Bu yazıda sırasıyla Unit Test, JUnit 4 ve 5, Test-Driven Development (TDD), Mockito ve son olarak Spring Profile hakkında genel bilgiler verilecektir.

Unit Test

Unit Test, yazılımcıların uygulama geliştirme aşamasında kullandıkları kodların en küçük alt birimlerini (sınıf, fonksiyon, metod) test etme yöntemidir. Unit test yazmadaki amaç her birimin beklendiği gibi çalıştığını doğrulamaktır. Unit testler projenin geliştirilmesi ve değişiklik durumlarından etkilenmemesi için bağımsız bir yapıda olmalıdırlar. Unit Testlerde yazılan testin ne yaptığını anlayabilecek bir şekilde isimlendirme yapılmalıdır.

Unit Test Avantajları:

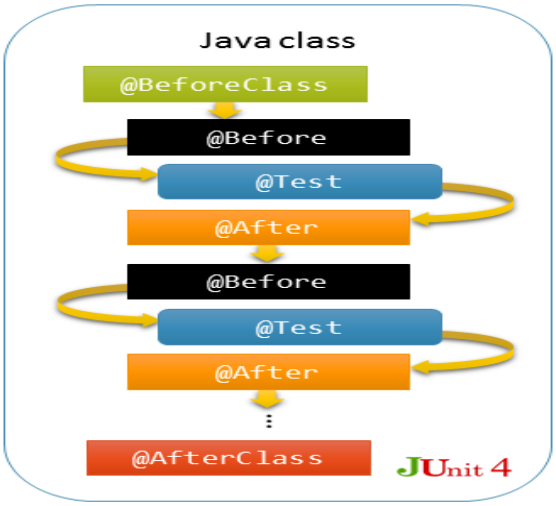
- Modüler yapısı sayesinde uygulamamız tam olarak bitmeden test edebilme fırsatı sağlar.
- Unit testler küçük kodlar oldukları için hataları daha rahat görüp giderebiliriz.
- Kodun yeniden kullanımına ve bağımlılıklarımızı azalmamıza yardımcı olurlar.
- Dokümantasyon görevi görebilirler

Unit test bir test framework aracılığıyla çalıştırabilir. Java kodlarını test etmek için JUnit kütüphanesi kullanılabilir. JUnit ile kolayca test senaryoları yazabilmemiz için yerleşik şablon ve raporlama özelliği vardır. Biz genel hatları ile JUnit 4 ve JUnit 5 versiyonlarını inceleyeceğiz.

JUnit 4

Bu yazıda sırasıyla JUnit4, JUnit4 Anotasyonları ve Assertions hakkında genel bilgiler verilecektir. JUnit 4, uygulamalarımızı test etmek için kullandığımız birim test çerçevelerindendir. JUnit 4, metotlarda kullanılan bazı Annotations (ek açıklamalar) sahiptir. Bunlar testleri çalıştırmamızı ve anlamamıza olanak sağlar. Üzerlerinde bazı konfigürasyonlar yapabiliriz.

JUnit 4 Anotasyonları:



@Test: public void method() olan metotları test etmek için kullanılır.

@Before: Her testten önce ön koşulları belirtmek için public void method() olan metotlarda kullanılır.

@After: Her testten sonra public void method() olan metotlarda çalıştırılır. Geçici verileri silmek ve varsayılan ayarları geri yüklemek için kullanılır.

@BeforeClass: Tüm testler başlamadan önce public static void method() olan metotlarda bir kez çalıştırılır. Veritabanı bağlantıları için kullanılır.

@AfterClass: Tüm testler bittikten sonra public static void method() olan metotlarda bir kez çalıştırılır. Veritabanı bağlantısı kesmek için kullanılır.

@Ignore: Bir testin çalıştırılmasını geçici olarak devre dışı bırakmak istediğinizde public static void method() olan metotlarda kullanılabilir.

@Test (expected = Exception.class): Verilen exception atılmadığı durumlarda test başarısız olur.

@Test(timeout=500): Testimiz 500 milisaniyeden uzun sürerse başarısız olur.

JUnit 4 Assertions:

Uygulamalarda belirli koşulları test etmek için kullanılır. Test çalıştıktan sonra "Error Message", "Expected Value", "Actual Value" ("Hata Mesajı", "Beklenen Değer", "Gerçek Değer") şeklinde anlamlı hata mesajları olarak belirtilebilir.
Beklenen değer ile gerçek değer karşılaştırılır ve değerler eşleşmiyorsa "AssertionException" hata mesajı döndürür.

JUnit 4 Assertions Kullanımları:

void assertEquals ("Message", boolean expected, boolean actual): Verilen iki değerin aynı olup olmadığını karşılaştırır.

void assertTrue("Message", boolean condition) : Boolean ifadesinin doğruluğunu kontrol eder.

void assertFalse("Message", boolean condition) : Boolean ifadesinin yanlış olup olmadığını kontrol eder.

void assertNull("Message", Object object) : Nesnenin null olup olmadığını kontrol eder

void assertNotNull("Message", Object object) : Nesnenin boş olmadığını kontrol eder

void assertSame("Message", boolean condition) : "İki nesnenin aynı nesneye atıfta bulunup bulunmadığını kontrol eder."

void assertNotSame(boolean condition) : “İki nesnenin aynı nesneye gönderme yapmadığını kontrol eder.”

***Örnek test kullanımını göstermek için github JUnit 4 developer guide bölümünden alınan bir örnek:*

Calculator.java

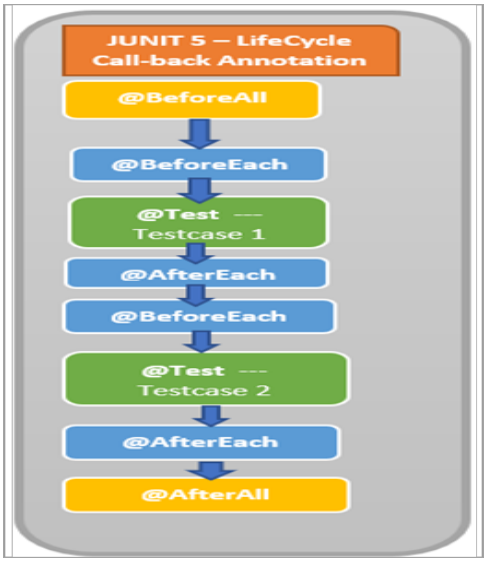
```java
package com.developersguide.junit;

public class Calculator {
        public int evaluate(String expression) {
                int sum = 0;
                for (String summand : expression.split("\\+"))
                        sum += Integer.valueOf(summand);
                return sum;
        }

}
```

CalculatorTest.java

```java
package com.developersguide.junit;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class CalculatorTest {
        @Test
        public void evaluatesExpression() {
                Calculator calculator = new Calculator();
                int sum = calculator.evaluate("1+2+3");
                assertEquals(6, sum);
        }
}
```

JUnit 5

JUnit 4 ‘ün üst versiyonlarından olan JUnit 5, JUnit Jupiter içerisinde yer alan yeni uzantılar ve kütüphanelere sahip test çerçevelerinden biridir. JUnit 5 bazı anotasyonları ve açıklamaları aşağıdaki gibidir:



@BeforeAll: Test sınıfındaki tüm test yöntemleri çalıştırılmadan önce bir kez çalıştırılır.

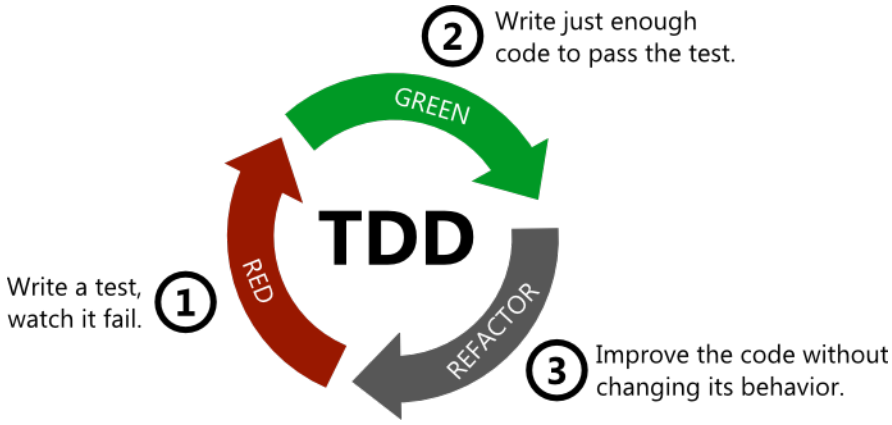
@BeforeEach: Her bir test senaryosunun çalışmasından önce çalıştırılır.

@Test: Test senaryosunun yapıldığını belirtir ve JUnit 4 ‘ten farklı olarak herhangi bir erişim belirleyici(public) belirtmez.

@AfterEach: Her test senaryosu çalıştıktan sonra çalıştırılır.

@AfterAll: Test sınıfındaki tüm test senaryoları tamamlandıktan sonra bir kez çalıştırılır.

Test-Driven Development (TDD)



Test-Driven Development basit ve yalın kod yazmayı, kodda refactoring yapmayı kolaylaştıran ve bağımlılıkların az olmasını sağlayan yazılım geliştirme yaklaşımlarından biridir. Birim testleri ile metodların görevini yerine getirip getirmediğini test eder.

TDD uygularken izlenecek bazı adımlar vardır. Amacımız adımları sürekli tekrar ederek daha anlaşılır iyi bir kod elde etmektir.

Adım 1: Testler kodlardan önce yazılmalıdır. İlk önce testlerin başarısız olduğu görülmelidir. Testi geçemediğimiz için kırmızı renk döner.
Nedeni: Uygulamadaki özellikler ve gereksinimler tam olarak kavramak için önce testi yazıp başarısız olmalıyız. Birinci adım bir yandan kodu tasarlamamıza ve gereksiz kodlardan arındırıp daha sade kod yazmamıza olanak sağlar.

Adım 2: Yeterli düzeyde kod yazılıp, test geçilmelidir. Testi geçtiğimizi için yeşil renktir.
Nedeni: İleride yazılacak durumların kodlamasının önceden yazılması istenmez. Yazılan kodla birim testi yapılmış kodların geçmesi beklenir. Az kodla test yapılır ve istenilen çıktı için test geçilmiştir.

Adım 3: Kod kalitesi arttırılmalı ve gereksiz kodlardan kurtulup test geçilmelidir. Refactoring yapılmalıdır.
Nedeni: Uygulama kodlarının ilerisi için daha test edilebilir hale getirip bağımlılıkları azaltarak gereksinimlerimize uygun sınıf ve metodların kalması amaçlanır.

TDD Faydaları:

- Önce tasarımı düşündürür. Yazılımın gereksinimleri birim testler sürekli çalıştığından kontrol altındadır.
- Yazılım geliştirmeyi kolaylaştırır ve proje maliyetini azaltır.
- Gereksiz kod ve karmaşıklığı azaltmayı amaçlar.
- Yazılım üzerinde mevcut kodların özellikleri bozulmadan rahatça değişiklik yapılabilir.

MOCKITO

Unit test yaparken bağımlı olduğumuz objelerin sahtelerinin yaratılmasını sağlayan Java tabanlı bir kütüphanedir. Sahte nesnelere örnek vermemiz gerekirse projemizde test için veritabanından sürekli bir nesne çağırmamıza gerek yoktur. Yeni bir kullanıcı eklemek istediğimizde yazılımcı test sırasında kullanıcı bilgileri için değerler gönderip, test sonucunun her zaman aynı olduğu mock nesnesi oluşturabilir. Bu objelerin davranışını istediğimiz gibi belirleyebiliriz. Aksi halde her bağımlığımızda somut bir mock objesi oluşturmamız gerekirdi. Asıl hedefimiz bağımlılıklardan kurtulup programımızın düzgün çalışmasını sağlamaktır.

Mockito private methodlar için JUnit 4'te PowerMockito.mock kütüphanesini kullanır.
Public methodlar için JUnit 5'in Mockito.mock kütüphanesi kullanılır.
@Mock annotation ile mocklayacağımız nesneyi tanımlayabilir ve tüm sınıfta kullanabiliriz.

Sonuç olarak Mockito ile karmaşık nesnelerden kurtulabiliriz. Henüz mevcut olmayan nesnelerde kullanabiliriz. Zaman ve maliyet tasarrufu sağlayabiliriz.

Spring Profile

Uygulamalarımız birçok farklı ortamlarda çalışabilir. Bu ortamlar localhost, test, hazırlama ve üretim ortamları olabilir ve bu ortamlara yapılandırma özellikleri sağlarız. Uygulamamız farklı ortamlarda farklı konfigürasyonlara gerek duyabilir. Bunlar veritabanları, mesajlaşma sistemleri, sunucu portları, güvenlik gibi konfigürasyonları ortamdan ortama farklılık gösteren ayarlara sahip olabilir. Örnek olarak uygulamayı localhostumuzda geliştirirken veritabanı için H2 veritabanı kullanabiliriz ama üretim aşamasında MySQL veritabanı kullanmamız gerekebilir. Uygulamamızı localhost ve üretim aşamasında çalıştırdığımızda farklı veritabanı bağlantı yapılandırmalarına sahip oluruz. Ortamdan ortama konfigürasyonları yönetmek çok uğraştırıcı olabilir. Bu nedenle Spring Profile doğru ortamlarda doğru konfigürasyon ayarları yapmak için kullanabiliriz.

Kaynaklar

1. https://siberci.com/unit-test-nedir/
2. https://www.swtestacademy.com/junit4/
3. https://junit.org/junit5/docs/current/user-guide/
4. https://www.softwaretestinghelp.com/junit-annotations-tutorial/
5. https://dergi.bmo.org.tr/teknoloji/test-gudumlu-programlama
6. DÖKER, A., KİREMİTÇİ, Ö., & KILINÇ, D. TGG (Test Güdümlü Geliştirme) ve Birim Testleri.
7. https://reflectoring.io/clean-unit-tests-with-mockito/#injecting-mocks-with-spring
8. https://bilisim.io/2021/01/11/spring-boot-profiles/
9. https://springframework.guru/spring-profiles/