BİRİM TESTİ

Bölüm A — Temel Tanımlar ve İlk Uygulamalar

Birim Testi Uygulamaları: Banka Hesabı Üzerinden Temel Yaklaşımlar

Yazılım geliştirme sürecinde, üretilen kodun doğru çalışıp çalışmadığını anlamanın en sistematik yollarından biri birim testleridir. Birim testleri, yazılımın en küçük mantıksal bileşenlerinin, yani fonksiyon ya da sınıf düzeyindeki yapıların, beklenen şekilde davrandığını doğrulamak için kullanılır. Çoğu zaman geliştiriciler, testleri yalnızca hataları bulmak için yazdıklarını düşünürler. Oysa testlerin asıl işlevi, yazılımın davranışını belgelemek, sürdürmek ve korumaktır.

Bu bölümde, bir banka uygulamasının parçası olan hesap yönetimi modülünü ele alarak, birim testlerinin nasıl kurgulandığını, test senaryolarının nasıl oluşturulduğunu ve testlerin nasıl güvenilir hale getirildiğini adım adım inceleyeceğiz. Örnekler Java programlama diliyle, JUnit 5 test çatısı kullanılarak hazırlanmıştır.

1.1 Banka Hesabı Sınıfının İncelenmesi

Gerçek dünyada bir banka hesabı, belirli bir bakiye tutarına sahip olan, para yatırma ve çekme işlemleriyle bu bakiyeyi değiştiren bir yapıdır. Yazılım dünyasında da bu kavramı benzer şekilde modelliyoruz. Bu modellemede önemli olan, sınıfın yalnızca işlevsel davranışını kodlamak değil, aynı zamanda bu davranışın sınırlarını da açık biçimde tanımlamaktır. Örneğin, negatif tutar yatırılmasına izin vermemek, çekilmek istenen miktar mevcut bakiyeden fazlaysa hataya neden olmak gibi durumlar hem iş mantığının hem de testlerin temelini oluşturur.

```
public class BankAccount {
    private final String accountNumber;
    private double balance;

    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public String getAccountNumber() { return accountNumber; }
    public double getBalance() { return balance; }

    public void deposit(double amount) {
        if (amount <= 0)
            throw new IllegalArgumentException("Invalid deposit amount");
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= 0)
            throw new IllegalArgumentException("Invalid withdraw amount");
        if (amount > balance)
            throw new IllegalStateException("Insufficient funds");
        balance -= amount;
    }
}
```

Bu sınıfın tasarımı, test yazımını kolaylaştırmak amacıyla yalın tutulmuştur. Dışa bağımlılığı yoktur; herhangi bir veritabanı, dosya sistemi veya ağ bağlantısı içermez. Bu, testin saf biçimde yalnızca işlevsel davranışa odaklanmasını sağlar.

1.2 Test Edilebilir Kodun Özellikleri

Bir sınıfın test edilebilir olabilmesi için bazı temel ilkeleri sağlaması gerekir. Birincisi, sınıfın açık ve belirli bir sorumluluğu olmalıdır. BankAccount yalnızca hesap bakiyesini yönetmekten sorumludur. İkincisi, sınıfın deterministik davranması gerekir; aynı girdiler aynı sonuçları üretmelidir. Üçüncüsü, sınıfın dış etkilerden bağımsız olması önemlidir. Kod, dış kaynaklara bağlı değilse, testleri hızlı, güvenilir ve izole biçimde çalıştırmak mümkün olur.

BankAccount sınıfının tasarımında dikkat edilmesi gereken nokta, dış dünyadan tam bağımsız olmasıdır. Örneğin, sınıfın içinde Scanner kullanarak kullanıcıdan veri almak test edilebilirliği düşürür. Çünkü test sırasında kullanıcı girdisi simüle edilemez. Aynı şekilde System.out.println() ifadeleriyle doğrulama yapmak da yanıltıcıdır; test çıktısı, kodun gerçek davranışını yansıtmayabilir. Bu nedenle test edilebilirlik, çoğu zaman "bağımlılıklardan kurtulma" sürecidir. Kodun ne kadar az bağımlılığı varsa, o kadar kolay test edilir.

1.3 JUnit 5 ile Testlerin Yazılması

JUnit 5, testlerin modüler ve okunaklı yazılmasını sağlar. Testler genellikle üç bölümden oluşur: hazırlık (Arrange), davranışın çağrılması (Act) ve doğrulama (Assert). Her test metodu tek bir davranışı ifade etmelidir.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
class BankAccountTest {
    @Test
    void deposit_ShouldIncreaseBalance() {
        BankAccount acc = new BankAccount("A1", 100);
        acc.deposit(50);
        assertEquals(150, acc.getBalance());
    }
    @Test
    void withdraw_ShouldDecreaseBalance() {
        BankAccount acc = new BankAccount("A2", 200);
        acc.withdraw(50);
        assertEquals(150, acc.getBalance());
    }
    @Test
    void withdraw_ShouldThrow_WhenInsufficientFunds() {
        BankAccount acc = new BankAccount("A3", 100);
        assertThrows(IllegalStateException.class, () -> acc.withdraw(200));
    }
}
```

1.4 Hata Durumlarının Test Edilmesi

Gerçek bir sistemde, hatalar yalnızca yanlış veri girişiyle değil, yanlış tasarımla da ortaya çıkabilir. Hata fırlatma davranışlarının test edilmesi, fonksiyonun başarılı çalıştığı durumları test etmek kadar önemlidir. Bu kültür, yalnızca doğru çalışmayı değil, yanlış davranışın da öngörülebilir olmasını sağlar.

1.5 Parametrik Testler ile Geniş Kapsam Sağlama

Aynı işlevi farklı giriş değerleriyle sınamak gerektiğinde parametrik testler kullanılır. JUnit 5, @ParameterizedTest ve @CsvSource ile bu yapıyı destekler.

```
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class ParameterizedBankTest {

    @ParameterizedTest
    @CsvSource({
        "200,50,150",
        "300,100,200",
        "100,0,Exception"
    })
    void testWithdrawScenarios(double start, double amount, String expected) {
        BankAccount acc = new BankAccount("X", start);
        if (expected.equals("Exception"))
            assertThrows(IllegalArgumentException.class, () -> acc.withdraw(amount));
        else {
            acc.withdraw(amount);
            assertEquals(Double.parseDouble(expected), acc.getBalance());
        }
    }
}
```

1.6 Testlerin Yorumlanması ve Kod Kalitesiyle İlişkisi

Testler yalnızca kodun doğruluğunu değil, aynı zamanda tasarım kalitesini de yansıtır. Test yazmakta zorlanıyorsanız bu çoğu zaman sınıf tasarımının iyileştirilmesi gerektiğine işaret eder. Dış bağımlılıkları azaltmak ve tek sorumluluk ilkesine uymak, test yazımını kolaylaştırır.

1.7 Bu Aşamaya Kadar Öğrenilenler

Test, yalnızca bir doğrulama aracı değil; davranışı belgeleyen ve tasarım kalitesini yükselten bir pratiktir. JUnit 5 yapısı, test mantığını sistematikleştirir. Kodun test edilebilirliği, yazılım mimarisiyle doğrudan ilişkilidir. Hata durumlarının test edilmesi, beklenmeyen senaryoları önceden öngörmeye yardımcı olur.

Bölüm B — Derinleştirme, Karşı-Örnekler ve Alternatif Yaklaşımlar

Banka Hesabı Üzerinden Birim Testi: Doğru Kullanım, Karşı-Örnekler ve Alternatifler

2.1 Test Edilebilir Kod ve Bağımlılıklardan Arınma

Test edilebilirlik, erişimden çok kontrol edilebilirliktir. Bağımlılıkların izole edilebilmesi, deterministik davranış ve açık sonuçlar esastır. Kullanıcı girdisi veya sistem saatine doğrudan bağlı metotlar, izole test yazımını zorlaştırır.

2.2 Testlerin Sırası ve Bağımsızlığı

Bir test sınıfı, kodun bir "senaryo akışı" değil, bağımsız gözlemlerinin bir koleksiyonudur. Yani testler birbirine bağlı olmamalıdır. deposit_ShouldIncreaseBalance testi withdraw_ShouldDecreaseBalance testinden sonra çalışmak zorunda olmamalıdır; her biri sıfırdan oluşturulmuş, izole bir durum üzerinde işlem yapmalıdır.

```
@Test
void deposit_then_withdraw_in_one_test_is_confusing() {
    BankAccount acc = new BankAccount("A1", 100);
    acc.deposit(50);
    acc.withdraw(20);
    assertEquals(130.0, acc.getBalance(), 1e-9);
}
```

2.3 assertEquals: Kayan Nokta, Mesajlar ve Toplu Doğrulama

assertEquals ifadesi genellikle en çok kullanılan doğrulama yöntemidir. Ancak bu ifade, bazı durumlarda yanlış güven hissi yaratabilir. Örneğin aşağıdaki iki durum, yüzeyde aynı gibi görünse de test açısından farklıdır:

- `assertEquals(150, acc.getBalance());`
- `assertEquals(150.0, acc.getBalance());`

Java'da double değerlerle yapılan karşılaştırmalarda küçük kayan nokta hataları ortaya çıkabilir. Bu tür hatalar, testin yanlış biçimde başarısız olmasına neden olabilir. Bu nedenle sayısal değerlerde belirli bir toleransla karşılaştırma yapmak gerekir:

- `assertEquals(150.0, acc.getBalance(), 0.0001);`

Birim testlerinde kayan nokta aritmetiği kullanılan tüm fonksiyonlarda bu yaklaşım tercih edilmelidir. Aksi halde test, işlevsel olarak doğru çalışan bir kodu hatalı olarak işaretleyebilir. Ayrıca assertEquals'in fazlaca kullanılması, testlerin **neden** başarısız olduğunu anlamayı güçleştirir. Örneğin şu ifade, hata verdiğinde hatayı bulmak zordur:

- `assertEquals(150, result);`

Oysa açıklayıcı bir mesajla kullanmak, testlerin okunabilirliğini artırır:

```
assertEquals(150, result, "Withdraw işlemi sonrası bakiye hatalı hesaplandı");
```

Parasal değerleri double ile doğrularken küçük yuvarlama hataları testleri haksız yere kırabilir. Tolerans (delta) kullanmak gerekir; başarısızlık mesajları testin belgeleme değerini artırır. Çoklu doğrulamalar için assertAll düzenli raporlama sağlar.

```
assertEquals(150.0, acc.getBalance(), 1e-9,
    "Withdraw(50) sonrası bakiye 150 olmalıydı");

assertAll(
    () -> assertEquals(120.0, acc.getBalance(), 1e-9),
    () -> assertEquals("A", acc.getAccountNumber())
);
```

2.4 assertThrows

Yalnızca istisna türünü doğrulamak yeterli olmaz. Mesajın içeriği ve sistem durumunun değişmemesi de denetlenmelidir.

```
IllegalStateException ex = assertThrows(IllegalStateException.class,() -> acc.withdraw(200));
assertTrue(ex.getMessage().contains("Insufficient funds"));
assertEquals(100.0, acc.getBalance(), 1e-9);
```

2.5 Aynı Davranışın Farklı İfadeleri: İstisna vs Sonuç Tipi

API sözleşmesi test stratejisini belirler. Akış kontrolünü istisnalarla ya da sonuç tipleriyle ifade edebilirsiniz; her birinin test yaklaşımı farklıdır.

```
public final class WithdrawResult {
    public enum Status { OK, INVALID_AMOUNT, INSUFFICIENT_FUNDS }
    public final Status status;
    public final double newBalance;
    private WithdrawResult(Status status, double newBalance) {
        this.status = status; this.newBalance = newBalance;
    }
    public static WithdrawResult ok(double newBalance){ return new
WithdrawResult(Status.OK,newBalance); }
    public static WithdrawResult invalid(){ return new
WithdrawResult(Status.INVALID_AMOUNT, Double.NaN); }
    public static WithdrawResult insufficient(double current){ return new
WithdrawResult(Status.INSUFFICIENT_FUNDS, current); }
}

public WithdrawResult withdrawSafe(double amount) {
    if (amount <= 0) return WithdrawResult.invalid();
    if (amount > balance) return WithdrawResult.insufficient(balance);
    balance -= amount;
    return WithdrawResult.ok(balance);
}

@Test
void withdrawSafe_returns_status_instead_of_throwing() {
    BankAccount acc = new BankAccount("A", 100);
    WithdrawResult r1 = acc.withdrawSafe(50);
    assertEquals(WithdrawResult.Status.OK, r1.status);
    assertEquals(50.0, r1.newBalance, 1e-9);

    WithdrawResult r2 = acc.withdrawSafe(200);
    assertEquals(WithdrawResult.Status.INSUFFICIENT_FUNDS, r2.status);
    assertEquals(50.0, acc.getBalance(), 1e-9);
}
```

2.6 Parametrik Testlerin Dikkatli Kullanımı

Parametrik testler veriyle ölçeklenir; ancak hangi verinin testi kırdığını anlaşılır kılmak için adlandırmayı zenginleştirmek gerekir.

```
@DisplayName("Withdraw senaryoları farklı başlangıç ve miktarlarla")
@ParameterizedTest(name = "[{index}] start={0}, amount={1} -> expected={2}")
```

2.7 Parayı double ile mi, BigDecimal ile mi Tutmalı?

Parasal hesaplarda BigDecimal tercih edilir. Test yaklaşımı da buna göre değişir; delta yerine compareTo kullanılır, ölçek ve yuvarlama açıkça doğrulanır.

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class MoneyAccount {
    private BigDecimal balance;
    public MoneyAccount(String accountNumber, BigDecimal balance) {
        this.balance = balance.setScale(2, RoundingMode.HALF_EVEN);
    }
    public BigDecimal getBalance() { return balance; }
    public void deposit(BigDecimal amount) {
        if (amount == null || amount.signum() <= 0)
            throw new IllegalArgumentException("Invalid deposit");
        balance = balance.add(amount).setScale(2, RoundingMode.HALF_EVEN);
    }
}

@Test
void money_deposit_uses_exact_arithmetic() {
    MoneyAccount acc = new MoneyAccount("M1", new BigDecimal("100.00"));
    acc.deposit(new BigDecimal("0.10"));
    assertEquals(0, acc.getBalance().compareTo(new BigDecimal("100.10")));
}
```

2.8 Yaygın Test Kokuları ve Düzeltmeler

Bir testte birden çok bağımsız davranış; dış dünyaya doğrudan bağımlılık; uygulama ayrıntısına aşırı bağımlı testler; muğlak adlandırmalar. Çözüm: davranış odaklı küçük testler, mock/izolasyon, gözlemlenebilir sonuçların test edilmesi, açık adlandırma.

ÖRNEK BİRİM TESTİ UYGULAMASI

Kuracağımız sistemin adı Order Management System (Sipariş Yönetim Sistemi) olsun.

Sistem Kodları:

```
public class Order {
    private final String orderId;
    private final String productId;
    private final int quantity;
    private final double price;

    public Order(String orderId, String productId, int quantity, double price) {
        this.orderId = orderId;
        this.productId = productId;
        this.quantity = quantity;
        this.price = price;
    }

    public String getOrderId() { return orderId; }
    public String getProductId() { return productId; }
    public int getQuantity() { return quantity; }
    public double getPrice() { return price; }
}
```

```
public interface InventoryService {
    boolean isInStock(String productId, int quantity);
    void reduceStock(String productId, int quantity);
}
```

```
public interface PaymentGateway {
    boolean charge(String orderId, double amount);
}

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

public class OrderProcessor {
    private final InventoryService inventoryService;
    private final PaymentGateway paymentGateway;
    private final NotificationService notificationService;
    private final OrderRepository orderRepository;

    public OrderProcessor(InventoryService inventoryService,
                          PaymentGateway paymentGateway,
                          NotificationService notificationService,
                          OrderRepository orderRepository) {
        this.inventoryService = inventoryService;
        this.paymentGateway = paymentGateway;
        this.notificationService = notificationService;
        this.orderRepository = orderRepository;
    }
    public boolean process(Order order) {
        if (!inventoryService.isInStock(order.getProductId(), order.getQuantity())) {
            notificationService.notify(order.getOrderId(), "Ürün stokta yok.");
            return false;
        }
```

```
        boolean paymentSuccess = paymentGateway.charge(order.getOrderId(), order.getPrice() *
order.getQuantity());
        if (!paymentSuccess) {
            notificationService.notify(order.getOrderId(), "Ödeme başarısız.");
            return false;
        }
        inventoryService.reduceStock(order.getProductId(), order.getQuantity());
        orderRepository.save(order);
        notificationService.notify(order.getOrderId(), "Sipariş başarıyla tamamlandı.");
        return true;
    }
}
```

Birim Tesleri:

```
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 OrderProcessorTest {
    private InventoryService inventory;
    private PaymentGateway payment;
    private NotificationService notification;
    private OrderRepository repo;
    private OrderProcessor processor;
```

```
@BeforeEach
void setup() {
    inventory = mock(InventoryService.class);
    payment = mock(PaymentGateway.class);
    notification = mock(NotificationService.class);
    repo = mock(OrderRepository.class);
    processor = new OrderProcessor(inventory, payment, notification, repo);
}
```

Başarılı Akış Testi

```
@Test
void shouldProcessOrderSuccessfully() {
    Order order = new Order("O1", "P1", 2, 100.0);

    when(inventory.isInStock("P1", 2)).thenReturn(true);
    when(payment.charge("O1", 200.0)).thenReturn(true);

    boolean result = processor.process(order);

    assertTrue(result, "Başarılı sipariş işleminde sonuç true olmalı");
    verify(inventory).reduceStock("P1", 2);
    verify(repo).save(order);
    verify(notification).notify("O1", "Sipariş başarıyla tamamlandı.");
}
```

Bu test, sistemin doğru akışını sınar.

```
import org.mockito.InOrder;
```

```
@Test
void shouldCallDependenciesInOrder() {
   Order order = new Order("O1", "P1", 1, 50);
   when(inventory.isInStock("P1", 1)).thenReturn(true);
   when(payment.charge("O1", 50)).thenReturn(true);

   processor.process(order);

   InOrder inOrder = inOrder(inventory, payment, repo, notification);
   inOrder.verify(inventory).isInStock("P1", 1);
   inOrder.verify(payment).charge("O1", 50);
   inOrder.verify(inventory).reduceStock("P1", 1);
   inOrder.verify(repo).save(order);
   inOrder.verify(notification).notify("O1", "Sipariş başarıyla tamamlandı.");
}
```

inOrder ile sıralamayı control edebiliriz.

Stok Yoksa

```
   @Test
   void shouldFailIfProductOutOfStock() {
      Order order = new Order("O2", "PX", 5, 10.0);
      when(inventory.isInStock("PX", 5)).thenReturn(false);

      boolean result = processor.process(order);

      assertFalse(result);
```

```
    verify(notification).notify("O2", "Ürün stokta yok.");
    verify(payment, never()).charge(anyString(), anyDouble());
    verify(repo, never()).save(any());
}
```

Ödeme Başarısızsa

```
@Test
void shouldFailIfPaymentFails() {
    Order order = new Order("O3", "P2", 1, 150.0);
    when(inventory.isInStock("P2", 1)).thenReturn(true);
    when(payment.charge("O3", 150.0)).thenReturn(false);

    boolean result = processor.process(order);

    assertFalse(result);
    verify(notification).notify("O3", "Ödeme başarısız.");
    verify(inventory, never()).reduceStock(anyString(), anyInt());
    verify(repo, never()).save(any());
}
```

Exception Durumları

```
@Test
void shouldHandleUnexpectedExceptionsGracefully() {
    Order order = new Order("O4", "P9", 2, 200.0);
    when(inventory.isInStock("P9", 2)).thenThrow(new RuntimeException("DB hatası"));
```

```
        assertThrows(RuntimeException.class, () -> processor.process(order));
        // Burada gerçek sistemde belki loglama veya özel exception yönetimi kullanılmalı
    }
}
```

Test Kontrol Tablosu

Durum	Beklenen Sonuç	Yan Etkiler	Test Türü
Başarılı akış	true	reduceStock, save, notify	Pozitif
Stok yok	false	notify(“stokta yok”)	Negatif
Ödeme başarısız	false	notify(“ödeme başarısız”)	Negatif
Exception	Hata fırlatılır	Hiçbiri	Dayanıklılık testi