1. Sistem Testinin Amacı, Kapsamı ve Karakteristikleri

Sistem testi, yazılım ürününün "bir bütün olarak" ele alındığı ilk test seviyesidir. Bu aşamada hedef, tek tek bileşenlerin doğruluğunu sınamaktan çok, onların bir araya geldiğinde ortaya çıkan davranışı iş gereksinimleri, kullanıcı beklentileri ve kalite standartları açısından değerlendirmektir. Başka bir ifadeyle sistem testi, yazılımın yalnızca "çalışıp çalışmadığına" değil, "doğru şeyleri, doğru şekilde yapıp yapmadığına" odaklanır. Bu bölümde, sistem testinin temel amaçlarını, neleri kapsadığını ve onu diğer test seviyelerinden ayıran karakteristik özellikleri ele alacağız.

1.1. Sistem Testinin Temel Amaçları

Sistem testinin en temel amacı, geliştirilen yazılımın, tanımlanmış sistem gereksinimlerini ne ölçüde karşıladığını ortaya koymaktır. Gereksinimler; fonksiyonel, fonksiyonel olmayan, yasal, güvenlik ve kullanım senaryosu temelli maddelerden oluşabilir. Sistem testi, bu gereksinimleri tek tek modüller seviyesinde değil, uçtan uca senaryolar bağlamında değerlendirir. Örneğin bir otopark yönetim sistemi için amaç, yalnızca "ücret hesaplama fonksiyonu doğru mu?" sorusuna değil, "kullanıcı park oturumunu başlatıp sonlandırdığında, abonelik durumu, süre ve tarife koşulları birlikte ele alındığında sistem doğru ücretlendirmeyi yapıyor mu?" sorusuna cevap aramaktır.

İkinci önemli amaç, entegrasyon seviyesinde fark edilmeyen etkileşim sorunlarını ortaya çıkarmaktır. Modüller kendi aralarında doğru veri alışverişi yapabilir, entegrasyon testlerini geçebilir; ancak gerçek kullanım senaryolarında farklı bileşenlerin belirli bir sırayla, belirli veri hacimleri altında ya da belirli hata durumlarında nasıl davrandığı yalnızca sistem testinde görünür hâle gelir. Bu yüzden sistem testi, "gerçek hayata yakın koşullarda" sistemin bütünsel davranışını gözlemleme aracı olarak kritik öneme sahiptir.

Üçüncü amaç, yazılımın canlı ortama alınmasına ilişkin riskleri azaltmaktır. Sistem testi sırasında tespit edilen hatalar, çoğu zaman kullanıcıya doğrudan dokunan, iş süreçlerini kesintiye uğratabilecek veya güvenlik açıklarına yol açabilecek türdendir. Canlıya çıkmadan önce bu hataların mümkün olduğunca azaltılması, geri çağırma (rollback), acil yama (hotfix) gibi maliyetli müdahalelerin önüne geçer. Böylece sistem testi, hem teknik riskleri hem de iş ve itibar risklerini yönetmeye hizmet eder.

Son olarak, sistem testi yazılımın kalite seviyesi hakkında paydaşlara nesnel bilgi sağlayan bir ölçüm noktasıdır. Çalıştırılan test sayısı, geçen/kalan test oranları, bulunan hata sayısı ve hataların şiddet dağılımı gibi metrikler, proje yöneticilerine ve iş birimlerine "bu ürün teslim edilmeye hazır mı?" sorusuna daha bilinçli cevap verebilmeleri için temel oluşturur. Bu nedenle sistem testi, yalnızca teknik bir doğrulama aracı değil, karar destek mekanizmasının da bir parçası olarak düşünülmelidir.

1.2. Kapsam ve Sistem Sınırları

Sistem testinin sağlıklı yürütülebilmesi için, öncelikle neyin test edileceğinin ve neyin test kapsamı dışında kalacağının netleştirilmesi gerekir. "Kapsam" (scope), bir sistem testi döngüsünde ele alınacak modüller, fonksiyonlar, senaryolar, veri türleri ve entegrasyon noktalarının sınırlarını tanımlar. Bu kapsamın açıkça yazılması, hem test ekibinin odaklanmasını sağlar hem de paydaşlarla beklenti yönetimi için temel oluşturur.

In-scope ifadesi, bu döngüde mutlaka test edilmesi hedeflenen alanları belirtir. Örneğin bir sürümde yalnızca "otopark ücretlendirme akışının" tamamen elden geçirildiği bir geliştirme yapılmışsa, sistem testinin kapsamına bu akışla ilgili tüm senaryoların (normal kullanım, abonelikli müşteriler, abonelik dışı müşteriler, hata durumları, iptal süreçleri vb.) alınması beklenir. Buna karşılık, sistemin başka bir modülü –örneğin kullanıcı profil güncelleme ekranı– bu sürümde değişmemişse, o modül mevcut sistem testi döngüsünün kapsamı dışında bırakılabilir. Bu tür sınır çizimleri, test eforunun kritik alanlara yönlendirilmesini sağlar.

Out-of-scope ifadesi, o sistem testi döngüsünde bilinçli olarak test edilmeyecek alanları tanımlar. Bazen bu alanlar teknik veya zamansal kısıtlar nedeniyle kapsam dışı bırakılır; bazen de bu alanların önceki sürümlerde yeterince test edildiği ve bu sürümde değişmediği varsayılır. Ancak hangi gerekçeyle olursa olsun, kapsam dışı bırakılan alanların açıkça belirtilmesi önemlidir; aksi takdirde, sistem testinin sonunda ortaya çıkabilecek eksik test eleştirilerinin kaynağı belirsiz kalır.

Kapsam kavramı, sistem sınırları ile birlikte ele alınmalıdır. Sistem sınırı, bu yazılımın sorumluluk alanının nerede başlayıp nerede bittiğini gösterir. Örneğin bir ödeme sistemi, kendi içinde kart bilgilerini işliyor olabilir; ancak banka tarafındaki onay süreci üçüncü taraf bir servise ait olabilir. Sistem testi, kendi sınırları içindeki davranışı doğrudan test eder; sınırın dışında kalan sistemler için ise genellikle test ortamı, sahte servisler (mock, stub) veya gerçek ama "sandbox" niteliğinde ortamlar kullanılır. Bu ayrımın netleştirilmesi, hem test tasarımını hem de ortam hazırlığını doğrudan etkiler.

1.3. Fonksiyonel ve Fonksiyonel Olmayan Gereksinimlerin Ele Alınışı

Sistem testinde gereksinimler iki ana grupta ele alınır: fonksiyonel ve fonksiyonel olmayan gereksinimler. Fonksiyonel gereksinimler, sistemin "ne yapması gerektiğini" tanımlar. Örneğin "Kullanıcı geçerli kimlik bilgileriyle giriş yaptığında sisteme erişebilmelidir" veya "Park süresi ve tarife türüne göre ücret hesaplanmalıdır" gibi gereksinimler fonksiyoneldir. Sistem testi, bu gereksinimlerin gerçek

senaryolar altında yerine getirilip getirilmediğini uçtan uca kontrol eder. Bu kontrol, çoğu zaman kullanıcı arayüzü, iş mantığı ve veri katmanını birlikte kapsar.

Fonksiyonel olmayan gereksinimler ise sistemin "nasıl davranması gerektiğine" odaklanır. Performans, güvenlik, kullanılabilirlik, ölçeklenebilirlik, güvenilirlik, taşınabilirlik gibi kavramlar bu başlık altındadır. Örneğin "Sistem, 1000 eşzamanlı kullanıcıyı kabul edilebilir cevap süreleriyle desteklemelidir" veya "Hassas veriler ağ üzerinden taşınırken şifrelenmelidir" ifadeleri fonksiyonel olmayan gereksinimlere örnektir. Bu gereksinimler de sistem testinin doğal parçasıdır; ancak genellikle ek test türleriyle (performans testi, güvenlik testi, kullanılabilirlik testi vb.) desteklenir.

Pratikte yaşanan önemli zorluklardan biri, fonksiyonel olmayan gereksinimlerin çoğu zaman belirsiz veya muğlak ifade edilmesidir. "Sistem hızlı olmalıdır", "kullanımı kolay olmalıdır" veya "güvenli olmalıdır" gibi cümleler, test edilebilirlik açısından yetersizdir. Sistem testinin etkin olabilmesi için bu tür gereksinimlerin ölçülebilir hâle getirilmesi gerekir; örneğin "ortalama cevap süresi 2 saniyenin altında olmalıdır", "yeni bir kullanıcının temel işlemleri öğrenmesi için gerekli adım sayısı X'i geçmemelidir" gibi. Bu nedenle test mühendisi, sistem testi planlama aşamasında gereksinim analistleri ve geliştirme ekibiyle birlikte çalışarak fonksiyonel olmayan gereksinimleri somutlaştırmaya katkı sağlar.

Sonuç olarak, sistem testi yalnızca "fonksiyonlar doğru mu çalışıyor?" sorusuna cevap aramakla yetinmez. Aynı zamanda bu fonksiyonların, beklenen hızda, güvenlikte, kullanılabilirlikte ve kararlılıkta çalışıp çalışmadığını da ele alır. Böylece yazılım ürününün gerçek hayatta karşılaşacağı koşullara hazır olup olmadığına dair daha bütünlüklü bir değerlendirme yapılmış olur.

1.4. Kara Kutu (Black-box) Yaklaşımı ve Gözlemlenebilir Davranış

Sistem testinin karakteristik özelliklerinden biri, büyük ölçüde kara kutu (black-box) yaklaşımına dayanmasıdır. Kara kutu yaklaşımında test mühendisi, sistemin içindeki sınıfları, fonksiyonları, algoritmaları bilmek zorunda değildir ve çoğu zaman bilmez; esas olan, dışarıdan verilen girdilere karşılık hangi çıktının üretildiği ve sistemin hangi yan etkileri oluşturduğudur. Bu yaklaşım, sistem testinin kullanıcı bakış açısını ve gereksinim odaklı yapısını güçlendirir.

Gözlemlenebilir davranış, sistem testinin temel inceleme nesnesidir. Bir web uygulaması için bu davranış, ekranlarda gösterilen mesajlar, sayfalar arası geçişler, butonların tepkileri, veritabanı değişiklikleri, e-posta/mesaj gönderimleri, log kayıtları veya üçüncü taraf sistemlere yapılan çağrılar şeklinde ortaya çıkabilir. Test mühendisi, bir test durumu yürütürken bu davranışları gözlemler, beklenen sonuçlarla karşılaştırır ve bir tutarsızlık durumunda bunu hata olarak raporlar.

Bu noktada önemli bir ayrıntı, kara kutu yaklaşımının "iç yapının tamamen önemsiz olduğu" anlamına gelmediğidir. İç mimari, sistem sınırları, entegrasyon noktaları ve veri akışları hakkında genel bir bilgiye sahip olmak, daha anlamlı test senaryoları tasarlamayı kolaylaştırır. Ancak testin kendisi, bu iç detaylara bağımlı olmadan tasarlanmaya çalışılır; böylece implementasyon değişse bile testlerin büyük kısmı geçerliliğini korur. Bu özellik, özellikle uzun soluklu projelerde ve bakım sürecinde sistem testini değerli kılar.

Kara kutu yaklaşımı, ayrıca sistem testinde kullanılacak test tasarım teknikleriyle de uyumludur. Eşdeğer bölgeleme, sınır değer analizi, karar tabloları, durum geçiş testleri gibi teknikler, genellikle sistemin dıştan görülen davranışını esas alır. Örneğin ücretlendirme sisteminde farklı tarife tipleri, süre aralıkları ve indirim kombinasyonları, eşdeğer sınıflar ve sınır değerler olarak ele alınarak sistem seviyesinde test edilebilir. Böylece kapsamlı bir test seti, iç kodu incelemeye gerek kalmadan oluşturulabilir.

1.5. Kullanıcı Odaklılık ve Kullanım Senaryosu Perspektifi

Sistem testini diğer test seviyelerinden ayıran en önemli özelliklerden biri de güçlü kullanıcı ve iş süreci odaklılıktır. Birim testleri ve entegrasyon testleri çoğunlukla geliştiricinin bakış açısıyla, teknik düzeyde ele alınırken; sistem testi, kullanıcıların sistemi nasıl kullanacağına ilişkin senaryoları merkezine alır. Bu senaryolar, çoğu zaman use case, user story veya iş süreçleri üzerinden türetilir.

Kullanım senaryosu perspektifi, testlerin "tek fonksiyon → tek çıktı" ilişkisinden daha zengin bir yapıya kavuşmasını sağlar. Örneğin bir bankacılık uygulamasında "para transferi" işlevini test etmek, sadece transfer butonuna basıldığında doğru hesap hareketinin oluşup oluşmadığını kontrol etmekten ibaret değildir. Kullanım senaryosu; kullanıcının giriş yapması, alıcı hesabı seçmesi, tutarı girmesi, doğrulama adımlarından geçmesi, olası hatalı girişlerde uyarı alması, başarılı işlem sonrası bildirimleri görmesi ve hesap özetinde işlemi görebilmesi gibi alt adımları içerir. Sistem testi, bu zincirin tamamını uçtan uca doğrulamaya çalışır.

Bu kullanıcı odaklı yaklaşım, aynı zamanda kullanılabilirlik sorunlarının ve kullanıcı hatalarına karşı sistemin ne kadar dayanıklı olduğunun görünür olmasını sağlar. Kullanıcı yanlış veri girdiğinde ne olur? Zorunlu alan boş bırakıldığında sistem nasıl tepki verir? Ağ bağlantısı kısa süreli koptuğunda kullanıcı deneyimi nasıl etkilenir? Bu tür sorular, çoğu zaman ancak senaryo temelli sistem testleri sırasında açığa çıkar.

Sonuçta sistem testi, yazılımın sadece "iç teknik mantığının" değil, dışarıdan bakıldığında bir ürün olarak kullanıcıya sunduğu deneyimin de değerlendirilmesine hizmet eder. Bu nedenle sistem test planı hazırlanırken, teknik gereksinimlerin yanı sıra personel rolleri, iş akışları, hata mesajlarının anlaşılabilirliği ve geri bildirim mekanizmaları da göz önünde bulundurulmalıdır. Böylece sistem testi, yazılımın hem işlevsel doğruluğunu hem de kullanım kalitesini birlikte ele alan bir aşama hâline gelir.

1.6. Risk Temelli Sistem Testi Yaklaşımı

Gerçek projelerde zaman, insan kaynağı ve bütçe sınırsız değildir. Tüm olası senaryoları, veri kombinasyonlarını ve ortam koşullarını test etmek pratikte mümkün değildir. Bu nedenle sistem testinde sıklıkla risk temelli (risk-based) yaklaşım benimsenir. Risk temelli test, test eforunun; hatanın gerçekleşme olasılığı ve gerçekleştiğinde yaratacağı etki dikkate alınarak, sistemin en kritik alanlarına yönlendirilmesi anlamına gelir.

Risk temelli yaklaşımda ilk adım, risklerin tanımlanmasıdır. İş birimleri, geliştiriciler, test mühendisleri ve bazen son kullanıcılar bir araya gelerek "bu sistemde hata olması durumunda nerede en büyük zarar ortaya çıkar?" sorusuna cevap arar. Örneğin finansal işlemler, hasta kayıtları, kimlik doğrulama mekanizmaları veya kritik altyapı kontrol panelleri genellikle yüksek riskli alanlar olarak değerlendirilir. Bu alanlarda hem hata olasılığı hem de hata durumunda oluşabilecek kayıp ve itibar zedelenmesi daha yüksek olduğu için, sistem testinde bu modüllere daha yoğun odaklanmak gerekir.

İkinci adım, risklerin önceliklendirilmesi ve test kapsamına yansıtılmasıdır. Yüksek riskli alanlar için daha fazla test senaryosu üretilir, daha fazla sınır durumu ve hata senaryosu ele alınır, mümkün olduğunda otomasyonla sürekli regresyon testleri gerçekleştirilir. Orta ve düşük riskli alanlarda ise kapsam daha sınırlı tutulabilir; sadece tipik kullanım senaryolarını kapsayan bir test seti yeterli görülebilir. Bu sayede sınırlı kaynaklar, sistemin en kritik noktalarında en yüksek faydayı sağlayacak şekilde kullanılmış olur.

Risk temelli sistem testi, aynı zamanda şeffaflık ve hesap verebilirlik de sağlar. Test planında "bu alanlar yüksek riskli olarak tanımlandığı için şu yoğunlukta test edilmiştir, bu alanlar ise düşük riskli olduğu için kapsam daha sınırlı tutulmuştur" gibi açıklamaların yer alması, proje sonunda ortaya çıkabilecek tartışmaların daha rasyonel zeminde yürütülmesine yardımcı olur. Bir hata yaşandığında, bu hatanın daha önce neden yakalanamadığı sorusuna verilecek cevap, genellikle risk değerlendirmesi ve test kapsamının nasıl belirlendiğiyle doğrudan ilişkilidir.

Özetle, sistem testinin amacı, kapsamı ve karakteristikleri birlikte ele alındığında karşımıza şu tablo çıkar: Sistem testi, gereksinimlere ve kullanıcı senaryolarına dayanan, kara kutu yaklaşımıyla sistemin gözlemlenebilir davranışını uçtan uca değerlendiren ve sınırlı kaynaklar altında risk temelli önceliklendirme yapan bir kalite güvencesi aşamasıdır. Bu çerçevenin netleşmesi, ilerleyen bölümlerde ele alınacak olan süreç adımlarının, test tasarım tekniklerinin ve otomasyon stratejilerinin daha sağlam bir zemin üzerinde inşa edilmesine imkân verir.

Aynen, şimdi o iskeleti "2. Sistem Test Süreci" başlığı altında ders kitabı metnine dönüştürelim. Numara yapısını senin dediğin gibi 2.x olarak kullanıyorum.

2. Sistem Test Süreci

Sistem testi, tek bir adımdan ibaret bir faaliyet değil, birbirini besleyen bir dizi adımdan oluşan **yapısal bir süreç** olarak ele alınmalıdır. Bu süreç, kabaca testin planlanması, analiz ve tasarımının yapılması, uygun test ortamının hazırlanması, testlerin yürütülmesi, hataların yönetilmesi ve son olarak sürecin kapanışı şeklinde özetlenebilir. Bu bölümde, sistem test sürecinin bu temel aşamaları ayrıntılı olarak ele alınmaktadır.

2.1. Test Planlama

Test planlama, sistem test sürecinin temelini oluşturur. Planlama yapılmadan başlatılan test faaliyetleri, genellikle dağınık, tekrar eden veya kritik alanları gözden kaçıran bir yapıya dönüşür. Bu nedenle, sistem testi başlamadan önce hedeflerin, kapsamın, kriterlerin ve kaynakların açık biçimde tanımlanması gerekir.

2.1.1. Hedeflerin ve Başarı Kriterlerinin Tanımlanması

Test planlamasının ilk adımı, sistem testiyle **ne elde edilmek istendiğinin** netleştirilmesidir. Hedefler, test faaliyetinin odak noktasını belirler ve daha sonra yapılacak değerlendirmelerin referansını oluşturur. Örneğin:

- "Yeni geliştirilmiş otopark ücretlendirme modülünün, tanımlanmış tüm tarife kombinasyonlarında doğru çalıştığını doğrulamak."
- "Sistem üzerinde yapılan mimari değişikliklerin, mevcut kritik iş senaryolarını olumsuz etkilemediğini göstermek."

Bu tür hedeflere ek olarak, **başarı kriterlerinin** de tanımlanması gerekir. Başarı kriterleri, test sonunda "başardık mı?" sorusuna yanıt verebilmeyi sağlar. Örneğin:

- Planlanan sistem test senaryolarının en az %95'inin yürütülmüş olması,
- Kritik ve yüksek şiddetli açık hatanın kalmaması,
- Toplam hata sayısının belirli bir eşik değerin altında olması.

Bu kriterler, hem teknik ekipler hem de iş birimleri için **ortak bir referans noktası** oluşturur.

2.1.2. Giriş (Entry) ve Çıkış (Exit) Kriterlerinin Belirlenmesi

Sistem testine ne zaman başlanacağı ve ne zaman tamamlanmış sayılacağı konusu, süreç açısından kritik öneme sahiptir. Bu amaçla **giriş (entry)** ve **çıkış (exit)** kriterleri tanımlanır.

Giriş kriterleri, sistem testine başlamadan önce sağlanması gereken ön koşulları ifade eder. Örneğin:

- İlgili sürüme ait geliştirmelerin tamamlanmış ve entegrasyon testlerinden geçilmiş olması,
- Kritik fonksiyonlara yönelik birim testlerinin belirli bir kapsamda çalıştırılmış olması,
- Test ortamının kurulmuş ve temel smoke testlerin başarılı şekilde tamamlanmış olması,
- Gerekli test verilerinin hazırlanmış olması.

Çıkış kriterleri ise sistem testinin **tamamlanmış sayılabilmesi** için sağlanması gereken koşulları tanımlar. Örneğin:

- Planlanan test senaryolarının belirli bir yüzdesinin veya tamamının yürütülmüş olması,
- Belirli bir seviyenin üzerinde şiddete sahip açık hatanın bulunmaması,
- İş birimleri tarafından kritik senaryoların kabul edilmiş olması.

Bu kriterlerin baştan tanımlanması, test sürecinin ne zaman başlaması ve ne zaman bitmesi gerektiği konusunda tartışmaları azaltır, sürece netlik kazandırır.

2.1.3. Zaman, Kaynak ve Bütçe Planlaması

Sistem testleri sınırsız zaman, sınırsız insan kaynağı ve sınırsız bütçeyle yürütülmez. Bu yüzden planlama aşamasında:

- Testin tahmini süresi,
- Test ekibinin büyüklüğü ve rol dağılımı (test mühendisi, test lideri, otomasyon uzmanı vb.),
- Gerekli araç lisansları, donanım kaynakları ve ortam maliyetleri

gibi konuların değerlendirilmesi gerekir.

Burada önemli noktalardan biri, test süresi ve kapsamı arasındaki ilişkinin gerçekçi kurulmasıdır. Kısa bir zaman diliminde çok geniş kapsam hedeflenirse, yüzeysel ve yetersiz test yapılması kaçınılmaz hâle gelebilir. Bu nedenle, **risk temelli yaklaşım** ile kapsam önceliklendirmesi yapılmalı ve kaynaklar kritik alanlara yönlendirilmelidir.

2.2. Test Analiz ve Tasarımı

Planlama tamamlandıktan sonra, sistem testinin **neye göre ve nasıl yapılacağı** aşamasına geçilir. Bu aşamada gereksinimler analiz edilir, test senaryoları ve test durumları tasarlanır, gerekli test verileri belirlenir.

2.2.1. Gereksinimlerin Test Edilebilirlik Açısından İncelenmesi

Sistem testi, gereksinimlere dayanır. Bu nedenle gereksinimlerin:

- Açık,
- Çelişkisiz,
- Tutarlı,
- **Test edilebilir**

olması gerekir. “Sistem hızlı olmalıdır”, “kullanımı kolay olmalıdır” gibi muğlak ifadeler, test edilebilirlik açısından sorunludur. Test ekibi, gereksinimleri incelerken bu tür belirsizlikleri tespit eder ve analiz/geliştirme ekipleriyle birlikte bunların netleştirilmesi için çalışır.

Bu aşamada bazen bir **izlenebilirlik matrisi** (Requirements → Test Cases) hazırlanır. Bu matris, her gereksinimin en az bir test durumuyla ilişkilendirilmesini sağlar ve “Bu gereksinimi gerçekten test ettik mi?” sorusuna yanıt verir.

2.2.2. Test Senaryosu ve Test Durumu (Test Case) Tasarımı

Test senaryosu, genellikle **uçtan uca bir iş akışını** temsil eder. Örneğin:

- "Kullanıcının sisteme kayıt olması, giriş yapması ve park oturumu başlatması."
- "Abonelikli kullanıcının park oturumunu sonlandırması ve indirimli ücretlendirme ile karşılaşması."

Her senaryo, birden çok **test durumu (test case)** içerebilir. Test durumu, daha ayrıntılı bir yapıya sahiptir ve genellikle şu bileşenleri içerir:

- Test durumu kimliği (ID),
- İlgili gereksinim(ler),
- Ön koşullar,
- Adım adım uygulanacak işlemler,
- Beklenen sonuçlar,
- Son koşullar veya beklenen sistem durumu.

Bu düzeyde ayrıntı, testlerin **tekrarlanabilir** ve **izlenebilir** olmasını sağlar. Farklı test mühendisleri aynı test durumunu uyguladığında benzer sonuçlara ulaşabilmelidir.

2.2.3. Test Verisi Tasarımı ve Veri Setlerinin Hazırlanması

Test verisi, sistem testinin kalitesini doğrudan etkileyen bir unsurdur. Sadece "mutlu yol" (happy path) verilerini içeren test setleri, gerçek hayatta karşılaşılacak pek çok durumu gözden kaçırır. Bu nedenle test verisi tasarımı yapılırken:

- Geçerli (valid) veriler,
- Geçersiz (invalid) veriler,
- Sınır değerler,
- Farklı kombinasyonlar (tarife türü, süre, abonelik durumu vb.),
- Büyük hacimli veri senaryoları

dikkate alınmalıdır.

Ayrıca, kişisel veri içeren sistemlerde **anonimleştirme** ve **maskelenmiş veri** kullanımı, yasal ve etik gereklilikler açısından önem taşır. Test verisi üretimi, hem teknik gereksinimleri karşılayacak hem de veri gizliliğini ihlal etmeyecek şekilde kurgulanmalıdır.

2.3. Test Ortamının Hazırlanması

Sistem testinin güvenilir sonuçlar üretebilmesi için, testlerin yürütüldüğü ortamın **kontrollü ve tutarlı** olması gerekir. Bu nedenle test ortamının planlanması ve hazırlanması ayrı bir aşama olarak ele alınır.

2.3.1. Donanım, Yazılım ve Ağ Konfigürasyonu

Test ortamında kullanılacak:

- Sunucu veya konteyner altyapısı,
- İşletim sistemleri ve sürümleri,
- Veritabanı yönetim sistemleri ve sürümleri,

- Uygulama sunucuları,
- Ağ konfigürasyonu (IP aralıkları, güvenlik duvarı ayarları, load balancer vb.)

önceden tanımlanmalı ve dokümante edilmelidir. Bu bilgiler, testler sırasında ortaya çıkabilecek performans, bağlantı veya uyumluluk sorunlarının analizinde kritik rol oynar.

2.3.2. Entegrasyon Noktaları, Dış Sistemler ve Simülasyon (Stub/Mocks)

Modern sistemler, çoğu zaman farklı üçüncü taraf servislerle veya kurum içi diğer sistemlerle entegredir. Sistem testinde bu entegrasyonların nasıl ele alınacağı belirlenmelidir:

- Bazı entegrasyonlar gerçek sistemlerle, ancak test/sandbox ortamlarında yapılabilir.
- Bazı durumlarda ise **stub** veya **mock** bileşenlerle dış sistemler simüle edilir.

Örneğin gerçek banka sistemine erişim mümkün değilse, banka sisteminin beklenen davranışını taklit eden bir simülasyon bileşeni kullanılır. Bu tercih, hem maliyet hem de güvenlik açısından avantaj sağlayabilir; ancak simülasyonun gerçek sistemi yeterince temsil edip etmediği de dikkatle değerlendirilmelidir.

2.3.3. Ortamın Canlı Sistemle Benzerlik Düzeyi

Test ortamının, mümkün olduğunda canlı (production) ortamla **benzer konfigürasyona** sahip olması arzu edilir. Donanım kapasitesi, veritabanı boyutu, ağ topolojisi ve konfigürasyon ayarları gibi unsurlar, sistemin gerçek davranışını etkiler. Test ortamı ile canlı ortam arasında büyük farklar bulunması durumunda, sistem testinde elde edilen sonuçlar canlı ortamdaki davranışı tam olarak yansıtmayabilir.

Bu nedenle test ortamı tasarlanırken:

- Kritik konfigürasyonların canlı sistemle uyumlu olması,
- Performans ve yük testleri için yeterli kapasitenin sağlanması,
- Test ortamındaki verinin, yapısı ve dağılımı itibarıyla canlı veriyi temsil edebilmesi

gibi hususlar dikkate alınmalıdır.

2.4. Testlerin Yürütülmesi

Planlanan ve tasarlanan sistem testleri, hazırlanan ortam üzerinde bu aşamada fiilen uygulanır. Testlerin yürütülmesi, hem **manuel** hem de **otomatik** yöntemleri içerebilir.

2.4.1. Manuel Sistem Testi Yürütme Adımları

Manuel testlerde test mühendisi, her bir test durumunu adım adım izler ve sistemin verdiği tepkileri gözlemler. Genel süreç şu şekilde özetlenebilir:

1. İlgili test durumunun ön koşullarını sağlama (gerekli kullanıcı, veri veya sistem durumunun hazırlanması),
2. Test adımlarını sırasıyla uygulama,
3. Her adımda sistemin verdiği çıktıları ve yan etkileri gözlemleme,
4. Gözlenen sonuçları beklenen sonuçlarla karşılaştırma,
5. Tutarsızlık durumunda hata kaydı oluşturma,
6. Testin “geçti/başarısız” durumunu test yönetim aracına işleme.

Manuel testler, özellikle karmaşık kullanıcı arayüzleri, görsel doğrulama gerektiren senaryolar veya ilk kez denenen keşifsel testler için önemlidir.

2.4.2. Otomatik Sistem Testi Yürütme Adımları

Otomatik testlerde test senaryoları, uygun araç ve çerçeveler (framework) kullanılarak **script** veya **kod** hâline getirilir. Bu sayede:

- Her sürümde aynı testlerin tekrar tekrar,
- İnsan hatasına daha az açık şekilde,
- Daha kısa sürede yürütülmesi

mümkün olur.

Otomatik sistem testlerinin yürütülmesinde genel adımlar şunlardır:

1. Otomasyon script'lerinin derlenmesi ve paketlenmesi (gerekliyse),
2. Test ortamının hazır olduğunun doğrulanması,
3. Script'lerin manuel veya CI/CD boru hatları üzerinden tetiklenmesi,
4. Çalışma sırasında oluşan log ve raporların toplanması,
5. Başarısız testlerin analiz edilmesi ve hata kayıtlarının oluşturulması.

Özellikle regresyon testlerinde otomasyon, sistem testinin sürdürülebilir ve tekrarlanabilir olmasına büyük katkı sağlar.

2.4.3. Test Sonuçlarının Kayıt Altına Alınması ve İzlenebilirlik

Testlerin yürütülmesi kadar, **sonuçların kayıt altına alınması** ve geçmişle ilişkilendirilebilmesi de önemlidir. Bu amaçla genellikle bir test yönetim aracı veya en azından yapılandırılmış bir dokümantasyon yöntemi kullanılır.

Kayıt altına alınması gereken temel bilgiler şunlardır:

- Hangi test durumunun, kim tarafından, ne zaman çalıştırıldığı,
- Test sonucunun (başarılı/başarısız/engel vb.),
- Başarısız testler için ilgili hata kayıtlarının referansları,
- Gerekirse ekran görüntüleri, log dosyaları veya ek açıklamalar.

Bu kayıtlar sayesinde, hem test sürecinin ilerleyişi izlenebilir hem de belirli bir gereksinim veya modül için geçmiş test geçmişi geriye dönük olarak incelenebilir.

2.5. Hata (Defect) Yönetimi

Sistem testinin doğal sonuçlarından biri, **hata (defect)** bulunmasıdır. Hataların verimli şekilde ele alınabilmesi için sistematik bir hata yönetimi süreci gereklidir.

2.5.1. Hata Raporlama Süreci ve Hata Kayıt Formları

Bir test durumunun başarısız olması, her zaman sistemde bir hata olduğu anlamına gelmeyebilir; ancak önce bunun analiz edilmesi gerekir. Hata olduğu teyit edildiğinde, bu hata:

- Benzersiz bir kimlik (ID),
- Kısa ama açıklayıcı bir başlık,
- Yeniden üretme adımları (steps to reproduce),
- Beklenen ve gözlenen sonuçlar,
- Ortam bilgileri (test ortamı, sürüm, tarayıcı, cihaz vb.),

- İlgili log veya ekran görüntüleri

ile birlikte bir hata kayıt formuna işlenir.

İyi yazılmamış hata kayıtları, geliştiricilerin hatayı yeniden üretememesine ve sürecin tıkanmasına yol açabilir. Bu nedenle hata raporlama disiplininin, sistem test sürecinin ayrılmaz bir parçası olduğu unutulmamalıdır.

2.5.2. Hata Yaşam Döngüsü (Lifecycle)

Bir hata, kaydedildiği andan kapatıldığı ana kadar belirli aşamalardan geçer. Kuruma göre değişmekle birlikte tipik bir hata yaşam döngüsü şu adımları içerebilir:

- **New (Yeni):** Hata test mühendisi tarafından ilk kez kaydedildi.
- **Open (Açık):** Hata geliştirme ekibi tarafından incelenmek üzere alındı.
- **In Progress (Üzerinde Çalışılıyor):** Hata üzerinde düzeltme çalışması yapılıyor.
- **Resolved (Çözüldü):** Geliştirici, hatayı giderdiğini ve ilgili sürüme dahil ettiğini belirtti.
- **Retest (Yeniden Test):** Test mühendisi, düzeltmenin gerçekten işe yarayıp yaramadığını kontrol ediyor.
- **Closed (Kapatıldı):** Hata, test tarafından doğrulandı ve kapatıldı.
- **Rejected/Not a Bug:** Hata olarak raporlanan durumun, aslında gereksinimlere uygun davranış olduğu veya tekrar üretilemediği tespit edildi.

Bu yaşam döngüsünün net biçimde tanımlanması, hem test hem de geliştirme ekipleri arasında **iletişim ve sorumluluk paylaşımını** kolaylaştırır.

2.5.3. Öncelik (Priority) ve Şiddet (Severity) Kavramları

Hatalar aynı derecede önemli değildir. Bu nedenle her hata için genellikle iki ayrı etiket kullanılır:

- **Şiddet (severity):** Hatanın teknik açıdan ne kadar ciddi bir etki yarattığını gösterir.
    - Örneğin: sistemin tamamen çökmesi, veri kaybı, kritik güvenlik açığı.
- **Öncelik (priority):** Hatanın ne kadar **acil** düzeltilmesi gerektiğini ifade eder.
    - Örneğin: canlıya çıkıştan önce mutlaka giderilmesi gerekenler, sonraki sürüme ertelenebilecekler.

Bazı durumlarda yüksek şiddetli bir hata, geçici iş kuralları veya alternatif çözüm yolları ile bir süre idare edilebilir; buna karşın düşük şiddetli ama kullanıcı deneyimini ciddi biçimde bozan bir hata yüksek önceliğe sahip olabilir. Bu denge, iş birimleri ve proje yöneticileriyle birlikte değerlendirilir.

2.6. Çıkış Kriterlerinin Değerlendirilmesi ve Test Kapanışı

Sistem test sürecinin sonunda, sadece "testler bitti" denip geçilmez; testin hedeflere ulaşıp ulaşmadığı, risklerin ne ölçüde azaltıldığı ve bir sonraki adım için sistemin ne kadar hazır olduğu değerlendirilir.

2.6.1. Test Özet Raporları ve Metrikler

Test kapanışı aşamasında genellikle bir test özet raporu hazırlanır. Bu raporda:

- Planlanan ve yürütülen test durumlarının sayısı,
- Başarılı ve başarısız testlerin dağılımı,
- Bulunan hata sayıları ve şiddet dağılımları,
- Hataların modüllere veya işlevlere göre dağılımı,
- Gözlenen genel kalite durumu

gibi metrikler sunulur. Bu metrikler, hem teknik hem de yönetim seviyesinde kararlar için girdi sağlar.

2.6.2. Kabul/Red Kararları ve Risk Değerlendirmesi

Test özet raporu ve mevcut hata durumu temel alınarak, ilgili sürümle ilgili bir **kabul veya erteleme** kararı verilir. Bu karar, yalnızca "kaç test geçti, kaç test kaldı?" sayılarından ibaret değildir; aynı zamanda:

- Açık kalan hataların şiddeti ve kapsamı,
- Bu hataların iş süreçlerine, güvenliğe ve kullanıcı deneyimine olası etkileri,
- Canlı ortamda ortaya çıkması muhtemel risklerin kabul edilebilirlik düzeyi

gibi unsurlar dikkate alınır. Bazı durumlarda belirli düzeyde risk kabul edilerek canlıya çıkma kararı verilebilir; bazı durumlarda ise kritik hatalar nedeniyle sürüm ertelenebilir.

2.6.3. Öğrenilen Dersler (Lessons Learned) ve Sürekli İyileştirme

Sistem test süreci, sadece hataların bulunup kapatıldığı bir faaliyet olarak görülmemelidir. Her test döngüsü, ekibe sonraki projelerde veya sürümlerde fayda sağlayacak **deneyimler ve dersler** kazandırır. Bu amaçla test kapanışı sonrasında:

- Nelerin iyi gittiği,
- Nelerin beklenenden fazla zaman veya efor harcattığı,
- Hangi tür hataların gereksinim, tasarım veya geliştirmenin daha erken aşamalarında yakalanabileceği,
- Test tasarımında veya ortam hazırlığında hangi iyileştirmelerin yapılabileceği

üzerine kısa bir değerlendirme yapılması yararlıdır.

Bu tür geri bildirimler, hem test süreçlerinin olgunlaşmasına hem de genel yazılım geliştirme yaşam döngüsünün daha verimli hâle gelmesine katkı sunar. Böylece sistem testi, sadece mevcut sürümün kalitesini güvence altına almakla kalmaz, gelecekteki sürümlerin ve projelerin de daha iyi planlanmasına zemin hazırlar.

3. Sistem Testi Türleri ve Boyutları

Sistem testi, tek tip bir faaliyetten ibaret değildir. Aynı sistem üzerinde, farklı kalite özelliklerini hedefleyen pek çok test türü uygulanır. Genel olarak bu türler iki ana grupta toplanabilir:

- Fonksiyonel sistem testleri: Sistem "doğru işi" yapıyor mu?
- Fonksiyonel olmayan sistem testleri: Sistem, bu işi "nasıl" yapıyor? (performans, güvenlik, kullanılabilirlik vb.)

Bunlara ek olarak, regresyon testleri ve keşifsel testler de sistem testinin önemli boyutlarıdır. Bu bölümde bu türler ayrıntılı biçimde ele alınmaktadır.

3.1. Fonksiyonel Sistem Testleri

Fonksiyonel sistem testleri, sistemin tanımlanmış fonksiyonel gereksinimlerini yerine getirip getirmediğini değerlendirir. Bu testler genellikle şu üç eksen etrafında kurgulanır:

- Gereksinim temelli testler,
- Kullanım senaryosu (use case) temelli testler,
- İş süreci ve uçtan uca (end-to-end) testler.

3.1.1. Gereksinim Temelli Fonksiyonel Testler

Bu yaklaşımda her test, doğrudan belirli bir gereksinime bağlanır. Örneğin:

GREQ-LOGIN-01: “Kullanıcı geçerli e-posta ve parola ile sisteme giriş yapabilmelidir.”

Bu gereksinim için sistem testi düzeyinde bir test durumu şu şekilde tanımlanabilir:

- Ön koşul: Kullanıcı hesabı sistemde kayıtlı ve aktiftir.
- Adımlar:
    - Giriş ekranına gidilir.
    - Geçerli e-posta ve parola girilir.
    - “Giriş” butonuna tıklanır.
- Beklenen sonuç: Kullanıcı ana sayfaya yönlendirilir; oturum açılmıştır.

API tabanlı bir sistemde aynı gereksinimi, örneğin REST Assured ile otomatik test ederek doğrulamak mümkündür:

```
// Örnek: GREQ-LOGIN-01 için basit bir API testi
@Test
void shouldLoginWithValidCredentials() {
    given()
        .contentType("application/json")
        .body("""
              { "email": "user@example.com", "password": "P@ssw0rd" }
              """)
    .when()
        .post("/api/auth/login")
    .then()
        .statusCode(200)
        .body("token", notNullValue());
}
```

Burada test kodu, belirli bir gereksinimi (GREQ-LOGIN-01) doğrudan doğrulamak üzere yazılmıştır. Sistem test yönetiminde bu test durumunun gereksinimle ilişkilendirilmesi, izlenebilirlik için önemlidir.

3.1.2. Kullanım Senaryosu (Use Case) Tabanlı Testler

Use case tabanlı testler, tek tek fonksiyonlardan ziyade kullanıcının sistemi nasıl kullandığını temel alır. Örneğin “Kayıt ol ve ilk kez giriş yap” senaryosu:

- Kullanıcı kayıt ekranını açar.
- Formu doldurur, kayıt olur.
- Aktivasyon e-postasını alır ve hesabını aktive eder.
- Giriş yapar ve profil sayfasına yönlendirilir.

Bu tip testlerde, gereksinimler arası akış önemlidir. İki farklı gereksinimi tek test içinde sınamak mümkündür; önemli olan, senaryonun kullanıcı bakış açısıyla anlamlı olmasıdır.

Basitleştirilmiş bir senaryoyu otomasyon ile test etmek için adım adım yaklaşım kullanılabilir. Örneğin bir uçtan uca API testi (kayıt + login) aşağıdaki gibi kurgulanabilir:

```
@Test
void userRegistrationAndFirstLoginFlow() {
    // 1. Kayıt ol
    String email = "newuser@example.com";
    String password = "Secret123!";

    given()
        .contentType("application/json")
        .body("""
              { "email": "%s", "password": "%s" }
              """.formatted(email, password))
    .when()
        .post("/api/auth/register")
    .then()
        .statusCode(201);

    // 2. Aktivasyon simülasyonu (test ortamında doğrudan endpoint ile)
    given()
        .queryParam("email", email)
```

```
    .when()
        .post("/api/auth/activate")
    .then()
        .statusCode(200);

    // 3. Giriş yap
    given()
        .contentType("application/json")
        .body("""
              { "email": "%s", "password": "%s" }
              """.formatted(email, password))
    .when()
        .post("/api/auth/login")
    .then()
        .statusCode(200)
        .body("token", notNullValue());
}
```

Bu test örneği, birçok gereksinimi tek bir use case akışı üzerinden sistem düzeyinde doğrulamaktadır.

3.1.3. İş Süreci (Business Process) ve Uçtan Uca (End-to-end) Testler

İş süreci testleri, tek bir modül veya ekranı değil, kurumun gerçek iş akışlarını baştan sona ele alır. Örneğin bir otopark sistemi için:

- "Müşterinin otoparka girmesinden, çıkış sonrası fatura kesilmesine kadar geçen tüm sürecin test edilmesi".

Bu tür uçtan uca testler, çoğu zaman birden çok sistem bileşenini kapsar:

- Web veya mobil arayüz,
- Arka uç servisler,
- Veritabanı,
- Dış ödeme servisleri vb.

Bu nedenle end-to-end testler, sistem testinin en geniş kapsamlı fonksiyonel testleridir. Otomasyon tarafında çoğu zaman UI test araçları (Selenium, Cypress vb.) ile birlikte API ve veritabanı kontrolleri de kullanılır. Örneğin Selenium ile bir uçtan uca testte, kullanıcının park başlatma ve bitirme akışı simüle edilebilir, ardından API veya DB üzerinden ücretlendirme doğrulanabilir.

3.2. Fonksiyonel Olmayan Sistem Testleri

Fonksiyonel olmayan testler, sistemin "ne yaptığı"ndan çok "nasıl davrandığı" ile ilgilenir. Yaygın başlıklar:

- Performans ve yük,
- Stres ve dayanıklılık,
- Güvenlik,
- Kullanılabilirlik,
- Uyumluluk ve taşınabilirlik,
- Güvenilirlik ve hata toleransı.

Bu test türleri genellikle sistem testi kapsamında ele alınır ve çoğu zaman özel araçlar ve ortamlar gerektirir.

3.2.1. Performans ve Yük Testleri (Performance & Load Testing)

Performans testi, sistemin belirli iş yükleri altındaki cevap sürelerini ve kaynak kullanımını ölçer. Yük testi ise sistemin beklenen maksimum kullanıcı sayısı, işlem hacmi veya veri hacmi altındaki davranışını inceler. Örneğin; "Sistem, 500 eşzamanlı kullanıcı, saniyede 50 istek yükü altında, ortalama cevap süresi 2 saniyenin altında kalmalıdır" gibi bir gereksinim performans/yük testi ile doğrulanır. Basit bir performans senaryosu, k6 gibi bir araçla şu şekilde ifade edilebilir (JS tabanlı script):

```
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
  vus: 50,          // sanal kullanıcı sayısı
  duration: '1m',   // test süresi
};

export default function () {
```

```
  const res = http.get('https://example.com/api/parking/tariffs');
  check(res, {
    'status is 200': (r) => r.status === 200,
    'response time < 2000ms': (r) => r.timings.duration < 2000,
  });
  sleep(1);
}
```

Bu test, 1 dakika boyunca 50 sanal kullanıcıyla endpoint'in cevap süresini ölçer.

3.2.2. Stres ve Dayanıklılık (Endurance) Testleri

Stres testi, sistemin tasarlanan kapasitenin çok üzerinde yük altında nasıl davrandığını inceler. Amaç, sistemin nerede "kırıldığını" görmek ve bu kırılma noktasında:

- Hataların kontrollü olup olmadığı,
- Sistem çökse bile veri bütünlüğünün korunup korunmadığı

gibi hususları gözlemlemektir. Dayanıklılık (endurance) testi ise sistemin uzun süreli yük altında kararlılığını ölçer. Örneğin; 8–24 saat boyunca orta düzey yük altında bellek sızıntısı (memory leak), kaynak tüketimi veya performans düşmesi olup olmadığına bakılır.

3.2.3. Güvenlik Testleri (Security Testing)

Güvenlik testleri, sistemin:

- Yetkisiz erişime karşı dayanıklılığı,
- Veri gizliliğini ve bütünlüğünü koruma düzeyi,
- Girdi doğrulama başarısı (SQL injection, XSS vb.)

gibi konularda incelenmesini kapsar.

Sistem testinde güvenlik genellikle şu yollarla ele alınır:

- Kimlik doğrulama ve yetkilendirme senaryolarının test edilmesi,
- Yetkisiz kullanıcı ile yasak işlemlerin denenmesi,
- Girdi alanlarına kötü niyetli verilerin gönderilmesi,
- Şifreleme ve sertifika kontrollerinin doğrulanması.

Örneğin, yetkisiz erişim testi için basit bir API testi:

```
@Test
void shouldRejectAccessWithoutToken() {
    given()
        .when()
        .get("/api/parking/sessions")
    .then()
        .statusCode(401); // Unauthorized
}
```

Daha ileri düzey güvenlik testleri için genellikle özel araçlar (örneğin dinamik güvenlik tarayıcıları) ve uzmanlık gereklidir.

3.2.4. Kullanılabilirlik (Usability) Testleri

Kullanılabilirlik testleri, sistemin ne kadar kolay öğrenilebilir, anlaşılabilir ve kullanılabilir olduğunu inceler. Bu testler çoğu zaman:

- Gerçek ya da temsilî kullanıcılarla yapılan oturumlar,
- Görev tamamlama sürelerinin ölçümü,
- Hata yapma oranlarının ve kullanıcı memnuniyetinin gözlemlenmesi

şeklinde yürütülür. Sistem testinde kullanılabilirlik boyutunu tamamen ihmal etmek, teknik olarak "doğru" çalışan ama kullanıcı açısından zor ve yorucu bir ürün ortaya çıkmasına neden olabilir. Bu nedenle, kritik iş akışları için en azından temel kullanılabilirlik kontrolleri yapılmalıdır.

3.2.5. Uyumluluk (Compatibility) ve Taşınabilirlik (Portability) Testleri

Uyumluluk testleri, sistemin:

- Farklı tarayıcılarda (Chrome, Firefox, Edge, Safari vb.),
- Farklı işletim sistemlerinde,
- Farklı ekran boyutlarında ve cihaz türlerinde

tutarlı davranıp davranmadığını inceler. Mobil ve web uygulamalarında bu tür testler, sistem testinin önemli bir parçasıdır. Örneğin aynı fonksiyonun mobil tarayıcıda, masaüstü tarayıcıda ve native mobil uygulamada nasıl çalıştığı karşılaştırılabilir. Taşınabilirlik testleri ise sistemin farklı altyapı veya platformlara görece düşük eforla taşınabilme yeteneğini değerlendirir (örneğin on-premise → bulut).

3.2.6. Güvenilirlik (Reliability) ve Hata Toleransı Testleri

Güvenilirlik testleri, sistemin belirli bir süre boyunca kesintisiz ve hatasız çalışma kapasitesini ölçmeye yöneliktir. Özellikle kritik sistemlerde (sağlık, finans, endüstriyel kontrol vb.) bu boyut önemlidir.

Hata toleransı testleri ise:

- Ağ kesintileri,
- Donanım hataları,
- Servis erişilemezliği

gibi durumlarda sistemin:

- Hataları zarif biçimde karşılayıp karşılamadığı (graceful degradation),
- Otomatik iyileşme (self-healing) mekanizmalarının çalışıp çalışmadığı

gibi özelliklerini inceler. Örneğin, dış servise ulaşılamadığında sistemin kullanıcıya anlaşılabilir bir hata göstermesi ve veri bütünlüğünü bozmaması beklenir.

3.3. Regresyon Testleri ve Sistem Testi İlişkisi

Regresyon testleri, temel olarak: "Sistemde yaptığımız yeni değişiklikler, daha önce çalışan fonksiyonları bozdu mu?" sorusunu yanıtlamaya çalışır. Sistem testinin her döngüsüyle doğrudan ilişkili, sürekli tekrar eden bir faaliyettir.

3.3.1. Regresyon Testinin Amacı

Her yeni sürümde:

- Yeni özellikler eklenir,
- Mevcut hata düzeltmeleri yapılır,
- Alt yapı güncellemeleri gerçekleştirilir.

Bu değişiklikler, doğrudan hedeflenen fonksiyonları iyileştirse bile, beklenmedik yan etkilere yol açabilir. Regresyon testinin amacı, daha önce başarıyla çalışan kritik senaryoları tekrar çalıştırarak, bu tür yan etkileri yakalamaktır.

3.3.2. Sistem Test Sürecinde Regresyon Stratejileri

Regresyon testleri için sistem testinde farklı stratejiler uygulanabilir:

- Tam regresyon: Tüm kritik sistem test senaryolarının her sürümde yeniden çalıştırılması. Maliyetlidir, otomasyon yoksa pratik değildir.
- Seçimli regresyon: Değişiklik yapılan modüllere en çok temas eden, en kritik iş akışlarının seçilerek tekrar test edilmesi.

- Risk temelli regresyon: Risk analizi sonucu yüksek riskli alanları kapsayan test setinin her sürümde koşulması; düşük riskli alanlarda ise daha seyrek test yapılması.

Otomasyon, regresyon testlerinin sürdürülebilirliği için kritik öneme sahiptir. Örneğin:

```
# CI/CD pipeline'da her build sonrası temel regresyon setini çalıştırmak için
mvn test -DtestSuite=regression
```

veya

```
# Örnek GitLab CI parçası
regression-tests:
  stage: test
  script:
    - mvn test -Dgroups=regression
  when: on_success
```

Bu şekilde, sistem testinin kritik senaryoları her değişiklikte tekrar doğrulanabilir.

3.4. Keşifsel (Exploratory) Test ve Sistem Testi

Keşifsel test, önceden ayrıntılı adımların yazılmadığı, test mühendisinin deneyim ve sezgisine dayanan bir yaklaşımdır. Sistem testi içinde, özellikle yeni geliştirilen veya riskli görülen alanlar için değerli bir tamamlayıcıdır.

3.4.1. Senaryo Bazlı Keşifsel Test

Tamamen rastgele denemelerden farklı olarak, keşifsel test çoğu zaman yüksek seviyeli senaryolar üzerinden yürütülür. Örneğin:

- "Deneyimli bir kullanıcı gibi sistemi yoğun şekilde kullan"
- "Hatalı veri girme, geri alma, iptal etme gibi işlemleri bolca dene"
- "Ağ kesintisi simüle ederek kritik akışları zorla"

Bu tür oturumlarda test mühendisi:

- Farklı veri kombinasyonlarını,
- Farklı gezinme yollarını,
- Farklı kullanıcı rollerini

deneyerek, dokümante test durumlarında düşünülmemiş hataları yakalamaya çalışır. Oturum sonunda bulunan hatalar ve ilginç gözlemler not edilir.

3.4.2. Keşifsel Testin Planlı Testlerle Dengelenmesi

Sistem testini tamamen keşifsel yaklaşıma dayandırmak, izlenebilirlik ve tekrar edilebilirlik açısından sorunlu olur. Bu nedenle:

- Ana iskelet, gereksinim temelli ve senaryoya dayalı planlı testlerden oluşmalı,
- Keşifsel testler ise özellikle riskli veya yeni alanlarda tamamlayıcı rol oynamalıdır.

Dengeli bir yaklaşımda yapılması gerekenler:

- Kritik gereksinimler için mutlaka dokümante test durumları tanımlanır.
- Her sürüm için belirli süreli keşifsel test oturumları planlanır (örneğin 2–3 saat).
- Keşifsel test sırasında keşfedilen önemli senaryolar, daha sonra kalıcı test durumlarına dönüştürülür.

Bu sayede sistem testi, hem yapılandırılmış ve izlenebilir hem de esnek ve keşfe açık bir yapıya kavuşur. Böyle bir denge, özellikle karmaşık ve sürekli evrilen yazılım sistemlerinde kalite güvencesinin etkinliğini belirgin şekilde artırır.

4. Sistem Test Tasarım Teknikleri

Sistem testi, “hadi biraz deneriz, bakarız” seviyesinden çıkarılıp, **yöntemli** hâle getirildiğinde değer kazanır. Test tasarım teknikleri tam burada devreye girer; aynı sistemi, aynı süre içinde, çok daha kapsamlı ve sistematik biçimde sorgulamanı sağlar. Bu bölümde, sistem seviyesinde en sık kullanılan test tasarım tekniklerini inceleyeceğiz.

4.1. Gereksinim Tabanlı Test Tasarımı

Gereksinim tabanlı test tasarımında, her test doğrudan bir veya daha fazla **gereksinim** ile ilişkilendirilir. Amaç:

- Her gereksinimin en az bir testle kapsandığından emin olmak,
- Gereksinim → Test Case izlenebilirliğini sağlamak,
- “Şu gereksinimi test ettik mi?” sorusuna net cevap verebilmek.

Örneğin otopark sisteminde:

- **GREQ-PARK-01:** “Kullanıcı, kayıtlı aracı için park oturumu başlatabilmelidir.”
- **GREQ-PARK-02:** “Park oturumu sonlandırıldığında ücret, tarife kurallarına göre hesaplanmalıdır.”

Bu gereksinimlere karşılık gelen test durumları:

- TC-01: Geçerli araçla park başlatma (GREQ-PARK-01)
- TC-02: Kayıtlı olmayan araçla park başlatmaya çalışma (GREQ-PARK-01 – negatif senaryo)
- TC-03: 2 saatlik park için ücret hesaplama (GREQ-PARK-02)
- TC-04: Abonelikli kullanıcı için indirimli ücret hesaplama (GREQ-PARK-02 + abonelik gereksinimi)

Basit bir izlenebilirlik matrisi:

Gereksinim ID	**Açıklama**	**İlgili Test Case’ler**
GREQ-PARK-01	Park oturumu başlatma	TC-01, TC-02
GREQ-PARK-02	Ücret hesaplama	TC-03, TC-04, TC-05 (sınır)
GREQ-AUTH-01	Kimlik doğrulama	TC-10, TC-11

Sistem testi seviyesinde bu yaklaşım, özellikle **eksik testleri tespit etme** konusunda güçlüdür: tabloda gereksinimi olup test cas’i olmayan her satır kırmızı bayraktır.

4.2. Use Case ve User Story Tabanlı Test Tasarımı

Gereksinim tabanlı yaklaşım “ne yapılmalı?”ya, use case ve user story tabanlı yaklaşım ise “kullanıcı bunu **nasıl** yapıyor?” sorusuna yaslanır. Örneğin bir user story:

US-01: “Bir kullanıcı olarak, park oturumumu mobil uygulamadan başlatmak istiyorum ki otoparka girişte beklemek zorunda kalmayayım.”

Bu story için tipik kabul kriterlerinden (acceptance criteria) türetilmiş senaryolar:

- Geçerli plaka ve aktif abonelik ile park başlatma,
- Abonelik yokken park başlatma,
- Sistemde kayıtlı olmayan plaka ile park başlatmaya çalışma,
- Lokasyon servisi kapalıyken park başlatma (hata mesajı vb.).

Bu yaklaşımla:

- Testler iş analistlerinin ve product owner'ların diliyle konuşur.
- "Kullanıcı hikâyesi tamamlandı mı?" sorusunu teknik test sonucuna bağlamak kolaylaşır.

Küçük bir örnek test durumu (user story'den türetilmiş):

TC-US01-01 – Aktif abonelikle park başlatma

- Ön koşul: Kullanıcının ABC-123 plakalı aracı kayıtlı ve aboneliği aktiftir.
- Adımlar:
    1. Kullanıcı mobil uygulamaya giriş yapar.
    2. "Park Başlat" ekranını açar.
    3. Plaka listesinden ABC-123'ü seçer ve "Başlat" butonuna basar.
- Beklenen sonuç: Park oturumu "ACTIVE" olarak oluşturulur, ekranda kalan süre gösterilir.

Bu şekilde her user story için 2–5 arası sistem testi senaryosu tanımlanabilir.

4.3. Durum Geçiş (State-based) Testleri ve Durum Diyagramları

Bazı sistemler, doğası gereği **durum makineleri** gibi davranır: nesneler veya oturumlar belirli durumlarda bulunur ve belirli olaylara göre diğer durumlara geçer. Örneğin ParkingSession için basit bir durum diyagramı:

- **NEW** → (park başlat) → **ACTIVE**
- **ACTIVE** → (park bitir) → **COMPLETED**
- **NEW** → (iptal) → **CANCELLED**
- **ACTIVE** → (zaman aşımı) → **EXPIRED**

Durum tabanlı test tasarımı şu soruları sorar:

- Tüm geçerli geçişler test edildi mi?
- Geçersiz geçişlere sistem nasıl tepki veriyor?

Örnek geçiş testleri:

- NEW → ACTIVE: Geçerli araç ve konum bilgisi ile park başlat.
- ACTIVE → COMPLETED: Devam eden bir oturumu bitir.
- ACTIVE → NEW: Buna izin verilmeli mi? Verilmemeli ise nasıl bir hata dönmeli?
- COMPLETED → ACTIVE: Tamamlanmış oturumu tekrar aktif etmeye çalıştır; hata beklenir.

Bu yaklaşım, "**geçersiz adımlar**"ın yakalanmasında çok etkilidir. Çoğu üretim hatası, iş mantığının izin vermemesi gereken bir duruma "kazaen geçilmesi"nden kaynaklanır.

4.4. İş Akışı (Workflow) ve Süreç Modellerine Dayalı Test

Kurumsal sistemlerde, iş kuralları genellikle BPMN diyagramları veya süreç modelleri ile ifade edilir. Örneğin bir "fatura onay süreci":

1. Fatura oluşturulur.
2. İlk onaylayan inceler (status: PENDING_LEVEL_1).
3. Onaylarsa ikinci onaylayana gider (PENDING_LEVEL_2).
4. O da onaylarsa faturanın durumu APPROVED olur, aksi hâlde REJECTED.

Workflow tabanlı test tasarımında:

- Süreçteki her yol (path) için test senaryosu çıkarılır,
- Hata yolları (örneğin “ikinci onaylayana ulaşamama”) ayrıca ele alınır.

Basit bir süreç modeli için test örnekleri:

- Normal akış: Her iki onaylayanın da onay verdiği durum.
- İlk onaylayanın reddettiği durum.
- İlk onaylayanın onayladığı, ikinci onaylayanın reddettiği durum.
- Onaylayanın sistemde yetkisiz olduğu bir durumda onay/ret denemesi.

Bu testler, sistem testinde **iş süreçlerinin doğru yürütüldüğünü** ve doğru rollere doğru adımların atandığını doğrular.

4.5. Eşdeğer Bölgeleme ve Sınır Değer Analizi Sistem Seviyesinde

Eşdeğer bölgeleme (equivalence partitioning) ve sınır değer analizi (boundary value analysis) genellikle birim ve entegrasyon testlerinde anlatılır; fakat sistem seviyesinde de çok etkilidir. Eşdeğer bölgeleme, giriş değerlerini mantıksal olarak benzer davranış gruplarına ayırır. Örneğin park ücreti için:

- 0–1 saat → sabit 20 TL,
- 1–3 saat → 20 TL + her ek saat için 10 TL,
- 3+ saat → maksimum 50 TL.

Burada süreyi şu bölgelere ayırabiliriz:

- Bölge 1: 0 < süre ≤ 1 saat,
- Bölge 2: 1 < süre ≤ 3 saat,
- Bölge 3: süre > 3 saat.

Her bölgeden **temsilci bir değer** seçip sistem seviyesinde senaryo oluşturabiliriz:

- 0.5 saat, 2 saat, 4 saat gibi.

Sınır değer analizi ise bölge sınırlarının hemen altı, kendisi ve hemen üstü için test üretir. Yukarıdaki örnekte:

- 1 saat için 59 dk, 60 dk, 61 dk (1 sınırı),
- 3 saat için 179 dk, 180 dk, 181 dk (3 sınırı).

Basit bir sistem testi tablosu:

Süre (dk)	**Beklenen Ücret (TL)**	**Not**
59	20	1 saate kadar sabit
60	20	Sınır: 1 saat
61	30	1–3 saat aralığına girdi
179	40	3 saatten 1 dk önce
180	40 veya 50 (kurala göre)	
181	50	Maksimum ücret tavanı

Bu değerler, sistem testinde **gerçek oturumlar** üzerinden yürütülerek doğrulanabilir. Böylece yalnızca "rastgele birkaç süre" yerine, mantıksal olarak anlamlı sınırlar test edilmiş olur.

4.6. Karar Tabloları ve Karar Ağaçları ile Sistem Testi

Karar tabloları (decision tables), çok sayıda giriş koşulunun farklı kombinasyonlarını sistematik biçimde test etmek için kullanılır. Örneğin park ücretlendirmede şu kuralları düşünelim:

- Koşul 1: Kullanıcı abonelikli mi? (Evet/Hayır)
- Koşul 2: Araç VIP mi? (Evet/Hayır)
- Koşul 3: Süre > 3 saat mi? (Evet/Hayır)

Basit karar tablosu:

K1: Abone	**K2: VIP**	**K3: Süre>3h**	**Beklenen Davranış**
H	H	H	%50 indirim, ancak 50 TL tavan uygulanmaz
H	H	H Değil	%50 indirim
H	H Değil	H	%50 indirim, 50 TL tavanı geçilmez
H	H Değil	H Değil	%50 indirim
H Değil	H	H	Normal ücret, VIP tavanı (örn. 70 TL)
H Değil	H	H Değil	Normal ücret
H Değil	H Değil	H	Normal ücret, 50 TL tavanı
H Değil	H Değil	H Değil	Normal ücret

Bu tablodan **doğrudan 8 farklı sistem test senaryosu** çıkarılabilir. Böylece "VIP + abone + uzun süre" gibi gözden kaçması kolay kombinasyonlar da kapsanmış olur. Karar ağaçları ise aynı mantığı görsel hâle getirir; her dal bir koşulu, yapraklar ise beklenen sonucu gösterir. Özellikle karmaşık iş kurallarında, karar ağacı üzerinden test tasarlamak hem geliştirici hem testçi için kavrayışı kolaylaştırır.

4.7. Senaryo Kombinasyonları ve Test Kapsamı (Coverage) Kavramı

Sistem seviyesinde test tasarlarken hep şu problemle karşılaşılır: "Olası senaryo kombinasyonları patlıyor, hepsini test etmem imkânsız. Ne kadarını test ettiğim yeterli sayılır?" Burada **kapsam (coverage)** kavramı devreye girer. Kod kapsamı (statement, branch coverage) daha çok birim testlerinde konuşulur, fakat sistem testinde de benzer bir mantıkla:

- **Gereksinim kapsamı:**

Kaç gereksinimin en az bir sistem test senaryosu ile kapsandığı.

- **Senaryo/kullanım yolu kapsamı:**

Use case içindeki olası yolların kaçı için test yazıldığı.

- **Karar kombinasyonu kapsamı:**

Karar tablosunda tanımlı kombinasyonların kaçı için senaryo üretildiği.

Tam kapsam çoğu zaman pratik değildir. Bu nedenle:

- Risk temelli önceliklendirme (kritik gereksinim ve akışlara daha fazla senaryo),
- Eşdeğer bölgeleme (benzer davranış gruplarından temsilci seçme),
- Karar tablosu ve durum tabanlı tekniklerin dengeli kullanımı

ile "**makul kapsam**" hedeflenir.

Basit bir planlama örneği:

- Tüm yüksek riskli gereksinimler → en az 2 senaryo (pozitif + negatif).
- Orta riskli gereksinimler → en az 1 pozitif senaryo.
- Düşük riskli gereksinimler → sadece ana akış (happy path).

Bu yapı, sistem testini hem **metodik** hem de **yönetilebilir** kılar.

5. Sistem Testinde Ortam ve Veri Yönetimi

Sistem testinin yalnızca "hangi senaryoları koşacağız?" sorusuna indirgenmesi, pratikte ciddi eksikliklere yol açar. Aynı test senaryosu, farklı ortam ve veri koşullarında tamamen farklı sonuçlar üretebilir. Bu nedenle sistem testinin güvenilir ve tekrarlanabilir olması için **test ortamı** ve **test verisinin** bilinçli biçimde tasarlanması, yönetilmesi ve korunması gerekir. Bu bölümde, sistem testinde ortam ve veri yönetiminin temel boyutları ele alınmaktadır.

5.1. Test Ortamının Mimarisi ve Katmanları

Test ortamı, sistem testlerinin yürütüldüğü **teknik altyapıyı** ifade eder. Bu altyapı genellikle aşağıdaki katmanlardan oluşur:

- **Uygulama katmanı:** Web sunucuları, uygulama sunucuları, mikro servisler, API ağ geçitleri (API gateway) vb.
- **Veri katmanı:** Veritabanları, önbellek (cache) bileşenleri, mesaj kuyrukları, dosya depolama sistemleri.
- **Sunum katmanı:** Web arayüzleri, mobil uygulamalar, admin panelleri, dış entegrasyon ekranları.
- **Altyapı katmanı:** Sanal sunucular, konteyner platformları (Docker/Kubernetes), yük dengeleyiciler, güvenlik duvarları, ağ topolojisi.

Sistem testinde hedef, bu katmanların **birlikte çalıştığı** koşulları temsil eden bir ortamda test yapmaktır. Yalnızca uygulama katmanını ayağa kaldırıp veritabanını "in-memory" bir yapı ile simüle etmek, pek çok entegrasyon ve performans sorununu gizleyebilir. Bu yüzden test ortamı tasarlanırken:

- Hangi bileşenlerin zorunlu olduğu,
- Hangi bileşenlerin simüle edilebileceği (mock/stub),
- Hangi bileşenlerin mutlaka gerçek konfigürasyonda çalışması gerektiği

önceden belirlenmelidir.

İyi tasarlanmış bir test ortamı, sadece sistem testine değil; performans, güvenlik, regresyon ve hatta eğitim/demonstrasyon gibi amaçlara da hizmet edebilir.

5.2. Canlı Ortam ile Test Ortamı Farklılıkları ve Riskler

Teoride ideal olan, test ortamının canlı (production) ortamla **mümkün olduğunca benzer** olmasıdır. Ancak pratikte, maliyet, lisans, donanım kapasitesi ve operasyonel kısıtlar nedeniyle test ortamı ile canlı ortam arasında çeşitli farklar bulunur:

- Daha düşük donanım kapasitesi (CPU, RAM, disk, IO).
- Farklı veritabanı boyutu ve veri dağılımı.
- Farklı ağ topolojisi, daha basit güvenlik kuralları.
- Bazı entegrasyonların gerçek yerine stub/mock ile çalıştırılması.

Bu farklar, sistem testinin sonuçlarını **yanıltıcı** hâle getirebilir. Örneğin, test ortamında az sayıda kayıt bulunan bir veritabanında çok hızlı çalışan bir sorgu, canlı ortamda milyonlarca kayıt üzerinde çalışırken zaman aşımına uğrayabilir. Benzer şekilde, test ortamında güvenlik duvarı kısıtları hafifletilmiş bir servise erişim sorunsuzken, canlı ortamda aynı çağrı engellenebilir. Bu nedenle sistem testinde:

- Ortam farklılıklarının sistematik biçimde **dokümante edilmesi**,
- Test raporlarında, belirli sonuçların "yalnızca test ortamı koşulları için geçerli olduğu"nun belirtilmesi,
- Kritik testler için, canlı ortama daha yakın ikinci bir ortam (örneğin pre-prod, staging) kullanılması

önemli risk azaltma adımlarıdır.

5.3. Test Verisi Türleri (Sentetik Veri, Maskeleme, Anonimleştirme)

Test verisi, sistem testinin **gerçekçiliğini** belirleyen temel unsurlardan biridir. Genel olarak üç yaklaşımdan söz edilebilir:

1. **Sentetik veri (yapay üretilmiş veri):**
    - Tamamen test ekibi veya otomatik araçlar tarafından üretilen, gerçek kullanıcı verisi içermeyen veri setleridir.
    - Avantajları: Gizlilik riski yoktur, istenen senaryolar için özel veri üretilebilir.
    - Dezavantajları: Gerçek hayattaki dağılımları, istisnaları ve veri kirlenmelerini her zaman iyi yansıtmayabilir.
2. **Maskeleme (masking):**
    - Canlı veriden alınan bir kopya üzerinde, kişisel veya hassas alanların belirli algoritmalarla değiştirilmesidir (örneğin TC kimlik numarası, telefon, e-posta gibi alanların maskelenmesi).
    - Amaç, verinin yapısını, uzunluğunu ve kısmen formatını korurken, kişiyi tanımlamayı imkânsız veya çok güç hâle getirmektir.
3. **Anonimleştirme (anonymization):**
    - Verinin, herhangi bir kişiyle **geri döndürülebilir biçimde ilişkilendirilemeyeceği** şekilde dönüştürülmesidir.
    - Anonimleştirme maskeden daha ileri bir adımdır: Maskede bazen orijinale dair izler kalabilir, anonimleştirmede bu bağ tamamen koparılmaya çalışılır.

Sistem testinde bu veri türleri çoğu zaman birlikte kullanılır. Örneğin:

- Temel yapıyı ve veri hacmini temsil etmek için anonimleştirilmiş canlı veri kopyası,
- Özellikle köşe durumları (edge case) için sentetik olarak eklenmiş veri satırları.

Böylece hem gerçek hayata yakınlık hem de gizlilik gereksinimleri arasında denge kurulur.

5.4. Kişisel Verilerin Korunması ve Gizlilik Odaklı Test Verisi Üretimi

Birçok sistem, doğrudan veya dolaylı olarak **kişisel veri** işler. KVKK, GDPR ve benzeri düzenlemeler, bu verilerin işlenmesi, saklanması ve paylaşılması üzerinde ciddi kısıtlar getirir. Test ortamı, bu açıdan çoğu zaman **en zayıf halka** hâline gelebilir; çünkü "nasıl olsa test ortamı" denilerek güvenlik politikaları gevşetilmeye meyillidir. Gizlilik odaklı test verisi üretiminde dikkat edilmesi gereken başlıca noktalar:

- Canlı ortamdan doğrudan veri kopyalamaktan kaçınmak; zorunlu ise, öncelikle **anonimleştirme veya maskeleme** süreçlerini işletmek.
- Test ortamı veritabanlarında, gerçek kişiyi tanımlamaya elverişli alanların (isim, TC kimlik no, telefon, adres, e-posta vb.) takma veri (pseudonym) ile değiştirilmesi.
- Test ortamına erişimi, sadece ilgili ekiple sınırlamak; log ve yedeklerde de hassas veri bulunmadığından emin olmak.
- Hassas alanlar üzerinde, sistem test senaryolarının gerektirmediği durumlarda gereksiz veri taşımamaya özen göstermek.

Örneğin, sistem testi için kullanıcıya ait **yaş aralığı** yeterliyse (18–25, 26–35 gibi), tam doğum tarihini canlı veriden taşımak yerine, bu alanı daha genel kategorik değerlere dönüştürmek hem gizlilik hem de analiz sadeliği açısından yararlıdır. Kısacası, test verisi üretimi

artık sadece teknik bir faaliyet değil; aynı zamanda **hukukî ve etik boyutları** olan bir süreçtir. Sistem testi planlanırken bu boyut mutlaka hesaba katılmalıdır.

5.5. Veri Hacmi ve Büyüklüklerinin Performans Üzerindeki Etkileri

Küçük veri setleri üzerinde "mükemmel" çalışan bir sistemin, gerçek veri hacimleriyle karşılaştığında performans sorunları yaşaması son derece yaygın bir durumdur. Bu nedenle sistem testinde:

- Veri hacminin canlı ortama **yaklaşacak şekilde** ölçeklendirilmesi,
- En azından kritik tablolarda satır sayısı, indeks yapısı ve veri dağılımının gerçekçi biçimde simüle edilmesi

büyük önem taşır. Örneğin, canlıda milyonlarca satır içeren bir ParkingSession tablosu varken, test ortamında yalnızca birkaç yüz satırla test yapmak:

- Karmaşık rapor sorgularının,
- Filtreleme ve sıralama işlemlerinin,
- Büyük join'lerin

gerçek davranışını yansıtmayacaktır. Bu nedenle performans açısından kritik olduğu bilinen alanlarda:

- Test veri setleri, canlı sistemden **anonimleştirilmiş büyük bir örnek (sample)** alınarak oluşturulabilir.
- Ek olarak, "en kötü durum" senaryolarını test etmek için veri hacmi özellikle şişirilerek, yoğunluk testleri yapılabilir.

Bu tür çalışmalar, sistem testini sadece "fonksiyonel doğrulama" olmaktan çıkarıp, aynı zamanda **ölçeklenebilirlik ve kapasite farkındalığı** sağlayan bir aşama hâline getirir.

5.6. Ortam ve Veri Tutarlılığının Sürdürülmesi

Test ortamı bir kez kurulduktan sonra "tamamdır, bir daha dokunmayız" denildiğinde, çok geçmeden şu tür sorunlar ortaya çıkar:

- Farklı ekiplerin ortamı "deney yapmak" için kullanması,
- Verinin zamanla kirlenmesi (çelişkili kayıtlar, eksik referanslar, bozuk testler),
- Bir sürümün ortasında veritabanı şemasının değiştirilmesi,
- Log ve cache'lerin kontrolsüz büyümesi.

Bu nedenle ortam ve veri tutarlılığını sürdürmek için:

- **Versiyonlanmış ortam konfigürasyonları** (infrastructure as code – örn. Terraform, Ansible) kullanmak,
- Veritabanı şema değişikliklerini migration araçlarıyla (Flyway, Liquibase vb.) yönetmek,
- Periyodik olarak test veritabanını "bilinen temiz bir snapshot"tan yeniden kurmak,
- Her test döngüsü öncesi temel smoke testler ile ortamın sağlığını kontrol etmek

iyi birer uygulamadır. Ek olarak, belirli test senaryoları veri üzerinde kalıcı değişiklikler yapıyorsa (örneğin park oturumu açıp kapatmak, ödeme yapmak), bu testlerin:

- Ya izolasyon sağlayacak şekilde kendi verisini üretip temizlemesi,
- Ya da her test döngüsünden sonra veritabanının ilk hâline geri alınması

gereklidir. Aksi hâlde, belirli bir süre sonra "bu test neden geçmiyor?" sorusunun cevabı çoğu zaman "ortam ve veri kirlendi de ondan" olur.

Özetle: sistem testinde ortam ve veri yönetimi, arka planda sessizce duran bir detay değil; test sonuçlarının **güvenilirliğini, tekrarlanabilirliğini ve hukuken savunulabilirliğini** belirleyen temel faktördür. Senaryolar ne kadar iyi olursa olsun, yanlış ortam ve yanlış veri üzerinde koşulduğunda, elde edilen sonuçlar en iyi ihtimalle "kısmen doğru" olacaktır.

6. Sistem Testinde Otomasyon

Sistem testi, elle yapılabildiği gibi otomasyonla da desteklenebilir. Özellikle sık tekrarlanan, regresyon niteliğindeki senaryolar için otomasyon, hem hız hem de tutarlılık açısından vazgeçilmez bir araç hâline gelmiştir. Bununla birlikte, "her şeyi otomatik test ederiz" yaklaşımı hem teknik hem de ekonomik olarak gerçekçi değildir. Bu bölümde, sistem testinde otomasyonun hedefleri, sınırları ve pratikte nasıl konumlandırılması gerektiği ele alınmaktadır.

6.1. Otomasyonun Hedefleri ve Sınırları

Sistem testinde otomasyonun temel hedefleri şöyle özetlenebilir:

- Tekrarlanabilirlik: Aynı testin, aynı adımlarla, istenildiği kadar tekrar çalıştırılabilmesi.
- Hız: Özellikle regresyon testlerinde, manuel testlere göre çok daha kısa sürede geniş kapsamlı testlerin yürütülebilmesi.
- Tutarlılık: İnsan hatasını azaltarak, her çalıştırmada aynı kontrollerin yapılmasının sağlanması.
- Geri bildirim döngüsünün kısaltılması: Geliştirilen özelliklerin kalite durumunun geliştirme ekibine hızlıca raporlanması.

Buna karşılık, otomasyonun doğal sınırları da vardır:

- Kullanılabilirlik, görsel kalite, metinlerin anlaşılırlığı gibi konular tamamen otomasyonla ölçülemez.
- Sık değişen kullanıcı arayüzlerinin otomasyonu, bakım maliyeti açısından verimsiz olabilir.
- Keşifsel testler ve "önceden tahmin edilmemiş" hata senaryoları, büyük ölçüde insanın sezgisine dayanır; otomasyon burada destekleyici ama sınırlı rol oynar.

Bu nedenle sistem testinde otomasyona şu gözle bakmak daha doğrudur: Otomasyon, test mühendisinin yerini alan değil, onun tekrarlı ve mekanik işlerini devralan yardımcı bir sistemdir.

Temel prensip, otomasyonu doğru yere ve doğru yoğunlukta uygulamaktır; aksi durumda test otomasyonu, testin kendisinden daha pahalı bir projeye dönüşebilir.

6.2. API Seviyesinde Sistem Testi Otomasyonu

Modern mimarilerde sistem davranışının önemli bir kısmı API katmanı üzerinden gözlemlenebilir. Bu nedenle, UI'dan bağımsız olarak API seviyesinde sistem testi otomasyonu yapmak hem daha kararlı hem de bakımı kolay bir çözüm sunabilir. API test otomasyonu ile:

- İş kuralları,
- Yetkilendirme mekanizmaları,
- Veri doğrulama,
- Farklı parametre kombinasyonları

UI'a bağlı kalmadan sınanabilir. Örneğin, otopark sisteminde park oturumu başlatan bir endpoint için basit bir API testi (JUnit + REST Assured):

```
@Test
void shouldStartParkingSessionForRegisteredPlate() {
    given()
        .contentType("application/json")
        .body("""
              {
                "plate": "ABC-123",
                "locationId": 42
              }
              """)
    .when()
        .post("/api/parking/sessions")
    .then()
        .statusCode(201)
        .body("status", equalTo("ACTIVE"))
        .body("plate", equalTo("ABC-123"));
}
```

Bu test, sistem seviyesinde şunları doğrular:

- Endpoint'in HTTP sözleşmesini (request/response yapısı),
- Başarılı durumda oluşturulan oturumun durumunu (ACTIVE),
- Dönen verinin iş kurallarına uygunluğunu.

Avantajları:

- UI değişikliklerinden görece bağımsızdır,
- Çalıştırma süresi kısadır,
- CI/CD içinde kolayca zincire eklenebilir.

Bu nedenle sistem test stratejisinde, kritik işlevlerin önemli bir bölümünü API seviyesinde otomatik test etmek, UI otomasyonuna göre daha "sağlam temelli" bir yatırım olarak görülür.

6.3. UI Otomasyon Araçları ve Çerçeveleri (Framework'ler)

Kullanıcı arayüzü (UI), sistem testinin dışarıdan görülen yüzüdür. İş süreçlerini gerçekten kullanıcının gördüğü şekliyle doğrulamak için UI otomasyonu da önemli bir yer tutar. Yaygın yaklaşım:

- Web uygulamaları için Selenium WebDriver, Cypress, Playwright vb.
- Mobil uygulamalar için Appium, Espresso, XCUITest vb.

UI otomasyonunda temel fikir, kullanıcının yaptığı eylemleri script ile simüle etmek ve ekran çıktıları üzerinden doğrulama yapmaktır. Örneğin Selenium ile basit bir oturum açma testi (Java):

```
@Test
void shouldLoginFromLoginPage() {
    WebDriver driver = new ChromeDriver();
    try {
        driver.get("https://testenv.example.com/login");

        WebElement emailInput = driver.findElement(By.id("email"));
        WebElement passwordInput = driver.findElement(By.id("password"));
        WebElement loginButton = driver.findElement(By.id("btnLogin"));

        emailInput.sendKeys("user@example.com");
        passwordInput.sendKeys("P@ssw0rd");
        loginButton.click();

        WebElement dashboardHeader = driver.findElement(By.id("dashboard-title"));
        assertEquals("Ana Sayfa", dashboardHeader.getText());
    } finally {
        driver.quit();
    }
}
```

Bu test:

- Gerçek tarayıcı açar,
- Giriş formunu doldurur,
- Dashboard'a yönlendirildiğini kontrol eder.

Avantajları:

- Kullanıcı perspektifine çok yakındır,
- Uçtan uca iş akışlarını görsel olarak da doğrular.

Dezavantajları:

- UI değişikliklerinden çok etkilenir,
- Çalışma süresi API testlerine göre uzundur,
- Flaky (arada sırada nedeni belirsiz şekilde başarısız olan) testler üretme riski yüksektir.

Bu nedenle UI otomasyonu, seçici ve kontrollü kullanılmalıdır. Tipik iyi pratik:

- En kritik uçtan uca senaryolar (login, ödeme, ana iş akışı) UI otomasyonuyla,
- Detaylı iş kuralları ve varyasyonlar ise API seviyesinde test edilir.

6.4. Devamlı Entegrasyon / Devamlı Teslim (CI/CD) Süreçlerine Entegrasyon

Test otomasyonu, tek başına bir hedef değil; geliştirme hattının (pipeline) doğal bir parçası hâline getirildiğinde gerçek değerini gösterir. Devamlı entegrasyon (Continuous Integration – CI) ve devamlı teslim (Continuous Delivery/Deployment – CD) süreçleriyle entegrasyonun temel amaçları:

- Her kod değişikliğinde otomatik testlerin tetiklenmesi,
- Hataların mümkün olan en erken aşamada fark edilmesi,
- Sürüm adaylarının kalitesine dair sürekli görünürlük sağlanması.

Basit bir pipeline kurgusu:

- Adım 1: Build ve birim testler
- Adım 2: Entegrasyon testleri
- Adım 3: API seviyesinde sistem/regresyon testleri
- Adım 4: Seçilmiş UI end-to-end testleri
- Adım 5: Raporlama ve kalite eşiği kontrolleri

Örneğin, bir CI aracı (GitLab CI, GitHub Actions, Jenkins vb.) üzerinde:

```
stages:
  - build
  - test
  - e2e

api-tests:
  stage: test
  script:
    - mvn test -Dgroups=api

ui-e2e-tests:
  stage: e2e
  script:
    - mvn test -Dgroups=ui-e2e
  when: on_success
```

Bu yapı, kod her ana dala (main/master) birleştiğinde temel API ve UI testlerinin otomatik olarak çalışmasını sağlar. Böylece sistem testindeki otomasyon, tek seferlik bir “kampanya” olmaktan çıkıp, sürekli çalışan bir kalite duvarı hâline gelir.

6.5. Test Script’lerinin Bakım Maliyeti ve Sürdürülebilirlik

Otomasyonun göz ardı edilen ama en kritik yönlerinden biri, bakım maliyetidir. Zaman içinde:

- Ekranlar değişir,
- Endpoint’ler güncellenir,
- İş kuralları revize edilir,
- Test verisi yapısı değişir.

Eğer test script’leri bu değişikliklere uyum sağlayacak şekilde tasarlanmamışsa, kısa sürede:

- Sürekli kırılan testler,
- Kimsenin güvenmediği raporlar,
- “Otomasyonu kapatsak da kurtulsak mı?” hissiyatı

ortaya çıkar.

Bakımı azaltmak için yaygın yaklaşımlar:

- Page Object pattern (UI testlerinde):
- Ekran elemanlarını ve etkileşimleri tek bir sınıfta toplayıp, testlerin bu sınıfı kullanması. Böylece UI değiştiğinde onlarca test yerine tek bir page objesi güncellenir.
- API client katmanı (API testlerinde):

- Endpoint çağrılarını ortak bir client sınıfı üzerinden yapmak; URL veya header değiştiğinde sadece bu client'ın güncellenmesi.
- Test verisini koddan ayırmak:
- Test verilerini JSON, YAML, CSV veya özel fixture dosyalarında tutmak; böylece hem değişiklik hem de varyasyon üretimi daha kolay olur.
- Tekrarlanan adımları soyutlamak:
- "Giriş yap", "varsayılan kullanıcıyı oluştur" gibi sık kullanılan adımlar için yardımcı fonksiyonlar veya helper sınıfları kullanmak.

6.6. Otomatik ve Manuel Test Dengesinin Kurulması

Sistem testinde asıl sanat, "her şeyi otomatikleştirmek" değil; **neyi manuel, neyi otomatik yapacağımıza bilinçli karar verebilmektir.**

Genel ilkeleri şöyle özetleyebiliriz:

- Otomasyona uygun olanlar:
    - Sık tekrarlanan regresyon senaryoları,
    - Kritik iş akışlarının "mutlu yol" ve temel hata senaryoları,
    - API seviyesinde karmaşık iş kurallarının varyasyonları,
    - Performans, yük ve bazı güvenlik kontrolleri (araç destekliyse).
- Manuel testin vazgeçilmez olduğu alanlar:
    - İlk kez geliştirilen veya köklü şekilde değiştirilen özelliklerin keşifsel testleri,
    - Kullanılabilirlik ve kullanıcı deneyimi (UX) değerlendirmeleri,
    - Karmaşık entegrasyon senaryolarının ilk keşfi,
    - Analitik, raporlama gibi "çıktıyı yorumlama" gerektiren alanlar.

Bu dengeyi kurarken sık kullanılan metafor, **test piramidi**dir:

- Taban: Çok sayıda birim testi,
- Orta: Daha az sayıda entegrasyon ve API testi,
- Tepe: Görece az sayıda ama kritik UI end-to-end testi.

Sistem testi perspektifinden bakınca, piramidin özellikle orta ve üst kısmı (API + UI E2E) ile ilgileniriz. Burada önemli olan:

- Tepeyi (UI E2E) gereğinden fazla şişirmemek,
- Orta katmanda (API) mümkün olduğunca çok iş kuralını yakalamak,
- Manuel test kapasitesini de keşifsel ve kullanıcı odaklı alanlara kaydırmaktır.