버리지 목표를 낮게 잡아야 한다.
    - 새로운 기능을 만드는 것보다 테스트를 만드는 데 더 오래 걸린다.

단위 테스트

도메인 엔티티 테스트

```
class AccountTest {

    @Test
    void withdrawalSucceeds() {
        AccountId accountId = new AccountId(1L);
        Account account = defaultAccount()
                .withAccountId(accountId)
                .withBaselineBalance(Money.of(555L))
                .withActivityWindow(new ActivityWindow(
                        defaultActivity()
                            .withTargetAccount(accountId)
                            .withMoney(Money.of(999L)).build(
                        defaultActivity()
                            .withTargetAccount(accountId)
                            .withMoney(Money.of(1L)).build())
                .build();

        boolean success = account.withdraw(Money.of(555L), ne

        assertThat(success).isTrue();
        assertThat(account.getActivityWindow().getActivities(
        assertThat(account.calculateBalance()).isEqaulTo(Mone
    }
}
```

유스케이스 테스트

```
class SendMoneyServiceTest {

    // (필드 선언부)
```

```
    @Test
    void transactionSucceeds() {

        Account sourceAccount = givenSourceAccount();
        Account targetAccount = givenTargetAccount();

        givenWithdrawalWillSucceed(sourceAccount);
        givenDepositWillSucceed(targetAccount);

        Money money = Money.of(500L);

        SendMoneyCommand command = new SendMoneyCommand(
                sourceAccount.getId(),
                targetAccount.getId(),
                money);

        boolean success = sendMoneyService.sendMoney(command)
        assertThat(success).isTrue();

        AccountId sourceAccountId = sourceAccount.getId();
        AccountId targetAccountId = targetAccount.getId();

        then(accountLock).should().lockAccount(eq(sourceAccou
        then(sourceAccount).should().withdraw(eq(money), eq(t
        then(accountLock).should().releaseAccount(eq(sourceAc

        then(accountLock).should().lockAccount(eq(targetAccou
        then(targetAccount).should().withdraw(eq(money), eq(s
        then(accountLock).should().releaseAccount(eq(targetAc

        thenAccountsHaveBeenUpdated(sourceAccountId, targetAc
    }

    // (헬퍼 메서드)
}
```

- 행동-주도 개발에서 일반적으로 사용되는 방식인 `given/when/then` 섹션
- 테스트가 코드의 행동 변경뿐만 아니라 코드의 구조 변경에도 취약해질 수 있음

- 코드 리팩토링에도 테스트가 변경될 수 있음
- 테스트에서 어떤 상호작용을 검증하고 싶은지, 핵심만 골라 집중해서 테스트할 것

통합 테스트

웹 어댑터 테스트

```
@WebMvcTest(controllers = SendMoneyController.class)
class SendMoneyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private SendMoneyUseCase = sendMoneyUseCase;

    @Test
    void testSendMoney() throws Exception {
        mockMvc.perform(
            post("/accounts/send/{sourceAccountId}/{targetAcc
                41L, 42L, 500)
                .header("Content-Type", "application/json"))
                .andExpect(status().isOk());

        then(sendMoneyUseCase).should()
            .sendMoney(eq(new SendMoneyCommand(
                new AccountId(41L),
                new AccountId(42L),
                Money.of(500L))));
    }
}
```

- `MockMvc`로 모킹 → 실제 HTTP 프로토콜로 테스트한 게 아님
- `@WebMvcTest` : 스프링이 필요한 전체 객체 네트워크를 인스턴스화하도록 만듦
- 웹 컨트롤러가 네트워크의 일부로서 잘 동작하는지 검증

영속성 어댑터 테스트

```
@DataJpaTest
@Import({AccountPersistenceAdapter.class, AccountMapper.class
class AccountPersistenceAdapterTest {

    @Autowired
    private AccountPersistenceAdapter adapterUnderTest;

    @Autowired
    private ActivityRepository activityRepository;

    @Test
    @Sql("AccountPersistenceAdapterTest.sql")
    void loadsAccount() {
        Account account = adapter.loadAccount(
            new AccountId(1L),
            LocalDateTime.of(2018, 8, 10, 0, 0));

            assertThat(account.getActivityWindow().getActivit
            assertThat(account.calculateBalance()).isEqualTo(
    }

    @Test
    void updatesActivities() {
        Account account = defaultAccount()
            .withBaselineBalance(Money.of(555L))
            .withActivityWindow(new ActivityWindow(
                defaultActivity()
                    .withId(null)
                    .withMoney(Money.of(1L)).build()))
            .build();

        adapter.updateActivities(account);

        assertThat(activityRepository.count()).isEqualTo(1);

        ActivityJpaEntity savedActivity = activityRepository.
```

```
        assertThat(savedActivity.getAmount()).isEqualTo(1L);
    }

}
```

- 단순히 어댑터 로직 검증이 아닌 데이터베이스 매핑도 검증하고 싶기 때문에 통합 테스트가 합리적
- `@DataJpaTest` : 데이터베이스 접근에 필요한 객체 네트워크를 인스턴스화
- `@Import` : 특정 객체가 이 네트워크에 추가됐음을 명확하게 표현
- 스프링은 테스트에서 기본적으로 인메모리 데이터베이스를 사용
  - 프로덕션 환경에서는 인메모리 사용 빈도 낮음
  - 즉 테스트 통과시에도 프로덕션에서 문제가 생길 수 있음
- `TestContainers` 라이브러리로 필요한 데이터베이스를 도커 컨테이너에 띄울 수 있음

시스템 테스트

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class SendMoneySystemTest {

    @Autowired
    private TestRestTemplate restTemplate;

    @Autowired
    private LoadAccountPort loadAccountPort;

    @Test
    @Sql("SendMoneySystemTest.sql")
    void sendMoney() {

        Money initialSourceBalance = sourceAccount().calculat
        Money initialTargetBalance = targetAccount().calculat

        ResponseEntity response = whenSendMoney(
                sourceAccountId(),
```

```
                targetAccountId(),
                transferredAmount());

        then(response.getStatusCode())
                .isEqualTo(HttpStatus.OK);

        then(sourceAccount().calculateBalance())
                .isEqualTo(initialSourceBalance.minus(transfe

        then(targetAccount().calculateBalance())
                .isEqualTo(initialTargetBalance.plus(transfer

    }

    private ResponseEntity whenSendMoney(
            AccountId sourceAccountId,
            AccountId targetAccountId,
            Money amount) {
        HttpHeaders headers = new HttpHeaders();
        headers.add("Content-Type", "application/json");
        HttpEntity<Void> request = new HttpEntity<>(null, hea

        return restTemplate.exchange(
                "/accounts/send/{sourceAccountId}/{targetAcco
                HttpMethod.POST,
                request,
                Object.class,
                sourceAccountId.getValue(),
                targetAccountId.getValue(),
                amount.getAmount());
    }

    // (헬퍼 메서드)

}
```

- `@SpringBootTest` : 스프링이 애플리케이션을 구성하는 모든 객체 네트워크를 띄운다. 느림

- `TestRestTemplate`을 통한 실제 요청 → 실제 HTTP 통신
- 테스트 가독성을 높이기 위해 지저분한 로직은 헬퍼 메서드 안으로 감추자.
- 시스템 테스트는 사용자 관점에서 애플리케이션을 검증할 수 있다.
  - 계층 간 매핑 버그를 수정하기 쉽다.
- 시스템 테스트는 유스케이스를 결합해서 시나리오를 만들 때 빛을 발한다.

테스트를 얼마나 만들어야 할까?

구현할 때는 이라는 문구에 주목 ! 테스트를 귀찮게 만들면 안된다.

- `도메인 엔티티`를 구현할 때는 **단위 테스트**로 커버하자.
- `유스케이스`를 구현할 때는 **단위 테스트**로 커버하자.
- `어댑터`를 구현할 때는 **통합 테스트**로 커버하자.
- `중요 애플리케이션 경로`는 **시스템 테스트**로 커버하자.

결론

- 헥사고날 아키텍처는 로직을 깔끔하게 분리한 덕에 핵심 도메인 로직은 단위 테스트, 어댑터는 통합테스트로 명확하게 테스트 전략을 정의할 수 있다.
- 입출력 포트는 뚜렷한 모킹 지점이다.
- 모킹이 버겁거나 어떤 테스트를 써야할 지 모르겠다면 경고 신호다.