JUnit ile Birim Test Geliştirme

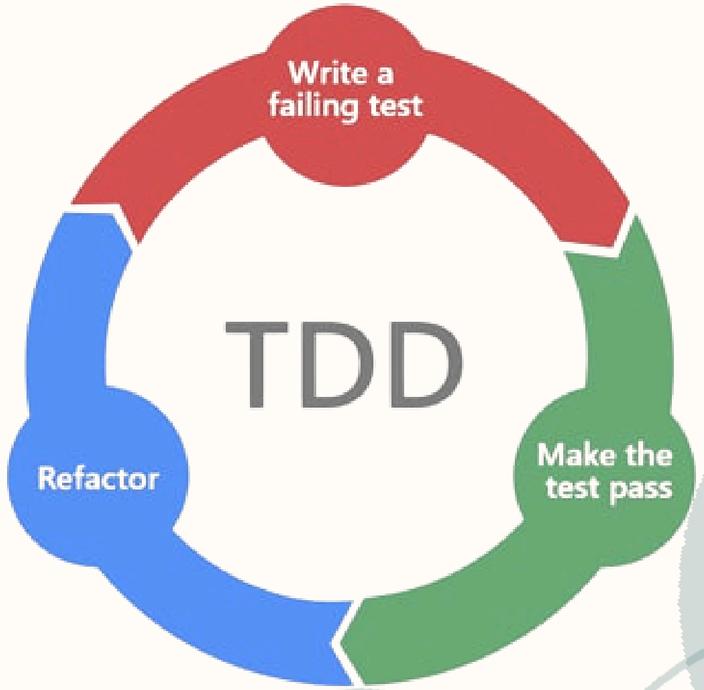
Mert Alptekin
Lead Software Consultant

Eğitim Kataloğu

1. Test Odaklı Yazılım Geliştirme (TDD) Nedir?
2. Yazılım Test Türleri
3. Birim Testlerin Yazılıma Etkisi
4. Birim Testlerin Temel Yapısı
5. JUnit Nedir, Ne İşe Yarar?
6. JUnit 4 ve JUnit 5 Arasındaki Farklar
7. JUnit Annotations Kavramı
8. JUnit Assertions Kavramı
9. Exception Testleri
10. Test Suites Kavramı
11. Kod Kapsamı Analizi (Run with Coverage)
12. Mock Test Kavramı

TDD (Test-Driven Development)

TDD, “önce test yaz, sonra kodu yaz, sonra temizle” döngüsünü izleyen bir geliştirme pratiğidir. Amaç sadece doğru çalışan kod değil; iyi tasarlanmış, güvenilir ve sürdürülmesi kolay kod üretmektir.



TDD Döngüsü

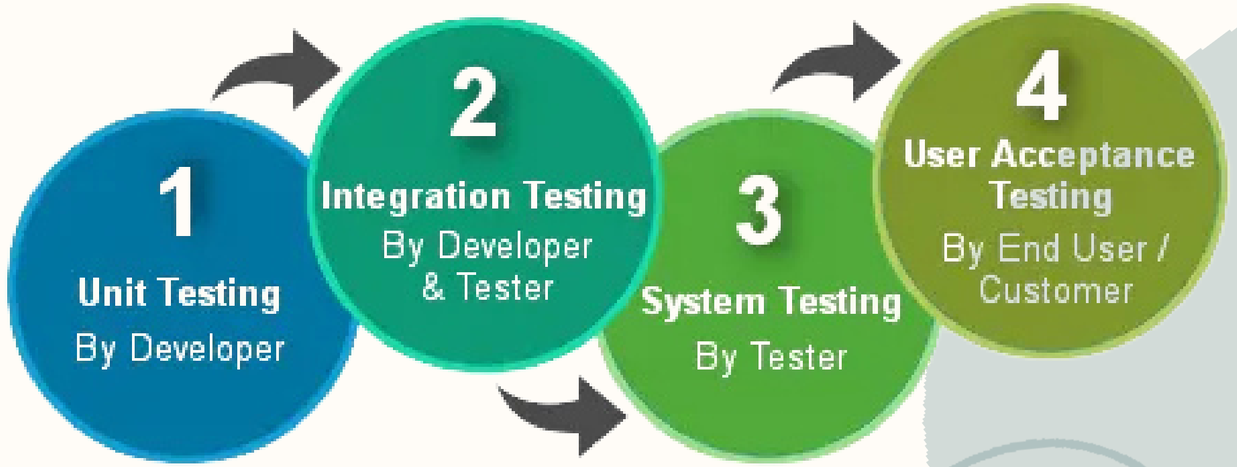
Red (**Fail**) **:** Test kodu yazılır, ama üretim kodu henüz yok veya hatalıdır. Bu noktada test fail olacaktır.

Green (**Pass**)**:** Testi geçirecek en basit kodu yaz. Hedef: testi hızlıca yeşile çevirmek

Refactor: Hem üretim kodunu hem test kodunu düzelt. Testler hep yeşil kalmalı.

Yazılım Test Türleri

Yazılım Testi, bir sistemin veya sistemin bileşenlerinin belirtilen gereklilikleri karşılayıp karşılamadığını öğrenmek amacıyla yapılan bir değerlendirme sürecidir. Bu süreç, yazılım geliştirme uzmanları tarafından başlayan ve son kullanıcıya kadar uzanan teknik seviyelerden oluşur.



Yazılım Test Türleri

Test Türü	Kapsam	Amaç
Unit Test	**Tekil metod/sınıf**	**Doğruluk**
Integration Test	**Modüller arası etkileşim**	**Uyum**
System Test	**Tüm sistem**	**İşlevsel bütünlük**
Acceptance Test	**Kullanıcı gereksinimleri**	**Onaylama**

Yazılım Testlerinin Amacı

- Hataları erken aşamada tespit etmek
- Beklenen davranışın doğrulandığından emin olmak
- Yazılımın bakımını kolaylaştırmak

Test, sadece hata bulmak değil; **doğruluk**, **kararlılık** ve **sürdürülebilirliği** sağlamaktır.

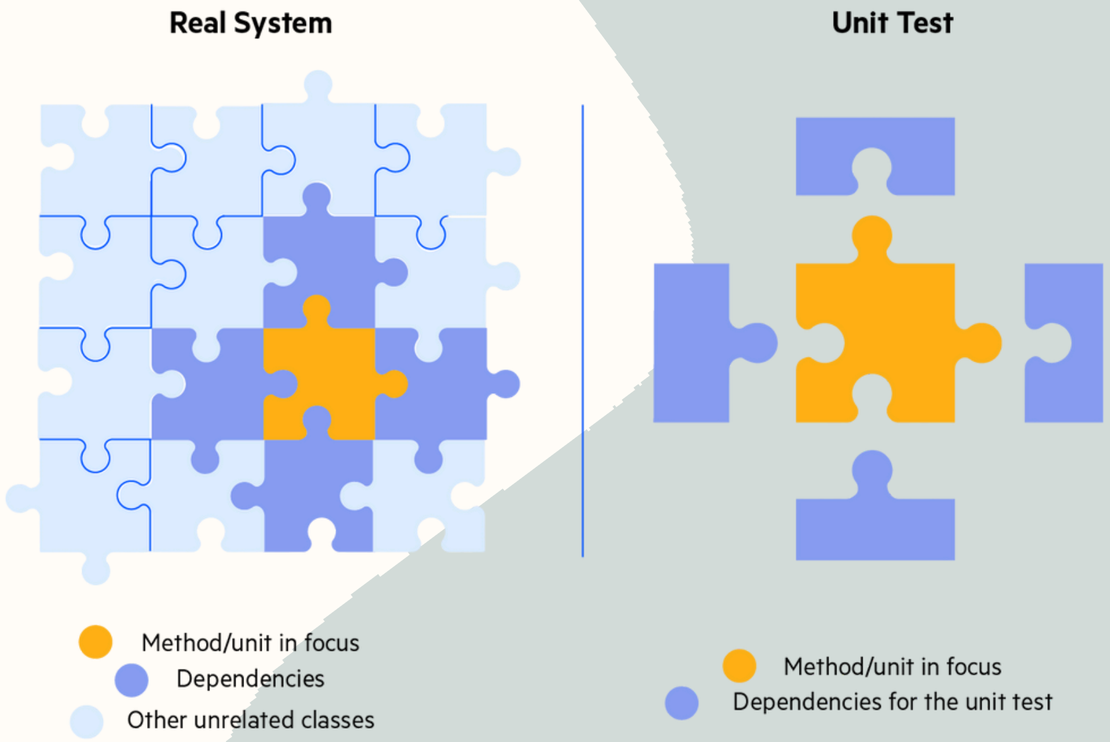
Birim Testleri

Yazılımın en küçük mantıksal parçalarını (**metot**, **fonksiyon**, **sınıf**) test etme yöntemidir.

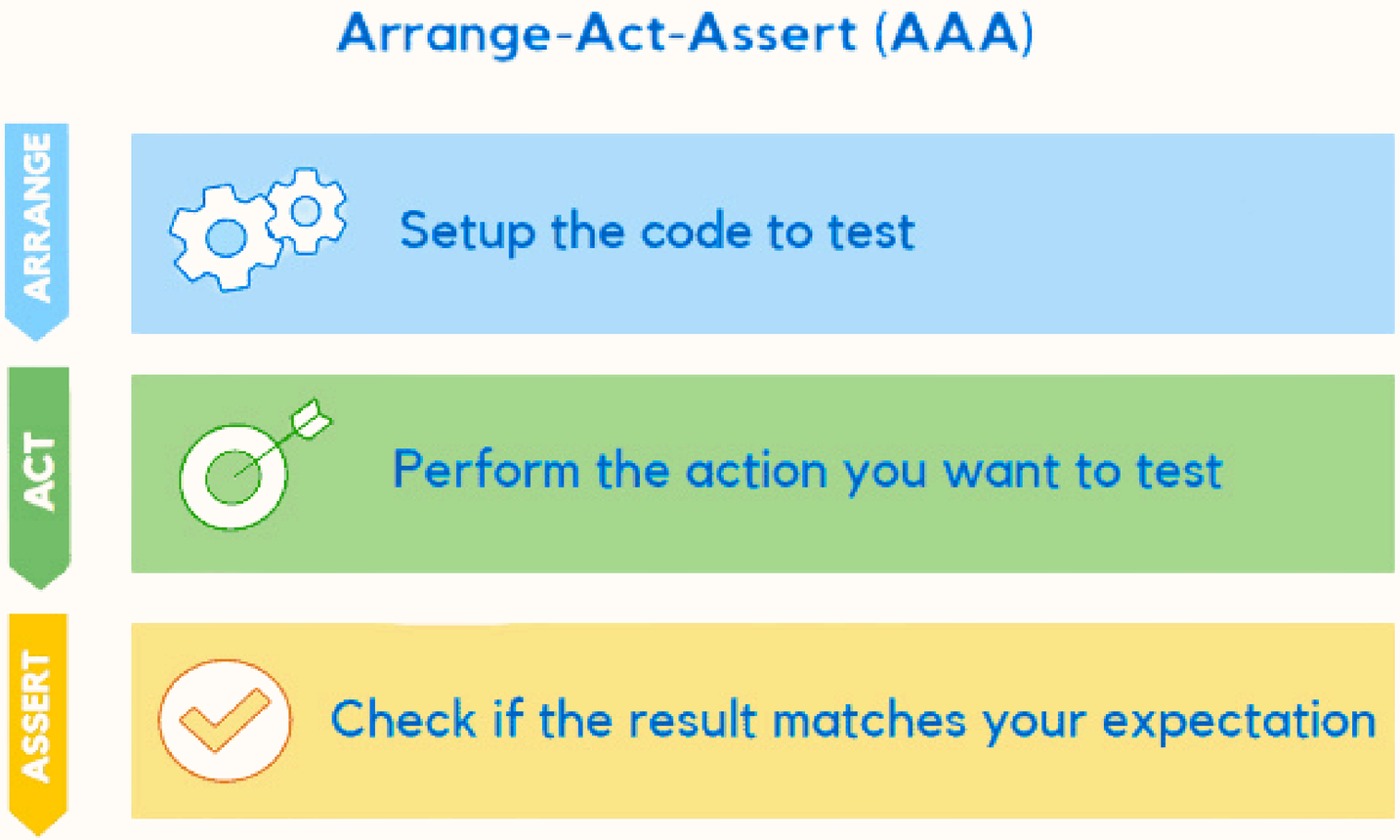
Amaç:

- Her bir bileşenin bağımsız şekilde doğru çalıştığını doğrulamak.

Genellikle otomatik olarak çalıştırılır.



Birim Testin Temel Yapısı



Arrange (Hazırlık) Aşaması

Test için gerekli tüm nesneler ve veri senaryosu hazırlandığı aşama;

Bu aşamada dış bağımlılıklar **mock** veya **stub** ile izole edilir (örneğin Mockito).

Amaç: sadece test edilen birime odaklanmak.

Act (Eylem) Aşaması

Test edilecek metodların çağırıldığı aşamadır. Sadece tek bir davranış test edilir.

Amaç; Test edilen metodun davranışını gözlemlemek.

Not: Yan etkisi olan kodlar (örneğin dosya yazma, ağ erişimi) birim test için uygun değildir.

Assert (Doğrulama) Aşaması

Sonucun (result), beklenen değerle (expected value) ile karşılaştırıldığı aşamadır.

Assertation: Kodun “doğru” çalıştığını kanıtlayan ifadelerdir.

- **assertEquals**(**expected, actual**)
- **assertTrue**(**condition**)
- **assertFalse**(**condition**)
- **assertNull**(**object**)
- **assertNotNull**(**object**)
- **assertThrows**()
- **asertTimeout**()

Not: İyi bir test, açık ve anlaşılır assertion’lar içerir.

Assertion Önemi

- Testin güvenilirliği assertion'ların doğruluğuna bağlıdır.
- Assertion yoksa test bir şey test etmiyordur.
- Kod geçerse, "test başarılı";
- Kod başarısızsa, "test failed" olur.

Not: Bazı testlerde sonucun negatif olması veya exception fırlatması testin başarılı olmasını sağlayabilir.

```
assertThrows(ArithmeticException.class, () -> calculator.divide(10, 0));
```

Birim Testi Avantajları

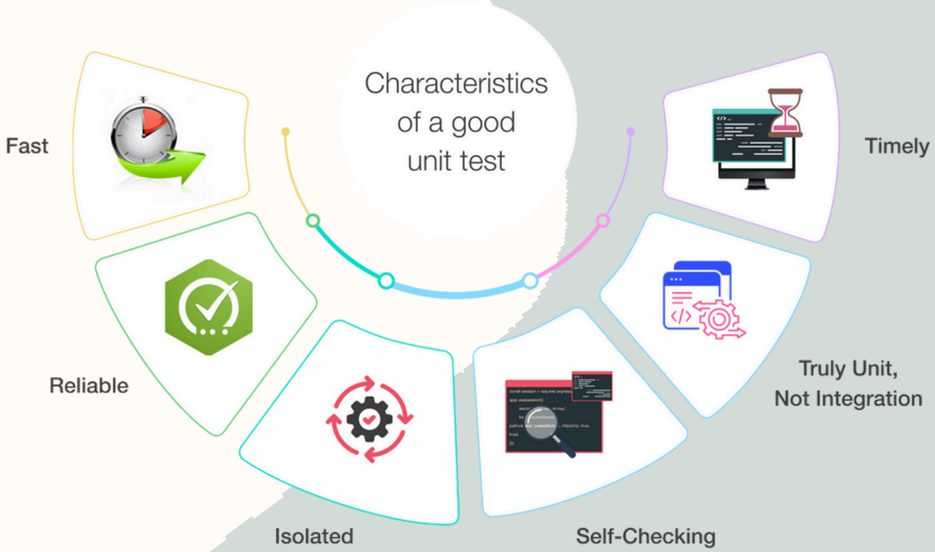
- Kodun kalitesini artırır
- Regresyon hatalarını (eski hataların geri dönmesi) önler
- Refactoring (yeniden düzenleme) işlemlerini güvenli hale getirir
- Geliştiriciye özgüven kazandırır
- TDD (Test Driven Development) için temel oluşturur

“Testi olmayan kod, tamamlanmış kod değildir.”

Birim Testleri Özellikleri

İyi bir birim testi aşağıdaki özelliklere sahip olmalıdır;

- Otomatik, Manuel müdahale olmadan çalışmalıdır
- Tekil, Sadece tek bir davranışı test etmelidir
- Bağımsız, diğer testlerden etkilenmemelidir
- Tekrarlanabilir, her çalıştırmada aynı sonucu vermelidir
- Hızlı, ms içersinde test sonuçlanmalıdır.



Birim Testleri ve Yazılım Kalitesi

- Kod kalitesi artar: Her modülün davranışı belgelenmiş olur.
- Bakım kolaylığı sağlar: Değişiklikler kolay test edilir.
- Sürdürülebilirlik: Kodun uzun vadeli stabilitesini sağlar.
- Ekip iletişimi: Testler, kodun nasıl davranması gerektiğini netleştirir.
- Maliyet azaltır: Hatalar erken tespit edildiği için sonradan düzeltme maliyeti düşer.

Birim test, “kaliteyi sonradan eklemek” değil, “baştan inşa etmektir.”

Yazılım Yaşam Döngüsündeki Rolü

1. **Planlama aşamasında:** Gereksinimler test edilebilir şekilde yazılır
2. **Geliştirme aşamasında:** Kodla birlikte testler oluşturulur
3. **Bakım aşamasında:** Yeni özellikler eklenirken testler çalıştırılır
4. **Sürekli Entegrasyon (CI)** süreçlerine entegre edilir (Jenkins, GitHub Actions, GitLab CI)

Test Method İsimlendirme

Kod okunabilirliği, sadece üretim kodu için değil, testler için de geçerlidir.

İyi bir test ismi:

- Testin amacını açıklamalı
- Ne beklediğini göstermeli
- Neden başarısız olabileceğini anlamayı kolaylaştırmalı

Yetersiz isimlendirme → **“Bu test neyi test ediyor?”** sorusunu doğurur

→ should<**ExpectedBehavior**>_when<**ConditionOrAction**>()

İyi İsimlendirme Örnekleri

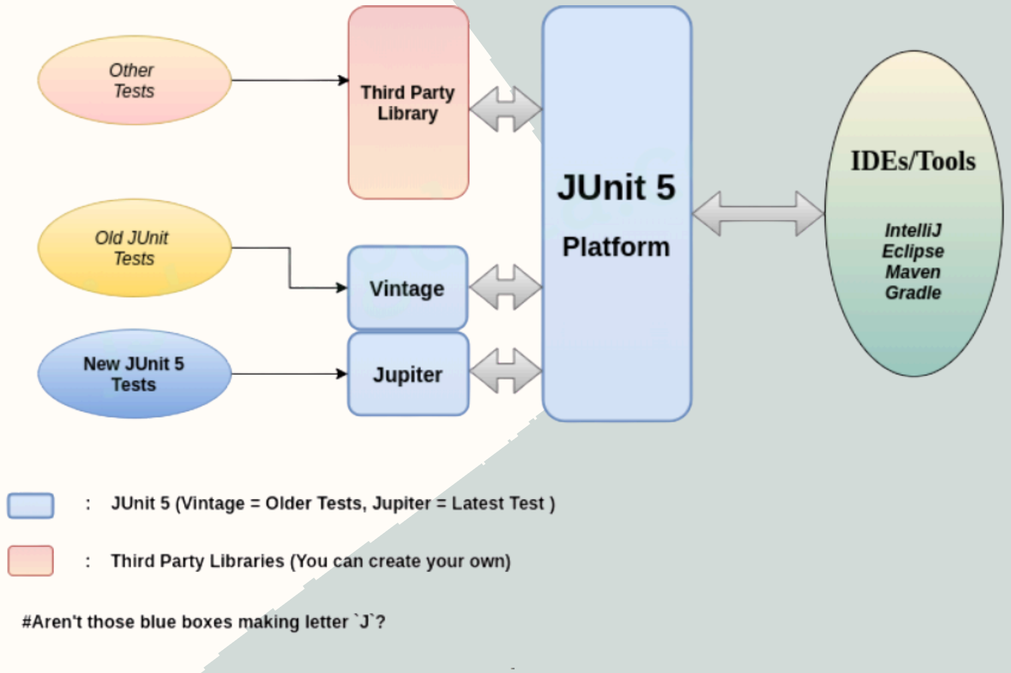
givenProductName_whenFindByName_ThenProductFound	**Doğru ve anlaşılır ifade**
givenTwoNumbers_whenAddThenSım()	**Açık, davranış temelli**
shouldThrowException_whenDivideByZero()	**Beklenen hata durumu net**
shouldReturnEmptyList_whenNoUsersFound()	**Boş sonuç senaryosu**
shouldUpdateEmailAddress_whenValidUserIdProvided()	**Güncelleme davranışı açıklanmış**

JUnit

JUnit, Java programlama dili için geliştirilmiş bir birim test framework'üdür.

1. Kodlarının doğruluğunu test etmesini,
2. Testleri otomatik olarak çalıştırmasını,
3. Hataları erken fark etmesini sağlar.

Açık kaynaklıdır ve Test Driven Development (**TDD**) yaklaşımının temel araçlarındandır.



UNIT TESTING WITH

JUnit

JUnit’in Sağladığı Özellikler

- Basit **@Test** anotasyonu ile test metotları tanımlama
- **@Before** / **@After** gibi (lifecycle) yönetimi
- **Assertion** (doğrulama) metotları ile beklenen sonuçların kontrolü
- **Test Suites** ile testleri gruplama
- Otomatik test raporları
- Mock kütüphaneleri (ör. **Mockito**) ile uyumlu çalışma
- Maven, Gradle, Jenkins, IntelliJ ile entegre

UNIT TESTING WITH

JUnit

JUnit 4

Klasik Nesil

- 2006’da yayınlandı
- Basit ve kararlı bir yapı sundu
- Annotation tabanlı test sistemi getirdi
- Ancak sınırlı genişletilebilirlik ve mimari esnekliğe sahipti

JUnit Vintage & **JUnit Platform** temel altyapıları üzerine kurulmuştur.

UNIT TESTING WITH

JUnit 5

Modern Nesil

- 2017’de tamamen yeniden tasarlandı
- **“Platform + Engine + API”** mimarisi ile modüler hale geldi
- JUnit 4 testlerini de destekler
- Jupiter API ise Java 8+ özelikleri (Lambda, Stream, Annotation ve Extension API) gibi modern Java özellikleriyle uyumludur

UNIT TESTING WITH

JUnit

JUnit Vintage

Klasik Nesil

- Eski JUnit 3 ve JUnit 4 testlerinin çalışabilmesi için köprü görevi görür.
- Büyük, eski projelerde geçiş sürecinde çok işe yarar.
- Böylece aynı projede hem @Test (JUnit 4) hem @Test (Jupiter) testleri sorunsuz coexist edebilir.

UNIT TESTING WITH

JUnit

JUnit Jupiter

Yeni Nesil

- JUnit 5'in yeni nesil test API'sidir.
- **@Test**, **@BeforeEach**, **@AfterAll**, **@DisplayName**, **@ParameterizedTest** gibi modern annotation'ları destekler
- Java 8+ özelliklerini (lambda, stream, method reference) destekler.

JUnit Platform

UNIT TESTING WITH JUnit

- Tüm testlerin çalıştırıldığı altyapı katmanıdır.
- IDE'ler, Maven, Gradle gibi araçlarla test motorlarının iletişimini sağlar. Test Runner olarak çalışır
- Kendi başına test içermez; sadece test motorlarını çalıştırır.

Kısaca:

- **JUnit Platform** motorları çalıştırır,
- **JUnit Jupiter** yeni testleri sağlar,
- **JUnit Vintage** eski testleri destekler.

UNIT TESTING WITH

JUnit

JUnit Testler

Junit de testler parametreli ve parametresiz olmak üzere 2 farklı tipte tanımlanır.

@Test ve **@ParameterizedTest** anatasyonlarından yararlanılır.

Not: TDD yaklaşımında annotation'lar, testleri daha okunabilir, sürdürülebilir ve organize hale getirir

JUnit Annotations

Testleri yönetmek, düzenlemek ve yaşam döngüsünü kontrol etmek için kullanılır. Test kodunu okunabilir ve sürdürülebilir hale getirir

@Test → method seviyesi → Test metodunu tanımlar

@Disabled → method veya class → Testi geçici olarak devre dışı bırakır

@DisplayName → method veya class → Raporlama için okunabilir özel isimlendirme

@Tag → method veya class → Testleri gruplamak, filtrelemek için kullanılır

@DataJpaTest → JPA için lightweight test, Hibernate otomatik konfigüre edilir

@ParameterizedTest → Parametrik test methodu tanımlar

Lifecycle Annotation’ları

Test sınıfındaki metotların çalışma zamanındaki **sırasını** ve **kapsamını** kontrol etmek için kullanılır.

@BeforeEach → Her test metodundan önce çalışır

@AfterEach → Her test metodundan sonra çalışır

@BeforeAll → Tüm testlerden önce, bir kez çalışır

@AfterAll → Tüm testlerden sonra, bir kez çalışır. **Metod static** olmalı

Not: Bunun ile ilgili basit bir örnek.

Parameterized Annotations

Testleri yönetmek, düzenlemek ve yaşam döngüsünü kontrol etmek için kullanılır. Test kodunu okunabilir ve sürdürülebilir hale getirir

@ParameterizedTest → Aynı testin farklı parametrelerle çalışmasını sağlar. Method seviyesi

@ValueSource → Tek boyutlu değerleri test için verir

@CsvSource → Çoklu değerleri virgül ile test eder

@EnumSource → Enum değerlerini test için kullanır

Not: **@ValueSource**, **@CsvSource**, **@EnumSource**, **@ParameterizedTest** anatasyonu ile kullanılır.

Assertations Kavramı

Test edilen kodun beklenen davranışı sağlayıp sağlamadığını kontrol eden mekanizmadır.

- Testin sonuçlarını otomatik olarak kontrol eder, manuel kontrol ihtiyacını ortadan kaldırır
- JUnit 5'te **org.junit.jupiter.api.Assertions** sınıfı ile sağlanır

En yaygın kullanılanlar

- **assertEquals**(**expected, actual**) → **assertNotEquals**(**unexpected, actual**)
- **assertTrue**(**condition**) → **assertFalse**(**condition**)
- **assertNull**(**object**) → **assertNotNull**(**object**)
- **assertArrayEquals**(**expectedArray, actualArray**)
- **assertThrows**(**expectedException.class, executable**)

Exception Testleri

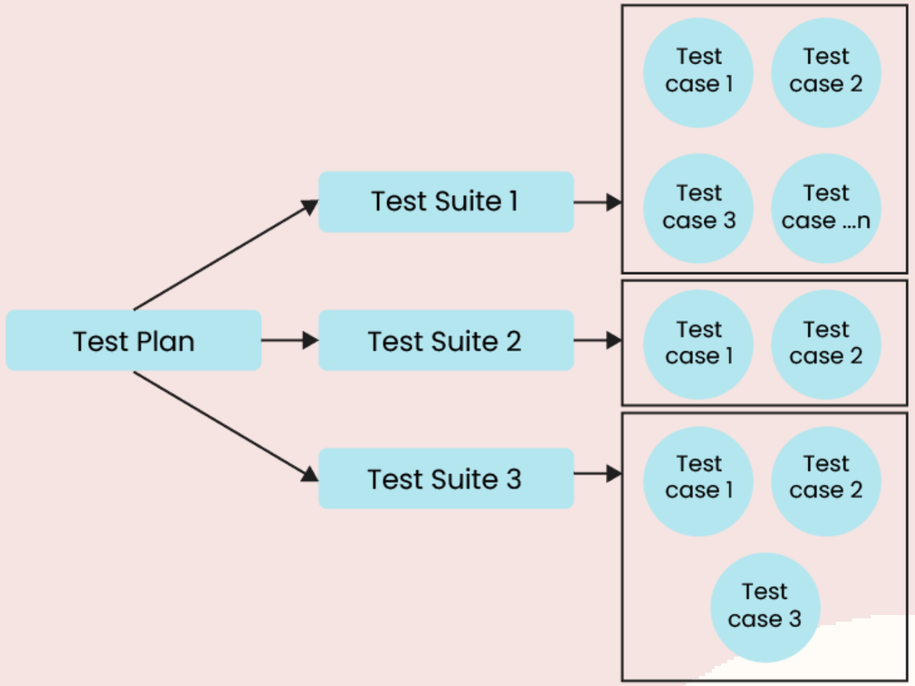
- Exception testi, bir metodun belirli bir koşulda hata fırlatıp fırlatmadığını doğrulamak için yapılır
- TDD yaklaşımında hatalı senaryoları test etmek kritik öneme sahiptir
- JUnit 5'te **assertThrows** ile yapılır

Örn: assertThrows(ArithmeticException.class, () => calc.divide(5, 0));

Test Suite Kavramı

Birden fazla test sınıfını tek bir çatı altında çalıştırmak için kullanılır

- Büyük projelerde test organizasyonu ve yönetimi sağlar
- Testlerin daha derli toplu ve raporlanabilir olmasını sağlar



Test Fixture Kavramı

Testin çalışması için gereken ortamı, nesneleri ve veriyi hazırlar.

Amaç: Her testin aynı ve güvenilir ortamda çalışmasını sağlamak

Not: Lifecycle Annotation'ları testin **"ne zaman çalışacağını"** kontrol ederken, **Test Fixture** testin "çalışması için neye ihtiyaç duyduğunu" hazırlar.

Test Suite Kavramı

JUnit 5'te test suites, **@Suite** ve **@SelectClasses** veya **@SelectPackages** annotation'ları ile yapılır.

@Suite → Test Suite olduğunu belirtir

@SelectClasses → Suite içinde çalışacak test sınıflarını seçer

@SelectPackages → Suite içinde belirli paketleri seçer

1. CI/CD pipeline'larında toplu test çalıştırma kolaylığı
2. Testleri gruplandırarak organize etmek
3. Test raporlamasını daha anlaşılır hale getirmek
4. Tek tıkla birden fazla testi çalıştırmak

Kod Coverage Kavramı

Testlerin kod üzerinde ne kadar kapsam sağladığını ölçer

Farklı seviyelerde ölçülebilir:

1. **Line Coverage** (**Satır Kapsamı**) → Kaç satır test edildi
2. **Branch Coverage** (**Dallanma Kapsamı**) → If/Else, switch gibi dallanmalar
3. **Method Coverage** (**Metot Kapsamı**) → Kaç metodun test edildiği
4. **Class Coverage** (**Sınıf Kapsamı**) → Kaç sınıfın test edildiği

Kapsam yüzdesi % olarak gösterilir:

- %100 → tüm satırlar/testler çalışıyor
- %0 → hiç test çalıştırılmamış

Code Coverage Ölçümü

JUnit doğrudan kod coverage ölçmez, ancak IDE ve araçlarla kolayca ölçülür.

IntelliJ IDEA'da Run with Coverage:

- Test sınıfına sağ tık → **Run 'Test' with Coverage**
- IDE, test sırasında hangi satırların çalıştığını işaretler (renklerle gösterir)
  - Yeşil → Test edilen kod
  - Sarı → Kısmen test edilen kod
  - Kırmızı → Hiç test edilmeyen kod

Not: JaCoCo (Java Code Coverage) gibi araçlar ile grafik ve tablo raporu oluşturulabilir.
Not: %80-90 coverage → çoğu endüstride hedeflenen aralık

Integration & System Test

Gerçek test (**Integration Test** / **System Test**),

Kodun gerçek ortamda, gerçek bağımlılıklarla doğru şekilde entegre olup olmadığını kontrol eder.

Gerçek Testler;

- Tüm Sistem Birlikte Test Edilecekse (Service, Repository, Controller)
- Gerçek entity kaydı, transaction veya REST endpoint testinde.
- @SpringBootTest, @DataJpaTest ve @WebMvcTest gibi anatasyonlardan yararlanılır.
- Uygulamanın tamamı değişiklik sonrası hâlâ doğru çalışıyor mu? (Regrasyon Testleri)

Mock Test

Yazılım testlerinde bağımlı bileşenleri (repository, API, queue, dosya sistemi, vb.) taklit eden sahte objelerle (mock) yapılan testtir.

Amaç; Gerçek bağımlılıklar olmadan, sadece test etmek istediğin sınıfın (unit) doğru çalışıp çalışmadığını kontrol etmektir.

- “Sadece bu sınıf doğru hesaplama yapıyor mu?” Bu durumda gerçek context açmaya gerek yoktur.
- Testin Hızlı Olması Gerekiyorsa
- Henüz Gerçek Servis Hazır Değilse
- Dış Bağımlılıklardan izole test etmek istersek

Mockito

Java için bir mocking framework'tür. Testlerde, gerçek nesnelerin yerine sahte (mock) nesneler kullanarak bağımlılıkları izole etmeyi sağlar

Amaç: Unit Testleri bağımsız, hızlı ve güvenilir çalıştırmak

Önemli Anatasyonlar

@ExtendWith(MockitoExtension.class): JUnit 5 testlerinde Mockito'nun aktif hale gelmesini sağlar.

@InjectMocks: "Bu sınıfın gerçek bir instance'ını oluştur ve varsa bağımlılıklarını mock nesneler ile otomatik olarak enjekte et."

@Mock : Sınıfın gerçek bir instance'ı yerine sahte (mock) bir nesne oluştur.

Mockito Temelleri

Mock (**Sahte nesne**): Testlerde sahte nesnelerden yararlanılır.

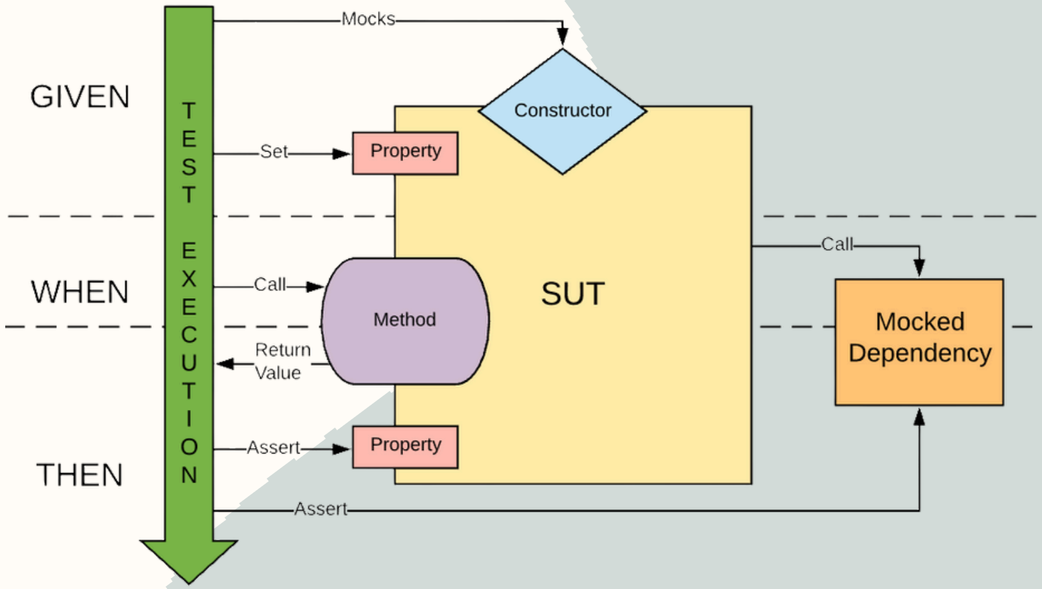
```
List<String> mockedList = Mockito.mock(List.class);
```

Stub / **When-Then:** Belirli bir method çağrısında beklenen davranışı temsil eder.

```
Mockito.when(mockedList.get(0)).thenReturn("Hello");
assertEquals("Hello", mockedList.get(0));
```

Verify: Metodun çağrılıp çağrılmadığını kontrol eder

```
mockedList.add("Test");
Mockito.verify(mockedList).add("Test");
```



Teşekkürler!

Teknoloji dolu günler geçirmeniz dileği ile hoşçakalın !