JUNIT TEST

Unit Platform: Bize sağladığı olanak, **JUnit4** ve **Junit5** ile birlikte çalışmayı sağlayan bir platformdur. JUnit Platform ile çalışmamızın sebebi projede private ve public methodları birlikte test etmeyi sağlar. Private methodlar için PowerMockito kullanıyoruz(JUnit4 te çalışıyor). Public methodlar için Mockito kullanıyoruz(JUnit5 te çalışıyor)

JUnit Vintage(JUnit4) : JUnit4 bize Private methodlarda Powermockito kullanarak database ile bağlantımızı kesip mocklamamızı sağlıyor.

JUnit Jupiter(JUnit5) : JUnit5 Public methodarda kullanacağımız nesneyi mocklayarak testimizi yazamaya olanak sağlar.

JUNIT4 Useful annotations

@Before(): Test durumundan önce oluşturmak istediğimiz kaşulları bu annnotationu yazıp oluşturuyoruz.

@BeforeClass():Tüm testler için önceden yapılması gereken işlemleri yaparız.

@After(): Değişkenleri sıfırlamak, geçici dosyaları silmek, değişkenleri silmek gibi her Test Durumundan sonra bazı ifadeleri çalıştırmak istiyorsanız bu açıklama kullanılabilir .

@AfterClass():Tüm test durumarından sonra çalıştırmak istediklerimizi yazarız.

@Test(): Public methodlar için test oluşturacaksak bu annotationu kullanırız ama private methodlar için oluşturacaksak **@org.junit.Test** annotationunu ekleriz. **@Test(timeout=500)** eğer süreli test edeceksek ve testin exceptionlardan etkilenmesini istemiyorsak **@Test(expected=IllegalArgumentException.class)** bu annotationu kullanırız.

@Ignore():Üzerine koyduğumuz testi çalıştırmaz.

JUNIT5 Useful annotations

@Test — Bunu bir test method olarak işaretler ve test plugini ile çalışmasını sağlar

@TestFactory — Private veya static methodlar için kullanılır, Stream, collection, iterable'a ait dinamik testler yazılır.

@DisplayName — Test isimlerinin yazılması için kullanılan bir annotationdur. @DisplayName("MyClass")

@BeforeAll/@BeforeEach — lifecycle method içinde yer alır. Her @Test, @RepeatedTest, @ParameterizedTest, @TestFactory annotationu gerçekleşmeden önce bu methodların içinde belirtilen classlar yeniden yaratılır.

@AfterAll/@AfterEach — lifecycle methods içinde yer alır. Test yöntemleri uygulanıktan sonra yapılacaklar bu methodlar altında yer alır.

@Tag -Testi bir özellik ile belirtmek istiyorsak bu kullanılır. @Tag("fast")

@Disabled –Kullanılmayan testlerde testi gçmesi için kullanılır.

@Nested- Testlerin sırasını control etmek için kullanılır.

@ExtendWith- Koşullu testler yapacaksak bu annotation kullanılır.

MOCKITO

Mockito, JAVA uygulamalarının birim testi için kullanılan, JAVA tabanlı bir kütüphanedir. Mockito, arayüzleri taklit etmek için kullanılır, böylece birim testlerinde kullanılabilecek sahte arayüze sahte bir işlevsellik eklenebilir. Java reflection'ı kullanırız.

*Mock objesini public methodlar için oluşturuyosak JUnit5'in bize sunduğu Mockito.mock'u kullanırız.

*Mock objesini private methodlar için çekeceksek JUnit4'un bize sunduğu PowerMockito.mock kütüphanesinden çekeriz.

Bunun için;

@Mock gibi bir annotation oluşturup altına mocklayacağımız nesneyi tanımlayabiliriz. Tüm classta kullanabiliriz.

Mock objesi yaratılması aşağıdaki şekildedir; Public methodlar için bu şekilde mock nesnesi oluşturulur. ServiceUtil mServiceUtil = **Mockito**.**mock**(ServiceUtil.class); Oluşturduğumuz mock objesine davranış atarken ise; doReturn(exampleList).when(mock objesi).(aldığı method).(parametreleri);

PowerMockito ile mock objesi aşağıaki şekilde yaratılır, private methodlarda bunu kullanırız.

EmployeeService mock = PowerMockito.mock(EmployeeService.class);
Oluşturduğumuz PowerMock objesine davranış atarken ise yapı biraz değişebilir;
WhiteBox.invoke(mock,”Kullandığımız method”,parametreler);

Test Güdümlü Geliştirme (Test-Driven Development) Nedir?

TDD olarak kısaltılır. Extreme programming'in bir parçası olarak icad edilen bir yaklaşımdır. "Test-first" (önce test) ilkesi üzerinden disiplin haline getirilmiştir**.**

TDD yaklaşımı ana hatlarıyla 5 aşamadan ibarettir:

1)Düşün ve test-case oluştur.

2)Birimi geliştirmeden önce programcı testinin başarısız olmasını sağla.

3)Birimi geliştir ve programcı testini başarılı hale getir.

4)Önceden oluşturulmuşlarla birlikte tüm programcı testlerinin başarılı olmasını sağla.

5)Son geliştirdiğin ile birlikte tüm birimleri tarayarak gerekli noktalarda refactoring yap.

Geliştirmeyi yapıp sonradan testini yapmak TDD değildir. TDD'de ilk sırada, fail veren bir test birimi yazmak vardır. Sonradan yazılan programcı testleri de çok faydalıdır fakat bu TDD'nin konusu değildir.

Peki TDD Ne Tür Faydalar Getirir?

Çoğunlukla, programcı testinin (ya da birim testinin) uygulama geliştirme sürecini uzatacağı düşünülür. Bu yanlış bir kanıdır. Programcı testlerinin tasarıma başlarken yazılması tasarımı kolaylaştırır. Kolay olan şey genelde kısadır.

Projedeki iş mantığı, koda tam olarak yansır. Özellikler, yetenekler, kısıtlar, sınırlar programcı testleri aracılığı ile code-base'e çerçeve olarak yansıtılır. Çerçevenin dışına çıkma söz konusu olduğunda kimse bağırmasa, birim testi bağıracaktır.

Sonradan ortaya çıkacak bug'lar, olumlu birim testleri yanında olumsuz birim testleri sayesinde önceden farkedilir. Bu da her tür zarardan erken dönmek demektir. Ayrıca bu yine zamandan da tasarruf demektir.

İşlevselliğin birinde yapılan değişiklik, diğer işlevsellik(ler)in davranışını bozsa bunun tespitinin çok kısa sürede olması muhtemeldir. Bunun için genelde birimler arası bağlaşıklığın test-case'de tamamen kaldırılmamış olması yeterlidir.

Projenin bakımı (maintenance) esnasında bug avına çıkıldığında, programcı testleri çalıştırıldığı an o bug için çember bir hamlede dramatik olarak daralmış olur.

TDD, top-down yaklaşımını getirir. Yani tasarımcıyı dıştan içe tasarıma, yani en üst seviye katmandan en alt seviye katmana doğru bir tasarıma çeker. Nesne yönelimli yaklaşım ise geliştiriciyi bottom-up yaklaşımına iter. Bottom-up yaklaşımı "önce nesneleri belirle" demektir. Nesne yönelimli yaklaşım, önce alt katmanlar ile uğraşmaya iter. Oysa nesne yönelimli programlamada top-down ve bottom-up yaklaşımları beraber kullanılabilir, kullanılmalıdır. Bir yandan interface'ler, contract'lar şekillendirilirken bir yandan test-case'ler ortaya çıkarılabilir. Mimarinin her katmanında aynı noktaya hem 0 derece hem 180 derece açıdan bakılarak TDD ve OOP birleştirilebilir.

Özetle;

TDD, tasarımı basitçe yapmamızı, basit ve yalın kodlama yapmamızı, düzenli refactoring yapmamızın kolaylaşmasını ve gevşek bağlaşıklığı sağlayan bir yaklaşımdır.

Spring Profile

Profile anotasyonu uygulamamızdaki farklı çalışma isterlerine göre programımızın hangi işlevinin çalışacağını çalışacağını isteğimize göre seçmemizi sağlamaktadır.

Ayrıca uygulamamızın geliştirme ortamlarına göre bildiride bulunmamızı sağlar.

Örneğin: dev, test, prod gibi.

İlgili class'ın üzerine @Profile("profil-adı") şeklinde belirtilir. Farklı farklı profillerimiz olabilir.

@Profile ile işaretleme yaptıktan sonra, **application.properties** içinde bir bildiride bulunmamız gerek.

spring.profiles.active= profil-adı