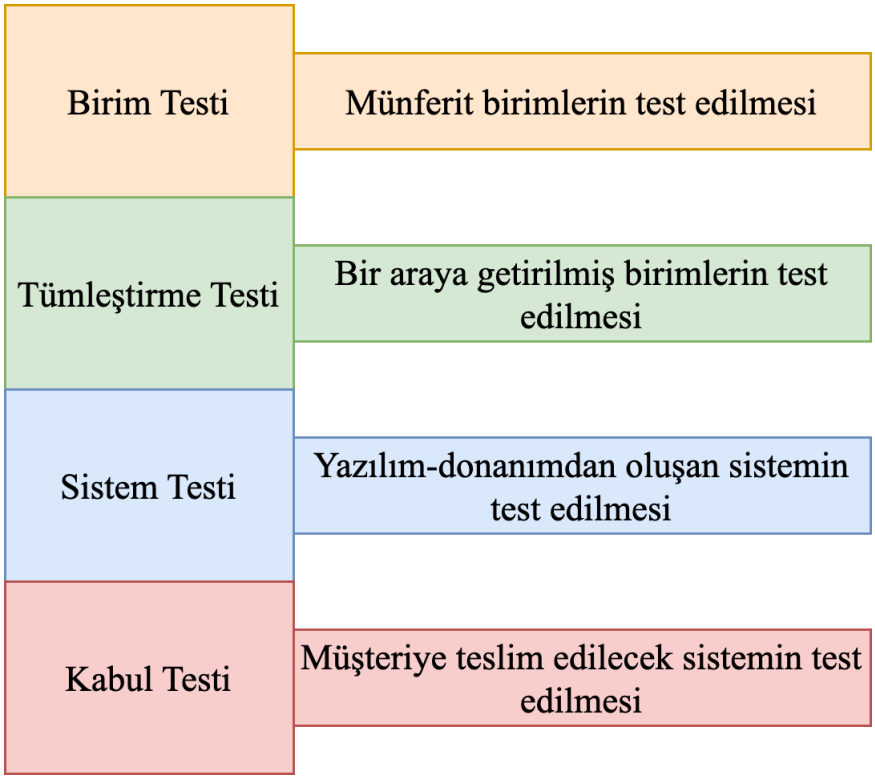
1. YAZILIM TEST SEVİYELERİ

Yazılım projelerinde amaçları birbirinden farklı dört seviyede test gerçekleştirilmektedir. Dört farklı test seviyesi bulunmaktadır. Bu seviyeler;

- Birim Testleri
- Tümleştirme Testleri
- Sistem Testleri
- Kabul Testleri seviyeleridir.

Şekil 1'de bu test seviyelerinin test ettikleri birimler verilmiştir.



Şekil 1. Yazılım test seviyeleri

1.1. Birim Testleri

Metot, fonksiyon ya da prosedür gibi bir kod parçasının, diğer kod parçalarından izole edilerek, kendisinden beklenen işlevselliği doğru olarak yerine getirdiğini ve hata içermediğini göstermek için **yazılım geliştiriciler** tarafından gerçekleştirilen testlerdir. Birim testleri, bazen proje takviminden, bazen yeterli tecrübe olmayışından ve bazen de az önem verilesinden dolayı ya yapılmamakta ya da acele ile yapılmaktadır. Birim testleri düzenli bir şekilde yapıldığında daha başarılı projeler ortaya çıkmaktadır.

Birim testleri genellikle yazılım test araçlarıyla yapılmaktadır. Eğer bir yazılım test aracı kullanılmayacaksa birim test kodları yazılmakta ve bu da yazılım geliştiriciler tarafından yapılmaktadır. Başarıyla sonuçlanan birim testler, tümleştirme ve sistem testlerinden önce

geliştirilen yazılımın genel olarak istenilen gereksinimleri karşıladığının göstergesidir. Ancak, tam olarak çalıştığını göstermez. Çünkü birim testleri ile arayüz testleri gerçekleştirilmez. Arayüz testleri tümleştirme ve sistem testleri aşamasında yapılmaktadır.

Birim testlerinin amacı, geliştirilen ve derlenebilen en küçük kod parçalarının (fonksiyon, metot, prosedür, sınıf, vb.) kendine ait görevleri doğru bir şekilde yerine getirdiğinin doğrulanmasıdır. Bu nedenle, test edilen kod parçası, diğer kod parçalarından izole edilmektedir. Test edilen kod parçasının yerine getireceği göreve göre test durumları oluşturulur. Bu durumlar kullanılarak test edilecek birim çalıştırılır ve elde edilen sonuçlar beklenen sonuçlar ile karşılaştırılarak testlerin başarılı olup olmadığına karar verilir. Eğer test başarısızlıkla sonuçlanmışsa birim kod tekrar gözden geçirilir, gerekli düzenlemeler yapılır ve tekrar test edilir. Bu işlem test başarılı oluncaya kadar gerçekleştirilir. Birim testleri, kod geliştirme safhasında **yazılım geliştiricileri** tarafından yapılmaktadır. Aşağıda verilen bir kod parçası üzerinde birim testin gerçekleştirilmesi gösterilmiştir.

```
using System;

public bool pozitifMi (int k)
{
    if (k < 0)
        return true;
    else
        return false;
}
```

Yukarıda verilen kod parçasının test edilmesi için aşağıda verilen test durumları oluşturulabilir;

Test Durumu 1

```
using System;

public void testDurumu_1()
{
    bool td1 = false;
    int m = 3;

    td1 = pozitifMi(m);

    if (td1==true)
        Console.WriteLine("Test Durumu 1: Geçti");
    else
        Console.WriteLine("Test Durumu 1: Kaldı");
}
```

Test Durumu 2

```
using System;

public void testDurumu_2()
{
    bool td2 = true;
    int m = -4;

    td2 = pozitifMi(m);

    if (td2==false)
        Console.WriteLine("Test Durumu 1: Geçti");
    else
        Console.WriteLine("Test Durumu 1: Kaldı");
}
```

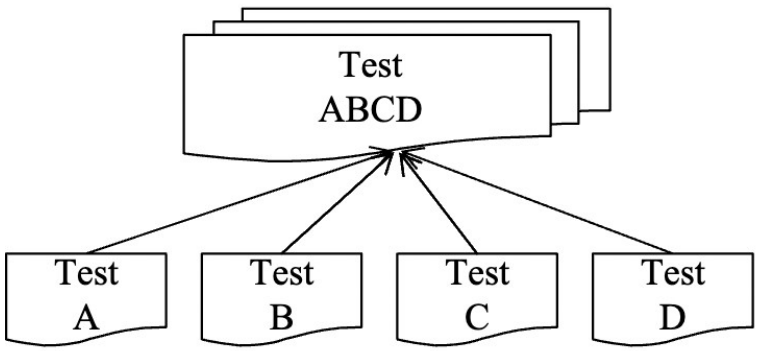
Birim test gerçekleştirilirken, birim içerisindeki tüm yollar ve karar noktaları test edilmelidir. Yukarıdaki örnek incelendiğinde, yazılan iki test durumu sırasıyla çalıştırıldığında, ilk test durumunun çalıştırılmasının ardından, %50'lik bir kapsamaya ulaşılmaktadır. Kısacası bu kodun sadece yarısı test edilmiş olmaktadır. Eğer diğer test durumu da çalıştırılırsa, kodun geri kalan kısmı da test edildiği için %100'lük bir kapsamaya ulaşılmaktadır. Kısacası bu metottaki tüm durumlar test edilmiş olur.

Birim testlerde bir diğer önemli husus ise sınır değerlerin kontrol edilmesidir. Bir metot içerisinde belirli bir aralıkta işlem yapılıyorsa bu aralığın üst ve alt sınır değerlerinin kontrol edilmesi gerekmektedir. Verilen birim test örneği incelendiğinde k = 0 durumunda ne gibi bir sonuç ortaya çıkar? Bu durumun kod içerisinde kontrol altına alınması gerekmektedir. Bu durum için üçüncü bir test durumu yazılmalı ve kod içerisinde ona göre değişiklik yapılmalıdır.

1.2. Tümleştirme Testleri

Bir yazılım projesinde birim testlerin başarılı bir şekilde sonuçlanmasının ardından tümleştirme testleri başlamaktadır. Tümleştirme testlerinin amacı bir araya gelerek entegre edilmiş olan bir yazılımın içerisindeki bileşenlerin birbirleriyle uyum içerisinde doğru şekilde çalıştığının ve bileşenlerin kendilerine ait gereksinimleri yerine getirdiğinin doğrulanmasıdır. Her bir modül kendi içerisinde başarılı birim testler oluştursa dahi, bir araya geldiklerinde hatalar üretebilir. Bunların ortaya çıkartılması için tümleştirme testleri kullanılmaktadır. **Test uzmanları** tarafından gerçekleştirilmektedir. Genellikle yazılım gereksinimleri ve arayüz gereksinimleri kullanılmaktadır. Modüller arası işlevsel, performans ve haberleşme gibi gereksinimlerin doğrulanması gerçekleştirilir. Tümleştirme testleri gerçekleştirilirken big bang, aşağıdan-yukarıya ve yukarıdan-aşağıya olmak üzere farklı stratejiler kullanılmaktadır.

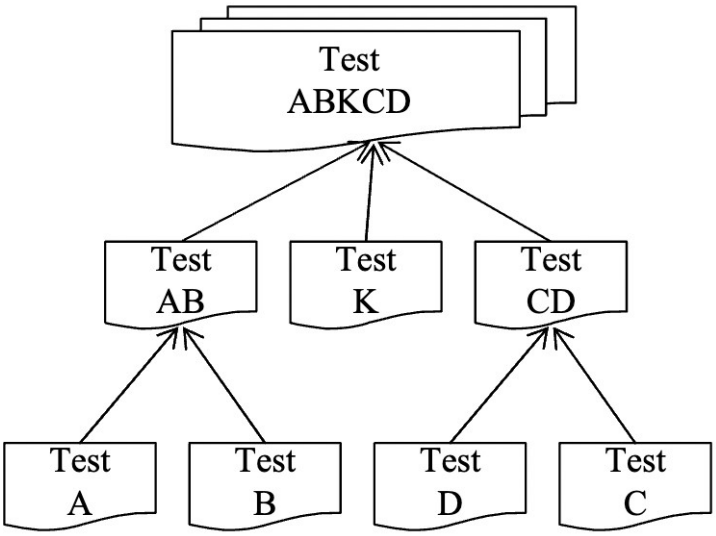
Big bang tümleştirme stratejisinde, geliştirilen tüm üst ve alt seviye modüller birbirleriyle entegre edildikten sonra testler başlamaktadır. Bu stratejide sistem bir bütün olarak ele alındığı için testlerin gerçekleştirilmesi hızlı olmakta ve zamandan tasarruf elde edilmektedir. Ancak, bir hata ortaya çıktığında hatanın hangi modülden ya da hangi seviyeden geldiğinin belirlenmesi zordur. Şekil 2'de bu stratejiye ait örnek verilmiştir.



Şekil 2. Big bang tümleştirme test stratejisi

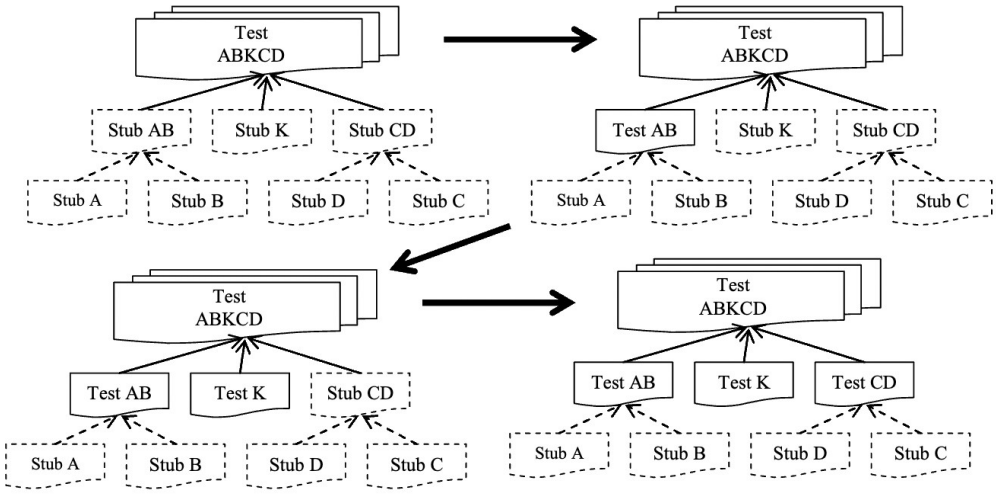
Aşağıdan-yukarıya tümleştirme test stratejisinde ise öncelikle birim testler yapılmaktadır. Birim testleri başarılı olan alt seviye modüller birbirleriyle entegre edilmektedir. Ardından bu modüller üzerinden tümleştirme testleri yapılmaktadır. Bu testlerin başarılı olmasının ardından üst seviye modüller sisteme entegre edilir ve gerekli tümleştirme testleri

gerçekleştirilir. Bu işlem tüm entegrasyon işlemi tamamlanana ve testler başarılı olana kadar devam etmektedir. Bu stratejide aranan en önemli şart tüm seviye modüllerin birim testlerinin başarılı olmasıdır. Şekil 3'te bu yaklaşımın görseli verilmiştir.



Şekil 3. Aşağıdan-yukarıya tümleştirme test stratejisi

Yukarıdan-aşağıya tümleştirme test stratejisinde öncelikle sistem için en üst modülün (genellikle kullanıcı arayüzü) testi gerçekleştirilir. Bu modülün ihtiyaç duyduğu alt modüller kullanılır. Bu modül başarılı bir şekilde test edildikten sonra bir alt seviye modüller sisteme entegre edilir. Ardından alt seviyedeki modüller için tümleştirme testleri yürütülür. Bu durum en alt seviyede modüller test edilinceye ve testler başarılı oluncaya kadar devam eder. Şekil 4'te bu yaklaşıma ait görsel verilmiştir.



Şekil 4. Yukarıdan-aşağıya tümleştirme test stratejisi

Tümleştirme test durumları, diğer test durumlarından farklı olarak modüller arası veri arayüzlerine ve bilgi akışına odaklanmaktadır.

Bir elektronik posta uygulaması üzerinden tümleştirme testi gerçekleştirilecektir. Uygulamada Giriş Sayfası, Posta Kutusu ve E-Postaları Sil'den oluşan üç modül bulunmaktadır. Mantıksal olarak kullanıcı adı ve şifre girildiğinde kişi kendi e-posta kutusuna erişecektir. Tümleştirme testinin odaklandığı durum kullanıcı adı ve şifrenin doğru girilmesi durumunda kullanıcıya kendi Posta Kutusu'nu açtığının veya kullanıcı adı ve şifrenin yanlış girilmesi durumunda Posta Kutusu sayfasının açılmadığının gösterilmesidir. Tablo 1'de tümleştirme test durumu için iki kısa örnek verilmiştir.

Tablo 1. Tümleştirme test durum örneği

TD ID	Test Durumu Amacı	Test Durumu Adımları	Beklenen Sonuç
TD-1	Giriş Sayfası ve Posta Kutusu sayfaları arasındaki arayüz bağlantısının kontrol edilmesi	Kullanıcı adı ve şifre girilir. Giriş butonuna basılır. Posta Kutusu'na girildiği doğrulanır.	Posta Kutusu'nun açılması
TD-2	Posta Kutusu ve E-postaları Sil modülü arasındaki arayüz bağlantısının kontrol edilmesi	Posta Kutusu'ndan bir e-posta seçilir. Silme butonuna basılır. Silme işlemi onaylanır. Çöp Kutusu klasörüne tıklanır. Silinen e-postanın bu klasörde olduğu doğrulanır.	Seçilen e-postanın silinmiş olması ve Çöp Kutusu klasöründe olması

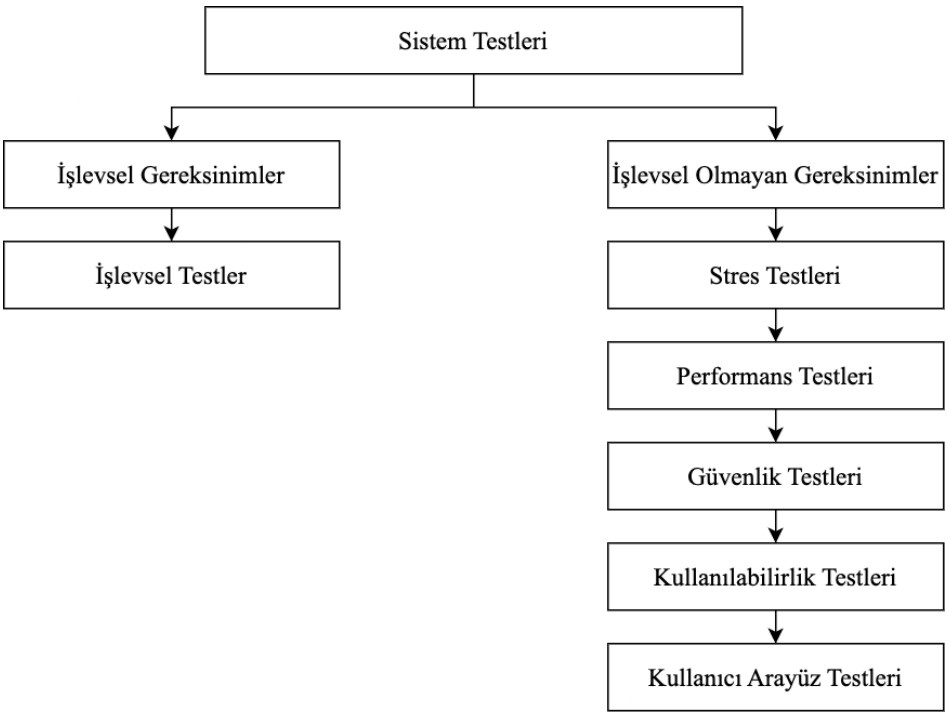
1.3. Sistem Testleri

Sistem testi, bütünleşme süreci tamamlanmış olan bir yazılımın testidir. **Proje test ekibi** tarafından gerçekleştirilir. Yazılımın işlevsel ve işlevsel olmayan gereksinimlerinin doğrulandığı testlerdir. İşlevsel gereksinimler yazılım çalıştırıldığı ortamda nasıl davranacağını ve nasıl davranmayacağını tanımlayan ifadelerdir. İşlevsel olmayan gereksinimler ise yazılımın doğruluk, performans, güvenilirlik, uyması gereken standartlar gibi kalite özelliklerini tanımlayan ifadelerdir.

Birim testlerde ve tümleştirme testlerinde geliştirilen yazılımın tasarıma uygunluğu doğrulanırken; sistem testlerinde, geliştirilen yazılımın gerçek çalışma ortamında müşterinin

sistemden beklediklerinin doğrulamayı ve bu beklentileri yerine getirirken sistemin en az seviyede hata içerdiğinin göstermeyi amaçlar. Dolayısıyla sistem testlerinde hem test durumları hem de test senaryoları uygulanır. Sistem test durumları ve sistem test senaryoları sistem gereksinimleri, kullanıcı hikâyeleri, kullanım senaryoları, risk analiz raporları gibi test esasları temel alınarak geliştirilir.

Sistem testleri yazılım ve donanım entegrasyonundan sonra "sistem test planına" göre gerçekleştirilir. Sistem testlerinde ilk adım işlevsel testlerin doğrulanmasının yapılmasıdır. Bunun başlıca nedeni işlevsel olarak çalışmayan bir yazılımın üzerinde diğer sistem testlerinin yapılmasının anlamsız olmasıdır. İşlevsellik yönünden doğrulanan sistem üzerinde, sonraki adım olarak işlevsel olmayan testler uygulanır. Sistem testleri ve yapılma sırası temsili olarak Şekil 5'te gösterilmiştir.



Şekil 5. Sistem test türleri

Stres Testleri: Sisteme girdi oranı, sistem tasarım oranı aştığı zaman sistemin davranışını gözlemlemek üzere gerçekleştirilen testlerdir.

Performans Testleri: Sistemin çıktılarının belirlenen ve kabul edilen zaman dilimi içerisinde üretilebildiğinin değerlendirilmesinin yapılabilmesi için gerçekleştirilen testlerdir.

Güvenlik Testi: Sistemin izinsiz kullanımlar halinde davranışlarının değerlendirilmesi için gerçekleştirilen testlerdir.

Kullanılabilirlik Testleri: Kullanıcı ile sistem arasındaki etkileşimi ve ergonomisini değerlendirmek üzere gerçekleştirilen sistemlerdir.

Kullanıcı Arayüz Testleri: Yazılım grafikleri ile kullanıcı arasında nasıl bir etkileşim

olacağını, kullanıcının klavye, ekran veya fare ile sisteme vereceği girdilerin sistem tarafından nasıl işleneceğini değerlendirmek için gerçekleştirilen testlerdir.

İşlevsel ve işlevsel olmayan testleri tamamlanan sistem artık doğrulanmış olacaktır ve kullanıcı kabul testlerine hazır olmuş olacaktır. Sistem testleri gerçekleştirilirken ortaya çıkan hatalar proje hata yönetim sürecine göre raporlanır ve gerekli düzeltme işlemleri gerçekleştirilir. Yapılan düzeltmelerden sonra sistemin geri kalanının bu düzeltmelerden etkilenip etkilenmediğinin değerlendirilmesi için sistem üzerinde yineleme testleri yapılmalıdır. Sistem testleri ile:

- Riskler minimum düzeye indirilir.
- Kalite metrikleri yönünden de test edildiğinden dolayı sistemin kalitesi adına güven oluşturulur.
- Kabul testleri öncesinde büyük ve önemli hatalar bulunup düzeltilir.

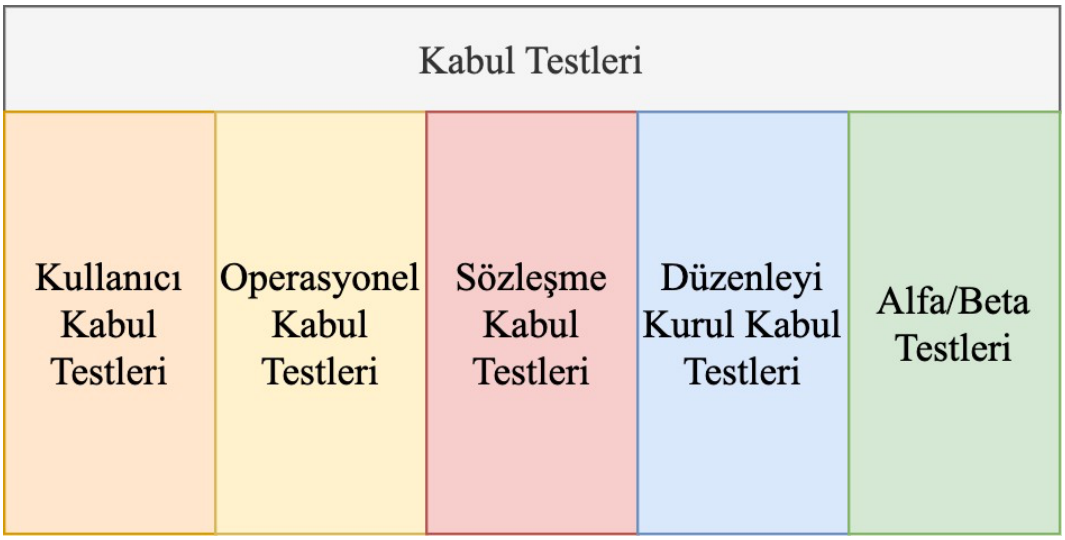
1.4. Kabul Testleri

Kabul testlerinde de, sistem testlerinde olduğu gibi geliştirilen ürünün tamamının kapasitesi ve yetenekleri görülmeye çalışılır. Kabul testlerinde müşteri ihtiyaçlarının ve gereksinimlerinin sistem tarafından yerine getirilip getirilmediğine odaklanılır. Bunun için tüm kullanıcı ihtiyaçlarını ve gereksinimini doğrulayacak olan test durumları ve test senaryoları oluşturulur. Oluşturulan bu senaryolar için gerçekleştirim öncesi müşterinin onayı alınır. Geliştirilen sistem üzerinde, tanımlanan bu test durumları ve senaryoları, müşterinin de katılımı ile “kabul test planına” uygun olarak gerçekleştirilir. Sistemin bu testlerden geçmesi ile kabul testleri gerçekleştirilmiş olur. Müşteri ile gerçekleştirilen testler olduğundan dolayı aynı zamanda “kullanıcı kabul testi” olarak da adlandırılmaktadır. Kabul testlerinin odak noktası bu yüzden müşteri ihtiyaçları olmalıdır. Sistemin tüm kabul testlerinden geçmiş olması ürünün müşteri tarafından kabul edileceği anlamına gelmez. Kabul testi ile hedeflenen:

- Sistemin bütünüyle sistemin kalitesine yönelik güven oluşturmak,
- Sistemin tamamlandığını ve beklendiği gibi çalıştığını göstermek,
- Sistemin işlevsel ve işlevsel olmayan davranışlarının belirtildiği gibi gerçekleştiğini doğrulamak

Kabul testlerinde hatalar bulunabilir. Ancak bu hatalar sistemin kabulü için büyük bir risk oluştursa da bu testlerin ana amacı müşteri ihtiyaçlarının karşılanmasıdır. Bulunan bu hatalar proje hata yaşam döngüsüne göre raporlanmalı ve gerekli düzeltmeler sistem üzerinde gerçekleştirilmelidir. Kullanıcı kabul testleri sırasında geliştirilen sistemin uyması gereken

standartlar ve/veya düzenleyici kurumların ortaya koydukları kuralları da karşıladığı doğrulanmalıdır. Kullanıcı kabul testleri Şekil 6'da gösterilmiştir.



Şekil 6. Kabul test türleri

Kullanıcı Kabul Testi: Sistemin amaçlanan kullanıcılar tarafından, sistemin çalıştırılacağı gerçek ortamda veya gerçek ortama benzer olarak oluşturulmuş bir ortamda, sistemin kullanıma uygunluğunu doğrulamaya yönelik gerçekleştirilen testlerdir. Amaç sistemin kullanılabilmesine yönelik güven oluşturmaktır.

Operasyonel Kabul Testi: Geliştirilen sistemin, sistem yöneticileri tarafından kabulünün yapılmasıdır. Bu testler sistem yöneticileri tarafından genellikle benzetilmiş bir çalışma ortamında gerçekleştirilir. Bu testler operasyonel konulara odaklanır ve aşağıdaki maddeleri içerebilir:

- Yükleme, Kaldırma ve Sürüm Yükseltme
- Yedekleme ve Geri Yükleme Testi
- Çöküşten/Hatadan Kurtulma/Kurtarma
- Kullanıcı Yönetimi
- Bakım Görevleri
- Veri Yükleme ve Verme/Göç Ettirme
- Güvenlik Testleri
- Sızma Testleri
- Performans Testleri

Sözleşme Kabul Testleri: Gerçekleştirilen sistemin, imzalanan sözleşmenin tüm kabul kriterlerinin yerine getirildiğini göstermek üzere gerçekleştirilen testlerdir. Sözleşmedeki kabul kriterleri işi talep edenin genellikle tanımladığı ve karşılıklı anlaşılan maddelerdir. Sözleşme kabul testi genellikle kullanıcılar veya bağımsız test uzmanları tarafından yapılır.

Düzenleyici Kurul Kabul Testleri: BDDK, EPDK gibi resmi düzenleyici kurulların sektör için koymuş oldukları düzenlemelerin geliştirilen sistem tarafından sağlandığını göstermek üzere yapılan testlerdir. Genellikle kullanıcılar veya bağımsız test uzmanları tarafından gerçekleştirilir. Bu testler sırasında düzenleyici kurul temsilcileri testleri gözlemleyebilir veya test sonuçlarını değerlendirip onaylayabilir.

Alfa/Beta Testi: Bu testler yazılım geliştirenler tarafından ürünü pazara sunmadan önce potansiyel veya mevcut kullanıcılar ya da müşterilerden geri bildirim almak amacıyla gerçekleştirilir. Alfa testi, yazılımı geliştiren kurumun sağladığı ortamda potansiyel ya da mevcut müşteriler ile gerçekleştirilen testtir. Beta testi ise mevcut ya da potansiyel müşteriler ya da kullanıcılar tarafından kendi ortamlarında gerçekleştirilen testlerdir. Beta testi, alfa testinden sonra yapılmaktadır. Ancak, alfa testi yapılmadan direkt beta testi de yapılabilir. Bu testlerin yapılmasındaki amaç, ürünün pazara hazır olup olmadığının değerlendirilerek kullanıcılar açısından bir güvenin oluşturulmasını sağlamaktır.

Sonuç olarak kabul testlerinin gerçekleştirilmesi ile aşağıdaki durumlar gerçekleştirilir:

- Ürünün müşterinin gereksinimlerini kapsadığını doğrulamak,
- Daha önce test eylemlerinde tespit edilemeyen problemlerin tespitini yapmak,
- Geliştirilen sistemin müşteri gereksinimlerini nasıl karşıladığı göstermek.

2. KOD KOKUSU

Kod kokusu, genellikle yazılım geliştirme süreçlerinde karşılaşılan ve yazılımın kalitesiz veya bakımsız olduğunu gösteren bir işarettir. Kod kokusu, yazılım geliştiricilerin veya başkalarının yazılımı anlamasını veya sürdürülebilir bir şekilde geliştirmesini zorlaştıran kod parçaları veya tasarım kararları içermektedir. Aynı zamanda kod kalitesinin düşük olduğu veya kodun işlevsel olmasına rağmen olması gerektiği gibi çalışmadığı durumları da ifade etmektedir.

Kod kokusu, bir yazılım projesinin uzun vadede sürdürülebilirliğini ve bakımını zorlaştırmaktadır. Bu nedenle yazılım geliştiricileri, kod kokularını belirlemek ve gidermek için çaba göstermek zorundadır. Yaygın kod kokusu örnekleri şu şekilde özetlenebilir:

- **Tek Fonksiyon:** Bir işlevin çok fazla iş yaptığı ya da çok uzun olduğu durumlardır. Her işlevin belirli bir işi yapması, kısa ve anlaşılır olması tercih edilir.
- **Teknoloji Bağımlılığı:** Yazılımın belirli bir teknolojiye veya ürüne fazla bağımlı olması durumudur. Bu, yazılımın taşınabilirliğini veya güncellenmesini zorlaştırabilir.
- **Yinelemeler:** Aynı kod bloklarının birçok yerde tekrarlanması olarak ifade edilmektedir. Bu, bakımı zorlaştırmakta ve hataların çoğalmasına neden olabilmektedir.
- **Büyük Sınıflar:** Bir sınıfın veya modülün çok büyük ve karmaşık olması durumudur. Sınıfın sorumluluklarının net bir şekilde belirlenmediği anlamına gelmektedir.
- **Sihirli Sayılar ve Metinler:** Kodun içinde sabit sayılar veya metinlerin doğrudan kullanılması, kodun anlaşılmasını zorlaştırmakta ve bakımını güçleştirmektedir. Bunun yerine bu değerlerin açıkça adlandırılması tercih edilmelidir.
- **Karmaşık İf Blokları:** Çok sayıda iç içe geçmiş koşullu ifadelerin olduğu kod bloklarıdır. Bu, kodun anlaşılmasını zorlaştırarak hata ayıklamayı karmaşık hale getirmektedir.
- **Kötü Adlandırma:** Değişkenlerin veya fonksiyonların anlaşılması güç adlar taşıması, kodun okunabilirliğini düşürmektedir.
- **Çoklu İşler:** Bir işlevin birden fazla işi yapmaya çalışmasıdır. Her işlevin tek bir sorumluluğu olmalıdır.
- **Eksik Dokümantasyon:** Kaynak kodun yetersiz veya eksik bir şekilde belgelenmesi, diğer geliştiricilerin veya bakım ekibinin kodu anlamasını ve kullanmasını zorlaştırmaktadır. Kodun düzgün bir şekilde belgelendirilmesi önem arz etmektedir.
- **Spagetti Kodu:** Karmaşık ve anlaşılmaz bir kod yapı olarak ifade edilmektedir. Bu, kodun akışını takip etmeyi zorlaştırmakta ve hataları tespit etmeyi güçleştirmektedir.
- **Kötü Tasarım Desenleri:** Yanlış veya kötü tasarlanmış tasarım desenlerinin kullanılması, kodun karmaşıklığını arttırabilmektedir.

Kod kokularını gidermek için genellikle yeniden kod düzenleme tekniği kullanılmaktadır. Bu teknik, kodun daha temiz, okunabilir, sürdürülebilir ve performanslı hale getirilmesine yardımcı olmaktadır. Yeniden kod düzenleme aşağıdaki adımları içermektedir:

- **Kodun Parçalara Bölünmesi:** Büyük ve karmaşık kod parçalarının daha küçük ve anlaşılır parçalara bölünmesidir.
- **İsimlendirme İyileştirmeleri:** Değişkenlerin, fonksiyonların ve sınıfların daha açıklayıcı isimlere sahip olmasıdır.
- **DRY Prensipleri:** "Don't Repeat Yourself" prensiplerine uygun olarak, tekrarlanan kodların ayrı bir işlevsel birimde toplanmasıdır.
- **Yineleme Kaldırma:** Aynı işi yapan kod bloklarının aynı yerde toplanması olarak ifade edilmektedir.
- **Tek Sorumluluk İlkesi Uygulamaları:** Her sınıf veya fonksiyonun sadece bir sorumluluğu olacak şekilde yeniden tasarlanmasıdır.
- **Kapsülleme ve Soyutlama:** Karmaşık işlemlerin daha basit ve soyutlanmış arayüzlerle temsil edilmesi olarak tanımlanmaktadır.

2.1. Çeşitli Kod Kokusu Örnekleri

2.1.1. Tek Fonksiyon Örneği

```
public class UrunYonetimi
{
    public void UrunEkle(string urunAdi, decimal urunFiyati, int
stokMiktari)
    {
        // Ürünün veritabanına eklenmesi
        // Stok güncellemesi
        // E-posta gönderme işlemi
        // Logging (günlüğe kaydetme)
    }
}
```

Verilen kod bloğunda **UrunYonetimi** sınıfının **UrunEkle** adlı metodu çok fazla işi aynı anda yapmaktadır. Ürün ekleme, stok güncelleme, e-posta gönderme ve günlüğe kaydetme aynı metod içinde bulunmaktadır. Bu, kodun bakımını zorlaştırmakta ve herhangi bir hata veya değişiklik durumunda sorunlar ortaya çıkabilmektedir.

Daha iyi bir uygulama yapısı için her işlevin yalnızca tek bir sorumluluğa sahip olmasını sağlamak önem arz etmektedir. Verilen örnekte, **UrunEkle** metodu farklı işlevlere bölünerek daha okunabilir ve sürdürülebilir hale getirilebilir.

```
public class UrunYonetimi
{
    public void UrunEkle(string urunAdi, decimal urunFiyati, int
stokMiktari)
    {
        // Ürünün veritabanına eklenmesi
    }

    public void StokGuncelle(string urunAdi, int yeniStokMiktari)
    {
        // Stok güncellemesi
    }

    public void EpostaGonder(string urunAdi)
    {
        // E-posta gönderme işlemi
    }

    public void Logla(string urunAdi)
    {
        // Logging (günlüğe kaydetme)
    }
}
```

2.1.2. Yinelemeler

Örnek olarak, bir uygulamada "Sisteme hoş geldiniz!" mesajının görüntülenmesinin gerektiğini düşünün. Ancak bu mesajı uygulamanın birçok farklı yerinde görüntülenmesi gerekmektedir. Aşağıdaki kod ile bu işlem yapılabilir.

```
Console.WriteLine("Sisteme hoş geldiniz!");
//...
//...
//...

// Başka bir yerde
Console.WriteLine("Sisteme hoş geldiniz!");
//...
//...
//...

// Ve başka bir yerde daha
Console.WriteLine("Sisteme hoş geldiniz!");
//...
//...
//...
```

Bu örnekteki gibi bir durum kodun bakımını zorlaştırmaktadır. "Sisteme hoş geldiniz!" mesajının daha sonra değiştirilmesi gerektiğinde, her bir tekrarın bulunup güncellenmesi gerekecek ve bu da hatalara ve gereksiz karmaşıklığı yol açacaktır.

Yinelemelerin giderilmesi ve kod kokusunun önlenmesi için aynı kod bloğunun birden çok kez yazılması yerine bu kodun bir yöntem içerisinde değerlendirilmesi daha doğru olacaktır.

```
public static void HosGeldinMesajiGoster()
{
    Console.WriteLine("Sisteme hoş geldiniz!");
}

// Başka yerlerde
HosGeldinMesajiGoster();
//...
//...
//...

// Ve başka yerlerde
HosGeldinMesajiGoster();
//...
//...
//...
```

2.1.3. Sihirli Sayılar ve Metinler

Sihirli sayılar, kod içinde doğrudan kullanılan sabit sayılar veya değerlerdir. Bu, kodun anlaşılmasını ve bakımını zorlaştırabilmektedir. Özellikle bu sayılar birden fazla yerde kullanılıyorsa, bir güncelleme gerektiğinde tüm bu kullanımları güncellemek karmaşık hale gelebilir.

```
int yariCap = 7;
double alan = Math.PI * (yariCap * yariCap);
```

Bu örnekte, "7" bir sihirli sayıdır ve kodun anlaşılmasını zorlaştırmaktadır. Daha iyi bir uygulama tasarımı için bu değerin bir sabit değişkene atanması uygun olacaktır.

```
const int YariCap = 7;
double alan = Math.PI * (YariCap * YariCap);
```

Bu şekilde, "YariCap" sabit değişkeni sayesinde kodun anlaşılabilirliği artmakta ve gerektiğinde sadece bir yerde güncellemek yeterli olmaktadır.

Aynı prensip, metinler için de geçerlidir. Metinleri doğrudan kod içinde kullanmak, kodun okunabilirliğini azaltarak bakımını zorlaştırmaktadır.

```
if (rol == "yönetici")
{
    // Admin işlemleri
}
```

Burada, "yönetici" bir sihirli metindir. Bu metnin bir sabit değişkene atanması kodun daha anlaşılır hale gelmesini sağlayacak ve kod kokusunu giderecektir.

```
const string YoneticiRol = "yönetici";

if (rol == YoneticiRol)
{
    // Yönetici işlemleri
}
```

Bu şekilde, "YoneticiRol" sabit değişkeni kodun okunabilirliğini arttırmakta ve metin değiştirilmek istendiğinde sadece bir yerde güncelleme yapmak yeterli olacaktır.

Sihirli sayılar ve metinleri önlemek, kodun kalitesini artırır, bakım maliyetlerini düşürür ve hataları azaltır. Bu nedenle, bu tür sihirli değerler yerine açıkça adlandırılmış sabitler veya enum türleri kullanılması iyi bir uygulama tasarımı olarak ifade edilmektedir.

2.1.4. Karmaşık İf Blokları

```
int x = 10;
int y = 20;
int z = 30;

if (x > 5)
{
    if (y < 15)
    {
        if (z == 30)
        {
            Console.WriteLine("Tüm koşullar doğru.");
        }
        else
        {
            Console.WriteLine("Z koşulu yanlış.");
        }
    }
    else
    {
        Console.WriteLine("Y koşulu yanlış.");
    }
}
else
{
    Console.WriteLine("X koşulu yanlış.");
}
```

Bu kod örneği, iç içe geçmiş if koşullarını kullanarak karmaşık bir kontrol yapısını göstermektedir. Bu tür kod yapısı, kodun anlaşılmasını ve bakımını zorlaştırmaktadır. Aynı zamanda herhangi bir hata ayıklama işlemi de karmaşık hale gelebilir.

```
int x = 10;
int y = 20;
int z = 30;

bool xKosulu = x > 5;
bool yKosulu = y < 15;
bool zKosulu = z == 30;

if (xKosulu && yKosulu && zKosulu)
{
    Console.WriteLine("Tüm koşullar doğru.");
}
else
{
    if (!xKosulu)
    {
        Console.WriteLine("X koşulu yanlış.");
    }
    else if (!yKosulu)
    {
        Console.WriteLine("Y koşulu yanlış.");
    }
    else
    {
        Console.WriteLine("Z koşulu yanlış.");
    }
}
```

Bu şekilde, kod daha okunabilir hale gelmekte ve her koşul açıkça kontrol edilebilmektedir. Bu sayede kaynak kodun anlaşılabilirliği ve bakımı daha kolay hale gelmektedir.

2.1.5. Kötü Adlandırma

Kötü adlandırılmış değişkenler veya fonksiyonlar, kodun okunmasını ve anlaşılmasını zorlaştırabilir.

```
int x = 10;
int y = 5;

int z = x + y;
```

Bu kod örneğinde, değişkenler "x" ve "y" gibi anlamsız ve açıklayıcı olmayan adlara sahiptir. Kodu daha okunabilir ve anlaşılabilir hale getirmek için değişkenlere daha açıklayıcı isimler vermek önemlidir.

```
int toplam = 10;
int eklenenDeger = 5;

int sonuc = toplam + eklenenDeger;
```

Bu örnekte değişkenler daha açıklayıcı adlara sahiptir, bu nedenle kodun ne yaptığı daha kolay anlaşılabilir hale gelmektedir. İyi adlandırma, kodun diğer geliştiriciler veya bakım ekibi tarafından daha kolay anlaşılmasını sağlamakta ve kod kalitesini arttırmaktadır. Ayrıca, kodun belgelendirilmesi ve yorumlarla desteklenmesi de kodun anlaşılmasına yardımcı olmaktadır.

2.2. Uygulama Örnekleri

Aşağıda verilen kod bloğundaki kod kokularını belirleyiniz.

```
public void ToplamaIslemi(int x, int y)
{
    int sonuc = 0;
    if (x > 5)
    {
        sonuc = x + y;
        if (y < 10)
        {
            sonuc = sonuc * 2;
        }
    }
    else
    {
        sonuc = x - y;
        if (y > 5)
        {
            sonuc = sonuc / 2;
        }
    }
    Console.WriteLine("Sonuç: " + sonuc);
}
```

Bu kod bloğunda karmaşık koşullu ifadeler, tek sorumluluk ilkesi ihlali, sihirli sayı ve kötü adlandırma bulunmaktadır.

```
public void IslemSirasi(int miktar, bool premiumMusteri)
{
    if (miktar > 0)
    {
        if (premiumMusteri)
        {
            double indirim = 0.1;
            double toplam = miktar * 10;
            double toplamIndirim = toplam * (1 - indirim);
            Console.WriteLine("Toplam Tutar (Premium Müşteri): " +
toplamIndirim);
        }
        else
        {
            double toplam = miktar * 10;
            Console.WriteLine("Toplam Tutar: " + toplam);
        }
    }
    else
    {
        Console.WriteLine("Hatalı miktar.");
    }
}
```

Bu kod bloğunda derin iç içe koşullu ifadeler, tek sorumluluk ilkesi ihlali, sihirli sayı kokuları bulunmaktadır.

```
using System;

class Program
{
    static void Main()
    {
        int n = 5;
        int a = 1;
        int b = 1;
        int c = 0;
        string sonuc = "";

        for (int i = 0; i < n; i++)
        {
            c = a + b;
            a = b;
            b = c;
            sonuc += "Fibonacci(" + i + ") = " + c + ", ";
        }

        Console.WriteLine("Sihirli Fibonacci Dizisi: " + sonuc);
    }
}
```

Bu kod bloğunda sihirli metin ve kötü adlandırma bulunmaktadır.

2.3. Çeşitli Kod Kokuları

2.3.1. Yöntem Düzeyinde Kod Kokuları

Ölü kod	Yöntemin hiç kullanılmaması durumudur.
Tembel nesne	Yöntemin çok az iş yapması durumudur.
Aracı	Yöntemin işlevini başka yöntemlere yaptırması durumudur.
Tanrı yöntemi	Yöntemin bir sürü iş yapması durumudur.
Uzun parametre listesi	Yöntemin çok sayıda parametre içermesi durumudur.
Döngüsel karmaşıklık	Yöntemin çok sayıda döngü ya da koşullardan oluşması durumudur.
Özellik kıskançlığı	Yöntemin kendi verisinden çok başka bir nesnenin verisine erişmesi durumudur.
Kara koyun	Yöntemin aynı sınıftaki diğer tüm yöntemlerden belirgin şekilde farklı olması durumudur.

2.3.2. Sınıf Düzeyinde Kod Kokuları

Tembel nesne	Sınıfın çok az iş yapması durumudur.
Aracı	Sınıfın işlevini başka sınıflara yaptırması durumudur.
Tanrı nesnesi	Sınıfın bir sürü iş yapması durumudur.

2.3.3. Genel Kod Kokuları

Girinti kaybı	Kaynak kodun amacını yerine getirememesi durumudur.
Tuhaf çözüm (oddball)	Bir sorunun farklı şekillerde çözülmesidir.
Yinelenen kod	Neredeyse aynı kaynak kodların görünmesi durumudur.
Shotgun cerrahisi	Tek bir değişikliğin çok sayıda uygulamayı kapsaması durumudur.
Deodorant yorumları	Kötü kodu güzelleştirmek için yapılan **GÜZEL** yorumlar.