JUnit Testi

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

Unit test veya birim testi programları oluşturulan alt parçaların-birimlerin test edilmesidir. Java gibi Nesne Tabanlı programlama dillerinde bu parçalar metot olarak adlandırılır.

Kullanım Amacı :

- 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.

JUnit kurulumu

JUnit 5 ile birlikte JUnit Platform, JUnit Jupiter, JUnit Vintage gibi alt parçalara ayrılarak platform haline gelmiştir.

Maven projesi oluşturalım.

```
mvn archetype:generate \
-DgroupId=com.patikadev \
-DartifactId=JUnitApp \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DinteractiveMode=false
```

JUnit kütüphanesini **pom.xml** dosyasına ekleyelim.

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter</artifactId>
   <version>5.7.0</version>
   <scope>test</scope>
</dependency>
```

JUnit testelerini Maven ile çalıştırmak için **maven-surefire-plugin** eklentisini ekleyelim.

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>3.0.0-M5</version>
      </plugin>
   </plugins>
</build>
```

Örnek sınıf ve metot aşağıda yer almaktadır.

```
public class App {

   public static void main(String[] args) {
      System.out.println("Hello JUnit!");
   }

   public boolean isPositive(Integer number) {
      return number > 0;
   }

}
```

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.

Junit5 Testi

Unit test genel olarak; bir metot (ya da fonksiyonun) kendi içerisindeki akışın doğru çalışıp çalışmadığı bilgisini developer' a gösteren bir testtir. Bu testi son kullanıcı testleri ile ayrı tutmamız lazım. Aslında ifade etmeye çalıştığım şey: bu test developer testidir. Kod doğru çalışıyor mu diye developer' in geliştirme esnasında oluşturduğu ve kodunu kontrol ettiği testtir. Monolotik olarak metot bazında yapılır.

Özetle Unit testin amacı teknik olarak metot(ya da fonksiyonun) herhangi bir exception olusturmadan hatasız calısıp, beklenen girdilere karşın beklenen sonucu verdi mi? sorunsalı için yazılır.

JUnit5 Annotationlar (Anotasyonlar)

@Disable: Test sınıfını ya da metodunu devre dışı bırakmak için kullanılır. (JUnit4'te @Ignore)

@AfterAll: Tüm test metodları çalıştırıldıktan sonra çalıştırılacak olan metodu belirtir.

@BeforeAll: Tüm test metodlarından önce çalıştırılacak olan metodu belirtir.

@AfterEach: Her test metodundan sonra çalıştırılacak olan metodu belirtir. (JUnit4'te @After)

@BeforeEach: Her test metodundan önce çalıştırılacak olan metodu belirtir. (JUnit4'te @Before)

@ExtendWith: Custom extension'ları eklemek için kullanılır. (Mesela spring boot extension' u için: @ExtendWith(SpringExtension.class))

@Tag: Testleri filtrelemek için Tag ekler.

@DisplayName: Sınıf ya da metodların isimlerinin test sonuçlarında nasıl görüneceğini belirler.

@Nested: Bu annotationu alan sınıfın bir nested sınıf yani non-static sınıf olduğunu belirtir.

@TestFactory: Metodun dinamik test olduğunu belirtir.

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.

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

TDD Testi nasıl yapılır

1. Bir test ekleyin.
2. Tüm testleri çalıştırın ve herhangi bir yeni testin başarısız olup olmadığına bakın.
3. Biraz kod yaz.
4. Testleri ve Refactor kodunu çalıştırın.
5. Tekrar et.

Bu bağlamda önceden kodu yazıp sonradan bunların testini yazmanın TDD olamadığını özellikle belirtmek gerekmektedir. TDD'de mutlaka ama mutlaka test ilk sırada yazılmalıdır.

TDD Avantajları

- TDD ile test kapsama alanı oldukça geniştir.
- Testler koda olan güveni artırır.
- Testler sistemin nasıl çalıştığını gösteren dokümantasyon olarak düşünülebilir.
- Design first mentalitesi kazandırır ve over-engineeringten kaçınmamızı sağlar.

TDD ile Geliştirme Yaparken Dikkat Edilmesi Gerekenler

- Test kodlarımız normal kodlar gibi ele alınmalı gelişime açık kodlar tasarlanmalıdır.
- Testler yazılırken negatif caseler'e de önem verilmeli.
- Testler bir işin nasıl yapıldığından çok ne yapılmaya çalışıldığıyla ilgilenmelidir.
- Testlerin çalışma sırası birbirini etkilememelidir.

Spring Profile