Birim Testi: İleri Uygulamalar

BankService, Mocklama, Davranış Testi ve Hata Senaryoları

Bağımlılıkların Test Edilmesi, Mocklama ve Davranış Doğrulama

Birinci haftada testlerin temelleri üzerinde durduk: izole sınıf, basit iş mantığı, deterministik sonuçlar ve farklı doğrulama biçimleri. Bu hafta, gerçek sistemlerin doğası gereği daha karmaşık hale gelen bir yapı üzerinde çalışacağız. Artık sınıflar birbirine bağımlı, işlemler zincir halinde, hatalar ise bir bileşenden diğerine geçebilir durumda olacak.

Bu bölümde amacımız, yalnızca fonksiyonel doğruluğu değil, bağımlılıklar arası etkileşimi de test etmektir. Bu nedenle iki yeni kavram sahneye çıkıyor: Mock nesneleri ve davranış testi.

2.1 Gerçek Kodun Evrimi: BankService ve NotificationService

İlk haftadaki BankAccount artık sistemin yalnızca bir alt bileşeni.

İkinci hafta, bir banka hizmeti (BankService) geliştirerek birden fazla hesabın etkileşimini ele alıyoruz.

Bu hizmet, para transferi gerçekleştirirken farklı katmanlara çağrılar yapar.

BankRepository

Veri erişimini temsil eden arayüzdür:

```
public interface BankRepository {
    BankAccount findByNumber(String accountNumber);
    void save(BankAccount account);
}
```

NotificationService

Gerçek sistemde bu bir e-posta, SMS veya mobil bildirim olabilir:

```
public interface NotificationService {
    void notify(String accountNumber, String message);
}
```

BankService

Artık sistemin çekirdek mantığı burada toplanıyor:

```
public class BankService {
    private final BankRepository repository;
```

```
    private final NotificationService notifier;

    public BankService(BankRepository repository, NotificationService
notifier) {
        this.repository = repository;
        this.notifier = notifier;
    }

    public void transfer(String fromAcc, String toAcc, double amount) {
        BankAccount src = repository.findByNumber(fromAcc);
        BankAccount dest = repository.findByNumber(toAcc);

        src.withdraw(amount);
        dest.deposit(amount);

        repository.save(src);
        repository.save(dest);

        notifier.notify(fromAcc, "You sent " + amount);
        notifier.notify(toAcc, "You received " + amount);
    }
}
```

Buradaki transfer metodu, artık birden fazla bağımlılığa sahiptir. Bu yapı, testleri hem daha ilginç hem de daha zor hale getirir. Artık yalnızca değerleri değil, davranışların sırasını ve bağımlılık çağrılarının yan etkilerini de doğrulamamız gerekiyor.

2.2 Bağımlılıklar Neden Mocklanır?

Gerçek sistemlerde dış bileşenler (örneğin veritabanı, API çağrısı, ağ iletişimi) testleri yavaşlatır, hata olasılığını artırır ve deterministikliği bozar. Mocklama, bu bileşenlerin davranışlarını simüle etmeye yarar. Test, gerçek nesnelerle değil, davranışını taklit eden nesnelerle çalışır. Mocklar iki tür doğrulamaya imkân verir:

1. **State verification (durum doğrulama):** Kod sonunda belirli bir durum oluşmuş mu?
2. **Behavior verification (davranış doğrulama):** Kod çalışırken beklenen çağrılar yapılmış mı?

Bu iki yaklaşım birbirini tamamlar.

2.3 Mock Kurulumu

JUnit 5 ve Mockito birlikte kullanılır:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

public class BankServiceTest {
    private BankRepository repo;
```

```
    private NotificationService notifier;
    private BankService service;

    @BeforeEach
    void setup() {
        repo = mock(BankRepository.class);
        notifier = mock(NotificationService.class);
        service = new BankService(repo, notifier);
    }

}
```

Burada mock() çağrılarıyla sahte nesneler oluşturulmuştur. Bu nesneler, gerçek sınıfların davranışlarını taklit eder; fakat hiçbir işlem gerçekten yapılmaz.

2.4 Pozitif Senaryo: Başarılı Transfer

```
@Test
void transferShouldMoveMoneyBetweenAccounts() {
    BankAccount src = new BankAccount("A1", 200);
    BankAccount dest = new BankAccount("A2", 100);

    when(repo.findByNumber("A1")).thenReturn(src);
    when(repo.findByNumber("A2")).thenReturn(dest);

    service.transfer("A1", "A2", 50);

    assertEquals(150, src.getBalance());
    assertEquals(150, dest.getBalance());

    verify(repo, times(2)).save(any(BankAccount.class));
    verify(notifier).notify("A1", "You sent 50.0");
    verify(notifier).notify("A2", "You received 50.0");
}
```

Bu test, bir transferin tüm davranışlarını doğrular:

- Bakiyeler doğru güncellenmiştir.
- Repository'ye iki kez save() çağrısı yapılmıştır.
- Bildirimler doğru mesajlarla gönderilmiştir.

Burada test edilen şey yalnızca sonuç (bakiye) değil; aynı zamanda davranışların gerçekleşme biçimidir.

2.5 Çağrı Sırasının Doğrulanması

Bazı durumlarda çağrı sırası önemlidir. Örneğin sistemde kayıt yapılmadan önce bildirim göndermemelidir. Mockito, InOrder sınıfı ile bu tür doğrulamayı destekler.

```
import org.mockito.InOrder;

@Test
void shouldCallDependenciesInOrder() {
    BankAccount src = new BankAccount("A1", 200);
    BankAccount dest = new BankAccount("A2", 100);

    when(repo.findByNumber("A1")).thenReturn(src);
    when(repo.findByNumber("A2")).thenReturn(dest);

    service.transfer("A1", "A2", 50);

    InOrder order = inOrder(repo, notifier);
    order.verify(repo).save(src);
    order.verify(repo).save(dest);
    order.verify(notifier).notify("A1", "You sent 50.0");
    order.verify(notifier).notify("A2", "You received 50.0");
}
```

Sıra testleri, davranışsal tutarlılığın belgelenmesi açısından önemlidir. Fakat burada kritik bir denge vardır: Sıralamayı test etmek, gereksiz kırılganlık yaratabilir. Kodun iç düzeni değişirse testler başarısız olabilir; bu nedenle yalnızca iş mantığı açısından anlamlı sıraları test etmek gerekir.

2.6 Negatif Senaryolar

Bir sistemin güvenilirliğini test etmek, çoğu zaman hatalı durumların beklenen biçimde ele alınıp alınmadığını incelemekle mümkündür.

Yetersiz Bakiye:

```
@Test
void transferShouldFailWhenInsufficientFunds() {
    BankAccount src = new BankAccount("A1", 20);
    BankAccount dest = new BankAccount("A2", 100);

    when(repo.findByNumber("A1")).thenReturn(src);
    when(repo.findByNumber("A2")).thenReturn(dest);

    assertThrows(IllegalStateException.class, () ->
service.transfer("A1", "A2", 50));

    verify(repo, never()).save(any());
    verify(notifier, never()).notify(anyString(), anyString());
}
```

Repository'de Beklenmeyen Hata

Gerçek sistemlerde veritabanı bağlantısı kopabilir, IO hatası oluşabilir. Böyle durumlarda sistemin nasıl tepki verdiğini de sınamak gerekir:

```
@Test
void shouldPropagateRepositoryException() {
    when(repo.findByNumber("A1")).thenThrow(new RuntimeException("DB error"));
    assertThrows(RuntimeException.class, () -> service.transfer("A1", "A2",
10));
}
```

2.7 Yan Etkilerin Yokluğunun Testi

Negatif akışta, sistemin "hiçbir şey yapmaması" da test edilmelidir. Bu kavram, **non-effect verification** olarak adlandırılır:

```
@Test
void shouldNotSendNotificationIfWithdrawFails() {
    BankAccount src = new BankAccount("A1", 10);
    BankAccount dest = new BankAccount("A2", 100);

    when(repo.findByNumber("A1")).thenReturn(src);
    when(repo.findByNumber("A2")).thenReturn(dest);

    assertThrows(IllegalStateException.class, () ->
service.transfer("A1", "A2", 50));
    verify(notifier, never()).notify(anyString(), anyString());
}
```

Bu test, olumsuz koşulda sistemin "pasif" kalıp kalmadığını sınar. Test yalnızca yapılması gerekenleri değil, **yapılmaması gerekenleri** de belgeler.

2.8 Coverage Analizi

Kod kapsama oranı (coverage), testlerin kodun hangi bölümlerini yürüttüğünü ölçer. Ancak yüksek coverage oranı, her zaman iyi test anlamına gelmez. Önemli olan, **mantıksal kapsam**tır:

- Tüm olası yollar (if/else, exception, success) test edilmiş mi?
- Her bağımlılık bir kez bile devreye girmiş mi?

Coverage aracı (ör. JaCoCo) kullanıldığında:

- BankService.transfer metodu için ideal hedef: %100 statement + %100 branch coverage
- BankAccount metodu için: tüm olası hata koşullarının test edilmesi

Öğrenciler, yalnızca yüzdeliklere değil, **testlerin davranış zenginliğine** odaklanmalıdır.

JaCoCo Nedir?

JaCoCo (Java Code Coverage Library), Java projelerinde testlerin hangi satırları, dalları ve sınıfları yürüttüğünü analiz eden bir araçtır. Testlerin yalnızca var olup olmadığını değil, hangi kod yollarını gerçekten test ettiğini gösterir. JaCoCo raporları şu metrikleri içerir:

Metrik	Açıklama
Instruction Coverage	Çalışan bytecode talimatlarının yüzdesi
Branch Coverage	Koşullu ifadelerdeki (if/else, switch) dalların ne kadarının yürütüldüğü
Line Coverage	Kaynak kodundaki satırların ne kadarının testlerce kapsandığı
Method Coverage	Metotların testlerce en az bir kez çağrılıp çağrılmadığı
Class Coverage	Sınıfların testlerde yürütülüp yürütülmediği

Pom.xml içerisine gerekli bağımlılıklar yüklendikten sonra " mvn clean test " komutu çalıştırıldığında "target/site/jacoco/index.html" dosyasına oluşan metrikler raporlanır.

Yanlış yorum: "%100 coverage = hatasız kod."
Doğru yorum: "%100 coverage = tüm yollar yürütülmüş, ancak doğruluğu garanti değil."

Bu nedenle coverage'ı hedef değil, **geri bildirim** olarak kullanmak gerekir. Amaç: "hiç test edilmeyen yollar kalmasın".

2.9 Refaktör ve Testin Dayanıklılığı

Kodda refaktör (örneğin repository'nin iç yapısını değiştirmek) testleri kırmamalıdır. Bu, testlerin uygulama ayrıntısına değil, davranış sözleşmesine bağlı olmasını gerektirir. Testler, kodun *nasıl* çalıştığıyla değil, *ne yaptığıyla* ilgilenmelidir. Bu nedenle iyi bir test, mimari değişimlere rağmen geçerli kalır.

2.10 Alternatif Tasarım Tartışması

Bazı ekipler, BankService.transfer metodunda hataları istisna yerine boolean dönüşüyle ifade eder. Bu durumda testlerin yapısı tamamen değişir: assertThrows yerine assertFalse kullanılır, yan etkisizlik yine verify(never()) ile kontrol edilir.

```
boolean success = service.transferSafe("A1", "A2", 100);
assertFalse(success);
verify(notifier).notify("A1", "Transfer failed due to insufficient funds.");
```