## Temiz Test Kodu Yazmak

Test tekrarından kaçınmak temiz test kodu yazmanın en önemli ilkesidir. Daha yüksek seviyeli bir test bir hata tespit ederse ve daha düşük seviyeli bir test başarısız olmazsa, daha düşük seviyeli bir test yazmanız gerekir. Testlerinizi test piramidinin olabildiğince aşağısına itilmelidir. Daha düşük seviyeli testler, hataları daha iyi daraltılmasına ve izole bir şekilde çoğaltmanıza izin verir. Daha hızlı çalışacaklardır. Ve gelecek için iyi bir regresyon testi olarak hizmet edecekler.

İkinci kural, testlerinizi hızlı tutmak için önemlidir. Tüm koşulları daha düşük seviyeli bir testte güvenle test ettiyseniz, testlerinizi daha yüksek seviyeli bir test tutmanıza gerek yoktur. Her şeyin çalıştığına dair daha fazla güven sağlamayacaktır. Gereksiz testlere sahip olmak can sıkıcı olabilir. Testlerinizin z daha yavaş olacak ve kodunuzun davranışını değiştirdiğinizde daha fazla test değiştirmeniz gerekecektir.

Test kodları önemlidir, iyi derecede önem gösterilmelidir

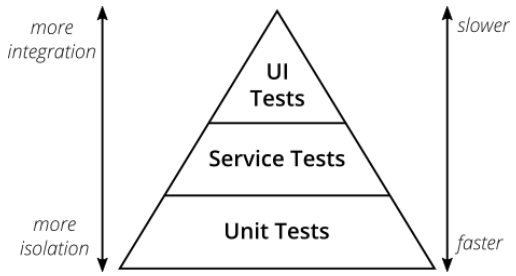
Her test için bir durumu test etmek, test kodlarını kısa ve hızlı olmasını saylayacaktır.

"arrange, act, assert" or "given, when, then"  testlerin kolay anlaşılmasını sağlar.

Testlerin okunabilirliği önemlidir. Bunun için metod isimleri uzun yazılabilir.

## The Test Pyramid

Test Pyramid , Mike Cohn tarafından yaratılmış bir metafordur. Bu metafor bize test yapmamız için 3 layer vermektedir. Bu layer'ların isimleri: Unit Tests, Service Tests, User Interface Tests.




Test Pyramid

Bu test pramidi çok basit göründüğü için ideal değildir. Piramit şekline bağlı kalarak hızlı ve bakımı kolay bir test paketi için: Çok sayıda küçük ve hızlı birim testi yazmaktır.

## Unit Test nedir?

Unit: en küçük öğre, en küçük birimdir. Bir programlama dilinde bir class veya bu class'ın içindeki method olabilir.  Unit Test ise, oluşturduğunuz test codebasinizdeki bir unit'in çalıştığını kontrol eder ve emin olur. Unit Testlerin sayısı diğerlerine göre fazla olur. Unit Testlerin production koduna çok yakın olmaması gerekir. Eğer yakın olursa unit test yazmak zor, sıkıcı ve uzun olur, aynı zamanda unit testlerin koruması avantajını kaybedersiniz.

Birim testler çok hızlı çalışacaktır. İyi bir makinede birkaç dakika içinde binlerce birim testi çalıştırılabilir. Kod tabanınızın küçük parçalarını ayrı ayrı test edilmeli ve testleri hızlı tutmak için veritabanlarına, dosya sistemine veya HTTP sorgularını başlatmaktan (bu parçalar için sahte ve taslaklar kullanarak) kaçılınmalıdır..

Sociable and Solitary Unit Testleri birbirine zıt iki farklı kavramdır. Martin Flower'a göre önemli olan bu kavramlar değil otomatik test yazmaktır.

Testlerde kullanılan bir diğer kavram ise Mock and Stub'lardır. Testte kullanacağınız bir sınıfın, modülün veya metodun sahtesini oluşturur. Sahte versiyonlar gerçek versiyonlar gibi çalışır ama sizin istediğiniz şekilde çıktı verirler.

Bir test kodunun yapısı 3 ksıımdan oluşur:

1. Test verilerini ayarlayın

2. Test etmek istediğiniz birimi çağırın
3. Beklenen sonuçların döndürüldüğünü kontrol edin

# Integration Tests

Unit test yazılırken veritabanları, diğer uygulamalara çağrılar kullanılmadan daha hızlı olavak testler yazılır. Ama bir porje yayınlanmadan önce bunlarında test edilmesi gereklidir. Bu durumlarda integration test kullanılır. Integration testler; uygulamada birlikte çalışması gereken parçaların doğru şekilde çalışabileceği, doğru iletişim kurdukları konusunda size güven verme avantajına sahiptirler. Birim testleri bu konuda size yardımcı olamaz.

Testlerde bir kere de olsa Database Integration test yapmak önemlidir. Bu repository'nin doğru çalıştığına emin olunmasını sağlar. Integration testin kod yapısı unit testeli gibi 3 kısımdan oluşur.

Uygulamamızda veri almak için kullandığımız bir apiyi test ederken, o apiye sorgu atmamak önemlidir. Apiye sorgu atmadan kullanılan api nin bir kopyası yaratılabilir.

```
@RunWith(SpringRunner.class)

@SpringBootTest

public class WeatherClientIntegrationTest {



  @Autowired
  private WeatherClient subject;

  @Rule
  public WireMockRule wireMockRule = new WireMockRule(8089);

  @Test
  public void shouldCallWeatherService() throws Exception {
    wireMockRule.stubFor(get(urlPathEqualTo("/some-test-api-key/53.5511,9.9937"))
        .willReturn(aResponse()
            .withBody(FileLoader.read("classpath:weatherApiResponse.json"))
            .withHeader(CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
            .withStatus(200)));

    Optional<WeatherResponse> weatherResponse = subject.fetchWeather();

    Optional<WeatherResponse> expectedResponse = Optional.of(new WeatherResponse("Rain"));
    assertThat(weatherResponse, is(expectedResponse));
  }
}
```

# Contract Tests

Uygulamalar birçok küçük hizmete bölünür ve genellikle bu servisler arabirimler aracılığıyla birbirleriyle iletişim kurarlar. Sistemlerde genelde sağlayıcı ve tüketici servisler vardır. Sağlayıcı servisler veri sunarken( write, publish), tüketici servisler(fetch, execute) ise bu veriyi talep edenlerdir(okumak). Veri sağlayan ve veri tüketen servislerin iletişimi için Contract'lar beklenir. Bunları tegst etmek için Automated contract tesleri bu sözleşmeye bağlı kalınıldığından emin olmamızı sağlar.

**Tüketiciye Dayalı Sözleşme testleri (CDC testleri),** tüketicilerin bir sözleşmenin uygulanmasını yönlendirmesine izin verir. CDC kullanarak, bir arabirimin tüketicileri, arabirimden ihtiyaç duydukları tüm veriler için arabirimi kontrol eden testler yazar. Tüketim ekibi daha sonra bu testleri yayınlar, böylece yayınlama ekibi bu testleri kolayca alabilir ve yürütebilir. Sağlayıcı ekip artık CDC testlerini çalıştırarak API'lerini geliştirebilir. Tüm testler geçtikten sonra, tüketen ekibin ihtiyaç duyduğu her şeyi uyguladıklarını biliyorlar.

## UI Tests

Çoğu uygulama bir arayüze sahip olduğundan, bu arayüzde test edilmesi gerekmektedir. UI testleri, uygulamanızın kullanıcı arayüzünün doğru çalıştığını test eder. Uygulamanıza ve kullanıcılarınızın ihtiyaçlarına bağlı olarak, kod değişikliklerinin web sitesinin düzenini bozulmadığını test edebilir. UI testlerinde örneğin kullanıcı girişi doğru eylemleri tetiklemeli, veriler kullanıcıya sunulmalı, UI durumu beklendiği gibi değişmelidir.

Arayüz Selenium gibi araçlarla test edilebilir. Bir REST API'yi kullanıcı arayüzsüz olarak düşünülüyorsa, API'nizin etrafına uygun entegrasyon testleri yazılarak ihtiyaç duyulan herşey elde edilebilir. Vanilla javascript , Jasmine veya Mocha gibitest araçları kullanılabilri. Daha geleneksel, sunucu tarafında oluşturulmuş bir uygulama ile Selenium tabanlı testler en iyi yoldur.

Kullanılabilirliği ve "iyi görünen" bir faktörü test etmek istediğinizde, otomatik test yapılamayabilir.

## End-to-End Tests

Uçtan uca testler, sisteminizin tamamını test eder. Uçtan uca testler (Geniş Yığın Testleri olarak da adlandırılır), yazılımınızın çalışıp çalışmadığına karar vermeniz gerektiğinde size en büyük güveni verir. Selenium ve WebDriver Protokolü, tıklamalar gerçekleştirerek, veri girerek ve kullanıcı arabiriminizin durumunu kontrol ederek testlerinizi otomatikleştirmenize olanak tanır. Selenium'u doğrudan kullanabilir veya üzerine inşa edilmiş araçları kullanabilirsiniz, Nightwatch bunlardan biridir.

Bu testler beklenmedik ve öngörülemeyen nedenlerden dolayı başarısız olabilirler. Ayrıca bu testi hangi birimin yazacağı da bir soru işareti oluşturabilir.

En-to-End testler olabilecek minumum sayıda yazılmalıdır. Ayrıca, uçtan uca testler çok fazla bakım gerektirir ve oldukça yavaş çalışır. Birkaç mikro hizmetten daha fazlasının bulunduğu bir ortamı düşündüğünüzde, uçtan uca testlerinizi yerel olarak bile çalıştıramazsınız - çünkü bu, tüm mikro hizmetlerinizi yerel olarak da başlatmanızı gerektirebilir.

**User Interface End-to-End Test:** Uçtan uca testler için Selenium ve WebDriver protokolü birçok geliştirici için tercih edilen araçtır. Selenium ile beğendiğiniz bir tarayıcı seçebilir ve web sitenizi otomatik olarak aramasına izin verebilir, burayı ve burayı tıklayın, verileri girin ve kullanıcı arayüzünde değişiklik olup olmadığı kontrol ettirebilir.

**REST API End-to-End Test:** Uygulamanızı test ederken bir grafik kullanıcı arabiriminden kaçınmak, end to end test yapmak için daha ideal olabilir. Yada web kullanıcı arayüzünüz olmadığınnda, bunun yerine bir REST API'si sunuyorsunuz. Subcutaneous Test bu iki durumda da size tesst yapma imkanı verebilir.

## Acceptance Tests

Bir noktada, yazılımınızın sadece teknik açıdan değil, kullanıcı açısından da doğru çalıştığı test edilmelidir. Bu testler belki çok fazla önemli olmayabilir. Kabul testleri farklı ayrıntı düzeylerinde olabilir. Çoğu zaman oldukça üst düzey olacaklar ve hizmetinizi kullanıcı arayüzü üzerinden test edeceklerdir. Ancak, teknik olarak test piramidinizin en üst seviyesinde kabul testleri yazmaya gerek yoktur. Uygulama tasarımınız ve elinizdeki senaryo, daha düşük bir düzeyde bir kabul testi yazmanıza izin veriyorsa, bunun için yapılabilir. Flower'a göre düşük seviyeli bir teste sahip olmak, yüksek seviyeli bir teste sahip olmaktan daha iyidir.

## Exploratory Test

En iyi kurgulanmış test otomasyonları bile bazen mükemmel olmayabilir. Bazen otomatik testlerinizde belirli uç durumlar kaçırılabilir. Bazen bir birim testi yazarak belirli bir hatayı tespit etmek neredeyse imkansızdır. Bazı kalite sorunları, otomatik testlerinizde bile belirgin hale gelmiyor olabilir. Bazı türlerin manuel olarak test edilmesi iyi bir fikir olabilir.

Çalışan bir sistemdeki kalite sorunlarını tespit etmek için testçinin özgürlüğünü ve yaratıcılığını vurgulayan manuel bir test yaklaşımıdır. Bir program üzerinde çalışan uygulama ele alınarak, uygulamadaki sorunları ve hataları kışkırtmanın yollarını bulunur, Bulunan her şeyşey belgelenir. Bu sorunlar, hatalar, tasarım sorunları, yavaş yanıt süreleri, eksik veya yanıltıcı hata mesajları olabilir.

Exploratory testleri sırasında, gözden kaçan ufak sorunları tespit edilebilir.