JUnit nedir?

Java tabanlı kodların test edilmesi için kullanılan bir Unit Test – Birim Testi kütüphanesidir.

Unit Test nedir?

Unit test veya birim testi programları oluşturulan alt parçaların-birimlerin test edilmesidir.

Neden kullanılır?

Programlama dilleri temel olarak giriş-input alarak çıktı-output üretir.

Girişlerin ve çıkışların kaynağı farklı olsa da aynı girişlerin aynı sonuç vermesi beklenir.

Bu beklentiyi doğrulamak için her bir giriş ve çıkış değerinin kontrolünün tek-tek yapılması gerekir.

Her bir giriş ve çıkış değerinin tek-tek yapılması yazılım geliştirme sürecini uzatır.

JUnit gibi araçlar test işlemlerini kolaylaştırır ve çeşitli ek özellikler sunarak bu süreyi kısaltır.

NOT: Unit test yazılan kodların hatalarını bulmak için kullanılmaz.

JUnit ayrıca TDD veya test odaklı geliştirme yapmayı sağlar.

JUnit kullanımı

JUnit kütüphanesinin kullanımı annotations kullanımından ibarettir.

```
public class AppTest {

    @Test
    public void pozitif_deger_dogru() {
        var number = 1453;
        var app = new App();
        boolean expResult = true;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

    @Test
    public void negatif_deger_yanlis() {
        var number = -1453;
        var app = new App();
        boolean expResult = false;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

}
```

Testi çalıştırmak için aşağıdaki komutu komut yorumlayıcısına yazmak yeterli olacaktır.

```
mvn test
```

Örnekte @Test ifadesi ile metodun test olduğu belirtilmiş Assertions sınıfında yer alan assertEquals metodu ile test işlemi yapılmıştır.

NOT: Netbeans, Eclipse ve Intellij gibi IDE özellikleri ile hızlıca oluşturma, yazma ve çalıştırma işlemi yapılabilir.

Assertion method

JUnit5 ile yapılan test işlemleri için Assertions sınıfında yer alan overload edilmiş statik assert metotları kullanılır.

Bazı assert metotları şunlardır;

- assertEquals
- assertNotEquals
- assertArrayEquals
- assertIterableEquals
- assertNull
- assertNotNull
- assertLinesMatch
- assertNotSame
- assertSame
- assertTimeout
- assertTimeoutPreemptively
- assertTrue
- assertFalse
- assertThrows
- fail

NOT: fail metodu test işleminin doğrudan başarısız olduğunu ifade etmek için kullanılır.

Örneğin; Sık kullanılan assertEquals metodu iki değerin eşit olup olmadığını test etmek için kullanılır.

Her bir test öncesi @BeforeEach, test sonrası @AfterEach ile belirtilen metotlar çalışır.

Test bitişinde @AfterAll ile belirtilen statik metotlar çalıştırılır.

```
public class AppTest {

    @BeforeAll
    public static void beforeAll() {
        System.out.println("@BeforeAll");
    }

    @BeforeEach
    public void beforeEach() {
        System.out.println("@BeforeEach");
    }

    @Test
    public void pozitif_deger_dogru() {
        System.out.println("@Test");
        var number = 1453;
        var app = new App();
        boolean expResult = true;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

    @Test
    public void negatif_deger_yanlis() {
        System.out.println("@Test");
        var number = -1453;
        var app = new App();
        boolean expResult = false;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

    @AfterEach
    public void afterEach() {
        System.out.println("@AfterEach");
    }

    @AfterAll
    public static void afterAll() {
        System.out.println("@AfterAll");
    }
```

Yaşam döngülerinin kullanım amacı test işlemi öncesi nesnelerin oluşturulması test işlemi sonrası kaynakların serbest bırakılmasıdır.

Önceki örnekte yer alan nesne oluşturma JUnit yaşam döngüsü ile aşağıdaki gibi yapılır.

```
public class AppTest {

    App app;

    @BeforeEach
    public void beforeEach() {
        app = new App();
    }

    @Test
    public void pozitif_deger_dogru() {
        System.out.println("@Test");
        var number = 1453;
        boolean expResult = true;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

    @Test
    public void negatif_deger_yanlis() {
        System.out.println("@Test");
        var number = -1453;
        boolean expResult = false;
        boolean result = app.isPositive(number);
        Assertions.assertEquals(expResult, result);
    }

}
```

Varsayılan olarak JUnit her bir metot öncesi yaşam döngüsünü TestInstance.Lifecycle.PER_METHOD ile çalıştırır.

Yaşam döngüsünün çalışmasını değiştirmek için test sınıfına @TestInstance(TestInstance.Lifecycle.PER_CLASS) eklemek yeterli olacaktır.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class AppTest {

    Integer myNumber = 0;

    @Test
    public void test1() {
        myNumber += 1;
        System.out.println(myNumber);
        Assertions.assertEquals(1, myNumber);
    }

    @Test
    public void test2() {
        myNumber += 1;
        System.out.println(myNumber);
        Assertions.assertEquals(2, myNumber);
    }

}
```

Test çalıştırıldığında her metot için yeniden yaşam döngüsü çalışmayacaktır.

@DisplayName – Test metodunun IDE ortamında görünen ismini değiştirir.

@Test

@DisplayName("Pozitif değer")

public void test1() {...}

NOT: Sonucun görünmesi için IDE ayarının yapılması gerekebilir.

@Disabled – Testi devre dışı bırakmak için kullanılır.

@Test

@Disabled

public void bir_test() {

// ...}

Test Driven Development Nedir?

Test Driven Development (TDD), kodun ne yapacağını belirlemek ve doğrulamak için test senaryolarının geliştirildiği bir yazılım geliştirme yaklaşımıdır.

TDD yaklaşımının avantajları arasında daha hızlı geri bildirim, yüksek kabul oranı, daha düşük proje kapsamı ve gereğinden fazla mühendislik, müşteri odaklı ve yinelenen süreçler, modüler, esnek ve sürdürülebilir kodlar yer alır.

Test odaklı geliştirme, kodlama, test ve tasarımın birlikte çalıştığı bir programlama tarzını ifade eder.

Basit bir ifadeyle, önce her işlev için test senaryoları oluşturulur ve test edilir. Bu aşamada test başarısız olursa, testi geçmek ve kodu basit ve hatasız hale getirmek için yeniden kod yazılır.

Test Driven Development Testi Nasıl Yapılır?

Test driven development sadece birkaç adımdan oluşmaktadır:

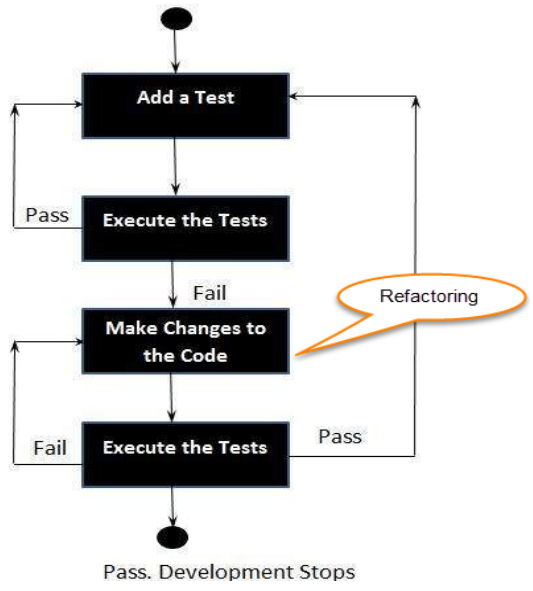
1. Bir test yazılır.

2.Tüm testleri çalıştırın ve yeni bir testin başarısız olup olmadığına bakın.

3. Test başarılı hale getirilir.

4. Mevcut bütün testlerin başarılı olması sağlanır.

5. Refactor kodunu çalıştırın.



TDD vs Geleneksel Test

Test driven development yaklaşımı, kaynak kodunuzun doğrulayıcı düzeyde test edilmesini sağlar. Geleneksel testlerde ise başarılı bir test, bir veya daha fazla kusur bulur.

Test driven development, sisteminizin kendisi için tanımlanan gereksinimleri karşılamasını sağlayarak sisteminiz hakkında güveninizi geliştirmenize yardımcı olur.

Geleneksel testler, test senaryosu tasarımına daha fazla odaklanılır.

TDD'de geleneksel testlerden farklı olarak her kod satırı test edilme sürecinden geçer.

Hem geleneksel test hem de TDD, sistemin test edilmesinin önemine yol açar.

Mockito Nedir?

Mockito bir java kütüphanesidir. Birim testlerde mock(taklit) sınıflar oluşturmamızı sağlar.

Mock kavramı:Gerçek nesneyi taklit ederek kopyalayıp o nesne üzerinde değişiklik yapmamızı sağlar.

Gerekli kütüphaneyi kurduktan sonra mock sınıfları oluşturabiliriz.

Mockito Kütüphanesinin Kullanımı

Mock kütüphanesinden nesne yaratma

Örnek:

Ornek mockSınıf = mock(Ornek.class);

yada farklı bir yöntemle nesne yaratma

```
@Mock
   Ornek mockSınıf;
   @Rule public MockitoRule mockitoRule = MockitoJUnit.rule(
```