T.C.
Fırat Üniversitesi
Fen Bilimleri Enstitüsü

Yazılım Test Süreçlerinin İncelenmesi ve Jenkins Platformunda Uygulamaları

Ayşe KAHVECİ YETİŞ

Yüksek Lisans Tezi

Yazılım Mühendisliği Anabilim Dalı

Şubat – 2022

T.C.
Fırat Üniversitesi
Fen Bilimleri Enstitüsü

Yazılım Mühendisliği Anabilim Dalı

Yazılım Test Süreçlerinin İncelenmesi ve Jenkins Platformunda Uygulamaları

Tez Yazarı

Ayşe KAHVECİ YETİŞ

Danışman

Prof. Dr. Resul DAŞ

Şubat – 2022
Elazığ

T.C.
Fırat Üniversitesi
Fen Bilimleri Enstitüsü

Yazılım Mühendisliği Anabilim Dalı

Yüksek Lisans Tezi

Başlık:	Yazılım Test Süreçlerinin İncelenmesi ve Jenkins Platformunda Uygulamaları
Yazar:	Ayşe KAHVECİ YETİŞ
Teslim Tarihi:	04/02/2022

Yüksek Lisans Tez Onayı

Fırat Üniversitesi Fen Bilimleri Enstitüsü tez yazım kurallarına göre hazırlanan bu tez aşağıda imzaları bulunan jüri üyeleri tarafından değerlendirilmiş ve akademik dinleyicilere açık yapılan savunma sonucunda OYBİRLİĞİ ile kabul edilmiştir.

Bu tez, Enstitü Yönetim Kurulunun / /2022 tarihli toplantısında tescillenmiştir.

Prof. Dr. Kürşat Esat ALYAMAÇ
Enstitü Müdürü

Beyan

Fırat Üniversitesi, Fen Bilimleri Enstitüsü tez yazım kurallarına uygun olarak hazırladığım "Yazılım Test Yöntemlerinin İncelenmesi ve Jenkins Platformunda Uygulamaları" başlıklı Yüksek Lisans tezimin içindeki bütün bilgilerin doğru olduğunu, bilgilerin üretilmesi ve sunulmasında bilimsel etik kurallarına uygun davrandığımı, kullandığım bütün kaynakları atıf yaparak belirttiğimi, maddi ve manevi desteği olan tüm kurum/kuruluş ve kişileri belirttiğimi, burada sunduğum veri ve bilgileri unvan almak amacıyla daha önce hiçbir şekilde kullanmadığımı beyan ederim.

04/02/2022

Ayşe KAHVECİ YETİŞ

Önsöz

Yüksek lisans öğrenimime başladığımdan beri çalışmalarımın her aşamasında yardımlarını esirgemeyen, öneri ve kılavuzluk yaparak çalışmalarıma yol veren, akademik hayatıma farklı bakış açısı kazandıran, akademik kariyerim boyunca da ilgi, destek ve tecrübelerini paylaşan, tertipli, düzenli ve titiz çalışarak bana nitelikli ve değerli iş üretmemi kazandıran kıymetli hocam danışmanım Prof. Dr. Resul DAŞ'a yürekten teşekkürlerimi ve şükranlarımı sunuyorum. Ayrıca, akademik çalışmalarımı yürütürken yanımda yardımlarını hissettiğim tüm çalışma arkadaşlarıma teşekkür ediyorum. Özellikle, beni büyüten ve bu yaşıma kadar tüm eğitim yaşamımda ve çalışmalarımda beni yüreklendiren canım anneme, babama ve değerli eşime yürekten teşekkür ediyorum.

Ayşe KAHVECİ YETİŞ

Elazığ, 2022

İçindekiler

Sayfa

Özet

Yazılım Test Süreçlerinin İncelenmesi ve Jenkins Platformunda Uygulamaları

Ayşe KAHVECİ YETİŞ

Yüksek Lisans Tezi

Fırat Üniversitesi
Fen Bilimleri Enstitüsü

Yazılım Mühendisliği Anabilim Dalı
Şubat 2022 , Sayfa: xiv + 122

Yazılım geliştirme yaşam döngüsü süreçlerinin her aşamasında doğrulama ve geçerleme işlemleri uygulanmaktadır. Müşteri/kullanıcı memnuniyetinin ve yazılım kalitesinin arttırılması için yazılım gerçekleştirildikten sonra birbirinden farklı birçok yazılım testleri uygulanmaktadır. Bu testler sonucunda, kullanıcı dostu, kod karmaşıklığı analiz edilmiş, sürekli geliştirilebilir ve yüksek güvenlik seviyesinde yazılımlar ortaya konulmaktadır.

Bu tez çalışmasında, yazılım uygulamalarında karşılaşılan hatalar irdelenmiş ve kategorize edilmiştir. Yazılım kalitesinin incelenmesinde önemli aşamalar olan, yazılım test yaşam döngüsü süreçleri, test teknikleri, test seviyeleri ve test türleri kapsamlı bir şekilde incelenmiştir. Yazılım kalitesinin arttırılması için kullanılan kalite nitelikleri ve yazılım ölçütleri detaylıca analiz edilmiştir. Ayrıca, Java programlama dilinde temel düzeyde bir kütüphane uygulaması geliştirilmiş ve *Java* platformunda *JUnit* ile birim testleri yazılmıştır. Özellikle açık kaynak kodlu ve sürekli entegrasyon biçiminde ilerleyen *Jenkins* platformu yardımıyla, geliştirilen yazılımın test uygulamaları gerçekleştirilmiştir. Geliştirilen uygulama, *Java Eclipse CodeMR* kütüphanesi yardımıyla kod karmaşıklığı analiz edilmiş, kod bloklarındaki risk dereceleri belirlenmiştir.

Anahtar Kelimeler: Yazılım test yaşam döngüsü, Yazılım test teknikleri, Yazılım test seviyeleri, Yazılım test türleri, Yazılım ölçütleri, Jenkins test platformu .

Abstract

Analysis of Software Testing Processes and Applications on Jenkins Software Platform

Ayşe KAHVECİ YETİŞ

Firat University
Graduate School of Natural and Applied Sciences

Software Engineering
February 2022 , Page: xiv + 122

Validation and verification processes are applied at every stage of the software development lifecycle processes. In order to increase customer/user satisfaction and software quality, many different software tests are applied after the software is developed. As a result of these tests, software that is user-friendly, whose code complexity is analyzed, that can be continuously improved and that has a high security level is revealed.

In this thesis, the errors encountered in software applications were examined and categorized. Software test life cycle processes, test techniques, test levels and test types, which are important stages in the examination of software quality, have been extensively studied. Quality attributes and software criteria used to increase software quality have been analyzed in detail. In addition, a basic library implementation was developed in the *Java* programming language and unit tests were written with *JUnit* on the Java platform. Test applications of the developed software were carried out, especially with the help of the open source and continuous integration *Jenkins* platform. The code complexity of the developed application was analyzed with the help of the *Java Eclipse CodeMR* library, and the risk levels in the code blocks were determined.

Keywords: Software testing life cycle, Software testing techniques, Software testing levels, Software testing types, Software metrics, Jenkins testing platform.

Şekiller Listesi

Sayfa

Tablolar Listesi

Sayfa

Kod Örnekleri Listesi

Kısaltmalar Listesi

#AF	: Number of Accessed Fields (Erişilen Alan Sayısı)
AHF	: Attribute Hiding Factor (Nitelik Gizleme Faktörü)
AI	: Attribute Inheritance Factor (Nitelik Kalıtım Faktörü)
ANA	: Avarage Number of Ancestors (Ortalama Ata Sayısı)
CAM	: Cohesion Among Methods (Metotlar Arası Uyumluluk)
CBO	: Coupling Between Object Classes (Nesne Sınıfları Arasındaki Bağımlılık)
CC	: Cyclomatic Complexity (Döngüsel Karmaşıklık)
CF	: Coupling Factor (Bağımlılık Faktörü)
CIS	: Class Interface Size (Sınıf Arayüz Boyutu)
CP	: Comment Percentage (Yorum Oranı)
DAM	: Data Access Metric (Veri Erişim Metriği)
DCC	: Direct Class Coupling (Doğrudan Sınıf Bağımlılığı)
DevOps	: Developers and Operations (Java Çalışma Zamanı Ortamı)
DIT	: Depth Of Inheritance Tree (Kalıtım Ağacının Derinliği)
ETSI	: European Telecommunications Standartds Institue (Avrupa Telekomünikasyon Standartları Enstitüsü)
EXEC	: Executable Statements (Çalıstırılabilir Yordam Sayısı)
GUI	: Graphical User Interface (Grafiksel Kullanıcı Arayüzü)
IEEE	: Institue of Electrical and Electronics Engineers (Elektrik ve Elektronik Mühendisleri Enstitüsü)
IIBA	: International Institude of Business Analysis (Uluslararası İş Analizi Enstitüsü)
ISO/ IEC	: International Organization for Standardization / International Electrotechnical Commission (Uluslararası Standardizasyon Kurumu / Uluslararası Elektroteknik Komisyonu)
ISTQB	: International Software Testing Qualifications Board (Uluslararası Yazılım Testi Yeterlilikler Kurulu)
JDK	: Java Development Kit (Java Geliştirme Kiti)
JRE	: Java Runtime Environment (Java Çalışma Zamanı Ortamı)
LCOM	: Lack of Cohesion in Methods (Metot İçi Uyumsuzluk)
LOC	: Lines Of Code (Kod Satırları)
LLOC	: Logical Lines of Code (Koddaki Mantıksal Satırlar)
#MC	: Number of Methods Called (Çağrılan Metot Sayısı)
MCC	: McCabe Cyclomatic Complexity (McCabe Döngüsel Karmaşıklık)
MFA	: Measure of Functional Abstraction (İşlevsel Soyutlama Ölçüsü)
MHF	: Method Hiding Factor (Metot Gizleme Faktörü)
MIF	: Method Inheritance Factor (Metot Kalıtım Faktörü)
MOA	: Measure Of Aggregation (Kümeleme Ölçüsü)
MOOD	: Object-Oriented Design Metrics (Nesneye Yönelik Tasarım Metrikleri)
NBD	: Nested Block Depth (İç İçe Bulunan Metot Derinlikleri)
NCNB	: Non-Comment Non-Blank (Yorum ve Boşluk İçermeyen Satır Sayısı)
NOC	: Number Of Children (Alt Sınıf Sayısı)
NOM	: Number of Methods (Metot Sayısı)
#Pa	: Number of Parameters (Toplam Parametre Sayısı)
PF	: Polymorphism Factor (Çok Biçimlilik Faktörü)
PLOC	: Physical Lines Of Code (Koddaki Fiziksel Satırlar)
RFC	: Response For a Class (Sınıfın Tetiklediği Metot Sayısı)
SDLC	: Software Development Life Cycle (Yazılım Geliştirme Yaşam Döngüsü)
SEI	: Software Engineering Institute (Yazılım Mühendisliği Enstitüsü)

SLOC	: Source Lines of Code (Kaynak Kod Satır Sayısı)
SWEBOK	: Software Engineering Body of Knowledge (Yazılım Mühendisliği Bilgi Grubu)
UML	: Unified Modelling Language (Birleşik Modelleme Dili)
QMOOD	: Quality Model for Object Oriented Design (Nesneye Yönelik Tasarım İçin Kalite Modeli)
WMC	: Weighted Methods Per Class (Sınıf Başına Ağırlıklı Metot Sayısı)
W3C	: World Wide Web Consortium

1. Giriş

Profesyonel yazılımın geliştirilme sürecinde, Yazılım Geliştirme Yaşam Döngüsü (SDLC-Software Development Life Cyle) adımları dikkate alınır. Bu bağlamda klasik ve çevik yazılım geliştirme süreç modellerinden uygun olan model tercih edilerek süreçler gerçekleştirilir. Bu süreçler ve işlemler, kullanıcı ihtiyaçlarının karşılanması ve kaliteli yazılım ortaya konulabilmesi için gereklidir. Bir süreç modeline göre uygulama gerçekleştirildiğinde yazılım kalite güvencesi kriterlerine uygun yazılımların ortaya konulması mümkün olmaktadır. Yazılım kalitesi, yaşam döngüsü boyunca arttırılması hedeflenen bir süreç olmalıdır. Yazılımda kalitenin yakalanması da, standartları içeren resmi süreçlere göre gerçekleştirilen test süreçlerine bağlıdır.

Hatasız yazılımın mümkün olmadığı kabul edilir bir gerçektir. Yazılım testlerinin gerçekleştirilme amacı minimum hata oranı ile maksimum kaliteye sahip yazılım ürünlerinin elde edilmesi ve bu kalitenin yazılım yaşam döngüsü boyunca artarak sürdürülmesidir. Yazılımda kalite kavramı, program geliştiriciler ve test ekibi arasındaki çatışmanın bir ürünü olarak tanımlanabilir. IIBA (International Instıtude of Business Analysis) ve ISTQB (Internatinal Software Testing Qualifications Board) gibi uluslararası otoritelerin açıklamalarına göre, testler bazı hedeflere sahip olmalıdır [1]. Bu hedefler hataları bulmak, kalite düzeyi hakkında güven kazanmak, karar vermek için bilgi sağlamak, hataları önlemek olarak belirlenir.

1.1. Tez Çalışmasının Amacı

Tez çalışmasının amacı, geliştirilen yazılım projelerinde büyük oranda göz ardı edilen test yaşam döngüsü sürecinin önemini detaylarıyla uygulamalı olarak incelemektir. Tez çalışmasında, yazılım hataları ve etkileri, test yaşam süreci, yaşam sürecinde gerçekleştirilen testlerin teknikleri ve seviyeleri, test türleri ve yazılım kalite ölçütleri alt başlıklar halinde irdelenmektedir. Yazılım kalite ölçütlerinde sürekli entegrasyon mantığı ile çalışan, açık kaynak kodlu Jenkins platformunda sistemin testleri gerçekleştirilmiş ve uygulama sonuçları verilmiştir. Tez çalışmasında, test işleminin test yaşam döngüsü verilerini verimli kullanmak adına erken evrelerde başlaması gerektiği, fark edilen hataların sınıflandırılması, raporlanması ve hatanın kategorisine uygun optimum adımlarla çözümlenmesi gerektiği açıkça ortaya konulmuştur.

1.2. Tez Çalışmasının Kapsamı

Bu tez çalışması, yazılım test süreci ile ilgili önemli konuları içeren 10 ana bölümden oluşmaktadır.

1. bölümde, yazılım hataları, yazılım testleri, yazılım kalitesi gibi konular hakkında literatür araştırması yapılmış ve bulunan kaynaklardan konuyla ilgili faydalı bilgilere yer

verilmiştir.

2. bölümde, yazılım hatalarının çözüm süreçlerine yer verilmiştir. Yazılım hataları, etki durumları ve sonuçlarına göre sınıflandırılmıştır.

3. bölümde, yazılım testi, test yaşam döngüsü, test geliştirme yaklaşımları, doğrulama ve geçerleme kavramları alt başlıklarıyla detaylıca incelenmiştir.

4. bölüm, yazılım testi gerçekleştirilirken faydalanılan kara kutu, gri kutu ve beyaz kutu test teknikleri detaylıca incelenmiştir.

5. bölüm, yazılım ürününün geliştirilme aşamalarında gerçekleştirilmesi gereken test adımları test seviyelerine göre detaylıca incelenmiştir.

6. bölüm, en yaygın kullanılan yazılım test türleri inceleenmiş, uygulandığı noktalar özetlenerek tablo halinde sunulmuştur.

7. bölümde, yazılımda kalite güvencesi ana başlığında, yazılım kalite nitelikleri açıklanmış ve ISO/IEC-9126 kalite standartları incelenmiştir.

8. bölüm, geleneksel ve nesne tabanlı yazılım kalite ölçütleri alt başlıklarıyla detaylıca incelenmiştir.

9. bölüm, Jenkins test platformu ve genel özellikleri detaylıca incelenmiştir. Yazılım testlerinin özelleştirilmesinde ve testlerin çalışma sırasının önceliklendirilmesinde kullanılan Junit notasyonları uygulanmıştır. Java ortamında geliştirilen kütüphane otomasyonu uygulaması sürekli entegrasyon mantığı eşliğinde Jenkins bağlantısıyla test edilmiştir. Eclipse destekli CodeMR kütüphanesi ile uygulamaya ait kod ölçütleri analiz edilmiştir.

10. bölüm, tezin genel sonucu sunulmuştur.

1.3. Literatür Taraması ve Değerlendirilmesi

2016 yılında geliştirilen Android Mobil Uygulamalar için Yazılım Testi adlı çalışmada, mobil kullanıcıların sayısı ile mobil uygulamalardan beklenen kalitenin aynı oranda arttığından söz edilmiştir. Özellikle yaygın kullanımı nedeniyle Android işletim sisteminde yer alan uygulamaların test edilmesinin kendine göre zorlukları olduğu belirtilmiştir. Çalışmada yer alan bilgiler dikkate alındığında bir mobil uygulamanın testi için dikkat edilmesi gereken unsurlar, kullanılan cihazların yazılımsal veya donanımsal olarak farklı özelliklere sahip olması, donanımda yer alan batarya, ekran boyutu, çoklu giriş aygıtları,

tuş takımı gibi özellikler, donanım platformundaki kaynak kısıtlılığı şeklinde örneklendirilebilir. Çalışmada mobil uygulamaların denetimi için otomasyon ile test yönteminin zaman, maliyet, test uzmanı tarafından oluşabilecek aksaklıklar yada hatalar gibi ölçütler için optimum başarı sağlayacağı vurgulanmıştır [2].

2013 yılında yayınlanan Türkiye'deki Yazılım Test Uygulamaları Anketi konulu bildiride amaç, Türk yazılım endüstrisinin test sürecine bakış açısını öğrenmek ve iyileştirilmesi gereken yönlerin keşfini sağlamak şeklinde belirtilmiştir. Bu bildiri çalışması sırasında, test seviyeleri ve bu seviyelere uygun test türleri, test için gerekli ölçütler ve Türk test mühendislerinin zorlandığı başlıklar incelenmiştir. Ayrıca Software Engineering Body of Knowledge (SWEBOK) kaynağında yer alan bilgiler de dikkate alınarak hazırlanan 54 soruluk çevrimiçi anketin 9'u yazılım testi odaklıdır. 163 yazılım mühendisinin katıldığı anket sonuçlarına göre Türkiye'de en yaygın kullanılan test türleri fonksiyonel test, kullanıcı kabul testi ve entegrasyon testi olarak belirtilir. En az kullanılan test tekniği ise stres testidir. Bu çalışmaya göre Türkiye'de kod satırına düşen hata sayısı, kullanıcı kabul testleri sonucu başarı, kalite metrikleri gibi kriterlerin yaygın olarak kullanılmadığı belirtilmiştir [3].

U.Zöngür ve diğerleri tarafından geliştirilen çalışmanın temel amacı, test yaşam döngüsü boyunca harcanan zaman ve maliyet metriklerini minimize etmektir. Çalışmada, Cobalt kütüphanesinin bir ağaç yapısıyla ikili verileri modellediği belirtilmiştir. Yapılan çalışmalar sonucu Cobalt kütüphanesinin uygulamanın geliştirilmesi ve sürdürülmesi sürecinde maliyeti düşürdüğü savunulmuştur. Cobalt kullanımı ile kaliteli ve uzun süreli kullanıma sahip uygulamaların geliştirildiği belirlenmiştir [4].

O.Özçelik tarafından 2015'te yayınlanan tez çalışmasında askeriye, ulaşım ve medikal gibi branşlarda yoğun olarak kullanılan yazılımların emniyetli olması gerektiği belirtilmiştir. Olası felaketleri önlemek amacıyla sisteme, planlama aşamasından itibaren düzenli ve sağlıklı test türlerinin uygulanması gerektiği vurgulanmıştır. Bu çalışma kapsamında, kritik öneme sahip yazılım sistemlerinin test edilebilirliği ile nesneye yönelik yazılım geliştirme yöntemleri arasındaki ilişki araştırılmıştır. Araştırma için Tübitak ve İstanbul Teknik Üniversitesi ortaklığında geliştirilen Ulusal Demiryolu Sinyalizasyonu Projesi (UDSP) incelenmiştir. Kullanılan yazılımın kalitesi, nesneye yönelik ve nesneye yönelik olmayan metrik türlerine göre ölçülmüş ve yazılımın test edilebilirliği değerlendirilmiştir. Çıkan sonuçlar dikkate alınarak kullanılan yazılımın ikinci sürümü geliştirilmiştir. Test yönelimli yazılım geliştirme teknikleri ile gerçekleştirilen sürümün test edilebilirliğinin ilk sürüme göre iyileştirildiği belirtilmiştir [5].

2012 yılında yazılan tez çalışmasında, model denetleme temelli yazılım test ve analiz yöntemlerinin uygulanabilirliği arttırdığı savunulmuştur. Testler gerçekleştirilirken daha az kaynak kullanımı için otomatikleştirmede artış görülmüştür. Bu duruma örnek gelişmelerden biri de model denetleme araçlarının kullanılmasıdır. Çalışmadaki amaç, güvenliğin büyük önem taşıdığı bir senaryoda, gereksinimlere bağlı test durumlarının otomatikleştirilmesinin bir prototip oluşturmasıdır. Çalışma için akıllı kart uygulaması

olarak hazırlanan WIM aracılığıyla iletim seviyesi güvenliğinin sağlanması kontrol edilmiştir. WIM, kablosuz iletişim sistemi olan GSM telefon standartlarında uygulama protokolü kimlik modülüdür. Vaka çalışmasında incelenen test durumları fonksiyonel, yapısal ve olasılıklı rastlantısal olmak üzere üç başlık altında toplanmıştır. Testlerin uygulanması ile model ve kart arasında var olan uyumsuzluklar bir tablo ile listelenmiştir. Çalışmadaki amaç, model bazlı testlerde belirli metotların olumlu bir takım yönleri ortaya çıkarmasıdır. Ortaya çıkan sonuca göre model bazlı test yöntemleri ile manuel test yöntemlerinin kabul edilebilir derecede bağdaştığı gözlemlenmiştir [6].

G.Kuday tarafından ele alınan Yazılım Mühendisliği Yöntemleriyle Yazılım Test Süreci adlı çalışmada, yazılım testinin, sistemin barındırdığı riskler hakkında bilgi veren eylemler olduğu belirtilmiştir. Çalışmada, yazılım testlerinin başarısızlıkları ortadan kaldırmak ve olası başarısızlıkları önlemek için gerçekleştirdiği belirtilir. Gerçekleştirilen test durumları sonucu hazırlanan raporlarda, testin görevi kısaca isimlendirilir. Test açıklaması ile kısaca isimlendirilen test görevinin neden ve nasıl yapıldığı belirtilir. Devamında beklenen sonuç bölümüyle gerçekleştirilen test durumlarında belirtilen tüm adımların gerçekleştirilmesi ile ortaya çıkan sonuçtur. Bu çalışmada Plague Inc. Evolved adlı oyunun Evrim 11 versiyonu için kullanıcı arayüzü testi gerçekleştirilmiştir. Ana teması, gerçek hayatta var olan yada bilim kurgusal olarak varsayılan hastalıkların, belirli ülkelerden başlanarak tüm dünyaya yayılması ve insanlığın yok edilmesi hedeflenir. Seviyelere göre hastalıklar, senaryolar ve ülkeler artış gösterir. Çalışma kapsamında, oyunun giriş filmindeki görüntü ve seslerin senkronizasyonu, ses düzeylerinin veya butonların çalışabilirliği, arayüzler ve senaryolar arasındaki geçişlerin sistematik bir şekilde ilerlemesi, açıklama ve yönlendirmelerin eksiksiz olması kontrol edilmiştir. Gerçekleştirilen testlere ve fark edilen eksikliklere göre gerekli düzenlemelerin yapıldığı çalışmada belirtilmiştir [7] .

Nesne yönelimli sistemlerde test işlemlerini ve metrik kümelerin değerlendirilmesi için geliştirilen modülün incelendiği 2011 tarihli çalışma, özellikle nesne yönelimli yazılım geliştirme süreçlerindeki test adımlarına odaklanmıştır. Yazılım test işlemlerinin adımlarını daha iyi yönetebilmek, yorumlayabilmek ve daha başarılı sonuçlar üretmek için metriklerden yararlanıldığı vurgulanmıştır. Metriklerin, daha fazla test gerektiren riskli alanları belirlediği, potansiyel problemlerin çözümüne yardım ettiği, testin etkililiğini arttırdığı vurgulanmıştır. Ayrıca çalışmada yazılım testinin sistemdeki eksiklikleri bulmak için gerekli bir süreç olduğu ve bu sürecin en iyi sonucu verebilmesi için metriklerden faydalanılması gerektiği belirtilmiştir. Test sürecinin dikkate alınmaması sonucu kalitesiz ve başarısız yazılımların elde edildiği belirtilmiştir. Çalışmayı destekler nitelikte, Metrics adı verilen programın bulguları, sınıf değerlendirmesi ve sürümler arası değerlendirme olmak üzere incelenmiş ve bunu sağlamak için bir uzman modül geliştirilmiştir. Sınıf değerlendirilmesinin amacı, nesne yönelimli yazılımların genel olarak sınıf mantığıyla çalışmasıdır. Uzman modül aracılığıyla .jar uzantılı dosya yada projelerde var olan metrik bulgular hatasız bir şekilde değerlendirilir. Modülden faydalanılarak, Chidamber ve Kemerer metriklerinin aralıklarının ortalaması ile otomatik değerlendirme gerçekleştirilir.

Değerlendirilecek olan bulgular belirtilen aralıklarda ise olumlu, değilse olumsuz sonuçlar elde edilir. Geliştirilen modülün değerlendirme kısmında, özelliklere göre metriklerin ölçümü ve yorumlanması gerçekleştirilmiştir. Sürümler arası değerlendirme kısmında, daha önceki sürümler ile yeni sürümün metrik bulgularında karşılaştırmalar yapılarak yazılım kalitesi yorumlanmıştır. Çalışma sonunda geliştirilen modülün, yazılımlarda kalite kriterinin artmasına fayda sağlanması amaçlanmıştır [8].

A.B.Karagöz tarafından gerçekleştirilen çalışmada amaç, yazılım test sürecinin önemini ve uygulamalarda test süreci sonunda artan kaliteyi gözler önüne sermektir. Çalışmada, yazılım testinin hataları keşfedip çözüme ulaştırması ile mevcut uygulamada tanımlanan kalite oranını arttırmanın hedeflendiğinden söz edilmiştir. Çalışmayı destekler nitelikte bir şirkette geliştirilen uygulamalar değerlendirilmiştir. Test sürecinin, projenin hangi aşamasında gerçekleştirildiği, test sonucu raporlanan hataların düzeltilme ölçütlerinin dikkate alındığı belirtilmiştir. Çalışmada uygulamanın kalitesinin artması için test sürecinin yazılım yaşam döngüsünün erken safhalarında başlanması gerektiği, test ekibinin titiz çalışması sonucu olası hataların sağlıklı bir şekilde raporlanması ve yok edilmesi gerektiği ifade edilmiştir [9].

2017 yılında A.K.Hancı tarafından kaleme alınan tez çalışmasında yazılım testi tasarım tekniklerinin matematiksel modellemesi ve bu modellemenin analizi yer almaktadır. Çalışmada, Türkiye'de gerçekleştirilen yazılım projelerinde yer alan test tasarım teknikleri ve kullanım sıklıkları, hata bulma oranı ve uygulanabilirlik oranı değerlendirilerek test süreci için analiz yapıldığı belirtilmiştir. Çalışmada gereksinim odaklı tekniklerden en çok kullanılan denklik paylarına ayırma, sınır değer analizi, karar tabloları, durum geçiş diyagramları ve kullanım senaryoları hakkında bilgi verilmiştir. Bu tekniklerin hata bulma oranlarına ve uygulanabilirlik oranına göre değerlendirilmesi tablo halinde verilmiştir. Yöntemlerde, Türkiye'deki yazılım testi alanında görevli olanları analiz etmek ve tasarım tekniklerinin kullanım oranlarını belirlemek için hazırlanan anketin çözümlemesi ve yorumlanması yer alır. Anket içeriğinin katılımcı, sektör, proje profilleri ve test teknikleri gibi konulardan oluştuğu belirtilmiştir. Gereksinim bazlı test tekniklerinin analizinde, kullanım sıklığına sahip olan tekniğin sınır değer analizi olduğu belirtilmiştir. Ancak yapılan matematiksel yaklaşımlar sayesinde gereksinim bazlı test tasarım teknikleri için en uygun modelin karar tablosu olduğu görülmüştür [10].

2012 yılında uzaktan eğitimde ölçme ve değerlendirme sistemini tasarlamak üzere yayınlanan çalışmada, çeşitli alanlarda sıkça kullanılan yazılım test teknikleri bu sisteme uygulanmıştır. Tüm test sonuçları dikkate alınarak hatalar giderilmiştir ve sistemin minimum hatayla kullanılırken farklı uzaktan eğitim ölçme ve değerlendirme sistemlerine model olması beklenmektedir. Sistem geliştirilirken yararlanılan platformlar ve sürümleri ASP.Net Framework 4.0 ve SQL Server 2005 olarak belirtilmiştir. Daha sonra geliştirilen sistem üzerinde Selenium IDE, Vega Subgraph, SqlMon, SqlQueryStress, PerformanceAnalyzer gibi test platformları yardımıyla testler gerçekleştirilmiş. Bahsedilen test platformları ile geliştirilen sistemin giriş sayfası, soru ekleme sayfası, linkleri, web

elemanlarının işlevleri, sistemin güvenliği, veri tabanı performansı, veri tabanı sorguları, CPU kullanımı gibi özellikler test edilmiştir. Bu testler sonucunda sistemin çalışmasını olumsuz etkileyen hatalar düzeltilmiştir. Son olarak, yazılım test araçlarının ve tekniklerinin kullanılmamasının sistemi olumsuz etkileyeceği ve bu durumunda bakım maliyetlerini arttıracağı belirtilmiştir [11].

2013 yılında N. Yağcı tarafından yayınlanan çalışmada, yazılım kalite kıstaslarının test sürecinde sarf ettiği efor incelenmiştir. Yazılım yaşam süresince, test efor tahminlemesinin, test işlemi boyunca harcanması planlanan kaynak ve zaman olduğu belirtilmiştir. Test edilecek çalışma için gereken zaman ve kaynakların projenin büyüklüğüne ve riskine bağlı olduğu vurgulanmıştır. Çalışmada, yazılım geliştirme eforundan yararlanma, işlev noktaları analizi, test noktası analizi ve kullanım durumu noktası analizi olmak üzere 4 farklı test efor tahmini yöntemi kullanılmıştır. Söz konusu tezde test efor kriteri, sınıf ve metot bazlı yazılım kalite metriklerine bağlı olarak 2 farklı çalışma üzerinden değerlendirilmiştir. İlk çalışmada projenin kodları üzerinde kalite metriklerine göre metot ve sınıf bazlı test işlemi uygulanır. Test işlemi sonrası ortaya çıkan eforların aritmetik ortalaması alınmış ve daha sonra yapılan kıyaslamalarda metriklerin hem metot hem de nesne bazında test eforu ile ilişkili olduğu görselleştirilmiştir. İkinci çalışmada ise ilk çalışmada yer alan metriklerden faydalanılmıştır. 3 farklı büyüklükteki projelerin toplamında yer alan metriklerin metot ve sınıf bazlı değerlendirilmesi dikkate alınmıştır. Alınan değerler normalize edilip toplanmıştır. Sonraki aşamada her proje için metot ve sınıf bazlı olarak toplam metrik değerleri hesaplanmıştır. Önceki aşamalarda hesaplanan test efor bilgileri ile toplam metrik değerleri kıyaslanmıştır. Çalışma sonunda metot bazlı metriklerin, test efor süresi belirlemede yetersiz kaldığı belirtilmiştir. Bununla birlikte sınıf bazlı metriklerin test efor süresi belirlemede faydalı olduğu vurgulanmıştır [12].

2. Yazılım Hatalarının İncelenmesi

Yazılım, kullanıcı istekleri doğrultusunda, bir bilgisayar sistemini komutlarla tasarlanan uygulamalar yardımıyla yönetme işlemidir. Yazılımın sağlıklı bir şekilde çalışması, belirli adımların tamamlanması ile mümkündür. Yazılım geliştirme yaşam döngüsü süresince kullanıcı kriterleri dikkate alınarak bir sistemin üretilmesi hedeflenir. Üretilen yazılımda amaç, optimum maliyet ve süre ile yüksek kaliteli bir ürün oluşturabilmektir. Yazılımın kalitesi, sistemin beklentileri karşılama düzeyinin yüksekliği ve minimum zaman ve maliyet ile ölçülür. Bunun yanında sistemin kalite ölçütlerinden bir diğeri ise minimum hataya sahip olmasıdır.

Geliştirilen sistemlerin büyüklüğü fark etmeksizin en az birkaç hataya sahip olduğu bilinen bir gerçektir. Hatasız yazılımın mümkün olmadığı göz önüne alınırsa, geliştirilen yazılımın minimum hataya sahip olması yazılımın kalite oranını gösterir. Projenin kapsamı baz alınırsa hataların önem derecesi ve buna bağlı olarak zararın büyüklüğü hakkında yorum yapılabilir.

Yazılım testi aslında bir sistemdeki hataları bulmayı amaçlayan bir süreçtir. Test, hata ayıklama ve kabul amacıyla da kullanılabilir. R. Vaderwall'un belirttiği gibi "Yazılım testi, bir ürünün davranışını tahmin etme ve bu tahmini gerçek sonuçlarla karşılaştırma sürecidir" [13]. Hatalı yazılımlar sonucu meydana gelen maddi ve manevi zararların tarihte pek çok örneği bulunmaktadır. Tarihteki bu hatalara aşağıdaki örnekler verilebilir [14]:

- *Patriot Füze Hatas:* 1991 yılında yaşanan Körfez Savaşı'nda, Amerika tarafından İsrail'e yerleştirilen Patriot füzelerinden biri, Irak'tan gelen Scud füzesini imha edemedi ve Amerikan askerlerine sahip bir barakaya isabet etti. Bu olay, 29 Amerikan askerinin ölümüne neden oldu.Daha sonra gerçekleştirilen incelemelerde, Patriot füzelerinde zaman hesaplaması için kullanılan 24 bitlik değişkende oluşan bir zaman hatası fark edilmiştir. Bu hata nedeniyle füzede 500 metrelik bir sapma oluşmuştur [15].

- *Pentium İşlemci Sorunu:* 1993 yılında Intel Pentium işlemci sorunu yaşanmıştır. Bilgisayar yongasında bulunan bir hata nedeniyle belirli aralıkta yer alan sayıların bölünmesi hatalı sonuçlar vermiştir. İşlem sonucu 0.00006'lık bir hata meydana gelmiş ve durum çok sayıda kullanıcıyı etkilemiştir. Piyasada bulunan çok sayıda yonga nedeniyle Intel şikayette bulunan tüm kullanıcıların yongasını değiştirme kararı almış ve bu durum Intel firmasına tam olarak 475 milyon dolarlık bir zarara sebep olmuştur [16].

- *Arienne-5 Füze Facias:* NASA, 4 Haziran 1996'da fırlatılması planlanan Arienne 5 uzay aracını kodlarken, Arienne 4 roketinin kodlarını kopyalayarak bir hata yaptığının farkında değildi. O gün fırlatma için geri sayım yapıldı ve roketin motorları ateşlenerek kalkış başladı. Hızlanarak yoluna 37 saniye boyunca devam eden Arienne 5 roketi o saniyeden sonra yanlış yöne doğru 90 derece dönmeye başladı. Bu durum, roketin

kendini imha etme mekanizmasını tetikledi ve uzay aracı, dünyanın en pahalı havai fişeği olarak akıllara kazındı. Bu kaza, NASA'ya aşağı yukarı 370 milyon dolara mal oldu. Tarihteki en pahalı yazılım hatalarından biri arasına giren bu kazanın sebebi, yazılımda oluşan bir bugdı. Bu bug, milyarlarca potansiyel değeri temsil edebilen 64 bit değişkeni, yaklaşık 65 bin değer alabilen 16 bit değişkene sığdırmaya çalışmasına sebep oldu. Roketler hızlandıkça kodlarda ortaya çıkan bu bug işleri sarpa sardırdı ve roket patladı [17].

- *Honda Yazılım Hatası:* 2011 yılında Honda'ya ait araçlarda hatalı yazılım sonucu bazı özellikler yanlış zamanda aktif edilmiştir. Yazılımda bulunan hataların, aracın kontrol modülü ile diğer parçalar arasındaki çalışma zamano uyumsuzluğundan oluşabileceği açıklanmıştır. Bu durumun geri görüş kamerasının çalışmasını engellemesi, sinyallerin yanlış yanması veya sileceklerin yanlış zamanlarda çalışması gibi sorunlara neden olabileceği belirtilmiş, bunun üzerine Honda firması, 2.5 milyon aracını geri çağırmak zorunda kalmıştır [18].

- *Boeing Yazılım Hatası:* Dünyanın en büyük sivil uçak üreticilerinden biri olan ve önceki yıllarda ölümcül kazalarla adını duyuran Boeing firması, 2020 yılı içerisinde Boeing 737 Max tipi yolcu uçaklarında iki farklı yazılımsal hatanın keşfedildiğini duyurdu. 2018 yılının Ekim ayında Endonezya'da, 2019 yılının Mart ayında ise Etiyopya'da yüzlerce kişinin hayatını kaybetmesine sebep olan 737 Max tipi uçakların uçuşu, Etiyopya'daki kazanın ardından birçok ülkede geçici olarak durdurulmuştu. Boeing firması, bu uçakların, uçuşlara yeniden başlaması için gerekli çalışmaları hızla sürdürürken yaptığı açıklamalarda, 737 Max tipi uçaklarda iki farklı yazılımsal hatanın daha fark edildiği yönündedir. Bu hatalardan birinin, bilgisayarın uçuşunu kontrol eden mikroişlemcideki varsayımsal hatalarla ilişkili olduğu açıklandı. Hata sonucu, uçağın pilotun kontrolünü kaybetmesine neden olabileceği belirtildi. Bunun yanında başka bir hatanın otomatik pilot işlevinin devre dışı bırakılmasına neden olabileceği belirtildi.

Tarihteki yazılım hatalarının ciddi sorunlara yol açması dikkate alınırsa yazılım maliyetini azaltmak için yazılım testinin, geliştirme sürecinin ilk aşamalarına entegre edilmesi gerekir. Test sürecinin projeye erken aşamalarda entegre edilmesi maliyeti düşürecek ve kusurları bulma şansını artıracaktır.

Yazılım test yaşam döngüsü ile sistemde var olan hataların belirlenmesi ve çözümlenmesi gerçekleştirilir. Kullanıcıya sunulmadan önce yoğunlaştırılan test aşaması yardımıyla sistem, minimum hata oranına sahip hale getirilir [9]. Bu aşamada esas hedef, hataların neden olabileceği maddi veya manevi zararların boyutunu en aza indirgemek, hatta mümkünse ortadan kaldırmaktır. Yazılım testi, uzun süreli kalite ve kullanım sağlar. Yazılım geliştirmenin erken aşamalarında fark edilen hataların düzeltilmesi zaman ve maliyet açısından daha avantajlıdır. Projenin modül ve kapsamları genişletilmeden fark edilen hatalar düzeltilmelidir. Erken fark edilen ve düzeltilen hataların, proje kapsamının

genişlemesinin ve modüllerin artmasının ardından düzeltilmesi yüksek maliyet ve uzun zaman gerektirir. Bu nedenle testlerin mümkün olunan en erken aşamada gerçekleştirilmesi tavsiye edilir.

Yazılımda hata, test ve kalite ilişkisi şu şekilde ifade edilebilir: Kaliteli yazılım minimum hata oranına sahip olan yazılımdır. Yazılım sisteminin minimum hata oranına sahip olması ise sağlıklı bir şekilde gerçekleştirilen test yaşam döngüsü ile mümkündür.

Yazılım hatası, sistemin işlevini yerine getirmesine engel olan her türlü durum şeklinde tanımlanabilir [19]. Yazılım hataları, IEEE Standard Classification for Software Anomalies sınıflandırılması için hazırlanan sözlükte yazılımın çalışmasını etkileyen kusur veya eksiklikler olarak tanımlanmıştır. Bu standarda göre, hazırlanabilecek bir hata raporunda yer alması muhtemel olan hatalar ve tanımları şu şekildedir:

- *Hata (Error)*: Geliştirici kaynaklı aksaklıklar nedeniyle sistemin beklenen çıktıyı vermemesi durumudur.
- *Arıza(Fault)*: Bir yazılım uygulamasında bulunan yanlış bir adım, işlem veya veri tanımıdır.
- *Başarısızlık(Failure)*: Bir yazılımda var olan hata nedeniyle, yanlış, eksik veya beklenenden farklı çıktının sonuç olarak üretilmesidir. Hata nedeniyle üretilen yanlış sonuçtur.
- *Sorun(Problem)*: Yazılım sisteminde bulunan, çalışma esnasında oluşan ve çözülmesi gereken durumlardır.

2.1. Yazılımda Hata Çözüm Süreci

Yazılımda hata sistemin işleyişini engelleyen ya da yanlış işleyişe neden olabilecek sorunlar olarak tanımlanır. Bu sorunların çözümü, sistemin sağlıklı ve kaliteli bir şekilde çalışması için büyük önem taşır. Sorunların çözülebilmesi için gereken işlemler bütününe hata süreci ya da hata yönetim süreci adı verilir. Hata yönetim süreci, yalnızca sistematik bir şekilde yürütüldüğü ve raporlandığı takdirde başarılı sonuçlanır [7]. Bir projede hata yönetim süreci, hataların belirlenmesi, sınıflandırılması, düzeltilmesi, doğrulanması, kapatılması ve raporlanmasını kapsar. Hatanın derecesi ve yazılımın büyüklüğü ne olursa olsun sistemde var olan bütün aksaklıklar büyük bir özenle çözüme kavuşturulmalıdır. Var olan hatalar, projenin yapım aşamasında bulunan maliyet ve zaman kriterlerini olumsuz etkileyebileceği gibi kalitesini de düşürür. Kısacası hataların önem derecelerine ve etkilerine göre kategorize edilip, çözümlenmesi gerekmektedir. Hatalar tanımlanıp, sınıflandırıldıktan sonra uygun yöntemlerle çözümlenmelidir. Hata yönetim süreci olarak tanımlanan bu aşamada, hataların sınıflandırılmaları ve derecelendirilmeleri büyük önem taşır. Yazılımda hata çözüm sürecine ait adımlar Şekil 2.1.'de verilmiştir:

Şekil 2.1. Yazılım hata çözüm süreci adımları

- *Hatanın Tespit Edilmesi:*Geliştirilmiş yazılım veya programlardaki hata tespitinde önemli olan nokta, yazılımların son kullanıcıya teslim edilmeden önce hatanın fark edilmesidir. Yazılım hatalarının yazılım geliştirme yaşam döngüsünün erken aşamalarında tespit edilmesi veya ortaya çıkarılması çok önemlidir. Hataların erken giderilmesi, sistem için belirlenen zaman ve maliyetin daha asgari seviyede olmasını sağlayacaktır.

- *Hatanın Sınıflandırılması:* Hatanın sınıflandırılması ile yazılım geliştiricilerin görevleri, öncelik sıralamasına göre belirlenir. Böylelikle birden fazla hatanın ortaya çıkması durumunda kritik boyuttaki hataların çözümlenmesi öncelik kazanır.

- *Hatanın Çözümlenmesi:* Hata çözümleme adımları, kusurlar hakkında geliştiricilerin bilgilendirilmesi ile başlar. Daha sonra geliştiriciler, hata sınıflandırılması ile öncelikli olarak düzeltilecek hataları planlar ve çözümü için gerekli çalışmaları başlatırlar. Son olarak test yöneticisine, geliştiriciler tarafından çözümlenen hataya ait bir çözüm raporu sunulur.

- *Hata Çözümünün Doğrulanması:* Geliştiricilerin, hatanın çözümünü test ekibine bildirmesini kapsayan aşamadır. Test ekibi tarafından test işlemi tekrar gerçekleştirilir, hatalı kısım incelenir ve çözümün yeni hatalara sebep olmadığı belirtilir.

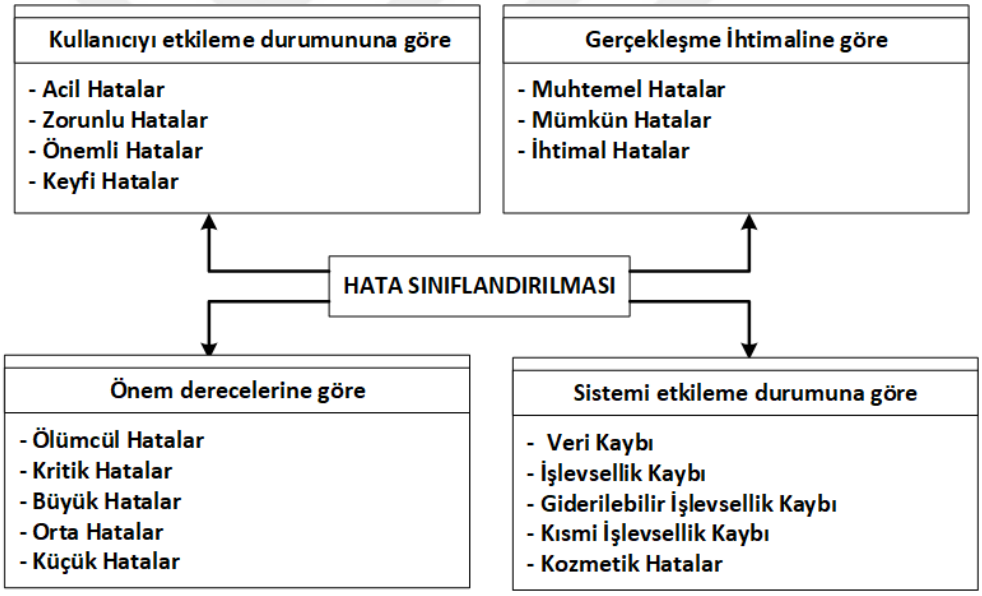
- *Hatanın Kapatılması:* Hatanın kapatılması aşaması, hatanın çözümünden sonra gerçekleştirilir. Çözülen ve çözümü doğrulanan hatanın durumunun kapalı olması, hatanın çözümlendiğini belirtir. Hatanın aktiflik durumunun kapatılması, tekrar kontrol edilmesini önlemek için gereklidir.

- *Hata ve Çözümünün Raporlanması:* Hatanın ve çözümün raporlanması aşamasında rapor içeriği, hata yönetim süreci adımları, hatanın durumu ve sınıflandırılması, hatanın çözümlenmesi ve doğrulanması hakkında detaylı bilgileri barındırır. Test yöneticileri tarafından hazırlanan hata raporu, yönetim ekibine gönderilir. Hata

yönetim süreci ve çözümün doğrulanması için yönetim ekibinden geri bildirim alınır. Yönetim ekibinin gerek gördüğü aşamalarda hatanın ortadan kalkması için daha fazla destek sağlanır. Hatanın çözümü raporda, sonraki adımlarda hatanın yeniden ortaya çıkma ihtimaline karşı detaylıca belirtilmelidir. Ayrıca hata çözümünün, sistemin diğer modüllerini etkileme ihtimali karşısında da hatanın raporlandırılmasından faydalanılır.

2.2. Yazılım Hatalarının Sınıflandırılması

Yazılım hatalarının çözümlenmesi için hata yönetim sürecinde hatanın tespitinden sonraki aşama, hatanın sınıflandırılmasıdır. Hata sınıflandırılması, keşfedilen hatanın sistem ve kullanıcılar üzerindeki etkisine ve bu etkinin derecesine göre gerçekleştirilir. Tespit edilen hataların büyüklüğü, önem derecesi ve risklerinin göz önüne alınması ile hataların çözümleme aşamasındaki öncelik sırası belirlenir. Şekil 2.2.'de yazılım hatalarının sınıflandırılması görsel olarak verilmiştir:



Şekil 2.2. Yazılım hatalarının sınıflandırılması

Yazılımda var olan ve fark edilen hatanın sınıflandırılması ile öncelikli hatalar belirlenir. Bu hataların çözümü büyük önem taşımaktadır. Öncelikli hataların çözümlenmesi ile sistemin çalışmasında meydana gelebilecek kesintilerin önüne geçilebilir. Sınıflandırma sonucu sistemi etkileme durumuna göre kategorize edilen hataların çözümlenmemesi veya göz ardı edilmesi durumunda veri kaybı veya işlevsellik kaybı gibi durumlar söz konusudur. Bu durumlar, yazılım sisteminin kalitesini ve işlevselliğini olumsuz etkiler.

2.2.1. Sistemi Etkileme Durumuna Göre Hatalar

Yazılım geliştirilirken sistemde var olabilecek çeşitli hatalar, geliştiriciden, donanımdan, yanlış algoritmadan ya da hatalı kodlamadan kaynaklanabilir. Bu hataların bazıları, doğrudan sistemin çalışmasını etkiler. Sistemin çalışmasını, işlevselliğini ya da çalışma esnasında dönen verileri etkileyen hatalar, sistemi etkileme durumuna göre sınıflandırılırlar.

- *Veri kaybı:* Hatanın, sistemin veri tabanına erişebildiği ve veri tabanında bulunan verileri sildiği hata çeşididir.
- *İşlevsellik kaybı:* İşlevsellik kaybı, sistemde bulunan hatanın sistemin işlevselliğini bloke ettiği durumları ifade eder.
- *Giderilebilir işlevsellik kaybı:* Hatanın, sistemde bulunan herhangi bir işlevi bloke ettiği durumları kapsar. Bu tür hatalarda meydana gelen bloke durumu, başka bir işlevsel metot kullanılarak giderilebilir ve bu hata türleri sistemin çalışmasını aksatmaz.
- *Kısmi işlevsellik kaybı:* Sistemin, önemli olmayan işlevlerinin bloke edildiği hata türüdür.
- *Kozmetik hatalar:* Kullanıcı isteklerine göre tasarlanan arayüz ve grafiklerin, kullanıcı kriterlerine uygun olmayan görüntü veya uyarı mesajları içermesi durumudur. Bu tür hatalar, sistemin işlevselliğini etkilemez.

2.2.2. Kullanıcıyı Etkileme Durumuna Göre Hatalar

Geliştirilen yazılım sistemlerinde oluşabilecek hataların sonuçları, sistemi olduğu kadar kullanıcıyı da etkiler. Bu sınıflandırmada hatalar, olumsuz sonuçlarının minimum olabilmesi için çözümlenme önceliklerine göre kategorize edilir.

- *Acil hatalar:* Sistemin çalışmasını durduran veya çalışmayı yanlış sonuçlandıran, performansı büyük ölçüde etkileyen ve düzeltilmesi aciliyet gerektiren hatalardır. Bu tür hatalar, büyük oranda maddi ve manevi zarara yol açabilir.
- *Zorunlu hatalar:* Bu tür hatalar, test aşamasında fark edilir. Bu kategorideki hataların çözümlenmeden sistemin kullanıcıya teslim edilmemesi gerekir.
- *Önemli hatalar:* Sistemin, yazılım yaşam döngüsünün planlama aşamasında belirlenen kriterlere uymaması sonucu kullanıcı tepkilerine neden olabilecek hatalar ve kusurlardır.
- *Keyfi hatalar:* Yeterli zaman kalması durumunda düzeltilmesi sisteme olumlu etki eden hatalardır. Ayrıca daha önce çözümlenen bir hata modülünün sonraki sürümlerde tekrar ortaya çıkması durumu da bu kategoride yer alır.

2.2.3. Önem Derecelerine Göre Hatalar

Yazılım sisteminde var olan hataların tümünün, çözümlenmeye gerek duyulacak kadar büyük önem taşıdığı söylenebilir. Ancak bazı hataların çözümlenmemesi, sonuçları itibariyle daha büyük olumsuzluklara neden olur. Bu kategoride hatalar, doğurabileceği olumsuzluklara göre sınıflandırılır. Sınıflandırılmaya göre hataların önem dereceleri, sonucunda meydana gelebilecek zararın büyüklüğü göz önüne alınarak hatanın öncelikli çözülme sırası hakkında bilgi verir.

- *Ölümcül hatalar:* Yazılımın işlevselliğinde meydana getirdiği kesinti nedeniyle testlerin devamını engelleyen hatalardır. Bu hatalar, çözümlenemediği takdirde büyük ölçüde maddi ve manevi zarara neden olabilir.
- *Kritik hatalar:* Bu tür hataların varlığında testler aksamaya uğramazlar. Ancak bu hatalar, ileri aşamalarda sistemi kesintiye uğratabilir. Bu hatalar çözümlenmeden sistemin, kullanıcıya teslim edilmemesi önerilir.
- *Büyük hatalar:* Bu tür hatalar ürünün kullanıcıya sunulmasına engel teşkil etmez. Ancak sistemin ileri aşamalarında bu hatadan kaynaklanan zararın telafisi ve buna bağlı olarak maliyeti zor olabilir.
- *Orta hatalar:* Bu hatalar, test işleminin devamını etkilemez. Sistem bu şekilde kullanıcıya sunulabilir ve bulunan hatalar sonucu meydana gelen zararların telafisi mümkündür.
- *Küçük hatalar:* Sistemin çalışmasını, test sürecini önemli derecede etkilemeyecek olan bu hataların varlığında, sistem kullanıcıya bu şekilde sunulabilir.

2.2.4. Gerçekleşme İhtimaline Göre Hatalar

Gerçekleştirilen yazılım sistemleri içerisinde pek çok potansiyel hatalar bulunur. Bu hatalar, sistemin ileri aşamalardaki işleyişinde ortaya çıkabilir ve sistemin çalışmasını etkileyebilir. Bu tür hataların sistemin kullanıma sunulmadan önce fark edilmesi ve çözümlenmesi, sistem kalitesi açısından büyük önem taşır. Bu kategoride sistemin derleme ya da çalışma esnasında ortaya çıkabilecek olası hataların ve sonuçlarının sınıflandırılması yapılmıştır.

- *Muhtemel hatalar:* Sistem işleyişini aksatacağına, kullanıcıya sunulan çıktı ile beklenen çıktının farklı olacağına veya sistemi bloke edeceğine kesin gözüyle bakılan hatalardır. Bu hataların düzeltilmemesi durumunda sistem işleyişini sağlıklı bir şekilde yerine getiremez. Böylelikle sistem bu hatalar çözümlenmeden kullanıcıya teslim edilemez.
- *Mümkün hatalar:* Bu tür hataların, sistem işleyişini etkilemeleri olasıdır. İşleyişte aksaklığa neden olabilecekleri gibi, herhangi bir kesintiye neden olmayabilirler. Kaliteli

yazılım, hata oranının en az olduğu yazılımdır. Bundan dolayı, bu tür hataların çözümlenmesi yazılım kalitesini olumlu etkiler.

- *İhtimal hatalar:* Sistemi etkileme ihtimali düşük olan hatalardır. Yazılımın işlevselliğini etkilemezler. Ancak her hata gibi bu hatalarında çözümlenmesi yazılımın kalitesini arttırır.

Tablo 2.1.'de kullanılan kısaltmalar (SE-Sisteme Etkisi, ME-Müşteriye Etkisi, ÖD-Önem Derecesi), yukarıda açıklamaları verilen hata türlerine göre kategorize edilmiştir.

Tablo 2.1. Geçmişte Yaşanmış Yazılım Hatalarının Sınıflandırılması (* SE-Sisteme etkisi, ME-Müşteriye etkisi, ÖD-Önem derecesi)

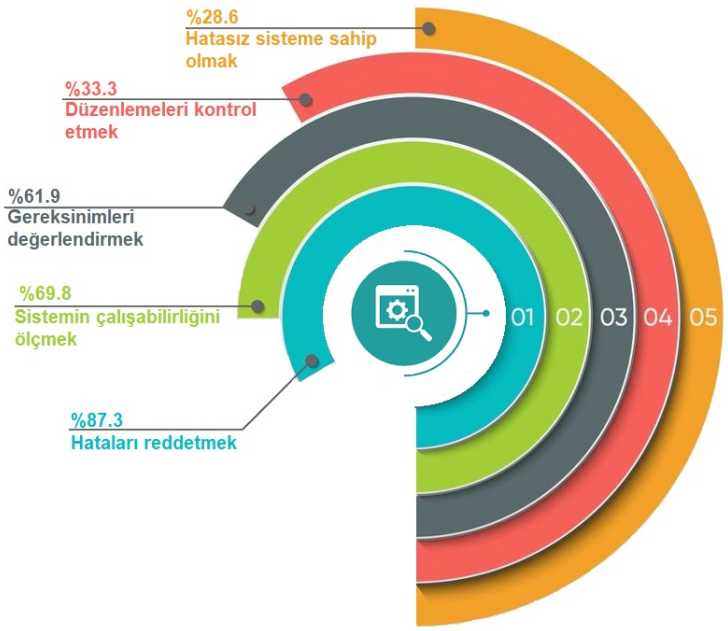
Ref.	Hatanın adı	Tarih	Hatanın nedeni	Hatanın türü	Sonuç
[20, 21]	Mariner-1 Uzay Roketi	22 Temmuz 1962	Kağıda doğru yazılan formülün bilgisayara yanlış yazılması	SE-İşlevsellik Kaybı, ME-Acil Hatalar, ÖD-Ölümcül Hatalar	Yörüngeden ayrılan roket yok edilmiş, NASA'ya 80 milyon dolar kayıp olmuştur.
[22, 23, 24]	Hartford Colesium Binası	1978	Çoklu değişkene sahip matematiksel formülün hesaplanmasındaki işlem hatası	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	CAD yazılımı ile tasarlanan Hartford Colesium binası çökmüş, 90 milyon dolar olan bir zarara neden olmuştur.
[21, 25, 26]	Sovyet Gaz Hattı	1982	Programdaki güvenlik açığı	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	CIA ajanlarının geliştirdiği böcek, tarihin en büyük nükleer olmayan patlaması gerçekleşmiştir.
[20, 21]	Therac-25 Tedavi Cihazı	1985-1987	Hastalarda kullanılan radyasyon ışınlarını dengeleyen plakaların yazılım kaynaklı devreye girmemesi	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	6 hastadan 3'ü aşırı radyasyona maruz kalması nedeniyle hayatını kaybetmiştir.
[21, 27, 28]	Berkeley Unix Sisteminde Tampon Bellek Taşması	1988	Tampon bellek yetersizliği	SE: İşlevsellik Kaybı, ME: Zorunlu Hatalar, ÖD: Kritik Hatalar	Bir grup siyah şapkalı bilgisayar korsanı tarafından 2000-6000 arası bilgisayara sızılmıştır.
[15, 29]	Patriot Füze Hatası	25 Şubat 1991	Zaman hesaplamasında kullanılan 24 bitlik sabit noktalı değişkende bulunan hata	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	Hata nedeniyle 29 Amerikan askeri hayatını kaybetmiştir.
[16, 20, 30]	Pentium İşlemci Sorunu	1993	Tam sayı aralığında bir değere sahip olması gereken matematiksel formülün yanlış yazılması	SE: İşlevsellik Kaybı, ME: Zorunlu Hatalar, ÖD: Kritik Hatalar	Firma, 475 milyon dolar zarara uğramıştır.
[21, 28]	Kerberos Rastgele Sayı Üretme Hatası	1988-1996	Programında kullanılan rastgele sayı üreticisinin uygun bir parametre ile beslenmemesi	SE: İşlevsellik Kaybı, ME: Zorunlu Hatalar, ÖE: Kritik Hatalar	Kerberos firmasına ait yetkili bilgisayarlardan bir kaçına izinsiz giriş sağlanmıştır.
[21, 30]	Bilgisayar Ağı Kontrol Komutları Hatası	1995-1996	IP numaralarının yanlış yazılması ve doğru yazılmayan bir "Ping" komutunun internet üzerinde herhangi bir yerden gönderilmesi	SE: İşlevsellik Kaybı, ME: Zorunlu Hatalar, ÖD: Kritik Hatalar	Bir çok bilgisayar mavi ekran hatası vermiştir. En çok etkilenen işletim sistemi Windows'tur.
[17, 21, 31]	Arienne-5 Füze Faciası	4 Haziran 1996	Aritmetiksel işlemlerde 64 bit kullanılması gerekirken, önceki sürümden kalan 16 bitlik kullanımın devam etmesi	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	Fırlatıldıktan 40 saniye sonra havada infilak eden füze, 500 milyon dolarlık bir zarara neden olarak en pahalı yazılım hataları arasında yer almıştır.

[29, 30, 31, 32]	Mars Climate Orbiter Hatası	23 Eylül 1999	Uzunluk ölçümlerinde kullanılan İngiliz ölçü birimleri ile metrik değerleri arasındaki uyumsuzluk	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Hatanın NASA'ya maliyeti 125 milyon dolardır.
[21, 28]	National Cancer Institue	2000	Yazılım kaynaklı hata nedeniyle hastalara yanlış dozda radyasyon verilmesi	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Ölümcül Hatalar	8 hasta hayatını kaybetmiştir ve bilinen 20'den fazla kişi ciddi sağlık sorunları yaşamıştır.
[20, 28, 29, 33]	Mercy Hastanesi Ölüm İlanları	2002	Hastane sistemlerindeki veri tabanı kayıtlarının düzensizliği	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Veri tabanında bulunan hasta kayıtlarındaki 8500 kişi ölü ilan edilmiştir.
[20, 29]	Michigan Erken Tahliye Hatası	Ekim 2005	Veri tabanında bulunan verilerin hatalı sorgulanması	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Michigan Düzeltme Bakanlığı'na ait bilgisayar programlarındaki aksaklık nedeniyle 239 mahkum, 39 ile 161 gün öncesinden tahliye edilmiştir.
[18, 20, 28]	Honda Yazılım Hatası	2011	Hatalı algoritma sonucu araç özelliklerinin yanlış zamanlarda aktif edilmesi	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Firma 2.5 milyon aracını geri çağırmak zorunda kalmıştır.
[30, 34, 35]	Apple Maps Yön Hatası	2012	Farklı veri tabanlarındaki bilgilerin entegrasyonunda oluşan karmaşıklık	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Büyük Hatalar	Apple ile Google Maps arasındaki harita(maps uygulaması) rekabet sonucu firmanın itibar kaybetmiştir.
[36, 37, 38]	Knight Capital Group Finans Hatası	2012	Yanlış algoritma kullanılması	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Firma, 30 dakika gibi kısa bir sürede 440 milyon dolar maddi kayıp yaşamıştır.
[21, 28]	Yanlış Jüri Daveti	2012	Veri tabanında gerçekleşen hatalı sorgulama	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Büyük Hatalar	Kaliforniya'da gerçekleşen olay sonucu, 1200 kişiye jüri davetiyesi gönderilmiş, tüm eyaletler arası yollarda trafik nedeniyle saatlerce aksaklıklar yaşanmıştır.
[39]	iCloud Güvenlik Açığı	Eylül 2014	Yazılım sisteminde var olan güvenlik açığı	SE: Veri Kaybı, ME: Zorunlu Hatalar, ÖD: Büyük Hatalar	Güvenlik açığı nedeniyle ünlüler dahil olmak pek çok insanın kişisel bilgileri ele geçirilmiş, bu olay üzerine Apple CEO'su Tim COOK, kullanıcı güvenini tazelemek için iCloud güvenlik özelliklerini güçlendirme sözü vermiştir.
[28, 40, 41]	Toyota Prius Aracı	2015	Motor kontrol ünitesindeki bir yazılımın aşırı ısınma ve güç kaybına neden olması	SE: İşlevsellik Kaybı, ME: Acil Hatalar, ÖD: Kritik Hatalar	Firma, 2014 yılında 2 milyon, 2015 yılında ise 625 bin aracını geri çağırmış, bu hata nedeniyle aşırı ısınan araçlar trafik sorunlarına neden olmuştur.

[42]	Google+ Güvenlik Açığı	2015-2018	Yazılım sisteminde var olan bir güvenlik açığı	SE: Veri Kaybı, ME: Zorunlu Hatalar, ÖD: Büyük Hatalar	Yer alan güvenlik açığı nedeniyle yaklaşık 500.000 Google+ kullanıcısının kişisel bilgileri izin olarak ele geçirildi. 2018'de farkedilen hata düzeltildi.
[43]	Ethereum 300 Milyon Dolarlık Hata	2017	Yazılım sisteminde var olan güvenlik açığı	SE: Veri Kaybı, ME: Zorunlu Hatalar, ÖD: Büyük Hatalar	Yer alan güvenlik açığı, fonların bir süreliğine kilitlenmesine ve 300 Milyon $'lık kripto paranın kaybına neden olmuştur.
[44]	Tesla Otomatik Frenleme Yazılım Hatası	23 Ekim 2021	Kendi kendine sürüş Beta yazılım sürümünde otomatik fren sisteminin zamansız aktifleşmesi durumunda arkadan çarpma ve araçtaki kişilerin yaralanmasına ihtimalleri	SE: Giderilebilir İşlevsellik Kaybı, ME: Zorunlu Hatalar, ÖD: Kritik Hatalar	Firma, bu hatadan etkilenen araçları geri çağırmak için Güvenlik Geri Çağırma Raporu yayınlamıştır. Aralarında Model S, Model Y, Model X ve Model 3'ünde bulunduğu 12.000'e yakın aracı geri çağırmıştır.

3. Yazılım Test Yaşam Döngüsü Ve Süreçler

Yazılım yaşam döngüsü sürecinde belirlenen kriterler göz önüne alınarak geliştirilen her modül, sistemin önemli bir parçasını oluşturur [13]. Geliştirilen yazılımlar ve güncellenen sürümlerde ortaya çıkan her hata, yazılımın kalitesini ve işlevselliğini olumsuz etkiler [45]. Yazılım yaşam döngüsü boyunca geliştirilen sistemin, kalitesini arttırmak için test yaşam döngüsü büyük önem taşımaktadır. Test yaşam döngüsü ile sistemde var olan hatalar tespit edilir. Gerçekleştirilen test işlemi daha sonra bu hataların bulunduğu modüller, nedenleri ve çözümleme yöntemleri raporlanarak sistemin ileri aşamalarında hataların tekrar etmesi durumunda kullanılır. Ayrıca test yaşam döngüsünün yalnızca sistemin hatalardan arınması için gerçekleştirilmediği, ISTQB tarafından 2013-2014 yılında yayınlanan raporda belirtilmiştir. Rapora göre, test işlemi hataların çözümlenmesinin yanı sıra sistem çalışabilirliğini kontrol etmek, gereksinimleri değerlendirmek, güncellemeleri ve son sürümün hata oranını kontrol etmek için de gerçekleştirildiği ifade edilmiştir [1]. ISTQB tarafından yayınlanan raporda, yazılım testinde önemsenen kriterler Şekil 3.1.'de gösterilmiştir.



Şekil 3.1. Yazılım testinin diğer amaçları [1]

Yazılım testi; "Ürün kullanıcılarının ihtiyaçlarını karşılama oranını belirlemek için yapılan bir hata tespit işlemi" olduğu söylenebilir. Test sürecinin en uygun şekilde gerçekleştirilebilmesi ve iyileştirilmesi için test uzmanları aşağıdaki önemli hususlara dikkat etmelidirler.

- Uygun test tekniklerine göre test seviyeleri ve test türlerinin belirlenmelidir.
- Hatanın bulunduğu kod blokları tespit edilmeli ve işaretlenmelidir.

- Tespit edilen hata, önem derecesi ve aciliyetine göre sınıflandırılmalıdır.
- Hatanın çözümlenebilmesi için öncelik durumu dikkate alınmalı, kullanılacak yöntemler doğru bir şekilde belirlenmelidir.
- Test süreçlerinde en uygun yazılım araçları belirlenmeli ve gerekli aşamalarda etkili kullanılabilmelidir.

3.1. Doğrulama ve Geçerleme

Doğrulama ve geçerleme kavramları yazılım test yaşam döngüsünün temel iki kavramıdır. Belirli kriterler altında gerçekleştirilen yazılım sistemlerinin kullanıcı isteklerini karşılaması ve belirlenen hedefler çerçevesinde geliştirilmesi sonucu bu iki kavram ortaya çıkmıştır. Birbirinden farklı olmasına rağmen benzerlik içeren bu iki kavram şöyle açıklanabilir:

- **Doğrulama(Verification):** Yazılım geliştirme sürecinde belirlenen kriterlerin sağlanıp sağlanmadığını kontrol etme aşamasıdır. Doğrulama süreci, yazılımın kullanıcı kriterleri dikkate alınarak hazırlanan raporlardaki hedeflere yönelik geliştirilmesini amaçlar. Doğrulama aşamasında temel amaç, kullanıcı taleplerinin karşılanmasını sağlamaktır. Bu aşamada, geliştirilen ürünün doğru olup olmadığını belirlemek için belgeler, tasarım ve kod yapısı incelenir. Bu aşamada temel soru şudur: *'Ürün doğru mu geliştiriliyor?'*
- **Geçerleme(Validation):** Geliştirilen sistemin kullanıcı ihtiyaçlarını tam olarak karşılayıp karşılayamadığının kontrol edildiği süreçtir. Yazılımın istenen kullanım şartlarını yerine getirmesini sağlamaya yardımcı olur. Bu aşamada temel soru şudur: *'Kullanıcı gereksinimlerinin tamamı karşılanmış mı? Doğru ürün mü geliştiriliyor?'*

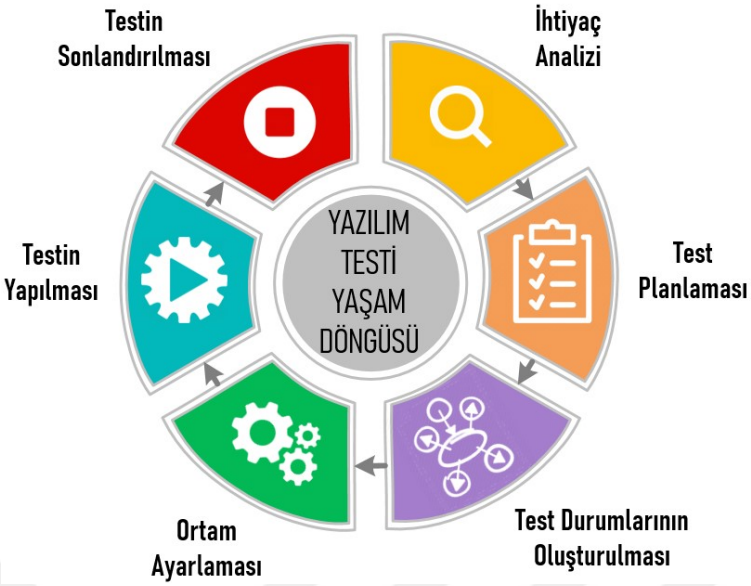
3.2. Yazılım Test Yaşam Döngüsü

Test işlemleri, yazılım geliştirme yaşam döngüsünün her aşamasında aslında gerçekleştirilmektedir. Her aşamada birbirinden farklı çok sayıda test işlemleri ve süreçleri yer almaktadır. *Planlama* sürecinde teknik personellerin sayısı ve teknik yeterliliklerinin doğrulanması, *Çözümleme* sürecinde geliştirilen UML diyagramlarının doğrulanması, *Tasarım* sürecinde gerçekleştirilen fiziksel arayüzlerin doğrulanması, *Kodlama* sürecinde proje kodlarının test edilmesi ve doğrulanması birer örnek olarak gösterilebilir. Ayrıca, yazılım geliştirme süreci tamamlandıktan sonra da çok sayıda farklı test işlemlerinin yapıldığı test stratejilerinin uygulandığı süreçler bulunmaktadır. Her bir süreç için büyük önem taşıyan test yaşam döngüsü ile birçok önemli hususlar analiz edilir. Test yaşam döngüsü ile ortaya çıkarılan bu önemli hususlardan bazıları aşağıda sıralanmıştır:

- Yazılım test yaşam döngüsü süreci ile sistemin hata oranı belirlenir ve buna bağlı olarak yazılımın kaliteli olmasına engel olan her durum ortadan kaldırılır.

- Test sonuçlarına göre sistem ile ilgili bilgi, görüş ve önerilerde bulunulur.
- Yazılımın kriterlere uygunluğu, performansı, doğruluğu ve geçerliliği ölçülür.
- Sistemde bulunan ve hata riski barındıran modüller, test edilip raporlanır. Sonraki aşamalarda hata riski barındıran modüller tekrar test edilerek olası hataları ortadan kaldırılır. Bu nedenle testler, yazılımın kalitesini arttırmada önemli rol oynar.
- Müşteri memnuniyetini arttırmak için sistem, doğrulama ve geçerleme işlemine tabi tutulur.
- Doğru test tekniğinin uygulanması sonucunda sistemle ilgili her türlü eksiklik veya kusurlar belirlenir.
- Sistemin planlanan zaman, maliyet ve kriterler doğrultusunda ilerleme düzeyi kontrol edilir.
- Sistemde var olan bütün yollar kontrol edilerek sistemin doğruluğu ortaya konulur.
- Sistemin beklenilen hız, alan gibi özellikleri kontrol edilerek daha performanslı çalışması hedeflenir.
- Sistemde var olan güvenlik zaafiyetleri tespit edilerek, sistem açıkları giderilir.
- Sistemin kullanıcı dostu olması açısından kullanılabilirlik testleri uygulanır.
- Sistemin olumsuz durumlardaki tavrı kontrol edilerek, bu tür durumlar için gerekli çözümler üretilir.
- Sistemin farklı cihazlarda veya işletim sistemlerinde kusursuz çalışması için gereken testler yapılır.

Yazılım testleri, yazılımın geliştirme yaşam döngüsünün her sürecinde yer alan bir döngüdür. Geliştirilen sistemler için oluşturulan yaşam döngüsünün büyük çoğunluğunun test işlemlerine verilmesi hataların daha erken aşamalarda fark edilmesi, daha hızlı çözümlenmesi ve buna bağlı olarak zaman-maliyet kavramlarının daha verimli kullanılması demektir [4]. Yazılım test işlemleri de belirli bir sürece yayılarak gerçekleştirilmeli, raporlanmalı ve sonuçlandırılmalıdır. Yazılım test yaşam döngüsü aşamaları Şekil 3.2.'de gösterilmiştir.



Şekil 3.2. Yazılım test yaşam döngüsü

Yazılımın kalitesi açısından önemli olan test işlemleri, manuel ve otomatik olarak iki şekilde gerçekleştirilebilir. Manuel test süreci, hataların bulunması için test uzmanlarının test senaryolarını çalıştırıp denetlemesi sürecidir. Bu süreçte, test uzmanları son kullanıcı rolü ile sistemi test eder ve herhangi bir otomasyon aracı olmaksızın raporları manuel olarak oluşturur. Bir diğer test gerçekleştirme işlemi otomatik testtir. Otomasyon testi ile hataların bulunması için bir yazılım otomasyon aracından yardım alınır. Bu süreçte otomasyon araçları ile otomatik olarak çalıştırılan test komut dosyaları sayesinde test sonuçlarına ulaşılır.

Yazılım testi, işlevsel (fonksiyonel) ve işlevsel olmayan (fonksiyonel olmayan) test olmak üzere iki ana başlık altında incelenebilir. *İşlevsel test türleri*, yazılım fonksiyonlarının gereksinimlere uygunluğunun doğrulanması için gerçekleştirilen testleri içermektedir. Test sonunda elde edilen çıktının beklenen çıktıyla eşdeğer olup olmadığı kontrol edilir. Fonksiyonların girdi-çıktı değerlerinin kontrol edildiği işlevsel testlerde uygulamanın kaynak kodu denetlenmediği için Kara Kutu testi kapsamında yer almaktadır. *İşlevsel olmayan test türleri* ise sistemin işleyişinin performans, yük, ölçeklenebilirlik, güvenlik, uyumluluk vb. açısından denetlendiği test türüdür. Esas amaç, sistemi bir talep karşısında ne kadar hızlı yanıt verdiğinin değerlendirilmesidir. Ayrıca test esnasında sisteme yöneltilen istek doğrultusunda güvenlik, yük, stres gibi özellikler de kontrol edilir [46] [47].

Yazılım testlerinin gerçekleştirilmesi sonucunda elde edilen çıkış değerleri, projelerin büyüklüğüne ve kullanım alanlarına göre farklılık gösterebilmektedir. ISTQB tarafından yayınlanan 2013-2014 yılı raporuna göre yapılan araştırmalar sonucu testin çıkış kriteri olarak %77.8'lik bir dilimle gereksinimlerin karşılanması ilk sırada yer almaktadır. İkinci sırada %50.8 oranla son teslim tarihi proje sonuçlanmasını etkileyen unsur olarak

gösterilmektedir. Bu oranları %49.2 ile risk kapsamı ve %11.1 ile bütçe kriterlerinin takip edildiği görülmektedir [1].

3.2.1. İhtiyaç Analizi

Test yaşam döngüsünün ilk aşaması olan ihtiyaç analizi, test işlemlerinin sağlıklı ve verimli bir şekilde gerçekleştirilmesi ve hedeflenen test çıktısına ulaşılması için büyük önem taşımaktadır. Test yaşam döngüsünde ihtiyaç analizi, test edilmesi gereken modüllerin ve özelliklerinin tanımlandığı aşamadır. Test işleminde hedeflenen sonuca varmak için test öncelikleri ve odak noktalar belirlenir. Test yaşam döngüsünün ilk aşamasında, Kalite Güvence ekibi tarafından, test edilecek sistem ve test için dikkate alınması gereken gereksinimler listelenir. Bu aşama, kontrol edilmesi gereken modüllerin, yapılacak testlerin türleri ve seviyelerinin analizi için gereklidir. İhtiyaç analizine göre ikinci aşama olan testin planlanması gerçekleştirilir. Analiz doğrultusunda gerçekleştirilecek test tekniği, test türü ve test stratejileri belirlenir.

3.2.2. Test Planlama

Test planlama aşaması, test yaşam döngüsüne dair tüm stratejilerin belirlendiği kısımdır. Testin amacı, kapsamı, test stratejisi ve seviyesi, riskler, test sonucunda ulaşılabilecek hedefler ve testin maliyeti bu aşamada belirlenir. Bu aşamada, test ekibi tarafından belirlenmesi gereken önemli hususlardan bazıları aşağıdaki gibidir:

- Yazılım sistemini analiz etmek,
- Testlerin amaç ve kapsamını belirlemek,
- Yapılacak test türlerinin belirlenerek, test stratejileri oluşturmak,
- Testlerin özelliklerine göre önemli kriterleri belirlemek,
- Testler için önemli ve uygun kaynakları belirlemek,
- Tüm projeyi test etmenin maliyetini hesaplamaktır.

Test tasarımı, test planlama aşaması içerisinde yer almaktadır. Test ekibinin yıllar boyunca elde etmişi oldukları tecrübeler yardımıyla, sistem risklerine uygun senaryo ve planlamaların gerçekleştirildiği aşamadır. Yani, tecrübelerin ön plana çıktığı, nitelikli uzman test personellerin aktif rol aldığı aşamadır. Bu aşamada sistem riskleri belirlenir ve bu adımla birlikte test koşulları tanımlanır. Test stratejilerinin analiz ve tasarımıyla sonraki test işlemleri hazırlanan senaryo çerçevesinde kolaylıkla yürütülebilir.

3.2.3. Test Senaryolarının Oluşturulması

Test planlamasın sonra test senaryolarının çalışması başlar. Test senaryosu, sistemin gereksinimleri karşılayıp karşılayamadığı ve doğru çalışıp çalışmadığını belirlemek için test sırasında kullanılan bir dizi işlem sıralaması olarak tanımlanabilir. Bu aşamada test ekibi

tarafından, testlerin yürütülmesi için test senaryoları ayrıntılı olarak tasarlanır. Test senaryolarının oluşturulmasında sistemi içeren her türlü husus göz önüne alınarak uygun ve birbirinden farklı yaklaşımlar hazırlanmalıdır. İyi bir test senaryosu, basit ve açık işlem adımlarını içerecek şekilde yazılmalıdır. Farklı senaryolarla gerçekleştirilen test uygulamaları ile aynı sonuca varılmalıdır. Yani, yapılan her test sonucunda elde edilecek çıktı aynı olmalıdır.

3.2.4. Test Ortamının Hazırlanması

Test ortamı, test senaryolarının yürütülmesi için hazırlanan yazılım ve donanımları içermektedir. Bu aşamada test ekibi, bütün test senaryolarını yürütebilmek için yazılım, donanım gibi sistemsel kaynakları kurmakla görevlidir. Test ortamında, ürünün test edileceği koşullara karar verilir. Birden çok test ortamının olması mümkündür. Test sunucusunun kurulumu, ağ kurulumu, bilgisayar, tablet ya da mobil cihaz kurulumu bu aşamada gerçekleştirilir. Test ortamının düzenli ve eksiksiz olması test işleminin kaliteli şekilde gerçekleşmesi ve doğru sonuçlar üretilebilmesi için önemli bir adımdır. Sağlıklı bir test ortamı için dikkat edilmesi gereken en önemli etkenler test için gerekli sistem ekipmanlarının tamamının mevcut olması, test sırasında kullanılması gereken yazılım uygulamalarının eksiksiz olması ve test sırasında kullanılacak veri setlerinin hazır olmasıdır. Kaliteli ve doğru test sonuçlarının elde edilmesi test ortamı ve test ekipmanlarının eksiksiz ve sağlıklı bir şekilde hazırlanması ile mümkün olur.

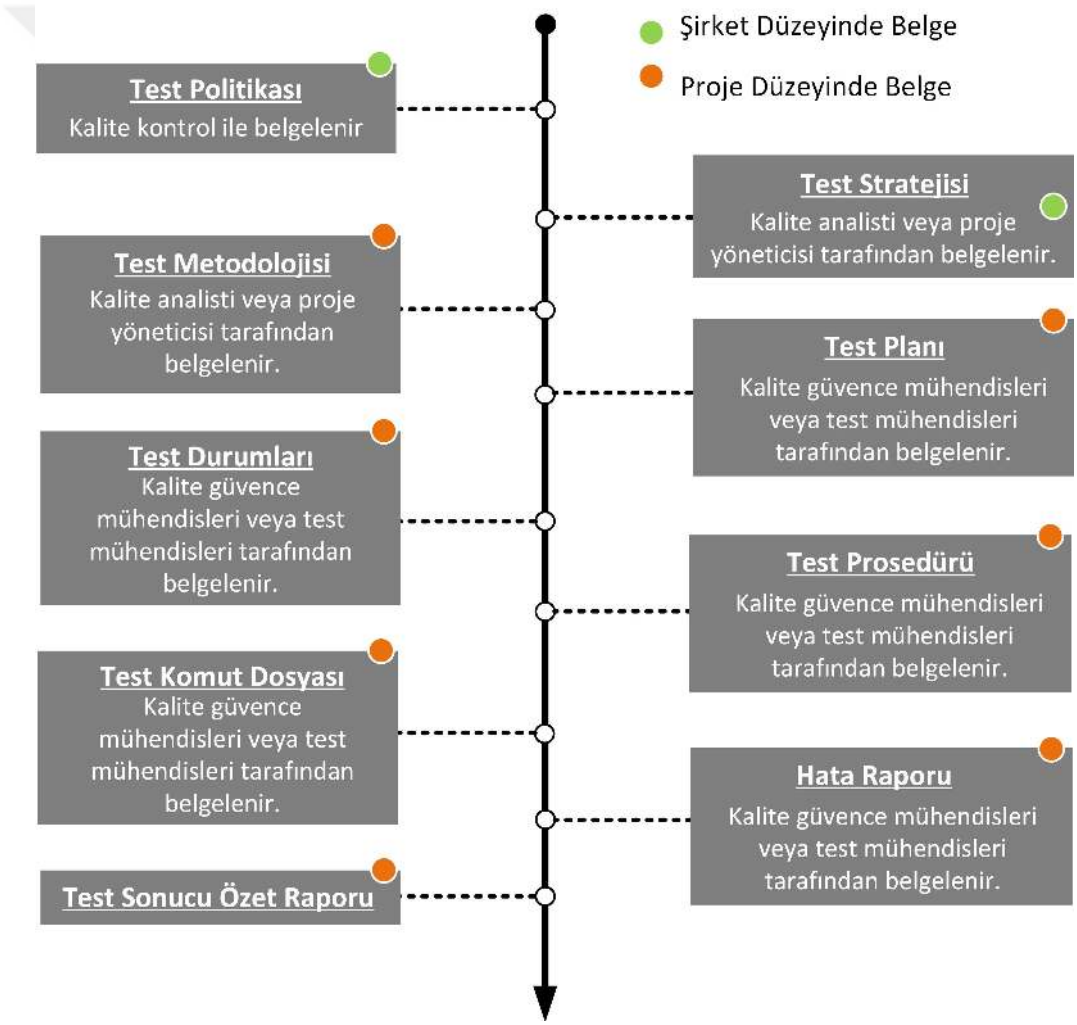
3.2.5. Testin Yürütülmesi

Bu aşamada test işlemi kod ile yürütülür. Test işlemi sonucu beklenen değer ile çıkan değer karşılaştırılır ve raporlanır [48]. Raporlanan değerlerin analizi de bu adımda gerçekleşir. Test senaryolarına ve iş-akış planlarına bağlı kalarak, manuel veya otomatik olarak sistemin muhtemel hatalarının fark edildiği aşamadır. Verimli test sonuçları için test gereksinimleri ve senaryoları dikkate alınmalı, belirlenen her sonuçlar beklenen sonuçlar ile karşılaştırılmalıdır. Her bir test durumu sonlandığında, 'Başarılı', 'Başarısız' veya 'Engellendi' olarak işaretlenir. Test senaryolarında bulunan hatalar sebebiyle engellenen belirli senaryolar 'Engellendi' olarak işaretlenir. Testlerin başarısız sonuçlanması durumunda ilgili kusurlar hata izleme sistemleri yardımıyla yazılım geliştirici ekibine bildirilir. Hatanın çözümlenmesinden sonra işlevselliğin kontrol edilmesi için bu test durumu yeniden çalıştırılır.

3.2.6. Testin Raporlanması

Son aşamada, testi yapılan sistemdeki hata dağılımının belirlenmesi ve test senaryolarının analizi için test lideri tarafından bir test döngüsü kapanış raporu hazırlanır. Raporlama aşamasında, mevcut test sürecinde karşılaşılan sıkıntılı durumlar belirtilerek, yeni test döngülerinin iyileştirilmesi adına referans olarak kullanılabilir. Testin raporlanması, sonraki aşamalarda hata dağılımlarının tür ve önem derecesine göre analiz edilmesinde fayda sağlar. Test sonucuna göre çıkış ölçütleri, testin sağlıklı bir şekilde

gerçekleştirildiği hakkında bilgi verir. Çıkış verileri ve beklenen verilerin denkliği veya farklılığı yeni test kriterlerini belirlemede yol gösterici olur. Süreç, proje boyutuna ve hataların riskine göre değişiklik gösterir. Test lideri tarafından hazırlanması gereken raporda test sürecinin özeti, test boyunca kullanılan tanımlayıcılar, testin kapsamlılık değerlendirmesi, sonuçların özeti ve değerlendirilmesi, faaliyetlerin özeti ve kalite güvence ekibi tarafından verilen onay bulunmalıdır. Test kapanış aktivitelerinde en önemli iki unsur, hata analizi ve test özetinin sağlıklı bir biçimde hazırlanmasıdır. *Hata analizi*, var olan hataların öncelik ve risk ortamlarını, bulunduğu fonksiyonları, yoğunlaştığı modülleri, çözüm için gerekli işlem adımlarını ve metotlarını içerir. *Test özeti kapsamı* ise testin amacı, test stratejileri ve metotları, test esnasında üzerinde yoğunlaşılan modüller, test sonuçları ve testlerin değerlendirme ve analizidir. Proje testi için hazırlanan belgeler ve seviyeler, şirket düzeyi ve proje düzeyi baz alınarak Şekil 3.3.'te ifade edilmiştir.



Şekil 3.3. Proje testinde hazırlanan belgeler ve seviyeleri

3.3. Testlerdeki Süreklilik Durumları

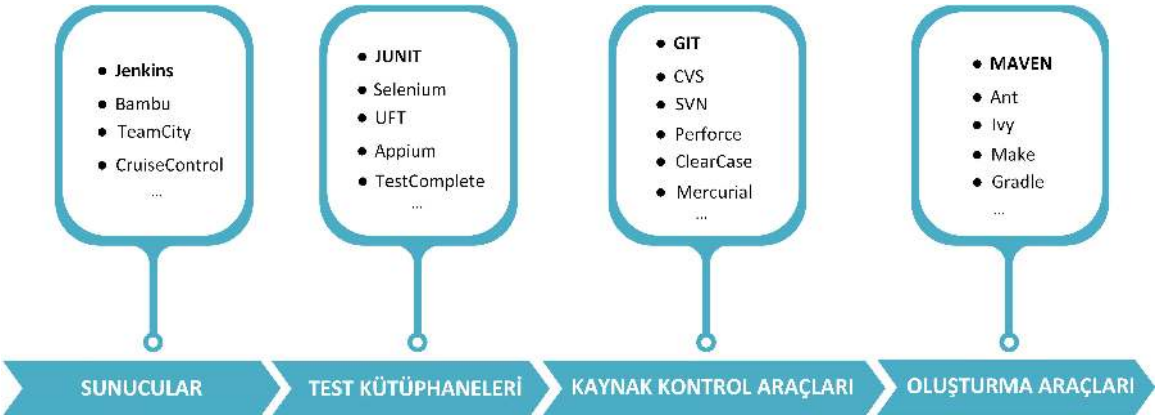
Yazılım sistemlerinin en az hataya sahip olması amacıyla, yazılım geliştirme yaşam döngüsü süresince doğrulama ve geçerleme unsurlarının kontrolü sürekli devam etmelidir. Hatasız yazılımın mümkün olmadığı kabul gören bir gerçektir. Önemli olan hataların büyüklük derecesinin az olması ve sistem sürekliliğini engellememesidir. Yazılımda hataların erken tespit edilmesi, hızlı çözülebilmesi ve sistem işleyişini etkilememesi için farklı yaklaşımlar uygulanmaktadır. Bu yaklaşımlar sürekli entegrasyon, sürekli teslimat ve sürekli dağıtım başlıkları altında yapılmaktadır. Bu yaklaşımların amacı, sistemde bulunan hataların erken tespit edilmesi ve çözümlerinin kısa sürede gerçekleştirilmesini sağlamaktır. Buna ilaveten, sistem sürümlerindeki değişikliklerin yayınlanmasında ortaya çıkabilecek hataların sayısını ve etkisini azaltmaktır.

Sürekli teslimat, sürekli entegrasyon ve sürekli dağıtım işlemleri, Çevik yazılım geliştirme yöntemlerinin ana manifestosu içerisinde yer alan en temel ve önemli özelliklerdendir. Bu özellikler yazılımın yaşam süresini arttırmakla birlikte, nitelikli ve kaliteli yazılım ortaya çıkmasını sağlamaktadır. Günümüzde, Ar-Ge merkezleri ile yazılım geliştirme şirketleri çevik model tabanlı yeni nesil yazılım geliştirme yöntemlerini kullanarak bu süreçleri aktif olarak kullanmaktadırlar. Bu süreçlerin önemsenmesi, müşteri memnuniyetini arttırarak iyileştirilmiş yazılımın geçerliliğini sağlamaktadır.

3.3.1. Sürekli Entegrasyon

Sürekli entegrasyon işlemleri, kod üzerindeki değişikliklerin sürekli bir şekilde izlenmesi, derlenmesi ve test edilmesi için tasarlanan işlem adımları bütünüdür. Sürekli entegrasyon, geliştirici ekibin sisteme günde en az bir defa entegre işlemi yapmasını gerektiren bir durumdur. Birden fazla yazılım geliştiricisinin görev aldığı büyük çaplı projelerde, gün içerisinde oluşturulan birçok değişiklik yazılıma entegre edilmektedir. Yani, sürekli entegrasyon, kod bloklarındaki her değişikliği otomatik olarak test etme mantığına dayanır. Bu mantıkla, kaynak kodların devamlı gelişerek yeni sürümler olarak yayınlanması sağlanır.

Sürekli entegrasyon yaklaşımı ile yazılımdaki herhangi bir hatanın varlığını tespit etmek için otomasyon testi esas alınır. Yazılımlarda gerçekleştirilen sürekli yeni sürümler sayesinde hata tespiti yapılır ve hata çözümlenir. Genellikle birim testleri kullanılarak gerçekleştirilir. Geliştiriciler her an kodlarını derleyerek sürüm oluşturma isteği gönderebilirler. Bu yöntem ile geliştirilen her bir entegrasyonun, otomatik bir yapı sayesinde hatalarından arındırılması amaçlanır. Sürekli entegrasyonda geliştiriciler tarafından doğrulanan sistem, derlenir ve hemen ardından test edilir. Yapılan test işlemi sonucu hatanın tespit edilmesi otomatik bir şekilde gerçekleştirilir. Test süreci başarılı olursa yazılan kod mevcut sisteme entegre edilir. Doğrulama, derleme, test işlemleri birbirini takip eden tekrarlı bir süreç olduğundan bu yöntem sürekli entegrasyon olarak isimlendirilir. Sürekli entegrasyon işleminde kullanılabilecek örnek araçlar Şekil 3.4.'te belirtilmiştir.



Şekil 3.4. Sürekli entegrasyonda kullanılan araçlar

Sürekli entegrasyon sürecinde aşağıda belirtildiği gibi bir yol izlenir.

1. Geliştiriciler tarafından yazılan kodlar sunucuya aktarılarak değişiklikler yapılır.
2. Sunucudaki kod değişiklikleri izlenir ve kontrol edilir.
3. Sunucudaki kodlar derlenerek sistem çalıştırılabilir hale getirilir. Daha sonra sırayla birim ve entegrasyon testleri yürütülür.
4. Testleri başarıyla sonuçlanan kodlar doğrulanarak kabul edilir.
5. Sürüm(versiyon) sistemi yardımıyla derlenen kodlara sürüm numarası ve etiket atanır.
6. Ekip sistemin son durumu hakkında bilgilendirilir. Projede yer alan personellere sistemin son durumu hakkında bilgi verilir. Başarısız testler olursa geliştirici ekip yeniden uyarılır. Böylece ekibin hatalı kod bloklarına hızlı bir şekilde müdahale etmesi sağlanır.

Yazılım yaşam sürecinde sürekli entegrasyon öncesi ve sonrası bazı özelliklerin karşılaştırılması Tablo 3.1.'de sunulmuştur:

Tablo 3.1. Sürekli entegrasyon kullanarak yazılım geliştirmenin karşılaştırılması

Sürekli Entegrasyon Olmadan Geliştirme	**Sürekli Entegrasyon İle Geliştirme**
Hata sayısı daha fazladır.	Hata sayısı azdır.
Entegrasyon sıklığı azdır.	Entegrasyon sıklığı fazladır.
Sürümler az sıklıkta ve geç yayınlanır.	Sürümler düzenli bir şekilde ve hızlı yayınlanır.
Test süreci, zorludur ve uzun sürer.	Test süreci, kolaydır ve hızlı sonuçlanır.
Raporlama süreci zor ve yavaş geçer.	Raporlama süreci hızlı, kolay ve verimli geçer.
Çıktı için yazılımın sonuçlanması beklenir.	Daha şeffaftır ve anında proje çıktısı yayınlanır.

Sürekli entegrasyon sürecinin aşağıda belirtildiği gibi birçok avantajları bulunmaktadır.

- Geliştirilen yazılımların kalitesinin arttırılarak nitelikli olması sağlanır.
- Geliştiricilerin bağımsız ve paralel olarak çalışmasına olanak sağlar.
- Gerçekleştirilen testlerin tekrarlanabilir olmasına yardımcı olur.
- Derleme işleminin tam otomatik bir biçimde gerçekleştirilmesi imkanını sunar.
- Dağıtım işlemleri daha hızlı gerçekleştirilir.
- Hata olduğunda geliştiricilere giden bildirim ile entegrasyon sırasında ortaya çıkabilecek hataların azaltılması sağlanır.

Bunların yanı sıra bir dizi de dezavantajları bulunmaktadır.

- Sürekli entegrasyon araçlarının kurulumu ve bu konuda personelin eğitimi için süre gereklidir.
- Yazılan testlerin prosedürlere uygun bir şekilde oluşturulması gerekmektedir.
- Kapsamlı testlerin gerçekleştirilmesi sürekli entegrasyon sunucusu için fazla sayıda kaynak gerektiren bir işlemdir.
- Birden fazla geliştirici tarafından yazılan kodların sisteme eş zamanda entegre edilmesi durumunda bekleme süreleri uzayabilir.

3.3.2. Sürekli Teslimat

Sürekli teslimat işlemleri, sürekli entegrasyonu kapsayan bir süreçtir. Sürekli teslimat, kısa bir süre içinde yazılım ürünlerinin kolayca yayınlanabilmesi için uygulanan bir süreçtir. Böylece geliştirme sürecindeki yazılım, sık sık küçük değişimler ile güncellenir. Buna bağlı olarak ürünün teslim süresi kısalır. Sürekli teslimat sürecinde kod yayınlanmaya her an hazırdır. Sürekli teslimatta, kodun yayınlanması belirli periyotlarda ayarlanabilir. Kod bloklarında bulunan hataların tespiti için kodun yayınlanma periyotları, mümkün olduğunca kısa süreli olmalıdır. Yayınlama periyotları uzun aralıklarda olması dahilinde kodlar için sorun tespiti ve sorunun düzeltilmesi daha karmaşık bir hal alır.

Sürekli teslimat işlemleri, yeni eklenti ve özelliklerin yapılandırılması ve çözümlenen hataların sistemdeki değişikliklerini yakalamayı amaçlar. Bu yaklaşım ile ortaya çıkan yeni sürümlerin yayınlanmaya hazır olduğu gösterilir. Sürekli teslimat işlemleri, sürekli entegrasyon sonucunda meydana gelen düzenleme veya değişikliklerin kullanıcıya doğru bir şekilde aktarılmasına olanak sağlar. Kod sürekli olarak yayına hazır şekilde versiyonlanır. Ancak yayınlanma işlemi yayıncı tarafından belirlenir. Sürekli teslimat mantığı ile çalışan geliştiriciler, yazılım güncellemelerini devamlı olarak kontrol ederler. Çalışma mantığının test edilmesi için testler kullanılır. Geliştiriciler tarafından kodlar kontrol edildikten sonra yayınlanması için sunucuya gönderilir.

3.3.3. Sürekli Dağıtım

Sürekli dağıtım, yazılım geliştirme yaşam döngüsünü kısa süreli gerçekleştirmeyi hedefler. Bu yaklaşım kaynak kodun yeni sürüm halini aldıktan sonra otomatik olarak yayınlanmasını ifade eder. Sürekli dağıtımda testi başarıyla geçen her kod doğrudan, otomatik bir şekilde yayına alınır. Tamamen otomatikleştirilmiş bir sistemle gerçekleşen bu süreci, yalnızca kod testinin başarısız sonuçlanması engeller. Sürekli dağıtım ile yapılan her değişiklik doğrudan kullanıcının kullanımına hazır hale gelir. Geliştirilen yeni özelliklerin ve eklentilerin, daha hızlı dağıtılmasını ve doğrulanmasını kolaylaştırır. Sürekli dağıtım işlemlerinde yeni sürümler anında yayınlandığı için kullanıcılardan geri bildirim alma imkanı doğar. Geliştirilen sistemin anında kullanıcıya sunulması ve proje bitimi için belirli bir tarihe bağlı kalınmaması nedeniyle geliştiriciler üzerinde teslimat zamanı baskısı olmaz. Herhangi bir test stratejisi kullanılabilir.

Yazılım test sürecinde önemli noktalar olarak ifade edilen sürekli entegrasyon, sürekli teslimat ve sürekli dağıtım terimleri ile ilgili özellikler Tablo 3.2.'de gösterilmiştir:

Tablo 3.2. Sürekli entegrasyon, Sürekli teslimat ve Sürekli dağıtım kıyaslaması

Özellik	**Sürekli Entegrasyon**	**Sürekli Teslimat**	**Sürekli Dağıtım**
Kod bloklarındaki değişikliğe göre otomatik test	X	X	X
Sürüm halinde devamlı gelişen kaynak kod	X	X	X
Erken aşamalarda gerçekleşen hata tespiti ve daha kolay çözümü	X	X	X
Ortaya çıkan sürümlerin yayınlanmaya hazır olması		X	X
Sürüm halindeki kaynak kodun otomatik yayına alınması			X
Yapılan her değişikliğin doğrudan kullanıcının kullanımına açılması			X
Yeni özellik ve eklentilerin daha hızlı dağıtılması ve doğrulanması			X

4. Yazılım Test Teknikleri

Yazılım test yaşam döngüsü, sistem davranışları ile geliştirici/kullanıcı beklentilerini kıyaslama sürecidir. Test süreci, daha kaliteli ve en az hata oranına sahip ürünlerin elde edilmesi amacıyla gerçekleştirilir. Yazılım test yaşam döngüsü sürecinde kullanılan test teknikleri, testin amaç ve kapsamına, sistem bilgisine, yazılım yaşam döngüsüne göre üç farklı şekilde gerçekleştirilebilir. Yazılım test tekniklerinin belirlenmesindeki en ayırt edici unsur, yazılım modülünün iç detayının bilinip bilinmeyeceğidir. Bu kriterleri dikkate alarak kullanılabilecek teknikler Şekil 4.1.'de gösterilmektedir. [7] [49] [50]:

Şekil 4.1. Yazılım test teknikleri

4.1. Kara Kutu Testi

Kara kutu test tekniğinde yazılım kodlarına ve içeriğine bakılmadan yazılımın işlevselliği değerlendirilir. Dikkat edilen en önemli husus, çalıştırıkan programın doğru sonuç vermesi ve arayüzler olarak da kullanıcı beklentisini karşılamasıdır. Kara kutu testinde ana odak noktası, bir bütün olarak sistemin işlevselliğinin değerlendirilmesidir. Test edilen uygulama kullanıcının bakış açısıyla kontrol edilip onaylanır. Bu da Kara kutu test yönteminin kullanıcı gereksinimlerine dayandığının kanıtıdır. Kara kutu test tekniğinde, sistemde önceden tanımlanan gereksinimlerin doğruluğu da kontrol edilir.

Kara kutu test yöntemi kullanılarak aşağıdaki hususlar kontrol edilir:

- Kullanıcılar tarafından gerçekleştirilen doğru eylemler
- Sistemin girdilerle etkileşimi

- Sistemin etkileşimlere yanıt verme süresi
- Veri yapıları ve veri türlerinin doğru kullanımı
- Kullanıcı fiziksel arayüzünde olabilecek sorunlar
- Yazılımın performans durumları
- Yazılımda olabilecek beklenmedik uygulama hataları

Kara kutu testi ile gerçekleştirilen test çeşitlerinde, uygulamanın gereksinim ve özelliklerine odaklanıldığı için, *Şartname Tabanlı Test*, *Girdi-Çıktı Testi*, *Kapalı kutu Testi* gibi isimlerle de bilinir. Kara kutu testleri yazılımın kod kısmıyla değil, doğru çalışması ve doğru sonuç vermesi ile ilgilenmektedir. Dolayısı ile yazılımın çalışmasının test edilmesi için örnek giriş değerlerinin belirlenmesi gerekmektedir. Örnek giriş değerleri belirlenirken farklı sonuçlara sebep olacak giriş değerlerinin seçilmesi testin daha verimli bir şekilde gerçekleştirilmesine sebep olacaktır. Kara kutu testlerinde, test verilerinin seçilmesi için farklı yaklaşımlar bulunmaktadır. Bu temel yaklaşımlar aşağıda verilmiştir:

1. Eşit Bölmelere Ayırma (Equivalence Partitioning)
2. Sınır Değerleri Analizi (Boundary Value Analysis)
3. Karar Tablosu (Decision Table)
4. Durum Geçiş Tablosu (State Transition Table)
5. Kullanım Durumları Testi (Use Case Testing)
6. Hata Tahmin Etme (Error Prediction)
7. Grafik Tabanlı Test (Graph-Based Testing)
8. Karşılaştırma Testi (Comparison Testing)

Bu bölümde yukarda verilen yaklaşımlar örnek bir senaryo üzerinden açıklanacaktır. Kullanılacak olan örnek senaryo kullanıcı girişi olacaktır. Senaryoya göre eşleşen kullanıcı adı ve şifre ile giriş yapan kullanıcıların sisteme başarılı bir şekilde girebilmesi için e-Posta onayı alması gerekmektedir. Başarılı bir şekilde giriş yapan kullanıcıların, daha önceden yapmış oldukları giriş sayısına göre yeni üye, sadık üye ve profesyonel üye gibi farklı sayfalara yönlendirilmesi sağlanacaktır. Üstelik art arda 3 kez hatalı şifre giren kullanıcıların üyelikleri askıya alınacaktır. Bu senaryo rastgele test verileri kullanılarak gerçekleştirilebilir. Ancak test verilerinin her bir durumu test etmesinin garantisini vermez. Bu sebeple Kara kutu test verilerinin belirtilen yaklaşımlarla seçilmesi önemlidir.

4.1.1. Eşit Bölümlere Ayırma

Eşit Bölümlere Ayırma, Denklik Paylarına Ayırma Tekniği (Equivalence Partitioning) olarak da bilinir. Bu yöntem ile sistemde aynı ya da benzer davranışı sergileyebilecek olan girdi veya çıktıları gruplandırıp veri setinde ortaya çıkabilecek yoğunluğu önlemek amaçlanır. Bu tekniğin esas amaçlarından bir diğeri de benzer çıktıyı üretebilecek durumları yok ederek, gereksiz senaryoların azaltılmasıdır. Denklik paylarına ayırma tekniğine örnek, kullanıcı girişinin e-Posta onayı ile başarılı olmasının sağlanması verilebilir. Dolayısı ile oluşturulması gereken test verileri eşit sayıda onaylı doğru şifre, onaysız doğru şifre, onaylı yanlış şifre ve onaysız yanlış şifre durumlarına ait veriler içermelidir. Tablo 4.1.'de farklı denklik sınıflarına ait birer adet veri verilmiştir.

Tablo 4.1. Farklı denklik sınıflarına ait test verileri

Kullanıcı Adı	Şifre	Girilen Şifre	e-Posta Onay
Kullanıcı1	123	123	1
Kullanıcı2	123	111	1
Kullanıcı3	123	123	0
Kullanıcı4	123	112	0

4.1.2. Sınır Değerleri Analizi

Sınır Değerleri Analizi (Boundary Value Analysis), geliştirilen test senaryolarındaki verilerin uç değerlerde olmasıyla gerçekleştirilir. Giriş verilerinin sınırlarından kaynaklanan kusurları veya hataları tanımlamak için kullanılan yöntemdir. Belirlenen uç değer aralıklarındaki formların çıktı sonuçları değerlendirilir. Bu formlarda sınır değerlerin girilmesi ile aynı sonucu vermesi beklenir.

Sınır değerleri analizi tekniği örneğinde, kullanıcının toplam giriş sayısına göre yönlendirme yapılması gerekmektedir. 0-1.000 kez giriş yapan kullanıcılar standart üye, 1.000-10.000 arası giriş yapanlar sadık üye ve 10.000-üzeri giriş yapanların kadim üye sayfalarına yönlendirilmesi gereksin. Bu durumda giriş sayısı olarak bütün değerleri test etmek yerine sadece kritik değerler test edilir. Bu değerler örnek senaryo için 999-1.000-1.001 ve 9.999-10.000-10.001 olacaktır. Kullanılabilecek örnek test verileri Tablo 4.2.'de verilmiştir.

Tablo 4.2. Farklı sınır değerlerine ait test verileri

Kullanıcı Adı	Şifre	Girilen Şifre	Toplam Giriş Sayısı	e-Posta Onay
Kullanıcı1	123	123	999	1
Kullanıcı2	123	123	1.000	1
Kullanıcı3	123	123	1.001	1
Kullanıcı4	123	123	9.999	1
Kullanıcı5	123	123	10.000	1
Kullanıcı6	123	123	10.001	1

4.1.3. Karar Tablosu

Karar Tablosu (Decision Table) yaklaşımında, farklı olasılıkları dikkate alarak çeşitli test senaryoları geliştirilir. Bir tablo ile bu test senaryoları ve çıktılar arasındaki sonuçlar belirtilir. Bu tablolar dikkate alınarak elde edilen test sonuçları arasında mantıksal bir bağ kurulur. Test uzmanları, mantıksal kuralları belirler ve sonraki aşamalarda her kurala özgü bir test senaryosu tasarlayabilir. Çeşitli giriş kombinasyonları ve çoklu olasılıklar olması durumunda tercih edilir.

Karar tablosu tekniği örneğinde, bütün koşullar ve koşullara göre muhtemel durumlar sonucu elde edilmesi gereken çıktıları göz önüne almak gerekir. E evet, H hayır olmak üzere, Tablo 4.3. karar tablosunda oluşacak tüm durumlar ve sonuçları örnek olarak verilmiştir.

Tablo 4.3. Karar tablosu tabanlı değerlendirmede oluşacak ihtimaller

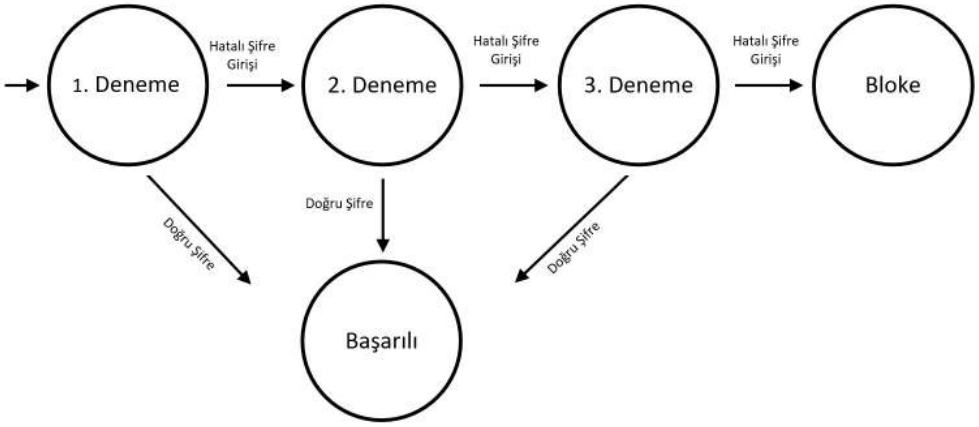
Eşleşen Kullanıcı Adı-Şifre	Onaylı e-Posta Adresi	Toplam Giriş Sayısı	Sonuç
E	E	0-1.000	Standart
E	E	1.000-10.000	Sadık
E	E	10.000+	Kadim
E	H	0-1.000	Onay
E	H	1.000-10.000	Onay
E	H	10.000+	Onay
H	E	0-1.000	Hata
H	E	1.000-10.000	Hata
H	E	10.000+	Hata
H	H	0-1.000	Hata
H	H	1.000-10.000	Hata
H	H	10.000+	Hata

4.1.4. Durum Geçişi

Durum Geçiş Tablosu (State Transition Table), sistemin durumu koşullara veya olaylara bağlı olarak değiştiğinde, senaryo haline gelen durumlar tetiklenir ve bir test uzmanı tarafından test edilir. Bir uygulamanın daha önce gerçekleştirdiği durumlarla alakalı olarak aynı girdi değerine karşılık farklı bir çıktı değeri verdiğinde uygulanması gereken test tekniğidir. Bu test tekniğinde, test aşamasında girdiler, çıktılar ve sistemin durumu kullanılır. Yazılım, test verileri arasındaki geçişlerin veya olayların sırasına göre

kontrol edilir.

Durum geçiş tablosu tekniği örneğinde, kullanıcının art arda 3 kez hatalı şifre girmesi durumunda hesabın dondurulması gerekmektedir. Bu işlem için gereken durum geçiş diyagramı Şekil 4.2.'de verilmiştir.



Şekil 4.2. Kullanıcı girişi için durum geçiş diyagramı

Sonuç olarak her durum için test verileri oluşturulmalıdır. Verilen senaryo için oluşturulması gereken örnek test verileri Tablo 4.4.'te verilmiştir.

Tablo 4.4. Farklı durum geçişlerinin testi için kullanılabilecek örnek test verileri

Kullanıcı Adı	Şifre	Girilen Şifre	Hatalı Giriş Sayısı	e-Posta Onay
Kullanıcı1	123	123	0	1
Kullanıcı2	123	111	0	1
Kullanıcı3	123	123	1	1
Kullanıcı4	123	111	1	1
Kullanıcı5	123	123	2	1
Kullanıcı6	123	111	2	1

4.1.5. Kullanım Durumları Testi

Kullanım durumları testi (Use-Case Testing) tekniğinde fonksiyonel ve fonksiyonel olmayan gereksinimler kullanılarak test senaryosu üretilir. Bu teknikteki en önemli özellik, senaryoların açık ve detaylı olmasıdır. Temel olarak test yöntemi bir kullanıcı ve bu kullanıcının sistemden beklediği durumların oluşup oluşmadığını ölçümler. Kullanım durumları tekniğini verilen senaryo üzerinde örneklendirecek olursak, kullanıcının giriş yapması için kullanıcı adı ve şifre alanlarını doldurmuş olması gerekir. Kullanım durumları testine göre oluşabilecek durumlar; kullanıcı adının girilmemesi, şifrenin girilmemesi ve hem kullanıcı adı hem şifrenin girilmemesi olarak karşımıza çıkacaktır. Kullanım durumları testi için bu durumların incelenerek sistemin tepkisi ölçülmelidir.

4.1.6. Hata Tahmin Etme

Belirli bir hata tahmin türü, test edilen sistemi etkileyebilecek bilinen yazılım açıklarını test etmektir. Bu teknik, geliştiriciler tarafından benzer sistemler oluşturulurken yapılan yaygın hataların test edilmesini içerir. Bu test senaryoları tasarlama yöntemi, sistemde mevcut olabilecek hataları düzeltmek için çıktı ve girdiyi tahmin etmekle ilgilidir. Test uzmanının tecrübe ve öngörülerine dayalı bir tekniktir.

4.1.7. Grafik Tabanlı Test

Bu test yöntemi ile sistemin işleyişini sağlayan girdiler ve çıktılar arasındaki bağlantıyı göstermek için bir grafik çizimine ihtiyaç duyulur. Bu test içeriğinde grafik çizimi yapılan girdi ve çıktıların farklı kombinasyonları kullanılır.

4.1.8. Karşılaştırma Testi

Aynı yazılımın bağımsız sürümleri ile gerçekleştirilen bir test yöntemidir. Test sonuçlarını karşılaştırmak ve doğrulamak için aynı yazılımın iki farklı sürümünü kullanır. Test uzmanları tarafından sürümlerdeki farklılıklar ve gereksinimlerin karşılanma oranları dikkate alınarak bir rapor hazırlanır.

4.2. Beyaz Kutu Testi

Genellikle birim ve entegrasyon test seviyesinde etkin kullanılan bu test, literatürde *Cam kutu, Saydam kutu, Şeffaf kutu, Filigran Test* veya *Yapısal Test* adlarıyla da bilinir. Bu test yöntemi ile uygulamaların temeli olan kod kısımları incelenir. Bu test tekniği ile bir kod bloğundaki güvenlik açıkları, kodlamadaki hatalı ya da eksik yapılandırılmış döngüler, gereksiz veya kısır döngüler, koşullu döngülerin işlevselliği gibi kriterler kontrol edilir. Bu tekniğin, temel amacı kodun iç yapısı incelemek ve doğrulamaktır [51]. Beyaz kutu testinde hatalar tespit edilerek kod iyileştirmeleri yapılır. Genellikle test senaryolarında kodlardaki birbirinden farklı tüm yolların incelendiği kapsamlı bir test tekniğidir. Bu yöntem gerçekleştirilirken, sistem modüler şekilde incelenir. Test çıktılarının doğru sonuçlanması için test ekibi, yazılım konusunda uzman ve tecrübeli olmalıdır. Bu sebeple karmaşık, uzun süre gerektiren ve pahalı bir teknik olan Beyaz kutu testi, kodun işlevselliğini kontrol ettiği için güvenilir bir tekniktir.

Beyaz kutu test teknikleri, kara kutu testinde olduğu gibi sadece işlevsellikten ziyade iç yapıları, kullanılan veri yapılarını, iç tasarımı, kod yapısını ve yazılımın çalışmasını analiz eder. Beyaz kutu testinin çalışma sürecinde aşağıdaki adımlar izlenir:

- *Girdi:* Gereksinimler, İşlevsel özellikler, tasarım belgeleri, kaynak kodu.
- *İşleme:* Tüm süreç boyunca rehberlik etmek için risk analizi yapmak.
- *Uygun test planlaması:* Test senaryolarının tüm kodu kapsayacak şekilde tasarlanması.

Hatasız yazılıma ulaşılana kadar durulama tekrarını yürütün. Ayrıca sonuçlar iletilir.

- *Çıktı:* Tüm test sürecinin nihai raporunun hazırlanması.

Beyaz kutu tesitinin avantajları şunlardır:

- Beyaz kutu testi, tüm kod ve yapılar test edildiğinden çok kapsamlıdır.
- Kod kaldırma hatalarının optimizasyonu ile sonuçlanır ve fazladan kod satırlarının kaldırılmasına yardımcı olur.
- Kara kutu testinde olduğu gibi herhangi bir arayüz gerektirmediğinden daha erken bir aşamada başlayabilir.
- Otomatikleştirmek kolay.

Beyaz kutu tesitinin dezavantajları şunlardır:

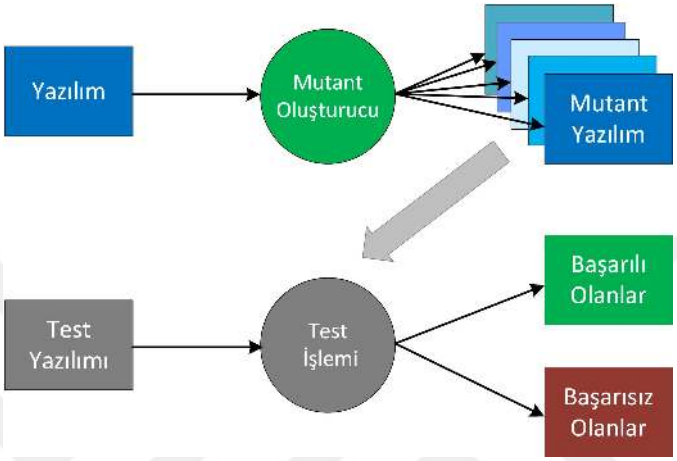
- Ana dezavantajı, çok pahalı olmasıdır.
- Kodun yeniden tasarlanması ve kodun yeniden yazılması için test senaryolarının yeniden yazılması gerekir.
- Test edenlerin, kara kutu testinin aksine kod ve programlama dili hakkında derinlemesine bilgi sahibi olmaları gerekir.
- Mevcut kod test edildiğinden eksik işlevler tespit edilemez. Çok karmaşık ve bazen gerçekçi değil.

4.2.1. Mutasyon Testi

Mutasyon testi, uygulamada yer alan belli bir hatanın düzeltildikten sonra test edilmesidir. Mutasyon testi, programda meydana gelen küçük değişiklikleri içerir. Bu tür değişiklikler, sistemde tespit edilen düşük seviyeli kusurların çözümlenmesi ile oluşur. Hataların çözümlenmesini doğrulamak için gerçekleştirilir. Zaman alıcı ve pahalı bir yöntem olan mutasyon testi, büyük ve karmaşık projeler için tercih edilmez. Ancak bununla birlikte mutasyon testi sayesinde kaynak koddaki tüm hatalar tespit edilerek hataların giderilmesi sağlanır. Böylelikle yazılımın kalitesi artar. Bu test metodu ile sistemde bulunan kod parçalarındaki bazı ifadelerin değiştirilmesi sonucu sistemin doğru şekilde çalışma durumu test edilir. Asıl amaç, mutasyona uğrayan kodun, sistemin çalışmasını etkileme durumuna göre sistemin hassasiyeti, test senaryoları ve gerçekleştirilen test sonuçlarının kalitesinin değerlendirilmesidir. Bu yöntem, programda oluşan bir hatanın sistemi etkileme durumunu ele aldığı için *Arıza Temelli Test Stratejisi* olarak da bilinir.

Beyaz kutu yöntemiyle gerçekleştirilen bu test türü, yazılımın kaynak kodunun yalnızca belirli bir bileşenini değiştirerek gerçekleştirilir. Asıl amaç, geliştirilen test senaryoları ile hataları bulup bulamayacağının doğrulanması için kaynak koddaki bazı sözdizimlerin kasıtlı

bir şekilde değiştirilmesine dayanır. Mutasyon testi, yazılım test senaryolarının doğruluğunu kanıtlamayı amaçlar. Bir nevi, testin yeniden test edilmesidir. Test uzmanları tarafından gerçekleştirilen bu yöntemde, mutant adı verilen sürümlerle hatalı kodlar oluşturulur. Her varyasyon bir çeşit hatayı içermelidir. Daha sonra geliştirilen test senaryoları orijinal ve mutant programa ayrı ayrı uygulanmalı, her iki sürüm değerlendirilir ve değerlendirme sonucuna göre hatalı kodun test senaryosuna karşı davranışı analiz edilir. Mutasyon testinin çalışma mantığı Şekil 4.3.'te gösterilmiştir.



Şekil 4.3. Mutasyon testi

- *Değer Mutasyonları:* Programdaki hataların tespiti için değerlerin değiştirildiği mutasyon testi yöntemidir. Temelde sabit olarak kullanılan değerlerden küçük değer, daha büyük değerle veya tam tersi durumla değiştirilir.
- *Karar Mutasyonları:* Aritmetiksel ve mantıksal operatörlerin yanında ilişkisel operatörlerle ilgili var olan hataların tespitinde kullanılır.
- *İfade Mutasyonları:* Sistemde bulunan kod bloklarının silinmesi veya farklı bir ifadeyle değiştirilmesi durumunda oluşan hataların tespitinde kullanılır.

4.2.2. Döngü Testi

Tamamen döngü yapılarına odaklanan bir test türüdür. Sisteme ait kodda bulunan döngülerin tekrarları sonucu doğru çıktıyı üretmesini, kodun döngüye giriş-çıkış noktalarını ve döngü içerisinde yer alan modülün doğru çıktıyı üretmesini kontrol eder.

4.3. Gri Kutu Testi

Kara kutu ve Beyaz utu test tekniklerinin birlikte yürütülmesiyle gerçekleştirilir. Bu yöntemde testi yapan kişiler, sistemi gerçekleştirirken oluşturulan tasarım belgelerine erişebilirler. Beyaz kutu testi ile sistemin kodunun incelenmesinde, bu belgeler yol haritası niteliğinde olmalıdır. Bu test tekniğinde, yazılımın iç işleyişi hakkında kısmi bilgi ile test

işlemi gerçekleştirilir. Testin gerçekleştirilmesi sırasında kaynak koda erişime gerek duyulmaz.

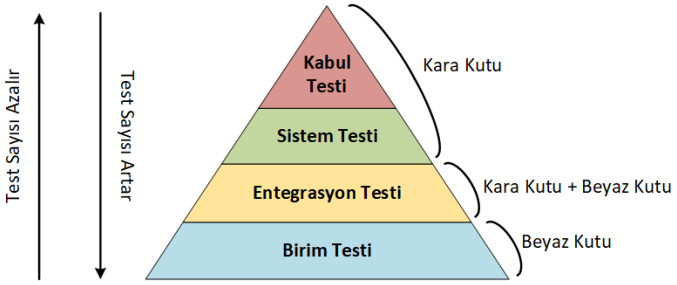
Gri kutu testi, sistemin dahili işlevselliği hakkında sınırlı bilgi ile gerçekleştirilen test tekniğidir [52]. Bu test yöntemiyle sistem hatalarının genel maliyeti ve test işlemi sırasında harcanan zaman azalır. Gri kutu testleri, hedef sistemin duruma dayalı modellerine, UML diyagramlarına veya mimari diyagramlarına göre oluşturulur.

5. Yazılım Test Seviyeleri

Yazılım gereksinimlerinin tespit edilmesiyle başlayan, planlama, çözümleme, tasarım, kodlama, bakım ve onarım aşamasıyla devam eden süreç, yazılım yaşam döngüsünü oluşturur. Yazılım yaşam döngüsünün test aşaması, daha kaliteli ve kullanışlı bir ürün geliştirmek amacıyla gerçekleştirilir. Test aşaması sistemin hatalarının tespit edilmesi, eksiklerin giderilmesi, isteklere uygunluğunun tespiti ve düzenlenmesi için gerçekleştirilir. Yazılım test süreci, yazılım yaşam döngüsü gibi belirli aşamalara sahip bir aktivitedir. Yazılımın beklenen seviyeye uygunluğunun belirlendiği, beklenen seviye ve kalitede değil ise kriterlere uygunluğunun sağlanmaya çalışıldığı bir süreçtir. Test sürecinde test edilecek kriterlerin veya eksikliklerin analiz edilmesi, testin gerçekleştirilmesi için uygun plan ve tasarım, teste hazırlık, testin gerçekleştirilmesi ve ürünün son hali aşamaları izlenmelidir. İzlenen aşamalar sonrasında yazılımın, kalite standartlarını ve müşteri beklentilerini karşılaması beklenir [7].

Yazılım test seviyeleri, temel olarak eksik alanların belirlenmesi ve geliştirilmesi, yaşam döngüsü aşamaları arasındaki tekrarlanmaların önlenmesi için dikkate alınması gereken bir noktadır. Test seviyelerinin amacı, yazılım testinin sistematik bir hale getirilmesi ve belirli bir seviyedeki olası tüm senaryoların kolayca belirlenip kontrol edilmesini sağlamaktır. Yazılım test seviyeleri dört adımdan oluşmaktadır. En temel adım olan ve küçük modüllerin testiyle başlayan birim testi, modüllerin birlikte kusursuz çalışmasını kontrol eden entegrasyon testi, sistemin işlevselliğini kontrol eden sistem testi ve son olarak kullanıcı gereksinimlerini karşılama durumunu değerlendirmek için kullanıcı tarafından gerçekleştirilen kullanıcı kabul testi test seviyelerinin adımlarıdır. Test seviyesi ilerledikçe test sayısı azalır. Bunun nedeni, modüler ve entegre şekilde sorunsuz çalışan sistemin doğruluğunun kanıtlanması ve bakım aşamasının öncelikli hale gelmesidir.

Yazılım test seviyeleri ve seviyelere göre genellikle kullanılan test teknikleri Şekil 5.1.'de gösterilmiştir.



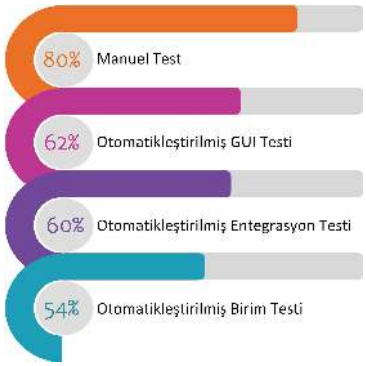
Şekil 5.1. Yazılım test seviyeleri

5.1. Birim Testi

Modül Testi, *Bileşen Testi* ve *Unit Test* isimleriyle de bilinir. Sistemin derlenebilir, çalıştırılabilir en küçük parçasının test edilmesidir. Geliştiriciler tarafından gerçekleştirilir.

Kaynak kodun her biriminin veya modülünün doğru şekilde çalışması ve beklenen çıktıyı üretmesi test edilir [53]. Bir yazılımın ayrı birim veya bileşenlerinin test edildiği yazılım tekniğidir. Büyüklüğü fark etmeksizin geliştirilen projelerin sınanması için faydalanılan ilk test seviyesidir. Sonraki aşamalarda entegrasyon testinin daha güvenilir sonuçlar verebilmesi için bu yöntemin sağlıklı bir şekilde gerçekleştirilmesi gerekir. Genellikle beyaz kutu yöntemi ile gerçekleştirilir. Amaç, modüllerde oluşan hataların çözülmesi ve modüllerin gereksinimlere uygun çalıştığının kontrol edilmesidir.

Birim testinin sağlıklı sonuçlanması, sistemin ileri aşamalarında gerçekleştirilecek olan diğer test seviyeleri ve türleri için zaman ve maliyet kazancı sağlar [54]. Birim testinin, kodlama işleminin erken aşamalarında gerçekleştirilmesi sayesinde test sonucu oluşabilecek hatalar en az maliyetle düzeltilebilir. Birim test seviyesi hata yakalamanın en kolay olduğu test seviyesidir. Birim testi manuel veya otomatik olarak gerçekleştirilebilir olmasına rağmen otomatik test kullanımı daha yaygındır. Ayrıca Uluslararası Yazılım Test Yeterlilikler Kurulu(ISTQB) 2020-2021 yılı raporunda, ülkemizde bulunan kuruluşların söz konusu yazılım test türü olunca gösterdikleri çabaya yönelik araştırma sonuçları belirtilmiştir. Yapılan analiz sonucu manuel gerçekleştirilen test türleri en yüksek yüzdeliğe sahip olurken, otomatik birim testin en düşük yüzdelik dilime sahip olduğu görülmüştür. Kuruluşlarca göz ardı edildiği belirtilen birim testinin, diğer testlere ve test seviyelerine oranla hatanın erken fark edilmesi, çözümlenmesi ve hatasız sistemin daha erken sürede kullanıma sunulması gibi faydaları dikkate alınmalıdır. Bu faydalar yardımıyla, sistemin zaman ve maliyet açısından maksimum verimle geliştirilmesi sağlanır [55]. Rapor sonucu belirtilen yüzdelik oranlar Şekil 5.2.'de gösterilebilir.



Şekil 5.2. Türkiye'de kuruluşların yazılım testlerini kullanma oranları [55]

5.2. Birleştirme Testi

Birleştirme (entegrasyon) testi, birbirinden bağımsız test edilen ve başarısı doğrulanan iki modülün, birbirine bağlanması ve bağlantı sonrasında bu iki modül arasındaki bağlantının hatasız işlem yapabilmesi için birlikte çalışabilirlik düzeyinin test edilmesidir. Entegrasyon testi ile farklı modüllerin geçişleri sırasında arayüzlerin davranışları ve yine farklı modüller arasındaki veri akışı kontrol edilir. Entegrasyon testinin amacı, daha önce

test edilen birimlerden ziyade, doğruluğu kanıtlanmış birimlerin entegre bağlantılarını test etmektir [56]. Birleştirme testi ile birlikte doğru çalışıp çalışmadıklarını değerlendirmek için sistemin farklı bölümleri bir arada test edilir [57]. Entegrasyon testi dört farklı yaklaşım ile gerçekleştirilir.

5.2.1. Büyük Patlama Yaklaşımı

Big Bang Testi olarak da bilinir. Bu yöntemde, birleştirilecek modüllerin testleri bir arada yapılır. Birim testlerinin tamamlanmasının ardından tüm modüllerin işlevselliğinin birleştirildiği ve doğrulandığı en basit entegrasyon test yaklaşımıdır. Bu yaklaşımın kullanımı için küçük projelerin uygun olduğu savunulur. Çünkü entegrasyon sırasında bulunan bir hatanın, potansiyel olarak entegre edilen modüllerden herhangi birine ait olduğu bilinse de hatanın yerelleştirilmesi oldukça zordur. Bu da hata lokalizasyonu konusunda güçlük yaşanacağını gösterir. Ayrıca bu yaklaşımda test edilmesi gereken çok sayıda arayüz göz önüne alındığından test edilecek bazı arayüz bağlantıları kolayca gözden kaçabilir. Ayrıca entegrasyon testinin başlaması tüm modüllerin tasarlanmasına bağlı olduğundan test ekibinin test aşamasını yürütmesi için kısıtlı zamanı olacaktır. Tüm modüllerin aynı anda test edilmesi, yüksek riskli kritik modüllerin izolesini engeller. Bu yöntemin avantajı, entegrasyon işleminden önce zamandan tasarruf sağlar. Ancak bu testlerde ortaya çıkan bir hatanın tespit edilmesi, nereden kaynaklandığının araştırılması ve giderilmesinde zorluk yaşanabilir.

5.2.2. Yukarıdan Aşağıya Yaklaşım

Bu test tekniği gerçekleştirilirken, yazılım sisteminin kontrol akışı için belirtilen adımlar, yukarıdan aşağıya entegrasyon yapılmasıyla sağlanır. Yazılım işlevselliğini kontrol etmek için önce daha yüksek seviyeli modüller test edilir ve ardından daha düşük seviyeli modüllerin testi gerçekleştirildikten sonra entegre edilir. Bu, en üst düzey birimlerin önce test edildiği ve alt düzey birimlerin bundan sonra adım adım test edildiği bir yaklaşımdır. Yöntem gerçekleştirilirken her bir modül "kök" adıyla belirtilir. Testin son aşamasında her bir modülün hatasız olarak test edilmesi amaçlanır ve bu sayede küçük modüllerin entegre edildiğinde hatasız bir şekilde çalışması beklenir.

5.2.3. Aşağıdan Yukarıya Yaklaşım

Bu test tekniği gerçekleştirilirken, yazılım sisteminin kontrol akışı için belirtilen adımlar, aşağıdan yukarıya entegrasyon yapılmasıyla sağlanır. Aşağıdan yukarıya test yönteminde, öncelik sırası düşük seviyeli olmakla birlikte tüm modüller birim test tekniğiyle test edilir [51]. Her bir modül test edildikten ve başarısı kesinleştikten sonra bir üst seviyede bulunan modül ile birleştirilerek tekrar test işlemi gerçekleştirilir. Testin ilerlemesinde mantık, önce alt modüllerin daha sonra ana modülün doğrulanmasıdır.

5.2.4. Hibrit Entegrasyon Testi

Bu test yaklaşımı, yukarıdan aşağıya ve aşağıdan yukarıya test tekniklerinin birlikte kullanılmasıyla gerçekleştirilir. Bu test türünde karmaşık yapıya sahip olan modüller üzerinde entegre edilmeden test işlemi uygulanırken, basit ve sade yapıya sahip olan modüller üzerinde entegrasyon sonrası test işlemi gerçekleştirilir. Bu yaklaşım, birkaç alt projesi olan çok büyük projeler için kullanışlıdır. Hibrit test yöntemi, test esnasında hem alt modüllerin hem de ana modüllerin eş zamanlı yürütülmesinden dolayı yüksek maliyet gerektirir.

5.3. Sistem Testi

Entegrasyon testinden sonraki adım ise sistem testidir. Bu test tekniğinde sistem bir bütün olarak ele alınır ve test edilir. Tamamı entegre edilen bir yazılım ürününün testidir. Sistemin işleyişinin doğruluğu, entegrasyon testinde gözden kaçabilecek hataların keşfi için yüksek kalitede bir ürün sunmak adına gerçekleştirilir. Kara kutu test tekniği ile sınanır.

Sistem testinde, bileşenlerin birbirleriyle ve bir bütün şekilde sistemle etkileşimi kontrol edilir [11]. Bu yöntemde sistem, kaynak koduna bakılmadan, uygulamanın dışardan görünüşü baz alınarak daha önce oluşturulan test senaryoları ile uçtan uca test edilir. Kullanıma sunulmadan önce gerçekleştirilir. Sistem testi, test uzmanları tarafından, eksiksiz ve tam entegre bir yazılım ürününün doğrulanması için gerçekleştirilen seviyedir. Bu test aşaması test uzmanlarının, ürünün iş gereksinimlerini karşıladığından emin olmalarını ve işletim ortamında sorunsuz bir biçimde çalıştığını belirtmeleri için önemli bir adımdır.

5.4. Kullanıcı Kabul Testi

Bu test seviyesi, son kullanıcılar ve test ekibi tarafından gerçekleştirilir. Bu test seviyesinde amaç, sistemin kullanıcı kriterlerine uygunluğunu ve kullanıcı tarafından kolay kullanılabilir olduğunu test etmektir. Kullanıcının sisteme dair talepleri ve sistemin bu taleplere uygunluğu kontrol edilir. Bir sistemin kabul edilebilirlik açısından değerlendirilmesi ile uçtan uca iş akışının doğrulanması, bu test seviyesinde gerçekleştirilir [7].

Kullanıcı kabul testi planı, yazılım yaşam döngüsünün erken safhalarında, kullanıcı gereksinimleri baz alınarak hazırlanmalıdır. Kullanıcıların, yazılım sisteminden beklentilerinin doğrulanması amacıyla yapılır. Yazılım geliştirme yaşam döngüsü ve alt süreçleri tamamlandıktan sonra birçok test metotlarından geçirilir. Yazılım, son süreçte yer alan kullanıcı kabul testi ile Alfa, Gama ve Beta testlerine tabi tutulmaktadır.

5.4.1. Alfa Testi

Sınırlı sayıdaki gerçek kullanıcıların dahil edildiği, geliştiriciler ve test ekibinin gözetim ve kontrolü altında gerçekleştirilen kullanıcı kabul testi aşamasıdır. Alfa testinde sistem,

müşteri ya da son kullanıcı tarafından geliştirme ortamında test edilir. Bu yöntemin amacı, sistemin kullanıcı kriterlerine uygunluğunu ve kullanıcı tarafından kolay kullanılabilir olduğunu test etmektir. Kullanıcının sisteme dair talepleri ve sistemin bu taleplere göre tutarlığı kontrol edilir. Yazılım sistemleri, Alfa testinde başarılı çıktıktan sonra Beta testi gerçekleştirilir. Alfa testi gerçekleştirilirken beyaz kutu veya kara kutu tekniklerinden yararlanılabilir. Alfa testi, kullanıcı kabul testinin başlangıç aşaması olarak görülebilir. Bu aşamada, sistemde kullanıcı isteklerine uymayan, kolay kullanıma engel olan ve sistemde var olan sorunlar düzenlenir. Alfa testinin başarılı sonuçlar vermesi için kusursuz ve her modülü kontrol edebilecek şekilde kapsamlı test senaryolarının geliştirilmesi gerekmektedir. Testin sonucunun raporlanması, Gama ve Beta testlerindeki adımların işleyişini de kolaylaştırmalıdır.

5.4.2. Beta Testi

Alfa testinin başarıyla sonuçlanmasından sonraki aşamadır. Sınırlı sayıda gerçek kullanıcı tarafından kullanıcı ortamında yürütülen bir test çalışmasıdır. Bu aşamada temel amaçlarından biri, farklı yazılım ve donanım konfigürasyonları, ağ bağlantı türleri ile yazılım uyumluluğunun doğrulanmasıdır. Beta testinin bir diğer amacı, yazılımın kullanılabilirliği ve işlevselliği hakkında kullanıcılardan geri bildirimler almaktır.

Beta testi, yazılım sisteminin ilk adımı olan gereksinimlerin toplanması aşaması ile bu gereksinimlerin uygulanması arasındaki eksikliklerin belirlenmesine yardımcı olur. Beta testi sayesinde müşterilerin dönütleri ile ürün kalitesi arttırılır. Müşterilerin geri bildirimine veya önerilerine göre test döngüleri artabilir. Buna bağlı olarak test süreci uzayabilir. Alfa testinin aksine, Beta testi tamamen kara kutu test tekniğiyle gerçekleştirilir.

5.4.3. Gama Testi

Yazılımın tamamlanmış sürümünden önce gerçekleştirilen son aşamada yapılan testtir. Sistemi, belirlenen tüm gereklilikleri dikkate alarak piyasaya sürülmesi için hazırlar. Gama testinin odak noktası sistemin güvenliği ve işlevselliğidir. Yalnızca sınırlı sayıda kullanıcı tarafından gerçekleştirilir. Gama testinde tespit edilen hatalar, yüksek öncelik ve önem derecesine sahip olmadığı sürece sistemde bir değişiklik gerçekleştirilmez. Sistemin yalnızca belirli noktaları test edilir. Yani test sırasında gerçekleştirilen kontrol, ürünün tamamında değil sadece belirli özelliklerin doğrulanmasını içerir. Fark edilen eksiklik veya hataların düzeltilmesi, sisteme sonraki sürümler için güncelleme olarak yansır. Bu aşamada, düzeltilmesi gereken kritik veya ölümcül hatalar dışında ürün üzerinde herhangi bir değişikliğe izin verilmez. Tüm bunların yanı sıra, sınırlı bir geliştirme ve kontrol döngüsüne sahip olması nedeniyle Gama Testi genellikle göz ardı edilen bir testtir.

6. Yazılım Test Türleri

Test süreci, yazılımın kaliteye ulaşması ve kullanıcı isteklerine uygunluğunun ölçülmesi için gerçekleştirilen bir süreçtir. Web, masaüstü ve mobil tabanlı yazılım uygulamalarının birçok farklı test türleriyle ölçülmesi mümkündür. Ancak her yazılım için bütün test türlerinin uygulanması mümkün değildir. Her yazılım önemlidir, ancak birbirlerine kıyasla önem derecesi aynı değildir. Bu nedenle test işlemlerinde yazılımın kapsamı, kullanıcı hedef kitlesi, kullandığı veri, büyüklüğü gibi birçok önemli husus dikkate alınarak test türleri uygulanmaktadır [50] [58] [59] [60]. Çok fazla sayıda ve çok çeşitliliğe sahip olan test türlerini hiyerarşik yapıda sınıflandırmanın mümkün olmadığı bu tez çalışmasında yapılan araştırmalar sonucunda görülmüştür. Her ne kadar literatürde test türlerinin sınıflandırılması şekiller veya tablolar şeklinde sunulmuş olsa da yapılan incelemeler sonucunda birçok test türünün dahil edilmediği tespit edilmiştir. Çünkü bir test hem İşlevsel hem de İşlevsel Olmayan Testler kategorisinde yer alabilmektedir. Bu nedenle yazılım dünyasında sıkça kullanılan test türleri, tanımları ve uygulanan kısımlar Tablo 6.1.'de sunulmaktadır.

Tablo 6.1. Yazılım test çeşitleri [61], [62], [63], [64], [65]

Test Adı	**Tanımı**	**Uygulandığı Kısımlar**
Arayüz Testi	İki veya daha fazla sistemleri arasındaki iletişimin sağlıklı bir şekilde gerçekleştirilmesini, yazılım modülü arasındaki bağlantıyı doğrulamak için gerçekleştirilir. En önemlilerden biri web ve uygulama sunucusunun arayüzü, diğeri ise uygulama ve veri tabanı sunucusunun arayüzüdür.	Veri akışı, Modüller arası iletişim, Modüller arası bilgi alışverişi.
Çapraz Tarayıcı Testi	Genellikle web siteleri gibi dinamik olan ve internet bağlantısı gerektiren sistemler için farklı tarayıcılardaki çalışma durumları test edilir.	Temel işlevler, Grafiksel kullanıcı arayüzü, İşlevin yanıt verme süresi ve çalışma hızı.
Çevik Test	Çevik yazılım geliştirme metodolojileri ile geliştirilen uygulamaları sınayan testtir. Bu test, yazılımın geliştirilmesi ve test edilmesini aynı anda gerçekleşmesini sağlar, zaman ve maliyet kaybını önler. Çevik test, geliştirme sürecine düzenli olarak geribildirim sağlar.	Sistem gereksinimlerinin testi, Modül testleri, Entegre modüllerin testleri.
Dayanıklılık Testi	Performans sorunlarının tespit edilmesi amacıyla uzun süre büyük değerlerde veri girişi kullanılarak olası hataların ortaya çıkması amacıyla uygulanır.	Geçersiz veri girişi, Yoğun şekilde veri girişi.
Depolama Testi	Sistemin veri tabanı test edilir. Verilerin uygun dizinlerde saklanması, yetersiz disk alanlarının hangi şartlarda ortaya çıktığı incelenir.	Veri formatları, Veritabanı tabloları.

Dokümantasyon Testi	Sistemin oluşturulması, incelenmesi, geliştirilmesi ve genişletilebilmesi için gerekli tüm belgelerin doğrulanması amacıyla gerçekleştirilir.	Gereksinim raporları, Diyagramlar, Test planları, Test senaryoları, Test sonuçları, Çizelgeler.
Erişilebilirlik Testi	Uygulamanın renk körlüğü, işitme engeli gibi özelliklere sahip bireyler tarafından erişilebilirliğini ve kullanım kolaylığını test etmek için şartlara uygun bireyler tarafından gerçekleştirilmesi gereken bir yazılım testidir.	Farklı özelliklere sahip bireyler için arayüzlerin kullanım kolaylığı ve arayüzler arası geçişler.
ETL Testi	Extract-Transform-Load testi, veri hareketlerinin, veri girdi ve çıktılarının, arayüzler arasındaki tablo ilişkilerinin doğrulanması için raporlanan testtir.	Verilerin kaynaktan hedefe hareketi, Veri girdi-çıktıları, Arayüzler arasındaki tablo ilişkileri.
Eşzamanlılık Testi	Sisteme birden fazla kullanıcının giriş yapması durumunda sistemin davranışı incelenir ve var olan hatalar tespit edilir.	Kullanıcılara verilen yanıt süresi, Çıktıların doğruluğu, Görev paylaşımı.
Güvenlik Testi	Geliştirilen sistemin, herhangi bir açıklık veya tehlikeden ötürü erişim sorunu yaşanmasını veya büyük oranda veri kaybına neden olmasını engellemek amacıyla gerçekleştirilir.	Ağ güvenliği, Erişim izinleri, Verilerin izinsiz kişilerce yayılması ve aktarılması.
Hacim Testi	Sistemin büyük miktarda veri girişine karşı gösterebileceği davranış ve yanıt verme süresinin ölçümünde kullanılır.	Yoğun veri girişi sırasında form işlemleri, Veri girişine karşılık yanıt verme süresi.
Hata Enjeksiyon Testi	Olası hataların ortaya çıkabilmesi, hatanın boyutlarının ve risklerinin fark edilmesi, sistemin hata durumunda davranışının görülebilmesi için kod bloğuna hatalı kodların yerleştirildiği bir test türüdür.	Modül içerisinde yapay hata oluşturmaya bağlı sistem davranışı.
Performans Testi	Donanım ürünleri yardımıyla sistemin performansını etkileyecek olumsuzlukları en aza indirgemeyi amaçlar. Hata tespiti için değil, sistemin verimliliği test edilir.	Çalışma hızı, Kapladığı alan, Eşzamanlı kullanıcı sayısı, Batarya kullanımı, Ortalama yanıt verme süresi, CPU kullanımı.
Rampa Testi	Sistemin sınırlarını kullanıcı veya veri girişi açısından zorlayarak maksimum kapasitenin belirlenmesini sağlar.	Eşzamanlı kullanıcı girişi, Form girişi, Veritabanı kayıtları.
Stres Testi	Yazılım ve donanımın birlikte çalışması durumunda sistemde oluşabilecek kırılma noktalarının belirlenmesini amaçlar.	Form girişi, Veritabanı kayıtları.
Yük Testi	Yazılımın, aynı anda birden fazla kullanıcı girişi durumunda göstereceği davranışları inceler. Performans testinin alt başlığı olarak ifade edilir.	Eşzamanlı kullanıcı girişi, Eşzamanlı modül kullanımı.

Senaryo Testi	Farklı senaryolara bağlı kalarak büyük ve karmaşık yapıdaki yazılımların tüm modüllerinin test edilmesidir.	Senaryo baz alınarak modüller arasındaki geçiş, Veri alışverişi, Arayüzler, Kullanıcı girdileri.
Yol Testi	Çoklu giriş ve çıkışa sahip karmaşık sistemlerde bulunan döngülerin veya koşullu ifadelerin tüm yollarının test edilmesi için kaynak kodun incelendiği bir test çeşididir.	Tekrarlı, koşullu ve döngüsel ifadelerdeki tüm olası durumlar.
Regresyon Testi	Hataların bulunduğu modüllerde çözümleme işleminden sonra gerçekleştirilir. Yazılımın kod değişikliğinden etkilenmediğinin doğrulanması için oluşan aksaklıkları test edilip onarılmasıyla gerçekleştirilir.	Hatanın çözümlendiği modüllerdeki işleyişler, Değişiklik yapılan modülün bağlı olduğu modüller arasındaki iletişim ve veri alışverişi.
Kullanılabilirlik Testi	Kullanıcı-sistem etkileşimi sırasında kullanıcı davranışlarını ve duygusal tepkilerini gözlemleyerek yazılımın kolay kullanılmasını ve müşteri memnuniyetinin arttırılmasını sağlar.	Arayüzlerin kullanım kolaylığı Arayüzler arası geçişler.
Küreselleşme Testi	Sistemin, herhangi bir kültürel ya da yerel ortama adapte olma durumunu test etmek için kullanılır. Sistemi, bölgesel özellikler bazında test etmeye yarar.	Dil, Bölge ve kültür, Saat, Tarihi gelişmeler.
Veri tabanı Testi	Veri tabanının verimliliğini, maksimum kararlılığını, performansını ve güvenliğini sağlamak için veriler arasındaki ilişkilerin kontrol edildiği test türüdür.	Veri tabanı girişleri, Verilerin doğruluğu ve tutarlılığı, sorguların doğru çıktıyı üretmesi.
Kova Testi	Genellikle web tabanlı sistemlerde gerçekleştirilir. En az iki farklı arayüz varyasyonları arasından kullanıcı kriterlerine en uygun olanı seçmek için gerçekleştirilir.	Yazı tipleri, Bilgi grafik görüntüleri, Renk düzenleri, Buton konumları ve tasarımları, Form ekranları.
Fuzz Testi	Fuzz testinde sistem beklenmedik, geçersiz veya rastgele veri girdileriyle sınanır. *Bulanık test* adıyla da bilinen yöntemde, işletim sistemleri veya ağ sistemlerinde bulunan hatalı kodlar ve güvenlik açıklarını bulmak hedeflenir.	Veri girişi nedeniyle olası çökmeler, Bellek sızıntıları, güvenlik açıkları, Sistem kırılmaları, Hata mesajları.
Kurtarma Testi	Sistemin, olumsuz bir durumdan sonra normal çalışma düzenine dönmesi için gereken işlem ve sürenin belirlenmesi amacıyla gerçekleştirilir.	Sistemin normale dönme hızı ve döndükten sonraki sağlıklı çalışma oranı, Normale dönmesi için gereken işlemler.

Kurulum Testi	Yazılım sisteminin tüm özellikleri ve donanımsal araçlarıyla birlikte doğru bir şekilde kurulduğunu veya tüm bileşenleri ile kaldırıldığını kontrol etmek için gerçekleştirilir.	Ağ sistemleri, Donanım aygıtları, İşletim sistemleri.
Uyumluluk Testi	Bu testin gerçekleştirilmesindeki esas amaç, sistemin her türlü platform, konfigürasyon, donanım ve yazılıma gelebilecek eklentilerle uyum içinde başarılı bir işleyişe sahip olduğundan emin olmaktır.	Donanımsal aygıtlar, Ağ sistemleri, İşletim sistemleri, Yazılım sürümleri, Farklı tarayıcılar.
Yük Devretme Testi	Bu test türünde, sistemde farklı nedenlerden dolayı olası kırılma durumuna karşı verilerin veya gerçekleştirilen işlemlerin yedeklenmesi durumu kontrol edilir.	Sunucu, Form işlemleri, Veri girişleri.
Gereksinim Testi	Yazılım yaşam döngüsünde belirtilen gereksinim aşamasının kullanıcı kriterlerine uygunluğun test edilmesidir.	Gereksinimlerin tam, net, doğru anlaşılabilir, tutarlı ve test edilebilir olması.
Uçtan Uca Test	Uygulamanın bağımlı olduğu noktalarla beklenen çıktıya ulaşmasını kontrol etmek ve verilerin modüller arasında doğru şekilde iletilmesini sağlamak için ilk adımdan itibaren olası tüm son adımların takibinin yapıldığı test türüdür.	Uygulamanın diğer sistemler ile olan entegre arayüz bağlantıları, Bağlı olan uygulamalarla doğru veri alışverişi, Veri tabanları arasındaki bağlantı.
Riske Dayalı Test	Olası hataların önlenmesi yada geçmişte sürekli tekrarlanan hatanın ortadan kaldırılması için hata olasılığı bulunan modüllerin her bir fonksiyonunun, çeşitli verilerle en ince ayrıntısına kadar test edilmesidir.	Hata olasılığı barındıran modüllerde bulunan veritabanı işlemleri, modüllerin form işlemleri, Çözümlenen hata sonrası modüller arası entegrasyon.
Onay Testi	Hataları çözümlenen sistemin, kullanıcı isteklerine karşı uygunluğu, sistem geliştirilirken hazırlanan taslak ve şemaların sistematik ve düzenli oluşu onaylanır. Bu test kusurların giderilmesinden sonra işlevselliğin başarısını doğrulamayı hedefler.	Çözülen hatanın bulunduğu modüldeki işlevsel görevler, testler sonucu hazırlanan raporların işleyiş ve basamaklarının gerçekliği.
İş Akışı Testi	Yazılım sisteminin iş akışını, modüllerin bağlantısını test eder. Uygulama içerisinde hedeflenen çıktıyı elde etmek veya istenilen modüle ulaşılabilmek için gerekli olan işlem dizininin doğruluğu test edilir.	Modüller arası geçişler, Sistemin hiyerarşik yapısı, Modüller arası veri akışı.

Kullanıcı Arabirim Testi	Sistemin grafik arayüzlerinin kullanıcı ve test ekibi tarafından test edilmesidir. Amaç, uygulamanın hiyerarşik ve grafiksel ön yüzünün test edilmesi, farklı cihazlarda grafiksel görünümün sabitliğinin test edilmesidir.	Menüler, Butonlar, Görsel destek elemaları, Simgeler, Sayfalar arasındaki akış.
Karşılaştırma Testi	Mevcut yazılım ve gelecekteki yazılım arasındaki iyileştirilmelerin belirlenmesi amacıyla piyasada bulunan aynı amaçla kullanılan farklı yazılımları belirli kriterler açısından kıyaslar.	Dokümanlar, Veritabanı içerikleri, Metrik değerleri.
Taşınabilirlik Testi	Sistemin farklı platformlar ve ortamlarda esnek kullanılabilirliği, her ortamda uygun şekilde ve fazla kaynağa ihtiyaç duymadan çalışabilir olmasının test edildiği yöntemdir. Test işlemi sırasında uygulamanın farklı bir fiziki ortama taşınması karşısındaki davranışları gözlenir.	Farklı yazılım ya da donanım kaynaklarıyla beraber çalışabilirlik, Farklı tarayıcılar, Farklı ağ sistemleri.
Konfigürasyon Testi	Sistemin çalışabileceği en uygun yazılım sistemleri ve donanım araçlarının belirlendiği test türüdür. Bu testin yapılması sırasında bir çok yazılım ve donanım kombinasyonu gerçekleştirilir.	Sistemin çalışması için en uygun işletim sistemi, Web tarayıcısı, En uygun işlemci hızı gibi teknik bilgiler.
Uygunluk Testi	Uygunluk testi, sistemin büyük ve bağımsız kuruluşlar tarafından tanımlanan *IEEE*, *W3C* ya da *ETSI* gibi büyük ve bağımsız kuruluşlarca belirlenen standartlara uygunluğunun uzman bir test ekibi tarafından kontrol edilmesidir.	Sisteme ait doküman, şartnameler ve gereksinim belgelerinin uygunluğu, Sistemin belirlenen standartlara uygunluğu.
Tahribat Testi	*Yıkıcı test* olarak da bilinir. Öngörülmeyen kullanıcı davranışları üzerinde yoğunlaşılarak, yazılımın yanlış şekilde kullanılması durumunda nasıl bir davranış izleyeceği test edilir. En zorlu çalışma koşulları yardımıyla gerçekleştirilen test süreci, uygulama kesilinceye kadar devam edilir.	Sistem kesintiye uğrayana kadar devam eden veri girişi, Geçersiz veri girişi, Hedeflenen işlem için farklı adımların seçilmesi.
Goril Testi	Bir modülün, işlevselliğini kontrol etmek için çok sayıda girdi kombinasyonu yardımıyla kapsamlı bir şekilde test edilmesidir. Sistemin, rastgele girdiler gibi stresli koşullar altındaki davranışı değerlendirilir.	Form girişleri, Kullanıcı girişleri, Kullanıcı bilgileri.
Maymun Testi	*Rastgele test* olarak da bilinen bu test türünde, yanlış veya hatalı bilgilere karşı sistemin performansı ve davranışı test edilir. Herhangi bir test senaryosu olmadan gerçekleşen testte farklı eylemlere karşı sistemin çökme durumu gözlemlenir.	Form girişleri, Beklenen ve girilen veri formatları, Kullanıcı bilgileri.

Paralel Sınama Testi	Farklı iki yazılım/sürüm veya test senaryosu üzerinde testlerin paralel olarak gerçekleştirilmesini amaçlar. Genellikle sürümler arasındaki farkların ve güncel sürümün performans kriterlerine uygunluğunun test edilmesinde kullanılır.	Sistemin farklı tarayıcılardaki davranışları, Farklı sürümlerdeki veri formatları arasındaki tutarlılık.
Yeterlilik Testi	Yazılımın kullanıcı ihtiyaçlarını ne düzeyde karşıladığını araştıran son aşama test türüdür.	Entegrasyon sonrası modüller arasındaki iletişim, Arayüz tasarımları.

6.1. İşlevsel Testler

Literatürde *Fonksiyonel Testler* olarak da bilinir. Bu testlerde sistemin her bir fonksiyonunun gereksinimlere ve isteklere uygunluğu test edilir. İşlevsel Testlerde uygulamanın kod kısmına bakılmaz. Esas olarak Kara Kutu tekniği ile uygulanır. Girilen veriler sonucunda, beklenen çıktıların tutarlılığı kontrol edilir. Yazılım sisteminin her bir fonksiyonunun gereksinimlere uygun çalışması ikkate alınır. Test işlemi sırasında dikkat edilen en önemli unsur, müşteriler tarafından daha önce belirlenen işlevsel kriterlerdir. Böylelikle sistem işlevlerinin kullanıcı beklentisini karşılama durumu test edilir.

İşlevsel testte amaç, üründe var olan hataların ortadan kaldırılması değildir. Fonksiyonel ya da İşlevsel Testte dikkat edilen husus, ürünün kullanıcı isteklerini karşılaması sonucu kaliteli bir ürün elde etmektir. Söz konusu testte, sistemin kullanıcı gereksinimlerine uygunluğu dikkate alınır. Bu nedenle kodda var olabilecek kısmi işlevsellik kayıpları gözden kaçabilir. Uluslararası Yazılım Test Yeterlilikler Kurulu'nun (ISQTB), 2018-2019 yılında yayınlanan raporunda şirketler tarafından kullanılan test türleri sınıflandırılmıştır. Bu sınıflandırmaya göre, şirketler tarafından en çok kullanılan test türleri arasında %92 oranla ilk sırada işlevsel test gelmektedir. Yüzdelik dilimde en çok kullanılan 16 test türü arasına giren fonksiyonel testini, en yakın %64 oran ile performans testi ve sonrasında %62 oran ile kullanılabilirlik testi takip etmektedir [66].

6.2. İşlevsel Olmayan Testler

Fonksiyonel Olmayan Testler olarak da isimlendirilir. İşlevsel olmayan yazılım testlerinde, bir yazılım sisteminin performans, kullanılabilirlik, güvenlik gibi işlevsel olmayan özellikler test edilir. İşlevsel olmayan testler yapılarak, ürünün kullanılabilirliği, verimliliği, sürdürülebilirliği, taşınabilirliği arttırılır. Ürüne ait işlevsel olmayan özelliklere bağlı üretim riski ve maliyetin azaltılmasına yardımcı olur.

7. Yazılım Kalite Güvencesi

Yazılımda kalite, yazılımın kullanıcı kriterlerine ve kullanım amaçlarına uygun olarak yazılım geliştirilmesi ve müşteri ihtiyaçlarını tamamen karşılayabilmesidir. Kısacası kalite kavramı, müşteri isteklerine maksimum düzeyde yanıt verebilmeyi içerir. Yazılım kalitesini tanımlayan başka faktörler de vardır. Yazılım kalitesinin güvence altına alınması, yazılım yaşam döngüsü süreçlerindeki her bir evrenin ciddi ve profesyonel işletilmesi ile mümkündür. Bu bağlamda, planlama aşamasından yayımlama aşamasına kadar her evre, her süreç ve her işlem önemle göz önünde bulundurulmalıdır.

Yazılım kalitesi sistemin, çeşitli yönler ve belirli ölçütler dikkate alınarak gerçekleştirilir. Yazılımlarda var olan hataların bulunması ve düşük zaman-maliyet eşliğinde çözümlenmesi, bakım ve yapılabilirliği, istatistiksel süreç ve kontrolü, test edilebilirliğin değerlendirilmesi, tasarım hata tespiti gibi yazılım iyileştirme adımları, yazılım kalitesinin belirlenmesinin amaçları olarak sıralanabilir [10]. Yazılım sistemleri için kalite kavramı, birbirinden farklı ölçüt ve unsurlardan oluşur. Yazılım kalitesinin ölçülmesinde kullanılan ölçütler ile maliyet-zaman, kullanılabilirlik, sürdürülebilirlik gibi unsurların maksimum düzeyde verimli olması hedeflenir. Yazılım kalitesinin belirlenmesinde kullanıcı isteklerini doğru ve geçerli bir şekilde yerine getirmenin yanı sıra birçok farklı unsur bulunmaktadır. Bu unsurlar test ölçütleri adı altında proje ölçütleri, süreç ölçütleri ve ürün ölçütleri şeklinde üç farklı kategoride sınıflandırılır. Kategorize edilen test ölçütlerinden ürün ölçütleri, kod bilgilerine bakılarak alt sınıflara ayrılarak, sisteme ait kalite oranını rakamsal biçimde ifade eder.

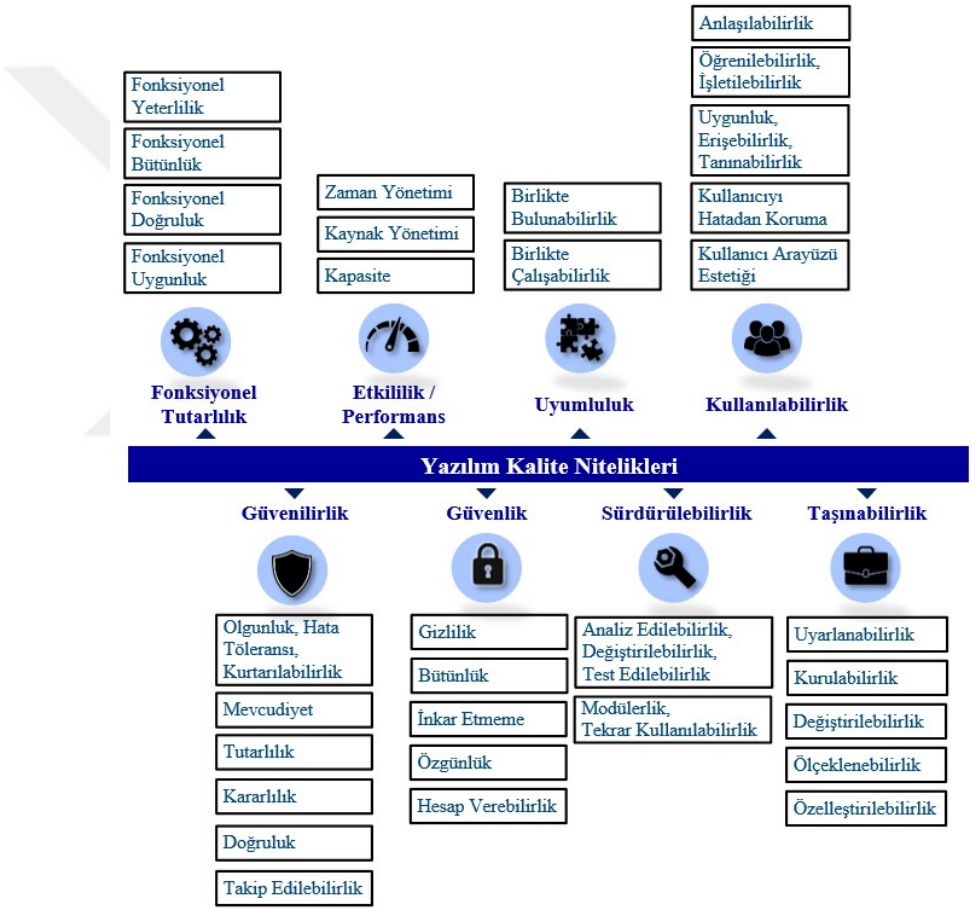
Günümüzde kullanılan ve ihtiyaç duyulan yazılımlar, yüksek beklenti ve düşük maliyet gibi artan kullanıcı talepleri çerçevesinde şekillenmektedir. Bu nedenle bu yazılımlar, genellikle büyük çaplı projelerin çatısı altında geliştirilmektedir. Projenin büyümesi iş yükünün artması anlamına gelmektedir. İş yükünün dağıtılması adına proje ekibinin büyütülmesi kaçınılmaz olmaktadır. Bu da projenin yönetilmesini zorlaştıran bir durumdur [67]. Projenin sağlıklı yürütülebilmesi, geliştirme aşamasında bazı kritik durumların göz önüne alınmasıyla mümkün olabilir. Bu kritik durumlara örnek olarak; geliştirilen yazılımın maksimum verim ve doğrulama ile çalışması, minimum düzeyde hata içermesi, düşük maliyete sahip olması, kodun kolay anlaşılabilirliği, sistemin kolay kullanılabilirliği ve güncellenebilir olması verilebilir [68]. Yazılım projelerinin uzman olmayan tecrübesiz kişiler tarafından yönetilmesi sonucunda aşağıdaki aksamalar ortaya çıkmaktadır.

- Proje bütçesi denkleştirilmeyebilir.
- İhtiyaç duyulan yazılımsal ve donanımsal kaynakların temin edilemeyebilir.
- Proje için ihtiyaç duyulan nitelikli ve uzman personel belirlenemeyebilir.
- Sonradan tespit edilen büyük hatalar büyük maddi kayıplara neden olur.

- Değişen ve gelişen kullanıcı istekleri doğrultusunda yazılım güncellemeleri mümkün olmayabilir. Sonuç olarak geliştirilen yazılımın kullanım ömrü kısa olur.
- Yazılım projelerinde kullanılan teknoloji ve yöntemler yanlış seçilebilir.

7.1. Yazılım Kalite Nitelikleri

Yazılımda kalite, yazılımı çeşitli yönleriyle inceleyerek, her yönüyle ortaya çıkan ürünün belirli standartlara sahip olmasını amaçlar. Yazılımda kalite nitelikleri; fonksiyonel tutarlılık, performans, uyumluluk, kullanılabilirlik, güvenilirlik, güvenlik, sürdürülebilirlik ve taşınabilirlik gibi önemli unsurları kapsamaktadır. Bu çalışmada yazılım kalitesi Şekil 7.1.'de gösterildiği gibi 8 ana başlık altında incelenmiştir.



Şekil 7.1. Yazılım kalite nitelikleri

7.1.1. Fonksiyonel Tutarlılık

Yazılımın içerdiği tüm fonksiyonellerin incelenmesi ve belirli standartlara göre incelenerek kontrol edilmesidir.

- *Fonksiyonel Yeterlilik:* İhtiyaç analizinde belirlenen tüm durumlar göz önüne alınarak, yazılım için gerekli tüm fonksiyonlar belirlenir. Geliştirilen yazılımın da bu belirlenen

tüm fonksiyonları içermesidir.

- *Fonksiyonel Bütünlük:* Büyük yazılım projeleri farklı geliştirici gruplar tarafından küçük modüller halinde geliştirilip entegre edilir. Birleştirme sürecindeki tüm modüllerin birbirleriyle uyumlu ve problemsiz çalışmasıdır.
- *Fonksiyonel Doğruluk:* İhtiyaç analizinde yer alan fonksiyonların doğru çalışması için yapılan testlerdir.
- *Fonksiyonel Uygunluk:* Yazılıma ait fonksiyonların kullanıcı isteklerine uygunluğu test edilir. Eksik özelliklere sahip veya kullanıcı tarafından istenmeyen özellikleri barındıran yazılımlar uygunluk bakımından yetersiz kalmaktadır.

7.1.2. Performans

Yazılımın kullandığı kaynak miktarı, çalışma süresi ve etkili bir şekilde kaç kişiye kadar hizmet verebileceği gibi faktörler performans adı altında incelenmektedir.

- *Zaman Yönetimi:* Yazılımın kalitesi söz konusu olduğunda, zaman yönetimi yazılımda yer alan fonksiyonların çalışma sırası ve süresi gibi etmenleri kapsar. Bu kapsamda paralel çalışabilecek fonksiyonların veya sırayla çalışması gereken fonksiyonların belirlenmesi önemlidir. Ayrıca yazılımın kısa sürede çalışması ve kullanıcıya kısa sürede cevap vermesi tercih edilir.
- *Kaynak Yönetimi:* Kaynak yönetimi, yazılımların daha geniş kitleler tarafından kullanılabilmesi için önemlidir. Kaynak yönetiminde, zaman yönetimi ve kaynak yönetimi birbirleriyle bağlantılıdır. Kaynak yönetimi, yazılımın kullanmış olduğu sabit disk ve bellek(RAM) gibi bilgisayar bileşenlerinin daha etkili kullanılmasını amaçlar. Yazılımlar, daha az kaynak ihtiyacı gerektirdiğinde, çok daha fazla kullanıcı tarafından tercih edilecektir.
- *Kapasite:* Geliştirilen uygulama çok kullanıcı tarafından kullanıldığında bile problemsiz bir şekilde çalışmaya devam etmesi gerekir. Yük testleri ile bu başarımlar ölçülebilir.

7.1.3. Uyumluluk

Uyumluluk yazılım açısından iki farklı şeyi ifade edebilir. Birincisi, diğer yazılımlar ile çalışabilirlik, ikincisi ise sürüm farklarından kaynaklanan problemlerin minimuma indirgenmesidir.

- *Birlikte Bulunabilirlik:* Farklı yazılımların bilgisayarda aynı anda bulunabilmesidir. Günümüzde işletim sistemleri birlikte çalışabilirlik durumlarını kontrol ettiğinden dolayı, bu tarz problemlerle daha az karşılaşılmaktadır.
- *Birlikte Çalışabilirlik:* Farklı yazılımların aynı anda çalışabilmesidir. İşletim sistemleri kaynak kontrolünü elinde bulundurduğu için bu kapsamdaki problemler

azalmaktadır. Ancak yine de ortak kaynak kullanımını (statik soket numarası gibi) gerektirecek şekilde geliştirilen yazılımlarda bu tarz problemler ortaya çıkmaktadır. Birlikte çalışabilirlik farklı sürümlerin aynı anda çalışabilirliğini de ifade etmektedir. Yeni versiyon çıktığında, eski versiyona sahip bilgisayarlarda da yazılımın bozulmadan kullanılabilmesi gerekir.

7.1.4. Kullanılabilirlik

Yazılımın kullanıcı tarafından kolay kullanılabilmesini amaçlayan kriterler bu başlık altında yer alacaktır.

- *Anlaşılabilirlik:* Arayüzün basit, sade ve anlaşılabilir olması gerekmektedir. Bu kapsamda menülerin uygun yerlere yerleştirilmesi, gerekli açıklamaların yapılması ve kullanılacak ikonların anlaşılır olmasına dikkat edilmelidir.
- *Öğrenilebilirlik-İşletilebilirlik:* Ortaya çıkan arayüzler kolay öğrenilebilir olmalıdır. Arayüzler, kullanıcı alışkanlıklarına cevap verir nitelikte olmalıdır. Bunun için alışılmışın dışında yerlere menüler eklenmesi gibi anormal konumlandırmalardan uzak durulmalıdır.
- *Uygunluk-Erişilebilirlik-Tanınabilirlik:* Ortaya çıkan yazılımların kullanıcılara uygun şekilde dizayn edilmesi gerekmektedir. Kullanıcıların engelli bireyler olması durumunda, yazılımın erişebilir olması büyük önem taşımaktadır.
- *Kullanıcıyı Hatadan Koruma:* Yazılımda kodlama zamanında belli olmayan ancak çalışma zamanında ortaya çıkan hatalar yer almaktadır. Bu hatalar işletim sisteminin o an kaynakları başka bir uygulama için ayırması, işletim sisteminin dosya okuma-yazma işlemine izin vermemesi veya kullanıcının beklenmeyen bir değer girmesi gibi çeşitli sebeplerden kaynaklanabilir. Yazılımın bu tarz hataları tolere ederek çökme yaşamadan çalışmaya devam etmesi gerekmektedir.
- *Kullanıcı Arayüzü Estetiği:* Her ne kadar çoğu zaman göz ardı edilse de kullanıcı arayüzü estetiği kullanıcıların yazılımı kullanmasını belirleyen önemli etkenlerden biridir. Bu noktada renk seçimi, görsel tasarım gibi işlemler uygun bir şekilde yapılarak, yazılımın her yerinde ortak bir arayüz izlenimi uyandırılmalıdır.

7.1.5. Güvenilirlik

Güvenilirlik yazılımın kullanılması açısından göz önünde tutulması gereken önemli bir faktördür. Güvenilirlik konusu, güvenlik konusundan tamamen farklıdır. Geliştirici firmanın kullanıcıya sağladığı olanaklar, satış sonrası destek, garanti süresi güvenilirlik kriterlerini ön plana çıkarmaktadır. Yazılım sisteminin kullanıcı tarafından güvenilirliğinin sağlanması için aşağıdaki hususları karşılaması beklenir.

- *Olgunluk-Hata Toleransı-Kurtarılabilirlik:* Yazılımın olası çökmelere ve hatalara karşı veri kaybının önlenmesidir. Geliştirilen yazılımların otomatik kaydetme veya

kayıt tutma(loglama) gibi özelliklere sahip olmalı, olası hatanın geri alınmasını sağlanmalıdır.

- *Mevcudiyet:* Bir yazılıma karşı duyulan güvenilirliğin artması için o yazılımın yayınlanmasının sürekli ve kullanılabilir olması gerekmektedir. Bir süre sonra yayından kaldırılan veya çalışamaz duruma gelen yazılımlar güvenilir değildir.
- *Tutarlılık:* Tutarlılık zamandan ve mekandan bağımsız olarak aynı girdiler için aynı çıktının üretilmesi anlamına gelmektedir.
- *Kararlılık:* Bir yazılımın en az hata ile problemsiz çalışma durumudur. Sürekli çöken yazılımlar veya sürekli hata veren yazılımlar Beta test aşamasında anlayışla karşılanabilirken, kararlı sürümlerinde tahammül edilemez olmaktadır.
- *Doğruluk:* Yanlış sonuç üreten veya kullanıcıyı yanlış yönlendiren yazılımlar kullanıcıya güven vermezler. Doğruluk kavramı da düzenli gerçekleştirilen testler ile sağlanabilir.
- *Takip Edilebilirlik:* Yapılan işlemlerin log kaydının alınması takip edilebilirlik açısından önemlidir. İlerde karşılaşılacak olası hata durumunda veya bir problemde bu loglar yardımı ile hangi kullanıcının hangi işlemi ne zaman gerçekleştirdiği gibi bilgilere ulaşılabilir. Bu sayede sistemin geri alınabilirliği ve kurtarılabilirliği arttırılmış olur.

7.1.6. Güvenlik

Kullanıcı bilgilerinin korunması, sistemin açık ve şeffaf bir şekilde çalışması gibi faktörler güvenlik başlığı altında incelenebilir.

- *Gizlilik:* Kullanıcı verilerinin kullanıcı isteği ve onayı dışında başka kişilerle paylaşılmamasıdır. Ayrıca bilgilerin korsan saldırılara karşı korunması ve olası saldırılar karşısında sistemin durumu gibi konular gizlilik kapsamı altında ele alınmalıdır.
- *Bütünlük:* Yazılım sistemlerinin bir bütün halinde güvenli olmasını amaçlar. Yazılımda bütünlük, kavramı, sistemde bulunan verilerin ve sistemde dönen işlemlerin birbirleriyle uyumlu olması şeklinde tanımlanabilir.
- *İnkar Etmeme:* Yazılımın kullanıcıya karşı şeffaf olmasını gerektirir. Sistemde meydana gelen kritik değişiklikler ya da hatalar kullanıcı bilgisine sunulmalıdır.
- *Özgünlük:* Bu kavram, yazılım geliiştiricilerin mümkün olduğunca farklı yaklaşımlar kullanarak proje gerçekleştirmeleridir. savunmaktadır. Örnek verilecek olursa; hazır şablonlar kullanmak yerine, yazılım geliştiricilerin kendine özgü şablon veya desen geliştirmeleridir.

- *Hesap Verebilirlik:* Yazılım sisteminde meydana gelen değişikliklerin kayıt altına alınması ve gerektiği zamanda sistemi ihlal eden davranışların kimin tarafından gerçekleştirildiğinin tespit edilmesi gerekir. Takip edilebilirlik maddesi ile ilişkilidir.

7.1.7. Sürdürülebilirlik

Yazılımların kullanıma açılmasının ardından kullanıcı istekleri doğrultusunda güncellenebilme özelliğinin devam etmesidir.

- *Analiz Edilebilirlik-Değiştirilebilirlik-Test Edilebilirlik:* Yazılımların sürdürülebilir olması için yapılacak güncellemelerin sisteme kolayca entegre edilebilmesi gerekir. Bu sebeple yazılımın dinamik alt yapıya sahip olması önemlidir.

- *Modülerlik – Tekrar Kullanılabilirlik:* Farklı ekipler tarafından geliştirilen modüler yazılımların farklı projelerde veya uygulamalara entegre edilebilmesi veya modüllerin yeniden kullanılmasıdır. Ayrıca modüler yazılım altyapısı sayesinde yazılan kodların (fonksiyonların) başka işleri gerçekleştirmek için ortak olarak kullanılabilmesi mümkündür.

7.1.8. Taşınabilirlik

Ortaya çıkan yazılımın farklı ortamlarda rahat bir şekilde kullanılması ve kullanıcı isteklerine cevap verebilmesidir. Yazılımın taşınabilir olması yazılım altyapısının esnek olmasıyla mümkündür.

- *Uyarlanabilirlik:* Geliştirilen yazılımın benzer modüllerde olması ve farklı ihtiyaçlara karşılık verebilmesidir. Örneğin kafe yazılımının, restoran sipariş yönetimi yazılımı olarak uyarlanması söyleenebilir.

- *Kurulabilirlik:* Yazılım farklı bilgisayarlara pratik bir şekilde kurulabilir olmalıdır.

- *Değiştirilebilirlik:* Sürdürülebilirlik başlığı altında da geçen bu madde, yazılımdaki değişikliklerin ve güncellemelerin pratik bir şekilde yapılabilmesidir. Bu özellik sayesinde yazılımın uyarlanabilir olması da artar.

- *Ölçeklendirilebilirlik:* Yazılımın dinamik bir altyapıya sahip olmasını gerektirir. Örneğin restoran otomasyonu için masa sayısının kullanıcı tarafından belirlenmesi ölçeklendirilebilirliğe örnek olarak gösterilebilir. Yazılımı kullanacak kişi sayısının belirlenmesi de bir ölçeklendirme örneğidir.

- *Özelleştirilebilirlik:* Geliştirilen yazılımlar, kullanıcı talepleri doğrultusunda özelleştirilebilir olmalıdır. Farklı restoranların kendi menülerini özelleştirmelerine izin verilmesi yazılımın özelleştirilebilirliğidir.

7.2. ISO/IEC-9126 Kalite Standardı

Yazılımlar farklı bilgisayar donanım ve yazılımının kullanıldığı ortamlara aktarılabilir olmalıdır. Yeni işletim sistemi sürümlerinin kullanılması, bilgisayar değiştirilmesi gibi durumlarda, daha önce geliştirilmiş olan yazılım en az çaba ile tekrar kullanılabilmelidir. ISO/IEC-9126 Kalite Standardı, 6 temel faktöre göre yazılımı değerlendiren, uluslararası bir kalite standardıdır. Bu 6 temel faktör Şekil 7.2.'de gösterilmektedir.

Şekil 7.2. ISO 9126 Yazılım kalite standardı

- *Fonksiyonellik(İşlevsellik)*, ISO-9126 kalite ölçütlerine göre yazılım sisteminin belirlenen koşullardaki ihtiyaç ve işlevleri yerine getirebilme yeteneğidir. Yani yazılım kullanıcının tüm isterlerini karşılayabilmelidir.

- *Güvenilirlik*, yazılım sisteminin belirlenen koşullara göre kullanımı gerçekleştirildiğinde ortaya çıkan performansın belirli bir sürdürülebilirliğe sahip olmasıdır. Yani, yazılımın kullanıcı tarafından güncelleme, ihtiyaçlara cevap verebilme gibi desteklerin devam etmesidir. Yazılım güncel bir sürüme sahip olmalı, sürekli ve hatasız çalışabilmeli, tüm işlevleri doğru yapmalı, hatalı girdilere ve kullanıcı yanlışlıklarına karşı korumalı olmalıdır.

- *Verimlilik*, yazılım işlevlerini yerine getirirken sistem öz kaynaklarını en uygun şekilde kullanmaktır. İşlemci, bellek, disk ve diğer kaynakların kullanımı en uygun şekilde yapılmalı, ileitşim, grafik sergileme ve yazıcı çıktıları yüksek başarıma sahip olmalıdır.

- *Taşınabilirlik*, yazılımlar farklı donanım ve işletim sistemine sahip bilgisayarların kullanıldığı ortamlara aktarılabilir olmalıdır. Yeni işletim sistemi sürümlerinin kullanılması, bilgisayar değiştirilmesi gibi durumlarda, daha önce geliştirilmiş olan yazılım en az çaba ile tekrar kullanılabilmelidir.

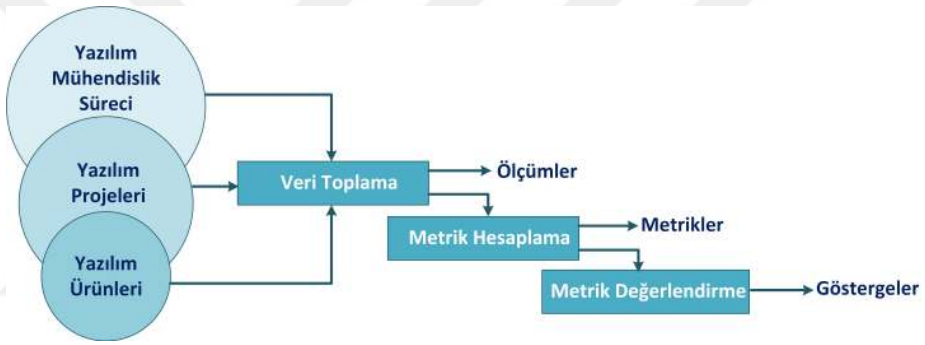
- *Kullanılabilirlik*, yazılım sisteminin kullanıcı tarafından anlaşılması, hızlı öğrenilmesi

ve kullanım kolaylığı olarak belirtilebilir. Üretilen yazılımı insanların rahatlıkla kullanabilmesi için gerekli kolaylıklar sağlanmalı, özellikle grafiksel kullanıcı ara yüzü düzenli, estetik ve kullanımı kolay olmalıdır.

- *Bakım kolaylığı*, başka bir yazılım geliştirici kişi ya da grup tarafından yazılım bakımının yapılabilmesi için kaynak kodun anlaşılabilir bir şekilde yazılmış olması, iyi belgelendirilmesi, sorun çözümlemesinin ve testinin kolay olması gereklidir.

8. Yazılım Ölçütleri

Yazılım mühendisliği bir yazılım veya sistemin geliştirilmesinde, gereksinimlerin belirlenmesi ve analizi, problemlerin çözümlenmesi, süreçlerin planlanması, fiziksel ve mantıksal işlemlerin tasarlanması, modellenmesi, kodlama veya uygulamaların gerçekleştirilmesi, test ve bakımlarının yapılması, yazılım ürünlerinin yönetilmesi gibi süreçleri içerir. Bu süreçlerin her birinde birbirinden farklı parametreler ve çeşitli ölçümler kullanılarak doğrulama işlemleri yapılır. Bu nedenle, ölçüm yazılım mühendisliğinde önemli bir rol oynar ve metrik olarak da ifade edilmektedir. Bir yazılım ürününün niteliklerini ölçmek için titiz bir yaklaşım gereklidir. Her ölçüm açıkça tanımlanmış ve kolayca anlaşılabilir çeşitli parametrelere ve özelliklere bağlı, belirli bir amaç veya ihtiyaçları ifade etmelidir. Yazılım mühendislik süreçleri, yazılım projeleri ve yazılım ürünleri Şekil 8.1.'de gösterildiği üzere iç içe girmiş önemli olgulardır. Yazılım kalite ölçütlerinin bu üç olgudan toplanan veriler sonucunda elde edildiği görülmektedir.



Şekil 8.1. Yazılım ölçütleri toplama süreci [69]

Projenin, ürünün, süreçlerin ve kaynakların durumunu değerlendirmek için ölçüm yapmak gereklidir. Ölçüm hedefleri, yöneticilerin, geliştiricilerin ve kullanıcıların bilmesi gerekenleri dikkate alarak spesifik olmalıdır. Bu ölçümler ile yazılım ürünlerine ait aşağıdaki önemli iyileştirmeler sağlanır:

- Süreçleri, hedefleri ve kod kalitesini iyileştirmek.
- Sistemde yer alan ve hata riski barındıran modülleri tespit etmek.
- Yazılım test sürecinin verimliliğini arttırmak.
- Test süreçleri için kullanılabilecek veri setlerini elde etmek.

Yazılım ölçütleri, çeşitli kriterler göz önünde bulundurularak sınıflandırılır. Ölçüt kümeleri, bilgilerin toplanma zamanına göre sınıflandırılırsa statik ve dinamik olarak iki ayrı grupta incelenir. *Statik ölçütler*, yazılım sisteminin çalıştırılmasına gerek kalmadan, yazılım sisteminin kod yapısıyla ilgili belgelerden elde edilebilen ölçütlerdir. *Dinamik ölçütler*, yazılım çalışmasına bağlı olarak elde edilen verilerden oluşmaktadır. Yazılım

ölçütlerinin bazılarında yalnızca parametre erişimleri dikkate alınırken, bazı ölçütlerde ise metotların tüm veri erişimleri değerlendirilir. Ölçütler, ürettikleri veri tiplerine göre de sınıflandırılabilirler. Ölçütlerden bazıları sonsuz bazıları sonlu, bazıları da sınırlı aralıkta gerçel ya da tam sayı değerleri üretmektedir.

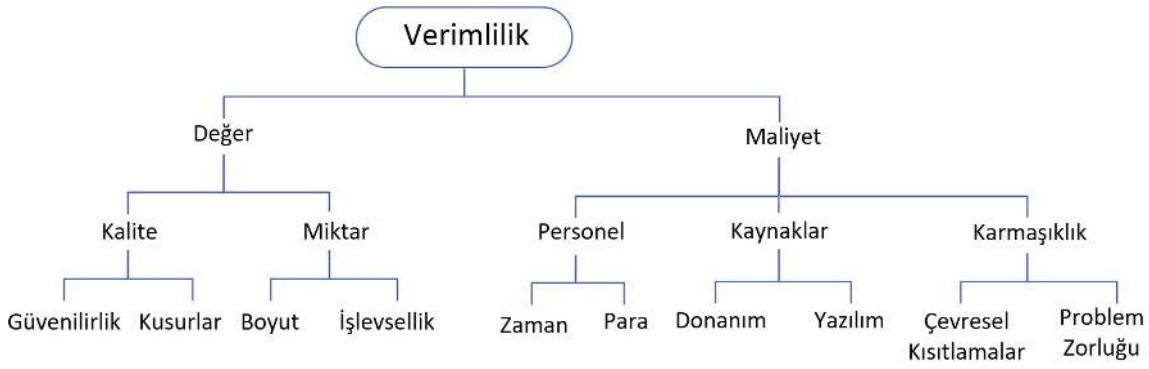
8.1. Yazılım Ölçütlerinin Kapsamı

Yazılım ölçümü, belirli bir aşamada yazılım proje maliyetlerini tahmin eden modellerden program yapısının ölçümlerine kadar uzanan bu faaliyetlerin çeşitli bir koleksiyonudur. Yazılım kalitesini ölçebilmek ve doğru bir şekilde değerlendirebilmek için ölçütleri birden çok kategoriye ayırmak mümkündür. Örneğin, bir projenin süreç içi kalite ölçütleri hem süreç ölçütlerine girer hem de proje ölçütlerini kapsar. Yazılım ölçütleri, aşağıda verilen birçok önemli etkinliği içerir:

8.1.1. Maliyet ve Efor Tahmini

Efor, programın boyutu, geliştiricilerin kapasitesi ve yeniden kullanım düzeyi gibi bir veya daha fazla değişkenin bir fonksiyonu olarak ifade edilir. Yazılım yaşam döngüsünün ilk aşamalarında proje maliyetini tahmin etmek için literatürde maliyet ve efor tahmin modelleri önerilmektedir. Önerilen farklı modeller Boehm'in COCOMO modeli, Putnam'ın ince modeli ve Albrecht'in fonksiyon nokta modelidir [70].

Verimlilik Ölçüleri ve Modeli: Verimlilik, değerin ve maliyetin bir fonksiyonu olarak düşünülebilir. Bu fonksiyonlardan her biri ölçülebilir boyuta, işlevselliğe, zamana, para gibi bir çok etmene ayrıştırılabilir. Bir üretkenlik modelinin farklı olası bileşenleri Şekil 8.2.'de ifade edilebilir.



Şekil 8.2. Verimlilik modelinin bileşenleri

8.1.2. Veri Toplama

Herhangi bir yazılımın doğru ölçülebilmesi, ölçüm programı yada işlemlerinin kaliteli ve dikkatli veri toplanmasına bağlıdır. Toplanan veriler, yöneticilere gelişimin ilerlemesini ve sorununu anlayabilmeleri için basit çizelgeler ve grafikler halinde sunulabilir. Veri toplama,

ilişkilerin ve eğilimlerin bilimsel olarak araştırılması için gereklidir.

8.1.3. Kalite Modelleri ve Ölçümleri

Kalite modelleri, ürün kalitesinin ölçülmesi için geliştirilmiştir. Bu kalite modelleri, doğru verimliliği en uygun şekilde ölçmek için verimlilik modeli ile birleştirir. Bu modeller genellikle ağaç benzeri bir tarzda inşa edilir. Üst dallar, güvenilirlik ve kullanılabilirlik gibi önemli üst düzey kalite faktörlerini içerir. Böl ve yönet yaklaşımı, yazılım kalitesini ölçmek için standart bir yaklaşım olarak uygulanmaktadır.

8.1.4. Güvenilirlik Modelleri

Çoğu kalite modeli güvenilirliği bir bileşen faktörü olarak içerir, ancak güvenilirliği tahmin etme ve ölçme ihtiyacı, güvenilirlik modellemesi ve tahmininde ayrı bir uzmanlaşmaya yol açmıştır. Güvenilirlik teorisindeki temel problem, bir sistemin sonunda ne zaman başarısız olacağını tahmin etmektir. Bu sebeple ürün seçiminde, kullanıcıların büyük çoğunluğu itibar sahibi büyük firma ürünlerini güvenilir olduğu için tercih etmektedirler. Bu durum, kurum veya kuruluşların yazılım teminlerinde yapmış olduğu ihalelerde de geçerlidir. Özellikle firmaların daha önceki iş bitirme durumları, kalite güvencesi sağlanmış veya akredite olmuş yazılım ürünlerinin referanları güvenilirlik için önemli parametrelerden sayılmaktadır.

8.1.5. Performans ve Değerlendirme Ölçütleri

Yazılımdaki tepki süreleri ve tamamlama oranları gibi harici olarak gözlemlenebilir sistem performans özelliklerini ve algoritmaların verimliliği gibi sistemin dahili çalışmasını içerir. Yani, kalitenin başka bir yönüdür.

8.1.6. Yetenek – Olgunluk Değerlendirmesi

Yetenek-olgunluk değerlendirilmesinde, mevcut olan yazılımın temsillerinin yapısal özellikleri ölçülmektedir. Ardından kalite güvencesini, kalite kontrolünü ve kalite tahminini desteklemek için deneysel olarak tahmine dayalı teoriler oluşturmaya çalışılır.

8.1.7. Ölçütlere Göre Yönetim

Ölçütlere göre yönetim modelinde, araçların kullanımı, standart uygulamalar ve daha fazlası dahil olmak üzere geliştirmenin birçok farklı özelliği değerlendirebilir. Uzmanların bilgi ve tecrübelerine bağlı olarak pratik çözümlere dayanır.

8.1.8. Ölçütlerle Yönetim

Yazılım projesini yönetmek için ölçümün hayati bir rolü vardır. Projenin yolunda olup olmadığını kontrol etmek için kullanıcılar ve geliştiriciler, ölçüme dayalı çizelge ve grafiğe güvenebilirler. Standart ölçümler ve raporlama yöntemleri, müşterilerin yazılım terminolojisinde genellikle iyi bilgili olmadığı bir ürüne yazılım gömülü olduğunda özellikle önemlidir.

8.1.9. Yöntem ve Araçların Değerlendirilmesi

Yöntem ve araçların değerlendirilmesi yaklaşımı, deney tasarımına, sonucu etkilemesi muhtemel faktörlerin doğru tanımlanmasına ve kalite faktör özelliklerinin uygun ölçümüne bağlıdır.
Yazılım ölçütleri, bir dereceye kadar birçok aktiviteyi içeren bir ölçüm standardıdır. Yazılım ölçütleri Şekil 8.3.'te gösterildiği gibi, Ürün ölçütleri, süreç ölçütleri ve proje ölçütleri olmak üzere üç kategoride sınıflandırılabilir.

Şekil 8.3. Yazılım ölçütleri

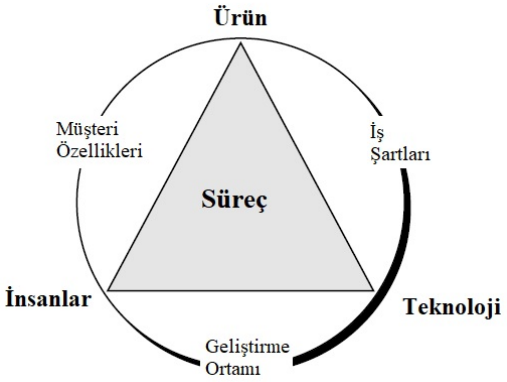
8.2. Süreç Ölçütleri

Süreç ölçütleri, yazılım yaşam döngüsü ile ilişkili özellikler ölçülür. Örneğin; süreç ölçütlerini tanımlarken, süreç içerisinde harcanan çaba-emek, üretilme zamanı, geliştirilen sistemde kusur bulma ve kusurun ortadan kaldırılması için gereken süre, sorunların çözümünde sistemin yanıt verme süresi gibi parametreleri içerir. Bir projede takım liderleri çoğunlukla bu metriklerle ilgilenmektedir. Bu gruptaki metriklerden en önemlileri zaman, çaba ve maliyet ile ilişkili olanlardır. Yazılım geliştiren firmalar ya da kurumlar organizasyonlarla ilgili bir değişiklik yaşadığında süreç metrikleri doğrudan etkilenebilmektedir. Bu sebeple, yazılım ölçüm alanında her firmanın uygulayabileceği ortak süreç ölçütleri oluşmamıştır. Her firma için maliyet tahmini, kusur tahmini, süre tahmini gibi süreç etkinliğini değiştiren parametreler değişebilmektedir. Örneğin; süreçleri yıllar içerisinde oturmuş ve süreçlerin belgelendirilmesi yapılmış, hatta CMMI gibi bir değerlendirmeden geçirilmiş belirli bir seviye belgesi almış olan firmalarda, bu tür metriklerin toplanması kritik öneme sahiptir [71].

Günümüzde geliştirilen yazılımların portföyüne bakıldığında yelpazenin çok geniş olduğu görülmektedir. Gömülü sistem yazılımları, otonom araçlara hükmeden yazılımlar, mobil tabanlı sistemleri kontrol eden yazılımlar ve uygulamalar, SCADA sistemlere entegre fabrikalardaki büyük makineleri çalıştıran yazılımlar, web tabanlı otomasyon sistemleri gibi birbirinden farklı binlerce yazılım uygulamaları bulunmaktadır. Birbirinden farklı teknolojileri içeren bu yazılımların ortaya çıkmasında da birbirinden farklı oranda

emek, zaman ve maliyetlerin harcandığı açıktır. Bu yazılımların her birindeki süreç, proje ve ürün ölçütlerinin değerlendirilmesindeki parametreler de birbirinden farklıdır. Yazılım ölçütleri konusundaki literatür incelendiğinde, yayımlanmış makale ve kitaplarda her yazılım portföyüne uygun farklı yaklaşımlar sunulmaktadır.

Yazılım geliştirme süreci genel olarak ürün, insan ve teknoloji üçgenine oturmaktadır. Herhangi bir süreci iyileştirmenin tek rasyonel yolu, sürecin belirli niteliklerini ölçmek, bu niteliklere dayalı bir dizi anlamlı metrik geliştirmek ve ardından metrikleri bir iyileştirme stratejisine yol açacak göstergeler sağlamak için kullanmaktır. Ancak yazılım ölçütlerini ve bunların yazılım süreci iyileştirme üzerindeki etkisini tartışmadan önce, yazılım geliştirme sürecinin yazılım kalitesini ve kurumsal performansı iyileştirmede bir dizi "kontrol edilebilir faktörden" yalnızca biri olduğunu belirtmek önemlidir. Süreç, Şekil 8.4.'te görüldüğü üzere, yazılım kalitesi ve kurumsal performans üzerinde derin etkisi olan üç faktörü birbirine bağlayan bir üçgenin merkezinde yer alır.



Şekil 8.4. Yazılım kalite ve organizasyonel etkilik için süreç üçgeni [69]

İnsanların beceri ve motivasyon ile kalite ve performansta en etkili tek faktör olduğu gösterilmiştir. Ürünün karmaşıklığı, kalite ve ekip performansı üzerinde önemli bir etkiye sahip olabilir. Süreci kapsayan yazılım mühendisliği yöntemleri ve teknolojilerinin sürece çok önemli bir etkisi vardır. Ayrıca süreç üçgeni, geliştirme ortamını (örn. CASE araçları), iş koşullarını (örn. son teslim tarihleri, iş kuralları) ve müşteri özelliklerini (örn. iletişim kolaylığı) içeren bir çevresel koşullar çemberi içinde bulunur [69].

Bir yazılım sürecinin etkinliği net sayısal değerlerle değil, dolaylı olarak ölçülmektedir. Süreçten elde edilebilecek sonuçlara dayalı bir dizi ölçüt türetilmektedir. Sonuçlar, yazılımın piyasaya sürülmesinden önce ortaya çıkarılan hataların ölçümlerini, son kullanıcılara teslim edilen ve son kullanıcılara bildirilen kusurları, teslim edilen iş ürünlerini (verimlilik), harcanan insan emeğini, harcanan takvim süresini, programa uygunluğu ve diğer önlemleri içerir. Ayrıca belirli yazılım mühendisliği görevlerinin özelliklerini ölçerek de süreç ölçümleri elde edilmektedir.

Grady, farklı türdeki süreç verileri için *özel ve genel* kullanımlar olduğunu savunur

[72]. Yazılım mühendislerinin bireysel olarak toplanan metriklerin kullanımına karşı hassas olmaları doğal olduğundan, bu veriler bireye özel olmalı ve yalnızca birey için bir gösterge olarak hizmet etmelidir. Özel metrik örnekleri arasında bireysel kusur oranları, projedeki modül bazında kusur oranları ve geliştirme sırasında bulunan hatalar yer alır.

Özel süreç verileri felsefesi, Humphrey Humphrey tarafından önerilen kişisel yazılım süreci yaklaşımıyla çok iyi uyum içindedir, yaklaşımı aşağıdaki şekilde tanımlar:

Kişisel yazılım geliştirme süreci, mühendislerin kişisel performanslarını iyileştirmelerine yardımcı olabilecek yapılandırılmış bir dizi süreç tanımları, ölçümler ve yöntemlerdir. Çalışmalarını tahmin etmelerine ve planlamalarına yardımcı olan formları, komut dosyalarını ve standartları sağlar. Onlara süreçleri nasıl tanımlayacaklarını ve kalite ve üretkenliklerini nasıl ölçeceklerini gösterir. Temel ilke şudurki; herkesin farklı olduğu ve bir mühendis için etkili olan bir yöntemin diğerine uygun olamayabileceğidir. Böylece kişisel yazılım geliştirme süreci, mühendislerin kendileri için en iyi yöntemleri bulabilmeleri için kendi çalışmalarını ölçmelerine ve izlemelerine yardımcı olur.

Humphrey, yazılım süreci iyileştirmesinin bireysel düzeyde başlayabileceğini ve başlaması gerektiğinin farkındadır [73]. Bireysel yazılım mühendisi geliştirmeye çalıştığı için özel süreç verileri önemli bir itici güç olarak hizmet edebilir. Bazı süreç ölçümleri yazılım proje ekibine özeldir ancak tüm ekip üyelerine açıktır. Örneğin; birkaç yazılımcı tarafından geliştirilen ana yazılım işlevleri için rapor edilen hataları, resmi teknik incelemeler sırasında bulunan hataları ve modül ve işlev başına kod satırlarını veya işlev noktalarını o çalışmayı içerir. Bu veriler, ekip performansını iyileştirebilecek göstergeleri ortaya çıkarmak için ekip tarafından gözden geçirilir. Kurum ve kuruluşlarda, özel firmalarda grup ya da ekip olarak yazılım geliştirilmesinde klasik veya çevik süreç modellerinin [74] türleri de süreç metriklerini etkilemektedir. Bu bağlamda süreç metrikleri için ölçüt parametrelerinin spesifik belirlenmesi oldukça zordur.

Genel ölçümler genellikle, başlangıçta bireylere ve ekiplere özel olan bilgileri özümser. Proje düzeyinde hata oranları kesinlikle bir bireye atfedilmez. Proje de harcanan çaba, takvim süreleri ve ilgili veriler, organizasyonel süreç performansını iyileştirebilecek göstergeleri ortaya çıkarmak amacıyla toplanır ve değerlendirilir. Bir kuruluş, genel süreç olgunluk düzeyini iyileştirmeye çalışırken, yazılım süreç ölçütlerinde önemli faydalar sağlayabilir. Ancak, tüm metrikler gibi, bunlar da yanlış kullanılabilir ve çözdüklerinden daha fazla sorun doğurabilir. Grady, bir süreç ölçüm programı oluştururken hem yöneticiler hem de uygulayıcılar için uygun olan bir *yazılım ölçümleri görgü kuralları* önermiştir [72].

- Metrik verilerini yorumlarken sağduyu ve kurumsal duyarlılığı kullanın.
- Ölçü ve metrikleri toplayan kişilere ve ekiplere düzenli geri bildirim sağlayın.
- Kişileri değerlendirmek için metrikleri kullanmayın.
- Açık hedefler ve bunlara ulaşmak için kullanılacak ölçütler belirlemek için uygulayıcılar ve ekiplerle birlikte çalışın.

- Ölçümleri asla bireyleri veya ekipleri tehdit etmek için kullanmayın.
- Bir sorun alanını belirten metrik verileri *negatif* olarak değerlendirilmemelidir. Bu veriler sadece süreç iyileştirme için bir göstergedir.
- Diğer önemli metrikleri hariç tutarak tek bir metriğe takılıp kalmayın.

Bir kurum ya da kuruluş, süreç ölçümlerinin toplanması ve kullanılması konusunda daha rahat hale geldikçe, basit göstergelerin türetilmesi, istatistiksel yazılım süreç iyileştirmesi hususunda daha titiz bir yaklaşıma yol açar. Bir uygulama, sistem veya ürün geliştirilirken ve kullanılırken karşılaşılan tüm hatalar ve kusurlar hakkında bilgi toplamak için yazılım hata analizini kullanır. Hata analizi şu şekilde çalışır:

1. Tüm hatalar ve kusurlar kökene göre sınıflandırılır.
2. Her bir hatayı ve kusuru düzeltmenin maliyeti kaydedilir.
3. Her kategorideki hata ve kusurların sayısı sayılır ve azalan düzende sıralanır.
4. Her kategorideki hata ve kusurların toplam maliyeti hesaplanır.
5. Elde edilen veriler, kuruluşa en yüksek maliyetle sonuçlanan kategorileri ortaya çıkarmak için analiz edilir.
6. En maliyetli hata ve kusur sınıfını ortadan kaldırmak veya sıklığını azaltmak amacıyla süreci değiştirmek için planlar geliştirilir.

8.3. Proje Ölçütleri

Yazılım süreç metriklerinin aaksine, proje metrikleri stratejik amaçlar için kullanılır. Yazılım projesi önlemleri taktikseldir. Yani proje metrikleri ve bunlardan türetilen göstergeler, proje iş akışını ve teknik faaliyetleri uyarlamak için bir proje yöneticisi ve bir yazılım ekibi tarafından kullanılır. Kaynak metrikleri olarak bilinen bu metrikler, proje kaynaklarını ortaya koyan teknik personel sayısını, geliştiricinin yeteneği, güvenilirlik ve performans örnek olarak verilebilir. Süreç ölçütleri organizasyon seviyesindeki ölçütler iken proje ölçütlerinin ilgili projelere özel olduğunu söylemek daha doğru olur. Örneğin; gerçek zamanlı gömülü sistem yazılımı geliştirilen bir projede analist sayısı, geliştirici sayısı, test uzmanı sayısı, geliştiricilerin deneyim süresi gibi bilgiler toplanır. Bu bilgiler proje sonunda kesin olarak belirlenecek süre ve maliyet gibi süreç ölçütleriyle birlikte değerlendirilerek sonraki projeler için tecrübe kazandıracaktır [71].

Çoğu yazılım projesinde proje metriklerinin ilk uygulaması tahmin sırasında gerçekleşir. Geçmiş projelerden toplanan metrikler, mevcut yazılım çalışması için çaba ve zaman tahminlerinin yapıldığı bir temel olarak kullanılır. Yani; önceki işlerde elde edilen tecrübelerdir. Bir proje ilerledikçe, harcanan çaba ve takvim süresi, proje iş-zaman çizelgesinde (Gannt diyagramı) belirtilen proje takvimiyle karşılaştırılır. Proje yöneticisi,

ilerlemeyi izlemek ve kontrol etmek için bu verileri kullanır. Bu durum, kullanılan süreç modeline göre proje yönetim aracı üzerinden takip edilir ve yönetilir.

Teknik çalışma başladığında, diğer proje metrikleri önem kazanmaya başlar. Belge sayfaları, inceleme saatleri, işlev noktaları ve teslim edilen kaynak satırları cinsinden temsil edilen üretim oranları ölçülür. Ayrıca, her bir yazılım mühendisliği görevi sırasında ortaya çıkan hatalar izlenir. Yazılım, spesifikasyondan tasarıma geliştikçe, tasarım kalitesini değerlendirmek ve kod oluşturma ve test etme yaklaşımını etkileyecek göstergeler sağlamak için teknik metrikler toplanır.

Proje metriklerinin amacı iki yönlüdür. Birincisi, gecikmeleri önlemek, olası sorunları ve riskleri azaltmak için gerekli ayarlamaları yaparak geliştirme programını en aza indirmeyi hedefleyecek yolların belirlenmesidir. İkincisi, ürün kalitesini sürekli olarak değerlendirmek ve gerektiğinde kaliteyi iyileştirmek adına teknik yaklaşımı değiştirmek için proje metrikleri kullanılır. Kalite arttıkça kusurlar en aza indirilir ve kusur sayısı azaldıkça proje sırasında gerekli veri işleme miktarı da azalır. Bu, toplam proje maliyetinde bir azalmaya yol sağlamış olur.

Yazılım projesi metriklerinin ölçülmesinde, her projenin asgari aşağıdaki durumları ölçmesi gerektiği literatürde verilmektedir.

- *Girdiler:* İşi yapmak için gerekli olan kaynakların (insanlar, çevre, donanım ürünleri) ölçümleri.
- *Çıktılar:* Yazılım mühendisliği süreci sırasında gerçekleşen çıktıların veya iş ürünlerinin ölçümleri.
- *Sonuçlar:* Çıktıların etkinliğini gösteren ölçüler.

Gerçekte, bu model hem sürece hem de projeye uygulanabilir. Proje bağlamında, model, her bir çerçeve etkinliği gerçekleştiğinde tekrarlı olarak uygulanabilir. Bu nedenle, bir faaliyetin çıktısı, bir sonraki faaliyetin girdisi olur. Sonuç metrikleri, bir çerçeve aktiviteden veya görevden diğerine akarken iş ürünlerinin kullanışlılığının bir göstergesini sağlamak için kullanılabilir.

8.4. Ürün Ölçütleri

Ürün ölçütleri, ürünün kalitesine ilişkin boyut, kod ölçümleri, karmaşıklık, tasarım özellikleri, performans, verimlilik, güvenilirlik, taşınabilirlik ve kalite düzeyi gibi özelliklerini tanımlar. Ürün metrikleri, gereksinimlerden yerleşik sistemlere kadar, geliştirmelerinin herhangi bir aşamasındaki yazılım ürünü ölçümleridir. Ürün metrikleri yalnızca yazılım özellikleriyle ilgilidir. Ürün metrikleri iki sınıfa ayrılır:

- *Dinamik ölçütler*, yürütülmekte olan bir programdan yapılan ölçümlerle toplanır. Dinamik metrikler bir programın verimliliğini ve güvenilirliğini değerlendirmeye yardımcı olurken, statik metrikler bir yazılım sisteminin karmaşıklığını anlamaya,

anlamaya ve sürdürmeye yardımcı olur. Genellikle yazılım kalite özellikleriyle oldukça yakından ilişkilidir. Belirli görevler için gereken yürütme süresini ölçmek ve sistemi başlatmak için gereken süreyi tahmin etmek nispeten kolaydır. Bunlar doğrudan sistem arızalarının verimliliği ve yazılımın güvenilirliği ile ilgili olup, arızanın türü kaydedilebilir.

- *Statik ölçütler*, tasarım, programlar ve belgeler gibi sistem temsillerinden yapılan ölçümlerle toplanır. Statik ölçütlerin kalite özellikleriyle dolaylı bir ilişkisi vardır. Karmaşıklık, anlaşılabilirlik ve sürdürülebilirlik arasındaki ilişkiyi türetmeye ve doğrulamaya çalışmak için bu metriklerin büyük bir kısmı önerilmiştir. Program veya bileşen uzunluğu ve kontrol karmaşıklığı tablosunda verilen kalite niteliklerini değerlendirmek için kullanılan çeşitli statik ölçütler, anlaşılabilirlik, sistem karmaşıklığı ve sürdürülebilirlik en güvenilir tahmin edicileri ölçütler olarak ön plana çıkmaktadır.

Yazılım ürün kalitesinin arttırılması amacıyla Ürün ölçütleri üzerine çok sayıda kalite metrikleri belirlenmektedir. Ancak, bu metrikler ile ilgili literatürde birbirinden tek bir yaklaşım yoktur. Geliştirilen ürünün amacı, boyutu, kullanmış olduğu teknolojiler, sistematik yapısı gibi birçok önemli kısımların incelenmesi gerekir. Bu incelemelerin hepsinde niceliksel ölçümler de mümkün olamamaktadır. Bu sebeple bu bölümde genel ürün ölçütleri ile birlikte en yaygın kullanılan kod ölçütleri verilmektedir.

8.4.1. Gereksinim Analizi için Ölçütler

Yazılım mühendisliğindeki teknik çalışma, gereksinim modelinin oluşturulmasıyla başlar. Bu aşamada gereksinimler türetilir ve tasarım için bir temel oluşturulur. Bu nedenle, analiz modelinin kalitesi hakkında fikir veren ürün ölçümleri arzu edilir. Literatürde nispeten az sayıda analiz ve spesifikasyon metriği ortaya çıkmış olsa da, proje tahmini için sıklıkla kullanılan metrikleri uyarlamak ve uygulamak mümkündür. Bu metrikler, ortaya çıkan sistemin *büyüklüğünü* tahmin etmek amacıyla gereksinim modelini inceler. Boyut her zaman olmasa da tasarım karmaşıklığının bir göstergesidir ve neredeyse her zaman artan kodlama, entegrasyon ve test uygulamlarının da bir göstergesidir.

Fonksiyon tabanlı ölçütler

İşlev noktası metriği, analiz modelinden türetilen bir sistemin boyutunu tahmin etmek için araç olarak kullanılabilir. Bu ölçütlerin kullanılmasında, sistemin veri akış diyagramı çizilerek metrik hesaplamaları yapılır. Veri akış diyagramı, fonksiyon noktası metriğinin hesaplanması için kullanıcı girişi sayısı, kullanıcı çıktılarının sayısı, kullanıcı sorgularının sayısı, dosya sayısı, harici arayüz sayısı gibi metrikler incelenir.

Kalite nitelik ölçütleri

Yazılım gerçekleştirme sürecinde gereksinim analizi çok önemli bir evredir. Geliştirilecek yazılımın kalitesini ortaya koyabilmek için kullanılabilecek önemli özellikler dikkate alınmalıdır. Bunlar belirsizliğin olmaması, tamlık, doğruluk, anlaşılabilirlik, doğrulama yeteneği, iç ve dış tutarlılık, ulaşılabilirlik, kesinlik, izlenebilirlik, değiştirilebilirlik, kesinlik ve yeniden kullanılabilirlik gibi önemli özelliklerdir [69].

8.4.2. Tasarım için Ölçütler

Bir aracın, bir bilgisayar entegresinin veya bir iş kulesinin kullanılacak tasarım ölçütleri dikkate alınarak etkili ve ölçekli bir tasarım yapılabilir. Tasarım ölçütleri yol gösterici göstergelerdir. Karmaşık yazılım tabanlı sistemlerin tasarımında genellikle detaylı metrikler kullanılmamaktadır. Bunun ironisi, yazılım için tasarım ölçütlerinin mevcut olması, ancak yazılım mühendislerinin büyük çoğunluğunun bunların varlığından habersiz olmaya devam etmesidir. Diğer tüm yazılım ölçütleri gibi, bilgisayar yazılımı için tasarım ölçütleri de mükemmel olmayabilir. Bunların etkinliği ve nasıl uygulanmaları gerektiği konusunda tartışmalar devam etmektedir. Birçok uzman, tasarım önlemlerinin kullanılabilmesi için daha fazla deneyler, anketler ve incelemeler yapılması gerektiğini savunmaktadır. Yine de ölçüsüz tasarım kabul edilemez.

Mimarisel tasarım ölçütleri

Mimari tasarım ölçütleri, mimari yapıya ve modüllerin etkinliğine vurgu yaparak program mimarisinin özelliklerine odaklanır. Bu metrikler, belirli bir yazılım bileşeninin iç işleyişi hakkında herhangi bir bilgi gerektirmediği için kara kutudur. Card ve Glass, yapısal karmaşıklık, veri karmaşıklığı ve sistem karmaşıklığı şeklinde üç yazılım tasarımı karmaşıklığı ölçüsü tanımlar [69].

Nesne tabanlı tasarım ölçütleri

Nesne tabanlı tasarım ölçütleri kapsamlı ve detaylı inclemelere açıktır. Program yapısı, modül bağımsızlığı, ilişkisiz modüller, veritabanı boyutu, veritabanı bölümlendirme, modül giriş-çıkış karakteristiği gibi önemli parametreler ölçülür. Nesne tabanlı tasarım modelinin boyutu ve karmaşıklığı büyüdükçe, tasarımın özelliklerinin daha nesnel bir görünümü ortaya konulur. Gerçekte, nesne tabanlı sistemleri için ürün metrikleri sadece tasarım modeline değil, aynı zamanda gereksinim modeline de uygulanabilir.

Yazılım sistemleri, her geçen gün değişen ve gelişen bir sektörde barınmaktadır. Bu değişme ve gelişme durumu, yazılımların kalitesine de iyileştirmeye yönelik olarak ışık tutmaktadır. Gittikçe büyüyen, çeşitli beklentileri karşılaması istenen ve neredeyse yüzlerce hatta binlerce kod satırından oluşan projeler için maliyet, çaba ve zaman

faktörlerinin bu ışık doğrultusunda optimum düzeyde olması, sistemin kalitesini büyük ölçüde arttırır [75]. Nesne tabanlı programlamanın etkisiyle boyutu ve karmaşıklığı gittikçe büyüyen projeler için gerekli bakım maliyetlerinin çaba ve zaman unsurları da göz önüne alınarak minimum seviyeye düşürülmesi hedeflenmektedir. Bu amacın gerçekleşmesinin en belirgin destekçisi, nesne yönelimli program geliştirme yaklaşımının birimsellik çerçevesinde ilerlemeyi esas almasıdır. Birimsel program geliştirme mantığı, bilginin gizli tutulması, verinin soyutlanması, kalıtım ve çok biçimlilik gibi nesne yönelimli programlama tekniklerinin etkin bir şekilde kullanılması ile gerçekleşir. Bu nedenle yazılımın bakımı ve aynı sistemde birçok kişinin çalışma durumu yazılım kalitesini arttırır [51].

Sınıf tabanlı tasarım ölçütleri

Sınıf, bir nesne tabanlı sistemin temel birimidir. Bu nedenle, nesne tabanlı tasarım kalitesinin değerlendirilmesinde, tek bir sınıf, sınıf hiyerarşisi ve sınıf işbirlikleri için ölçüler ve metrikler çok değerlidir. Bir sınıf, verileri ve verileri işleyen işlevleri kapsamaktadır.

Chidamber ve Kemerer (CK), en yaygın olarak başvurulan nesne tabanlı yazılım metriklerini önermişlerdir [76]. CK metriklerinde bir sistemin bütününden ziyade sistemdeki sınıflar dikkate alınır [77]. CK metrik paketi olarak adlandırılan Chidamber ve Kemerer tarafından göre sınıf tabanlı ölçütler için 6 (altı) metrik önerilmiştir. Önerilen ölçütler Tablo 8.1. 'de sunulmaktadır.

Tablo 8.1. Chidamber ve Kemere'a göre sınıf tabanlı ölçütleri [69], [76], [78], [79], [80]

Ölçüt adı	**Açıklama**
Sınıf Başına Ağırlıklı Metot Sayısı *(WMC-Weighted Methods per Class)*	Bir sınıftaki metotların karmaşıklık derecesinin veya sayısının toplamı olarak belirlenir. WMC, sınıfın geliştirilmesi ve bakımı için harcanacak sürenin tahminini belirlemede yardımcı olur. Sınıf başına ağırlıklı metot sayısı ile nesne yönelimli bir yazılımın anlaşılırlık, yeniden kullanılabilirlik ve dayanıklılık gibi ölçütleri değerlendirilir. Bundan dolayı, WMC makul olduğu kadar düşük tutulmalıdır.
Kalıtım ağacının derinliği *(DIT-Depth of Inheritance Tree)*	Bu metrik, düğümden ağacının köküne kadar olan maksimum uzunluktur. Derin bir sınıf hiyerarşisi, daha fazla tasarım karmaşıklığına yol açar. Olumlu tarafı, büyük DIT değerleri, birçok yöntemin yeniden kullanılabileceğini belirtir. Herhangi bir sınıftan türetilmeyen değerler için DIT, 0 olarak belirlenir. Çoklu kalıtımın söz konusu olduğu sistemlerde ölçüt için köke en uzak olan kalıtım dikkate alınır. Bu ölçütten yararlanarak yazılımın verimliliği, yeniden kullanılabilirliği, anlaşılırlığı, test edilebilirliği gibi kavramların oranları belirlenir.

Alt Sınıf Sayısı *(NOC-Number Of Children)*	Alt Sınıf Sayısı (Number Of Children-NOC), bir sınıftan doğrudan türetilen toplam alt sınıfların sayısıdır. Çocukların sayısı arttıkça yeniden kullanım artar. Ancak alt sınıf sayısı yüksek olan sistemler, daha karmaşık, düşük bakılabilirlik oranına ve daha çok hataya sahip, yüksek zaman-çaba-maliyet ile test edilebilirlik özelliklerine sahiptir.
Nesne Sınıfları Arasındaki Bağımlılık *(CBO-Coupling Between Object Classes)*	Bir sınıfın bağımlı olduğu sınıfların sayısı toplamı olarak belirlenir. Genel olarak, her sınıf için CBO değerleri makul olduğu kadar düşük tutulmalıdır. CBO, yeniden kullanılabilirlik ve verimliliğin ölçülmesinde kullanılır. Bir sınıfın az bağımlılığa sahip olması, farklı uygulamalarda daha kolay yeniden kullanılabilirliğe sahip olması demektir. Yüksek bağımlılık oranı, değişim duyarlılığını arttırır. Bu durumda yazılımın bakımı daha zor olur. Bağımlılığın fazla olması testlerin daha dikkatli bir şekilde yapılması gerektiğini, dolayısıyla test maliyetlerinin de aynı oranda artacağını gösterir.
Sınıfın Tetiklediği Metot Sayısı *(RFC-Response For a Class)*	RFC, bir sınıfta bulunan nesnenin metotlarının çağrılması ile bu nesnenin tetikleyebileceği tüm metotların sayısı ile belirlenir. RFC arttıkça, test dizisi büyüdüğü için gereken test çabaları da artar. Ayrıca, RFC arttıkça, genel tasarım karmaşıklığının da artar. RFC, test edilebilirlik, bakım ve test zaman-maliyet değerleri hakkında fikir verir. Bu nedenle RFC değerinin düşük olması, sistemin test edilebilirlik oranının yüksek, test için harcanan zaman-maliyet değerlerinin düşük, sistem dayanıklılığının yüksek ve bakımın kolay olduğunu gösterir.
Metot İçi Uyumsuzluk *(LCOM-Lack of Cohesion in Methods)*	LCOM, metriği ile metotların birbirlerine olan benzerliği ölçülür. LCOM yüksekse, yöntemler nitelikler aracılığıyla birbirine bağlanabilir. Bu, sınıf tasarımının karmaşıklığını artırır. LCOM için yüksek bir değerin haklı gösterilebileceği durumlar olsa da, uyumun yüksek tutulması arzu edilir; yani, LCOM'un düşük tutulması en iyi durumdur. LCOM değeri ne kadar yüksekse, o kadar çok durum test edilmelidir. LCOM yardımı ile verimlilik ve yeniden kullanılabilirlik derecesi hakkında bilgi edinilir.

Brito e Abreu MOOD ölçüt kümesi, nesne yönelimli yöntemin yapısal noktalarını esas alır [77]. MOOD ölçüt kümesinin prensipleri, kapsülleme, kalıtım, çok çesitlilik, mesaj aktarımı olarak sıralanır. Brito e Abreu MOOD tarafından nesneye dayalı sınıf tabanlı ölçütlerin belirlenmesinde 6(altı) metrik kullanılmıştır. Önerilen ölçütler Tablo 8.2.'de sunulmaktadır.

Tablo 8.2. Brito e Abreu MOOD'a göre sınıf tabanlı ölçütleri [69] [81] [82]

Ölçüt adı	**Açıklama**
Metot Gizleme Faktörü *(MHF-Method Hiding Factor)*	Metot Gizleme Faktörü(Method Hiding Factor-MHF), sistemde tanımlı olan bütün sınıflardaki çağrılabilen metotların, var olan bütün metotlara oranıdır. Kalıtımla gelen metotlar hesaplamaya dahil edilmez. Bu ölçüt, sınıf tanımının görünürlüğü hakkında yorum yapma şansı tanır.
Nitelik Gizleme Faktörü *(AHF-Attribute Hiding Factor)*	Kalıtımla gelen niteliklerin dahil edilmemesi ve sistemde tanımlı olan tüm sınıflardaki görünür niteliklerin, tüm niteliklere oranı olarak hesaplanır. MHF metriğinde olduğu gibi AHF metriğinin hesaplanmasında da kalıtım ile gelen metotlar hesaplamaya dahil edilmez ve MHF gibi sınıfların görünürlüğü hakkında bilgi sahibi olmak için bu ölçütten yararlanılabilir. Bu ölçütle sınıf tanımının görünürlüğü ölçülür.
Metot Kalıtım Faktörü *(MIF-Method Inheritance Factor)*	Var olan bütün sınıflardaki kalıtım ile gelen metot sayısının, sınıflarda bulunan toplam metot sayısına oranı olarak tanımlanır.
Nitelik Kalıtım Faktörü *(AI-Attribute Inheritance Factor)*	Sistemde tanımlı olan tüm sınıflardaki kalıtım yardımıyla gelen niteliklerin, sistemde var olan bütün niteliklere oranı ile belirlenir.
Çok Biçimlilik Faktörü *(PF-Polymorphism Factor)*	Belirlenen bir sınıf için sistemdeki farklı çok şekilli olan durumların, maksimum olası diğer çok şekilli durumlara oranı olarak belirlenir.
Bağımlılık Faktörü *(CF-Coupling Factor)*	Sistemde sınıflar arasındaki bağımlılık sayısının toplamının, oluşabilecek maksimum bağımlılık sayısına oranı olarak belirlenir. Oranların belirlenmesinde kalıtım ile gelen bağımlılıklar dikkate alınmaz.

Bansiya ve Davis QMOOD Ölçüt Kümesi yazılım toplam kalite endeksinin hesaplanması için faydalanılan ölçüt kümesidir. 4 farklı seviye şeklinde tanımlanmıştır. Seviyelerin ilk adımı sınıf, metot gibi nesne yönelimli yaklaşımların belirlenmesi yer alır. Bir diğer seviyede QMOOD ölçütleri yardımıyla karmaşıklık, uyumluluk gibi bir üst seviyedeki özelliklerin hesaplanması gerçekleştirilir [77].

- *Yazılım Kalite Nitelikleri (Attribute):* QMOOD ölçüt kümesinde işlevsellik, verimlilik, genişleyebilirlik, yeniden kullanılabilirlik ve esneklik gibi niteliklerden bahsedilir.
- *Nesneye Dayalı Yazılım Özellikleri (Property):* QMOOD ölçüt kümesinin 2. seviyesinde artık ölçümler gündeme gelir. Bu aşamada, kalıtım, kapsülleme, çok biçimlilik, soyutlama, bağımlılık, hiyerarşi, yazılımın büyüklük ve karmaşıklığı hakkında bilgilere ulaşılabilir [78].
- *Nesneye Dayalı Yazılım Bileşenleri (Components):* Nesne yönelimli yazılım bileşenleri, nitelik, metot, sınıf, ilişkiler ve sınıf hiyerarşisini içinde barındırır [83].
- *Nesneye Dayalı Yazılım Ölçütleri:* Nesne yönelimli kalite değerlendirilmesinde hiyerarşik bir yaklaşım tanımlanır. Bu yaklaşımda, nesne yönelimli değerlendirme

için kullanılan ölçütler sınıflar, nesneler ve bu iki yapı arasındaki ilişkilerin yapısal ve davranışsal özellikleri ile belirlenir [77].

Bansiya ve Davis tarafından irdelenen QMOOD(Quality Model for Object Oriented Design) ölçütlerinde kullanılan değerler Tablo 8.3.'te sunulmaktadır.

Tablo 8.3. Bansiya ve Davis QMOOD'a göre sınıf tabanlı yazılım ölçütleri [5], [84], [83], [80]

Ölçüt adı	**Açıklama**
Ortalama Ata Sayısı *(ANA-Avarage Number of Ancestors)*	Tüm sınıflarda bulunan ve Chidamber-Kemerer ölçütleriyle hesaplanan DIT değerlerinin ortalaması olarak hesaplanır. Yazılımda soyutlamanın kullanımı hakkında bilgi verir.
Metotlar Arası Uyumluluk *(CAM-Cohesion Among Methods)*	Bansiya tarafından tam olarak tanımı verilmemiş olsa da literatür tanımı olarak metotların imzalarına bakarak ölçülen uyumluluğun hesaplanması olarak ifade edilebilir.
Sınıf Arayüz Boyutu *(CIS-Class Interface Size)*	Sınıfta bulunan erişime açık (public) metotların sayısıdır. Bu sayı sayesinde yazılım sisteminin mesajlaşma özelliği hakkında yorum yapılabilir.
Veri Erişim Metriği *(DAM-Data Access Metric)*	Yazılım sisteminin sınıflarında bulunan özel erişimli (private) ve korumalı (protected) olarak tanımlanan niteliklerin, var olan tüm niteliklere oranı olarak belirlenir. Bu ölçüt ile yazılımın kapsülleme özelliği hakkında bilgi sahibi olunabilir.
Doğrudan Sınıf Bağımlılığı *(DCC-Direct Class Coupling)*	Bir sınıfın parametre olarak kabul edildiği sınıfların toplam sayısı ile bu sınıf nitelik olarak içeren sınıf sayılarının toplamı olarak değer alır. Sınıflar arası bağımlılık bu ölçüt ile belirlenir.
Kümeleme Ölçüsü *(MOA-Measure Of Aggregation)*	Tanımlanan sınıf bildirimlerinin, tanımlı olan temel veri tiplerine oranı olarak belirlenir.
İşlevsel Soyutlama Ölçüsü *(MFA-Measure of Functional Abstraction)*	Sistemdeki tüm sınıflarda bulunan türetilmiş metot sayıları toplamının, tüm metot sayıları toplamına oranıdır. Yazılım sisteminin kalıtım özelliği bu ölçüt ile değerlendirilir.
Metot Sayısı *(NOM-Number of Methods)*	Sınıfta tanımlı olan metot sayısının toplamıdır. Sınıflarda bulunan metot sayısı, sınıf karmaşıklığının da göstergesidir.

Web tabanlı tasarım ölçütleri

İnsan-bilgisayar etkileşiminde arayüzler çok önemlidir. Arayüzlerinin tasarımı hakkında literatürde zengin bilgi paylaşımı olmasına rağmen, arayüzün kalitesi ve kullanılabilirliği hakkında nispeten daha az bilgi yayımlanmıştır. Web sayfası metrikleri üzerine bir çalışma, yerleşimin öğelerinin basit özelliklerinin, GUI tasarımının algılanan kalitesi üzerinde önemli bir etkisi olabileceğini göstermektedir [85]. Bir Web sayfasında bulunan sözcüklerin, bağlantıların, grafiklerin, renklerin ve yazı tiplerinin sayısı, o sayfanın algılanan karmaşıklığını ve kalitesini etkiler. Aşağıda verilen sorulara verilecek cevaplar, web uygulamaları için yararlı bir dizi nicel yanıtlar sağlamaktadır [69].

- Kullanıcı arayüzü kullanılabilirliği destekliyor mu?

- Web uygulama arayüzü estetiği, uygulama alanına uygun ve kullanıcıyı memnun ediyor mu?
- İçerik, en az çabayla en fazla bilgiyi verecek şekilde mi tasarlandı mı?
- Navigasyon, yani webde gezinti yapmak uygun, verimli ve anlaşılır mı?
- WebApp mimarisi, kullanıcılarının özel amaç ve hedeflerine, içerik ve işlevsellik yapısına ve sistemi etkin bir şekilde kullanmak için gereken gezinme akışına uyum sağlayacak şekilde tasarlandı mı?
- Bileşenler, prosedürel karmaşıklığı azaltacak ve doğruluğu, güvenilirliği ve performansı artıracak şekilde mi tasarlandı?

Soruların herbiri yalnızca niteliksel olarak ele alınabilir, çünkü nicel cevaplar sağlayacak doğrulanmış bir metrik takımı henüz mevcut değildir. Takip eden paragraflarda, literatürde önerilen web ve mobil uygulama tasarım ölçütlerinin temsili bir örneği verilmiştir. Bu öneriler resmi bir standart değil, öneri niteliğinde literatürde birçok kaynakta sunulmaktadır. Web uygulama arayüzler için Tablo 8.4.'te bulunan hususlar dikkate alınabilir.

Tablo 8.4. Web tabanlı tasarımlar için arayüz ölçütleri

Önerilen Metrik	**Açıklama**
Sayfa düzeni	Arayüz içindeki varlıkların görüş alanına uygun konumları.
Düzen karmaşıklığı	Bir arabirim için tanımlanan farklı bölgelerin tanımlanması.
Düzen bölgesi karmaşıklığı	Bölge başına eklenen ortalama farklı bağlantı sayısı.
Tanıma karmaşıklığı	Kullanıcının gezinme veya veri girişi kararı vermeden önce bakması gereken ortalama farklı öğe sayısı.
Tanıma süresi	Bir kullanıcının belirli bir işlemi yaparken uygun eylemi seçmesi için geçen ortalama süre.
Klavye bilgi girişi	Belirli bir işlev için gereken ortalama tuş vuruşu sayısı.
Fare klikleri	İşlev başına ortalama fare seçme sayısı.
Seçim karmaşıklığı	Sayfa başına seçilebilecek ortalama bağlantı sayısı
İçerik edinme süresi	Web sayfası başına düşen ortalama metin kelime sayısı.
Bellek yükü	Belirli bir hedefe ulaşmak için kullanıcının hatırlaması gereken farklı veri öğelerinin ortalama sayısı

Web grafik tasarımlarla belirlenen estetik metrikleri, doğası gereği niteliksel yargıya dayanır ve genellikle niceliksel ölçülere uygun değildir. Bununla birlikte, Ivory ve meslektaşları estetik tasarımın etkisini değerlendirmede faydalı olabilecek bir dizi önlem önermektedir [85]. Web tabanlı tasarımlarda estetik ölçütlerine ait değerler ve açıklamaları Tablo 8.5.'te sunulmuştur.

Tablo 8.5. Web tabanlı tasarımlar için estetik ölçütleri

Önerilen Metrik	**Açıklama**
Kelime sayısı	Bir sayfada görünen toplam kelime sayısı.
Gövde metni yüzdesi	Gövde ve görüntü metni arasındaki kelimelerin yüzdesi.
Vurgulanan gövde metni yüzdesi	Gövde metninin kalın, büyük harfle yazılmış vurgulanan kısmı.
Metin konumlandırma sayısı	Soldan hizalı metin konumunda değişikliklerin sayısı.
Metin kümesi sayısı	Renkle vurgulanan metin alanları, kenarlıklı bölgeler, kurallar veya listeler.
Bağlantı sayısı	Bir sayfadaki toplam bağlantı sayısı.
Sayfa boyutu	Sayfanın yanı sıra öğeler, grafikler ve stil sayfaları için toplam bayt
Grafik yüzdesi	Grafikler için olan sayfa baytlarının yüzdesi
Grafik sayısı	komut dosyalarında, uygulamalarda ve nesnelerde belirtilen grafikler hariç bir sayfadaki toplam grafik ve resim sayısı.
Renk sayısı	Kullanılan toplam renkler.
Yazı tipi sayısı	Kullanılan toplam farklı yazı tipleri.

Web sayfalarınındaki içeriklerin uygunluğu açısından geliştirilen metrikler, içerik karmaşıklığına ve sayfalar halinde düzenlenen içerik nesneleri kümelerine odaklanırlar. Literatürde yaygın olarak önerilen web tabanlı Tablo 8.6.'da tasarım ölçütleri sunulmaktadır [86] .

Tablo 8.6. Web tabanlı tasarımlar için içerik ölçütleri

Önerilen Metrik	**Açıklama**
Sayfa bekleme süresi	Bir sayfanın farklı bir bağlantıda indirilmesi için gereken ortalama süre.
Sayfa karmaşıklığı	Sayfada kullanılan metin, grafik, tablo, tüm görsel ve işitsel unsurların farklı ortam türlerinin ortalama sayısı.
Grafik karmaşıklığı	Sayfa başına ortalama grafik ortamı sayısı.
Ses karmaşıklığı	Sayfa başına ortalama sesli medya sayısı.
Video karmaşıklığı	Sayfa başına ortalama video medyası sayısı.
Animasyon karmaşıklığı	Sayfa başına ortalama animasyon sayısı.
Taranan görüntü karmaşıklığı	Sayfa başına ortalama taranan görüntü sayısı.

Web sayfalarındaki bütünlük, akış ve kullanıcıların gezintiler sağladığı yol kılavuzu (navigasyon) kullanıcı tercihlerini etkilemektedir. Bu kategorideki yazılım metrikler,

navigasyon akışının karmaşıklığını ele alır [86]. Yalnızca dinamik olarak oluşturulmuş bağlantılar ve sayfalar içermeyen statik web uygulamaları için geçerlidir. Önerilen navigasyon ölçütleri Tablo 8.7.'de verilmiştir.

Tablo 8.7. Web tabanlı tasarımlar için navigasyon ölçütleri

Önerilen Metrik	**Açıklama**
Sayfa bağlama karmaşıklığı	Sayfa başına bağlantı sayısı
Bağlantı	Dinamik olarak oluşturulmuş bağlantılar hariç toplam dahili bağlantı sayısı
Bağlantı yoğunluğu	Bağlantının sayfa sayısına bölünmesi

8.4.3. Kaynak Kod için Ölçütler

Yazılımın kalitesinin değerlendirilmesinde sürdürülebilirlik önemli bir etkendir. Sürdürülebilirlik için kodun sürekli gelişime açık olması ve bakımının kolay olması gerekmektedir. Dolayısıyla kodun okunabilirliği ve anlaşılabilirliğinin yüksek olması kalite oranını da arttırır. Yazılım kod kalitesinin belirlenmesi için çeşitli ölçütler önerilmiştir. Önerilen ölçütlerden en temel olanı geleneksel McCabe kalite ölçütleridir. Geleneksel kalite ölçütleri proje büyüklüğü, kodun okunabilir ve anlaşılabilir olması, algoritmik karmaşıklık gibi içerdiği 3 farklı bileşenle en temel şekilde kodun iç yapısını değerlendirir. Geleneksel ölçütlerde temel konular yardımıyla yazılım sisteminin kalitesi hakkında bilgi sahibi olunur. Kod bakımının kolay yapılabilirliği ve anlaşılabilirliğini arttırmak için yorum satırları kullanılmaktadır. Kod karmaşıklık analizi, program kod satırlarının sayısı dikkate alan ölçüttür. Kod bloklarındaki satırlarda bulunan boşluk, yorum, mantıksal ifadeler gibi satırların sayılmasıyla belirlenen ölçütler ile yapılan ölçme işlemidir. Tablo 8.8.'de kod ölçütleri verilmiştir.

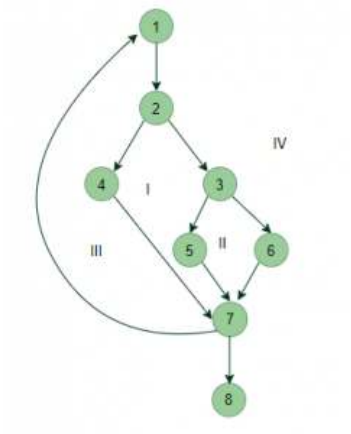
Tablo 8.8. McCabe'e göre kaynak kodlar için ölçütler [69], [87]

Ölçüt adı	**Açıklama**
Kod satırları *(LOC-Lines Of Code)*	Kod bloğundaki satır sayıları.
Kaynak kod satırları *(SLOC-Source Lines of Code)*	Kod bloklarındaki boşluk satırları hariç ama yorum satırları da dahil olan tüm kod satırlarının sayısıdır.
Koddaki fiziksel Satırlar *(PLOC-Physical Lines Of Code)*	Kod bloklarındaki açıklama satırları ve boş satırlar dışında kalan tüm fiziksel satırların sayısıdır. Yorum ve boşluk içermeyen satırlar *(Non-Comment Non-Blank-NCNB)* olarak da bilinir.
Koddaki mantıksal satırlar *(LLOC-Logical Lines of Code)*	Kod bloklarındaki `if, while, for, switch-case, do-while` gibi mantıksal ifadelerin bulunduğu satırların sayısıdır.
Çalıştırılabilir yordam sayısı *(EXEC-Executable Statements)*	Kod bloklarında bulunan içinde hesaplama adımları bulunduran, fonksiyon, altyordam, altprogram, metot gibi kod parçacıklarının toplam sayısıdır.
Yorum oranı *(CP-Comment Percentage)*	Program içerisinde bulunan yorum satırlarının, programda bulunan yorum ve boşluk içermeyen toplam satır sayısına oranıdır. Yorum oranının yüksek olması, programın anlaşılabilir ve okunabilirliğini arttırır.
Döngüsel karmaşıklık *(CC-Cyclomatic Complexity)*	Koddaki koşullu gidiş yollarının doğrusal dallanmaları şeklinde ifade edilir.

Döngüsel karmaşıklık (Cyclomatic Complexity - CC): Döngüsel karmaşıklık ölçütü, koşullu ifade ve yapıların sayısı ile hesaplanır. Kodda bulunan kontrol blokları ve döngüsel yapılar koşullu dallanmaya sebebiyet veren ölçütler olarak değerlendirilir [88]. Döngüsel karmaşıklığı yüksek olan modüllerin, döngüsel karmaşıklığı daha düşük olan modüllere göre hataya eğilimli olma olasılığı daha yüksektir. Bu nedenle, bir sisteme entegre edilmeden önce bu tür modüllerdeki hataları ortaya çıkarmak için ortalamanın üzerinde çaba sarf edilmelidir. Kodun sahip olduğu ve koşullu dallanmaya neden olabilecek kırılma noktalarının çokluğu, döngüsel karmaşıklığın yüksek olmasına sebep olur. Döngüsel karmaşıklık, yazılımın mantıksal karmaşıklığının bir ölçüsüdür. Bu ölçü, bağımsız yolların sayısını tanımlamak için kullanılır.

$$CC = (KenarSayisi) - (DugumSayisi) + 2$$

Şekil 8.5.'te verilen çizgesel diyagram Döngüsel Kod karmaşıklığı açısından incelendiğinde, bağımsız 4 farklı alternatif yol olduğu görülmektedir.



Şekil 8.5. Çizgesel diyagram örneği

V(G) = 4 (Yukarıdaki formüllerden herhangi birini kullanarak bulunur.)

1. P1: 1 – 2 – 4 – 7 – 8
2. P2: 1 – 2 – 3 – 5 – 7 – 8
3. P3: 1 – 2 – 3 – 6 – 7 – 8
4. P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

Formülde bulunan düğümlerden her biri mantıksal bir dalı, kenarlardan her biri düğümler arasındaki bağlantıyı temsil eder. Programın döngüsel karmaşıklığının fazla olması test işlemi için gerekecek durum sayısının fazla olmasına sebep olacaktır. Dolayısıyla bir yöntemin ya da sistemin döngüsel karmaşıklığının düşük olması, kolay test edilebilirliğini gösterir. Yazılım Mühendisliği Enstitüsü (Software Engineering Institute - SEI) tarafından kabul gören döngüsel karmaşıklık ve risk değerlendirmesi Tablo 8.9.'da verilmiştir [89].

Tablo 8.9. Döngüsel karmaşıklık değerlerine karşı gelen risklerin değerlendirilmesi

Döngüsel Karmaşıklık	**Risk Değerlendirmesi**
1-10	Düşük riske sahip karmaşık olmayan modül
11-20	Orta riske sahip az karmaşık modül
21-50	Yüksek derecede riske sahip olan karmaşık modül
50-∞	Çok yüksek derece riskli test edilemez modül

Literatür incelendiğinde, yazılım ölçütleri konusunda farklı öneriler ve yaklaşımları olduğu görülmektedir. Maurice Howard Halstead tarafından 1977'de [90] yazılımın ölçülebilir özelliklerini ve aralarındaki ilişkileri belirlemek amacıyla ölçütler belirlemiştir. Önerilen ölçütler, algoritmaların farklı dillerdeki uygulamasını veya ifadesini yansıtmakta, ancak bunların belirli bir platformda yürütülmesinden bağımsız olması gerektiği

gözlemlerini yapmıştır. Dolayısıyla Halstead ölçütleri aslında sadece karmaşıklık ölçütleri değildir. Halstead'in çalışması deneysel doğrulamaya uygundur ve yazılım bilimini araştırmak için çok sayıda araştırma yapılmıştır.

8.4.4. Test için Ölçütler

Yazılım geliştirme yaşam döngüsünün her evresinde yazılım projesine uygun olan yazılım ölçütlerinin belirlenerek süreçlere uygulanması önemlidir. Yazılım kalitesinin arttırılması için her aşamadaki metrikler ayrı bir önem kazanmaktadır. Test için önerilen metriklerin çoğu, testlerin teknik özelliklerine değil, test sürecine odaklanır. Genel olarak, test uzmanları, test senaryolarının tasarımında ve yürütülmesinde kendilerine rehberlik etmesi için analiz, tasarım ve kod metriklerine güvenmelidir. Yazılım kalitesinin arttırılabilmesi doğru test metotları ve doğru test yöntemlerinin test seviyelerine göre etkili bir şekilde uygulanmasına bağlıdır. Bu kısmlar tez çalışmasının ilgili bölümlerinde sunulmuştur.

8.4.5. Bakım için Ölçütler

Bir yazılım ürününün geliştirilme süreci tamamlanıp piyasaya sürüldüğünde, yaşam döngüsünün bakım aşamasına girer. Bu aşamada, zaman aralığına göre kusurların ortaya çıkması ve müşterinin tespit ettiği sorunları bildirilmesi gerçek zamanlı ölçümlerdir. Bununla birlikte, kusur veya sorunların bildirilme sayısı büyük ölçüde bakım aşamasından önceki geliştirme süreci tarafından belirlenir. Bu nedenle, bu iki fiili ölçü, önemli olmasına rağmen, yazılım bakımının kalitesini yansıtmaz. Bakım aşamasında yapılacak en önemli husus, kusurları en kısa sürede ve mükemmel düzeltme kalitesiyle iyileştirmektir. Bu tür eylemler, ürünün kusur oranını hala iyileştiremese de, müşteri memnuniyetini büyük ölçüde artırabilir. Bu nedenle bakım aşaması için aşağıdaki ölçütler çok önemlidir ve dikkate alınmalıdır [91].

- Birikmiş iş yığını ve iş paketleri yönetimi dizininin organize edilmelidir.
- Tepki süresini düzeltilmelidir.
- Kısa, orta ve uzun vadedeki düzeltmeler belirlenmeli, gecikmeli düzeltmeler belirtilmelidir.
- Yazılım kalite kriterlerine uygun olmayan unsurlar belirlenmeli ve kalitesi arttırılması için iyileştirmeler yapılmaldır.

9. Jenkins Test Platformu ve Uygulamaları

Tez çalışmasının bu bölümünde, Java programlama diliyle bir kütüphane uygulaması gerçekleştirilmiştir. Geliştirilen uygulama için JUnit kütüphanesinden faydalanılarak birim testleri yazılmış ve bu test sonuçları Assert metotları ile değerlendirilmiştir. Test uygulamaları, sürekli entegrasyonu ve sürekli teslimatı destekleyen Jenkins test platformu yardımıyla otomatikleştirilmiştir. Ayrıca Java Eclipse destekli CodeMR kütüphanesi kullanılarak uygulamanın yazılım kod ölçütleri çıkarılmıştır.

9.1. Jenkins Test Platformu

Jenkins, sürekli entegrasyon, sürekli teslimat ve sürekli dağıtım süreçlerinin yürütülmesi için kullanılan bir otomasyon sunucusudur. Açık kaynak kodlu olması nedeniyle diğer test platformlarına göre daha çok tercih edilir. 2004 yılında Sun Microsystems bünyesinde çalışan Kohsuke Kawaguchi tarafından, yazılan kodun sürekli hata vermesi sonucu geliştirilmiştir [92]. Jenkins test platformu, kod değişikliklerinin test edilmesi ve raporlanmasını sürekli hale getirerek, otomatik bir yapı oluşturur. Bu sayede kodun farklı versiyonlarının entegrasyonu kolaylaşır. Ayrıca, kod tabanında var olan hataların tespiti hızlandırılır, kod yapılarının testi otomatikleşir ve kod ilgili sunuculara dağıtılır. Jenkins, GitHub, Bitbucket veya GitLab gibi platformlarda herhangi bir kod değişikliğini takip etmek için ayarlanabilir. Proje geliştirme sürecini yönetmek için Jenkins gibi farklı entegrasyon araçları da mevcuttur. Diğer platformlar ile karşılaştırıldığında Jenkins platformunun daha iyi özelliklere sahip olduğu görülmüştür. Jenkins platformunun açık kaynak kodlu olması, tamamen ücretsiz olması ve işletim sistemi desteğin olması bu özelliklerden en önemli ve avantajlı olanları arasındadır. Sürekli entegrasyon söz konusu olunca sektörde bulunan en popüler entegrasyon araçları ve bu araçlara ait özellikler Tablo 9.1.'de verilmektedir:

Tablo 9.1. Jenkins ve muadillerinin kıyaslanması

Özellik	**Jenkins**	**TeamCity**	**Bamboo**	**Travis**	**Circle**	**Codeship**
Açık kaynak	+	-	-	+	-	-
Ücretsiz Tam Sürüm	+	-	-	-	-	-
Windows Desteği	+	+	+	-	-	+
Linux Desteği	+	+	+	+	+	-
Yerel Sunucu Desteği	+	+	+	+	-	-
Bulut Sunucu Desteği	+	-	+	+	+	+
Eklenti Çeşitliliği Puanı	5/5	4/5	2/5	4/5	3/5	4/5
Kaynak ve Destek	Yeterli	İyi	İyi	Yetersiz	İyi	Yetersiz
Kullanım Kolaylığı	Kolay	Orta	Orta	Kolay	Kolay	Kolay

Tablo 9.1.'de verilen tüm özellikleri Jenkins platformunun karşıladığı görülmektedir. Jenkins, sunduğu 1700'den fazla eklenti sayesinde, bir çok araçla entegre şekilde çalışma imkanı sunar [92]. Jenkins platformu kullanılarak gerçekleştirilen test uygulamalarının avantajları aşağıda verilmiştir.

Jenkins Kullanılmadan Gerçekleştirilen Sistemler

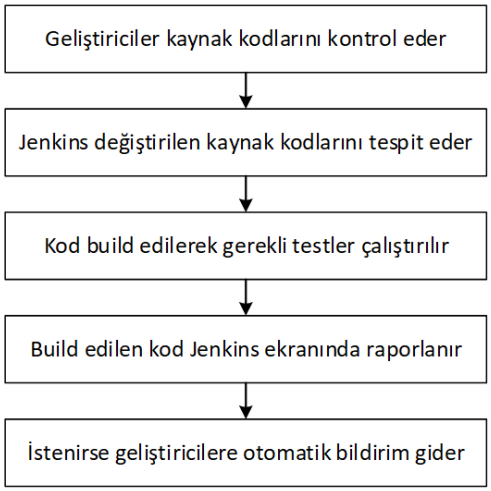
- Belirlenen görev dağılımları sonucu birçok geliştirici tarafından kod blokları hazırlanır. Bu kod blokları, aynı anda entegre edilir. Daha sonra derleme ve test işlemleri yürütülür. Diğer kullanıcılar için dağıtım işlemi gerçekleştirilir. Testler seyrek zaman dilimlerinde gerçekleştirilir ve bir süre sonra entegrasyon işlemi gerçekleştirilir.
- Derlemelerin kontrolü için bütün geliştiricilerin kodlamayı bitirmesi gerekmektedir. Kodlama işleminin tüm geliştiriciler tarafından eş zamanlı yürütülür. Bu nedenle kontrol işlemi uzun sürer.
- Entegrasyon sonucu ortaya çıkan hatanın birden fazla olması durumunda, hataların tespiti ve düzeltilmesi, uzun uğraş ve süreç gerektirir.
- Kodun oluşturulması ve test edilmesi manuel olarak gerçekleştirilir. Bu nedenle var olan hataların gözden kaçması olasıdır.
- Kodlar, tüm hataların tespiti ve düzenlenmesinden sonra test edilir. Dağıtılma işlemi daha sonra gerçekleştirilir.
- Sistemin kodlanması, test edilmesi, düzenlenmesi ve entegrasyonu adımlarını genel olarak ifade eden geliştirme döngüsü yavaştır.

Jenkins Kullanılarak Gerçekleştirilen Sistemler

- Kod blokları hazırlanır ve anında test işlemi gerçekleştirilir. Jenkins platformu yardımıyla kodlar, gün içinde birçok kez derlenir ve test edilir. Derleme ve test işleminin başarılı olması durumunda kaynak kod Jenkins yardımıyla test sunucusuna yerleştirilir ve dağıtım ekibi bilgilendirilir. Derlemenin başarısız olması durumunda ise ortaya çıkan hatalar, Jenkins tarafından geliştirici ekiplere sunulur.
- Derlemenin kontrol edilmesi, geliştiricinin kodu geliştirme sunucusuna göndermesinden hemen sonra gerçekleştirilir.
- Kodun geliştirilmesi, bir geliştiricinin her onayından sonra gerçekleşir. Bu durum, kodun başarısız olmasına neden olan hatanın tespitini kolaylaştırır.
- Otomatik derleme ve test süreci, zaman ve maliyetten kar sağladığı gibi; hataların gözden kaçması zorlaşır.
- Kodların dağıtımı, her başarılı derleme ve test sonrası gerçekleştirilir.

- Geliştirme döngüsü, sistemin devamlılığını sağlayacak şekilde planlanır ve Jenkins ile gerçekleştirilen sistemlerde geliştirme döngüsü hızlıdır.

Jenkins platformunun temelinde yatan mantık, sürekli entegrasyon ve sürekli dağıtım desteğiyle yazılımların kısa süreçlerle kullanıma sunulmasıdır. Böylelikle yazılımın kullanımı sırasında da var olan ya da oluşabilecek hataların çözümü gerçekleştirilir. Geliştirme süresince yazılımın kullanılmaya devam edilmesi ile daha fazla artırımlı güncellemeler yapılabilir. Artırımlı güncellemeler yardımıyla yazılımda meydana gelen değişiklikler için maliyet, süreç ve riskler azalır. Jenkins sunucu tabanlı bir uygulamadır. Apache Tomcat gibi bir web sunucusu gerektirir. Jenkins ile yazılım yaşam döngüsü daha sağlıklı bir şekilde ilerler ve yazılım geliştirme süreçleri hızlanır. Jenkins test platformundaki işlem adımları Şekil 9.1.'de verilmiştir:



Şekil 9.1. Jenkins platformunun çalışma adımları

9.1.1. Jenkins Kurulumu

Jenkins, Unix ve Windows işletim sistemleri desteği olan, sürekli entegrasyon ve sürekli teslimat mantığına dayanan bir test platformudur. Jenkins, Java tabanlı bir otomasyon aracı olduğu için kurulacağı bilgisayarda Java Development Kit (JDK) veya Java Runtime Environment (JRE) ortamlarından birisinin yüklü olması gerekir. Yüklü olan ortamın indirilen Jenkins sürümü ile uyumlu çalıştığı kontrol edilmelidir. Programın bilgisayara kurulumu için Jenkins test platformu resmi sitesinden indirilecek cihazın işletim sistemi seçilir ve indirme işlemine başlatılır [92]. Açılan ekranda programın kurulması istenen dosya uzantısı seçilir. Gelen ekranda kur butonuna basılarak kurulum tamamlanır. Kurulum tamamlandıktan sonra açılan tarayıcı sekmesinde yönetici şifresi sorulur. Yönetici şifresi genellikle Jenkins yükleme konumu içinde */Secret* klasöründe *initialAdminPassword* dosyası altında bulunabilir. Ardından istenilen eklentiler seçilerek yüklenir. Son olarak admin kullanıcısı oluşturularak Jenkins'in çalıştırılacağı port

numarası ayarlanır.

9.1.2. Jenkins Pipeline

Jenkins pipeline eklentisi, sürekli teslimat için işlem yollarının Jenkins'e uygulanmasını ve entegre edilmesini destekleyen bir pakettir [92]. Yazılım sisteminde gerçekleştirilen değişiklikler, kullanıcılara sunulmak için karmaşık bir yol takip eder. Bu yolun, yazılımın güvenilir ve sürekli bir şekilde oluşturulması gerekir. Kısacası pipeline, otomasyon araçları yardımıyla sistemin yeni sürümünün son kullanıcılara ulaşmasını sağlayan adımlar bütününü oluşturur. Jenkins Pipeline kavramları aşağıda verilmektedir.

- *Ardışık Düzen(Pipeline):* Sürekli entegrasyon, sürekli teslimat ve sürekli dağıtımı sağlamak için kod blokları şeklinde verilen talimatlar dizisidir. Pipeline ile uygulama test edilebilir ve testin başarılı sonuçlanmasından sonra teslim edilebilir. Geliştirme süreçlerinin tamamını içeren işlemler kümesidir.
- *Düğüm(Node):* Jenkins test platformunun çalıştığı cihaz, Node olarak isimlendirilir. Tüm iş akışının yürütüldüğü cihazı tanımlamak için kullanılır.
- *Evre(Stage):* Jenkins çalışma ardışık düzeninde bulunan işlem adımlarının bir kısmını içerir. Derleme, test ve dağıtım gibi süreçlerin görselleştirilerek tek bir aşama gibi işleme alınmasını sağlar.
- *Adım(Step):* Belirlenen bir görevin, belirlenen zamanda yürütülmesi olarak tanımlanır. Bir ardışık düzen, birkaç adımın birleşmesiyle oluşur. Jenkins Pipeline, hem kullanıcı arayüzünden hem de JenkinsFile dosyasından ayarlanabilir. Ancak pipeline, Jenkins dosyasında tanımlanmalıdır.

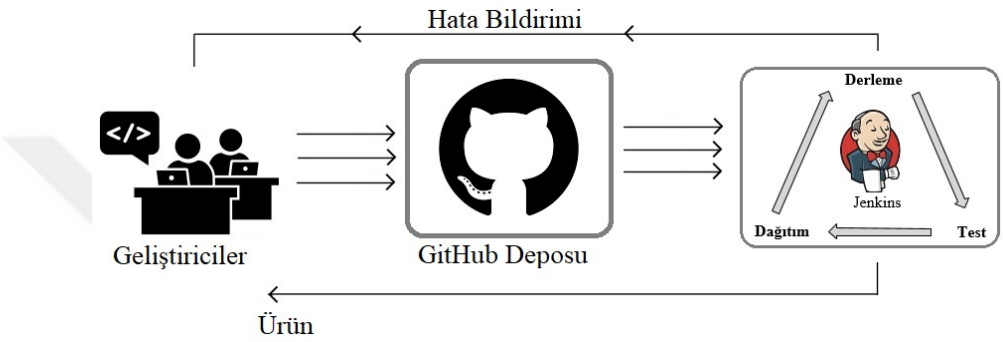
Jenkins pipeline kullanımının avantajları aşağıda maddeler halinde verilmektedir.

- Jenkins pipeline, birçok kullanıcı tarafından takip edilebilir ve düzenlenebilir.
- Sunucu tarafından öngörülemeyen bir şekilde, yeniden başlatma olursa pipeline adımları otomatik olarak devam eder.
- Pipeline işlemi durdurulabilir ve yeni bir işlem olduğunda devamı sağlanabilir.

9.1.3. Jenkins GitHub Entegrasyonu

Web tabanlı bir kod deposu olan GitHub, DevOps mantığında önemli bir rol oynar. DevOps (Developers and Operations), yazılımın geliştirilmesi ve işletilmesinden sorumlu takımlar arasındaki işbirliğini arttırmada kullanılır. Takım çalışması mantığını destekler. GitHub aynı kod ya da proje üzerinde çalışan birçok geliştiricilerin güncel kodlarını yüklediği ortak bir platformdur. Bu sayede sürekli entegrasyon mantığı kolaylıkla işler. Jenkins kurulum aşamasında, komut istemine yanıt olarak Git eklentisi kurulduysa yeniden GitHub eklentisi kurulmasına gerek kalmaz. Ancak Git eklentisi kurulmadıysa, Jenkins gösterge tablosunda bulunan Manage Jenkins'e tıklanır. Manage Plugins

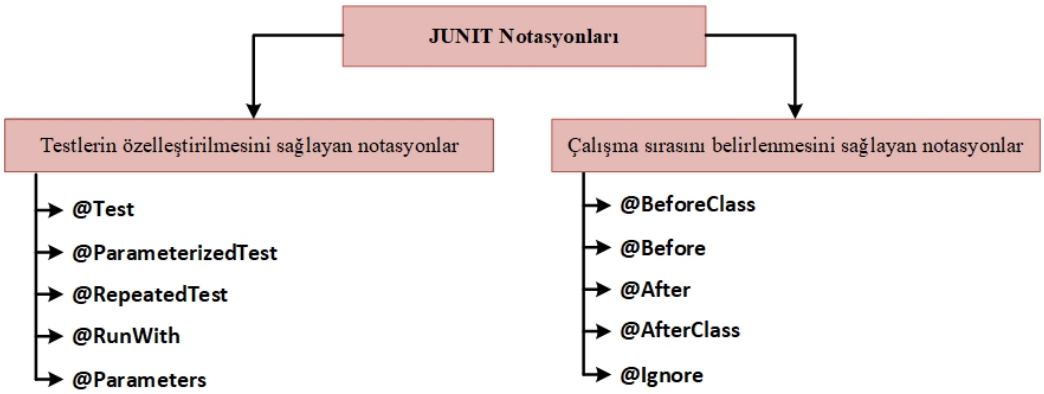
(Eklentileri Yönet) opsiyonu seçilir. Gelen eklentiler sayfasında GIT eklentisi seçilir. Daha sonra ekranda bulunan *Yeniden başlatmadan yükle*, *Yeniden başlattıktan sonra yükle* ya da *Şimdi indir* butonlarından birine tıklanır ve kurulum seçilen butona göre gerçekleştirilir. Jenkins'i Github'a entegre edebilmek için cihazda GitHub Desktop kurulu olmalıdır. Jenkins'te, oluşturulan yeni proje adı girildikten sonra iş türü seçilir. Yönlendirilen proje formunda proje bilgileri girilir. Git Eklentisi, Jenkins'e yüklendiğinde Kaynak Kod Yönetimi altında görünen Git seçeneği tıklanır. Kodun alınacağı GitHub deposundaki URL girildikten sonra GitHub, Jenkins'e başarılı olarak entegre edilmiş olur. Jenkins test platformu ve Github arasındaki çalışma düzeni Şekil 9.2.'de gösterilmiştir.



Şekil 9.2. Jenkins platformu ve GitHub bağlantısı

9.2. JUnit Notasyonları

Testlerin özelleştirilmesi ve çalışma sırasının belirlenmesi gibi durumlarda notasyonlardan yararlanılır. Notasyonlar, genel olarak testlerin çalışma sırasını belirleyen ve test özelliklerini tanımlamayan notasyonlar olarak iki grupta incelemek mümkündür. Şekil 9.3. JUnit'de kullanılan notasyonların gruplandırılmasını içerir.



Şekil 9.3. JUnit notasyonları

9.2.1. Testlerin Özelleştirilmesini Sağlayan Test Notasyonları

Bir yazılım sisteminde test olarak çalışması istenilen kod parçasının önüne test notasyonları eklenir. Bu notasyonlar, birkaç defa kullanılabilir. Test işlemi, yazılan test notasyon sayısı kadar tekrarlanır. Eğer herhangi bir parametre tanımlanmadıysa testler alfabetik olarak çalışır.
Aşağıdaki parametreler, hangi testin hangi sırayla çalışacağını, koşturulup koşturulmayacağını, birbirleriyle olan bağımlılıklarını, gruplandırılmalarını ve ne için çalıştıklarını belirtir.

- `@Test(description=`*`test ile ilgili açıklama`*`)`
  Tanım ekleme: Testin ne yaptığına dair açıklama eklenmesine olanak sağlar. Bu sayede geliştirici, dönen hata hakkında daha fazla bilgi sahibi olabilir.

- `@Test(priority=1)`
  Öncelik belirleme: Test yöntemlerinin önceliklendirilmesini ve bu sayede sırayla çalıştırılmasını sağlar. Verilen değerler kıyaslanınca, testlerin çalışma öncelikleri sıralandırılmalarına bağlıdır. Düşükten yükseğe göre daha öncelikli çalıştırılması gereken testlerdir. Yani, en küçük değerden yüksek değere göre daha öncelikli çalışılması gereken testler izlenir.

- `@Test(dependsOnMethod=`*`Login`*`)`
  Bağımlılık belirleme: Birbirine bağımlı olan test durumlarını belirtmek için bu özellik kullanılır. Örneğin; bir siteye ait profil bilgilerinin görülmesi için o siteye giriş yapılması gereksin. Bu durumda profil bilgilerini gösteren metodun çalışması, giriş metoduna bağlıdır. Profil bilgilerini gösteren metoda ait testin çalışması için de öncesinde giriş metoduna ait testin çalışması gerekmektedir.

- `@Test(enabled=`*`true`*`)`
  Aktiflik durumu belirleme: Bu özellik sayesinde, yazılan herhangi bir test metodunun çalıştırılma özelliği kontrol edilebilir. Bu özelliğin *True* değer alması durumunda, test durumu çalıştırılır. *False* değer alması durumunda ise yazılan test metodu göz ardı edilir. Bu özelliğin değeri, aksi belirtilmedikçe true'dur.

- `@Test(groups=`*`Smoke Suite`*`)`
  Gruplar: Bu özellik yardımıyla test durumları bir gruba alınabilir. Çalışması istenilen test grubunun, TestNG XML dosyasında belirtilmesi ile yalnızca bu grupta bulunan testlerin çalışması sağlanır. Bu durumda diğer testler göz ardı edilir.

- `@Test(timeout=milisaniye)`
  Zaman Aşımı: Her test senaryosunun belirlenen süre zarfında gerçekleşmesi için kullanılır. Zaman aşımı süresi milisaniye olarak belirtilir. Örneğin; belirli test durumunun yürütülmesinin yaklaşık 10-15 saniye sürdüğü ve yürütülmesi 15 saniyeden uzun sürerse testin başarısız olmasının istenildiği varsayılsın. Bu özellik

$@Test(timeout = 15000)$ olarak belirlenmelidir. Test işleminin süreyi aşması durumunda, ilerleme çubuğunun rengi yeşil yerine kırmızı olur. Birden fazla test senaryosunun, aynı süre zarfında çalışması isteniyorsa her test ek açıklamasına timeout özelliğinin eklenmesine gerek yoktur. Bu tür durumlarda $@Rule$ komutu yardımıyla, tüm test senaryoları için zaman aşımı sınırlandırılması yapılır. *@Test(timeout)* notasyonuna uygun kod bloğu, Kod Örneği 1'de gösterilmiştir.

```
1    @Rule
2    public Timeout
3    globalTimeout=Timeout.seconds(10);
4    // Her test metodu için ayrı şekilde maksimum 10 saniye çalışma süresi belirlenir.
```

Kod Örneği 1 @ParameterizedTest kod örneği

- `@Test(expected= ArithmeticException.class)`
  Test senaryosunun yürütülmesinde, istisna oluşması beklenen durumlar için bu notasyonlar kullanılır. $NoSuchMethodException$, $ArithmeticException$, $IndexOutOfBoundsExceptions$ gibi farklı istisnalar kullanılabilir. Örneğin; olarak 10 elemanlı bir dizide 11. elemana erişmek istenildiğinde, $IndexOutOfBoundsExceptions$ çıktısının gösterilmesi beklenir. Beklenen istisna oluşmazsa, test yürütülmesi başarısız olarak değerlendirilir.

- `@ParameterizedTest`
  Bu notasyon $@Test$ notasyonuna benzemektedir. Fazladan farklı test parametrelerinin tanımlanmasında da kullanılır. Bu notasyondan sonra $@ValueSource$ notasyonu kullanılabilir. Kullanılan veri tipi $String$ veya $Integer$ olabilir. Bu notasyonun temel amacı, aynı testi farklı parametrelerle daha kullanışlı bir biçimde çalıştırmaktır. Kullanılmadan önce aşağıdaki sınıfların projeye dahil edilmesi gerekmektedir. *@Parameterized* notasyonunun kullanımı Kod Örneği 2'de sunulmuştur.

```
1    @ValueSource(strings={"Parametre1", "Parametre2", "Parametre3", "Parametre4"})
2    void ExampleCode(String data)
3    {
4    assertNotNull(data);
5    }
```

Kod Örneği 2 @ParameterizedTest kod örneği

- `@RepeatedTest`
  Bu notasyon, aynı metodun birden fazla çalıştırılması gerektiği durumlarda

kullanılır. JUnit 5'te tanıtılan bu ek açıklama, gereksinime göre test yöntemini birkaç kez çalıştırmaya yarar. Bir testin geçmesi istenilen tekrar sayısı, @*RepeatedTest* notasyonunda ek açıklama olarak belirtilmelidir. *@RepeatedTest* notasyonuna ait kullanım Kod Örneği 3'te belirtildiği gibidir.

```
1    @Test
2    @RepeatedTest(5)
3    public void ToplamTest(){
4    int a=10;
5    int b=3;
6    int c;
7    c=a+b;
8    assertEquals(15,c);
9    }
```

Kod Örneği 3 @RepeatedTest kod örneği

- `@RunWith`

  Farklı test sınıflarının, tek bir sınıf içerisinde çalıştırılmasını sağlayan notasyondur. Çağrılan test sınıfları, yazıldıkları sıra ile çalıştırılır. Bu notasyonun çalıştırılması için *@SuiteClasses* komutu yazılmalıdır. Bu notasyonun kullanılması için aşağıdaki kütüphane dosyasının projeye dahil edilmesi gerekir. *@RunWith* notasyonunun kullanımı Kod Örneği 4'te gösterilmiştir.

```
1    @RunWith(Suite.class)
2    @Suite.SuiteClasses({
3    TestSinifi1.class,
4    TestSinifi2.class,
5    TestSinifi3.class
6    })
```

Kod Örneği 4 @RunWith kod örneği

- `@Parameters`

  @ParameterizedTest ile *@Parameters* notasyonunun birbirleriyle karıştırılmaması gerekir. @ParameterizedTest notasyonundan sonra @ValueSource kullanılması gerekirken, *@Parameters* notasyonundan sonra @Runwith notasyonu kullanılmalıdır. *@Parameters* notasyonu, parametrelerin belirlenmesini sağlayan fonksiyonların başında yazılmalıdır. Kod Örneği 5'te parametre değerleri, *@Parameters* ile belirlenen fonksiyon ile oluşturulur. Bu notasyon ile testin farklı test parametreleri ile tekrar tekrar çalışması mümkün hale gelir. Bu sayede test ekibinin zamandan tasarrufu sağlanmış olur. *@Parameterized* notasyonunun kullanımı Kod Örneği 5'te

belirtilmiştir.

```
@RunWith(Parameterized.class)
public classs testBlok{
String kullaniciAdi, sifre;
@Parameters
public Object[][]
getData()
{
Object[][]
    info={{"ayse", "Kahveci"},
          {"admin", "123"}};
    return info;
}
public testBlok (String isim, String parola)
{
this.kullaniciAdi=isim;
this.sifre=parola;
}
@Test
public void Sample()
{
SampleCode.login(kullaniciAdi, sifre);
}

}
```

Kod Örneği 5 @Parameterized kod örneği

9.2.2. Çalışma Sırasının Belirlenmesini Sağlayan Notasyonlar

Çalışma sırasının belirlenmesinde kullanılan notasyonlar aşağıda açıklanmıştır.

- `@BeforeMethod Notasyonu`
  Bu notasyonun tanımlandığı yöntem, her test yönteminden önce çalıştırılır. Veri tabanı sorgusunun sağlıklı bir şekilde çalıştırılabilmesi için öncelikle veri tabanı bağlantısının gerçekleştirilmesi bu duruma örnektir. Veri tabanı bağlantısını gerçekleştiren metodun her metottan önce çalıştırılması gerekir. Böyle durumlarda bu notasyon, veri tabanı bağlantısının gerçekleştirildiği metot üstüne eklenmelidir.

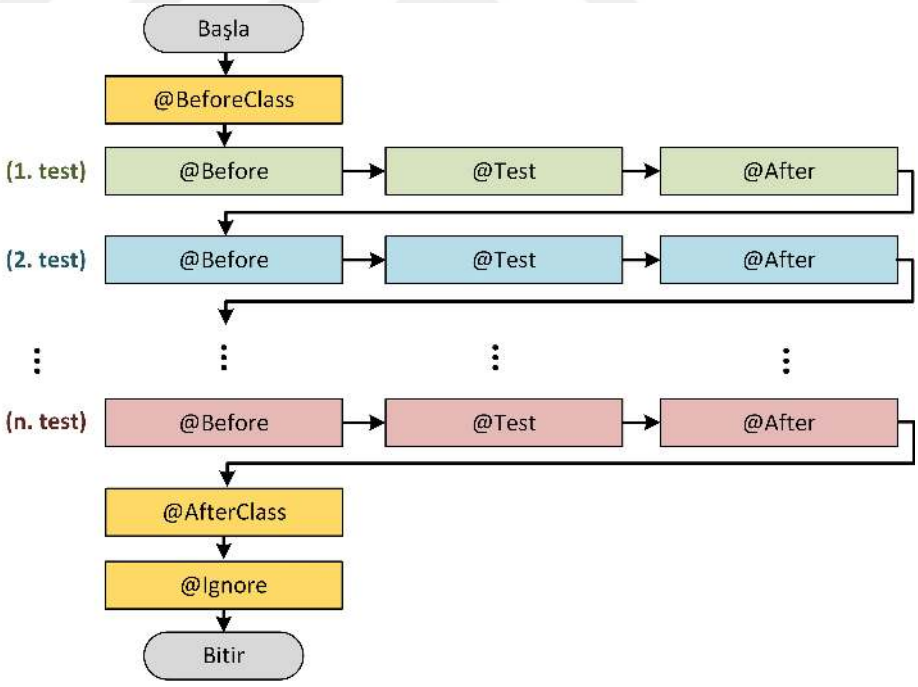
- `@AfterMethod Notasyonu`
  Bu notasyonun tanımlandığı yöntem, her test yönteminden sonra çalıştırılır. Veri tabanı sorgusunun sağlıklı bir şekilde çalıştırılabilmesi için her sorgudan sonra veri tabanı bağlantısının koparılması bu duruma örnektir. Veri tabanı bağlantısını sonlandıran metodun her metottan sonra çalıştırılması gerekir. Böyle durumlarda bu notasyon, veri tabanı bağlantısının koparılmasını sağlayan metot üstüne eklenmelidir.

- `@BeforeClass`
  Tüm metotlardan önce, sadece bir kez çalıştırılması istenilen metodun başına eklenir. Testler öncelik sırasına göre çalıştırılır. *BeforeClass* etiketli test ise *Test(priority=1)* ile belirtilen testten de önce çalıştırılır.

- `@AfterClass`
  Var olan bütün testler bittikten sonra bu metot çalışır. Tüm test senaryolarının yürütülmesinden sonra kaynakların serbest bırakılması için bu metottan faydalanılır. Sınıf içindeki tüm test metotlarının bitmesi sonucu devreye girerek ne yapılması gerekiyorsa onu yapar.

- `@Ignore`
  Yazılan test bloğunun göz ardı edilmesi gereken durumlarda kullanılması gereken bir notasyondur. *Test(active='false')* ile aynı işlevi yerine getirir. Ancak *Test(active='false')* ifadesi, sınıfta var olan tüm testlerin gözardı edilmesinde kullanılamazken; *@Ignore* notasyonunun, sınıf tanımlamasından önce kullanılması ile sınıfta var olan tüm testler göz ardı edilebilir.

Açıklamaları yapılan notasyonların çalışma düzeni Şekil 9.4.'te ifade edilmiştir:



Şekil 9.4. Çalışma sırasına göre notasyonlar

9.3. JUnit Assert Metotları

JUnit programlamada *Assertions* yapısı, yazılan test senaryosu sonucu elde edilen çıktı ve beklenen sonucun karşılaştırılması için kullanılır. Kısacası *Assert* metotları, JUnit

notasyonlarıyla yazılan testlerin kontrolünü sağlamak için kullanılır. Test senaryosunun sorunsuz çalışması ile elde edilen sonuç ve beklenen sonucun aynı olması, test işleminin başarılı olduğunu gösterir. *Assertion* kullanımlarında, beklenen sonucun ve test çıktısının tamamen aynı olması gerekmektedir.

Test senaryolarındaki kullanımlarına göre iki tür assertions yapısı mevcuttur: *Hard Assertions* ve *Soft Assertions*. Bu yapılardan *Hard Assertions* yapısı, test senaryosunda bulunan onaylama şartı gerçekleştirilmediği takdirde, testin bir sonraki adıma geçişine izin verilmez. Bu durumda *Exception* yapısı ile test işlemi başarısız olarak sonuçlandırır. *Soft Assertions* yapısı ise test senaryosunda bulunan testlerin çalıştırılması, önceki çalışan testlerin başarılı olup olmama durumundan bağımsız olarak sürdürülür ve herhangi bir Exception yapısının kullanılmasına gerek kalmadan sonraki test senaryosu adımı devam ettirilir.

- `AssertAll(grupHataMesajı, assertionGrup)`
  Bu metot assertionların gruplandırılarak çalıştırılmasını sağlar. Gruplanarak çalıştırılan assertionlar sonucu fırlatılan hataMesajı hepsi için ortak olarak görüntülenir. İlk parametre olarak hata mesajı belirtilirken, ikinci parametre olarak çalıştırılması gereken assert grubu tanımlanır. İkinci parametre olarak tanımlanan assertionGrup parametresi *Executable*, *Stream* veya *Collection* tiplerinde oluşturulmuş assert grupları olabilir. *AssertAll* örnek kullanımı Kod Örneği 6'da verilmiştir.

```
1    public void cokluAssert(){
2        assertAll(
3        "grupAssert",
4        0->assertEquals(4, 2*2, "2 kere 2, 4 eder."),
5        0->assertEquals("java", "JAVA".toLowerCase()),
6        0->assertEquals(null, null, "null değer, null değere eşittir.")
7        );
8    }
```

Kod Örneği 6 AssertAll kod örneği

- `AssertEquals (hataMesajı, beklenenDeğer, gerçekDeğer)`
  Beklenen değer ve gerçek sonucun karşılaştırılması için kullanılır. Bu yapı belirlenen iki nesnenin eşit olması durumunu kontrol eder. Bu değerlerin eşit olmaması durumunda yani karşılaştırılmanın eşit olmadığı durumda *AssertionError* parametresi ile detayları belirtilen bir hata mesajı tanımlanır. Ayrıca bu yapının kullanılması sırasında *AssertionError* yanında, iki parametrenin daha tanımlanması gerekmektedir. Bu parametreler, karşılaştırma için kullanılan beklenen ve gerçek değeri ifade ederler. Opsiyonel olan hataMesajı parametresinin tipi String iken, beklenen ve gerçek değerler farklı veri tiplerinden olabilir.

- `AssertNotEquals (hataMesajı, beklenmeyenDeğer, gerçekDeğer)`
  Belirlenen iki nesnenin eşit olmama durumunun kontrol edilmesinde kullanılan metottur. Bu metot ile kullanılması gereken parametrelere bakılacak olursa, hataMesajı parametresi ile metodun doğrulanması sonucu kullanılacak hata mesajı tanımlanır. beklenmeyenDeğer parametresi, eşitliğin olmaması gereken değeri, gerçekDeğer ise kontrol edilecek değeri ifade eder. Bu metot, diğer metotlar gibi hataMesajı parametresi yazılmadan kullanılabilir. Bu şekilde kullanıldığında, eşitsizliğin olması halinde ekranda bir mesaj belirtilmez.

- `AssertArrayEquals (hataMesajı, dizi1, dizi2)`
  Bu metot ile iki dizi yapısının birbirine eşit olma durumu kontrol edilir. Bu kontrol metodu ile her iki dizinin de parametre türleri, eşit sayıda hücreye sahip olması ve hücrelerde bulunan değerlerin aynı olması dikkate alınır. Her iki dizide de null değeri varsa bunlar da eşit olarak kabul edilir. Dizilerin eşit olmaması halinde hataMesajı parametresi ile tanımlanan hata mesajı görüntülenir.

- `AssertIterableEquals (hataMesajı, beklenmeyenDeğer, gerçekDeğer)`
  Bu metot ile iterasyonların aynı değeri alması ve eşit olup olmadığı kontrol edilir. Kontrol edilen her iki tekrarlanan öğenin aynı sırayla, eşit öğeleri döndürmesi gerekmektedir. Söz konusu metodun başarılı olabilmesi için aynı türden listelerin olması gerekmez. Farklı türden oluşan fakat aynı iterasyonlarda aynı değeri döndüren listelerin, bu metotla kontrol edilmesi başarılı sonuç verir.

- `AssertLinesMatch (beklenenListe, gerçekListe)`
  Bu metot, parametre olarak aldığı iki listenin, aynı sıradaki elemanlarının aynı olup olmadıklarını kontrol eder. beklenenListe ve gerçekListe String tipinde olmalıdır. assertIterableEquals'tan farkı, bu metod ilgili indisteki elemanların eşit olmaması durumunda ifadenin regüler olmasını kontrol edip, buna göre sonuç üretir. Örneğin */ /d+* regüler ifadesi ile bir tam sayı değeri bu metot için eşit kabul edilir. Ayrıca bu metot ile satır atlamaları da mümkündür. Temel olarak aşağıdaki algoritma takip edilerek gerçekleşir:

  – İki eleman aynı mı kontrol et, elemanlar aynı ise sonraki satırlara bak.

  – Aynı değilse, String'i düzenli ifade olarak ele al ve tekrar kontrol et, aynı ise sonraki satırlara bak.

  – Aynı değilse, elemanın satır atlama komutu olup olmadığını kontrol et. Satır atlama komutu var ise ilgili satır kadar atla (örneğin *»5»* için 5 satır sonrası) ve 1. adıma git.

  AssertLinesMatch metodunun kullanım şekli Kod Örneği 7'de gösterilmiştir.

```
1    @Test
2    public void ListeAssert(){
3    List<String>beklenen=asList("Java","\\d+","JUnit");
4    List<String>gercek=asList("Java", "11", "Junit");
5
6    assertLinesMatch(beklenen, gercek);
7    }
```

Kod Örneği 7 AssertLinesMatch kod örneği

- `AssertTrue (kontrolEdilenİfade, gösterilecekMesaj)`
  Beklenen bir sonucun True değer olması kontrol edilirken kullanılır. Parametre olarak iki değer alır. İlk parametre, mesajın gönderildiği parametredir. İkinci parametre ise gönderilen mesajın doğruluğu için belirtilen koşulu ifade eder. Söz konusu ikinci parametre, *AssertionError()* görevine benzer olarak ekranda gösterilecek mesajı yazdırır. Kısacası iki parametre belirtilir ve tek bir değişkenin doğruluğu yansıtma durumu kontrol edilir.

- `AssertFalse (kontrolEdilenİfade, gösterilecekMesaj)`
  Bu metot *assertTrue()* gibi kullanılır. Tek farkı *assertTrue()* metodunda belirlenen değerin doğruluğu kontrol edilirken, *assertFalse()* metodunda belirlenen değerin yanlış olması kontrol edilir. Beklenen bir sonuç veya yargının yanlış olma durumu dikkate alınır. *AssertTrue()* metodu gibi iki parametre alır. Bu parametrelerden biri koşul ifadesi, diğeri mesajdır. *AssertFalse()* metodu ile koşul yerine getirilmezse mesaj parametresi sayesinde assertionError() hatası fırlatılır.

- `AssertNull (kontrolEdilenNesne, gösterilecekMesaj)`
  Bu metot ile belirtilen nesnenin ya da değişkenin başlangıç değeri veya mevcut değerindeki boş olma durumu kontrol edilir. Bu metot kullanılırken bir nesne parametre olarak alınır ve eğer nesne null değilse *assertionError* ile hata mesajı gönderilir.

- `AssertNotNull (kontrolEdilenNesne, gösterilecekMesaj)`
  Bu metot yardımıyla *AssertNull()* metodunda olduğu gibi belirtilen parametrenin veya nesnenin değerine bakılır ve sonucun null olmaması halinde başarılı geri dönüş yapar. Kontrol edilen parametre null ise assertionError ile hata mesajı gönderir.

- `AssertSame (gösterilecekMesaj, beklenenNesne, gerçekNesne)`
  Bu metot yardımıyla, farklı parametre olarak belirlenen iki nesnenin, beklenen değer ve gerçek değerin aynı nesneden türemesi durumu kontrol edilir. Nesnelerin aynı nesneden gelmemesi halinde assertionError hata mesajı gönderilir.

- `AssertNotSame (gösterilecekMesaj, beklenmeyenNesne, gerçekNesne)`
  Parametre olarak belirtilen iki nesnenin aynı nesneye karşılık gelmemesi durumu

kontrol edilir. *AssertSame()* yapısının tersi olan kontrollerde kullanılan bir metottur. Nesneler aynı nesneye karşılık geliyorsa hata mesajı gönderilir.

- `AssertTimeOut (belirlenenÇalışmaSüresi, çalıştırılacakFonksiyon)`
  Bir assert komutunun, çalışma işleminin belirli bir sürede bitmesini kontrol etmek amacıyla kullanılır. Bu metot, çağrıldığı fonksiyon ile aynı iş parçacığında yürütülür. Dolayısıyla fonksiyonda belirtilen sürenin aşıma uğraması durumunda fonksiyonların çalışması iptal olmaz. *AssertTimeOut* metodunun kullanım şekli Kod Örneği 8'de belirtildiği gibidir.

```
1   @Test
2   public void ZamanAsimiAssert(){
3       assertTimeout(
4       ofSeconds(2),
5       0->{
6       //2 dakikadan daha az süre içerisinde çalışması gereken kod bloğu
7       Thread.sleep(1000);
8       }
9       );
10  }
```

Kod Örneği 8 AssertTimeOut kod örneği

- `AssertTimeoutPreemptively (belirlenenÇalışmaSüresi, çalıştırılacakFonksiyon)`
  Bir assert komutunun, çalışma işleminin belirli bir sürede bitmesini kontrol etmek amacıyla kullanılır. Bu metot, çağrıldığı fonksiyon ile farklı iş parçacığında yürütülür. *AssertTimeOut* metodundan farklı olarak, belirtilen sürenin aşıma uğraması durumunda fonksiyonların çalışması iptal olur.

- `Fail(hataMesajı)`
  Testin doğruluğu ne olursa olsun ekrana hata döndürmesi için kullanılan metottur. Test işleminin, kasıtlı olarak hata üretmesini sağlar. Genellikle, tamamlanmamış veya yazımı devam eden test fonksiyonları için kullanılır. *Fail* metodunun kullanımı Kod Örneği 9'da gösterilmiştir.

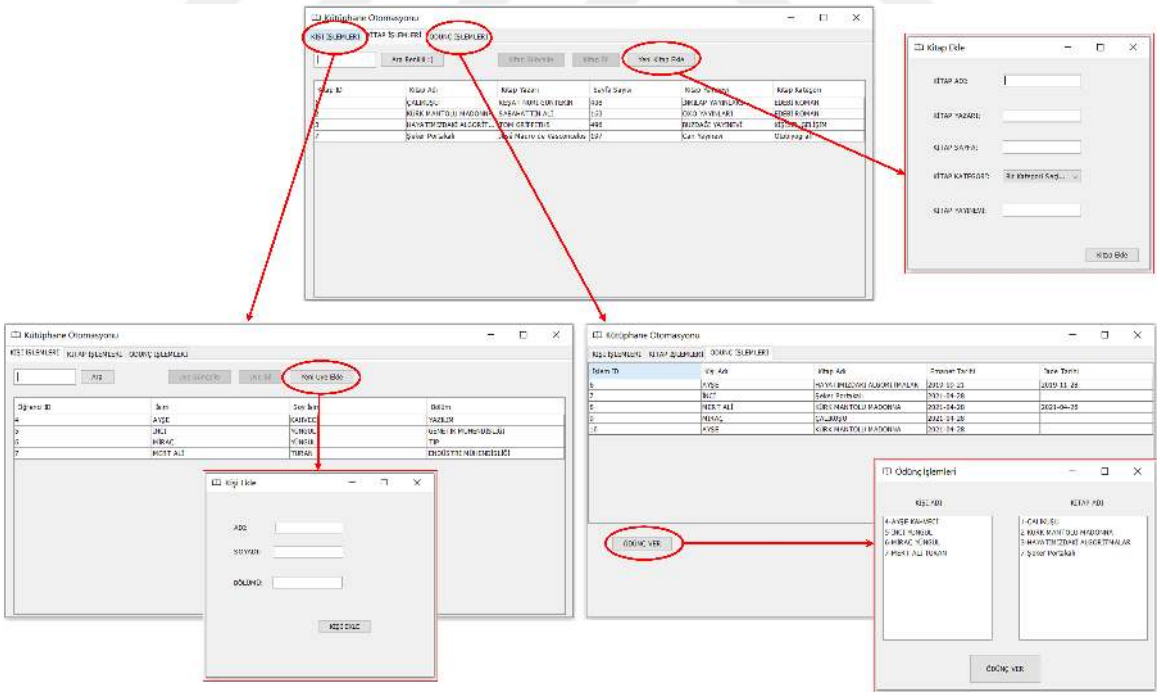
```
1   @Test
2   public void HataMesaji(){
3   //Test tamamlanamadı.
4   fail("HATA-Test tamamlanamadı.");
5   }
```

Kod Örneği 9 Fail kod örneği

9.4. Test Uygulamaları

Tez kapsamında gerçekleştirilen uygulama ile temel düzeyde bir kütüphane otomasyonu sürekli entegrasyon mantığı çerçevesinde Jenkins platformu aracılığıyla test edilmiştir. Geliştirilen projede test sınıfı yazılarak, derleme işleminden sonra Jenkins ile test sınıflarının çalıştırılması sağlanmıştır. Bu sayede sistemin geliştirilmesi ve güncellenmesi gibi durumlarda meydana gelebilecek hataların erken safhalarda ortaya çıkması amaçlanmaktadır.

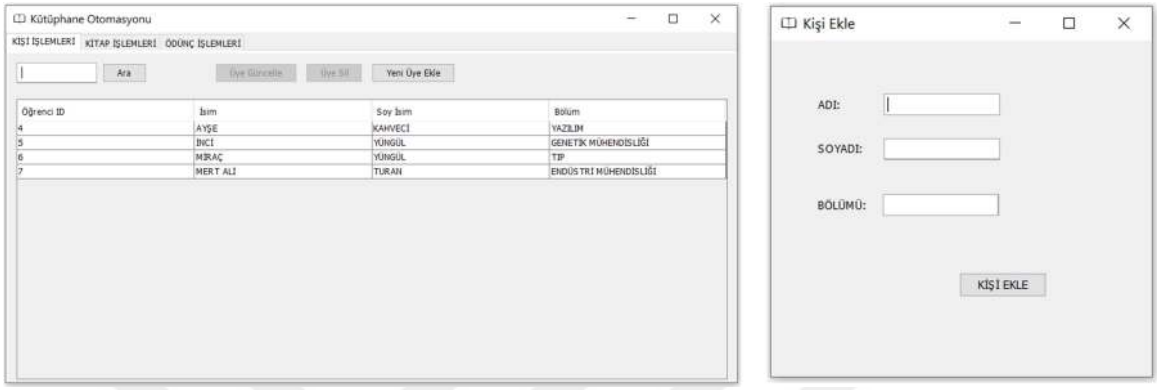
Geliştirilen uygulama kapsamında JUnit ile Java'da birim testleri uygulanmıştır. Ardından bu testler Jenkins test platformu aracılığıyla sürekli entegrasyon mantığı ile gerçekleştirilmek için Jenkins Pipeline mimarisi ile bağlanmıştır. Proje olarak temel fonksiyonlara sahip bir kütüphane otomasyonu seçilmiştir. Geliştirilen uygulama aracılığıyla kişiler ve kitaplar, veri tabanına eklenebilmektedir. Ardından kullanıcı, belirlenen kişiye istenilen kitabı verebilmektedir. Başkaları tarafından alınan kitaplar bir ekrandan izlenebilmekte ve kitap iade işlemleri gerçekleştirilebilmektedir. Uygulama kapsamında kitap, kişi ve ödünç bilgilerinin görüntülendiği temelde üç adet panel oluşturulmuştur. Üye ekleme, kitap ekleme ve ödünç verme işlemleri için ise üç adet pencere oluşturulmuştur. Oluşturulan projeye ait arayüzler ve program akışı Şekil 9.5.'te verilmiştir.



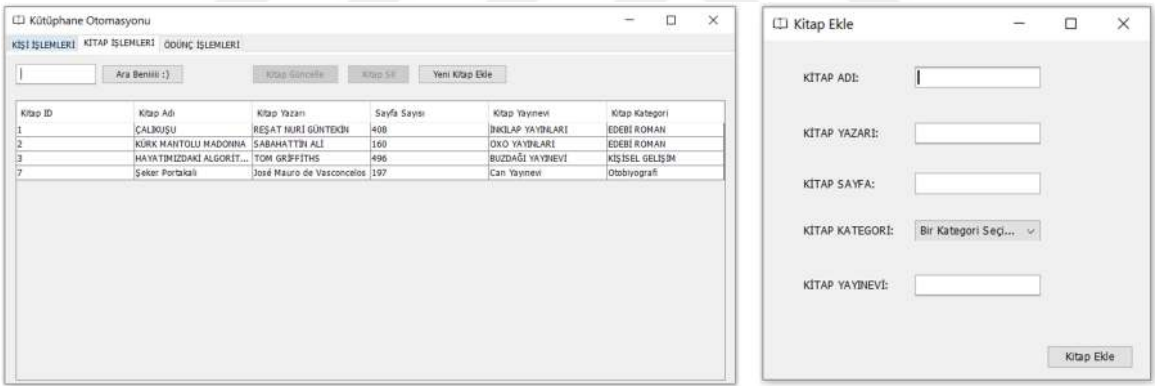
Şekil 9.5. Uygulamada bulunan arayüz geçişleri

9.4.1. Arayüzler

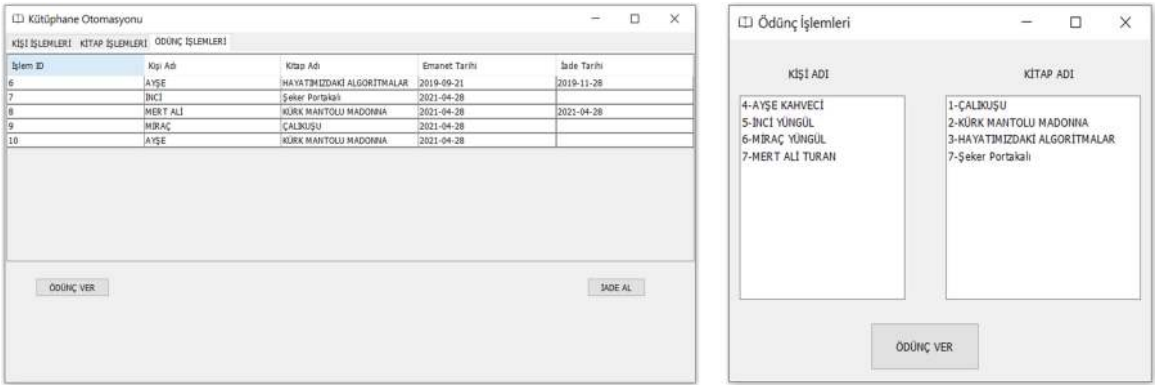
Projede bir ana pencere, üç panel ve üç de frame olmak üzere toplam yedi grafik kullanıcı arayüzü mevcuttur. Kitap, kişi ve ödünç işlemleri için oluşturulan arayüz geçişleri kişi işlem arayüzü Şekil 9.6.'da, kitap işlem arayüzü Şekil 9.7.'de ve ödünç işlem arayüzü Şekil 9.8.'de sunulmuştur.



Şekil 9.6. Uygulamada bulunan kişi işlem arayüzü



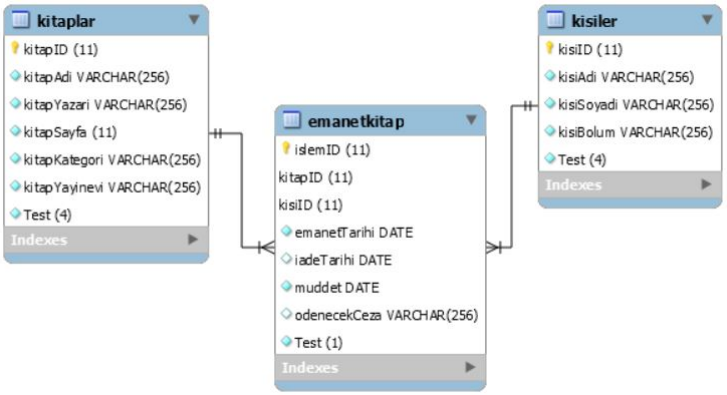
Şekil 9.7. Uygulamada bulunan kitap işlem arayüzü



Şekil 9.8. Uygulamada bulunan ödünç işlem arayüzü

9.4.2. Veri Tabanı Tasarımı

Uygulamada temel olarak kitap, kişi ve emanet bilgileri tutulmaktadır. Bu bilgilerin tutulması için MySql veri tabanı seçilmiştir. Toplamda üç adet veri tabanı tablosuna ihtiyaç duyulmuştur. Kitaplar tablosunda, kitapların adı, yazarı, sayfa sayısı, kategorisi ve yayınevi bilgileri yer almaktadır. Kişiler tablosunda, kişilerin isim, soyisim ve bölüm bilgilerini tutmak yeterli görülmüştür. Bu iki tablo arasında m*n ifadesiyle tanımlanabilen bir ilişki yer almaktadır. Bu ilişki tablosunda emanet tarihi, iade tarihi, müddet ve ödenecek ceza bilgileri de tutulmaktadır. Projede kullanılan tablolar ve ilişkileri Şekil 9.9.'da verilmiştir.



Şekil 9.9. Uygulama veri tabanı tablosu

Veri tabanına kontrollü testlerin gerçekleştirilmesi adına *testMi* alanları eklenmiştir. Bu alan 1 ise veri tabanındaki kayıt test verisine ait anlamına gelmektedir. Var olan üç tablo için de üçer adet test verisi eklenmiştir. Veritabanına kontrollü testlerin gerçekleştirilmesini ifade eden veriler Tablo 9.2., Tablo 9.3. ve Tablo 9.4.'te gösterilmiştir:

Tablo 9.2. Veri tabanına kontrollü testlerin gerçekleştirilmesi için eklenen kişi verileri

kisiAdi	**kisiSoyadi**	**kisiBolum**	**Test**
KisiAdi Test 1	kisiSoyadi Test 1	kisiBolum Test 1	1
KisiAdi Test 2	kisiSoyadi Test 2	kisiBolum Test 2	1
KisiAdi Test 3	kisiSoyadi Test 3	kisiBolum Test 3	1

Tablo 9.3. Veri tabanına kontrollü testlerin gerçekleştirilmesi için eklenen kitap verileri

kitapAdi	**kitapYazari**	**kitapSayfa**	**kitapKategori**	**KitapYayinEvi**	**Test**
kitapAdi Test 1	kitapYazari Test 1	1	kitapKategori Test 1	kitapYayinevi Test 1	1
kitapAdi Test 2	kitapYazari Test 2	2	kitapKategori Test 2	kitapYayinevi Test 2	1
kitapAdi Test 3	kitapYazari Test 3	3	kitapKategori Test 3	kitapYayinevi Test 3	1

Tablo 9.4. Veri tabanına kontrollü testlerin gerçekleştirilmesi için eklenen ödünç verileri

kisiID	**kitapID**	**emanetTarihi**	**iadeTarihi**	**muddet**	**odenecekCeza**	**Test**
3	1	2021-01-07	NULL	2021-01-22	NULL	1
3	2	2021-01-07	NULL	2021-01-22	NULL	1
3	3	2021-01-07	NULL	2021-01-22	NULL	1
2	1	2021-01-07	NULL	2021-01-22	NULL	1
1	1	2021-01-03	NULL	2021-01-18	NULL	1

9.4.3. Maven Konfigürasyonu

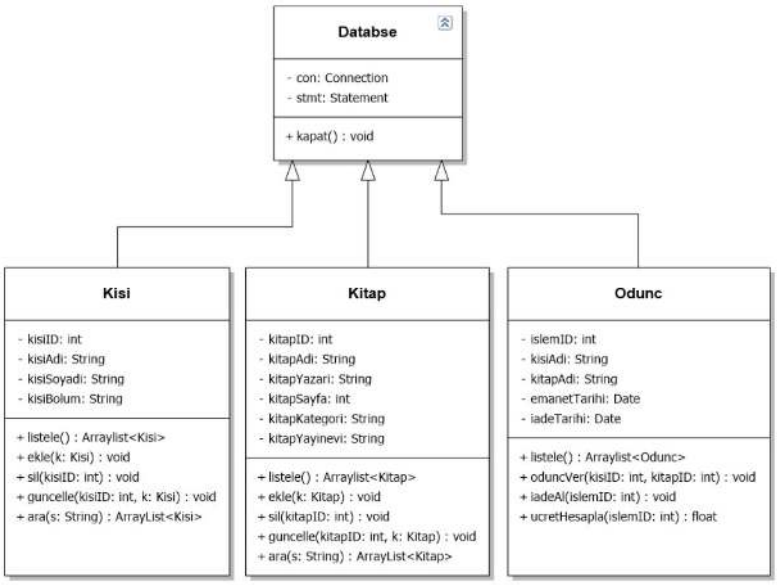
Projede faydalanılan Maven aracı yardımıyla kullanılacak kütüphanelerin otomatik olarak indirilip projeye dahil edilmesi sağlanmıştır. Maven tarafından projeyi çalıştırılırken öncelikle yerel depoda ihtiyaç duyulan kütüphaneler aranır, yerel depoda yer almayan kütüphaneler için global depoya bakılır. Global depodan ilk seferde indirilen kütüphaneler yerel depoya kaydedilir. Sonraki seferlerde indirilmeden yerel depodaki kütüphaneler kullanılır. Projede MySQL ve JUnit kütüphanelerine ihtiyaç duyulduğundan bu kütüphaneler dependencies altına eklenmiştir. Proje için gerekli Maven konfigürasyonunun yapılandırılması için dependencies altına eklenmesi gereken kod bloğu Kod Örneği 10'da verilmiştir.

```xml
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.19</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.5.2</version>
    </dependency>
</dependencies>
```

Kod Örneği 10 Proje için gerekli Maven konfigürasyonunun yapılandırılması

9.4.4. Modellerin Oluşturulması ve Testlerin Yazılması

Projenin geliştirilmesi için model yapılarından yararlanılmıştır. Modeller bir uygulamada temel işlemleri gerçekleştiren sınıflardır. Otomasyon uygulamalarında modeller, genellikle veri tabanı işlemlerini gerçekleştirirler. Veri tabanına veri eklemek veya veri tabanından veri çekmek model sınıflarının görevidir. Bunun için uygulamada 1 adet genel veri tabanı bağlantısını gerçekleştirecek sınıf tanımlanmıştır. Diğer model sınıfları bu sınıftan kalıtım almaktadır. Database sınıfının içinde veri tabanı bağlantısını sağlamak ve sorgu yazmak için kullanılacak Connection ve Statement nesneleri oluşturulmuştur. Ardından Kişi, Kitap ve Ödünç sınıfları bu sınıftan türetilmiştir. Her sınıfta veri tabanındaki alanlara karşılık gelecek özellikler tanımlanmıştır. Kişi ve Kitap sınıfları için listele, ekle, sil, güncelle ve ara fonksiyonları yazılmıştır. Ödünç sınıfı için ise listele, oduncVer, iadeAl, ucretHesapla gibi fonksiyonlar yazılmıştır. Modeller oluştururken testMi parametresi tanımlanmıştır. testMi parametresi 1 olduğu zaman modeller veri tabanında test olarak işaretlenmiş alanları getirmekte, bu sayede testlerin kontrolü sağlanabilmektedir. Varsayılan yapılandırıcılar ile oluşturulduğunda testMi özelliği false olarak tanımlanmakta, bu sayede normal veriler için kullanılmaktadır. Test işlemi için ise nesne oluşturulduktan hemen sonra testMi özelliğinin true yapılması gerekmektedir. Projedeki modellere ait diyagramını Şekil 9.10. içermektedir.



Şekil 9.10. Uygulamaya ait tablolar arası ilişki diyagramı

Kişi Modeli

- *Listele Metodu:* Veri tabanındaki tüm kişi kayıtlarını getirmek için yazılan metottur. Bu metot parametre almaz ve Arraylist<Kisi> tipinde bir liste döndürür. Veri tabanında kayıtlı kişilerin tümünün listelenmesini sağlar. testMi parametresi 1 olduğunda sadece test verileri çekilir, diğer durumlarda uygulama verileri getirilir. Kişi listeleme metoduna ait kodlar Kod Örneği 11'de verilmiştir.

```
1  public ArrayList<Kisi> listele() throws Exception{
2      ResultSet rs=stmt.executeQuery("select * from kisiler WHERE Test = "+testMi);
3      ArrayList<Kisi> elemanlar = new ArrayList<Kisi>();
4      while(rs.next())  {
5          Kisi t = new Kisi();
6          t.kisiID = rs.getInt("kisiID");
7          t.kisiAdi = rs.getString("kisiAdi");
8          t.kisiSoyadi = rs.getString("kisiSoyadi");
9          t.kisiBolum=rs.getString("kisiBolum");
10         elemanlar.add(t);
11     }
12     return elemanlar;
13 }
```

Kod Örneği 11 Kişi listeleme metodu

Kişi modeli içinde yer alan kişi listele metoduna ait test kodu Kod Örneği 12'de gösterilmiştir.

```
@Test
public void testListele() throws Exception {
    System.out.println("listele");

    ArrayList<Kisi> veritabaniKisiler = k.listele();
    ArrayList<Kisi> kontrolKisiler = new ArrayList<>();
    kontrolKisiler.add(new Kisi(1,"kisiAdi Test 1","kisiSoyadi Test 1",
    "kisiBolum Test 1"));
    kontrolKisiler.add(new Kisi(2,"kisiAdi Test 2","kisiSoyadi Test 2",
    "kisiBolum Test 2"));
    kontrolKisiler.add(new Kisi(3,"kisiAdi Test 3","kisiSoyadi Test 3",
    "kisiBolum Test 3"));

    for (int i = 0; i < veritabaniKisiler.size(); i++) {
      assertEquals(veritabaniKisiler.get(i).kisiAdi,
      kontrolKisiler.get(i).kisiAdi);
      assertEquals(veritabaniKisiler.get(i).kisiSoyadi,
      kontrolKisiler.get(i).kisiSoyadi);
      assertEquals(veritabaniKisiler.get(i).kisiBolum,
      kontrolKisiler.get(i).kisiBolum);
    }
}
```

Kod Örneği 12 Kişi listele metodu için yazılan test kodu

- *Ekle Metodu:* Veri tabanına yeni kişilerin kayıtlarını eklemek için yazılan metottur. Kişilere ait ad, soyad, bölüm gibi parametreleri kullanır. Parametre olarak gelen kişi nesnesi veri tabanına kaydedilir. Yeni kişilerin kaydı için kullanılan kişi ekle metoduna ait kod bloğu, Kod Örneği 13'te gösterilmiştir.

```
public void ekle(Kisi k) throws SQLException {
    //testMi=1 ise kisiID = "999" degilse kisiID = "null" yap
    String kisiID = (testMi==1)?"999":"null";

    String query = " insert into kisiler (kisiID, kisiAdi, kisiSoyadi, kisiBolum,
    Test)"
    + " values ("+kisiID+", '"+k.kisiAdi+"', '"+k.kisiSoyadi+"', '"+k.kisiBolum+"',
    "+testMi+")";

    stmt.execute(query);
}
```

Kod Örneği 13 Kişi ekle metodu

- *Sil Metodu:* Veri tabanında var olan kişilerin kayıtlarını silmek için yazılan metottur. kisiID parametresi ile var olan kişilerin bilgilerini veri tabanından siler. Veri tabanında kayıtlı kişilerin silinmesi için yazılan kişi sil metodu Kod Örneği 14'de verilmiştir.

```
public void sil(int kisiID) throws SQLException {
    String query = " delete from kisiler where kisiID = "+kisiID;

    stmt.execute(query);
}
```

Kod Örneği 14 Kişi sil metodu

Kişi ekle ve kişi sil metotları için yazılan test kodu Kod Örneği 15'te gösterildiği gibidir.

```
public void testEkleSil() throws Exception{
    System.out.println("ekle");

    Kisi a =new Kisi(999, "Ayse", "Kahveci", "Yazilim Muhendisligi");
    k.ekle(a);
    k.sil(999);
}
```

Kod Örneği 15 Kişi ekle ve sil metodu için yazılan test kodu

- *Güncelle Metodu:* Veri tabanında var olan kişilerin mevcut bilgilerini değiştirmek için yazılan metottur. kisiID'ye denk gelen satırın bilgilerini Kisi nesnesinin bilgileri ile günceller. Kişi bilgilerinin güncellendiği kod bloğı Kod Örneği 16'da verilmiştir.

```
public void guncelle(int kisiID, Kisi k) throws SQLException {

    String query = " update kisiler set kisiAdi='"+k.kisiAdi+"',
    kisiSoyadi='"+k.kisiSoyadi+"', "
            + "kisiBolum='"+k.kisiBolum+"' where kisiID = "+kisiID;

    stmt.execute(query);
}
```

Kod Örneği 16 Kişi güncelle metodu

Veri tabanı kayıtlarında bilgilerin güncellenmesini sağlayan metot için yazılan kodlar Kod Örneği 17'de sunulmuştur.

```java
@Test
public void testGuncelle() throws Exception{
    System.out.println("guncelle");
    Kisi a=new Kisi(15,"Incisu","Yungul","Matematik");
    k.ekle(a);
    a.kisiAdi="Mirac";
    k.guncelle(999, k);
    k.sil(999);
}
```

Kod Örneği 17 Kişi güncelle metodu için yazılan test kodu

- *Ara Metodu:* Bu metot, kişiler listesinde bulunan ve belirli bir veriyle eşleşen kaydın veya kayıtların listelenmesi için kullanılır. Kişi adı veya soyadında parametre olarak girilen değeri arar. Kişiler listesini belirli parametreye göre sıralayan ara metodu için yazılan kod bloğu Kod Örneği 18'de verilmiştir.

```java
public ArrayList<Kisi> ara(String aranacakIfade) throws Exception{
    ResultSet rs=stmt.executeQuery("select * from kisiler "
            + "where (kisiAdi LIKE '"+aranacakIfade+"%'
            or kisiSoyadi LIKE '"+aranacakIfade+"%') "
            + "AND Test="+testMi);

    ArrayList<Kisi> elemanlar = new ArrayList<Kisi>();
    while(rs.next())  {
        Kisi t = new Kisi();
        t.kisiID = rs.getInt("kisiID");
        t.kisiAdi = rs.getString("kisiAdi");
        t.kisiSoyadi = rs.getString("kisiSoyadi");
        t.kisiBolum=rs.getString("kisiBolum");
        elemanlar.add(t);
    }
    return elemanlar;
}
```

Kod Örneği 18 Kişi ara metodu

Kişi ara metodu için yazılan test bloğu Kod Örneği 19'da gösterilmiştir.

```java
@Test
public void testAra() throws Exception{
    System.out.println("ara");
    ArrayList<Kisi> Kisiler = k.ara("kisiAdi Test 1");

    ArrayList<Kisi> kontrolKisiler = new ArrayList<>();
    kontrolKisiler.add(new Kisi(1,"kisiAdi Test 1","kisiSoyadi Test 1",
    "kisiBolum Test 1"));

    for (int i = 0; i < Kisiler.size(); i++) {
        assertEquals(Kisiler.get(i).kisiAdi, kontrolKisiler.get(i).kisiAdi);
        assertEquals(Kisiler.get(i).kisiSoyadi,kontrolKisiler.get(i).kisiSoyadi);
        assertEquals(Kisiler.get(i).kisiBolum, kontrolKisiler.get(i).kisiBolum);
    }
}
```

Kod Örneği 19 Kişi ara metodu için yazılan test kodu

Kişi için oluşturulan model sınıf ve bu sınıf için yazılan testler yukarıda verilmiştir. Kitap için de benzer model ve test sınıfları oluşturulmuştur. Ancak ödünç sınıfı için oluşturulan metotlar farklılık göstermektedir.

Ödünç Modeli

- *Listele Metodu:* Veri tabanındaki ödünç kayıtlarını getirmek için yazılan metottur. Bu metot parametre almaz ve Arraylist<Odunc> tipinde bir liste döndürür. Veri tabanında kayıtlı kişilerin tümünün listelenmesini sağlar. testMi parametresi 1 olduğunda sadece test verileri çekilir, diğer durumlarda uygulama verileri getirilir. Ödünç modeli için oluşturulan listeleme metoduna ait kod blokları, Kod Örneği 20'de sunulmuştur.

```
public ArrayList<Odunc> listele() throws Exception{
    ResultSet rs=stmt.executeQuery("SELECT * FROM emanetkitap "
            + "INNER JOIN kitaplar ON kitaplar.kitapID = emanetkitap.kitapID "
            + "INNER JOIN kisiler ON kisiler.kisiID = emanetkitap.kisiID "
            + "HAVING emanetkitap.Test="+testMi);

    ArrayList<Odunc> elemanlar = new ArrayList<Odunc>();
    while(rs.next())  {
        Odunc t = new Odunc();
        t.kisiAdi = rs.getString("kisiAdi");
        t.kitapAdi=rs.getString("kitapAdi");
        t.emanetTarihi=rs.getString("emanetTarihi");
        t.iadeTarihi=rs.getString("iadeTarihi");
        t.islemID=rs.getInt("islemID");
        elemanlar.add(t);
    }
    return elemanlar;
}
}
```

Kod Örneği 20 Ödünç listeleme metodu

Ödünç modeline ait listeleme metodu için yazılan test kodu Kod Örneği 21'de gösterilmiştir.

```java
@Test
public void testListele() throws Exception {
    System.out.println("listele");

    ArrayList<Odunc>  veritabaniOdunc = o.listele();

    ArrayList<Odunc> kontrolOdunc = new ArrayList<>();
    kontrolOdunc.add(new Odunc("kisiAdi Test 3","kitapAdi Test 1", "2021-01-07",null));
    kontrolOdunc.add(new Odunc("kisiAdi Test 3","kitapAdi Test 2", "2021-01-07",null));
    kontrolOdunc.add(new Odunc("kisiAdi Test 3","kitapAdi Test 3", "2021-01-07",null));
    kontrolOdunc.add(new Odunc("kisiAdi Test 2","kitapAdi Test 1", "2021-01-07",null));
    kontrolOdunc.add(new Odunc("kisiAdi Test 1","kitapAdi Test 1", "2021-01-03",null));

    for (int i = 0; i < veritabaniOdunc.size(); i++) {
        assertEquals(veritabaniOdunc.get(i).kitapAdi,
        kontrolOdunc.get(i).kitapAdi);
        assertEquals(veritabaniOdunc.get(i).kisiAdi,
        kontrolOdunc.get(i).kisiAdi);
        assertEquals(veritabaniOdunc.get(i).iadeTarihi,
        kontrolOdunc.get(i).iadeTarihi);
        assertEquals(veritabaniOdunc.get(i).emanetTarihi,
        kontrolOdunc.get(i).emanetTarihi);
    }
}
```

Kod Örneği 21 Ödünç listeleme metodu için yazılan test kodu

- *Ödünç Ver Metodu:* Parametre olarak gelen kisiID'ye sahip kişiye yine parametre olarak gelen kitapID'ye sahip kitabı sistem tarihi baz alınarak ödünç verilir. Ayrıca baz alınan sistem tarihinin 15 gün sonrası için müddet tarihi belirlenir. Ödünç ver metoduna ait kod satırları Kod Örneği 22'de verilmiştir.

```
public void oduncver(int kisiID, int kitapID) throws SQLException{
    //testMi=1 ise islemID = "999" degilse islemID = "null" yap
    String islemID = (testMi==1)?"999":"null";
    Date simdikiZaman = new Date();
    DateFormat df = new SimpleDateFormat("yyyy-MM-dd");
    String tarih = df.format(simdikiZaman);
    long theFuture = System.currentTimeMillis() + (86400 * 64 * 1000);
    Date ahy = new Date(theFuture);
    String muddet = df.format(ahy);

    String query = " insert into emanetkitap (islemID, kisiID, kitapID, emanetTarihi,
    muddet, Test)" + " values ("+ islemID +",'"+kisiID+"', '"+kitapID+"',
    '"+tarih+"', '"+muddet+"',"+testMi+")";

    stmt.execute(query);
}
```

Kod Örneği 22 Ödünç ver metodu

- *Sil Metodu:* Veri tabanında var olan ödünç kayıtlarını silmek için yazılan metottur. Projede test amaçlı eklenen ödünç kayıtlarını silmek için kullanılmıştır. Ödünç modeline ait sil metodu Kod Örneği 23'te gösterilmiştir.

```
public void oduncSil(String ID) throws SQLException{
    String query = "delete from emanetKitap where islemID="+ID;

    stmt.execute(query);
}
```

Kod Örneği 23 Ödünç sil metodu

- *İade Al Metodu:* Kitap iadesi işlemini gerçekleştirir. ucretHesapla fonksiyonunu kullanır. Üyenin ödemesi gereken borç var ise bu borcu program kullanıcısına gösterir. Sistem tarihi alınarak veri tabanında iadeTarihi kısmını günceller. Ödünç modeli için yazılan iade al metodunun kodları Kod Örneği 24'te sunulmuştur.

```
public void iadeal(String islemID) throws Exception{
    Date simdikiZaman = new Date();
    int yil = simdikiZaman.getYear()+1900;
    int ay = simdikiZaman.getMonth()+1;
    int gun = simdikiZaman.getDate();
    String veritabaniFormatindaTarih = yil+"-"+ay+"-"+gun;
    //gec gelme suresi hesapla
    double ucret = ucretHesapla(islemID);

    if(ucret > 0)
        JOptionPane.showMessageDialog(null, "Odenmesi gereken ucret = "+ucret+"TL");
    String query = "update emanetkitap set iadeTarihi='"+veritabaniFormatindaTarih+"' "
    + "where islemID="+islemID;
    stmt.execute(query);
}
```

Kod Örneği 24 İade al metodu

Ödünç modelinde yazılan ödünç ver ve iade al metotlarının test kodu Kod Örneği 25'te yer alır.

```
@Test
public void testOduncverIadeAl() throws Exception {
    System.out.println("oduncver");
    int kisiID = 3;
    int kitapID = 1;
    o.oduncver(kisiID, kitapID);
    o.iadeal("999");
    o.oduncSil("999");
}
```

Kod Örneği 25 Ödünç ver ve iade al metodu için yazılan test kodu

- *Ücret Hesapla Metodu:* emanetGunHesapla fonksiyonunu kullanır. Kullanıcıların 15 güne kadar kitapları geri getirmesi beklenir. Ücret negatif değer çıktığında iadeal fonksiyonu tarafından yok sayılır. Geçen gün sayısının birim ücret olan 2 TL ile çarpılması sonucu ödenmesi gereken tutar hesaplanır. Ücret hesapla metodu için yazılan kod blokları, Kod Örneği 26'da verildiği gibidir.

```
public double ucretHesapla(String islemID) throws Exception {
    double gecGelenGunSayisi = emanetGunHesapla(islemID)-15;
    // 10 gunden fazlasi gec gelmis sayilsin
    return gecGelenGunSayisi * 2;
}
```

Kod Örneği 26 Ücret hesapla metodu

- *Emanet Gün Hesapla Metodu:* Emanet alınan tarih ile o kitabın teslim tarihi arasında geçen gün sayısı hesaplanır. Hesaplanan tarihler arasındaki fark milisaniye cinsinden olduğu için son aşamada 86.400.000 (24*60*60*1000)'e bölünür. Ödünç modeline ait emanet gün hesapla metodunun yazıldığı kod, Kod Örneği 27'de gösterilmiştir.

```
1  public double emanetGunHesapla(String islemID) throws Exception {
2      String emanetTarihi;
3      ResultSet rs=stmt.executeQuery("SELECT * FROM emanetkitap WHERE islemID = "+islemID);
4      if(rs.next())  {
5          emanetTarihi =rs.getString("emanetTarihi");
6
7          //Okunan tarihi integere cevirme
8          String tarih[] = emanetTarihi.split("-");
9          int yil = Integer.parseInt(tarih[0]) - 1900;
10         int ay = Integer.parseInt(tarih[1]) - 1;
11         int gun = Integer.parseInt(tarih[2]);
12
13         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
14         Date simdikiTarih = sdf.parse(sdf.format(new Date()));
15         Date alinmaZamani = new Date(yil,ay,gun);
16
17         long gecenSure = simdikiTarih.getTime() - alinmaZamani.getTime();
18         return Math.ceil(gecenSure / 86400000);
19     }
20     return -1;
21 }
```

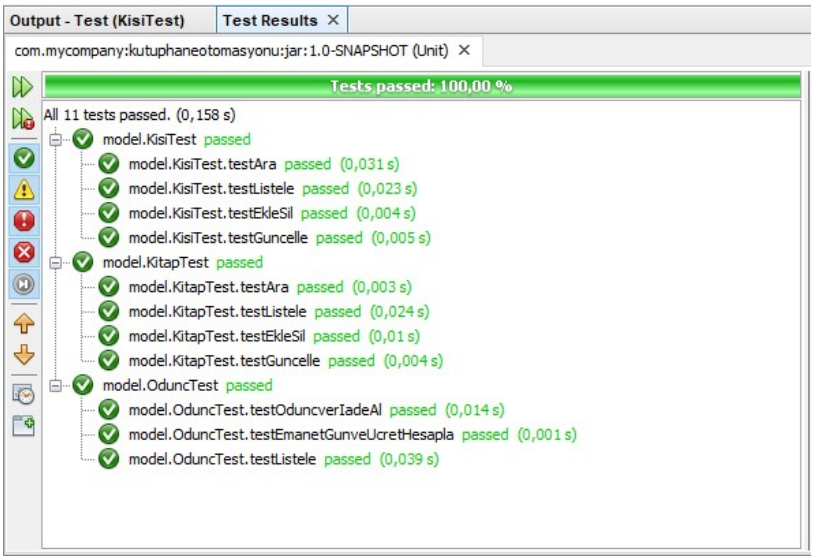
Kod Örneği 27 Emanet gün hesapla metodu

Emanet gün sayısının ve buna bağlı olarak ücret hesaplamasının yapıldığı metot için yazılan test kodu, Kod Örneği 28'de gösterilmiştir.

```java
@Test
public void testEmanetGunveUcretHesapla() throws Exception {
    System.out.println("emanetGunHesapla");

    //07.01.2021 'den bugune gecen surenin hesaplanmasi
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    Date simdikiTarih = sdf.parse(sdf.format(new Date()));
    Date alinmaZamani = new Date(121,0,7);
    long gecenSure = simdikiTarih.getTime() - alinmaZamani.getTime();
    double beklenenGecenSure = Math.ceil(gecenSure / 86400000);

    //Fonksiyon sonucunda elde edilen gecen gun sayisinin beklenen deger
    //ile karsilastirilmasi
    String islemID = "1";
    double result = o.emanetGunHesapla(islemID);
    assertEquals(beklenenGecenSure, result, 0.0);

    //Fonksiyon sonucunda elde edilen ucretin beklenen ucret degeri ile
    //karsilastirilmasi
    System.out.println("ucretHesapla");
    double beklenenUcret = (beklenenGecenSure - 10)*2;
    double result2 = o.ucretHesapla(islemID);
    assertEquals(beklenenUcret, result2, 0.0);
}
```

Kod Örneği 28 Emanet gün sayısı ve ücret hesapla metodu için yazılan test kodu

Yukarda bir kısmı verilen testler Apache Netbeans üzerinde çalıştırılmıştır. İşlemlerin başarı durumu ve çalışma süreleri Şekil 9.11.'de verilmiştir.



Şekil 9.11. Netbeans ortamında çalıştırılan test çıktısı

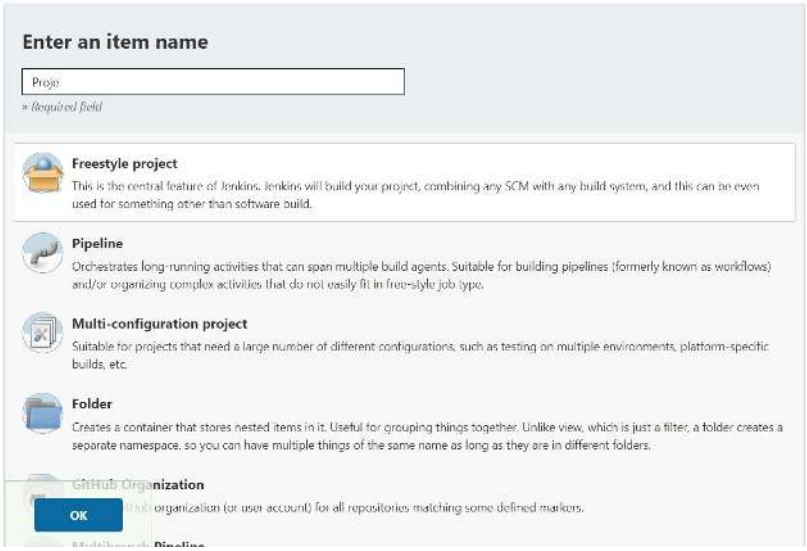
9.4.5. Jenkins Test Platformunun Yapılandırılması

Bu bölümde, gerçekleştirilen proje için Jenkins test platformunun yapılandırılması belirtilmektedir. Geliştirilen proje, yazılan JUnit testleri yardımı ile Jenkins platformunda sürekli entegrasyon mantığında test edilmiştir. Jenkins platformu Java tabanlı çalıştığı için Jenkins çalıştırılacak sistemde Java yazılımının kurulu olması ve asgari versiyonlarının yüklü olması gerekmektedir. Tez kapsamında Jenkins platformunu çalıştırmak için Java 8 versiyonu kullanılmıştır.

İlk olarak Jenkins uygulamasının ve Java 8 yazılımının web sayfalarından indirilmesi gerekmektedir. Ardından indirilen Jenkins klasörünün içindeki *jenkins.war* dosyası cmd ekranında yazılan $java - jardosyadizini/jenkins.war$ komutu ile çalıştırılır. Java komutunun cmd ekranında tanınabilir olması için sistem path altında tanımlı olması gerekir. Aksi taktirde Java komutunu kullanabilmek için *cd* komutu ile Java 8'in yüklü olduğu klasöre gidilmesi ve $java - jardosyadizini/jenkins.war$ komutunun verilmesi gerekir. Java'nın Windows sistem path değişkenine kaydedilip kaydedilmediğini anlamak için *java -version* komutu verilebilir. Bu komut sonrasında ekranda sürüm numarası veriliyorsa path kısmını başarı ile tanımlanmış demektir. Buradaki komutun başarı ile çalışması durumunda cmd ekranında bir dizi işlemler yapılarak *localhost* : 8080 adresinden Jenkins arayüzü ulaşabilir hale gelecektir.

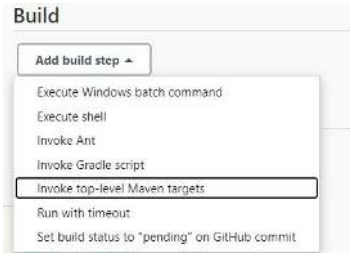
Jenkins arayüzü açıldığında başlangıç şifresi sormaktadır. Bu şifre hem cmd ekranında gözükmekte hem de *C:UserswindowsKullanıcıAdı/jenkins/ secrets* dizini altında initialAdminPassword dosyasına kaydedilmektedir. Bu şifrenin arayüze girilmesinin ardından kullanılacak eklentiler sorulacaktır. Bu aşamada varsayılan eklentiler seçeneği seçilerek hızlı kurulum gerçekleştirilebilir. Jenkins platformunda Java ortamının tanımlanması için $Managejenkins- > GlobalToolConfiguration- > AddJDK$ yolu izlenir. Buraya JDK'nın yüklü olduğu dizindeki bin klasörünün adresi verilmelidir. Maven ile projeleri çalıştırmak için yine aynı yerde yer alan Add Maven butonu kullanılarak Maven'in yüklü olduğu dizin adresi buraya verilir.

Proje oluşturmak için create new item sekmesi kullanılır. Burada açılan pencerede projeye bir isim verilerek freestyle project seçeneği seçilir. Proje oluşturma adımı Şekil 9.12.'de gösterilmiştir.



Şekil 9.12. Jenkins ile yeni proje oluşturulması

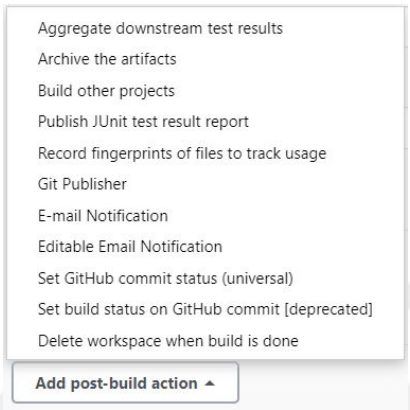
Proje oluştur butonuna tıklandıktan sonra projenin ayarlarının yapılması gerekmektedir. Burada Maven aracılığıyla projeleri çalıştırmak için build sekmesinin altında Invoke top-level Maven targets seçeneği eklenir. Maven versiyon kısmına daha önceki adımda sisteme tanıtılan Maven yüklemesi, Goals kısmına Maven hedefleri (compile, test vs.) ve POM kısmına ise projenin pom.xml dosyasının yer aldığı dizin adresi eklenerek kaydedilir. Jenkins ortamında Maven projelerinin çalıştırılması için gereken adımlar Şekil 9.13. ve Şekil 9.14.'te verilmiştir.



Şekil 9.13. Jenkins Maven yapılandırılmasının gerçekleştirilmesi - 1

Şekil 9.14. Jenkins Maven yapılandırılmasının gerçekleştirilmesi - 2

Proje oluşturulurken build aşamasında veya buildin bitmesinin ardından çalıştırılması için çeşitli komutlar ayarlanabilir. Build aşamasında çeşitli cmd komutlarının da kullanımına izin verilmektedir. Build sonrasında ise başka bir projeyi derleme, JUnit testlerini raporlama, Git ile versiyonlama, email ile bilgilendirme gibi çeşitli işlemler gerçekleştirilmesine izin verilmektedir. Build aşaması sonunda tetiklenebilecek işlemler Şekil 9.15.'te verilmiştir.



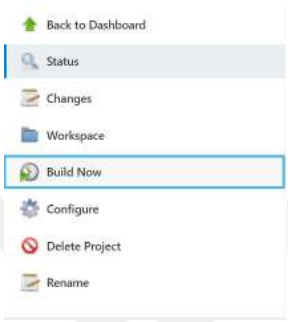
Şekil 9.15. Jenkins Build sonrası tetiklenebilecek işlemler listesi

Aynı konfigürasyon Jenkins Pipeline yardımı ile de gerçekleştirilebilir. Jenkins Pipeline bağlantısının kullanılması için gereken kod bloğu, Kod Örneği 29'da verilmiştir.

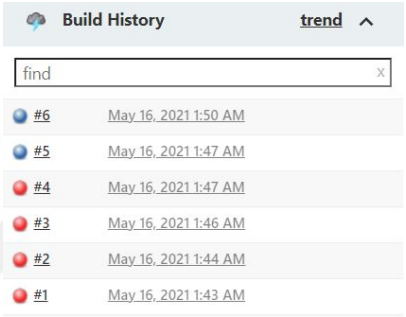
```
pipeline {
    agent any
    tools {
        jdk 'JDK'
    }
    stages {
        stage('Build') {
            steps {
                sh 'cd proje_dizini'
                sh 'mvn -f proje_dizini/pom.xml clean compile'
            }
        }
        stage('Test') {
            steps {
                sh 'cd proje_dizini'
                sh 'mvn -f proje_dizini/pom.xml test'
            }
        }
    }
}
```

Kod Örneği 29 Jenkins Pipeline bağlantısı için gereken kod bloğu

Yapılandırılması tamamlanan projeler artık Jenkins platformunda derlenebilir hale gelmiştir. Jenkins ile projenin derlenebilmesi için Jenkins arayüzünde, bir test projesi açılır. Açılan proje altında yer alan *build project* butonu tıklanır. Buton tıklandığında Jenkins çalışma adımları otomatik bir şekilde gerçekleştirilir. Bu sayede geliştiriciler kodun derlenip, raporlanıp, depoya yüklenmesi ve takım arkadaşlarına bildirilmesi gibi işlemleri tek bir yerden kolayca gerçekleştirebilir. Testlerin başarısız olması durumunda ise geliştiriciler bu hataları düzeltmek için erkenden uyarılmış olur. Jenkins platformunda çalışma adımlarının tetiklenmesi Şekil 9.16.'da gösterilmiştir. Derleme işlemi sonucunda gerçekleşen işlemlerin başarılı ya da başarısız çıktısı Şekil 9.17.'de verilmiştir.



Şekil 9.16. Yapılandırılan Jenkins projesinin Build edilmesi



Şekil 9.17. Build işlemleri sonucu üretilen başarılı ve başarısız denemeler

Yazılan testlerin başarılı olması durumundan elde edilen Jenkins konsol çıktısı Şekil 9.18.'deki gibidir.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running model.KisiTest
ara
listele
ekle
guncelle
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.132 sec
Running model.KitapTest
ara
listele
ekle
guncelle
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 sec
Running model.OduncTest
oduncver
listele
emanetGunHesapla
ucretHesapla
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.055 sec

Results :

Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
```

Şekil 9.18. Yazılan testlerin başarılı olması sonucu elde edilen Jenkins konsol çıktısı

Pipeline ile konfigüre edilmiş Jenkins derleme işlemine ait ekran çıktısı Şekil 9.19.'da verilmiştir. İlk deneme veri tabanı kapalı iken gerçekleştirildiği için işlem sonucu hata vermiştir. İkinci ve üçüncü denemelerde test işlemi başarısız olduğundan kodun mantıksal hatalara sahip olduğu söylenebilir. Son işlemde ise build ve test işlemi başarı ile gerçekleştirilmiştir.

Şekil 9.19. Jenkins Pipeline ile konfigüre edilmiş sistemin derleme adımları

9.5. Geliştirilen Uygulamanın Kod Ölçütlerinin Çıkarılması

Geliştirilen uygulamanın Eclipse eklentisi olan CodeMR ile metrikleri değerlendirilmiştir. Projenin genel metrikleri Tablo 9.5.'te verilmiştir:

Tablo 9.5. Genel proje metrikleri

Toplam Satır Sayısı	1133
Sınıf Sayısı	11
Paket Sayısı	2
External Paket Sayısı	29
Problemli Sınıf Sayısı	0
Yüksek Problemli Sınıf Sayısı	0

CodeMR kütüphanesinde uygulamaya ait metrikler farklı şekillerde kategorize edilmiştir. Bu gruplandırma *sınıf*, *paket* ve *metot* metrikleridir. Uygulamaya ait *sınıf* metriklerinde *CBO*, *RFC*, *DIT*, *NOC*, *WMC*, *LCOM* gibi değerler yer almakta ve Şekil 9.20.'de gösterilmektedir.

Element	Quality Attributes	LOC	Coupling	Complexity	Size	Lack of Cohesion	CBO	RFC	DIT	NOC	WMC	LOC	LCOM
kutuphaneotomasyonu													
model		187	low	low	low	low					34	187	
DataBase		11	low	low	low	low	0	6	1	3	2	11	1.0
Kisi		48	low	low-medium	low	low	0	13	2	0	10	48	0.375
Kitap		59	low	low-medium	low-medium	low	0	13	2	0	10	59	0.361
Oduncc		69	low	low	low-medium	low-medium	0	26	2	0	12	69	0.84
view		958	low-medium	low	low-medium	low					148	958	
AnaPencere		66	low	medium-high	low-medium	low	3	27	6	0	12	66	0.0
KisiEkle		141	low	medium-high	low-medium	low	2	61	6	0	19	141	0.778
KisiPanel		173	low	medium-high	low-medium	medium-high	2	66	5	0	17	173	0.796
KitapEkle		166	low	medium-high	low-medium	low	2	67	6	0	25	166	0.542
KitapPanel		175	low	medium-high	low-medium	medium-high	2	63	5	0	17	175	0.796
OduncPanel		103	low	medium-high	low-medium	low-medium	2	57	5	0	10	103	0.833
OduncVer		134	low	medium-high	low-medium	low	4	67	6	0	14	134	0.688

Şekil 9.20. Projeye ait sınıf metrikleri

Uygulamaya ait CodeMR metrik sınıflandırılması içerisinde yer alan *paket* metrik değerleri Şekil 9.21.'de gösterilmiştir.

Element	Quality Attributes	LOC	Coupling	Complexity	Size	Lack of Cohesion	LOC	#(Cl)	#C	AC	EC	Abs	Ins	ND	WMC
kutuphaneotomasyonu															
model		187	low	low	low	low	187	4	4	6	0	0.0	0.0	1.0	34
DataBase		11	low	low	low	low	11								2
Kisi		48	low	low-medium	low	low	48								10
Kitap		59	low	low-medium	low-medium	low	59								10
Oduncc		69	low	low-medium	low-medium	low	69								12
view		958	low-medium	low	low-medium	low	958	7	7	0	6	0.0	1.0	0.0	148
AnaPencere		66	low	medium-high	low-medium	low	66								12
KisiEkle		141	low	medium-high	low-medium	low	141								19
KisiPanel		173	low	medium-high	low-medium	medium-high	173								17
KitapEkle		166	low	medium-high	low-medium	low	166								25
KitapPanel		175	low	medium-high	low-medium	medium-high	175								17
OduncPanel		103	low	medium-high	low-medium	low-medium	103								10
OduncVer		134	low	medium-high	low-medium	low	134								14

Şekil 9.21. Projeye ait paket metrikleri

CodeMR kütüphanesine göre, geliştirilen uygulamaya ait *metot* metrikleri Şekil 9.22.'deki gibi sınıflandırılmıştır.

Element	Quality Attributes	LOC	Coupling	Complexity	Size	Lack of Cohesion	LOC	MCC	NBD	#Pa	#MC	#AF
kutuphaneotomasyonu												
model		187	low	low	low	low	187					
DataBase		11	low	low	low	low	11					
Kisi		48	low	low-medium	low	low	48					
Kitap		59	low	low-medium	low-medium	low	59					
Oduncc		69	low	low-medium	low-medium	low	69					
view		958	low-medium	low	low-medium	low	958					
AnaPencere		66	low	medium-high	low-medium	low	66					
KisiEkle		141	low	medium-high	low-medium	low	141					
KisiPanel		173	low	medium-high	low-medium	medium-high	173					
KitapEkle		166	low	medium-high	low-medium	low	166					
KitapPanel		175	low	medium-high	low-medium	medium-high	175					
OduncPanel		103	low	medium-high	low-medium	low-medium	103					
OduncVer		134	low	medium-high	low-medium	low	134					

Şekil 9.22. Projeye ait metot metrikleri

Kişi sınıfına ait metotların metrikleri incelendiğinde, ara metodunun en fazla *LOC* değerine yani en fazla satır sayısına sahip metot olduğu görülmektedir. Döngüsel karmaşıklık (*MCC*), en fazla ara, ekle ve listele metotlarında görülmektedir. En fazla toplam parametreye sahip *#Pa* metriğini barındıran metot kisi metodu, en fazla çağrılan metot sayısını belirten metrik olan *#MC* metriği 7 olarak belirtilen ara metodu ve listele

metodunda yer almaktadır. $\#AF$ erişim alanı sayısı en fazla olan metotlar 6 değere sahip olan ara, ekle ve listele metotlarıdır. Kişi sınıfına ait metriklerden bazıları Şekil 9.23.'te olduğu gibidir:

Element	LOC	WMC	MCC	NBD	#Pa	#MC	#AF
model	187	34					
DataBase	11	2					
Kisi	48	10					
ara(String): ArrayList	13		2	2	1	7	6
ekle(Kisi): void	5		2	1	1	1	6
guncelle(int, Kisi): void	4		1	1	2	1	4
Kisi( ): void	2		1	1	0	0	1
Kisi(int, String, String, Stri	5		1	1	4	0	4
listele( ): ArrayList	11		2	2	0	7	6
sil(int): void	3		1	1	1	1	1

Şekil 9.23. Kişi sınıfına ait metot metrikleri

Kitap sınıfına ait metrikler değerlerinden bazıları Şekil 9.24.'te mevcuttur. Bu metrik sonuçlandırılmasına göre LOC yani en fazla satır satısına sahip metot kitap sınıfında bulunan ara metodudur. MCC ile döngüsel karmaşıklığı en yüksek metotlar ara, ekle ve listele metodu olarak belirtilebilir. NBD ile iç içe metot derinliklerinin en fazla olduğu metotlar ise ara ve listele metotlarıdır. Toplam parametre sayısı olarak belirtilen $\#Pa$ en fazla kitap metodunda mevcuttur. Erişim alanı olarak belirtilen $\#AF$ metodu en fazla ara, listele ve ekle metotlarında yer almaktadır.

Element	LOC	WMC	MCC	NBD	#Pa	#MC	#AF
kutuphaneotomasyonu							
model	187	34					
DataBase	11	2					
Kisi	48	10					
Kitap	59	10					
ara(String): ArrayList	15		2	2	1	9	8
ekle(Kitap): void	7		2	1	1	1	8
guncelle(int, Kitap): void	5		1	1	2	1	6
Kitap( ): void	2		1	1	0	0	1
Kitap(int, String, String, ir	7		1	1	6	0	6
listele( ): ArrayList	13		2	2	0	9	8
sil(int): void	3		1	1	1	1	1

Şekil 9.24. Kitap sınıfına ait metot metrikleri

Ödünç sınıfına ait metriklerden bazıları Şekil 9.25.'te görüldüğü gibidir. CodeMR eklentisi aracılığıyla farklı metrikler filtrelenerek tabloda gösterilebilir. Sonuçların değerlendirilmesi adına seçilen metriklerde LOC ile belirtilen kod satır sayısı en fazla olan metotlar emanet gün hesaplaması ve ödünç işleminin listelendiği metotlar olarak belirtilmiştir. Yazılan kod bloklarında bulunan toplam parametre sayısı $\#Pa$ en yüksek olan metot 4 olarak belirlenen Odunc metodudur. En fazla çağrılan metot sayısı değeri olan $\#MC$'ye sahip kod bloğu, 12 olarak belirlenen emanet günün hesaplandığı kod bloğudur. Erişim alanı sayısı $\#AF$ en yüksek olan metot 7 ile listele metodudur.

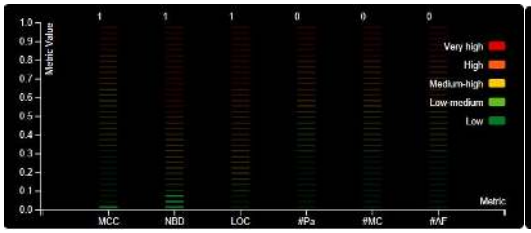
Element	LOC	WMC	MCC	NBD	#Pa	#MC	#AF
kutuphaneotomasyonu							
model	187	34					
DataBase	11	2					
Kisi	48	10					
Kitap	59	10					
Odunc	69	12					
emanetGunHesapla(Strin	15		2	2	1	12	1
iadeal(String): void	12		2	1	1	6	1
listele( ): ArrayList	15		2	2	0	8	7
Odunc( ): void	1		1	1	0	0	0
Odunc(String, String, Strir	5		1	1	4	0	4
oduncSil(String): void	3		1	1	1	1	1
oduncver(int, int): void	11		2	1	2	4	3
ucretHesapla(String): do	3		1	1	1	1	0

Şekil 9.25. Ödünç sınıfına ait metot metrikleri

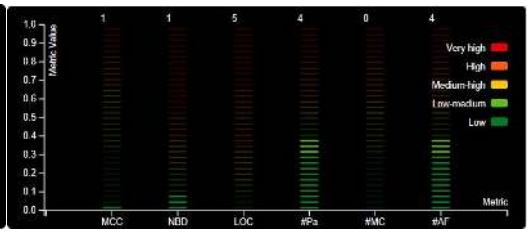
Metrik değerleri yukarda verilen metotların problemli olup olmadıkları metrik değerlerinin varsayılan sınırlar içerisinde olup olmaması ile kontrol edilir. Ödünç sınıfına ait metrik değerleri bu açıdan değerlendirilmiştir. Değerlendirme işlemi Ödünç altında yer alan fonksiyonlar için teze eklenmiştir.

Tablo 9.6.'da ödünç sınıfında bulunan metotlar için seçilen metrik değerlerinin risk derecelerine göre kritiklik durumu belirtilmiştir. Ödünç sınıfına ait boş yapılandırıcının, dolu yapılandırıcının, listele metodunda, sil metodunda, ödünç ver metodunda, iade al metodunda, emanet gün hesaplanması için oluşturulan metotta ve ücretin hesaplandığı metotta bulunan metriklerin değerlendirilmesi ve kritiklik durumları risk derecelerine göre görselleştirilmiştir. Grafikler göz önüne alındığında *#MC* olarak belirtilen çağrılan metot sayısı metriğinin, ödünç sınıfında bulunan emanet gün hesaplama metodunda yüksek değerde olduğu söylenebilir. *#MC* ifadesinin yüksek olduğu sıralamada ikinci metot ödünç listele metodudur. Ödünç sınıfında seçilen metrik değerlerinin tamamı göz önüne alındığında en az risk barındıran metotların; ödünç boş yapılandırıcısı, sil metodu ve ücret hesaplama metodu olduğu ifade edilir.

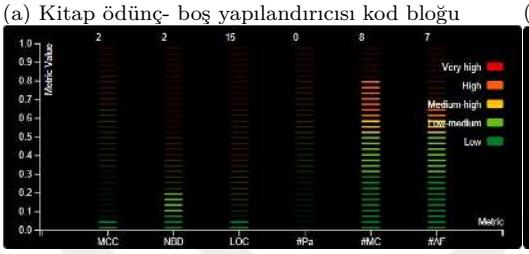
Tablo 9.6. Kütüphane otomasyonu modüllerindeki kod ölçütlerine göre risk seviyelerinin grafiksel gösterimi



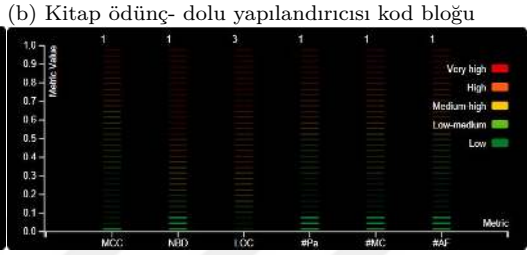
(a) Kitap ödünç- boş yapılandırıcısı kod bloğu



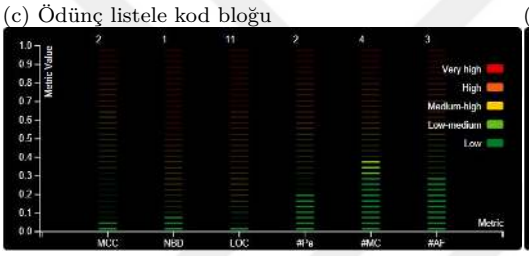
(b) Kitap ödünç- dolu yapılandırıcısı kod bloğu



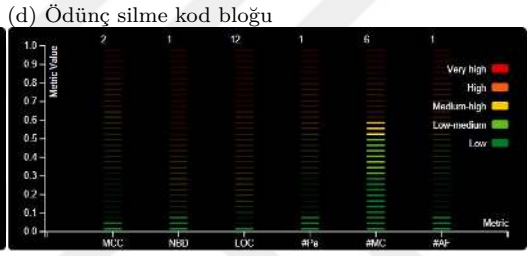
(c) Ödünç listele kod bloğu



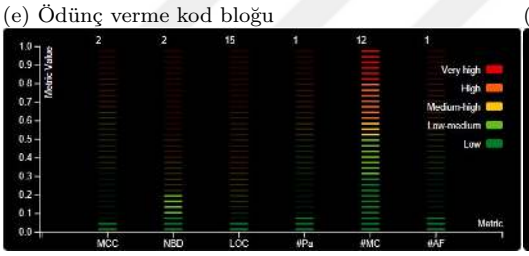
(d) Ödünç silme kod bloğu



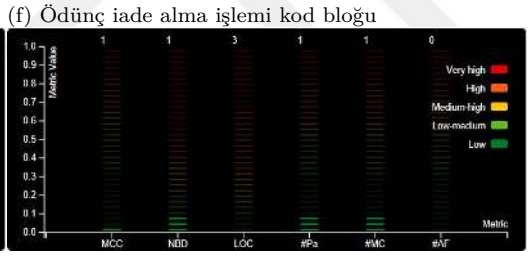
(e) Ödünç verme kod bloğu



(f) Ödünç iade alma işlemi kod bloğu



(g) Emanet gün hesaplama kod bloğu



(h) Geciken emanet için ücret hesaplama kod bloğu

10. Sonuç

Yazılım testleri, sistem kalitesini arttırmak ve minimum hataya düşürmek amacıyla yapılır. Yazılım testlerinin profesyonel bir şekilde icra edilmesi sonucunda yaşam döngüsü boyunca sürdürülebilir bir yapı sağlanır ve hataların giderilmesiyle toplam maliyet büyük ölçüde azaltılır.

Bu tez çalışması ile yazılım test süreçlerini ihtiva eden ilişkili tüm konular incelenmiş, bu konular doğrultusunda uygulamalar gerçekleştilerek yazılım testlerinin önemi ortaya konulmuştur. Dolayısıyla aşağıdaki hususlar araştırılmış ve uygulamalarla sonuçlar irdelenmiştir.

- Öncelikle, yazılım hataları incelenmiş, kullanıcı, sistem, önem derecesi ve gerçekleşme ihtimaline göre dört kategoride yazılım hataları sınıflandırılmıştır. Yazılım testlerinin gerçekleştirilmesi ve yazılımların iyileştirilmesinin nedeni olan yazılım hatalarına ışık tutulmuştur.

- Yazılım test yaşam döngüsü ve test süreçleri detaylıca incelenmiş, test aşamaları sunulmuştur. Ayrıca yazılım test süreçlerinde bir zorluk olarak karşılaşılan sürekli entegrasyon, sürekli teslimat ve sürekli dağıtım işlemleri analiz edilmiştir.

- Kara kutu, beyaz kutu ve gri kutu test teknikleri incelenmiş; kara kutu test tekniği bir senaryo yardımı ile örneklendirilmiştir.

- Dört aşamadan oluşan yazılım test seviyeleri irdelenmiştir.

- Literatürde birçok farklı kaynaklarda yazılım test türlerinin, yanlış ve düzensiz olarak kategorize edildiği görülmüştür. İç içe girmiş yazılım test türlerinin tek bir grafik altında kategorize edilemeyeceği yapılan analizler sonucunda görülmüştür. Bu nedenle günümüzde yaygın olarak kullanılan yazılım test türleri ve açıklamaları bir tablo halinde sunulmuştur.

- Yazılım kalite güvencesi için birçok önemli nitelikler vardır. ISO 9126 kalite standart ölçütlerine göre belirlenen unsurlar sunulmuştur.

- Yazılım kalitesinin ortya konulmasında ölçütler önemli rol oynar. Yazılım ölçütlerinin kapsamına göre Süreç, Proje ve Ürün ölçütleri ayrı ayrı ele alınmış, kod karmaşıklık analizi için literatürdeki metrikler tablolar halinde sunulmuştur.

- Java ortamında geliştirilen örnek projenin Jenkins platformu yardımıyla *JUnit* test uygulamaları sürekli entegrasyon mantığı ile otomatik test edilmiştir. İşlevsel olarak test edilen örnek projenin, *Eclips CodeMR* eklentisi yardımıyla yazılım ölçütleri çıkarılmıştır. Bu sayede geliştirilen yazılımın işlevsel olarak doğru çalışması ve kalite niteliklerine uygunluğu güvence altına alınmıştır.

Tez çalışması kapsamında geliştirilen basit bir kütüphane uygulamasına kara kutu ve beyaz kutu test tekniği uygulanmıştır. Projenin geliştirilmesi için *Java* programlama dili kullanılmış, uygulama arayüzleri için *Swing* kütüphanesinden yararlanılmıştır. Fonksiyonel testlerin yazılmasında *JUnit* kütüphanesinden, testlerin her geliştirme adımından sonra otomatik bir şekilde tetiklenmesi için Jenkins platformundan yararlanılmıştır. Sonuç olarak yapılan yazılım geliştirme ve güncelleştirme adımlarının daha kontrollü bir şekilde gerçekleştirilmesi ve ortaya çıkan programın daha az hataya sahip olması sağlanmıştır. Sonuç olarak Java platformunda geliştirilen kütüphane uygulaması için süreç, proje ve ürün metrikleri açısından değerlendirilmiştir. Özellikle, geliştirilen kod bloklarının kalitesinin değerlendirilmesi ve ölçülebilmesi için sınıf ve kod metrik değerleri ortaya konulmuştur.

Yazılım geliştirme sürecinde hataların tamamen bitmeyeceği ve yazılım geliştiricilerin yazılım hataları ile yaşamayı bilmeleri gerektiği açıkça görülmüştür. Yazılım hatalarını en aza indirgemek amacıyla, geliştirilme sürecindeki her değişiklik sonrasında test işlemleri uygulanmaktadır. Bu sayede yazılım hataları daha erken safhada tespit edilerek, düzeltilmesi mümkün olmaktadır. Daha az hatalı yazılım ile hata giderme maliyetleri ve hataların sebep olacağı muhtemel zararların minumuma ineceğinin farkında olması önemlidir. Ancak diğer taraftan projelerin geliştirilme aşamasında testlerin tekrar tekrar çağrılması geliştiriciler açısından zahmetli olmaktadır. Çözüm olarak test işleminin otomatik olarak tetiklenmesi sağlanmalıdır.

Kaynaklar

[1] Turkish Testing Board (2013). Turkey software quality report 2013-2014. Technical report, International Software Testing Qualifications Board.

[2] B. Takgil, R. (2016). Android mobil uygulamalar için yazılım testi. Technical report, Ulusal Mühendislik Araştırmaları Sempozyumu.

[3] Garousi, V.; Demirors, O.; Coşkunçay, A.; Can, B. A. (2013). Türkiye'deki yazılım test uygulamaları anketi. *CEUR Workshop Proceedings*.

[4] Zöngür, U.; Erdoğan, T. (2014). Cobalt: Test uygulamaları için protokol kütüphanesi. Technical report, CEUR Workshop Proceedings.

[5] Onur, Ö. (2015). Emniyet kritik yazılım test edilebilirliğinin iyileştirilmesi. *Biomass Chem Eng*, C. 49, Sayı 23–6, 22–23.

[6] Giray, G. M. (2012). *Model denetleme temelli yazılım testi ve analizi yöntemlerinde uygulanabilirliğin arttırılması*. Doktora Tezi, Haliç Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 318555.

[7] Güneş, K. (2014). Yazılım mühendisliği yöntemleriyle yazılım test süreci. Yüksek Lisans Tezi, Haliç Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 389639.

[8] Calp, H. (2011). Nesneye yönelimli yazılım testi ve metrik kümesi değerlendiren uzman modülün gerçekleştirilmesi. Yüksek Lisans Tezi, Gazi Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 285476.

[9] Karagöz, A. B. (2017). Yazılım test süreci ve uygulama örneği ile süreç iyileştirme çalışmaları ve sonuçları. *Akademik Bilişim*, 1–7.

[10] Hancı, A. K. (2017). *Yazılım testi tasarım tekniklerinin matematiksel model yaklaşımı ile analizi*. Doktora Tezi, İstanbul Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 483717.

[11] Öztürk, M. M. (2013). *Uzakta eğitimde ölçme değerlendirme sistem tasarımı ve yazılım test teknikleri ile performans analizi*. Doktora Tezi, Sakarya Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 318362.

[12] Yağcı, N. (2013). *Yazılım kalite metrikleri ile test eforu arasındaki ilişkinin belirlenip tarihsel verinin oluşturulması*. Doktora Tezi, Sakarya Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 343290.

[13] Tuna, O. (2005). *Software testing according to development process and architectural description*. Doktora Tezi, Dokuz Eylül Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 202582.

[14] Kahveci, A.; Daş, R. (2019). Yaşanmış önemli yazılım hatalarının incelenmesi ve kritik öneriler. *2019 IEEE International Artificial Intelligence and Data Processing Symposium (IDAP)*, 4, 1–6, Inonu University, Malatya, Turkey. IEEE.

[15] [URL] Patriot, https://missilethreat.csis.org/system/patriot, Erişim Tarihi: 17-2-2019.

[16] [URL] 1st Intel Pentium processor is shipped, March 22, 1993, https://www.edn.com/electronics-blogs/edn-moments/4369686/1st-Intel-Pentium-processor-shipped–March-22-1993, Erişim Tarihi: 1-2-2019.

[17] [URL] Investigators say erroneous navigation input led Ariane 5 rocket off course, https://spaceflightnow.com/2018/02/23/investigators-say-erroneous-navigation-input-led-ariane-5-rocket-off-course, Erişim Tarihi: 10-2-2019.

[18] [URL] The lost decade: Honda exercises hopeful humility and acknowledges mistakes made, https://www.thetruthaboutcars.com/2017/09/honda-exercises-hopefully-humility-acknowledges-mistakes-made, Erişim Tarihi: 28-1-2019.

[19] [URL] Debugging, https://www.airs.com/ian/essays/debug/debug.html, Erişim Tarihi: 12-2-2019.

[20] [URL] The 10 worst programming mistakes in history, https://www.makeuseof.com/tag/worst-programming-mistakes-in-history, Erişim Tarihi: 2-2-2019.

[21] [URL] Yazılımcıların yaptığı 10 ölümcül hata, https://onedio.com/haber/yazilimcilarin-yaptigi-10-olumcul-hata-334888, Erişim Tarihi: 19–2019.

[22] [URL] Almost a Tragedy: The collapse of the hartford civic center, https://connecticuthistory.org/almost-a-tragedy-the-collapse-of-the-hartford-civic-center, Erişim Tarihi: 19-3-2019.

[23] [URL] Coliseum roof collapses at Hartford civic center, https://www.nytimes.com/1978/01/19/archives/new-jersey-pages-coliseum-roof-collapses-at-hartford-civic-center.html, Erişim Tarihi: 10-3-2019.

[24] [URL] Hartford stadium collapse: why software is just a tool and should be used wisely, http://www.engineersjournal.ie/2018/01/23/hartford-stadium-collapse-software, Erişim Tarihi: 12-4-2019.

[25] [URL] CIA trojan causes Siberian gas pipeline explosion, https://www.risidata.com/index.php?/Database/Detail/cia-trojan-ca uses-siberian-gas-pipeline-explosion, Erişim Tarihi: 18-1-2019.

[26] [URL] Soviets burned by CIA hackers?, https://www.wired.com/2004/03/soviets-burned-by-cia-hackers, Erişim Tarihi: 16-2-2019.

[27] [URL] 1988: U.C. Berkeley paper catalyses interest in RAID, https://www.computerhistory.org/storageengine/u-c-berkeley-paper-catalyses-interest-in-raid, Erişim Tarihi: 13-3-2019.

[28] [URL] Geçmişten günümüze kaos oluşturan 15 yazılım hatası, http://www.yazilimheryerde.com/2014/05/gecmisten-gunumuze-kaos-olusturan-15.html, Erişim Tarihi: 7-1-2019.

[29] [URL] Top 15 worst computer software blunders, https://www.intertech.com/Blog/15-worst-computer-software-blunders, Erişim Tarihi: 16-3-2019.

[30] [URL] 5 Most embarrassing software bugs in history, https://www.scientificamerican.com/article/pogue-5-most-embarrassing-software-bugs-in-history, Erişim Tarihi: 23-2-2019.

[31] [URL] The 5 most infamous software bugs in history, https://www.bbvaopenmind.com/en/technology/innovation/the-5-most-infamous-software-bugs-in-history, Erişim Tarihi: 3-2-2019.

[32] [URL] Nov. 10, 1999: Metric math mistake muffed Mars meteorology mission, https://www.wired.com/2010/11/1110mars-climate-observer-report, Erişim Tarihi: 12-2-2019.

[33] [URL] Hospital revives its "Dead" patients, http://www.baselinemag.com/c/a/Projects-Networks-and-Storage/Hospital-Revives-Its-QTEDeadQTE-Patients, Erişim Tarihi: 22-1-2019.

[34] [URL] Wrong turn: Apple's buggy iOS 6 maps lead to widespread complaints, https://www.theverge.com/2012/9/20/3363914/wrong-turn-apple-ios-6-maps-phone-5-buggy-complaints, Erişim Tarihi: 10-2-2019.

[35] [URL] Apple's maps mistake, https://www.theguardian.com/technology/blog/2012/oct/01/apple-30-billion-maps-mistake, Erişim Tarihi: 1-5-2019.

[36] [URL] Knight Capital says trading glitch cost it $440 million, https://dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million, Erişim Tarihi: 12-3-2019.

[37] [URL] Goldman Sachs' massive trading error bears a scary resemblance to the one that brought down Knight Capital, https://www.businessinsider.com/goldman-knight-capital-trading-errors-2013-8, Erişim Tarihi: 15-2-2019.

[38] [URL] Knight shows how to lose $440 million in 30 minutes, https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes, Erişim Tarihi: 6-1-2019.

[39] [URL] Top Software Failures In Recent History, https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html, Erişim Tarihi: 15-1-2019.

[40] [URL] Toyota car fault prompts massive recall, https://www.bbc.com/news/business-45756676, Erişim Tarihi: 13-2-2019.

[41] [URL] Toyota failed to fix defect that can cause Prius to overheat and lose power, dealer claims in lawsuit, https://www.latimes.com/local/california/la-fi-toyota-prius-defect-20180207-story.html, Erişim Tarihi: 18-3-2019.

[42] [URL] The Biggest Software Failures in Recent Years, https://dzone.com/articles/the-biggest-software-failures-in-recent-years, Erişim Tarihi: 10-1-2019.

[43] [URL] $́300m In Cryptocurrency' Accidentally Lost Forever Due to Bug, https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether, Erişim Tarihi: 14-1-2019.

[44] [URL] 10 Biggest Software Bugs And Tech Fails of 2021, https://www.testdevlab.com/blog/2021/12/27/10-biggest-software-bugs-and-tech-fails-of-2021, Erişim Tarihi: 1-2-2019.

[45] Uzun, B. (2019). *Yazılım Testi Süreç Yönetimi ve Raporlama Sisteminin Geliştirilmesi*. Doktora Tezi, İzmir Dokuz Eylül Üniversitesi. Tez No: 569983.

[46] [URL] Test process in software testing, https://www.toolsqa.com/software-testing/test-process-in-software-testing, Erişim Tarihi: 20-3-2019.

[47] [URL] The A to Z guide to the software testing process, https//reqtest.com/testing-blog/the-a-to-z-guide-to-the-software-testing-process, Erişim Tarihi: 1-1-2019.

[48] Bilgin, M. B.; Şit, G. (2019). Test Olgunluk Modeli Entegrasyonu (TMMi) ile Yazılım Test Süreçlerinin İyileştirilmesi. *International Symposium on Applied Science*, C. 4, Sayı 1, 44–55.

[49] [URL] Types of Software Testing, https://www.guru99.com/types-of-software-testing.html, Erişim Tarihi: 1-4-2019.

[50] [URL] Types of software testing, https://www.geeksforgeeks.org/types-software-testing, Erişim Tarihi: 17-5-2019.

[51] Hanefi Calp, M.; Arici, N. (2011). Nesne yönelimli tasarım metrikleri ve kalite özellikleriyle ilişkisi. *Politeknik Dergisi Journal of Polytechnic Cilt Digital Object Identifier*, C. 14141, Sayı 10, 9–14.

[52] Tauhid Ullah Shah, S.; Khan, F. (2016). An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology*, C. 2, Sayı 2, 682–689.

[53] Sawant, A. A.; Bari, P. H.; Chawan, P. (2012). Software Testing Techniques and Strategies. *Journal of Engineering Research & Applications*, C. 2, Sayı 3, 980–986.

[54] Şahin, Ö. (2016). Yazılım Test Verisi Üretiminde Yapay Zeka Tekniklerinin Performans Analizi. Yüksek Lisans Tezi, Erciyes Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 373774.

[55] Turkish Testing Board (2020). Turkey software quality report 2019-2020. Technical report, International Software Testing Qualifications Board.

[56] [URL] Entegrasyon testi nedir, https://www.muhendisbeyinler.net/entegrasyon-testi-nedir/, Erişim Tarihi: 12-5-2019.

[57] Kan, M. (2019). Finans sektörü yazılım test sürecinin kalite yönetimi açısından incelenmesi. *Akademik Bilişim*, Ordu Üniversitesi, Ordu, Türkiye.

[58] [URL] Software testing, professionalqa.com/software-testing, Erişim Tarihi: 6-2-2020.

[59] [URL] Types of software testing – Complete list, https://www.testingexcellence.com/types-of-software-testing-complete-list, Erişim Tarihi: 14-3-2019.

[60] [URL] Types of Software Testing, https://www.softwaretestingmaterial.com/types-of-software-testing, Erişim Tarihi: 2-4-2019.

[61] [URL] Gorilla testing vs. Monkey testing, https://yourstory.com/mystory/difference-between-gorilla-testing-and-monkey-test, Erişim Tarihi: 1-2-2021.

[62] [URL] What is parallel testing?, https://www.bmc.com/blogs/what-is-parallel-testing-parallel-testing-explained, Erişim Tarihi: 1-5-2019.

[63] [URL] Risk based testing in software testing, https://www.softwaretestingclass.com/risk-based-testing-in-software-testing, Erişim Tarihi: 28-1-2019.

[64] Kasapoğlu, M. (2019). Yazılım Test Planlama ve Raporlama Aracının Tasarlanması ve Kodlanması. *Journal of Chemical Information and Modeling*, C. 53, Sayı 9, 1689–1699.

[65] [URL] What is interface testing?, https://www.guru99.com/interface-testing.html, Erişim Tarihi: 19-3-2019.

[66] Turkish Testing Board (2018). Turkey software quality report 2017-2018. Technical report, International Software Testing Qualifications Board.

[67] Dönmez, A.; Üçüncü, E. (2016). Test Metrikleri Üzerinden Test Efor Tahminlemesi. *10. Ulusal Yazılım Mühendisliği Sempozyumu*, Onsekiz Mart Üniversitesi, Canakkale, Türkiye.

[68] Serdaroğlu, D. (2015). *Yazılım test süreci ve TMMİ modelinde Prisma yaklaşımı uygulaması*. Doktora Tezi, Başkent Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 382314.

[69] Pressman, R. S.; Maxim, B. R. (2015). Software engineering: A practitioner's approach. Eighth, E., editor, *Software Engineering*, C. 1. Raghu Srinivasan, 8 edition.

[70] A. Fadhıl, A.; GH. Alsarraj, R.; M. Altaie, A. (2020). Software Cost Estimation Based on Dolphin Algorithm. *Biomass Chem Eng*, C. 8, Sayı 10-1109, 75279–75287.

[71] Catal, C. (2012). Yazılım mühendisliği yöntemleri-İleri konular. Çölkesen, D. R., editor, *Yazılım Mühendisliği Yöntemleri-İleri Konular*. Dr. Cengiz Uğurkaya, 1 edition.

[72] Grady, R. B. (1992). Practical software metrics for project management and process improvement. *Practical Software Metrics For Project Management and Process Improvement*. Prentice-Hall, 1 edition.

[73] Humphrey, W. S. (2005). Psp: A self-improvement process for software engineers. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 1 edition.

[74] Gurgoze, G.; Das, R.; Turkoglu, I. (2017). Comparison of software development process models. *The 8th International Advanced Technologies Symposium (IATS-2017)*, 1923–1932, Fırat Üniversitesi, Elazığ, Türkiye.

[75] Kıral, A.; Erçelebi, T. (2018). Yazılım kalite metriklerinin kıyaslanması: Örnek bir olay incelemesi. *12. Ulusal Yazılım Mühendisliği Sempozyumu*, Sabancı Üniversitesi, Istanbul, Turkey.

[76] Chidamber, S. R.; Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, C. 20, Sayı 6, 476–493.

[77] Erdemir, U.; Tekin, U.; Buzluca, F. (2008). Nesneye dayalı yazılım metrikleri ve yazılım kalitesi. *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu*, 9–14, Kültür Üniversitesi, Istanbul, Turkey.

[78] Ozdemir, N.; Dinçer, K.; Gezici, B. (2016). Measurement of the Quality Characteristics of Mobile Applications (Mobil Uygulamaların Kalite Özelliklerinin Ölçümü). *10. Türkiye Ulusal Yazılım Mühendisliği Sempozyumu*, 337–348, Onsekiz MartUniversity, Canakkale, Turkey.

[79] Gezici, B.; Tarhan, A.; Chouseinoglou, O. (2019). Mobil uygulamaların evriminde karmaşıklık, boyut ve iç kalite gelişimi: keşifsel bir çalışma. *Gazi Üniversitesi Mühendislik Mimarlık Fakültesi Dergisi*, C. 34, 1483–1500, Gazi University, Ankara, Turkey.

[80] Alakus, T. B.; Das, R.; Turkoglu, I. (2019). An overview of quality metrics used in estimating software faults. *2019 IEEE International Artificial Intelligence and Data Processing Symposium (IDAP)*, 4, 9–14, Inonu University, Malatya, Turkey. IEEE.

[81] Brito e Abreu, F.; Carapuça, R. (1993). Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, C. 26, Sayı 1,.

[82] Brito e Abreu, F.; Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. *Evaluating the Impact of Object-Oriented Design on Software Quality*, C. 26, 90–99. IEEE.

[83] Demirbaş, R. M. (2014). Nesne yönelik yazılım projelerinde öncelikli olarak test edilecek sınıfların yazılım ölçütleri yardımıyla belirlenmesine yönelik bir yöntem. Yüksek Lisans Tezi, Maltepe Üniversitesi Fen Bilimleri Enstitüsü, Bilgisayar Mühendisliği Anabilim Dalı. Tez No: 373774.

[84] Bansiya, J.; Davis, C. G. (2002). A Hierarchical Model for Object-Oriented Design Quality Assessmentn. *IEEE Transactions on Software Engineering*, C. 28, Sayı 1, 4–17.

[85] Ivory, M. Y.; Melody Y., R. R. S.; Hearst, M. A. (2001). Empirically validated web page design metrics. *Human Factors in Computing Systems*.

[86] Mendes, E.; Mosley, N.; Counsell, S. (2001). Web metrics— estimating design and authoring effort. *Web Metrics— Estimating Design and Authoring Effort*, C. 8, 50 – 57. IEEE.

[87] McCabe, T. J.; Watson, A. H. (140). *Structured testing a testing methodology using the cyclomatic complexity metric*. National Institute of Standards and Technology Technical Publications.

[88] Lee, M.-C. (2014). Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance. *British Journal of Applied Science & Technology*, C. 4, Sayı 21, 3069–3095.

[89] Murphy, J.; Robinson, J. (2007). Design of a research platform for en route conflict detection and resolution. *7th AIAA ATIO Conf, 2nd CEIAT Int'l Conf on Innov and Integr in Aero Sciences, 17th LTA Systems Tech Conf; followed by 2nd TEOS Forum*, 7803.

[90] Halstead, M. H. (1977). Elements of software science (operating and programming systems series). *Elements of software science (Operating and programming systems series)*. Elsevier Science Inc.

[91] Kan, S. H. (2002). Metrics and models in software quality engineering. *Metrics and models in software quality engineering*, 512. Addison-Wesley Longman Publishing Co., 2 edition.

[92] [URL] Jenkins, https://www.jenkins.io/, Erişim Tarihi: 1-1-2020.

Özgeçmiş

Ayşe KAHVECİ YETİŞ

KİŞİSEL BİLGİLER