NESNE YÖNELİMLİ YAZILIM TESTİ VE METRİK KÜMESİ DEĞERLENDİREN UZMAN MODÜLÜN GERÇEKLEŞTİRİLMESİ

M. Hanefi CALP

YÜKSEK LİSANS TEZİ
ELEKTRONİK VE BİLGİSAYAR EĞİTİMİ

GAZİ ÜNİVERSİTESİ
BİLİŞİM ENSTİTÜSÜ

Haziran 2011
ANKARA

M. Hanefi CALP tarafından hazırlanan NESNE YÖNELİMLİ YAZILIM TESTİ VE METRİK KÜMESİ DEĞERLENDİREN UZMAN MODÜLÜ GERÇEKLEŞTİRİLMESİ adlı bu tezin Yüksek Lisans tezi olarak uygun olduğun onaylarım.

Yrd. Doç. Dr. Nursal ARICI
Tez Yöneticisi

Bu çalışma, jürimiz tarafından oy birliği / oy çokluğu ile Elektronik ve Bilgisay Eğitimi Anabilim Dalında Yüksek lisans tezi olarak kabul edilmiştir.

Başkan : Doç. Dr. M. Ali AKCAYOL

Üye : Yrd. Doç. Dr. Nursal ARICI (Danışman)

Üye : Doç. Dr. O. Ayhan ERDEM

Tarih : 13/06/2011

Bu tez, Gazi Üniversitesi Bilişim Enstitüsü tez yazım kurallarına uygundur.

TEZ BİLDİRİMİ

Tez içindeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edilerek sunulduğunu, ayrıca tez yazım kurallarına uygun olarak hazırlanan bu çalışmada orijinal olmayan her türlü kaynağa eksiksiz atıf yapıldığını bildiririm.

M.Hanefi CALP

NESNE YÖNELİMLİ YAZILIM TESTİ VE METRİK KÜMESİ DEĞERLENDİREN UZMAN MODÜLÜN GERÇEKLEŞTİRİLMESİ

(Yüksek Lisans Tezi)

M. Hanefi CALP

GAZİ ÜNİVERSİTESİ
BİLİŞİM ENSTİTÜSÜ
Haziran 2011

ÖZET

Nesne yönelimli yazılımlar; kapsülleme, kalıtım ve çok biçimlilik gibi bazı özelliklere sahiptir. Bu özellikler, nesne yönelimli yazılımların test, bakım ve onarım faaliyetlerini önemli ölçüde etkilemektedir. Sözkonusu özellikleri ölçebilmek, test faaliyetlerini daha iyi yönetebilmek, riskleri azaltabilmek ve geliştirilen yazılımların kalitesi hakkında daha anlamlı yorumlar yapabilmek amacıyla da test metrikleri kullanılmaktadır. Hem nesne yönelimli yazılım test faaliyetlerinde hem de bu faaliyetlerde kullanılan metriklerin değerlendirilmesinde büyük eksiklikler bulunmaktadır. Dolayısıyla, nesne yönelimli yazılımların testi, analizi ve tasarım metrik değerlendirilmesi ile ilgili konuların yeniden ele alınması gerekir. Tam da bu noktada bu tez konusunun önemi ortaya çıkmaktadır.

Bu tez çalışmasında, özellikle nesne yönelimli yazılım geliştirme sürecindeki test faaliyetleri araştırılmıştır. Ayrıca, geliştirilen nesne yönelimli yazılımların tasarım metriklerini değerlendiren bir uzman modül gerçekleştirilmiştir.

Dolayısıyla bu çalışma ile nesne yönelimli test faaliyetleri konusunu açıklığa kavuşturarak nesne yönelimli tasarım metrik değerlendirilmesi ve bunun sonucu olarak da yazılım kalite değerlendirilmesi konularında bir otomasyon geliştirilmiştir. Çalışmanın literatüre önemli katkılar sağlayacağı düşünülmektedir.

Bilim Kodu : 702.1.014
Anahtar Kelime : nesne yönelimli test, test stratejisi, kalite, uzman modül, yazılım testi, test teknikleri, metrik
Sayfa Adedi : 114
Tez Yöneticisi : Yrd. Doç. Dr. Nursal Arıcı

OBJECT ORIENTED SOFTWARE TESTING AND IMPLEMENTATION OF EXPERT MODUL THAT EVALUATING THE METRIC SUITE
(M.Sc. Thesis)

M. Hanefi CALP

GAZİ UNIVERSITY
INFORMATICS INSTITUTE
June 2011

ABSTRACT

Object oriented software has some characteristics such as encapsulation, inheritance and polymorphism. These characteristics significantly influence the testing, maintenance and repair activities of object oriented software. Test metrics are used in order to measure these characteristics, to better manage the testing activities, to minimize the risks and to make more meaningful comments about the quality of the developed software. Both in the object-oriented software testing activities and in the evaluation of metrics used in these activities have major shortcomings. Therefore, the object-oriented software testing, analysis and issues such as design metric evaluation of this softwares must be addressed again. This is exactly appears where importance of this thesis project.

In this thesis project, specifically object oriented software test activities have been investigated. Additionally, an expert module that evaluates design metrics of object oriented softwares has been developed. It is aimed to develop an

otomation about object oriented desing metric evaluation and hence software quality evaluation by explaining object oriented test activities. It thought that the study will provide important contributions to the literature.

Science Code : 702.1.014
Key Words : object oriented testing, testing strategy, quality, expert module, software testing, testing techniques, metric
Page Number : 114
Adviser : Assist. Prof. Dr. Nursal Arıcı

TEŞEKKÜR

Çalışmalarım boyunca her konudaki yardım ve katkılarıyla beni yönlendiren çok kıymetli danışmanım Hocam Yrd. Doç. Dr. Nursal ARICI'ya, her zaman yanımda olan Bilişim Enstitüsü idarecileri ve personeline, manevi destekleriyle beni hiçbir zaman yalnız bırakmayan çok değerli eşim Emine CALP'a ve aileme teşekkürü bir borç bilirim.

İÇİNDEKİLER

Sayfa

Sayfa

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

Sayfa

SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış bazı simgeler ve kısaltmalar, açıklamaları ile birlikte aşağıda sunulmuştur.

Kısaltmalar	**Açıklama**
ABD	Amerika Birleşik Devletleri
CBO	Coupling Between Object Classes (Nesne Sınıfları Arasındaki Bağımlılık)
CKMDM	Chidamber ve Kemerer Metrik Değerlendirme Modülü
DIT	Depth of Inheritance (Kalıtım Ağacının Derinliği)
Id	Identity (Kimlik)
IEC	International Electrotechnical Commission (Uluslararası Elektroteknik Komisyonu)
IEEE	Institute of Electrical and Electronics Engineers (Elektrik Elektronik Mühendisleri Enstitüsü)
ISO	International Organization for Standardization (Uluslararası Standartlar Teşkilâtı)
JRE	Java Runtime Environment (Java Çalışma Anı Ortamı)
LCOM	Lack of Cohesion in Methods (Metotlardaki Uyum Eksikliği)

Kısaltmalar	**Açıklama**
NOC	Number Of Children (Alt Sınıf Sayısı)
RAM	Random Access Memory (Rastgele Erişimli Bellek)
RFC	Response For Class (Sınıfın Tetiklediği Metot Sayısı)
SEH	Structured Exception Handling (Yapılandırılmış Özel Durum İşleme)
UDM	Uzman Değerlendirme Modülü
UML	Unified Modelling Language (Birleşik Modelleme Dili)
US	Uzman Sistemler
VCL	Visual Component Library (Görsel Bileşen Kütüphanesi)
WMC	Weighted Methods Per Class (Sınıfın Ağırlıklı Metot Sayısı)
YBT	Yazılım Birim Testleri

1. GİRİŞ

Dünyanın en büyük endüstrilerinden biri hiç şüphesiz yazılım endüstrisidir. Geliştirilen ürünlerin büyüklüğü ve insan yaşamının her alanında yer alması bu endüstrinin önemini daha da arttırmaktadır. Yazılım endüstrisi tarafından üretilen yazılımlar cep telefonlarımızdan bilgisayarlara, vatandaşlık işlemlerinden sağlık sektörüne, askeriyeden enerjiye hem üretim hem de tüketim alanında kullanılmaktadır. Bu kapsamda akla gelen her alanla ilgili ihtiyaçların karşılanması için de projeler hazırlanmaktadır. Bu projelerde başarılı yazılımlar geliştirmenin temel yolu bu konudaki standartların ve belirlenmiş süreçlerin en iyi şekilde uygulanması ve daha önceki projelerdeki hatalardan en iyi şekilde ders alınmasından geçmektedir. Yazılım dünyasının gelişimine bakıldığında, yıllar geçtikçe geliştirilen yazılımların büyüklüğünün ve karmaşıklığının arttığı görülmektedir. Bu büyüme ve karmaşıklıkta, yazılımlarda hataların çok olması, güvenirlilik ve güvenliğin tam sağlanamaması gibi problemleri beraberinde getirmiştir [1].

Geliştirilen yazılımlarda, kaynağı ve sebebi değişmekle birlikte çeşitli hataların olması kaçınılmazdır. Çalışılan uygulama alanının kritikliğine göre bu hataların doğuracağı sonuçların biçimi değişebilmektedir. Bir finans uygulamasında yapılan küçük bir hata çok büyük miktarlarda para kaybına yol açabilecekken, bir askeri uygulamada yapılması muhtemel küçük bir hata mal kaybının yanı sıra can kaybına da neden olma riskini taşımaktadır. Bu nedenle profesyonel kullanımı planlanan tüm yazılımların içerisindeki hataların bulunması ve düzeltilmesi gerekir. Bu da yazılım test faaliyetleri ile mümkündür [2].

ABD'de farklı sektörlere farklı ölçeklerde yazılım geliştiren firmaların gerçekleştirdikleri 30000 yazılım projesi incelenerek 2004 yılı rakamlarına göre oluşturulan raporda, yazılım projelerinin sadece %34'ünün başarılı bir şekilde sonuçlandığı, %15'inin başladıktan sonra iptal edildiği; %51'inin ise tartışmalı, yani zamanında bitirilememiş veya gereksinimleri karşılamamış şekilde bitirilen projeler olduğu görülmektedir. Bu rakamlar farklı bir şekilde ifade edilirse yazılım sektöründe yapılan gemilerin %15'inin daha suya indirilmeden battığı sonucuna ulaşılır. Ayrıca rapora göre yazılım hataları ABD ekonomisine yılda 60 milyar $'a

mal olmaktadır. Bu rakam yazılım hatalarının ne kadar önemli bir kayba neden olduğunun açık ve çarpıcı bir göstergesidir [1].

Günümüzde yazılım hatalarının sebeplerini ortadan kaldıracak veya bu hataların oluşmasını engelleyecek çalışmalar yapılmaktadır. Ancak tüm bu çalışmalar yapılmasına rağmen yazılımlardaki veya projelerdeki hatalar tam manasıyla engellenememektedir. Dolayısıyla, bu noktada yazılım test faaliyetleri konusunun önemi ve buna bağlı olarak da yapılan çalışmanın amacı ortaya çıkmaktadır.

Yazılım test faaliyetleri içerisinde, sözkonusu faaliyetleri daha iyi yönetebilmek, geliştirilen yazılımlar hakkında daha anlamlı yorumlar yapabilmek ve daha başarılı sonuçlar elde etmek amacıyla test metrikleri kullanılmaktadır. Test metrikleri, yazılım kalite seviyesinin anlaşılması ve yazılım risklerinin azaltılması için kullanılan anahtar olaylardır. Buna ilaveten, çok etkili risk yönetim araçlarıdır ve hâlihazırdaki performansı ölçmeye yardım eder. Özetle metrikler, yazılım projelerini daha iyi test etmeyi, ürünün fonksiyonları hakkında daha fazla bilgi edinmeyi ve projelerin gelecekteki kalitesini daha iyi tahmin etmeyi sağlar.

Bu tez çalışmasında, özellikle nesne yönelimli yazılım geliştirme sürecindeki test faaliyetleri üzerine odaklanılmıştır. Ayrıca, geliştirilen nesne yönelimli yazılımların tasarım metriklerini değerlendiren bir uzman modül gerçekleştirilmiştir. Sözkonusu uzman modülün, bu alandaki çalışmalardan en ayırt edici özelliği geliştirilen yazılımların metrik bulgularını farklı yöntemlerle yorumlayarak veya değerlendirerek bu bulguları daha anlamlı kılmasıdır. Böylece, yazılım metrik değerlendirilmesi ve bunun sonucu olarak da yazılım kalite değerlendirilmesi konularında bir otomasyon sağlanmıştır.

Tezin ikinci bölümünde, nesne yönelimli yazılım testleri konusundaki temel kavramlar açıklanmıştır. Literatürde yaygın olarak kullanılan, ancak sık sık karıştırılan test, ölçme, ölçüm, metrik ve yazılım testi gibi kavramlar kısaca tanımlanmıştır. Yazılım hataları, sebepleri ve sonuçları ele alınmıştır. Yazılım geliştirme sürecinden, bu süreçteki test ve test metriklerinin öneminden, nesne yönelimli yazılım testlerinde bulunan sorunlardan ve bu sorunları belirlemede

kullanılan test araçlarından, yazılım testlerini olumlu veya olumsuz yönde etkileyen nesne yönelimli yazılım karakteristiklerinden bahsedilmiştir. Yazılım test süreçlerinden olan; bilgi toplama, test planlama, test tasarım, test çalıştırma ve test değerlendirme süreçleri açıklanmıştır. Son olarak yazılım test modelleri, nesne yönelimli test stratejileri ve test tasarım teknikleri geniş bir şekilde ele alınmıştır.

Üçüncü bölümde, yazılım kalite faktörleri ve kalite özellikleri kısaca açıklanmış olup kaliteli yazılımlar geliştirmek için hangi adımların izlenmesi gerektiği açıkça ortaya konulmuştur.

Dördüncü bölümde, literatürde yaygın olarak kullanılan Chidamber ve Kemerer'in nesne yönelimli metrik kümesi kısaca açıklanmış olup sözkonusu metrikleri değerlendirmek üzere geliştirilen uzman modül anlatılmıştır.

Son bölümde ise, genel olarak tez çalışması dahilinde yapılan çalışmalardan elde edilen bilgiler değerlendirilmiş olup bazı sonuçlar çıkarılmış ve çeşitli önerilerde bulunulmuştur.

2. NESNE YÖNELİMLİ YAZILIM TESTİ

Nesne yönelimli yazılım test faaliyetlerinde, literatürde yaygın olarak kullanılan, ancak sık sık karıştırılan birçok kavram vardır. Bu kavramlar, aşağıdaki bölümlerde kısaca açıklanmıştır.

2.1. Temel Kavramlar

Test

Bir sistemi el ile veya otomatik yollarla deneyerek belirli gereksinimlerin karşılanıp karşılanmadığının doğrulanması veya beklenen ile gözlenen sonuçlar arasındaki farkların belirlenmesi sürecidir [2].

Ölçme

Kavram olarak varlıkların özelliklerinin sayısallaştırılması olarak tanımlanabilir. Somut veya soyut bir varlığın sahip olduğu bir özelliğini, sayısal veya derecelendirilmiş bir veri olarak ifade etmektir [3].

Metrik (Ölçüt)

En temel ifadesiyle, direk ölçülebilen veya yapılan ölçümlere göre hesaplanan değerlere verilen addır [3].

Ölçüm

Belli bir metriğe (ölçüte) göre yapılan ölçme eyleminin sonucu veya bir ölçüt kullanılarak yapılan ölçme eyleminin sonucunda elde edilen veriye ölçüm denir [3].

Yanlış/Hata (Bug)

İstemeden yapılan veya beklenmeyen bir duruma sebep olan programdaki bir aksaklıktır [4].

Hata (Error)

Hesaplanan, gözlenen, ölçülen değerler veya durumlar ile gerçek, belirli, teorik olarak doğru olan değer veya durumlar arasındaki tutarsızlıktır [4].

Bozukluk (Failure)

Belirli performans gereksinimleri içerisindeki gerekli fonksiyonları uygulamada sistem veya bileşenin yetersizliğidir [4].

Aksaklık (Fault)

Programı uygularken istemeden oluşan veya beklenmeyen bir duruma sebep olan bir bilgisayar programındaki yanlış bir adım, süreç veya veri tanımlamasıdır [4]. Başka bir tanıma göre aksaklık (fault), aslında bir hatanın sebep olduğu sonuçtur [5].

2.2. Yazılım Testinin Tanımı ve Amacı

Yazılım testleri, geliştirilen bir yazılım sisteminin kontrolünün yapılarak ihtiyaçların ve gereksinimlerin tam olarak karşılanması için var olan işlemler bütünüdür [6]. Boch, yazılım testini, "korunmak için, akla gelmeyecek kadar gizli-kapalı belirsizlikleri karşılaştırma sürecidir." şeklinde tanımlamıştır. Test etme, aslında bir süreçtir. Amacı, bir sistemdeki hataları ve eksiklikleri bulmaktır. R. Vaderwall' göre, "Yazılım testi, ürünün davranışlarını öngörme/tahmin etme ve bu tahminlerin gerçek sonuçlarla karşılaştırılması sürecidir." Test sürecinin projeye ilk evrelerde entegre edilmesi maliyeti azaltır ve hata bulma şansını arttırır [7].

Literatürdeki farklı yazılım test tanımlarından bazıları şunlardır [1]:
- Yazılım testi, bir programın davranışının beklenen davranışa uymadığı durumları bulma işlemidir [1, 8].
- Bir sistemin veya bileşenin belirli koşullar altında çalıştırılması, sonuçların gözlenmesi, kaydedilmesi ve belirli özelliklerin değerlendirilmesi sürecidir [1,9].
- Test, hata bulma amaçlı planlı bir şekilde gerçekleştirilen eylemler dizisi, bir doğrulama yöntemidir [1, 10].

- Bir yazılım öğesinin mevcut ve olması gereken koşullar arasındaki farkın bulunarak analiz edilmesi sürecidir [1, 11].
- Test bir yazılım ürününün zayıf yönlerini veya makul hatalarını keşfetmek için gerçekleştirilen bir süreçtir [1, 12].

Testler, sadece test durumlarının iyi olmasını değil, aynı zamanda test edilecek tüm gereksinimlerin de iyi olmasını araştırır. Bununla birlikte, hataların tamamı test esnasında bulunamayabilir. Yazılım testinin önemi, olayların zamanında test edilmemesi ve kalitesiz bir ürün elde edilmesiyle ortaya çıkar [13]. Yazılım testi, bir yazılımın test durumları kullanılarak dinamik doğrulanmasıdır [14].

Testler, bir ürünün hazır olup olmadığını denemek için kullanılmasına rağmen diğer amacı da, hazır olmayan ancak doğru bir şekilde çalışan ürünü de sınamaktır. Bir başka görüşe göre testin amacı, yazılımın standartlara ve kabul şartnamesine uygunluğunu tespit etmek, yazılım geliştirme sürecine katkı sağlamak ve yazılım kalite güvencesi aktiviteleri uygulamaktır. Bunlara ilaveten, yazılım hataları ve yazılımın çalışma ekosistemine dair riskleri yönetilebilir kılmak, kodun ileriye dönük geliştirilme masraflarını azaltmak, ürün çalıştırılmadan önce kalitesini ve senaryolara uygunluğunu denetlemek, geliştirme sırasında gözden kaçan yanlışları bulmak ve bu yanlışların ileride de tekrarlanmasını önleyerek zaman ve maliyet tasarrufu yapmaktır. Özetle, kalite veya kabul edilebilirlik hakkında karar vermek ve problemleri keşfetmektir [15].

Bu kapsamda dikkat edilmesi gereken hususlar şunlardır:

- Dinamik olarak: Yazılım mutlaka çalıştırılarak test edilmelidir.
- Sınırlı sayıda: Yazılımın neredeyse sonsuz sayıda olabilecek çalışma alanlarının tümünün testi imkânsız olacağından, kritiklik düzeylerine göre sıralanıp yeterli görülen sayıda en kritikleri test edilmelidir.
- Uygun şekilde seçilmiş: Test edilecek davranışın doğasına uygun ve muhtemel riskleri göz önünde bulunduran testlerin gerçekleştirilmesidir.

– Beklenen davranışlar: Test edilecek yazılımın, kullanıcı beklentilerine, gereksinimlerine ve akla uygun, mantıklı beklentilere cevap verebildiğinin test edilmesidir [2].

Muller ve arkadaşları ise yazılım testini şu şekilde özetlemektedir [16];

- Bir kişiye, bir ortama veya bir şirkete zarar verebilecek yazılım hatalarını belirlemek.
- Hataların ana sebepleri ve etkileri arasındaki farkı ayırt edebilmek.
- Elde edilen örnekler yardımıyla testin niçin gerekli olduğunu tayin edebilmek.
- Test etmenin niçin kalite güvencenin bir parçası olduğunu ve elde edilen örneklere bakarak daha yüksek kalite sağlamak için nasıl bir testin olması gerektiğini saptamak.
- Hata(error), kusur(defect), aksaklık (fault), bozukluk(failure), yanlışlık(mistake) ve yanlış(bug) gibi terimleri yeniden hatırlamak.
- Yazılım projesinin başlangıcından bitmesine kadar olan süreçte hataları en aza indirebilmek.
- Projenin teslim edildikten sonraki teknik desteğini kolaylıkla yapabilmek, yeni gereksinimleri projeye daha rahat entegre edebilmek ve kullanım kolaylığı gibi konularda eksik olmamak.

2.3. Yazılım Hatalarının Sebepleri ve Sonuçları

Yazılım hataları iletişim eksikliği, programlama hataları, gereksinim değişikliği, zaman baskısı, dokümantasyon eksikliği, geliştirme araçları eksikliği ve donanım hataları gibi sebeplerden oluşmaktadır. Bu sebeplerin tam manasıyla giderilememesi sonucunda ülke ekonomisine veya insan sağlığına çok büyük zararlar veren yazılım felaketleri meydana gelmiştir. Yazılım hatalarının sebep olduğu felaketlerden bazıları şunlardır:

Arienne 5 Füzesi Faciası

Avrupa Uzay Ajansı ve Havacılık Dairesi 1996 yılında uydu taşıma amaçlı 7 milyar Euro'luk Arienne 5 isimli bir roket geliştirdi. 4 Temmuz 1996 günü fırlatıldıktan 37

saniye sonra roket kontrol yazılımındaki hatadan dolayı havada infilak etti. Arienne 5 füzesinde kullanılan bazı yazılımlar Arienne 4 füzesindeki bazı kodların yeniden kullanılmasıyla geliştirilmişti. Modül bazında testler gerçekleştirilmesine rağmen her zaman olduğu gibi genel sistem testleri zaman sıkışıklığından dolayı tam olarak gerçekleştirilemeden kullanıldı. Arienne 5 füzesinin uçuş profili Arienne 4'ten farklı olduğundan kalkıştan kısa bir süre sonra 16 bit/ 64 bit çevrimi sırasında bir değişkendeki taşma sonucu, kontrol sistemi devre dışı kalarak 500 milyon dolarlık kayba mal olan bu hata tarihteki en pahalı yazılım hataları arasında yerini almıştır[1].

Denver Havaalanı Otomatik Bilgi Sistemi

ABD'nin Denver kentinde inşa edilen dünyanın ikinci büyük uluslar arası havaalanının uçuşlardaki beklemeleri azaltmak, daha az çalışan maliyetiyle, daha hızlı ve daha doğru bagaj hizmeti sunmak için geliştirilen otomatik bagaj sistemi 186 milyon dolarlık bir yazılımla yönetilerek 31 Ekim 1993'de açılması planlanıyordu. Ancak bagaj sisteminde ortaya çıkan yazılım hataları nedeniyle sistemin hizmete alınması gecikmeli olarak 28 Şubat 1995 tarihinde gerçekleşti. Bu gecikmenin günlük maliyetinin 1 milyon dolara yakın olduğu ve gecikme nedeniyle oluşan toplam zararın 340 milyon doları bulduğu hesaplanıyor. Nihayetinde 70 milyon dolarlık yedek bir proje devreye sokuldu. O zamandan beri çeşitli sorunlarla çalıştırılan bu yazılımın da 2005 yılında artık iş göremeyeceği belirlenerek yenilenme kararı alındı [1].

Therac-25 Tedavi Cihazı

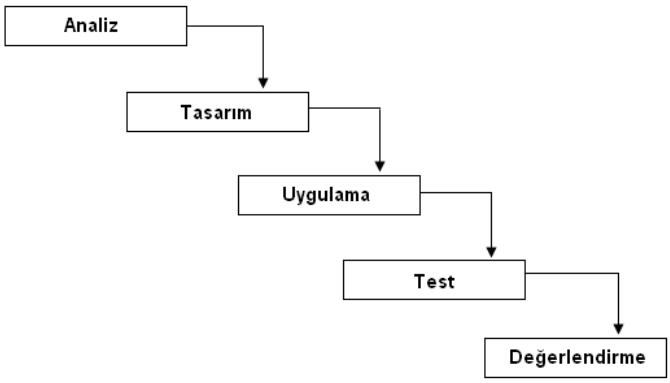
Therac 25 radyoterapide hastalara belirli dozlarda radyasyon uygulayarak tümörlerin yok edilmesinde kullanılan bir cihazdır. Bu cihaz, 1985-1987 yılları arasında farklı Amerikan hastanelerinde toplam altı kişinin ölümüne neden olmuştur. Cihaz "düşük güç modu" ve "yüksek güç modu" olmak üzere iki temel modda çalışabilmekteydi. Cihazın temel çalışma prensibinde "düşük güç modunda" düşük seviyedeki elektron ışını hastaya direk olarak uygulanmakta, "yüksek güç modunda" ise otomatik olarak hastayla cihaz arasına metal bir plaka konulmaktaydı. Bu cihazın kullanımı sırasında operatörün cihaza önce yanlışlıkla "yüksek güç moduna geç" komutu verilmiş ve

ardından bunu iptal edip "düşük güç moduna" alınmıştır. Bunun sonucunda cihaz hem işlemin iptal edildiği yönde bir hata mesajı vermiş hem de işlem iptal etmeyerek farkında olunmadan yüksek güçlü radyasyon, hastalara metal koruma kullanmaksızın uygulanmıştır. Hata mesajından dolayı operatörlerin işlemi birkaç kere tekrar etmesiyle de hasta çok büyük oranda radyasyona maruz kalmıştır. Bu sebeple hastalara normalden 100 kat fazla doz verilmiştir. Kontrol yazılımındaki bu hatadan dolayı Therac-25'in kontrol yazılımı ölümle sonuçlanan yazılım hataları arasında yerini almıştır [1].

2.4. Yazılım Geliştirme Süreci ve Bu Süreçte Testin Önemi

Yazılım hacminin, geliştirme gruplarının ve başarısız projelerin artmasıyla, projelerin başarısını arttırmak için geliştirme süreçleri tanımlanmaya başladı. Projelerin karmaşıklığının artmasıyla, havaalanı kontrolü ve uçuş rezervasyon sistemleri gibi, nesne yönelimli programlama ve gerçek zamanlı işletim sistemleri gelişti [7]. Yazılım geliştirme süreci bir yazılım ürününün geliştirilmesi ile ilgili bir grup eylemdir [1, 17]. Geliştirme süreci, temel olarak, yazılımın geliştirilmesini veya değerlendirilmesini amaçlayan faaliyetler kümesidir ve yazılım sürecindeki genel faaliyetler; belirleme (specification), geliştirme (development), onaylama (validation) ve değerlendirme (evolution)dir [7].

Yazılım mühendisliği, iyi tanımlanmış süreç içerisinde geliştiriciye kılavuzluk yapmak ve doğru yazılımın geliştirilmesini sağlamak için vardır. Yaşam döngüsü, yazılım geliştirme sürecini ve yazılımın bakım-onarımını organize etmek için kullanılır. Nihai amaç, makul bir fiyatta kaliteli yazılım ürünü üretmektir [18]. Eksiksiz bir proje yürütmek için uygulanması gereken yazılım geliştirme süreci Şekil 2.1'de gösterilmektedir.



Şekil 2.1. Geliştirme süreci: Şelale modeli [18]

- Analiz; üstesinden gelindiği düşünülen problemin çözümünü tanımlar. Bu kısımda; tam, kesin, açık ve anlaşılabilir biçimde problemin gereksinimlerini içeren çeşitli gereksinim belgeleri bulunur.
- Tasarım; sistem modelinin bileşenlerinin detaylı yapısı ve davranışlarını tanımlar. Bu kısımda, modelin her bir bileşeninde bulunan veri yapıları ve algoritmalar detaylı bir şekilde tanımlanır.
- Uygulama; programlama dilindeki tasarım çözümünün değişimini içerir. Bu, uygulamadaki program(lar)ın gerçek yapısıdır. Bu bölüm, oluşturulmuş programlar kümesidir.
- Test etme; elde edilen örnek kümesine ve uygun veriyle gereksinimlere göre program(lar)ın çalışıp çalışmadığını doğrular.
- Değerlendirme; kullanıcı için programın kurulumunu, teslimini ve sonuçların değerlendirilmesini kapsar.

Bu sürecin tamamından ortaya çıkan sonuç, uygun verilerle bilgisayar ortamında test edilen ve makul sonuçlar elde edilen iyi belgelenmiş bilgisayar programıdır. Yazılım geliştirme süreci, yazılım yaşam döngüsünün bir parçası olarak düşünülebilir. Yazılım geliştirme süreci, bir program elde etmek için üretilen görevlerin iyi tanımlı sırasıdır [18].

Yazılım geliştirme sürecinde asıl hedef, tasarımda iyi iş yapmayı, testte iyi iş yapmaya tercih etmek (hata ayıklayıcı yerine tasarım aracı kullanmak), kaliteyi hataları düzelterek değil önleyerek sağlamak, bakımları da hata önleyici bakım

olarak görmek, tasarım ile gereksinimleri eş zamanlı olarak güncellemek, sürece odaklanmayı etkinliklere tek tek odaklanmaya tercih etmek, çapraz-fonksiyonel (cross-functional) ekipler kurmaktan ve bu ekiplere müşteriyi katmaktan çekinmemek ve sorumluluğu tüm ekibe dağıtmaktır [19].

2.5. Yazılım Test Sürecinde Metriklerin Önemi

Paul Goodman [20] yazılım metriklerini, "Ölçüm temelli tekniklerin yazılım geliştirme sürecine sürekli uygulanması, süreç ve (onun) ürünlerinin gelişmesi için bu tekniklerin kullanılmasıyla birlikte, anlamlı ve zamanında bilgi yönetimini sağlamak" olarak tanımlar.

G.Gordon Schulmeyer [21] ise metriği, "bir sistemin, bileşenin veya sürecin verilen özelliğe sahip olma derecesinin nicel ölçümü" olarak tanımlar.

Test metrikleri, yazılım test sürecinin etkililiğinin önemli bir göstergesidir. Test metriklerini belirlerken ilk olarak, objektif bir şekilde ölçülebilen yazılım test süreci belirlenir. Bu belirleme, metrikleri tanımlamak ve hangi bilginin izleneceğini, bilgiyi kimin izleyeceğini ve ne kadar sıklıkta olacağını kararlaştırmak için kullanılabilir. Bu sürecin etkili bir şekilde izlenmesi, hesaplanması ve yönetilmesi gerekir. Buna ilaveten belirlenen metrikler açıklamalarıyla birlikte gösterilmelidir [22]. Test metrikleri; proje yöneticilerinin, metriklerin durumlarını anlamak ve yazılım teslimi üzerine çerçeve planındaki riskleri azaltmak için kullanabildiği anahtar "olaylar" dır. Test metrikleri, çok etkili risk yönetim araçlarıdır ve hâlihazırdaki performansı ölçmeye yardım eder. Metrikler, yazılım projelerini daha iyi kontrol etmeyi, ürünün fonksiyonları hakkında daha fazla bilgi edinmeyi ve projelerin gelecekteki kalitesini daha iyi tahmin etmeyi sağlar [23].

İyi metriklere sahip olmanın faydaları [23];

- Test metrikleri için veri toplama işlemi uzun dönem yönetimini, organizasyonun genişliğini ve üst düzey amaçların belirtilerini tahmin etmeye yardım eder.
- Tahmin için bir temel oluşturur ve performans eksikliklerini kapatmak için planlamayı kolaylaştırır.

- Kontrol/durum raporlaması için bir araç sağlar.
- Daha fazla test gereken risk alanlarını belirler.
- Daha bilinçli karar alarak faaliyetleri daha hızlı sürdürecek ölçütler sağlar.
- Potansiyel problemleri hızlı bir şekilde belirler, bu problemlerin çözülmesine yardım eder ve gelişim alanlarını belirler.
- Testin etkililiğini ve verimliliğini objektif bir şekilde ölçmeyi sağlar.

Genel anlamda, yazılım endüstrisindeki yazılım metriklerinin çok önemli olmasının sebebini açıklamak için şu temel nedenler gösterilir [24, 25];

Yazılım metrikleri;

- proje yöneticilerine proje gelişimi hakkında daha çok bilgi sağlar,
- özellikle, bir yazılım sisteminin tasarım ve mimari yapısı hakkında geliştirme yaşam döngüsünü daha iyi anlamaya yardımcı olabilir,
- yazılım geliştirmenin tüm evreleri esnasında süreç değerlendirmesini uygulayarak geliştirme sürecini daha iyi anlamaya yardımcı olabilir,
- çaba ve zaman konularında daha fazla zarara sebep olmadan yazılım geliştirme yaşam döngüsünün ilk evrelerinde yazılım tasarımındaki hataları ortaya çıkarmaya yardımcı olabilir,
- yazılım test aktivitelerini kolaylaştırır,
- yazılım projelerinin yaklaşık maliyet tahminini hesaplamada ve yazılım kalitesini değerlendirmede yardımcı olabilir,
- yeni aktivitelerin planlamasını ve tahminini kolaylaştırır. Metrikler yoluyla mevcut aktiviteyi ölçerek, yazılımların gelecekte daha maliyetli olmasını engelleme süreci giderek kolaylaşır.
- ürün, kalite, zaman yönetimi, bakım-onarım ve yeniden kullanılabilirlik gibi nitelik değerlendirme kriterlerini kullanarak nesne yönelimli teknolojinin yazılım geliştirme üzerindeki etkisini belirlemede yardımcı olabilir.
- farklı yeniden kullanım stratejilerinin faydalarını ve maliyetlerini tahmin etmede yardımcı olabilir.

- özellikle yeniden kullanılabilir metrikler, yazılım bileşenlerinin yeniden kullanılabilirliğini ve kalitesini belirlemek ve potansiyel faydalarını ortaya çıkarmak için yardımcı olabilir.

2.6. Yazılım Test Prensipleri

Günümüze ait yazılımlarda değişik hatalar vardır ve bu hatalar bazen bilgi kayıplarına ve zararlara sebep olur. Kalite, piyasaya sürülen yazılımlardaki hataları azaltmak için yazılım testi tarafından değerlendirilir. Geleneksel olarak yazılım testi, yazılımdaki mümkün olabilecek hataları bulmayı aramaktan ziyade, yazılımın karakteristiklerini ortaya çıkarmaya çalışır. Kaliteli bir yazılım ürünü ortaya çıkarabilmek için uyulması gereken bazı prensipler bulunmaktadır. Bu prensipler şunlardır:

- Test işlemi, bağımsız gruplar tarafından yapılmalı,
- Test için en iyi personel seçilmeli,
- Test, maksimum hata sayısı elde etme amacıyla planlanmalı,
- Geçersiz ve beklenmeyen giriş durumları, geçerli durumlar kadar iyi test edilmeli,
- Test esnasında test altındaki yazılımda değişiklik yapılmamalı,
- Test durumları ve test sonuçlarını içeren test raporu hazırlanmalı,
- Mümkünse, beklenen sonuçlar belirlenmeli ve rapora dahil edilmeli,
- Test ilerledikçe planlanmalı ve daha sonra güncellenmeli,
- Uygun bir test metodu seçilmeli [4].

2.7. Nesne Yönelimli Yazılım Testinde Bulunan Sorunlar

Geleneksel sistemlerin testi hakkında bir çok uygulama bilinmesine rağmen, nesne yönelimli gelişmeler için yeni çözümler gerektiren temel test problemleri mevcuttur. Bunlar, dört ana başlık altında açıklanır.

Hata Hipotezleri

Nesne yönelimli yazılımlar içerisindeki hata içerebilecek muhtemel durumları ortaya koyar.

Test Durum Tasarımı

Modellerden, özelliklerden ve uygulamalardan test durumlarını oluşturmak için hangi tekniklerin kullanılabileceğini tasarlamaya yardım eder.

Test Otomasyon Sorunları

Test durumlarının nasıl gösterileceği, nasıl uygulanacağı ve test sonuçlarının nasıl gösterilip değerlendirileceği gibi konuları ortaya koyar.

Test Süreç Sorunları

Testin ne zaman başlatılacağı, kim tarafından uygulanacağı ve etkili bir testin hangi süreçle sağlanacağı gibi konuları içerir [26].

2.8. Yazılım Test Araçları

İyi organize edilmiş bir test süreci doğru test yazılımlarıyla desteklendiğinde daha başarılı sonuçlar verecektir. Test yazılımları; test planlama, kontrol, tasarım, test verisi üretimi, test çalıştırması (koşturum) veya değerlendirilmesi gibi test eylemlerini desteklemek amacıyla geliştirilmiş olan yardımcı programlardır [1]. Literatürde, Radar, QuickBugs, Bugtrack, ZeroDefect, Roundup, Abuky gibi ücretli veya ücretsiz birçok yazılım test aracı mevcuttur [27]. Bu bağlamda, nesneye yönelik yazılım metriklerini kullanarak yazılım testi yapan birçok test aracı da geliştirilmiştir. Aşağıda, bu zamana kadar yapılan araştırmalar çerçevesinde açık kaynak kodlu veya araştırmacılara ücretsiz temin imkânı veren test araçlarından bazıları görevleri ile birlikte sunulmaktadır;

FindBugs, Maryland Üniversitesi tarafından geliştirilmiş açık kaynaklı bir statik kod analiz aracıdır. Yazılımın mevcut Java kodu üzerinde çeşitli analizler yaparak, yaygın yazılım hatalarını ve tasarım kusurlarını otomatik olarak kısa sürede bulabilmektedir [28].

Metrics, Eclipse projesine bir eklenti olarak geliştirilen açık kaynak kodlu bir başka yazılım test aracıdır. Eclipse geliştirme ortamındaki Java projelerine, tümleşik

biçimde çalışabilen program, yaygın kullanılan birçok yazılım metriğini otomatik olarak ölçerek geliştiriciye raporlamaktadır [29].

PMD ise, başka bir statik kod analiz aracıdır. Yine Java kodları üzerinde çalışan bu program mevcut kod üzerinde otomatik analizler yaparak olası kusurları kullanılmayan, tekrarlanmış ve gereksiz kod parçalarını hızlıca bulmaktadır [30].

Coverlipse, gerçekleme yani yazılım kodu ile gereksinimler ve test senaryolarının arasındaki örtüşme ilişkilerini inceleyen açık kaynak kodlu bir Eclipse eklentisidir. Program yazılım geliştirmenin bu üç temel aşaması arasındaki örtüşme ilişkilerini analiz ederek aradaki boşlukları ortaya çıkartmaktadır [31].

CheckStyle test aracı, yazılımın yapısından çok formatı ile ilgilenen bir başka açık kaynak kodlu yazılım aracıdır [32],

SDMetrics, yukarıda açıklanan yazılımlardan farklı olarak kod üzerinde değil de UML tasarım dökümanları üzerinde çeşitli görsel ve sayısal analizler yapabilen bir test aracıdır [33],

Coverity, Java'nın yanı sıra C++ ve C program kodları üzerinden çalışabilen ticari ve oldukça kapsamlı bir kod analiz aracıdır. Analiz sonuçlarının yanı sıra bazı otomatik düzeltmeler de yapabilen program yazılım dünyası tarafından çok büyük maliyetli ve kapsamlı projelerde kullanılmaktadır. Bu programın diğer bir önemli özelliği de, statik analizin yanı sıra çalışma esnasında dinamik analizleri de yapabilmesidir [34].

2.9. Testi Etkileyen Nesne Yönelimli Yazılım Karakteristikleri

Test faaliyetlerini olumlu veya olumsuz yönde etkileyebilecek nesne yönelimli yazılım karakteristikleri ve kısaca açıklamaları aşağıda verilmiştir.

2.9.1. Durum bağımlı davranış (State dependent behavior)

Nesneler, durum ve durumlara bağımlı davranışlara sahiptir. Yani, bir işlemin bir nesne üzerindeki etkisi o nesnenin durumuna bağlıdır ve nesnenin durumu

değişebilir. Bu yüzden, işlemlerin toplam etkisi test edilmelidir [35]. Bir nesnenin durumu, metotların giriş ve çıkışlarının bir parçası olduğu için, sistematik olarak nesne durumlarını ve bağlantılarını keşfetmeye ihtiyaç duyulur [36].

Nesnelerin durum davranış testi, program veya özellik temelli olabilir. Nesnelerin durum bağımlı davranışları yazılımın kaynak kodundan çıkarılabilir ve bu bilgi o nesnenin durum diyagramında görülür. Daha sonra, programı test etmek ve çıktıları gözlemlemek için nesne durum diyagramından test durumları ve test verisi oluşturulur. Dolayısıyla, durum kartı veya durum diyagramı sayesinde nesne yönelimli yazılım test faaliyetlerinde karmaşıklığın ve daha çok zaman harcamanın önüne geçilmiş olur [35].

2.9.2. Kapsülleme (Encapsulation)

Kapsülleme, fiziksel veya mantıksal bir alan içerisindeki bir veya daha fazla elemanı tamamen kapsayan bir araçtır. Nesne yönelimli bir içerikte kullanılan kapsülleme kavramı, sözlük tanımından temel olarak farklı değildir. Bu, bazı şeylerin etrafındaki bir kapsül, kavramsal bir engel oluşturmak anlamına gelir. Bir sistemi tanımlamak için kullanılan tasarım yaklaşımına bağlı kapsüllemenin farklı seviyeleri vardır. Kapsülleme üç farklı şekilde tanımlanır; düşük seviye, orta seviye ve yüksek seviye. Düşük seviye kapsülleme, diziler ve kayıtlar gibi şeyleri içerir. Kapsüllemenin orta seviyesinde, alt programlar ve alt işlemler vardır. Üst seviye veya yüksek seviye kapsüllemede ise sınıflar, paketler ve nesneler bulunur. Nesne yönelimli yazılım testi üzerinde kapsüllemenin en büyük etkisi, bir birimin tanımındaki değişikliktir [37, 38].

Daşdemir [39], kapsüllemeyi şu şekilde tanımlamaktadır;

"Tek bir birim (unit) içerisinde prosedur, fonksiyon ve verinin birbirine bağlanmasına (verinin fonksiyon veya prosedür tarafından kullanılmasına) kapsülleme denir."

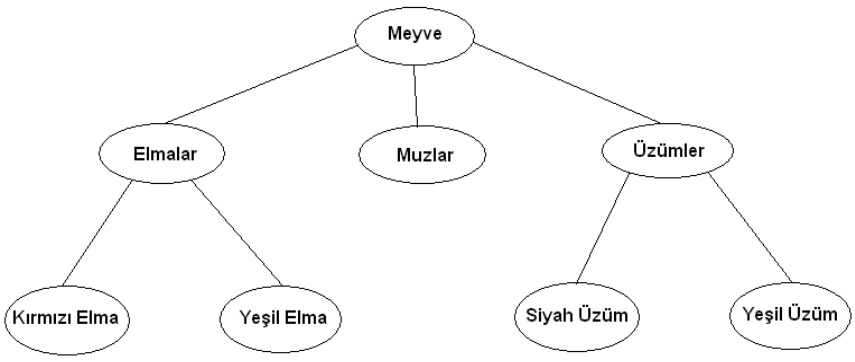
Kapsüllemenin fazla kullanılması karmaşıklığa yol açar. Dolayısıyla, geliştirilen yazılımlarda kapsüllemenin gereğinden fazla uygulanması, nesne yönelimli bir

yazılımın anlaşılırlığını, yeniden kullanılabilirliğini ve dayanıklılığını olumsuz yönde etkiler. Bu durum, yazılımın geliştirilmesine, bakımına ve testine daha fazla zaman-çaba harcanmasına sebep olur (37).

2.9.3. Kalıtım (Inheritance)

Nesne yönelimli programlama, hiyerarşik bir yapıya dayalı sistemdir. Bu sistemdeki akış genel bir sınıftan özel bir sınıfa doğrudur. Yani genel bir sınıftan birçok sınıf türetilebilir. Bu mekanizma kalıtım olarak adlandırılır. Konu daha basite indirgenirse "Kalıtım, başka bir nesneden yeni bir nesne türetme olayıdır. Bu nesne önceki ebeveyn nesnenin verilerini (değişken veya alan) ve metotlarını (prosedür veya fonksiyon) devralır." diye söylenebilir [39].

Kalıtım, ebeveyn nesnelerin özelliklerini ve davranışlarını koruyarak, yeni nesneler oluşturma becerisidir. Bu kavram, VCL (Visual Component Library – Görsel Bileşen Kütüphanesi) gibi nesne hiyerarşilerin oluşturulmasını sağlar. İlk önce jenerik nesneler oluşturulur ve daha sonra bu nesnelerin daha dar işlevselliğe sahip daha özel yavruları oluşturulur. Kalıtımın avantajı, ortak kodun paylaşımına imkân vermesidir. Şekil 2.2'de bir kalıtım örneği gösterilmiştir. Kök nesne olan Meyve, bütün meyvelerin ata nesnesidir. Meyve, Elmalar nesnesinin atası ve Elmalar da, diğer bütün elmaların atasıdır.



Şekil 2.2. Bir kalıtım örneği [40]

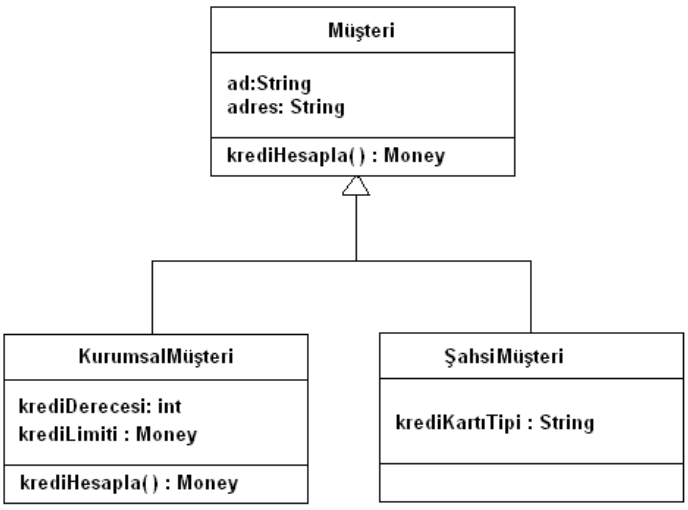
Booch'a göre [41] kalıtım, bir sınıfın bir veya daha fazla sınıfta tanımlı davranış veya yapıyı paylaştığı sınıflar arasındaki ilişkiyi tanımlar.

Kalıtım derinliğinin çok yüksek olması test edilebilirliğin çok düşük olduğunu, aksi halde nesne yönelim ilkelerinin fazla kullanılmadığını gösterir (42). Bu durum verimliliği, yeniden kullanımı, anlaşılırlığı ve test edilebilirliği olumsuz yönde etkiler. Dolayısıyla geliştirilen yazılımın test, bakım ve onarım faaliyetleri de hem bütçe hem de zaman bakımından artacaktır (43).

2.9.4. Çok biçimlilik (Polymorphism)

Çok biçimlilik biyolojiden nesne yönelimliye aktarılmış bir kavramdır. Biyolojide bir türdeki canlılar aynı türden olmalarına rağmen kendilerine has bazı davranışlar sergileyebilirler. Nesne yönelimli programlama dillerinde de bir sınıftan türetilen canlıların farklı şekillerde davranmaları sağlanabilir [44].

Çok biçimlilik, farklı sınıflardan yaratılmış olan nesnelerin, aynı mesaja farklı tepkiler vermesi olarak da tanımlanabilir. Alt sınıfın miras aldığı bir işlem, genellikle olduğu gibi kullanılır. Şekil 2.3'te "Müşteri" sınıfında tanımlı olan bir krediHesapla() metodu, "ŞahsiMüşteri" alt sınıfında kullanılır. Ancak, alt sınıfın ihtiyaçlarına uygun olarak bu işlemlerin tekrar tanımlanması gerekebilir ve buna geçersiz kılma (iptal etme- overriding) adı verilir. Alt sınıflarda yeniden tanımlanan işlemler, bu halleriyle kullanılırlar. Bu durumda Şekil 2.3'te "KurumsalMüşteri" sınıfında "krediHesapla()" işleminin çok biçimli olduğu söylenebilir.



Şekil 2.3. Çok biçimliliği gösteren UML diyagramı [45]

Çok biçimliliğin yazılımcıya sağladığı en büyük avantaj, farklı türden nesneleri, ana sınıflarını baz alarak aynı ortamda yönetebilme ve gerektiğinde özel işlevselliği tekrar kazanabilmesidir. Bu da hem karmaşıklığı hem de yazılım test faaliyetlerini azaltan bir özelliktir [46].

2.9.5. Soyut sınıflar (Abstract classes)

Bazı programlama modellerinde üst sınıfların direkt olarak kullanılması istenmez. Bu modellerde sadece üst sınıftan türetilmiş olan sınıflardan oluşturulmuş nesnelerin kullanılması istenir. Bu tip durumlarda yardıma soyut sınıflar koşar [44].

Soyut sınıf, üzerinden doğrudan nesne tanımlanamayan sınıftır. Ancak, kendinden türeyen alt sınıflardan nesne oluşturulabilir. Çok biçimliliğin sağlanabilmesi için üst sınıftaki işlem soyut tanımlanmalıdır. Böylece alt sınıfta yeniden tanımlanan işlem, çok biçimlilik özelliği kazanır [45]. Soyut bir sınıf, bir veya daha fazla soyut metodu içerir [18]. Soyut sınıflar geliştirilen yazılımları karmaşıklıktan kurtarır, dolayısıyla yazılım test, bakım ve onarım faaliyetleri daha da kolay olacaktır [37].

2.9.6. İstisna yönetimi (Exception handling)

Yapılandırılmış istisna yönetimi (SEH-Structured Exception Handling), hata yönetimini merkezleştirmeyi ve normalleştirmeyi sağlayan bir metottur. Kaynak içinde istilacı olmayan (non-invasive) hata yönetimini ve hemen her türlü hata koşulunu hassas bir şekilde yönetme becerisini sunar. İstisnalar, en temel düzeyde, belirli bir hatanın doğası ve konumu hakkında bilgi veren sınıflardır. Bu, istisnaların kolay uygulanmasını ve diğer sınıflar gibi uygulamalarda kullanılmasını sağlar [40].

2.9.7. Bilgi gizleme (Information hiding)

Nesne yönelimli dünyadaki diğer kavramların pek çoğu gibi bilgi gizleme de standart bir tanıma sahip değildir. Bir nesnenin tüm ayrıntılarını gizleme süreci, tipik olarak o nesnenin yapısı iyi bir şekilde gizlenerek mümkündür. Bilgi gizleme, başka bir ifadeyle, bazı gereksiz ayrıntıların tamamını saklamaya veya gizlemeye izin verir. Bilgi gizleme, ulaşılmaz bir nesne oluşturur [37].

2.9.8. Yeniden kullanım (Reuse)

Nesne yönelimli yazılımlarda bölünmüş sınıflardan (derived class) bahsedilir. Bu bölünmüş sınıflar, temel sınıfın ara yüzünün tamamı veya bir kısmı yeniden kullanılır. Bu yeniden kullanım, dile bağlı çok profesyonel bir konudur [37].

Yeniden kullanım, bir modelin belirli ölçüde diğer modeller tarafından yeniden kullanılabilmesidir [47]. Bir problemi yeniden uygulamak için tasarıma izin veren nesne yönelimli tasarım karakteristiklerinin varlığını belirtme anlamındadır [48]. Tasarım kalitesinin mümkün olduğu kadar yeniden kullanımı desteklemesi gerekir. Ne kadar çok desteklerse yazılım test faaliyetlerini o kadar olumlu yönde etkiler [49].

2.10. Yazılım Test Süreci

Yazılım test süreci; önce planlanan, sonra icra edilip sonuçları kayıt altına alınarak belgelendirilen bir dizi eylemden oluşur. Bu süreç, geliştirilen yazılımdaki hataların varlığına odaklanır [1]. Testin kalitesi, yazılım test sürecinin olgunluğuna bağlıdır ve iyi bir test süreci;

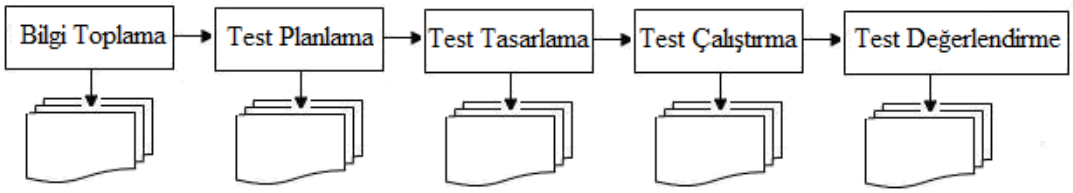
- test planı ve risk yönetimi,
- özel test ortamı ve test araçları,
- test durum tanımı,
- test otomasyonu,
- test bölümüne teslimdeki formaliteyi,
- testi çalıştırmayı,
- test sonuç analizi,
- test raporu,
- test etkililiğinin ölçümü,
- son durum incelemesini ve süreç geliştirmeyi kapsar [43].

İyi bir test sürecinde, testin etkililiğini kararlaştırmak için ölçümler sürdürülür ve test tamamlanmasından sonra proje gereksinimlerinin karşılanıp karşılanmadığını belirlemek için son durum incelemesi yapılır. Test süreci geliştirme, test gruplarının

etkililiğini artıracak test sürecinin sürekli gelişimi için önemlidir ve test grupları, aşağıdakine benzer soruları cevaplamak için ölçümler kullanır.

- Yazılımın mevcut kalitesi nedir?
- Şu anda ürün ne kadar dayanıklıdır?
- Şu anda ürün piyasaya sunulması için hazır mıdır?
- Teslim edilen yazılımın kalitesi ne kadar iyidir?
- Ürün kalitesi diğer ürünlerle nasıl karşılaştırılır?
- Yazılıma uygulanan test ne kadar etkilidir?
- Ne kadar problem vardır?
- Geriye yapılması gereken ne kadar test kalır? [43].

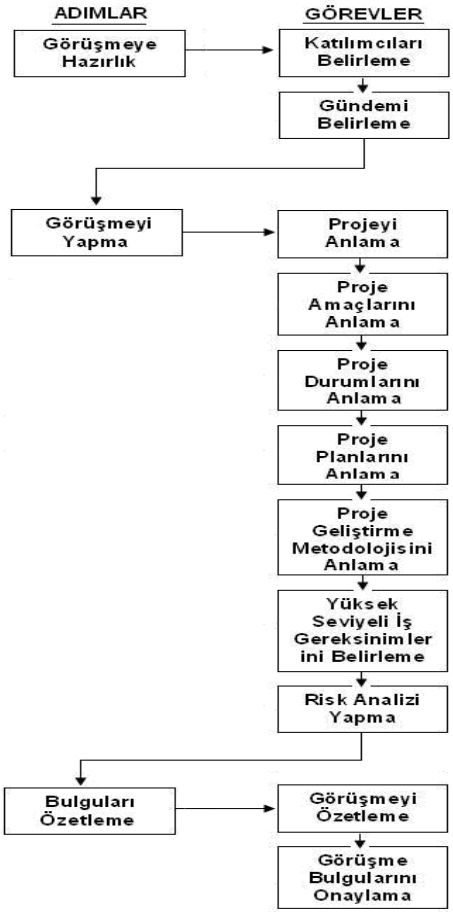
Yazılım test süreci, Şekil 2.4'te açıkça gösterilmektedir.



Şekil 2.4. Sadeleştirilmiş test süreci

2.10.1. Bilgi toplama

Bilgi toplamanın amacı, yazılım geliştirme projesiyle ilgili bilgileri elde etmek, proje geliştirme alanını anlamaya çalışmak ve bir test planı yapmaya başlamaktır. Diğer bilgiler, gerekli görüldüğü takdirde proje gelişimi esnasında toplanabilir.



Şekil 2.5. Bilgi toplama (adımlar/görevler) [13]

2.10.2. Test planlama

Yazılım projelerinde proje başarısını doğrudan etkileyen eylemlerin başında iyi bir test planlaması gerekmektedir. Bu amaçla yazılım projelerinin farklı aşamalarında test planları hazırlanır. Bu planlarda test edilecek yazılım parçaları, özellikler (işlevsellik, performans, güvenlik, kullanılırlık vb.), icra edilecek görevler, çıktılar, gerekli kaynaklar, sorumluluklar, takvim ve gerekli onaylar tanımlanır. Aynı zamanda projelerde test eylemlerini daha doğru ve etkin tanımlamak için farklı seviyelerde birden fazla test planı üretilebilir. Örneğin, projenin en genel test yaklaşımını belirten Test Ana Planı, sistem testleri yaklaşımını belirten Sistem Test Planı veya yazılım testleri yaklaşımını belirten Yazılım Test Planı gibi. Tüm bu planlar IEEE 829-1998 standartları kapsamında belirtilen test planı şablonuna göre geliştirildiğinde aşağıdaki başlıkları kapsar:

1. Test plan tanımlayıcısı
2. Giriş
3. Test öğeleri
4. Test edilecek özellikler
5. Test edilmeyecek özellikler
6. Yaklaşım (Strateji)
7. Geçme / Kalma kriterleri
8. Testi durdurma ve teste yeniden devam etme kriterleri
9. Test çıktıları
10. Test görevleri
11. Çevresel gereksinimler
12. Sorumluluklar
13. Personel ve eğitim gereksinimleri
14. Takvim
15. Riskler ve öngörülmeyenler
16. Onaylar [1].

Bir test planı, test aktiviteleri ve hizmetleri gibi yaklaşımları tanımlayan bir belgedir. Test planı, gelişim döngüsünün ilk evrelerinde hazırlanmalı ve analiz, tasarım ve kodlama aktivitelerinin etkileşimlerini geliştirmeye yardımcı olmalıdır. Test planı; testin amaçlarını, alanını, stratejisini ve yaklaşımını, test prosedürlerini, test ortamını, test tamamlama kriterlerini, test durumlarını, test edilecek bileşenleri, uygulanacak testleri, test zaman çizelgesini, personel gereksinimlerini, raporlama prosedürlerini, tahminleri, riskleri ve olasılık planlamasını belirler. Bir test planı geliştirilirken, test planının uygun bireyler tarafından erişilebilir ve hazır olduğundan emin olunmalıdır.

Test planı hazırlamanın iki yöntemi vardır. İlk yaklaşım, her bir test planının detaylı bir şekilde gösteren ana test planıdır. Detaylı test planı, şelale geliştirme yaşam döngüsündeki belirli evreleri doğrular. Test planı örnekleri; sınıf/birim, entegrasyon, sistem ve kabul testlerini kapsar. Diğer detaylı test planları, uygulama yükseltmelerini, regresyon testini ve paket kurulumlarını kapsar. Birim test planları, kod merkezlidir ve çok detaylıdır. Fakat, sınırlı alanları olduğu için kısadır. Sistem

veya kabul test planları, tam olarak yazılım birimi değil de tüm sistemin kara kutu görünümünü veya fonksiyonel testi üzerine odaklanır. İkinci yaklaşım, bir test planıdır. Bu yaklaşım, sık sık kabul/sistem test planı olarak isimlendirilir fakat birim, entegrasyon, sistem ve kabul testi ve testlerin tamamlanması için planlama düşüncelerinin hepsini içeren bir test planıdır. Bir test durumu, adım adım süreci tanımlar ve test koşturulur. Bu durum, testin amaçlarını ve koşullarını test etmek için gerekli adımları, veri girdilerini, beklenen sonuçları ve gerçek sonuçları kapsar. Yazılım, ortam, sürüm, test ID, ekran ve test tipleri gibi diğer bilgileri de sağlar [43].

İyi bir test planı hazırlamak için tamamlanması gereken adımlar şunlardır;

1. Test amaçlarını belirlemek
2. Test yaklaşımı geliştirmek
3. Test ortamı belirlemek
4. Test özelliklerini geliştirmek
5. Testi planlamak
6. Test planını gözden geçirmek ve onaylamak**.**

Test planı, yazılımın yüksek kalitede olup olmadığını, herkes tarafından anlaşılıp anlaşılmadığını ve sorumluluklarını yerine getirip getirmediğini tahmin etmek için etkili gruplar tarafından düzenlenir. Test planı, proje esnasında kontrollü bir şekilde tekrar düzeltme yapılabilir [43].

Etkili bir test, iyi bir planlama ve çalıştırmayı gerektirir. Test planı;

- yapılacak olan testin hedefini belirlemeyi,
- zaman tahmini, kaynakları, personeli, donanımı, yazılımı ve araçları,
- ihtiyaç duyulan kaynakları sağlamayı,
- test ortamını sağlamayı,
- görevlere personel tayin etmeyi,
- planı belirlemeyi,
- risk ve olasılık planlarını tanımlamayı,
- süreç takibini ve doğru adımlar atmayı,
- geçilen, engellenen ve başarısız testlerin düzenli test durumlarını sağlamayı,
- projenin hedefi değişirse yeniden plan yapmayı,

- herhangi bir bölümü anlamak için son durum incelemesi yapmayı içerir [43].

Test proje planının amacı, belirli bir olayda başarılı test için kural oluşturmaktır. Burada en önemlisi belgelemedir. Çünkü belgeler, test projesini yönetmeye yardım eder. Eğer bir test planı kapsamlıysa ve dikkatlice düşünülmüşse, test koşturma/çalıştırma ve analizi düzgün bir şekilde yürümelidir. Test proje planı, özellikle spiral ortamda sistem sürekli değiştiği için gelişen bir belgedir. İyi bir test planı için şunlar söylenebilir;

- hataların çoğunu tespit etme şansı yüksektir,
- kodların birçoğu için test kapsamı sağlar,
- esnektir,
- kolay ve otomatik bir şekilde çalıştırılır ve tekrarlanabilir,
- uygulanacak test tiplerini belirler,
- beklenen sonuçları açıkça belgeler,
- hata bulunduğunda hatayı düzeltme imkânı sağlar,
- test amaçlarını açıkça tanımlar,
- test stratejileri aydınlığa kavuşur,
- test çıkış kriterlerini açıkça tanımlar,
- gereğinden fazla değildir,
- riskleri teşhis eder,
- test gereksinimlerini belirler,
- testin teslim edilebilirliğini belirler [13].

2.10.3. Test tasarlama

Test planlama süreci başarılı bir şekilde tamamlandıktan sonra "Test Tasarımı" evresi başlar. Test tasarımı, öncelikle belirli test gereksinimlerini ve mümkün olan tüm fonksiyonel çeşitlerin gelişimini içerir. Test etme, test tasarım mühendislerine test verilerinin uygunluğunun doğrulanmasına izin verir ve test verilerinin düzenini sağlar. Testleri ve test verilerini tasarlama, test sürecinin çok fazla zaman alan kısmıdır. Aktivitelerin ayarlanması da çok önemlidir. Eğer testler, gereksinimleri test etmezse, o testler geçersiz sayılır. Test verileri, testlerin amacını yansıtmazsa, o testler de geçersizdir [50]. Test tasarım süreci; test ortamının hazırlanması, test

durumlarının yazılması ve test yordamlarının hazırlanması faaliyetlerinin icra edilmesiyle tamamlanır [13].

Test ortamının hazırlanması

Test planı geliştiricileri fiziksel test imkânlarını, donanımı, yazılımı, ağları, gerekli teknik desteği, test etmek için gerekli olan özel yazılım yerlerini denetlerler ve sözkonusu öğeler için bir plan hazırlarlar. Test ortamının amacı, test aktiviteleri için gerekli fiziksel ortamı sağlamaktır. Buna göre, test ortamının ihtiyaçları belirlenir ve uygulamaya başlamadan önce tekrar gözden geçirilir. Test ortamının içerdiği ana bileşenler, fiziksel test imkânı, teknolojiler ve araçlardır. Test imkânı bileşeni; fiziksel durumu içerir. Teknolojiler bileşeni; donanım platformları, fiziksel ağ ve onun tüm bileşenleri, işletim sistemi yazılımı ve yardımcı yazılımlar gibi diğer yazılımları içerir. Araçlar bileşeni ise otomasyon test araçları, test etme kütüphaneleri ve yazılım desteği gibi herhangi bir özel test yazılımını içerir. Test ortamı ve çalışma alanı kurulması gerekir. Bu, bireysel çalışma alanından resmi test laboratuarlarına kadar çeşitlilik gösterir. Test ediciler ve geliştirme takımları arasında her olayda bir yakınlık olması önemlidir. Bu iletişim kolaylığı ve ortak amaçta buluşma kolaylığı sağlar. Elde edilen test araçları, donanım ve yazılım teknolojileri kurulması gerekir. Bu, test donanımının ve yazılımının kurulumunu, satıcı, kullanıcı ve bilgi teknolojileri personeliyle koordinasyonu kapsar. İletişim ağlarının da kurulması ve test edilmesi gerekir [13].

Test ortamı, organizasyon, iş ve proje gereksinimlerine göre değişir. Yüksek kalitenin kısıtlı olduğu önemli alanlarda, laboratuar mühendislerine tahsis edilen test laboratuarı kullanılabilir ve yazılım test ediciler tarafından laboratuar zamanının belirlenmesi gerekebilir. Küçük bir proje için küçük organizasyonlardaki bir çalışma yeri test ortamı olarak yeterli olabilir. Test ortamı, yazılımın hatasız olduğunu doğrulamada projeyi desteklemek içindir ve bir test ortamı belirlemek için önemli derecede maddi yatırım gerekebilir. Test ortamı, yazılımın doğruluğunu onaylamak için gerekli donanım ve yazılımı içerir [1, 51].

Test durumlarının yazılması

Test durumu, belirli bir program parçasının çalıştığının veya bir gereksinimin doğrulandığının gösterilmesi için kullanılan girdiler, gerçekleştirilmesi gereken adımlar ve beklenen sonuçların belirtilmesidir [1, 51]. Bir test durumu adım adım bir testin nasıl icra edileceğini tanımlar. Bir test durumunda olması gerekenler;

1. Testin durumunun amacı ve gerçekleştirilme şartı,
2. Test durumu ile ilgili test ortamının adım adım kurulması,
3. Girdi verileri,
4. Beklenen sonuç,
5. Gerçekleşen sonuç,
6. Yazılımın sürüm tanımı,
7. Yazılımın çalışma ortamı,
8. Test ID [1].

Bir test durumu yazılıma sorulan bir sorudur ve test edilen öğenin sadece bir özelliğini test etmelidir [1, 52].

Test yordamlarının hazırlanması

Test yordamı, her bir test durumunun test ortamının kurulması, koşturulması ve sonuçlarının değerlendirilmesi için ayrıntılı direktifler, açıklamalar listesi içeren ve test planı temel alınarak geliştirilen belgedir [1, 9]. Diğer bir ifade ile test yordamı "bir yada birden fazla test durumunun koşturulması için hazırlanan detaylı açıklamaları içeren belgedir". Her bir test durumu için ayrı test yordamları olabileceği gibi bir grup halinde olan test durumları için de bir test yordamı hazırlanabilir [1].

2.10.4. Test çalıştırma

Test çalıştırma, ilk olarak test amaçlarının karşılanmasıyla başlar ve test etmek için tüm kriterler uygulanır. Test çalıştırma esnasında test edicilere birçok sayıda gözlemlenen test senaryoları ve davranışları sağlanmasına rağmen testler, test prosedürlerine göre çalıştırılmalıdır. Test çalıştırma aktivitelerinin merkezinde

beklenen sonuçlarla gerçek sonuçlar bulunur. Test ediciler çok dikkatli olmalı ve bu görevlere odaklanmalıdırlar, aksi takdirde, hatalar bulunmadığında veya doğru olması gereken davranışlar yanlış bulunduğunda, testi tasarlama ve uygulama işlerinin tamamı boşa yapılmış olabilir. Eğer, beklenen ve gerçek sonuçlar hesaplanmazsa, hata olayı meydana gelir. Olaylar, (test hedefinde bir hata olsun veya olmasın) sebepleri tespit etmek ve olayların sonuçlarıyla veri toplamak için dikkatle irdelenmelidir. Bir hata belirlendiğinde, test özelliklerinin doğru olduğunu tespit etmek için dikkatlice değerlendirilmelidir. Bir test özelliği, test verilerinde problemler içerme, test belgelerinde veya onun çalıştığı yolda hata gibi sebeplerin çoğu konuda yanlış olabilir. Test temelinde ve test amacında değişiklikler olduğu için, çoğu kez başarılı bir şekilde çalışsa bile test özelliği yanlış olabilir. Test ediciler, gözlenen sonuçların yanlış test sonuçları olabileceği durumunu da düşünmelidirler. Test çalıştırma esnasında, test sonuçları, uygun bir şekilde kaydedilmelidir. Çalışan fakat sonuçları kaydedilmeyen testler, etkisizliğini ve gecikmeleri göstererek doğru sonuçları belirlemek için tekrarlanmak zorundadırlar. Test amaçları, test araçları ve test ortamlarının tamamı değerlendirildiği ve kaydedildiği için test edilmiş özel sürümler belirlenmelidir. Test kaydetme, testin çalışmasıyla ilgili detayların kronolojik kayıtlarını sağlar. Sonuçları kaydetme, hem bireysel testleri hem de olayları kapsar [16].

Test uygulama ve çalıştırmayı izlemek için metrikler;

- test ortamlarının ayarlanma yüzdesini,
- test veri kayıtlarının yüklenme yüzdesini,
- test durumları ve vakaların çalıştırılma yüzdesini,
- test vakaların otomatikleştirilme yüzdesini kapsar [16].

2.10.5. Test değerlendirme

Öncelikle metrikler analiz edilmelidir. Metrikler, daha etkili kararlar verebilmek ve geliştirme sürecini desteklemek için kullanılır. Bunun amacı, test sürecini kontrol etmek için metrik prensiplerini uygulamaktır. Daha sonra, proje durum raporu, test hata detayları raporu ve hata raporu gibi raporlar hazırlanarak ara rapor yayınlanır [13]. Test çerçeve planı çıkartılır. Geliştirme esnasında test çerçeve planının sürekli

izlenmesi gerekir. Amaç, son durumu göstermek için test planını güncellemektir. Son olarak, gereksinim değişikleri belirlenir. İlk gereksinim belirlemede, işlevsel kompozisyon, fonksiyonel pencere yapısı, pencere standartları ve sistem gereksinimleri gibi test fonksiyonları analiz edilir. Değişen yeni gereksinimler ise şu konuları içerebilir;

- yeni GUI arayüzleri ve bileşenleri,
- yeni fonksiyonlar,
- değiştirilmiş fonksiyonlar,
- elenmiş fonksiyonlar,
- yeni sistem gereksinimleri, donanım gibi,
- ilave sistem gereksinimleri,
- ilave kabul gereksinimleri.

Her bir yeni gereksinimin test planında, test tasarımında ve test dilinde tanımlanması, kaydedilmesi, analiz edilmesi ve güncellenmesi gerekir [13].

2.11. Nesne Yönelimli Yazılım Test Stratejileri

Test stratejisi, bir yazılım projesinde test hedeflerine nasıl ulaşılacağını, proje içerisinde uygulanacak olan test sürecini ve geliştirilecek yazılımın özelliğine göre icra edilecek olan test tiplerini ve test seviyelerini tanımlar [1].

Stratejinin amacı, birçok görev ve test projesine karşı yaklaşımı açıklamaktır. Strateji çok açık, kesin ve projenin amacına göre olmalıdır. Test stratejileri testin büyük kısmını ve iş konularını kapsar. Örneğin;

- Müşteri, proje ve uygulamaya genel bakış,
- Testin alanı,
- Eklemeler, çıkarmalar ve tahminler,
- İyileştirme planları, grup yapısı ve hazırlık çalışması,
- Test yönetimi, metrikler ve geliştirme,
- Test ortamı, değişim kontrolü ve ürün stratejisi,
- Test teknikleri, test verisi ve test planlanması,

- Roller ve sorumluluklar,
- Üst düzey çerçeve planı,
- Giriş ve çıkış kriterleri ,
- Hata raporlama ve süreç izleme,
- Regresyon test süreci,
- Performans testi analiz yaklaşımı,
- Test otomasyonu ve test aracı belirleme,
- Organizasyon içinde risk belirleme, maliyet ve kalite yaklaşımları,
- Kriterleri belirleme ve analiz etme,
- Test projesinin teslimi [13].

Nesne yönelimli yazılım testi için, sınıf hiyerarşi testi ve çok biçimlilik stratejilerini kapsayan standart herhangi bir metodoloji yoktur. Harrold, McGregor ve Fitzptrick tarafından [53], kök sınıftan dal sınıfa doğru türeyen ve bir ağacı test etmeyi gösteren hiyerarşik bir test stratejisi ileri sürülmüştür. Bu strateji; hangi testin uygulanacağını göstermemektedir. Harrold [46], nesne yönelimli test konusu üzerinde çalışmış ve kalıtımla ilişkilendirerek sınıfların hiyerarşik yapısını kullanan sınıf test metodolojisini tanımlamıştır. Bu metodolojiler şu aktiviteleri içerir;

- Temel sınıflar için bir test kümesi tasarlama,
- Her bir fonksiyon elemanını ve fonksiyon elemanları arasındaki etkileşimi test etme,
- Birbiriyle kalıtımlı yeni alt sınıflar tanımlama,
- Kalıtım testini sürekli güncelleme,
- Alt sınıflardaki test edilebilen yeni özellikleri belirleme,
- Alt sınıfların içerisindeki kalıtımlı özellikleri yeniden test etme,
- Alt sınıfların ve yeni test durumları gerektiren alt sınıfların özelliklerini doğrulamak için yeniden kullanılabilen test kümesindeki test durumlarını tanımlama.

Kung et al ise [54], nesne yönelimli yazılım test stratejisini başka bir yol ileri sürerek tanımlamaktadır. Nesne yönelimli test, analiz ve tasarım modellerinin doğruluğunu ve uygunluğunu değerlendirerek başlar. Nesne yönelimli test stratejisi, genel

düşünceyi değiştirir; birim, kapsüllemeden dolayı genişler; entegrasyon, sınıflara odaklanır ve onaylama(validation) geleneksel kara kutu teknikleri kullanarak uygulanır.

2.11.1. Yazılım test modelleri

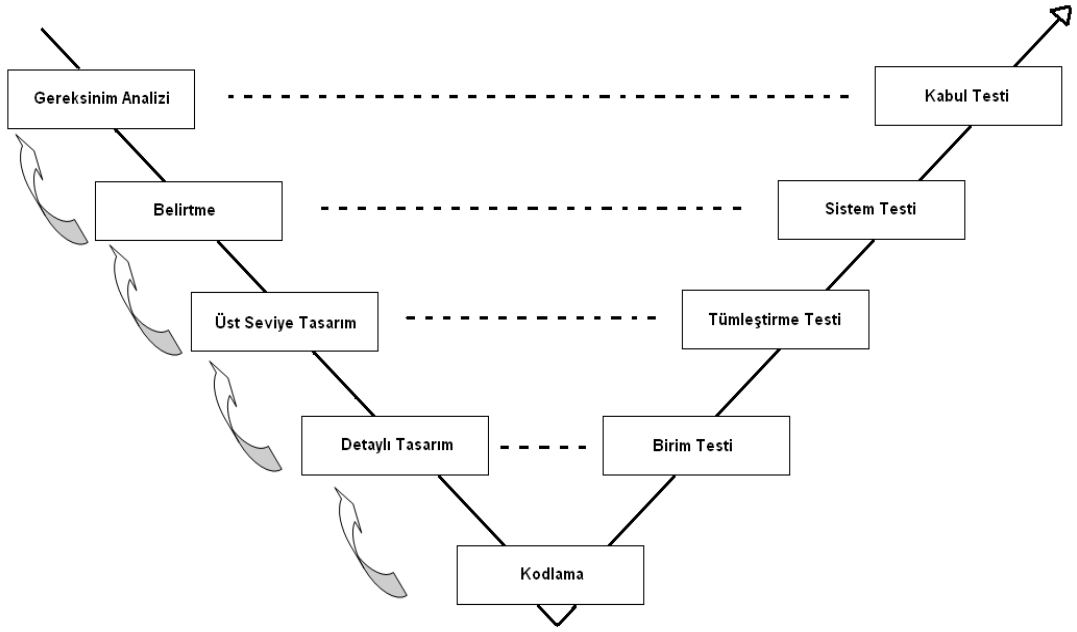
Yazılım testinde bir model; bir fikri, bir diyagramı veya fiziksel bir sembolü temsil eder. Bir model, bir sistemi veya bir görüşü belirli yollarda düşünmeye yardım eder. Modelleme uygulamaları, çeşitli teknolojilerin kullandığı teknik bilgiler ve deneyimler üzerine temellenmiş potansiyel problem alanları hakkında fikir edinmeye yardımcı olur. Modelleme, aynı zamanda, uygulamada bulunan hataların tekrar edilebilir durumlarını elde etmeye veya daraltmaya ve gereksinim belgelerine dayanarak farklı test etme fikirleri üretmeye de yardımcı olur. Yazılım testinin çeşitli modelleri vardır. Bunlar, V Modeli, W Modeli ve B Modeli'dir [4].

V Modeli

V Modeli, aslında, şelale yazılım süreç modelinden geliştirilmiştir. V Modeli, evrelerin başlı başına şekilsel sırasını tanımlar. V modeli aynı zamanda, doğrulama ve onaylama ile eş anlamlıdır. Yazılım geliştirme sürecinin gerçek evrelerini tanımlar. Ele alınan belirli evrelerin mantıksal sırasını sağlar ve evreler arasındaki mantıksal ilişkileri tanımlar. V modeli, yazılan test belgelerini mümkün olduğunca gösterir ve geliştirme ile test etme faaliyetlerine eşit derecede ağırlık verir. Buna ilaveten, yazılım geliştirme sürecinin haritasını izlemek için basitlik ve kolaylık sağlar [4].

V Modelinde dört süreç evresi vardır;

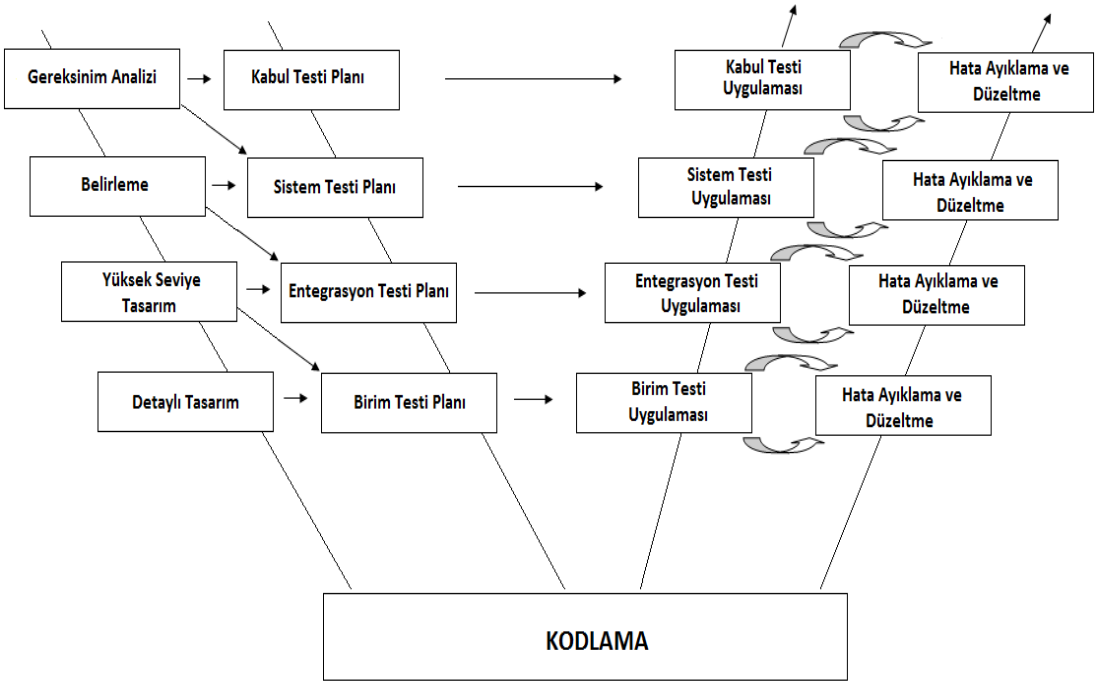
- Gereksinim Analizi
- Belirleme
- Üst Seviye Tasarım
- Detaylı Tasarım



Şekil 2.6. V modeli [4]

W Modeli

Genellikle modeller, geliştiriciler tarafından çeşitli eksikleriyle kabul edilir. Çoğu durumda, test aktiviteleri uygulamadan sonra başlar. Genellikle, çeşitli test evreleri ve test kuralları arasındaki ilişki net değildir. Modellerin hiçbiri, test dönemi esnasında test, hata ayıklama ve değişik görevler arasındaki ilişkide yeterli değildir. Bu dezavantajların hepsi, W modeli diye adlandırılan başka bir model geliştirmeyi gerektirmiştir. W modeli, aslında V modeli üzerine temellenir. Genellikle test sürecine, halihazırdaki modellerde çok az önem verilir. Test eylemi eşit bir şekilde icra edilsin diye, W modelinin temelini oluşturan model içerisine ikinci bir V modeli entegre edilir. Testlerin önemi ve test esnasındaki özel aktivitelerin düzeni W modelinde açıktır. Bu model açıkçası, test durumlarının yapısından, çalışmasından ve değerlendirilmesinden daha çok test faaliyetlerini vurgular. W modeli gerçek uygulamalar için daha uygun görünür ve yazılım karışıklıklarının hızlı bir şekilde arttığı durumlarda uygulanabilir. Bununla birlikte, farklı aktivitelerin zaman veya personel gibi kaynaklar için eşit gereksinimlere sahip olduğu da gözükmektedir [4].



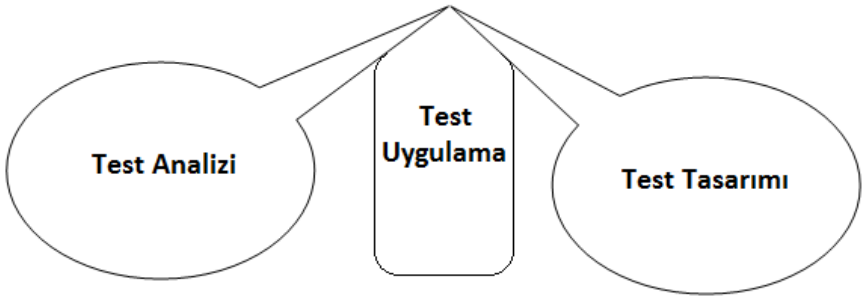
Şekil 2.7. W modeli [4]

B Modeli

Yazılım ürünlerini test etmek için diğer önemli bir model de kelebek modeli olan B Modeli'dir. Bu modelin üç kısmı kelebeğin yapısını verir. Modelin ilk kısmı kelebeğin sol kanadını gösterir ve *Test Analizi'*ni tanımlar. İkinci kısım sağ kanadını gösterir ve *Test Tasarımı*'nı tanımlar. Son olarak, üçüncü kısım olan kelebeğin gövdesi *Test Uygulaması*'nı gösterir. Analiz, etkili planlamayı sağlayan anahtar bir faktördür. Test analizi esnasındaki gerekli belgeler, test personeli tarafından dikkatli bir şekilde toplanır ve sonuç analiz raporu; yazılım gereksinim belirleme, fonksiyonel özellikler, yapı ile ilgili belgeler ve kullanım-durum dokümanları belgelendirilir. Analiz raporu, uygulamanın fonksiyonel akışının, modüllerin sayısının ve etkili test zamanın anlaşılmasını sağlar. Analist, her bir gereksinimi doğrular [4].

Test tasarımı, kelebek modelin diğer bir önemli bileşenidir. Test durumlarının tasarımı ve uygulaması, uygulamadaki tekrarlar gibi tasarım hatalarını doğrulamaya ihtiyaç duyar. Test tasarımı, yazılımdaki hataların giderilmesi, tespit edilmesi ve

önlenmesi gibi faaliyetlerdir. Test analiz evresindeki çıktılar, test tasarımın temelidir. Test Uygulaması esnasında ise, tasarlanan test durumları işleme konur. Bu test durumları eksiksiz, ekonomik çalışan, tekrar edilebilir, görevlere uygun ve yeteri kadar izlenebilir olmalıdır. Test durumlarının çalışmasından sonra test uygulama raporu ve hata raporu teslime hazır bir final olarak teslim edilir [4].



Şekil 2.8. B modeli [4]

2.11.2. Nesne yönelimli yazılım test seviyeleri

Literatürde kabul gören nesne yönelimli test seviyeleri; "Gereksinim Testi, Tasarım Testi, Birim Testi, Entegrasyon Testi, ve Sistem Testi" olmak üzere beş başlık altında toplanmaktadır [4]. Bunların dışında, "Kabul Testi" de mevcuttur ancak genelde "Sistem Testi" içerisinde kabul edilmektedir.

Gereksinim testi

Gereksinim testi, yazılım geliştirme dünyasında ihmal edilen alanlardan biridir. Gereksinim test araçları çok az bulunur. Modern yazılım mühendisliği tekniklerinin gelmesiyle gereksinimleri test etme, yazılım testinin temel bir bileşeni olmuştur. Nesne yönelimli teknolojinin ortaya çıkmasıyla, analiz ve tasarım evreleri birleşmiştir. Analiz, bir problemi açık ve eksiksiz bir şekilde tanımlamayı amaçlar. Nesne yönelimli gereksinim analiz evresinde analist, müşteriyle konuşur ve tasarımcılara bilgiyi iletir. Gereksinim analizinin sonucu, performans ve gereksinim kaynakları ile birlikte sistem fonksiyonlarının tanımını içerir. Bu sonuçlar, aşağıda kısaca tanımlanmış çeşitli resmi veya resmi olmayan belgeleri ortaya çıkarır [4].

Ön modelleme (Prototyping)

Problemin gerçek parametrelerini eksiksiz bir şekilde belirlemek için kullanılan nesne yönelimli analizin yaygın bileşenlerinden biridir. Aynı zamanda müşterilerin, sistemde istedikleri şeyleri anlamaları için yardımcı olur.

Grafik kullanıcı arayüzü (Graphical user interface)

Müşterilerin sistemle iletişime geçtikleri bileşenlerden biridir. Farklı görüntü planları, geri besleme ve bir sonraki düzeltmeler için müşterilere gösterilir.

Gereksinim belirleme belgesi (Requirement specification document)

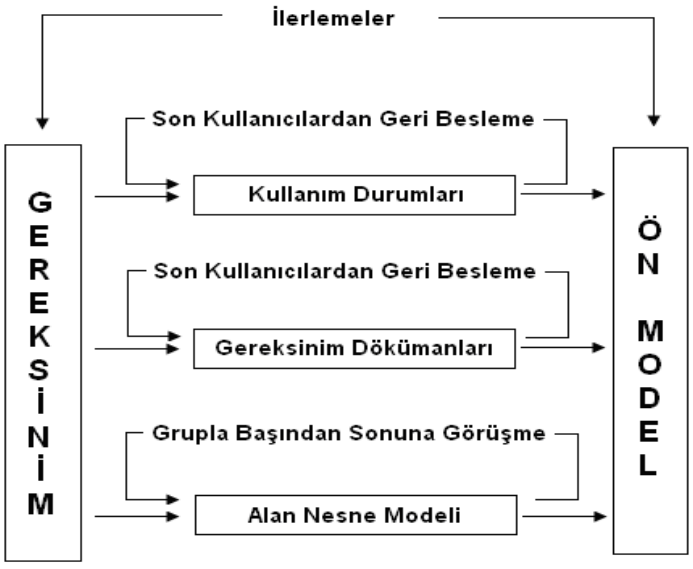
Bu belge, istenen sistem için tüm gereksinimleri içerir. Buna ilaveten, istenen özelliklere ve beklenen sistemin davranışlarına bakarak müşteri ve geliştirme grubu arasında anlaşmaya hizmet eder.

Alan nesne modeli (Domain object model)

Bu model, problemin alanını belirlemek için kullanılır.

Kullanım durumu (Use case)

Kullanım durumları, istenen sistemin kullanabildiği yol haritasıdır. Bu, istenen sistemin davranışlarını eksiksiz bir şekilde belirlemek için en faydalı araçlardan biridir [4].



Şekil 2.9. Nesne yönelimli gereksinim testi [4]

Gereksinim testinin temel amacı, toplanan gereksinimlerin kalitesini sağlamak için analiz evresinden elde edilen sonuçları doğrulamak ve onaylamaktır. Bu evrede yapılan test, geliştirme yaşam döngüsünde ilk süreçteki hataları yok etmeye yardım eder. Gereksinim testleri aşağıdaki konuları kapsar [4]:

Doğruluk (Correctness): Tasarımcılar için en temel ilkelerden biridir. Gereksinim analizi, yazılım geliştirme sürecinin ilk adımı ve süreçteki diğer aktivitelerin temeli olduğu için, çeşitli gereksinim analizi araçlarından elde edilen durumlar doğru olmalıdır. Gereksinimlerin doğruluğu, müşterilerin isteklerindeki belirsizliği ortadan kaldırır ve gereksinim durumlarındaki gereksiz kelimeleri yeniden düzeltmeyi sağlar [4].

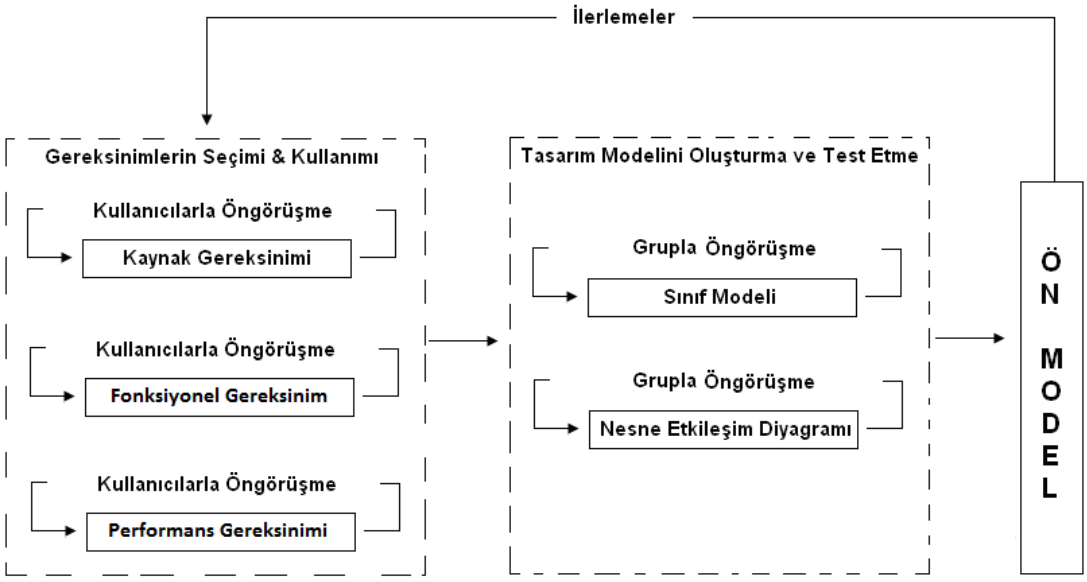
Bütünlük/Tamlık (Completeness): Gereksinimler; fonksiyonel, performans, kaynak ve kalite gereksinimleri olarak sınıflandırılabilir. Fonksiyonel gereksinimler, belirli girdi koşulları altında sistemin istenilen davranışlarını belirler. Performans gereksinimleri, çeşitli fonksiyonlar için sistemin yanıt sürelerini düzenler. Kaynak gereksinimi, istenen sistem için kaynaklara erişilebilirliği gösterir. Sistemin belirli koşullar altındaki güvenirliliği ve sağlamlığı, kalite gereksinimleri tarafından belirlenir. Dolayısıyla, sözkonusu gereksinim sınıflarının tamamının eksiksiz bir şekilde elde edilip test edilmesi gerekir [4].

Tutarlılık (Consistency): Gereksinim testinin başka bir yönü de tutarlılıktır ve çelişkili gereksinimleri araştırır. Bu çelişkili gereksinimler belirlenmeli ve gereksinim testi esnasında elenmelidir. Gereksinim testleri, yazılım geliştirme yaşam döngüsünün ilk başlarında yazılım sistemi içerisinde yanlışlıkla yapılan hataları ortaya çıkarmaya yardım eder. Gereksinim testi için genel yaklaşım; kulanım durumlarını, gereksinimleri belirleme dökümanlarını onaylayarak ve kullanıcılara ön modeli göstererek tamamen son kullanıcılardan geri besleme almaktır. Son kullanıcılardan alınan bu geri besleme, gereksinim testindeki temel konuları gösterir. Analiz, gereksinim dökümanlarının temel bileşenlerini belirlemede kullanıcılara kılavuzluk yapar. Alan nesne modeli, daha resmi olaylardaki problem alanını belirlemek için oluşturulur. Geri besleme, daha sonra, gereksinim dökümanlarındaki boşlukları doldurmak için geliştirme grupları tarafından tekrar gözden geçirilir.

Geliştirme grupları, sınıflar ve onların davranışları arasındaki yanlış ilişkileri düzelterek oluşturulan alan modelini geliştirmek için üzerinden geçme/inceleme (walktrough) yolunu takip eder [4].

Tasarım testi

Tasarım, bazı resmi belgelere bakarak temel problemin araştırılmasından başlayan bir süreçtir. Tasarım testi, analiz evresinden uygulamadaki mantıksal modellere, gereksinimlerin tam bir şekilde ortaya çıkarılmasıyla alakalıdır. Yazılım tasarım testi çaba, sözleşme, kaynaklar ve uygun yaklaşımı ortaya koyar. Tasarım testi kolay değildir. Ön tiplemeler oluşturulana kadar uygun bir şekilde hesaplanamaz. Tam ve doğru olması için analiz modelinden tasarım modeline dönüşüm testi, temel olarak el ile uygulanan bir süreçtir. Şekil 2.10'da nesne yönelimli tasarım testindeki temel aktiviteler gösterilmektedir [4].



Şekil 2.10. Nesne yönelimli tasarım testindeki temel aktiviteler [4]

Nesne yönelimli bir tasarım testinin amacı, normal olarak yazılımın tasarım hatalarını ortaya çıkarmaktır.

Bütünlük: İyi bir tasarımın önemli özelliklerinden biri, çözülmesi düşünülen tüm problemleri tümüyle çözmektir. Tasarım problemlerinin çözülmesi için, fonksiyonel gereksinimler kadar, performans ve kaynak gereksinimlerinin de karşılanması gerekir. Böylece tasarım testinin diğer temel amacı; fonksiyonel, kaynak ve performans gereksinimlerinin tümünün karşılanıp karşılanmadığını sağlamaktır [4].

Yapılabilirlik: Tasarım çözümü, iyi bir tasarım için gerçekleştirilebilmelidir. Tasarım testi, tasarımın belirli zaman içinde ve ekonomik çerçevede uygulanmasının mümkün olup olmadığını sağlar [4].

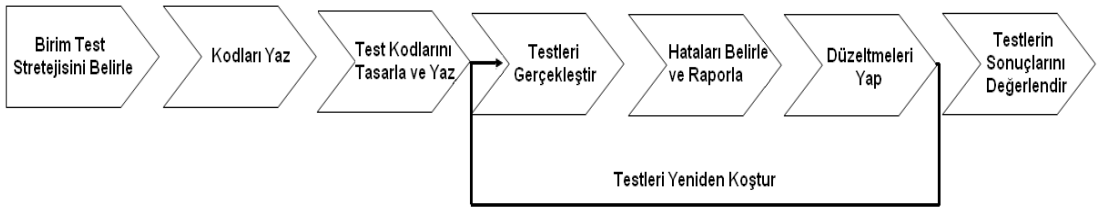
Doğruluk/Tamlık: Bir tasarım, girdi ve çıktılarının tamamı için doğrulanmalıdır. Yani, girdi ve çıktı ilişkisinin doğru veya yanlış olduğu ispatlanması halinde tamlıktan, doğruluktan bahsedilebilir. Yazılım karışıklığı ve gerekli çaba, tasarım testi esnasındaki temel sorundur. Tasarım testi hem el ile hem de otomatik olarak uygulanabilir. El ile uygulanan teknikler, tasarımı tekrar gözden geçirme, tasarımı baştan sona uygulama ve tasarımı denetlemedir. Tasarımı baştan sona uygulama ve ön modelleme, nesne yönelimli yazılım test etmede endüstri profesyonelleri tarafından uygulanan iki temel yaklaşımdır [4].

Sınıf / birim testi

Nesne yönelimli yazılımda sınıf, testin en temel birimidir. Sınıf-seviyeli test, mühendislikte sağlam bir çalışmayı gerektirir. Sınıf testi, olabilecek en pratik tasarımı temsil eder ve sınıflar için en iyi belgeleme biçimi sağlar. Genelde, birimi geliştiren kişiler tarafından uygulanır. Sözkonusu test için önceden gerekli olan şey, sınıfın kalitesini sağlamak amacıyla motivasyondur. Bu test, yazılım testinin bir parçasıdır, tam ve doğru bir şekilde uygulanmalıdır. Bir sınıf; fonksiyonel, yapısal ve etkileşim olmak üzere üç farklı açıdan test edilmelidir. Bir sınıf test faaliyeti; doğruluk, bütünlük ve erken test kelimelerini kapsar. Genelde, bir sınıfın her metodu birim testinden geçmelidir. Bununla birlikte, tam bir şekilde çalışan metotlara birim testi uygulamaya gerek yoktur [4].

Sınıf/birim testinde, modülün arabirimi, veri yapısı, kontrol yapıları arasındaki ana yollar, hata arama yolları ve modül sınırları sınanmaktadır. Sınıf/birim testi, kaynak program düzenlenerek gözden geçirilip sözdizimi hataları düzeltildikten sonra uygulanmaktadır [45]. Yordam, fonksiyon veya prosedür gibi bir kod parçasının kendisinden beklenen işlevselliği doğru olarak yerine getirildiğini ve hata içermediğini göstermek için gerçekleştirilen testlerdir [1, 11].

Sözkonıısu testi uygulamanın en rahat yolu bir yazılım test aracı kullanılmasıdır. Eğer bir test yazılım aracı kullanılmayacak ise test kodları yazılım geliştirmeye paralel olarak geliştirilmelidir. Bu nedenle sınıf/birim testler yazılım geliştirmenin bir parçası olarak düşünülmeli ve planlama buna uygun olarak gerçekleştirilmelidir. Başarıyla sonuçlanan sınıf/birim testler, yazılım tümleştirme ve sistem testlerinden önce geliştirilen yazılımın genel olarak istenen gereksinimleri karşıladığının bir göstergesidir. Bu testlerin amacı, geliştirilen ve derlenebilen en küçük kod parçalarının (metot, prosedür, fonksiyon, sınıf vb.) kendilerine ait görevleri doğru şekilde yerine getirdiğinin doğrulanmasıdır. Bu amaçla, testler gerçekleştirilirken test edilecek olan birim diğer birimlerden izole edilir. Test edilen birim için gerekli olan girdileri sağlayacak diğer birimlerin yerine o metotların koçanları (stub) kullanılır. Böylece diğer birimdeki olası bir hatanın test edilen birimi etkilemesi engellenir. Şekil 2.11'de genel birim test süreci görülmektedir [1]



Şekil 2.11. Genel birim test süreci [1]

Sınıf/birim testler yazılım geliştiricileri tarafından gerçekleştirilir. Bu testlerin başarılı bir şekilde yapılmasının ilk adımı, süreçte görüldüğü gibi birim test stratejisinin net olarak belirlenmesi ve tüm geliştiricilerin belirlenen bu strateji üzerine eğitimlerinin gerçekleştirilmesidir. Aşağıda, basit bir birim testi örneği gösterilmektedir [1].

Örneğin, pozitifMi(int k) isimli bir yordamı olsun. Bu yordam aşağıdaki kodu içermektedir.

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

Bu yordamın test edilmesi için aşağıdaki test durumları oluşturulabilir;

Test Durumu 1:

```
public void testDurumu_1() {
bool td1=false;
int m=3;
td1=pozitifMi(m);
if(td1==true)
system.out.println("Test Durumu 1: GEÇTİ");
else
system.out.println("Test Durumu 1:  KALDI");
}
```

Test Durumu 2:

```
public void testDurumu_2() {
bool td2=false;
int m=-4;
td2=pozitifMi(m);
if(td2==false)
system.out.println("Test Durumu 2: GEÇTİ");
else
system.out.println("Test Durumu 2:  KALDI");
```

} [1].

Yazılım sınıf/birim testlerinde dikkat edilmesi gereken noktalar

Genellikle projelerde tasarıma ait bilgiler içeren veriler alt seviye gereksinimleri olarak adlandırılmaktadır. Alt seviye gereksinimleri ile fonksiyonel gereksinimler ve birim testleri arasında izlenebilirlik tabloları oluşturularak her fonksiyonel gereksinim için hangi tasarımın yapıldığı ve bu tasarımın hangi testlerde test edildiği rahatlıkla izlenebilmektedir. Nesne yönelimli yazılımlarda, yazılacak olan testler için en uygun yöntem sadece "public" tipindeki metotların test edilmesi, "protected" ve "private" tipindeki metotların ise dolaylı olarak public metotlar üzerinden test edilerek kapsanmasıdır. Ancak "protected" veya "private" metotların karmaşık algoritmalar içerdiği bazı özel durumlarda bu metodlar için ayrı gereksinimler ve testler yazılabilir. Nesne yönelimli olmayan yazılımlarda ise birimin diğer birimlere sağladığı fonksiyonlar/servisler üzerine yoğunlaşmak, birimin kendi içinde gerçekleştirdiği işlemleri mümkün olduğunca servis fonksiyonları üzerinden test etmek gerekmektedir. Metodları tanımlayan gereksinimlerin aşağıdaki bilgileri sağlaması çoğu zaman yeterlidir:

- Tanım
- Ön Kosullar
- Parametreler
- Çıktı (Return)

Yazılım Birim Testlerinin (YBT) yapıları itibariyle, test edilen birimin bağlı olduğu diğer birimlerden bağımsız olarak test edilmesi gerekir. Bunu gerçekleştirebilmek için bağlı olunan diğer birimlerin koçanlanması gereklidir. Bağımlı olunan birimin de başka birimlere bağımlı olabileceği düşünüldüğünde koçanlama işleminin en düşük seviyede tutulması sonraki test güncelleme işgücünün azaltılması için büyük önem taşımaktadır. Bunun yanı sıra bir yazılım dosyası içinde yer alan metot/fonksiyonlar arasında sadece istenilen metodun koçanlanması ve koçanın yer aldığı dosyayı kopyalamadan kullanım sağlaması yöntemleri de kullanılmaktadır. Bu yöntemler sayesinde test edilen birim, kütüphane olarak test içinden çağrılabilmektedir. Test dinamik olarak gerekli yerlerde akışı koçanlara yönlendirmektedir. Bu yöntemin en önemli getirilerinden biri kullandıkları koçanlar farklı olsa da farklı testleri bir arada derleyip koşturabilme imkanıdır. Geleneksel yöntemlerde her bir test kullandıkları

koçanlar farklı olduğu için ayrı ayrı derlenmekte ve bunun sonucu olarak her bir test ayrı yüklenip koşturulmaktadır [55].

Yazılımlarda bulunan tüm kod parçalarının tamamiyle test edilmesi gereklidir. Bu da yazılacak olan testlerin sayısının normal projelere göre çok daha fazla olmasını ve daha sık güncellenmesini beraberinde getirir. Bu nedenle bu testlerin otomatik olarak koşması büyük önem arzetmektedir. Yazılım Gelistirme süreçleri gereğince, "Yazılım Yasam Döngüsü" içerisinde geliştirilen her ürün ilk kez yayımlandığında kontrol altına alınır. Bu ürünlerde sonradan yapılacak olan her değişiklik kayıt altında olmak zorundadır ve yapılacak olan değişikliğin doğru bir şekilde yapıldığının bağımsız kişilerce doğrulanması gereklidir. Bu nedenle herhangi bir ürün yeterince olgunlaşmadan yayımlandığında hatalar çok çıkmakta ve bu da sonraki fazlarda üretilen ürünlerin de çok sık güncellenmesini beraberinde getirmektedir. Örneğin kod veya gereksinim yeterli olgunluk seviyesine ulaşmadan yayımlandığında birim testlerinin bulunan kod ve gereksinim hatalarının sonrasında güncellenmesi, çok sık olamamakla beraber bazen de baştan yazılmasını gerektirebilmektedir. Kod ve gereksinimler için teknik bir gözden geçirme yapılarak hataların büyük bir kısmı önceki aşamalarda giderilebilir. Yukarıda anlatılan nedenlerden dolayı, YBT'leri geliştirmeye başlamadan önce kodun ve gereksinimlerin yapılacak gözden geçirmelerle belirli bir olgunluğa gelmesini beklemek gereklidir. Çoğu projede, birim testleri geliştirmek için ayrılan süre, kod geliştirme sürecinin uzaması nedeniyle yetersiz kalmaktadır. Bu nedenle bu fazda yapılan aktiviteleri olabildiğince hızlandırmak büyük önem taşır. Buna ek olarak yazılım geliştiricilere kısa zamanda geri besleme yapabilmek için testlerin hızlı güncellenip koşturulması gerekmektedir [55].

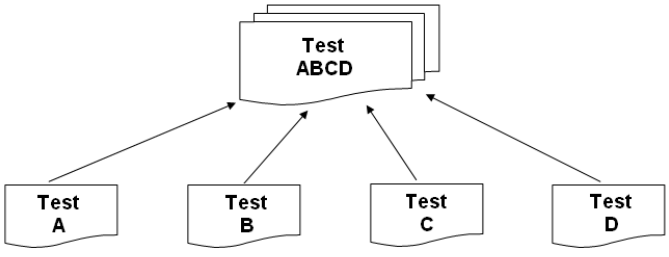
YBT'lerinin temel amacı daha önceden belirtildiği gibi o birime ayrılmış olan fonksiyonalitenin test edilmesidir. Nesne Yönelimli C++ gibi programlama dillerinde her class'ın sorumlu olduğu ve diğer class'lara sunduğu metotlar "public" olarak tanımlandığı için, sadece "public" metotların test edilmesi çoğu zaman yeterlidir. "protected" veya "private" metotlar karmaşık algoritmik hesaplamaları içermediği sürece public metotlar üzerinden dolaylı olarak test edilebilir. Sadece basit verileri

içeren ve tek görevi bu verileri depolamak ve geri döndürmek olan class'ların test edilmesi önemli bir bulgu içermediği için herhangi bir yarar sağlamamaktadır. Ayrıca hemen hemen her class içerisinde yer alan, class içerisindeki değişkenlere değer atanmasını ve geri döndürülmesini (Getter/Setter) sağlayan metodlar da test edilmeyerek YBT'leri geliştirmek için gerekli olan işgücünden tasarruf sağlanabilir. Bunlara ilaveten yazılım birim testlerinde bulunan hata tiplerini; kod hataları, gereksinim hataları ve test hataları olarak özetlemek de mümkündür [55].

Tümleştirme (entegrasyon) testi

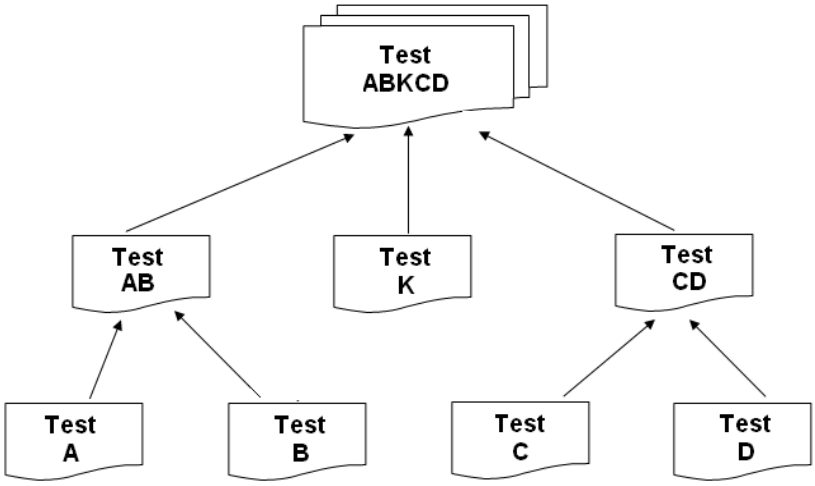
Yazılım projelerinde birim testlerin başarılı bir şekilde sonuçlandırılmasından sonra tümleştirme (entegrasyon) testleri başlar. Birim testlerde amaç modüllerin ayrı ayrı kendilerinden beklenen işlevleri yerine getirdiğinin doğrulanması iken, tümleştirme testlerinin amacı bir araya gelerek entegre edilmiş olan bir yazılımın içerisindeki bileşenlerin birbirleriyle uyum içerisinde doğru bir şekilde çalıştığının ve bileşenlerin kendilerine ait gereksinimleri yerine getirdiğinin doğrulanmasıdır. Tümleştirme testlerinde kara kutu test yöntemi kullanılır. Tümleştirme testleri gerçekleştirilirken big bang, aşağıdan yukarıya veya yukarıdan aşağıya olmak üzere farklı stratejiler kullanılabilir [1, 56].

Big bang stratejisine göre; geliştirilen tüm üst ve alt düzey modüller birbirleriyle entegre edildikten sonra testler başlar. Bu stratejide sistem bir bütün olarak ele alınarak testler icra edildiğinden testlerin gerçekleştirilmesi hızlıdır ve zamandan tasarruf edilir. Ancak bir hata çıkması durumunda bu hatanın hangi seviye katmanında oluştuğu veya hangi modülden kaynaklandığının tespitinde zorluklar yaşanabilir. Şekil 2.12'de big bang test yaklaşımı örneklendirilmiştir [1, 56].



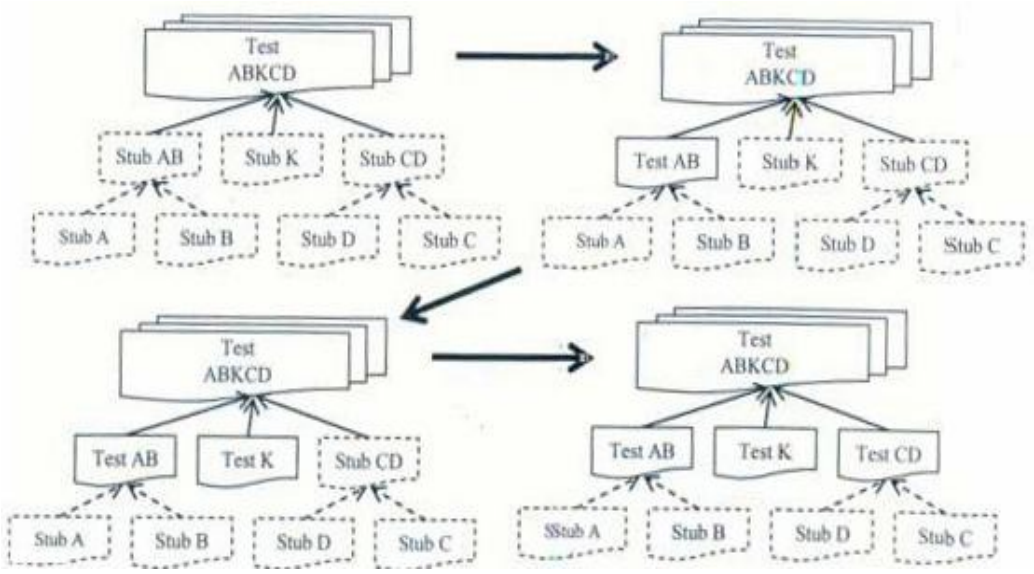
Şekil 2.12. Big bang test yaklaşımı örneği [1]

Aşağıdan yukarıya tümleştirme test stratejisi, öncelikle birim testlerin yapılmasıyla başlar. Birim testleri başarıyla tamamlanan alt düzey modüller birbirleriyle entegre edilir. Bu entegrasyondan sonra alt düzey modüller ile ilgili entegrasyon testleri icra edilir. Bu testlerin başarılı bir şekilde sonlandırılmasından sonra bir üst katman modülleri sisteme entegre edilir ve gerekli entegrasyon testleri icra edilir. Bu entegrasyon tüm sistem inşa edilinceye ve entegrasyon testleri başarıyla sonuçlanıncaya kadar devam eder. Bu yaklaşımda temel şart, entegre edilecek modüllerin birim testlerinin başarıyla tamamlanmış olmasıdır. Şekil 2.13'te aşağıdan yukarıya test yaklaşımı örneklendirilmiştir [1].



Şekil 2.13. Aşağıdan yukarıya test yaklaşımı örneği [1]

Yukarıdan aşağıya tümleştirme test stratejisi, öncelikle sistem için en üst modülün (en dış modül, genellikle kullanıcı grafik arayüz modülü) test edilmesi ile başlar. Bu modülün ihtiyaç duyduğu alt modüllerin koçanları kullanılır. Bu yaklaşımda birçok koçan yazılması gerekir. En üst modül başarılı test edildikten sonra bir alt düzey modüller sistemle bütünleştirilir. Bu entegrasyondan sonra gerekli alt düzey tümleştirme testleri icra edilir. Bu durum en alt düzey modüller entegre edilinceye ve entegrasyon testleri başarıyla sonuçlanıncaya kadar devam eder. Bu yaklaşımda entegrasyon testleri kara kutu testleri olarak başlar ve gittikçe beyaz kutu testlerine döner. Şekil 2.14'te yukarıdan aşağıya test yaklaşımı örneklendirilmiştir [1].



Şekil 2.14. Yukarıdan aşağıya test yaklaşımı örneği [1]

Küçük ölçekli projeler genellikle, tek bir seferde tüm modüllerin entegre edilmesi ile test edilebilirken büyük ölçekli projelerde bu yöntem tercih edilmemektedir [22]. Büyük ölçekli yazılım projelerinde, entegrasyon testlerinin tek seferde değil aşamalı olarak yapılması gerekir. Yani, öncelikle modüllerden birkaçı ile küçük bir sistem kurularak daha sonra bu sistemlerin birleştirilmesi yöntemi, özellikle görülen hataların analizini kolaylaştıracağından iyi bir tercih olabilir [57].

Tümleştirme testleri, arayüz ve etkileşimle entegre edilmiş bileşenler arasındaki hataları ortaya çıkarır. Bu süreç, test edilen yazılım bileşenleri bir sistem gibi çalışana kadar sürekli olarak devam eder [24]. Tümleştirme testleri sonucunda herhangi bir hataya rastlanırsa, bu hatalar düzeltildikten sonra yeni kodlanan yazılım, yeni bir sürüm olarak tekrar test işlemine tabi tutulur. Yapılan değişikliklerin yazılımdaki etkileri görülmesi için de "yineleme testleri"nin yapılması gerekir. Tümleştirme testleri icra edilirken "yazılım tümleştirme test planı" kullanılmalıdır. Tümleştirme testi için geleneksel yaklaşımlar; yukarı-aşağı, alt-üst ve üst üste koyma yaklaşımlarını içerir. Tümleştirme testi, dikkatli planlamayı gerektirir. Kalıtım testi, sınıf bir diğerinden kalıtımı alır almaz başlar. Tümleştirme stratejileri, sınıf seviyesinde, bileşen seviyesinde, alt sistem seviyesinde ve sistem seviyesinde gereklidir. Sınıflar, metotların entegrasyonunu gerektirir. Çeşitli sınıf tipleri, farklı

test stratejilerini gerektirir. Tümleştirme testi, geliştirilen iki veya daha fazla bileşenin fonksiyonlarını çalıştırmak için birleştirildiğinde dahili ve harici hatalar, zamanlama hataları ve ürün hatalarını içeren herhangi bir hatayı bulmak için uygulanır. Nesne yönelimli sistemde, tek sınıf içerisindeki üyelerin entegrasyonu, baştan sona kalıtımlı iki veya daha fazla sınıfın entegrasyonu, birbirini içeren iki veya daha fazla sınıfın entegrasyonu, bir bileşen biçimi için iki veya daha fazla sınıfın entegrasyonu ve tek bir uygulamadaki bileşenlerin entegrasyonu gibi entegrasyonlar olabilir. Tümleştirme testinin birincil amacı, bir sistemin istenen fonksiyonelliğini başarmak için geliştirilen birim veya bileşenlerin doğru bir şekilde etkileşimini sağlamaktır. Her bir bileşen doğrulansın diye ilk adım, sistemin iskeletinin mümkün olduğu kadar erken oluşturulmasını sağlamaktır [4]. Literatürde birkaç tümleştirme test stratejileri mevcuttur.

Çalıştırma temelli tümleştirme testi

Çalışmaları izlenerek birimlerin hatalı etkileşimlerini ortaya çıkarmak için kullanılır.

Değer temelli tümleştirme testi

Belirli değerleri çalıştırarak birimlerin etkileşimlerini yerine getirmek için kullanılır.

Fonksiyon temelli tümleştirme test

Bileşenler etkileşirken onların fonksiyonelliğini değerlendirmek için kullanılır [4].

Sistem testi

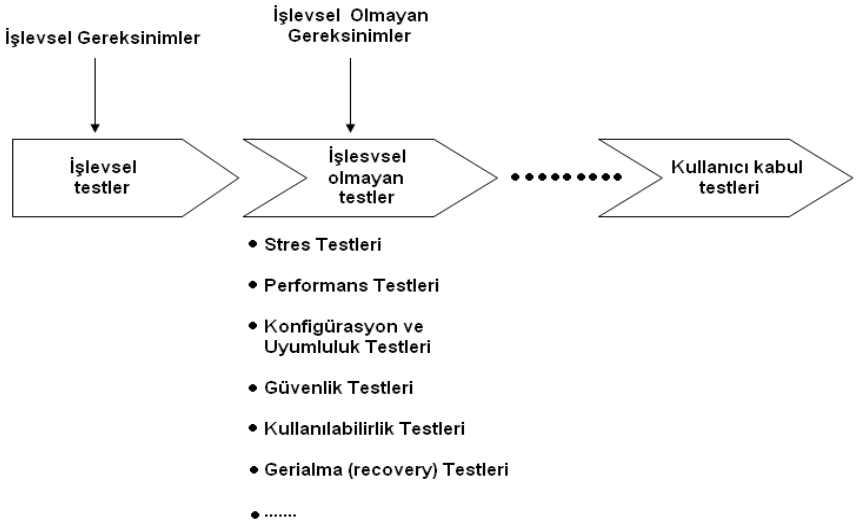
Geliştirilen yazılımın performans, güvenirlik, işlevsellik gibi özelliklerini değerlendiren testlerdir. Birim/sınıf testlerde ve tümleştirme (entegrasyon) testlerinde geliştirilen yazılımın tasarıma uygun olarak geliştirildiği doğrulanır. Sistem testleri ise, müşterinin sistemden istediklerini doğrulamayı amaçlar. Bu nedenle sistem test durumlarında sistem gereksinimleri temel alınarak, sistemin çalışacağı, gerçek ortamda karşılaşılabilecek olan senaryolar tanımlanır [1].

Sistem testleri, gereksinimlerin karşılandığını doğrulamak için entegre edilmiş sistemi tamamen test eder. Bir yandan, sistem performans, güvenlik, dayanıklılık veya harici sistemlerle etkileşim gibi fonksiyonel olmayan gereksinimlere karşı

doğrulanabilir. Diğer yandan, sistem tarafından uygulanan fonksiyonalite özellikleriyle karşılaştırılabilir [24].

Birleştirilmiş sistem testi, yazılım test yaşam döngüsünün temel evresidir. Buna ilaveten, hatalar ilk başlarda tespit edilirse test etmek için gerekli çaba azalır. Birleştirilmiş sistem testi; sağlamlık testi, dökümantasyon testi, performans testi, stres testi ve sınır testini içerir. Sistem testi, yazılım geliştirme yaşam döngüsünün son evreleri esnasında uygulanır. Sistem testinin, hem birleştirilmiş sistem testini hem de kabul testini içerdiği düşünülebilir. Sistem testinin temel amaçlarından biri, yazılım sisteminin belirli bir zaman diliminde davranışları gözlemleyerek kullanıcılar için hazır olup olmadığını kararlaştırmaktır. O, gereksinimlerin uygunluğu için entegrasyon testini tam bir şekilde test eder. Sistem testi, bir sistemin fonksiyonel, performans, stres ve kaynak gereksinimlerin tümüne uygulanmalıdır [4].

Sistem testleri, kullanıcı kabul testlerinden bir önceki adım olarak yazılım ve donanım entegrasyonundan sonra "sistem test planlama"ya göre gerçekleştirilir. Sistemin işlevsel, işlevsel olmayan gereksinimlerinin doğrulanması hedeflenir[1, 56]. Şekil 2.15'te sistem testi adımları görülmektedir.



Şekil 2.15. Sistem testinin adımları [1]

Sistem testlerinin ilk adımı olarak işlevsel gereksinimlerin doğrulanması gerçekleştirilir. Bu adımdan bir sonraki adım ise işlevsel olmayan gereksinimlerin karşılandığını göstermek amacıyla işlevsel olmayan testler gerçekleştirilir. Bu testlerden bazıları şunlardır.

Stres testi (Stres testing)

Sisteme girdi oranı sistem tasarım oranını aştığı zaman sistemin davranışını gözlemlemek üzere gerçekleştirilen testlerdir.

Performans testi (Performance testing)

Sistem çıktılarının belirlenen ve kabul edilebilecek olan zaman dilimi içerisinde üretebildiğinin değerlendirilmesinin yapılabilmesi için gerçekleştirilen testlerdir [1]. Gerçek zamanlı sistemlerde yazılım işlem süresinin bilgisayara dayalı sistem ile uyarlığının sınanmasıdır. Performans sınanması, her test basamağında uygulanmaktadır. Ancak, sistemin performansı tam olarak sistemin bütünleştirilmesinden sonra anlaşılabilmektedir [45].

Konfigürasyon ve uyumluluk testleri (Configuration and compatibility testing)

Geliştirilen sistemin farklı platformlarda ve donanımlarda nasıl davrandığının değerlendirilmesi için gerçekleştirilen testlerdir.

Güvenlik testi (Security testing)

Sistemin izinsiz kullanım teşebbüslerindeki davranışlarının değerlendirilmesi için gerçekleştirilen testlerdir [1]. Sistemin zararlı dış müdahalelerden ve bilgi hırsızlığından korunabildiğinin kanıtlanmasıdır [45].

Kullanılabilirlik testi (Usability testing)

Kullanıcı-sistem etkileşimini ve ergonomisini değerlendirmek üzere gerçekleştirilen testlerdir.

Geri alma testi (Recovery testing)

Bir hata durumunda sistemin otomatik veya elle yeniden normal duruma dönmesini değerlendirmek için gerçekleştirilen testlerdir.

Kullanıcı arayüzü testi (User interface testing)

Kullanıcının ve yazılımın grafik gösterimi olarak nasıl bir etkileşim içerisinde olacağını, kullanıcının klavye, ekran veya fare ile sisteme vereceği girdilerin sistem tarafından nasıl işleneceğini değerlendirmek için gerçekleştirilen testlerdir. Sistem testleri sırasında ortaya çıkan hatalar proje hata yönetim sürecine göre raporlanır ve gerekli düzeltme işlemleri gerçekleştirilir. Gerekli düzeltmelerden sonra, düzeltmelerden sistemin kalanının etkilenmediğinin değerlendirilmesi için sistem üzerinde yineleme testleri gerçekleştirilir [1].

Yükleme testleri (Load testing)

Aynı anda maksimum sayıda sanal kullanıcı tarafından yapılan testlerdir.

Kabul testi

Proje kabul testleri müşteri gereksinimlerinin doğrulanması ile gerçekleştirilir. Bunun için her bir kullanıcı gereksinimini doğrulayacak olan test durumları ve test senaryoları oluşturulur. Geliştirilen sistem, tanımlanan bu test durumları ve test senaryoları müşterininde katılımı ile "kabul test planına" uygun olarak koşturulur. Sistemin tamamının bu testlerden geçmesi ile müşteri tarafından kabul edilmiş olur. Kabul testi literatürde, kullanıcı ile birlikte gerçekleştirilen testler olduğu için aynı zamanda kullanıcı kabul testi olarak da isimlendirilir [1, 58].

Kabul testleri ile, müşteri gereksinimlerinin kapsandığı doğrulanarak, yazılımın bu gereksinimleri ne oranda karşıladığı ölçülür. Daha önceki test eylemlerinde tespit edilemeyen problemlerin tespiti yapılabilir. Geliştirlen sistemin müşteri gereksinimlerini nasıl karşıladığı gösterilir [1, 59].

Kabul testleri elle yapılabileceği gibi sistemin durumuna göre otomatik olarak da gerçekleştirilebilir. Kabul testleri sistemin başarısını doğrudan gösterdiğinden gerçekleştirilmesine ayrı bir önem verilmelidir. Kabul testlerinde karşılaşılan en önemli soru kabul test durumu ve senaryolarının kimin tarafından yazılacağıdır. Kabul testlerinin yazılması doğrudan müşteri tarafından olabileceği gibi, projenin test ekibi tarafından da yazılabilir. Eğer test durumları ve senaryoları test ekibi tarafından yazılmışlar ise testler başlamadan mutlaka müşteri onayı alınmalıdır. Bazı projelerde ise ortak bir ekip kurularak birlikte geliştirlir [1, 60] .

Kabul test durumları ve senaryoları müşteri gereksinimlerinden çıkartıldıklarından dolayı yazılım tamamıyla gerçekleştirilmeden yazılmış olmalıdırlar. Yazılan test durumları ve senaryoları ile müşteri gereksinimleri arasında izlenebilirlik kurulmalıdır. Bu izlenebilirlik ile müşteri gereksinimlerinin tamamı için test eyleminin gerçekleştirileceği garanti altına alınmış olur. Ancak bu durum, tüm gereksinimlerin başarıyla gerçeklendiği anlamına gelmez.

Kabul testleri bir yazılım projesinin başarısında en kritik adımdır. İyi planlanmış ve belgelendirilmiş kabul testleri projenin başarısına büyük katkı sağlar ve müşteri memnuniyetini arttırır. Geliştirilen yazılımlar üzerinde son olarak uygulanan kabul testleri ile müşteri gereksinimlerinin doğrulanması gerçekleştirilir [1]. Kabul testlerinde, alfa ve beta testleri uygulanır.

Alfa testi

Programın beta testi kullanıcısına verilmeden önceki son test aşamadır ve alfa testi olarak adlandırılır. Program bu aşamada gerçeğe yakın koşullarda, simüle edilen test dataları ile test edilir. Alfa testi, yazılım ekibi içerisinde yer alan yazılım test uzmanları, iş analistleri, sistem analistleri ve yazılım uzmanları tarafından gerçekleştirilir. Bu testin amacı müşterinin kullanıcı kabul testleri sırasında çok fazla hata, eksiklik, tutarsızlık ve performans düşüklüğü ile karşılaşmasını engellemek ve kullanıcıların geliştirilen sistemi benimsemesini kolaylaştırmaktır [61].

Beta testi

Beta testi, ileride yazılımı satın alacak olası müşterilere, yazılan programın bir kopyasının ücretsiz olarak dağıtılıp programı belirli bir süre için ya da süresiz olarak kullanmalarını sağlamaktır. Bu kişiler programı gerçek çalışma koşullarında sınayacak ve gördükleri hataları geliştirici (developer) grubuna raporlayacaktır. Raporlanan bu hatalar(bugs) düzeltilecek ve yazılım ürünü bu aşamalardan sonra satışa sunulacaktır. Beta testi, geliştirilen sistemi kullanacak kişiler tarafından gerçekleştirilen, geliştirilen sistemin tanımlanan gereksinimleri karşıladığını teyit ettikleri test çalışmasıdır. Kullanıcı kabul testi veya müşteri kabul testi olarak da bilinir [61].

Yazılımı geliştiren ekip, geliştirdikleri yazılımın, tüm fonksiyonel ve fonksiyonel olmayan gereksinimlerin karşılandığını iddia ettiği zaman beta testine geçilebilir. Beta testi başlatılmadan önce, geliştirilen sistemi kullanacak kullanıcılar arasından heterojen bir grup oluşturulur ve sistemi tanımlı gereksinimlere göre belirli süre içerisinde sınaması istenir. Bu çalışmaya başlanmadan önce kullanıcıların geliştirilen sistem konusunda eğitilmeleri gerekir. Kullanıcılar eğitimi aldıktan sonra yapacakları sınama işlemini planlar ve gerekli test verilerini ve senaryolarını oluştururlar. Kabul testi dahilinde sistemin fonksiyonları test edildiği kadar, sistemin fonksiyonel olmayan gereksinimleri de karşılayıp karşılamadığı kontrol edilir [61].

2.11.3. Test tasarım teknikleri

Yazılım testlerinde analiz sonuçlarına bakarak uygun bir test tekniği belirlenmelidir. Her yazılımın yapısına veya büyüklüğüne göre yazılım test tasarım teknikleri değişiklik göstermektedir. Tasarım teknikleri her şeyden önce dinamik ve statik olmak üzere ikiye ayrılır. Bu iki bileşende kendi içerisinde maddelere ayrılır. Çizelge 2.1'de bahsedilen test teknikleri gösterilmektedir.

Çizelge 2.1. Test tasarım teknikleri

<table>
<tr><td rowspan="7">Dinamik Test</td><td rowspan="5">Fonksiyonel Test (Kara Kutu Testi)</td><td>Denklik Sınıfı Test Tekniği
(Equivalance Class Testing Technique)</td></tr>
<tr><td>Uç Nokta Değer Analizi Test Tekniği
(Boundary Value Analysis Testing Technique)</td></tr>
<tr><td>Karar Tablosuna Dayalı Test Tekniği
(Decision Table Based Testing Technique)</td></tr>
<tr><td>Sistem Durumu Test Tekniği
(State Transition Testing Technique)</td></tr>
<tr><td>İş Senaryosu Test Tekniği
(Use Case Testing Technique)</td></tr>
<tr><td rowspan="2">Yapısal Test (Beyaz Kutu Testi)</td><td>Komut Testi (Statement Testing)</td></tr>
<tr><td>Karar Testi (Decision Testing)</td></tr>
<tr><td colspan="2">Hem Fonksiyonel Hem Yapısal Test Tekniği
(Gri Kutu Testi - Gray Box Testing)</td></tr>
<tr><td colspan="2">Tecrübeye Dayalı Test Tekniği (Experience Based Testing Technique)</td></tr>
<tr><td rowspan="4">Statik Test</td><td colspan="2">Gözden Geçirmeler (Reviews) / Kod Geçişleri (Walkthroughs)</td></tr>
<tr><td colspan="2">Kontrol Akış Analizi (Control Flow Analysis)</td></tr>
<tr><td colspan="2">Veri Akış Analizi (Data Flow Analysis)</td></tr>
<tr><td colspan="2">İnceleme (Inspection)</td></tr>
</table>

Dinamik test

Geliştirilen yazılımın dinamik davranışının gözlemlenmesi için gerçekleştirilen testlerdir. Dinamik testler kapsamında sistemin değişen veriler karşısında nasıl tepki verdiği gözlemlenir [1]. Dinamik testler, fonksiyonel testlerdir. Kara kutu testi, beyaz

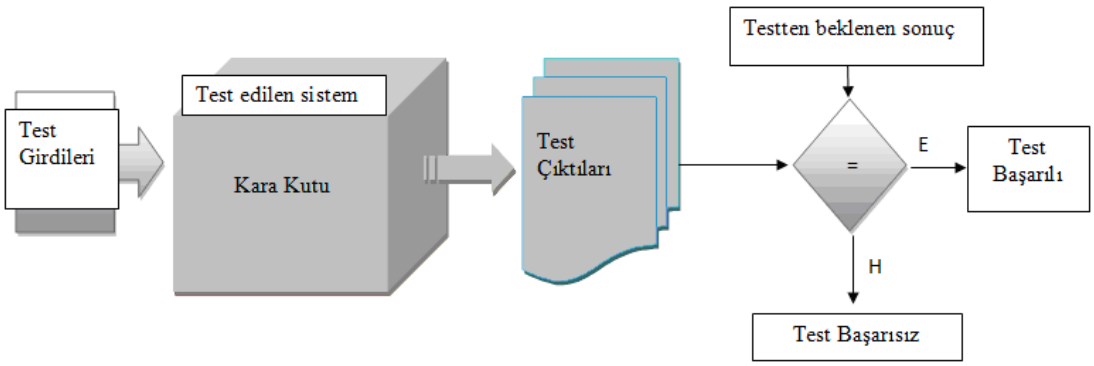
kutu testi, gri kutu testi ve tecrübeye dayalı test tekniği olmak üzere kendi içerisinde maddelere ayrılır. Aşağıda bu tekniklerin tümü kısaca açıklanmaktadır.

Fonksiyonel test (Kara kutu testi)

Fonksiyonel testler, bir programın işlevselliği üzerine temellenir. Bu yüzden, fonksiyonel testler, belirli girdi değerlerine göre sadece çıktıları gözlemleyen testleri sağlar. Bir çıktı üreten kodun analizini yapmaz. Kodun iç yapısı görmezden gelinir veya yok sayılır. Dolayısıyla fonksiyonel testler, içinde ne olduğu bilinmeyen kara kutu olarak bilindiği için bu testlere kara kutu testleri de denilmektedir.

Kara kutu testleri, test edilecek olan yazılımın iç işleyişi düşünülmeden gerçekleştirilen testlerdir. Bu yaklaşımda test edilecek olan yazılımın, sonucu bilinen bir davranışını doğrulamak için davranışın gerektirdiği girdi/giriş değerleriyle çalıştırılarak sınanır. Daha sonra yazılımın bu girdi karşısında elde edilen çıktısı beklenen sonuçla karşılaştırılır. Bu yaklaşımın kara kutu olarak adlandırılmasının nedeni, testçinin testi uygularken yazılımın iç yapısı ile kesinlikle ilgilenmemesidir. Bu nedenle bu testler yazılım geliştiriciden çok test ekipleri tarafından uygulanır. Kara kutu testinde bir yazılım parçası test ediliyor ise test mühendisi bu yazılım parçasının girdisini ve buna karşılık sistem çıktısını bilir. Fakat bu çıktıya nasıl ulaşıldığıyla ilgilenmez. Çünkü kara kutu testlerinin amacı verilen girdiler ile istenilen çıktının elde edilmesidir. Bu nedenle yazılımın iç işleyişi ile ilgili bilgilerle ilgilenilmez. Şekil 2.16'da durum, temsili olarak gösterilmiştir.

Kara kutu test tekniğinde diğer önemli bir nokta da geliştirilen yazılımın tasarımından veya kodlamasından kaynaklanabilecek hataların bulunmasıdır. Bu amaçla kara kutu testlerinin gerçekleştirilmesi ve istenilen amaca ulaşılabilmesi için yazılım geliştiriciler ile test ekibi birbirinden ayrı olarak çalışmalıdır. Böylece test ekibi tasarım ayrıntılarını bilerek yazılıma karşı ön yargı taşımazlar [1, 11].



Şekil 2.16. Kara kutu test yaklaşımı [1]

Kara kutu test tekniği ile geliştirilen yazılım içerisinde aşağıdaki hata türleri tespit edilmeye çalışılır [1, 62].

1. Doğru olmayan veya hiç olmayan işlevlerin tespiti
2. Arayüz hataları
3. Performans hataları
4. Veri tabanlarına ulaşma hataları veya veri yapılarındaki hatalar
5. İlklendirme veya sonlandırma hataları
6. Sınır değer hataları

- Kara kutu testinin avantajları

Kara kutu testlerinin avantajları şunlardır [1, 9, 10]:

- Yazılımlarda hataların bulunması için etkin ve hızlı bir tekniktir.
- Test durumları yazılırken gereksinimlerden hareket edildiği için gereksinimlerdeki tutarsızlıkların ve belirsizliklerin tespit edilmesinde önemlidir.
- Testçilerin yazılım ayrıntılarını bilmelerine gerek yoktur.
- Testçiler ve yazılımcılar birbirinden bağımsız olarak çalışabilirler.
- Testçiler gereksinimleri doğrulamak ve gereken testleri gerçekleştirmek için yazılıma kullanıcı gözü ile bakarlar [1,10]. Bu da yazılımcılar tarafından fark edilemeyen pek çok olası hatanın ve eksiğin bulunmasına yardımcı olur.
- Testleri gerçekleştirecek kişilerin sistem hakkında teknik ayrıntı bilmesine gerek yoktur.

- Kara kutu testinin dezavantajları

- Kara kutu testleri yazılımın belirli parçasını hedeflemez. Bu nedenle birçok hata tespit edilmeden kalabilir.
- Sadece belirli sayıda girdi/giriş değeriyle testler gerçekleştirilir. Tüm girdiler ile testlerin gerçekleştirilmesi nerede ise sonsuza kadar sürebilir [1,9].
- Yazılım içerisinde bazı kod parçalarında birden fazla test gerçekleşir iken bazı kod parçaları hiç test edilmeden kalabilir [1,10].
- Açık ve yalın olmayan gereksinimlerin test durumlarını tasarlamak ve testlerini gerçekleştirmek kara kutu testlerinde oldukça zordur.

- Kara kutu test stratejisi

Kara kutu test tekniği tümleştirme, sistem ve kabul test aşamalarında kullanılır. Bu testlerde en etkin sonuçlara ulaşılabilmesi için aşağıdaki kara kutu test stratejisinin izlenmesinde yarar vardır [1, 63, 64]:

- Kara kutu testleri rastgele belirlenmiş girdilerle gerçekleştirilmeli, testçiler sadece girdi aralığını belirtmelidir.
- Yazılımın sağlamlığının kontrolü için belirtilen aralığın dışındaki değerlerin de testi gerçekleştirilmelidir.
- Sınır değerler mutlaka test edilmeli, en yüksek ve en düşük değerlerin beklenen çıktıyı verdiği mutlaka doğrulanmalıdır.
- Değer artışlarında artış miktarı ayrıca test edilerek doğrulanmalıdır. Artış miktarı dışı artımlarda yazılımın tanımlı davranışına göre hareket ettiği doğrulanmalıdır.
- Sayısal girdilerde sıfır (0) mutlaka girdi olarak sınanmalıdır.
- Özellikle gerçek zamanlı sistemlerde stres testi yapılmalıdır. Programın aşırı yüklenme (en yüksek kapasite) altında nasıl çalıştığı test edilmelidir.
- Emniyet kritik, görev kritik gibi yazılımlar için yazılımın hatası durumunda nasıl davrandığı test edilmelidir.bu amaç için kritik yazılımların kara kutu testlerinde test izleme yazılımlarından yararlanılabilir.

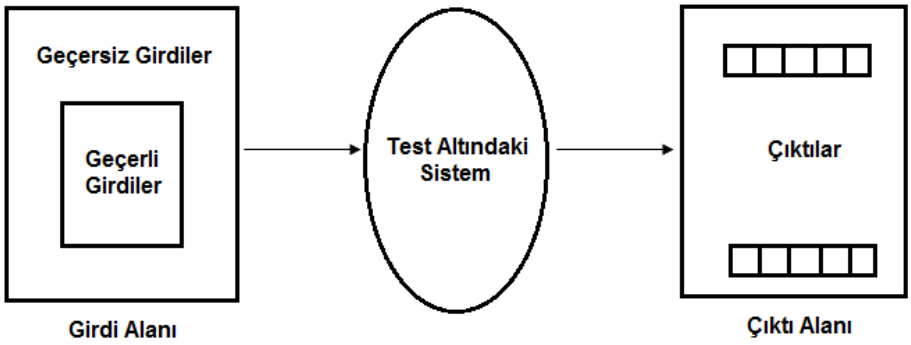
- Kara kutu testlerinde diğer bir amaç gereksinimlerin doğrulanması olduğundan her bir gereksinim için en az bir test durumu yazılmalı ve bu şekilde gereksinim kapsama gerçekleştirilmelidir.

Test edilecek yazılımların özelliklerine göre kara kutu test strateji adım sayısında değişiklik vardır. Stratejinin başarılı bir şekilde uygulanması deneyimli bir test grubuna bağlıdır [1].

Kara kutu test tekniği, projeye bağlı olarak değişebilen aşağıdaki alt tekniklere sahiptir.

Denklik sınıfı test tekniği (Equivalance class testing technique)

Bu metotta, bir programın girdi alanı kabul edilebilen denk sınıfları sınırlı sayıda parçalara ayrılır. Daha sonra her bir sınıfın temsili değerinin testi, başka herhangi bir sınıfı test etmek için denk kabul edilir. Yani, eğer bir test durumu, bir sınıftaki bir hatayı tespit ediyorsa o sınıftaki diğer tüm test durumlarının benzer hataları bulması beklenir.



Şekil 2.17. Denklik sınıf bölümleme [26]

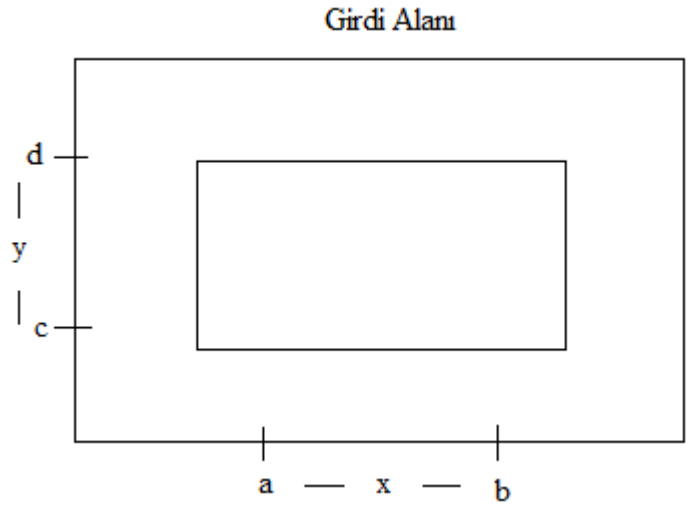
Bu metodu uygulamak için iki adım gereklidir:

1. Denklik sınıfları, her bir girdi durumları elde edilerek ve geçerli yada geçersiz sınıflar içerisinde bölünerek belirlenir. Örneğin, bir girdi durumu, 1-999 değer aralığında ise, geçerli denklik sınıfı [1<sayı<999]; geçersiz denklik sınıfı ise [sayı<1] ve [sayı>999] olarak tanımlanır.

2. Önceki adımda tanımlanan denklik sınıflarını kullanarak test durumları üretilir. Bu, tüm denklik sınıflarını kapsamak şartıyla test durumları yazarak uygulanır. Sonra, her bir geçersiz denklik sınıfı için test durumu yazılır [26].

Uç nokta değer analizi test tekniği (Boundary value analysis testing technique)

Daha önceki tecrübeler, sınır/uç noktalara yakın olan test durumları, bir hatayı tespit etme de daha yüksek şansa sahip olduğunu gösterir. Buradaki uç noktada olma durumu, bir girdi değerinin Şekil 2.18'de gösterildiği gibi sınır üzerinde olabilmesi anlamındadır [26].



Şekil 2.18. x ve y olmak üzere iki girdi değerine sahip program için girdi alanı [26]

Özetle, hataların en sık rastlandığı alanlar uç noktalardır. Uç noktalar minimum ve maksimum değerlerin alındığı durumlardır. Denklik sınıfları uç nokta değerleri denklik gruplarından birini oluşturur.

Karar tablosuna dayalı test tekniği (Decision table based testing technique)

Karar tabloları, 1960'ların başlarından beri karmaşık mantıksal ilişkileri analiz etmek ve göstermek için kullanılır. Karar tabloları, testin nasıl uygulanabileceğini ve test edicinin kuralları nasıl kabul edeceğini göstermek için kullanılır [26]. Karar tablolarında sebep ve sonuç ilişkisi vardır. Test edilmesi gereken karmaşık

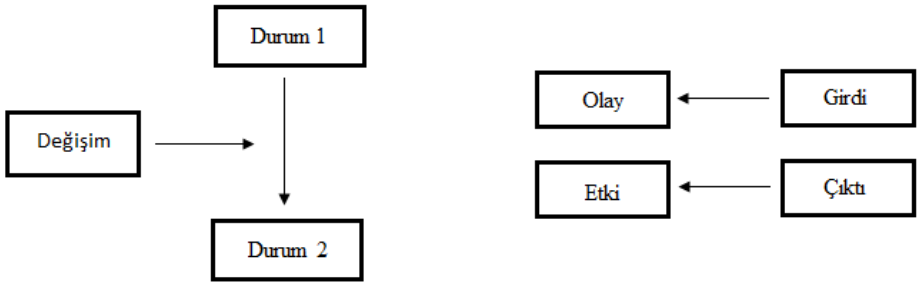
senaryoların, iş kurallarının ve karmaşık gereksinimlerin olduğu durumlarda uygulanan test tekniğidir [65]. Bir kararın dört parçası vardır. Durum parçası, eylem parçası, durum girdileri ve eylem girdileri. Örneğin, c_1 ve c_2 durumlarının tümü doğru olduğunda a_1 ve a_2 oluşur. Çizelge 2.2'de örnek bir karar tablosu gösterilmektedir [26].

Çizelge 2.2. Örnek bir karar tablosu [26]

Koşul Koçanı	c_1	Girdiler						
	c_2	Doğru				Yanlış		
	c_3	Doğru		Yanlış		Doğru		Yanlış
	c_4	Doğru	Yanlış	Doğru	Yanlış	Doğru	Yanlış	-
Eylem Koçanı	a_1	X	X			X		
	a_2	X		X			X	
	a_3		X			X		
	a_4				X		X	X

Sistem durumu test tekniği (State transition testing technique)

Sistem durumundaki değişikliklerden oluşan hataların tespitine ilişkin test tekniğidir. En çok gömülü yazılımların olduğu sistemlerde kullanılır [65]. Örneğin, belirli girdilerin işleme sokulması sonucu belirli değişiklikler veya çıktılar elde edilir. Daha sonra sistemin bir önceki duruma göre nasıl değiştiği kayıt altına alınır. Test işlemlerinde, yazılımda gerçekleşen ile beklenen karşılaştırılır ve daha önce elde edilen kayıtlara göre değerlendirilir. Şekil 2.19'da sistem durumu test tekniğim yapısı gösterilmektedir.



Şekil 2.19. Sistem durumu test tekniğinin yapısı [65]

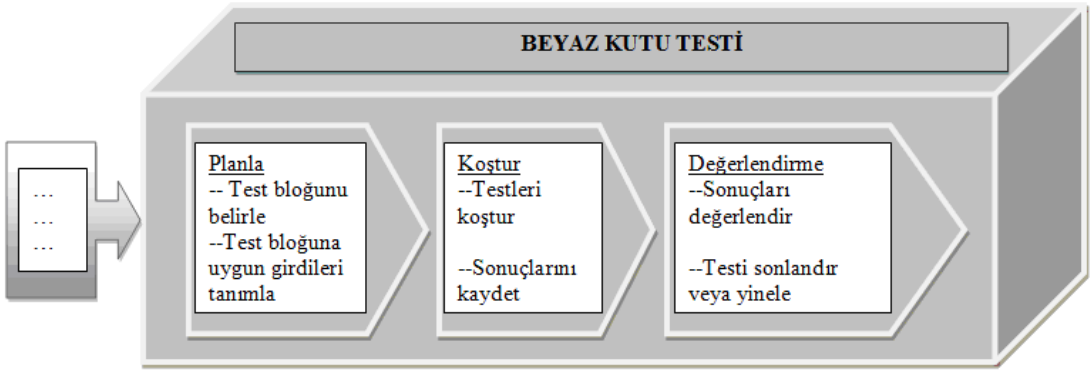
İş senaryosu test tekniği (use case testing technique)

İş senaryosu test tekniği, başlangıcından sonuna tüm sistemi test eden test durumlarını belirlemeye yardım eder. Test durumları, gerçek yaşam senaryoları çalıştırmak için tasarlanır. Onlar, entegresyon hatalarını çözmeye yardım eder [66].

Yapısal test (Beyaz kutu testi)

Fonksiyonel testleri tamamlayıcı bir yaklaşım olan yapısal testler, aynı zamanda beyaz kutu testleri olarak da bilinir. Bu testler, geliştirilen programın iç yapısını incelemeye imkan sağlar. Bu stratejide, programın mantığını sınayacak test durumları oluşturulur. Buna göre sözkonusu testler gerçekleştirilir.

Beyaz kutu testi diğer bir test tekniğidir. Beyaz kutu testleri yazılımın iç yapısı bilinerek tasarlanır. Bu nedenle beyaz kutu testlerini gerçekleştirenler genellikle sistemin iç yapısını bilen yazılımcılardır [11]. Beyaz kutu testiyle programın iç yapısındaki birimlerin içindeki hatalar araştırılır. Kaynak kod beyaz kutu testlerinin en önemli girdisi olduğundan koda ulaşım olmadan beyaz kutu testleri gerçekleştirilemez. Bunun yanında risk analizleri ve tasarım kısıtları da beyaz kutu testlerinin planlanmasında, test stratejisinin belirlenmesinde, test araçlarının seçilmesinde ve test verilerinin oluşturulmasında kullanılırlar [12]. Bu nedenle eğer sistem için risk analizleri tamamlanmamış ise beyaz kutu testinin ilk adımı olarak bu analizler gerçekleştirilir.



Şekil 2.20. Beyaz kutu test şeması [1]

- *Beyaz kutu testinin avantajları*

- Kod içerisinde gizli kalmış mantıksal hatalar bulunur.
- Beyaz kutu testleri ile yazılan kodun optimizasyonuna katkıda bulunulur.
- Kod içerisindeki fazla satırlar ayıklanarak ölü kod parçaları bulunur.
- Kaynak kodun analiz edilmesi ve bu analize göre testlerin gerçekleştirilmesi ile yazılım içerisindeki hatalar daha erken aşamada ve daha hızlı bulunması sağlanır.
- Yazılımın geliştirilmesi için belirlenmiş olan kodlama rehberine uyumluluk, tasarım kararlarına kodlama içerisinde uyulup uyulmadığı beyaz kutu testleri ile net olarak görülür.
- Beyaz kutu testleri ile yazılımcıların kod geliştirme yetenekleri desteklenir ve güçlendirilir.

- *Beyaz kutu testinin dezavantajları*

- Eğer birim tümleştirme testleri test ekibi tarafından yapılacaksa bu iş için kodun iç yapısını bilmesi gerekir. Bu da maliyeti arttırır.
- Beyaz kutu testleri ile sadece modül ve birimlerin iç işleyişleri test edildiğinden tümleştirme sonrasında ortaya çıkabilecek hatalar tespit edilemez [1].

Beyaz kutu testleri veri, kontrol ve bilgi akışlarının, kodlama standartlarının, hata yakalama ve ayıklama yapısının analizlerini içerir [1]. Beyaz kutu test yaklaşımı; tüm işlemlerden giderek komut testi (statement testing) ya da tüm yollardan giderek karar testi ile (decision testing) uygulanır.

Komut testi (Statement testing)

Komut test tekniği, belirli komutları çalıştırmak ve genelde komut kapsamını artırmak için test durumları oluşturur. Komut kapsamı, test durumlarının kapsadığı çalıştırılabilir komut sayısı olarak tanımlanır [65].

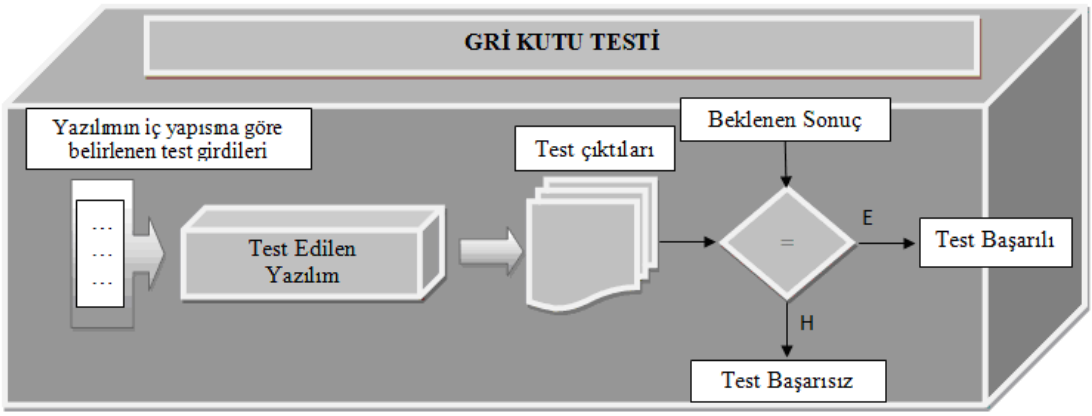
Karar testi (Decision testing)

Karar testi, tüm karar noktalarının akışını izlediği için kontrol akış testinin bir biçimidir. Karar kapsaması, komut kapsamasından daha güçlüdür. %100 karar kapsama durumu bazı şeyleri garantiler ancak tam tersi (%100 komut kapsama) doğru değildir [65].

Hem fonksiyonel hem yapısal test tekniği (Gri kutu testi)

Yazılımların, hem işlevselliğini hem de iç yapısını, kodlarını test eden tekniktir. Literatürde buna gri kutu test tekniği karşılık gelmektedir.

Gri kutu testleri, beyaz kutu test tekniği ile kara kutu test tekniklerinin birlikte kullanılmasıdır [51]. Kara kutu testleri sistemin iç yapısı bilinmeden gereksinimler temel alınarak gerçekleştirilir; beyaz kutu testinde ise sistemin iç yapısı tam olarak bilinir ve testler bu iç yapı üzerinden gerçekleştirilir. Gri kutu tekniğinde ise gereksinimleri doğrulayacak test durumları kodun iç yapısı esas alınarak yazılır. Böylece gereksinimlerin gerçeklendiği doğrulandığı gibi yazılımın iç yapısı sınanmış olur.



Şekil 2.21. Gri kutu test şeması [1]

Gri kutu testlerinde test ekibinin yazılımın iç yapısını bilmesi, tasarıma ve kod yapısına karşı şartlanma olabileceğinden bazı hataları ortaya çıkartabilecek testlerin yapılmadan kalmasına sebep olabilir [1].

Tecrübeye dayalı test tekniği (Experience based testing techniques)

Test faaliyetinin titiz, sistematik ve tam olması gerekmesine rağmen, bir kişinin bilgisi, tecrübesi, hatırası ve sezgisi üzerine temellenen bazı sistematik olmayan teknikler vardır. Bir hata yakalayıcı, genellikle sistemdeki yakalanması zor bir kusuru bulabilir. Özetle bu test tekniği, test eden kişinin sisteme ilişkin tecrübesine dayanarak yapılır [66].

Statik test

Statik testler, yazılım veya yazılım parçasının çalıştırılmadan gerçekleştirilen test türüdür. Statik testlerde yazılımın doğruluğu kontrol edilir. Bu amaçla kodun çalıştırılması yerine yazılan kod, hata bulmak amacıyla okunur. Bu okumada kod gözden geçirilerek incelenir ve statik olarak analiz edilir [1]. Statik testlerin önemi, hataların yazılım yaşam döngüsü içerisinde erken safhalarda yakalanmasını sağlamasıdır [67].

Gözden geçirme (Review)

Bir yazılım ürününün proje elemanlarına, yöneticilere, kullanıcılara veya diğer ilgili kişilere yorum yapmaları, eksiklik ve hataları ortaya koymaları veya onaylamaları için sunulması süreci veya bu amaçla yapılan toplantılardır [1].

Kod geçişleri (Walkthroughs)

Bir yazılımın geliştiricisi tarafından diğer geliştirme ekip üyelerine anlatılarak, yazılım ürününün iyileştirilmesine yönelik görüşlerin alınması ve standartların ihlali veya olası hataların belirlenmesidir [1].

Kontrol akış analizi (Control flow analysis)

Bir kontrol akış analizi, çalışma zamanında uygulanabilen fonksiyonların derleme zamanındakilere benzemesidir. Örneğin, bu analiz verilen bir programın tahmini kontrol akışını tanımlar. Genelde program analizleriyle ve özellikle de kontrol akış analizleriyle benzerlikler tahmin edilir [68].

Veri akış analizi (Data flow analysis)

Yazılım içerisindeki değişkenlerin tanımlanması, değişkenlerin kullanılması, metot/yordam dışından gelen değerlerin değişkene atanması veya değişkenin metot dışına değer göndermesi, değişkenlerin kullanıldıktan sonraki durumunun analiz edilmesidir. Bu analiz sonucunda elde edilen bulgular kontrol akış grafikleri veya çağrı grafikleri kullanılarak gösterilir. Veri akış analizleri ileri yönde ve geri yönde olmak üzere iki kapsamda gerçekleştirilir. İleri yönde gerçekleştirilen veri akış analizlerinde akış kontrolü yapılan değişkenin ileriye doğru değer aktarması incelenerek kontrol akış grafiği çizilir. Geri yönde veri akış analizlerinde ise, akış kontrolü yapılacak olan değişkenin en son yapıldığı yere kadar incelenerek kontrol akış grafiği çizilir [1].

İnceleme (Inspection)

Bir yazılım ürünündeki hatalar ve standartlardan sapmalara neden olan anormalliklerin belirlenip tanımlanması için inceleme teknikleri konusunda eğitimli, tarafsız kişilerin rehberliğinde denk kişilerin katılımıyla gerçekleştirilen sorgulamadır [1].

İnceleme, yazılımı geliştiren değil eğitimli yönetici tarafından yönetilir. Genelde emsal sınamalar olarak yürütülür, roller tanımlanır, metrik bilgileri içerir. Resmi süreç, kural ve kontrol listeleri üzerine dayanır, ön toplantı hazırlığı yapılır, inceleme raporu elde edilen bulguların listesini içerir. Temel amaç, hataları bulmaktır [65].

3. YAZILIMDA KALİTE

Yazılım sistemlerinin günlük hayatımızdaki rolünün artmasıyla, yazılım kalitesinin önemi her geçen gün artmaktadır. Yazılım kalitesi, yazılım test sürecinin verimliliğinden etkilenir [69].

Yazılım kalitesi kavramı, halen belirsiz ve farklı kişilerce farklı anlamlar ifade edilen bir kavramdır [70, 71]. Geliştiricinin bakış açısıyla ya da içsel bakış açısı ile kalite, maliyet ve gecikmenin doğru tahminine, daha kolay teste, daha iyi bakıma giden yoldur [70, 72]. Kullanıcı tarafından ya da dışsal bakış açısı ile bakıldığında ise kullanım kolaylığı, estetik, anlaşılabilirlik vb. kavramlar ifade edebilmektedir. Yazılım kalitesinin ölçümü yazılımın birçok yönden değerlendirilmesi ve iyileştirilmesine olanak tanıması nedeniyle önem arz etmektedir [70].

Literatürde kullanılan en yaygın yazılım kalitesi tanımları şunlardır;

- Yazılım kalitesi bir kalite faktörleri kümesi tarafından kararlaştırılır. (Örneğin, ISO 8402-1986 ve IEEE 610.12-1990 gibi).
- Yazılım kalitesi, kullanıcı memnuniyeti tarafından kararlaştırılır.
- Yazılım kalitesi, yazılımdaki hatalar veya beklenmeyen davranışlar tarafından kararlaştırılır [73-75].

Kaliteli yazılım, önceden belirlenmiş olan işlevsel gereksinimlere, yazılım geliştirme standartlarına ve bir yazılımdan beklenen bütün özelliklere tamamen uygun bir yazılımdır.

3.1. Yazılım Kalitesi Faktörleri

Yazılım kalitesini oluşturan faktörler şunlardır;

Doğruluk, spesifikasyonlara uygunluk ve müşteri isteklerini karşılama derecesi,

Güvenirlik, tasarlanan işlevleri istenilen duyarlıkta yerine getirme olanağı,

Verimlilik, programın işlevlerini yerine getirebilmesi için kullandığı bilgi işlem kaynaklarının miktarı,

Güvenlik, yetkisiz kişilerin yazılma müdahalesini veya veri girişini önleme olanağı,

Kullanışlılık, öğrenme, işletme, girdi hazırlama ve çıktı yorumlamada kolaylık derecesi,

Hata bulma kolaylığı, hatanın yerini bulma ve düzeltme kolaylığı,

Esneklik, yazılımda değişiklik yapma kolaylığı,

Sınama kolaylığı, yazılımın doğruluğunun sınanmasındaki kolaylık,

Taşınabilirlik, programın farklı donanımlarda ve yazılım sistemi ortamlarında kullanılma olanağı,

Tekrar kullanılabilirlik, yazılımın tamamının ya da bir bölümünün farklı bir uygulamada kullanılma olanağı,

Bağlanabilirlik, bir sistemin başka bir sisteme bağlanma olanağı [45].

3.2. Kalite Özellikleri

Yazılım kalitesinin, günlük yaşamı çeşitli bakımlardan etkilemesi sebebiyle önemli olması, bir yazılım ürünü için birçok kalite model yaklaşımının geliştirilmesine sebep olmuştur [76]. Bu kalite modelleri, yazılımların amacını belirlemek ve yazılımları değerlendirmek için geliştirilir. Örneğin, literatürde McCall [77], Boehm [78], ISO/IEC 9126, Dromey [79] ve Bansiya [80] tarafından geliştirilen kalite modelleri mevcuttur. Aşağıda literatürde yaygın bir şekilde kabul gören ve kullanılan ISO/IEC 9126 kalite modeli ve özellikleri açıklanmıştır.

3.2.1. ISO/IEC 9126 kalite modeli

Uluslararası ISO/IEC 9126 standardı dünyada yazılım için kullanılan en yaygın kalite modelidir. ISO/IEC 9126 standardı yazılım ürününü değerlendirme standardıdır. ISO 9126'ya göre kalite özellikleri ve alt sınıfları şu şekildedir [81].

<u>İşlevsellik (Functionality)</u>

Belirlenen şartlar altında, tanımlanan ya da ima edilen ihtiyaçları sağlamak üzere bir araya gelmiş fonksiyonlar ve ilgili fonksiyonlara bağlı özellikler bütünüdür [81].

Yazılımın ihtiyaçları karşılama becerisi olarak tanımlanmaktadır. Uygunluk, doğruluk, birlikte çalışabilirlik ve güvenlik konuları bu kategori altında incelenmektedir [82].

Uygunluk: Uygulamanın belirlenen görevleri yerine getirmek için gerekli fonksiyonlara sahip olduğunu gösteren özelliklerdir.

Doğruluk: Uygulamanın taahhüt edilen verimlilik ve sonuçlarla çalıştığını gösteren özelliklerdir.

Birlikte işlerlik: Uygulamanın belirlenen sistemlerle entegre olabildiğini gösteren özellikleridir.

Güvenlik: Uygulamanın, istenen verilere ve fonksiyonlara yetkisiz erişimleri engellediğini gösteren özellikleridir [81].

Fonksiyonel testler; farklı birimleri, tümleştirilmiş birimleri, uygulamaları ve sistemleri test etmek için çeşitli zamanlarda uygulanır ve çalıştırılır [69, 83].

Güvenilirlik (Reliability)

Uygulamanın belirlenen koşullar altında performansını kaybetmeden çalışabilmesi kabiliyetidir [81]. Yazılımın düzgün çalışma halini muhafaza edebilme becerisi olarak tanımlanmaktadır. Olgunluk, hata toleransı ve geri kurtarma konuları bu kategori altında incelenmektedir [82].

Olgunluk: Uygulamanın en az hata ile çalışabilmesi kabiliyetidir.

Hata Toleransı: Uygulamanın, karşılaştığı yazılım hatalarında veya arayüz sorunlarında performansını kaybetmeden çalışabilmesi kabiliyetidir.

Geri Kurtarma /Yüklenebilirlik: Uygulamanın karşılaştığı hata sonucunda hatadan etkilenen veriyi geri yükleyerek kurtarabilme kabiliyetidir [81].

Kullanılabilirlik (Usability)

Uygulamanın kolay öğrenilebilme, anlaşılabilme ve kullanılabilme kabiliyetidir [81]. Yazılımın kullanım kolaylığı sağlayan yetenekleri olarak tanımlanmaktadır.

Öğrenebilme, anlaşılabilirlik, işletilebilirlik ve kullanıcı etkileşimi konuları bu kategori altında incelenmektedir [82].

Anlaşılabilirlik: Uygulamanın belirli koşullar ve özel durumlarda kullanıcının ilgili fonksiyonu ne sadelikte uygulayabileceğini ve ne rahatlıkta anlayabileceğini belirleyen uygulama özellikleridir [81]. Bir model biçimini anlamak için gerekli çaba miktarıdır. Anlaşılırlık, değişiklik ve yeniden kullanılırlık ile ilişkilidir. Bir model ne kadar kolay anlaşılırsa, değişiklik ve yeniden kullanım o kadar kolay olur [84]. Kolayca öğrenmeye ve anlamaya izin veren tasarım özelliğidir [48]. Kısaca, tasarımın fiziksel karmaşıklığı arttırıp arttırmadığını sınamaktadır [49].

Öğrenilebilirlik: Kullanıcıların uygulamayı ne zorlukta öğreneceğini belirleyen uygulama özellikleridir.

İşlerlik: Kullanıcıların uygulamayı kontrol etmede ve işletmede kullandıkları uygulama özellikleridir [81].

Verimlilik (Efficiency)

Uygulamanın kısıtlı kaynakla ortaya koyduğu performans özellikleridir [83]. Nesne yönelimli tasarım özelliklerini kullanarak gerekli işlevsellik ve davranışı başarmadaki tasarım yeterliliği olarak tanımlanır [48]. Yapılar etkili bir şekilde tasarlanmış mı [49]? Yazılımın ihtiyaç duyulan ölçüde yeterli performansla çalışabilme becerisi olarak tanımlanmaktadır. Zaman ve kaynak kullanımı konuları bu kategori altında incelenmektedir [82].

Zaman Davranışı: Uygulamanın belirli koşullar altında fonksiyonu çalıştırırken sistemin verdiği yanıt süresini, fonksiyonu işletme süresini ve işin net çalışma süresini belirleyen özelliklerdir.

Kaynak Yararlanımı: Uygulamanın çalışırken tükettiği donanımsal kaynağı belirleyen özellikleridir [81].

Bakılabilirlik/Değiştirilebilirlik (Maintainability/Changeability)

Uygulamanın yeni ihtiyaçlara uyum sağlama, düzeltici ya da geliştirmeye yönelik modifikasyonlara cevap verebilme özelliğidir [81]. Yazılımın değişiklik veya düzeltme isteklerine adaptasyon yeteneği olarak tanımlanmaktadır. Değiştirilebilirlik, test edilebilirlik, analiz edilebilirlik ve bağışıklılık konuları bu kategori altında incelenmektedir [82].

Analiz edilebilirlik: Uygulama da hata tespitinde, hatanın belirlenebilmesi için uygulamanın sağladığı özelliklerdir.

Değiştirilebilirlik: Uygulamada yapılacak değişikliğin uygulanabilmesi kabiliyetidir.

Durağanlık/Bağışıklık: Uygulamada yapılan değişikliğin uygulamada beklenmedik bir durum oluşturmama özelliğidir.

Test edilebilirlik: Uygulamada yapılan değişikliğin doğrulanabilme kabiliyetidir [81]. Bir programın hatalarını ortaya çıkarma ve özelliklerini toplama faaliyetlerindeki test etme kolaylığıdır [76]. Yapının, test kolaylığını ve değişiklikleri destekleyip desteklemediğini sorgular [49].

Taşınabilirlik (Portability)

Uygulamanın bir ortamdan başka bir ortama taşınabilme kabiliyetidir [81]. Yazılımın farklı çalışma ortamlarına uyum sağlayabilme yeteneği olarak tanımlanmaktadır. Adaptasyon/uyum yeteneği, yüklenebilirlik özellikleri**,** ortam değiştirme imkânı ve diğer yazılımlarla uyum konuları bu kategori altında incelenmektedir [82].

Uyum Yeteneği: Uygulamanın amacında ve aksiyonlarında değişiklik yapmadan uygulamanın farklı ortamlara uyum sağlayabilme özelliğidir.

Kurulum Kolaylığı: Uygulamanın belirlenen ortama kurulabilme kabiliyetidir.

Uygunluk: Uygulamanın aynı ortamda başka bir ürün ile birlikte çalışabilme kabiliyetidir.

Değiştirilebilirlik: Uygulamanın aynı amaç için aynı ortamda başka bir ürün ile değiştirilebilmesi kabiliyetidir [81].

3.3. Yazılım Kalitesinin Sağlanması

Bir yazılım ürünü, çok sayıda ve bütünleşik alt programlardan oluşmaktadır. Yazılımın geliştirilmesi de, çok sayıda sistem analistinin katkısı ile, uzun bir sürede ve basamaklar halinde gerçekleştirilmektedir. Yazılım ürününün kullanılması aşamasında ortaya çıkan hata ve eksiklerin giderilmesi, donanımın değiştirilmesi ya da bilgisayar sisteminin geliştirilmesi hallerinde de yazılımın yeni durumuna uydurulması gerekmektedir (yazılımın bakımı ve onarımı). Bu işlemler sırasında çeşitli hatalar yapılabilmekte ve yazılım kalitesi yetersiz olabilmektedir.

Yazılım kalitesini sağlamak için, hata ve eksikliklerin anında bulunması ve düzeltilmesi büyük önem taşımaktadır. Çünkü bir basamakta yapılan hata, sonraki basamakları da etkilemektedir. İleri basamaklarda bu hatanın bulunması olasılığı giderek azalmakta, düzeltilmesi için gerekli emek ve gider de zamanla orantılı olarak artmaktadır.

Yazılım kalitesi,

- geliştirmenin planlanması aşamasında kalite kontrolü yöntem ve araçlarının belirlenmesi,
- geliştirme sürecinin durak noktalarında yapılanların gözden geçirilmesi,
- kaynak programın sınanması aşamalarında gerçekleştirilmelidir [45].

4. CHIDAMBER VE KEMERER METRİK DEĞERLENDİRME MODÜLÜNÜN (CKMDM) GERÇEKLEŞTİRİLMESİ

Metrikler, yazılım kalitesinin ölçülmesi ve geliştirilmesi faaliyetlerinde önemli bir yer tutmaktadır. Bu faaliyetler aşırı bağımlı, uyumsuz, karmaşık ve hatalı modüllerin belirlenmesini sağlar. Dolayısıyla, sözkonusu belirlemeler hangi modüllerin öncelikli olarak test edileceği veya hangi testlerin öncelikli olarak uygulanacağı, bakım-onarım için ne kadar bütçe ve zaman harcanacağı gibi bilgilere ulaşmada önemli ipuçları verir. Böylece, yazılım kalitesinin arttırılması sağlanmış olur. Ancak, tüm bu faaliyetlerin olumlu sonuçlara ulaşması için doğru metriklerin belirlenmesi, belirlenen metriklerin doğru biçimde ölçülmesi ve doğru bir şekilde yorumlanması gerekir [82].

Geliştirilen veya tanımlanan metriklerin yorumlanması veya değerlendirilmesi noktasında literatürde, bazı yöntemler mevcuttur. *Özellik tabanlı metrik ölçümü* bunlardan birisidir. Bu yöntemde sözkonusu metriklerin özellikleri dikkate alınarak değerlendirme veya yorumlama faaliyetleri gerçekleşmektedir. Örneğin, geliştirilen yazılımdaki kod sayısı veya alt sınıf sayısı o yazılımın karmaşıklığı hakkında bilgi verir. Bir diğeri ise *belirlenen metriklerin ürettiği değerler ile aynı amaca hizmet eden başka metriklerin ürettiği değerler arasındaki ilişki* ölçülerek yapılan değerlendirme yöntemidir. Bir başka yöntemde ise *aynı metrik grubunun ölçülecek olan yazılımın sürümleri üzerinde uygulanması* ile yapılan değerlendirmelerdir. Başka bir ifadeyle, önceki yazılım sürüm/lerin metrik değerleri ve kusur bilgileri kullanılarak bir model ortaya çıkarılır. Ardından yeni sürümdeki metrik değerleri ve kusur bilgileri ile karşılaştırılarak kalite düzeyi yorumlanır. Bu metrik değerlerini ve kusur bilgilerini yorumlamada, genetik algoritmalar, yapay sinir ağları, uzman sistemler veya karar ağaçları gibi farklı yöntemler kullanılır [82, 85-88].

Literatürde Bansiya [80], Amstel ve ark [84], Chidamber ve Kemerer [89] ve Brito'e Abreu [90] gibi bilim adamlarının bu konudaki çalışmaları olmakla birlikte değerlendirme yöntemlerinin bir yazılım aracı kullanılarak otomasyon sağlanması noktasında eksiklikler vardır [82].

Aslında, günümüzde Findbugs [28], Metrics [29], PMD [30] ve Coverity [34] gibi ücretli veya ücretsiz nesne yönelimli yazılım test araçları geliştirilmiştir. Ancak, bu araçlar genellikle metriklerin sayısal değerlerini vermekle kalmaktadır. Elde edilen değerlerin yorumlanması veya değerlendirmesini yapan herhangi bir yazılım geliştirilmemiştir. Sözkonusu eksiklikten yola çıkarak bu çalışmada Chidamber ve Kemerer'in metrik kümesini değerlendiren bir uzman modül tasarlanmış ve gerçekleştirilmiştir. Chidamber ve Kemerer metrik kümesinin seçilmesinin sebebi, hem literatürde yaygın olarak kullanılması hem daha çok nesne yönelimli yazılım özelliklerini (kapsülleme, kalıtım, uyum vb.) kapsaması, hem de bir sistemin bütün olarak değerlendirilmesinden ziyade sistemdeki sınıfları geniş bir şekilde değerlendirme imkanı tanımasıdır. Geliştirilen uzman modül, metrik bulgularını hem el ile hem de otomatik olarak değerlendirme imkanı sağlamaktadır. Değerlendirme, uzman sistem yaklaşımı ile *özellik tabanlı metrik ölçümü* ve *metrik bulgularının sürümler arası karşılaştırılması* yöntemleri kullanılarak yapılmıştır. Sözkonusu modüle, "Chidamber ve Kemerer Metrik Değerlendirme Modülü" ifadesinin baş harflerinden oluşan CKMDM adı verilmiştir. Bu modülü geliştirerek, metrik değerlendirilmesinin yaygınlaştırılması ve bir otomasyona bağlanması amaçlanmıştır.

Aşağıda, öncelikle uzman modülde kullanılan Chidamber ve Kemerer metrik kümesi kısaca anlatılmıştır. Daha sonra geliştirilen uzman modülün UML diyagramları, yapısı, uygulama adımları ve işleyişi ekran görüntüleri ile desteklenerek açıklanmıştır.

4.1. Chidamber ve Kemerer'in Nesne Yönelimli Metrik Kümesi

Nesne yönelimli tasarım için kullanılan bu metrik grubu, bir sistemin bütün olarak değerlendirilmesinden ziyade sistemdeki sınıfları geniş bir şekilde değerlendirmeyi amaçlar. Chidamber ve Kemerer, nesne yönelimli tasarım için altı adet metrik tanımlar. Metrikler ve özellikleri aşağıda kısaca açıklanmıştır.

Sınıfın ağırlıklı metot sayısı (Weighted Methods per Class - WMC)

Sınıfın ağırlıklı metot sayısı (WMC), bir sınıftaki metotların karmaşıklık derecesi veya sayısıdır. Bir sınıfın metotlarının karmaşıklığı ve sayısı, sınıfın geliştirilmesine ve bakımına harcanacak zaman-çaba hakkında fikir verir (82). WMC, nesne yönelimli bir yazılımın anlaşılırlığını, yeniden kullanılabilirliğini ve dayanıklılığını/bakılabilirliğini ölçmede kullanılır (49).

Kalıtım ağacının derinliği (Depth of Inheritance Tree - DIT)

Sınıfın kalıtım ağacının köküne uzaklığıdır (82). Bu değerin çok yüksek olması test edilebilirliğin çok düşük olduğunu, aksi halde nesne yönelim ilkelerinin fazla kullanılmadığını gösterir (91). Bu metrik verimliliği, yeniden kullanımı, anlaşılırlığı ve test edilebilirliği ölçer (49).

Alt sınıf sayısı (Number of Children - NOC)

Bir sınıftan direk türetilmiş alt sınıfların sayısıdır. NOC, kalıtımlı ifadeler dikkate alınarak hesaplanır. Bir sınıf hiyerarşisinin genişliğini ölçer. Alt sınıf sayısının fazla olması; yeniden kullanımın yüksek olduğunu, daha çok hatanın oluşabileceğini (89), test esnasında harcanacak zamanı-çabayı ve kalıtımın yanlış kullanıldığını gösterir (82). NOC, verimlilik, yeniden kullanılabilirlik ve test edilebilirlik düzeyini ölçer (49).

Nesne sınıfları arasındaki bağımlılık (Coupling Between Object Classes - CBO)

Bir sınıf içindeki özellik (attribute) ya da metotların (method) diğer sınıfta kullanılması ve sınıflar arasında kalıtımın olmaması durumunda iki sınıf arasında bağımlılıktan bahsedilebilir (89). CBO, verimliliği ve yeniden kullanılabilirliği ölçmede kullanılır (49).

Sınıfın tetiklediği metot sayısı (Response for a class - RFC)

Bir sınıftan bir nesnenin metotları çağrılması durumunda, bu nesnenin tetikleyebileceği tüm metotların sayısı RFC değerini verir. Yani, bir sınıfta yazılan ve çağırılan toplam metot sayısıdır (82). Bu metrik, sınıf seviye tasarım metriklerinden olup (48) anlaşılabilirliği, dayanıklılığı, karmaşıklığı ve test edilebilirliği ölçmede kullanılır (49).

Metotlardaki uyum eksikliği (Lack of cohesion in methods - LCOM)

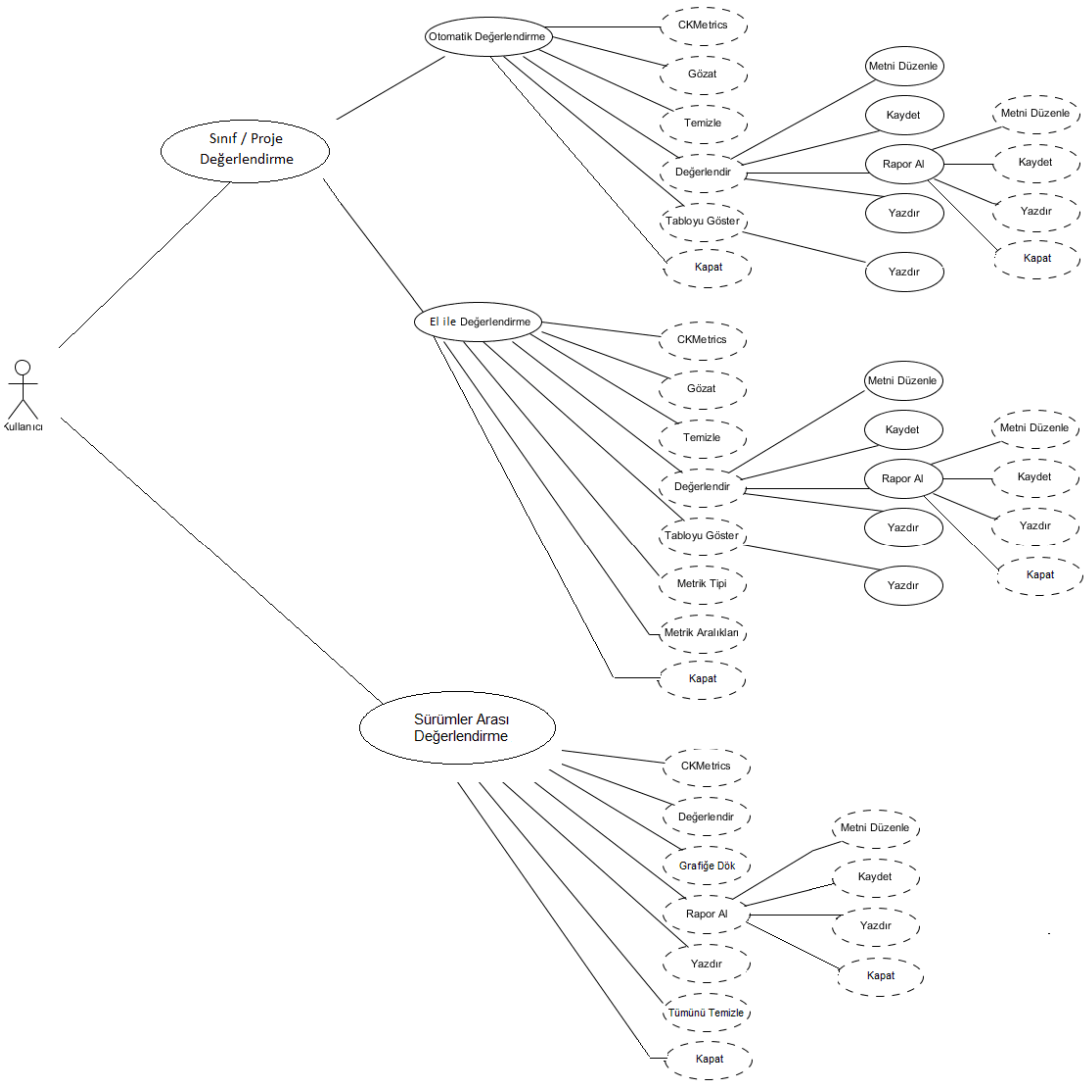
LCOM, n adet kümenin kesişiminden oluşan kümelerdeki uyumsuzlukların sayısıdır ve metotlardaki benzerlik derecesini ölçer. Metotlardaki uyum eksikliği; bir sınıfın, iki veya daha fazla alt sınıfa ayrıldığını gösterir ve karmaşıklığı arttırır. (82,92). Yapılan bir çalışmada, LCOM ölçütünün uyum özelliğini çok da iyi ayırt edemediği ispatlanmıştır (3). Literatürde LCOM2, LCOM3 (89) ve LCOM4 (82,42) adlarıyla yer alan farklı LCOM metrik tanımları da mevcuttur. LCOM metriği test ediciye, verimlilik ve yeniden kullanılabilirlik derecesi hakkında bilgi verir (49).

4.2. CKMDM UML Diyagramları

Bu çalışmada, uzman modül geliştirilirken kodlamanın kolaylaştırılması, yazılımdaki hataların minimuma indirgenmesi, yazılım maliyetinin ve iş gücünün düşürülmesi amacıyla UML diyagramlarından olan kullanım senaryo diyagramları (use case diagrams), bileşen diyagramları (component diagrams) ve aktivite diyagramlarından (activity diagrams) yararlanılmıştır.

4.2.1. Kullanım senaryo diyagramı (Use case diagram)

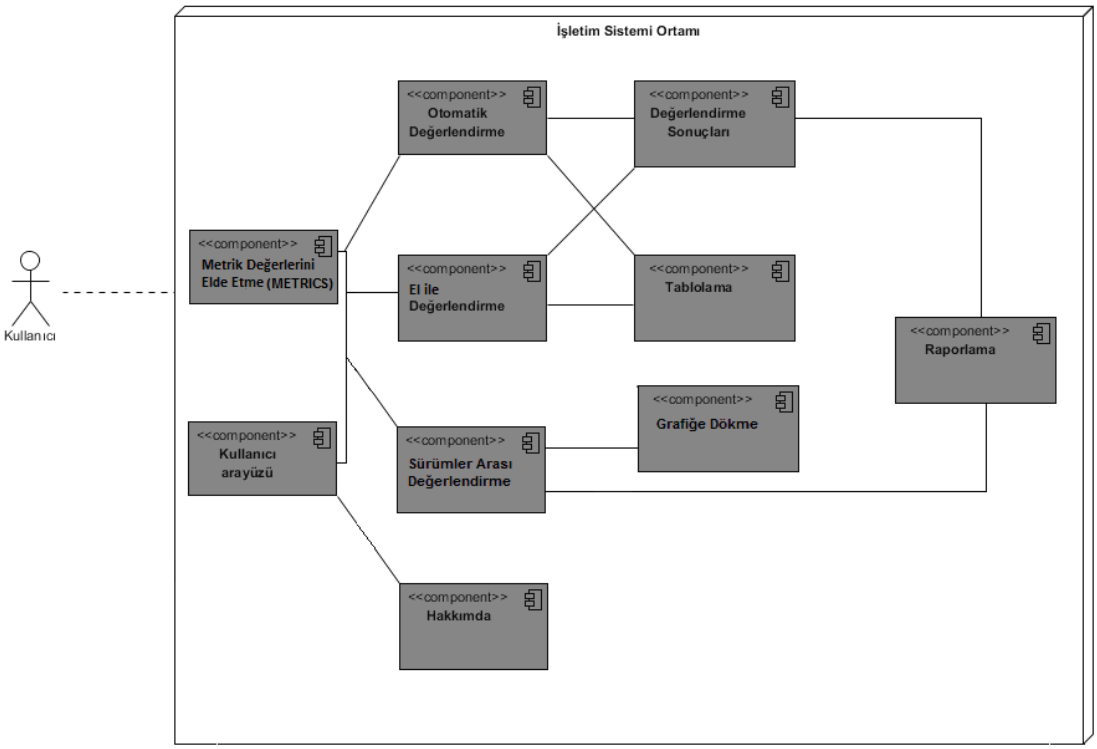
Geliştirilen uzman modülün davranışlarının kullanıcı gözüyle incelenebilmesi için kullanım senaryo diyagramı hazırlanmıştır. Sözkonusu diyagram Şekil 4.1'de gösterilmektedir.



Şekil 4.1. CKMDM kullanım senaryo diyagramı (use case diagram)

4.2.2. Bileşen diyagramı (Component diagram)

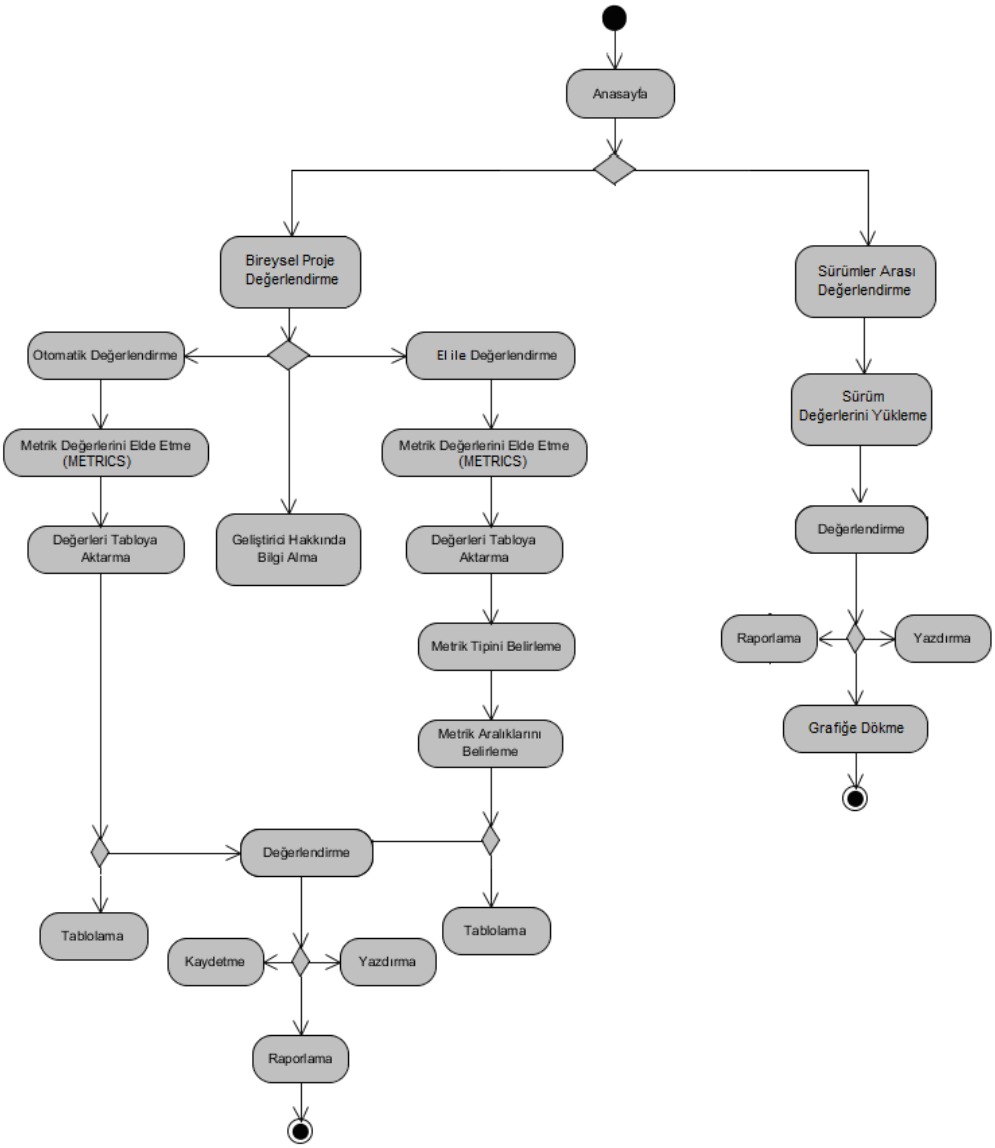
Birden fazla geliştirici tarafından uygulanan veya birden fazla modülü olan yazılım projelerinde sistemi, bileşen (component) denilen parçalara ayırmak, geliştirmeyi kolaylaştırmaktadır. Bunları modellemek için bileşen diyagramlar kullanılır. Sözkonusu uzman modüle ilişkin bileşen diyagramı Şekil 4.2'de gösterilmektedir.



Şekil 4.2. CKMDM bileşen diyagram (component diagram)

4.2.3. Aktivite diyagramı (Activity diagram)

Bir nesnenin durumu, zamanla kullanıcı ya da nesnenin kendi fonksiyonları tarafından değişebilir. Bu değişim aktivite diyagramları tarafından gösterilir. Sözkonusu diyagram Şekil 4.3'te gösterilmektedir.



Şekil 4.3. CKMDM aktivite diyagramı (activity diagram)

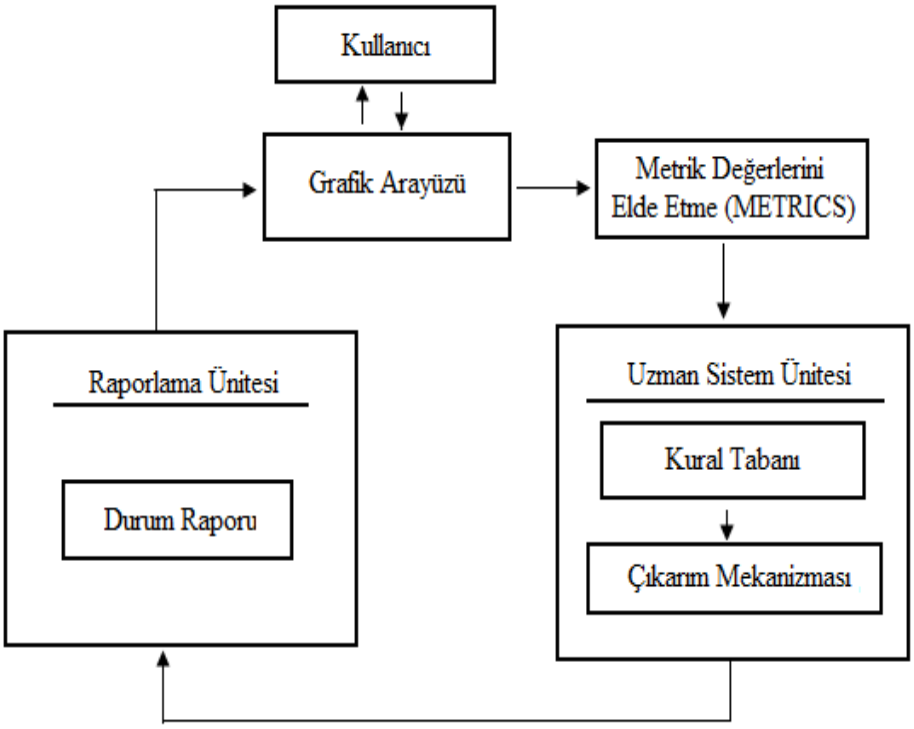
4.3. Uzman Sistemler

Uzman sistemler (US), belirli bir konuda uzman olan bir veya birçok insanın yapabildiği muhakeme ve karar verme işlemlerini modelleyen bir yazılım sistemidir [93]. Bir US programı, belli bir algoritmaya dayanmayan, kendi bilgi tabanı içerisinde, girilen veya önceden belli olan verilere göre arama yaparak bu veriye veya verilere uygun olan bilginin (kuralın) aktifleşmesini sağlayan ve bu aktifleşme sonucu yeni bir veri elde ederek aramaya devam eden bir sistem şeklinde çalışmaktadır [94].

Literatürde uzman sistemler kullanılarak, yazılım analizi [95], test durumlarını seçme ve test planı oluşturma [96], yazılım risk analizi yapma [97], yazılım mühendisliği yönetimi [98] ve nesne yönelimli davranışları durum değişikliklerine göre değerlendirme [99] gibi çalışmalar yapılmıştır. Ancak, nesne yönelimli tasarım özelliklerinin ele alınması ve değerlendirilmesi ile ilgili eksiklikler vardır. Sözkonusu eksiklikler bu çalışmanın hazırlanmasına sebep olmuştur.

4.4. CKMDM Uzman Modülün Yapısı

Geliştirilen uzman modülün yapısı Şekil 4.4'te gösterilmektedir.



Şekil 4.4. Geliştirilen uzman modülün yapısı

Kullanıcı

Uzman modülün kullanıcı kısmında bilgisayar kullanıcıları, yazılım geliştiricileri, testçiler, test yöneticileri, proje yöneticileri vs. olabilir.

Grafik Arayüzü

Bu bölüm, kullanıcılar ile sistem arasındaki iletişimin sağlandığı bölümdür. Kullanıcılar, grafik arayüzünü kullanarak sisteme giriş yaparlar, istedikleri durumda daha önce oluşturulmuş metrik bulgularını kullanırlar veya yeni metrik bulgularını oluştururlar. Elde ettikleri bulguları değerlendirerek sistem tarafından üretilen raporlara erişebilirler ve bu raporları belleklerine kaydedebilirler.

Metrik Değerlerini Elde Etme

Kullanıcı, uzman modüle entegre edilen "Metrics" [29] adlı programı kullanarak değerlendirme yapmak için gerekli olan Chidamber ve Kemerer'in metrik bulgularını hem sınıf hem de proje bazında elde eder.

Uzman Sistem Ünitesi

Verilerin ve kuralların saklandığı ve gerekli çıkarımların yapıldığı bölümdür. Bu bölüm iki alt bölümden oluşur. Bunlar; "Kural Tabanı" ve "Çıkarım Mekanizması"dır. Kurallar, kural tabanında tutulur. Çıkarım mekanizması ise kurallara göre verileri işleyerek anlamlı çıkarımların yapılmasını sağlar.

Kural Tabanı ve Çıkarım Mekanizması

Kural tabanı, değerlendirmek için kullanılan kuralların tutulduğu birimdir. Bu birimde, Chidamber ve Kemerer'ın metrikleri için, literatürde kabul gören ve kullanılan aralıklar kural tabanını oluşturur. Ancak, bu aralıklar farklı kaynaklarda muhtelif eşik değerlere sahiptir [48,49,82,85,89,91]. Bu yüzden bu çalışmada, sözkonusu eşik değerlerin ortalamasından yararlanılarak yeni bir kural tabanı oluşturulmuştur. Oluşturulan kural tabanı şöyledir;

WMC: 0-15
DIT: 0-7
NOC: 0-6
CBO: 0-5
RFC: 0-55
LCOM: 0-25

Oluşturulan bu kural tabanı, uzman modülde "Bilginin 'eğer - o halde' kuralıyla sunulması" yöntemi ile kullanılmıştır. "Otomatik Değerlendirme" başlığındaki metrik değerlendirmelerde kurallar, modülü tasarlayan tarafından kod şeklinde yazıldığından, kullanıcı bu kuralları görememekte ve herhangi bir ekleme veya değişiklik yapamamaktadır. Ancak, "El ile Değerlendirme" başlığı altındaki sayfada kural oluşturma modülü sayesinde yazılım bilgisine sahip olmadan da yeni kurallar belirleme işlemleri kolaylıkla yapılabilmektedir. Bu şekilde kullanıcılar metrik aralıkları için farklı kurallar/aralıklar oluşturabilmekte, bu da esnekliğe neden olmaktadır. Son olarak, "Sürümler Arası Değerlendirme" yönteminde ise, sürümlerin metrik değerleri karşılaştırılarak değerlendirme yapılır.

Çıkarım mekanizması ise, kurallar tabanındaki kuralların kullanılarak anlamlı çıkarımların yapıldığı birimdir. Geliştirilen uzman modül ile kurallar elde edilen bulgulara/verilere uygulanarak çıkarımlar elde edilir. Oluşturulan kurallar ve çıkarımlar şöyledir;

Kurallar ve Çıkarımlar

WMC metriği için;

Otomatik değerlendirme - El ile değerlendirme
Kural 1
Eğer WMC=0 ise;
Çıkarım 1
Sınıflarda hiç metot yoktur, bu da karmaşıklığın olmadığını ve herhangi bir hatadan da bahsedilemeyeceğini gösterir. Dolayısıyla, sınıfların metotlara sahip olmaması,

test faaliyetleri için herhangi bir bütçe harcanmayacağı ve kalite düzeyinden de söz edilemeyeceği anlamına gelir.

El ile değerlendirme
Kural 2
Eğer (WMC>=0,000001) VE (WMC<strtoint(ComboBox2.Text)) ise;
Çıkarım 2
Geliştirilen yazılımın az sayıda metoda sahip olduğunu ve dolayısıyla da herhangi bir karmaşıklıktan söz edilemeyeceği söylenebilir. Bütün bunlara bağlı olarak, bakım-onarım ve test için normalin de altında zaman ve iş gücü harcanacaktır. Aynı zamanda, elde edilen WMC değeri kalite özellikleri bakımından değerlendirildiğinde anlaşılırlığın ve dayanıklılığın yüksek, karmaşıklığın ise çok düşük olduğu görülür.

Otomatik değerlendirme
Kural 2
Eğer (WMC>=0,000001) VE (WMC<15) ise;
El ile değerlendirme
Kural 3
Eğer (WMC>=strtoint(ComboBox2.Text) VE (WMC<=strtoint(ComboBox6.Text) ise;
Otomatik değerlendirme
Çıkarım 2
El ile değerlendirme
Çıkarım 3
Geliştirilen yazılımın WMC değeri, istenen değerler aralığındadır. Yani bu durum, geliştirilen yazılımın istenen sayıda metoda sahip olduğunu, karmaşık olmadığını ve kod kalitesinin de yüksek olduğunu gösterir. Dolayısıyla, sözkonusu yazılımın geliştirilmesi, bakım-onarım-test faaliyetleri için az zaman ve iş gücü harcanacaktır. Aynı zamanda, bu WMC değeri kalite özellikleri bakımından incelendiğinde, anlaşılırlığın, dayanıklılığın ve yeniden kullanılabilirliğin çok yüksek, karmaşıklığın ise düşük olduğu görülmektedir.

Otomatik değerlendirme

Kural 4

Eğer (WMC>=15) ise;

El ile değerlendirme

Kural 4

Eğer (WMC>strtoint(ComboBox6.Text) ise;

Çıkarım 4

Elde edilen bulgunun çok fazla olması, geliştirilen yazılımın çok karmaşık olduğunu ve kod kalitesinin de çok düşük olduğunu gösterir. Dolayısıyla, sözkonusu yazılımın geliştirilmesi, bakım-onarım ve test faaliyetleri için normalden fazla zaman ve iş gücü harcanacaktır. Aynı zamanda, bu WMC değeri, kalite özellikleri bakımından incelendiğinde, anlaşılırlığın, dayanıklılığın ve yeniden kullanılabilirliliğin çok düşük; karmaşıklığın ise çok yüksek olduğu anlaşılır.

DIT metriği için;

Otomatik değerlendirme - El ile değerlendirme

Kural 1

Eğer (DIT=0) ise;

Çıkarım 1

Bu değerin 0(sıfır) olması ağaç derinliğinden söz edilemeyeceği anlamına gelir. Bu da sözkonusu yazılımda karmaşıklığın olmadığını, kalıtım özelliklerinin kullanılmadığını ve dolayısıyla da herhangi bir hatadan da bahsedilemeyeceğini gösterir. Bütün bunlara ilaveten, ağaç derinliliğinin olmaması, bakım-onarım ve test faaliyetleri için herhangi bir bütçe harcanmayacağı anlamına gelir.

El ile değerlendirme

Kural 2

Eğer (DIT>=0,000001) VE (DIT<strtoint(ComboBox3.Text)) ise;

Çıkarım 2

Elde edilen değer belirlenen ilk aralığında altındadır. Geliştirilen bu yazılımın davranışlarını tahmin etmek çok da zor olmayacaktır. Aynı zamanda kalıtım

metotlarının tekrar kullanılma potansiyeli çok düşüktür. Bu da, bir yazılım ürününde istenen bir durumdur. Bu değer, kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik, anlaşılabilirlik ve test edilebilirlik derecesi yüksek, karmaşıklık bakımından ise düşük olduğu görülmektedir.

Otomatik değerlendirme
Kural 2
Eğer (DIT>=0,000001) VE (DIT<=7) ise;
El ile değerlendirme
Kural 3
Eğer (DIT>=strtoint(ComboBox3.Text)) VE (DIT<=strtoint(ComboBox7.Text)) ise;
Otomatik değerlendirme
Çıkarım 2
El ile değerlendirme
Çıkarım 3
Geliştirilen yazılımın DIT değeri, istenen aralıktadır. Yani bu durum, geliştirilen yazılımın ağaçlarının derinlik seviyesinin düşük olduğu ve daha az sayıda metot veya sınıfa sahip olduğu anlamına gelir. Bu da, tasarımı karmaşıklıktan kurtaracaktır. Buna ilaveten, geliştirilen bu yazılımın davranışlarını tahmin etmek çok kolay olacaktır. Aynı zamanda kalıtım metotlarının tekrar kullanılma potansiyeli düşüktür. Bu da, bir yazılım ürününde istenen bir durumdur. Geliştirilen yazılımlarda DIT metrik değerinin düşük olması yeğlenir. Bu değer kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik, anlaşılabilirlik ve test edilebilirlik derecesinin çok yüksek, karmaşıklık bakımından ise çok düşük olduğu görülmektedir.

Otomatik değerlendirme
Kural 4
Eğer (DIT>7) ise;
El ile değerlendirme
Kural 4
Eğer (DIT>strtoint(ComboBox7.Text)) ise;

Çıkarım 3

Geliştirilen yazılımın DIT değeri, istenen aralığın üstündedir. Yani bu durum, geliştirilen yazılımın ağaçlarının derinlik seviyesinin yüksek olduğu ve daha fazla sayıda metot veya sınıfa sahip olduğu anlamına gelir. Bu da, tasarımı karmaşık hale getirecektir. Aynı zamanda kalıtım metotlarının tekrar kullanılma potansiyeli yüksektir. Bu da, bir yazılım ürününde istenmeyen bir durumdur. Geliştirilen yazılımlarda DIT metrik değerinin düşük olması yeğlenir. Bu değer kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik, anlaşılabilirlik ve test edilebilirlik düzeyleri çok düşük; karmaşıklık düzeyinin ise çok yüksek olduğu görülmektedir.

NOC metriği için;

Otomatik değerlendirme - El ile değerlendirme

Kural 1

Eğer (NOC=0) ise;

Çıkarım 1

Alt sınıf sayısı hiç yoktur, bu da yeniden kullanımın olmadığını ve dolayısıyla herhangi bir hatadan da bahsedilemeyeceğini gösterir. Eğer bir sınıf, alt sınıfa sahip değilse, bu durum kalıtımın hiç kullanılmadığının bir göstergesi olabilir. Dolayısıyla, alt sınıfa sahip olmaması, bakım-onarım ve test faaliyetleri için herhangi bir bütçe harcanmayacağı anlamına gelir. Buna ilaveten bu değer, kalite özellikleri bakımından incelendiğinde, herhangi bir yorum yapılamaz.

El ile değerlendirme

Kural 2

Eğer (NOC>=0,000001) VE (NOC<strtoint(ComboBox4.Text)) ise;

Çıkarım 2

Elde edilen değer belirlenen ilk aralığında altındadır. Yani, alt sınıf sayısı çok az düzeydedir, bu da yeniden kullanımın çok düşük olduğunu ve daha az hatanın olabileceğini gösterir. Eğer bir sınıf, çok az alt sınıfa sahipse, bu durum kalıtımın çok da fazla kullanılmadığının bir göstergesi olabilir. Dolayısıyla, geliştirilen bu

yazılımdaki alt sınıf sayısının çok az olması, metotları daha az test etmeyi gerektirdiğinden, test faaliyetleri için harcanacak bütçenin de çok az düzeyde olmasını sağlayacaktır. Buna ilaveten sözkonusu yazılım kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik ve test edilebilirlik derecelerinin yüksek olduğu görülmektedir.

Otomatik değerlendirme
Kural 2
Eğer (NOC>=0,000001) VE (NOC<=6) ise;
El ile değerlendirme
Kural 3
Eğer (NOC>=strtoint(ComboBox4.Text)) VE (NOC<=strtoint(ComboBox8.Text)) ise;
Otomatik değerlendirme
Çıkarım 2
El ile değerlendirme
Çıkarım 3
Alt sınıf sayısı normal/istenen aralıktadır, bu da yeniden kullanımın düşük düzeyde olduğunu ve daha az hatanın olabileceğini gösterir. Eğer bir sınıf, normal düzeyde alt sınıfa sahipse, bu durum kalıtımın doğru kullanıldığının bir göstergesi olabilir. Dolayısıyla, geliştirilen bu yazılımdaki alt sınıf sayısının az olması, metotları daha az test etmeyi gerektirdiğinden, bakım-onarım ve test faaliyetleri için harcanacak bütçenin de daha düşük seviyede olmasını sağlayacaktır. Buna ilaveten sözkonusu yazılım kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik ve test edilebilirlik derecelerinin çok yüksek olduğu görülmektedir.

Otomatik değerlendirme
Kural 4
Eğer (NOC>6) ise;
El ile değerlendirme
Kural 4
Eğer (NOC>strtoint(ComboBox8.Text)) ise;

Çıkarım 4

Alt sınıf sayısı çok fazladır, bu da yeniden kullanımın yüksek olduğunu ve daha çok hatanın olabileceğini gösterir. Eğer bir sınıf, çok fazla alt sınıfa sahipse, bu durum kalıtımın yanlış kullanıldığının bir göstergesi olabilir. Dolayısıyla, geliştirilen bu yazılımdaki alt sınıf sayısının fazla olması, metotları daha çok test etmeyi gerektirdiğinden, bakım-onarım ve test faaliyetleri için harcanacak bütçenin de fazla olmasına sebep olacaktır. Buna ilaveten sözkonusu yazılım kalite özellikleri bakımından incelendiğinde, verimlilik, yeniden kullanılabilirlik ve test edilebilirlik derecelerinin de çok düşük olduğu görülmektedir.

CBO metriği için;

Otomatik değerlendirme - El ile değerlendirme

Kural 1

Eğer (CBO=0) ise;

Çıkarım 1

Geliştirilen yazılımın bağımlılık düzeyi sıfırdır. Bu da, modüler tasarım hakkında herhangi bir ipucu vermez. Sözkonusu yazılımın bakım-onarımı ve test faaliyetleri için herhangi bir maliyet de olmayacaktır. Aynı zamanda, kalite özellikleri bakımından incelendiğinde, verimlilik ve yeniden kullanılabilirlik hakkında da herhangi bir yorum yapılamaz.

El ile değerlendirme

Kural 2

(CBO>=0,000001) VE (CBO<strtoint(ComboBox5.Text)) ise;

Çıkarım 2

Elde edilen değer belirlenen ilk aralığın da altındadır. Yani, geliştirilen yazılımın bağımlılık düzeyi çok düşüktür. Bu da, modüler tasarım ve tekrar kullanılma ihtimali hakkında anlamlı yorum yapılamayacağı anlamına gelir. Sözkonusu yazılımın bakım-onarımı çok kolaydır ve test maliyeti de oldukça düşüktür. Aynı zamanda, verimlilik ve yeniden kullanılabilirlik gibi kalite özellikleri hakkında yapılacak yorumlar herhangi bir anlam ifade etmeyecektir.

Otomatik değerlendirme

Kural 2

Eğer (CBO>=0,000001) VE (CBO<=5) ise;

El ile değerlendirme

Kural 3

Eğer (CBO>=strtoint(ComboBox5.Text)) VE (CBO<=strtoint(ComboBox9.Text)) ise;

Otomatik değerlendirme

Çıkarım 2

El ile değerlendirme

Çıkarım 3

Geliştirilen yazılımın bağımlılık düzeyi istenen aralıktadır. Bu da, modüler tasarımın çok iyi olduğu ve tekrar kullanılma ihtimalinin yüksek olduğu anlamına gelir. Dolayısıyla, sözkonusu yazılımın bakım-onarım ve test faaliyetleri kolay olmakla birlikte test maliyeti de düşük olacaktır. Aynı zamanda, kalite özellikleri bakımından incelendiğinde, verimlilik ve yeniden kullanılabilirlik düzeyinin yüksek olduğu görülmektedir.

Otomatik değerlendirme

Kural 4

Eğer (CBO>5) ise;

El ile değerlendirme

Kural 4

Eğer (CBO>strtoint(ComboBox9.Text)) ise;

Çıkarım 4

CBO değerine göre;

Geliştirilen yazılımın bağımlılık düzeyi eşik değerinin üstündedir ve yüksektir. Bu da, modüler tasarımın çok kötü olduğu ve tekrar kullanılma ihtimalinin çok düşük olduğu anlamına gelir. Dolayısıyla, sözkonusu yazılımın bakım-onarım-test faaliyetleri çok zor olmakla birlikte test maliyeti de çok yüksek olacaktır. Aynı zamanda, bu yazılımın kalite özellikleri bakımından incelendiğinde, verimlilik ve yeniden kullanılabilirlik düzeyinin çok düşük olduğu görülmektedir.

RFC metriği için;

Otomatik değerlendirme - El ile değerlendirme
Kural 1
Eğer (RFC=0) ise;
Çıkarım 1
Geliştirilen yazılımın bu metrik değerine bakılırsa sınıfların tetiklediği metotlardan söz etmenin mümkün olmadığı görülmektedir. Yani, bu sınıflar için bakım-onarım-test ve hata ayıklama faaliyetleri olmayacaktır. Buna ilaveten, bakım-onarım ve test faaliyetleri için zaman ve iş gücü de harcanmayacaktır. RFC değerinin 0 (sıfır) olması, kalite özellikleri bakımından incelendiğinde, değerlendirme yapmanın anlamsız olacağını gösterir.

El ile değerlendirme
Kural 2
Eğer (RFC>=0,000001) VE (RFC<strtoint(ComboBox10.Text)) ise;
Çıkarım 2
Elde edilen değer belirlenen ilk aralığın da altındadır. Yani, geliştirilen yazılımın testi ve hata ayıklama faaliyetleri kolay olacaktır. Buna ilaveten, bakım-onarım ve test faaliyetleri için çok da fazla zaman harcanmayacaktır. RFC değerinin çok düşük olması, kalite özellikleri bakımından; anlaşılabilirliği, dayanıklılığı ve test edilebilirliği de arttırmaktadır.

Otomatik değerlendirme
Kural 2
Eğer (RFC>=0,000001) VE (RFC<=55) ise;
El ile değerlendirme
Kural 3
Eğer (RFC>=strtoint(ComboBox10.Text)) VE (RFC<=strtoint(ComboBox13.Text)) ise;
Otomatik değerlendirme
Çıkarım 2

El ile değerlendirme

Çıkarım 3

Geliştirilen yazılımın bu metrik değerine bakılırsa sınıfların tetiklediği metot sayısının istenen aralıkta olduğu görülmektedir. Yani, bu sınıflar için bakım-onarım-test ve hata ayıklama faaliyetleri düşük seviyede olacaktır ve sözkonusu faaliyetler için az zaman harcanacaktır. RFC değerinin düşük olması istenmekle birlikte, geliştirilen bu yazılım, kalite özellikleri bakımından incelendiğinde, anlaşılabilirlik, dayanıklılık ve test edilebilirlik düzeyinin çok yüksek olduğu görülmektedir.

Otomatik değerlendirme

Kural 4

Eğer (RFC>55) ise;

El ile değerlendirme

Kural 4

Eğer (RFC>strtoint(ComboBox13.Text)) ise;

Çıkarım 4

Geliştirilen yazılımın bağımlılık düzeyi eşik değerinin üstündedir ve yüksektir. Yani, sözkonusu yazılım için bakım-onarım-test ve hata ayıklama faaliyetleri çok zor olacaktır. Dolayısıyla da bu faaliyetler için çok fazla zaman harcanacaktır. RFC değerinin düşük olması istenmekle birlikte, geliştirilen bu yazılımın RFC değerinin yüksek olması anlaşılabilirlik, dayanıklılık ve test edilebilirlik düzeyini fazlasıyla düşürmektedir.

LCOM metriği için;

Otomatik değerlendirme - El ile değerlendirme

Kural 1

Eğer LCOM=0 ise;

Çıkarım 1

Geliştirilen yazılımın bu metrik değeri 0 (sıfır) dır. Yani, yazılımın uyumluluk, yeniden kullanılabilirlik, verimlilik ve karmaşıklık düzeyi hakkında yapılacak

yorumlar anlamsız olacaktır. Buna ilaveten, bakım-onarım ve test faaliyetleri için de zaman ve iş gücü harcanmayacaktır.

El ile değerlendirme
Kural 2
Eğer (ort6>=0,000001) and (ort6<strtoint(ComboBox11.Text)) ise;
Çıkarım 2
Elde edilen değer, belirlenen aralığın ilk değerinden de düşüktür. Yani, geliştirilen yazılımın uyumluluk, yeniden kullanılabilirlik, verimlilik ve karmaşıklık düzeyi belirlenen aralıkta değildir. Buna ilaveten, bakım-onarım ve test faaliyetleri için çok da fazla zaman harcanmayacaktır. LCOM değerinin geliştiricilerin yazılımlar için düşük olması beklenir. Ancak, burada istenen belirlenen aralıklar arasında olmasıdır.

Otomatik değerlendirme
Kural 2
Eğer (LCOM>=0) VE (LCOM<=25) ise;
El ile değerlendirme
Kural 3
Eğer (LCOM>=strtoint(ComboBox11.Text)) VE
(LCOM<=strtoint(ComboBox14.Text)) ise;
Otomatik değerlendirme
Çıkarım 2
El ile değerlendirme
Çıkarım 3
Bu değere bakılırsa, kabul edilen aralıktadır. Bu da, geliştirilen yazılımın uyumluluk düzeyinin, yeniden kullanılabilirlik ve verimlilik derecesinin yüksek, karmaşıklık düzeyinin de düşük olduğu anlamına gelir. Dolayısıyla, geliştirme süreci esnasında hata olma ihtimali de çok düşük olacaktır. Yani, yazılımın geliştirilmesi ve bakım-onarım-test faaliyetleri için az zaman ve iş gücü harcanacaktır. LCOM değerinin, geliştirilen yazılımlar için düşük olması beklenir.

Otomatik değerlendirme
Kural 4
Eğer (LCOM>25) ise;
El ile değerlendirme
Kural 4
Eğer (LCOM>strtoint(ComboBox14.Text)) ise;
Çıkarım 4
Bu değere bakılırsa, kabul edilen aralığın üstündedir. Bu da, geliştirilen yazılımın uyumluluk düzeyinin, yeniden kullanılabilirlik ve verimlilik derecesinin düşük, karmaşıklık düzeyinin de yüksek olduğu anlamına gelir. Dolayısıyla, geliştirme süreci esnasında hata olma ihtimali de yüksek olacaktır. Yani, yazılımın geliştirilmesi ve bakım-onarım-test faaliyetleri için çok fazla zaman ve iş gücü harcanacaktır. LCOM değerinin, geliştirilen yazılımlar için düşük olması beklenir.

Raporlama Ünitesi

Uzman değerlendirme modülü'nün (UDM) yapısı içerisindeki uzman sistem ünitesinin elde ettiği sonuçların kullanıcılara iletilmek üzere raporlandığı bölümdür. Raporun doğruluğu geri bildirim için önemlidir. UDM, Chidamber ve Kemerer'in daha önce elde edilen metrik bulgularının değerlendirmeleri ile ilgili kapsamlı bir rapor verecek şekilde geliştirilmiştir.

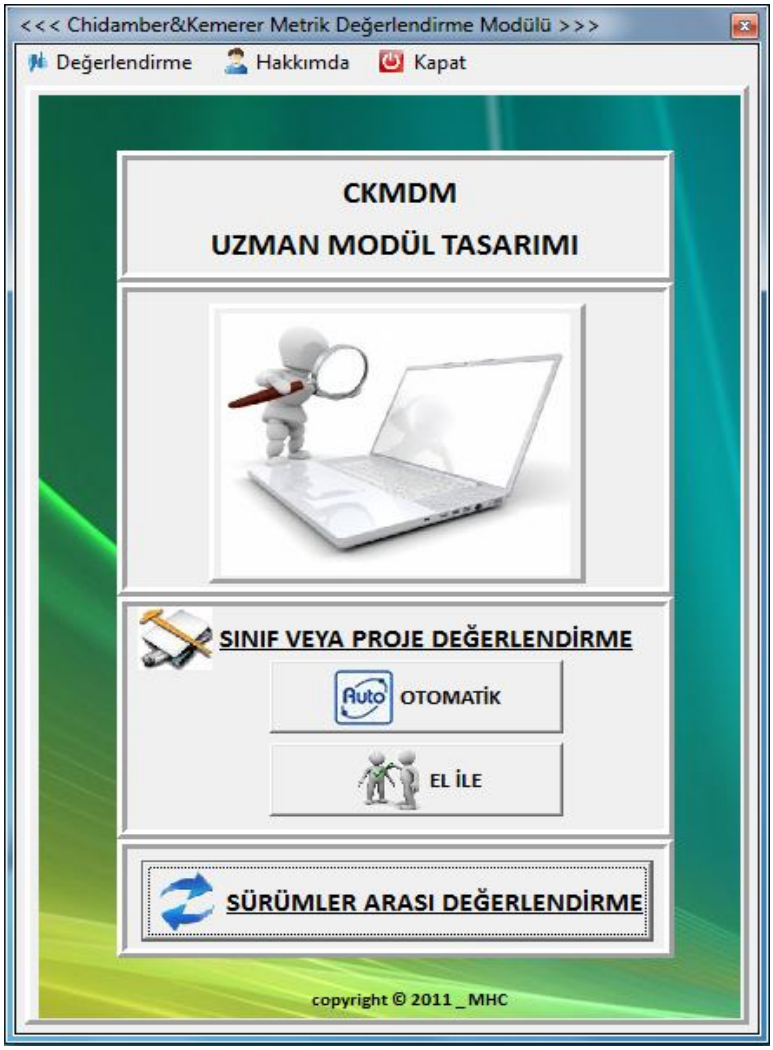
4.5. CKMDM Uzman Modülün Uygulanması

Nesne yönelimli java sınıf dosyalarının veya .jar dosyalarının metrik değerlerini kullanıcıya sunan "Metrics" programını kullanarak elde edilen bulguları "Sınıf veya Proje Değerlendirme" ve "Sürümler Arası Değerlendirme" olmak üzere iki durumda değerlendiren bir uzman modül geliştirilmiştir. "Sınıf veya Proje Değerlendirme" başlığı altında hem otomatik hem de el ile değerlendirme yapılabilmektedir. Sözkonusu uzman modül görsel programlama dili olan Delphi 7.0 ile geliştirilmiştir. CKMDM yazılımı, "Anasayfa", "Hakkımda", "Otomatik Değerlendirme", "El ile Değerlendirme", "Sürümler Arası Değerlendirme", "Tablonun Tümünü Gösterme", "Değerlendirme Sonuçları", "Grafiğe Dökme" ve "Raporlama" olmak üzere dokuz

formdan oluşmaktadır. Bu bölümde program hakkında genel bilgi verilmekle birlikte sözkonusu programın formları, menüleri, uygulama adımları ve işleyişi ekran görüntüleri ile desteklenerek açıklanmıştır.

Form 1. Anasayfa

Programın anasayfası iki şekilde hazırlanmıştır. Birincisi, kullanım bakımından çok etkili olması nedeniyle menüleri, ikincisi ise düğmeları içermektedir. Buna ilaveten sözkonusu sayfada, proje sahibi hakkında bilgi sunan "Hakkımda" menüsü bulunmaktadır.



Şekil 4.5. CKMDM anasayfa görünümü

Form 2. Hakkımda

Bu sayfada, proje sahibi hakkında kısa bilgiler bulunmaktadır.

Form 3. Otomatik Değerlendirme

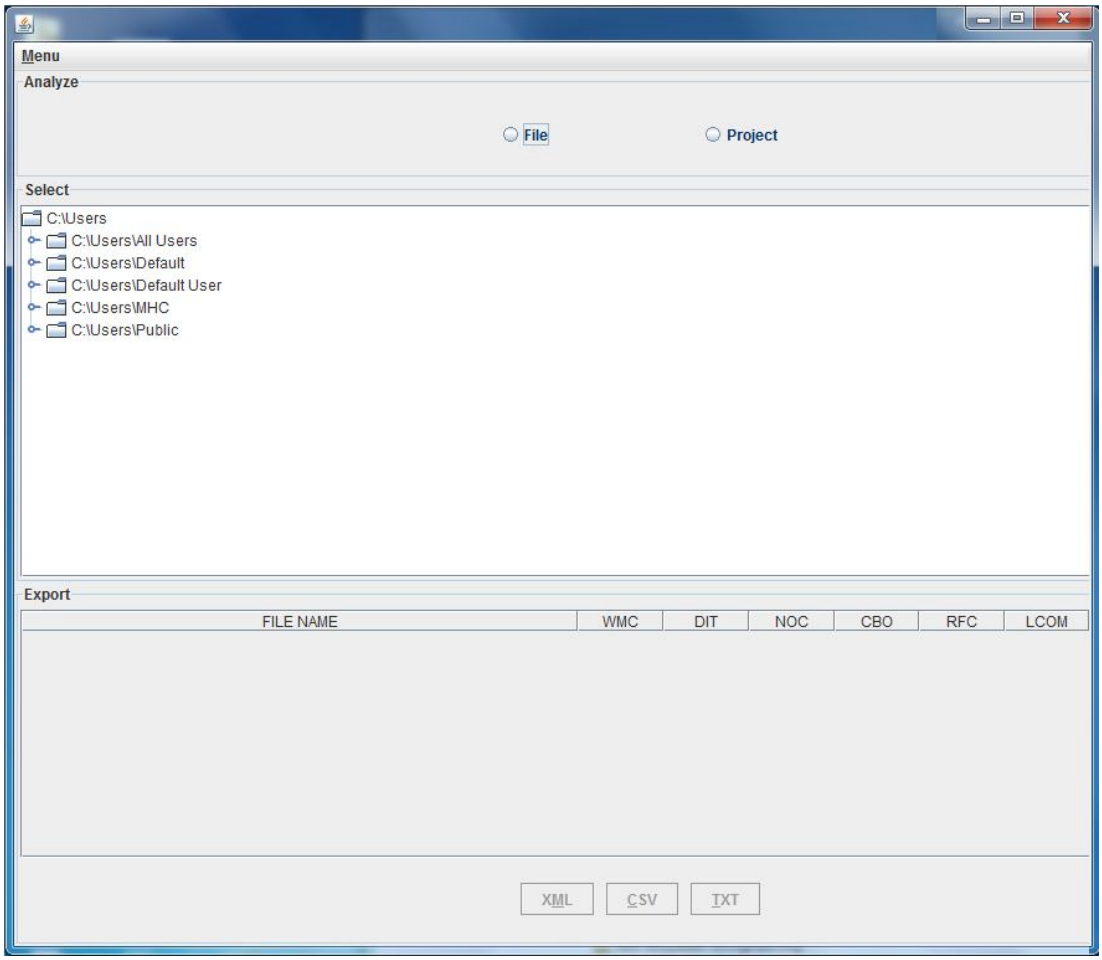
Otomatik değerlendirme sayfası; "CKMetrics", "Gözat", "Temizle", "Değerlendir", "Tabloyu Göster" ve "Kapat" olmak üzere yedi düğmeden oluşmaktadır. Her birinin görevleri aşağıda kısaca açıklanmıştır.

Şekil 4.6. "Otomatik Değerlendirme" sayfasından bir görünüm

a. "CKmetrics" Düğmesi

Bu düğme tıklanarak "Metrics" ölçüm programı açılmaktadır. Bu program, nesne yönelimli java sınıf dosyalarının veya .jar dosyalarının metrik değerlerini kullanıcıya sunmaktadır. Bu çalışmada CKMetrics programının sadece Chidamber ve Kemerer metrik bulgularından yararlanılmıştır. Sözkonusu programın kullanımı ve özellikleri şöyledir;

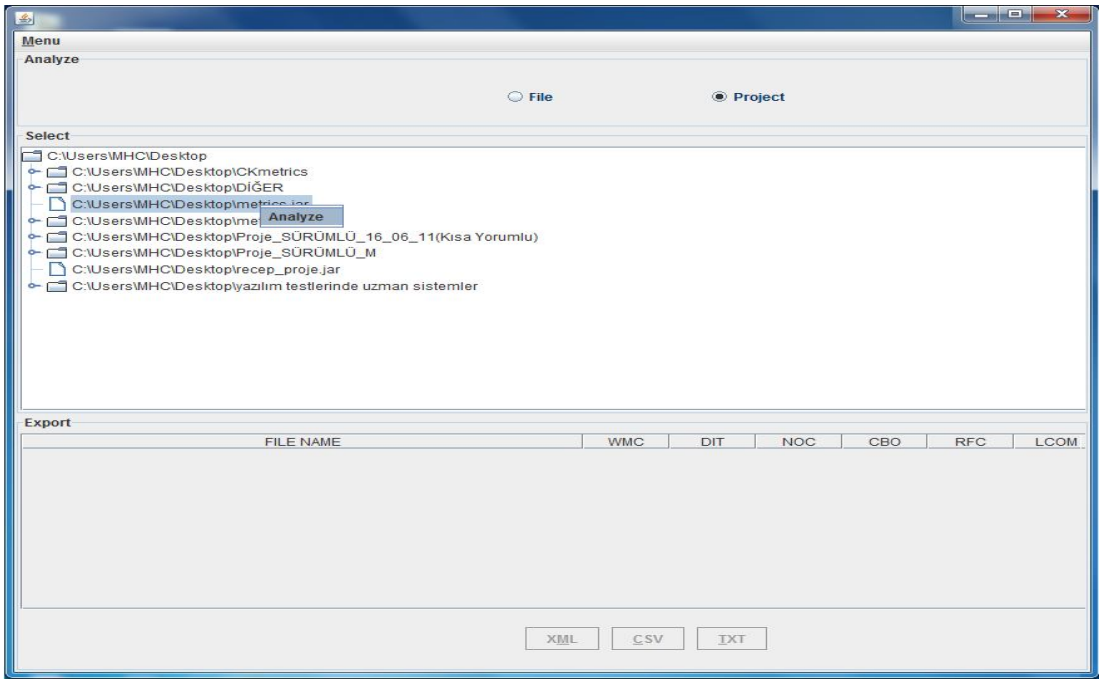
Öncelikle, bu programın (metrics.jar) çalıştırılabilmesi için bilgisayarlarda en azından java çalışma anı ortamının (JRE) kurulu olması gerekmektedir. Jar dosyası, üzerine çift tıklanıldığında doğrudan çalıştırılabilir şekilde oluşturulmuştur.



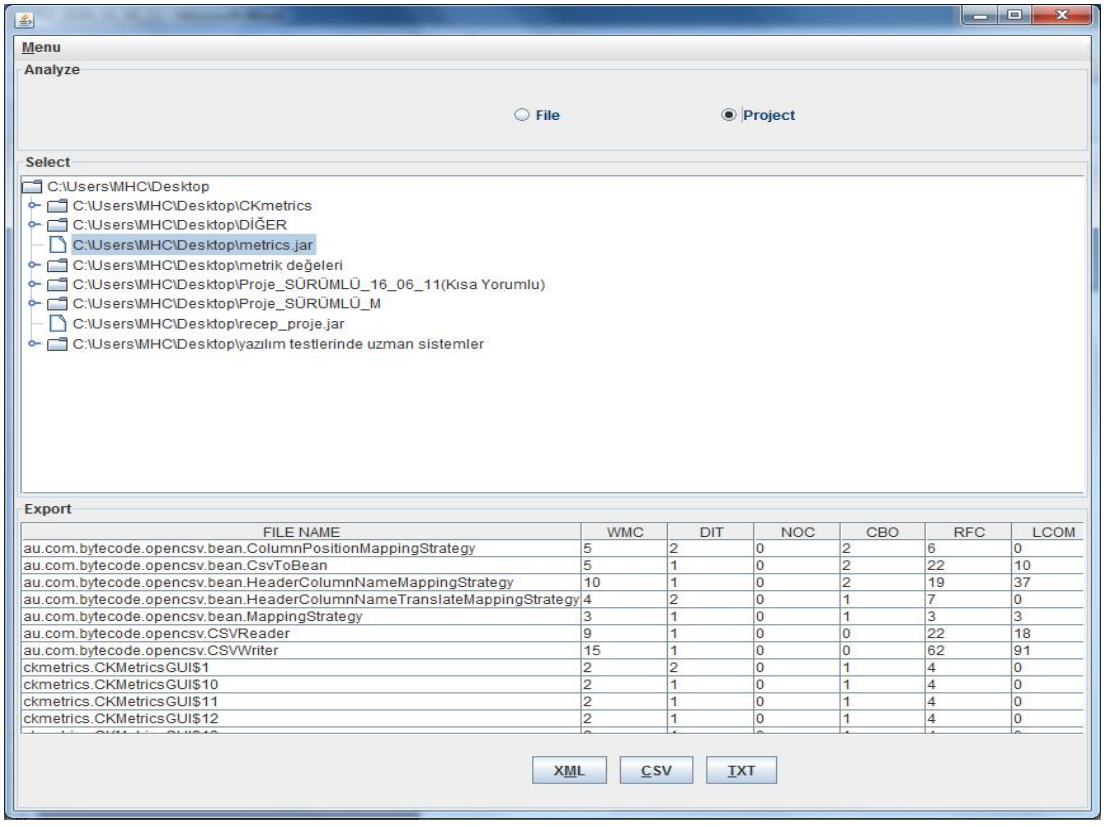
Şekil 4.7. "Metrics" programının anasayfa görünümü

Bu programda üç ana alan bulunmaktadır. Bunlardan ilki olan "Analyze" alanından "File" düğmesine tıklanarak tek tek sınıf (class) dosyaları üzerinde ölçüm yapılmasını, "Project" düğmesine tıklanarak ise bir jar dosyası olarak paketlenmiş birden fazla class dosyası üzerinde ölçüm yapılmasını sağlar. Bu adımda amaca uygun seçenek belirlenerek uygulama başlatılmaktadır.

Programın mevcut sürümünde, ölçülecek dosyanın Türkçe karakterler ve boşluk içermeyen bir dizinde olması zorunludur. Ölçülecek dosya seçilip açıldıktan sonra aşağıdaki görünüm elde edilir. Bu adımda seçilen dosya, bu aşamada programın ikinci ana alanı olan "Select" kısmında yer almaktadır. İstenilen bulguları elde etmek için belirlenen dosya üzerinde sağ tıklanarak "Analyze" seçeneği seçilmelidir.



Şekil 4.8. "Analyze" düğmesi tıklanarak ölçümün başlatılması



Şekil 4.9. Ölçüm sonuçlarının elde edilmesi ve incelenmesi

Ölçüm sonuçları programın üçüncü ana alanı olan “Export” kısmında yer almaktadır. Bu sonuçlar incelenebilir ve bu kısmın en altındaki ilgili düğmelere basarak sonuçlar istenilen biçimde kaydedilebilir. Geliştirilen bu uzman modülde sadece “.txt” uzantı ile kaydedilen dosyalar değerlendirilebilecektir.

b. “Gözat” Düğmesi

Bu düğme aracılığıyla, daha önce “Metrics” programı kullanılarak “.txt” olarak kaydedilen dosya tabloya aktarılmaktadır. Arka planda (kod kısmında), hem sadece “.txt” uzantılı dosyaları almak hem de “.txt” uzantılı olup bulguları içermeyen dosyaları almamak için bazı kontroller mevcuttur.

c. “Temizle” Düğmesi

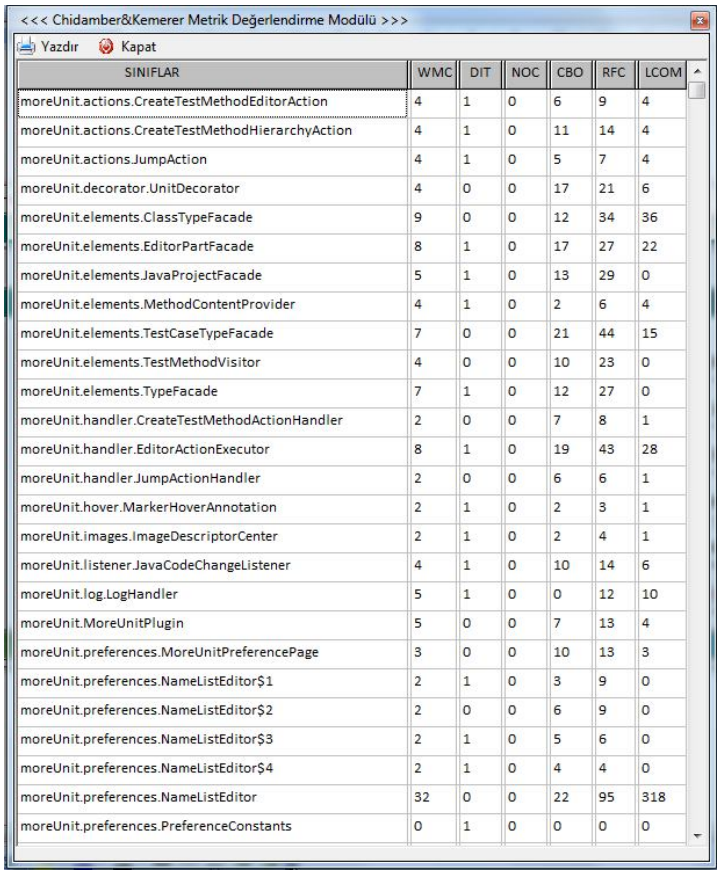
Bu düğme, daha önce “Gözat” düğmesi ile aktarılan bulguların temizlenmesi veya silinmesi görevini üstlenmektedir.

d. “Değerlendir” Düğmesi

Bu düğme ile, Chidamber ve Kemerer’ın literatürde kabul gören ve kullanılan muhtelif metrik aralıklarının ortalamasından yararlanılarak otomatik değerlendirme yapılmaktadır. Değerlendirilecek olan bulgular belirtilen aralıklar dahilinde ise, olumlu; aksi takdirde olumsuz yorumlar elde edilmektedir.

e. “Tabloyu Göster” Düğmesi

Bu düğme, Şekil 4.10’da da görüldüğü üzere formlarda bulunan küçük tabloların büyütülmüş halini gösterir. Dolayısıyla, kullanıcı elde edilen bulguları çok daha rahat analiz etme şansını bulur.



SINIFLAR	WMC	DIT	NOC	CBO	RFC	LCOM
moreUnit.actions.CreateTestMethodEditorAction	4	1	0	6	9	4
moreUnit.actions.CreateTestMethodHierarchyAction	4	1	0	11	14	4
moreUnit.actions.JumpAction	4	1	0	5	7	4
moreUnit.decorator.UnitDecorator	4	0	0	17	21	6
moreUnit.elements.ClassTypeFacade	9	0	0	12	34	36
moreUnit.elements.EditorPartFacade	8	1	0	17	27	22
moreUnit.elements.JavaProjectFacade	5	1	0	13	29	0
moreUnit.elements.MethodContentProvider	4	1	0	2	6	4
moreUnit.elements.TestCaseTypeFacade	7	0	0	21	44	15
moreUnit.elements.TestMethodVisitor	4	0	0	10	23	0
moreUnit.elements.TypeFacade	7	1	0	12	27	0
moreUnit.handler.CreateTestMethodActionHandler	2	0	0	7	8	1
moreUnit.handler.EditorActionExecutor	8	1	0	19	43	28
moreUnit.handler.JumpActionHandler	2	0	0	6	6	1
moreUnit.hover.MarkerHoverAnnotation	2	1	0	2	3	1
moreUnit.images.ImageDescriptorCenter	2	1	0	2	4	1
moreUnit.listener.JavaCodeChangeListener	4	1	0	10	14	6
moreUnit.log.LogHandler	5	1	0	0	12	10
moreUnit.MoreUnitPlugin	5	0	0	7	13	4
moreUnit.preferences.MoreUnitPreferencePage	3	0	0	10	13	3
moreUnit.preferences.NameListEditor$1	2	1	0	3	9	0
moreUnit.preferences.NameListEditor$2	2	0	0	6	9	0
moreUnit.preferences.NameListEditor$3	2	1	0	5	6	0
moreUnit.preferences.NameListEditor$4	2	1	0	4	4	0
moreUnit.preferences.NameListEditor	32	0	0	22	95	318
moreUnit.preferences.PreferenceConstants	0	1	0	0	0	0

Şekil 4.10. Elde edilen bulguların büyük formatta gösterimi

f. “Kapat” Düğmesi

Açılan formun kapatılmasını sağlar.

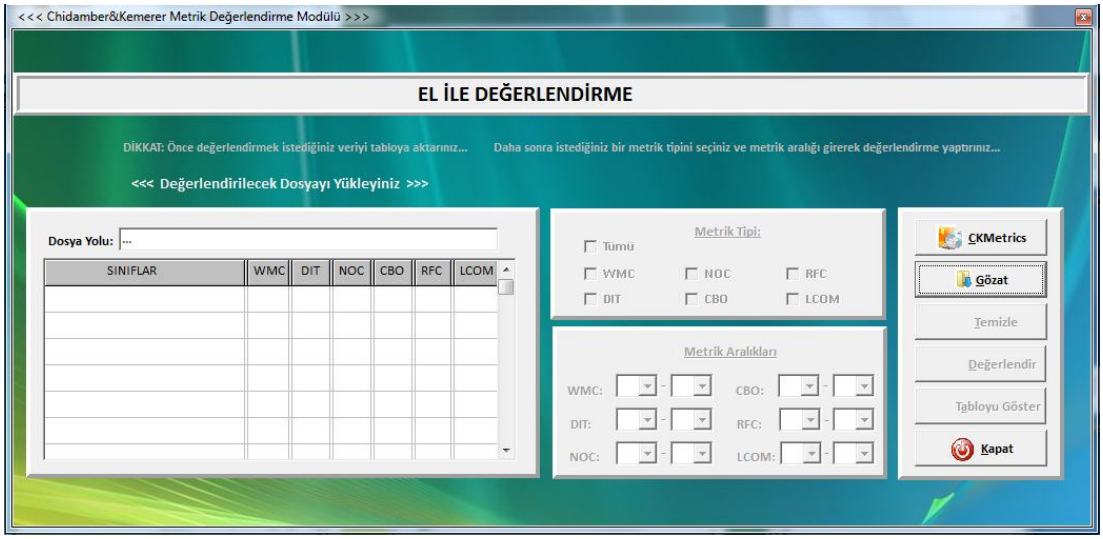
Form 4. El ile Değerlendirme

Bu sayfa “Otomatik Değerlendirme” sayfasıyla benzer özelliklere sahip olmakla birlikte bazı farklılıklar da içermektedir. Tekrar olmaması sebebiyle sözkonusu bölüm olan “El ile Değerlendirme” sayfasının sadece farklı özelliklerinden bahsedilecektir.

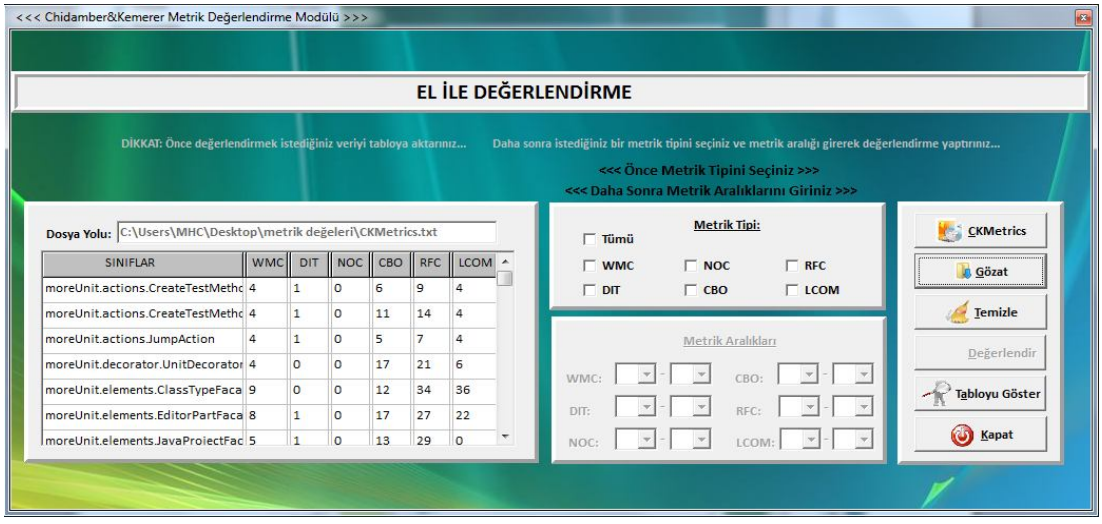
Öncelikle, “Otomatik Değerlendirme” de daha öncede belirtildiği üzere literatürde kabul görmüş aralıklar baz alınarak otomatik bir şekilde değerlendirme yapılmaktadır. “El ile Değerlendirme” de ise; metrik tipleri ve metrik aralıkları kullanıcının isteğine göre belirlenip bu tip ve aralıklar dikkate alınarak

değerlendirme yapılmaktadır. Yanıp sönen uyarılar ve kontrollerle programın kullanılabilirliğinin arttırılması amaçlanmıştır.

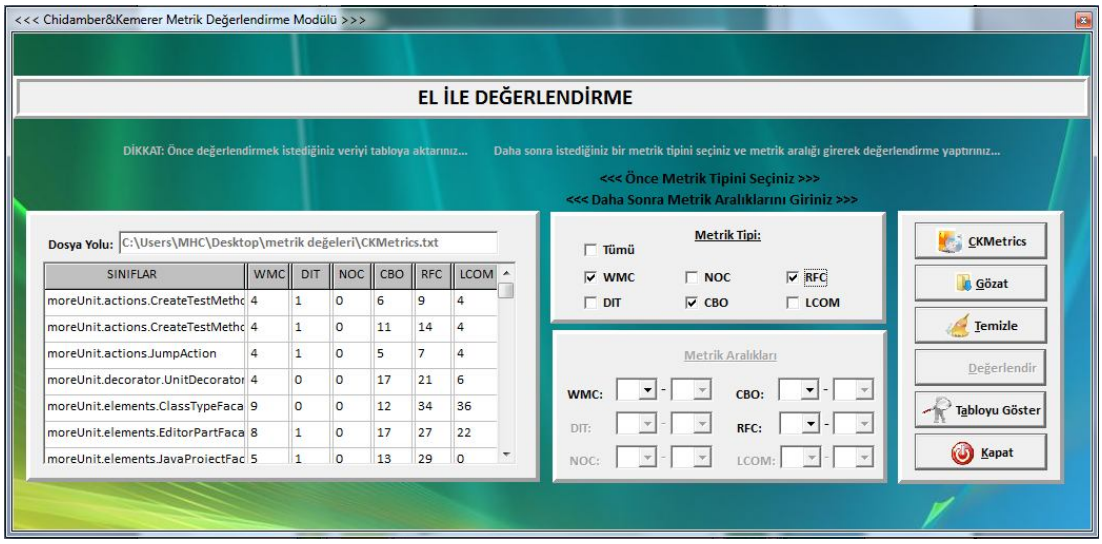
"Gözat" düğmesi ile tabloya bulgular aktarılmadan önce "Değerlendirilecek Dosyayı Yükleyiniz" yazısı belirmektedir. Bulgular tabloya; "Otomatik Değerlendirme" sayfasında olduğu gibi, sadece ".txt" dosyaları üzerinde işlem yapabilme kontrolleri sağlanarak aktarıldıktan sonra "Önce Metrik Tipini Seçiniz" ve "Daha Sonra Metrik Aralıklarını Giriniz" yazıları yanıp sönerek kullanıcı uyarılmaktadır. Bahsedilen özellikler Şekil 4.11, Şekil 4.12, Şekil 4.13 ve Şekil 4.14'te gösterilmektedir.



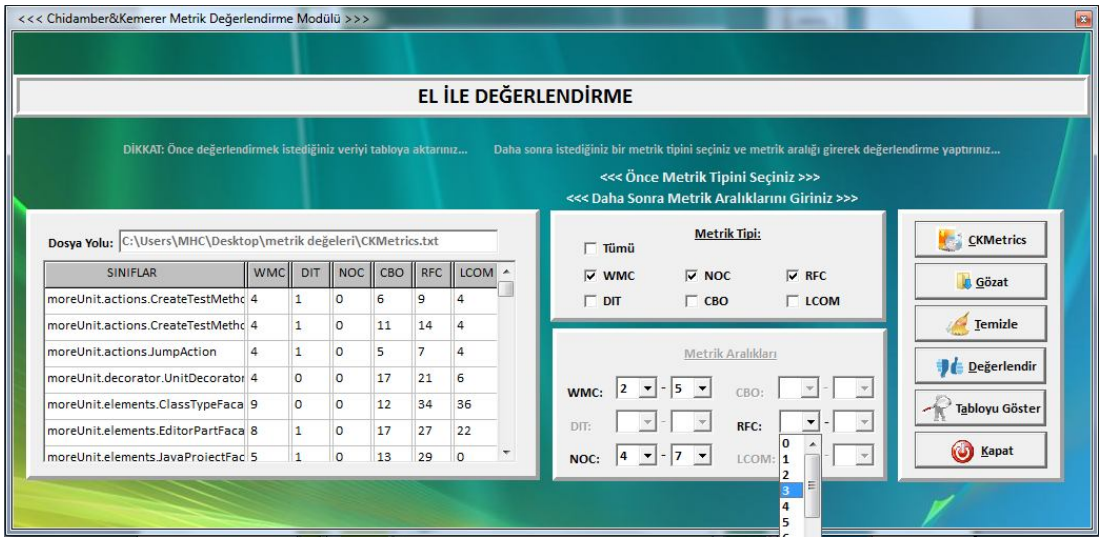
Şekil 4.11. "El ile Değerlendirme" sayfasından bir görünüm



Şekil 4.12. "Gözat" düğmesi tıklanarak bulguların tabloya aktarılması



Şekil 4.13. Metrik tipinin belirlenmesi



Şekil 4.14. Metrik aralığının belirlenmesi

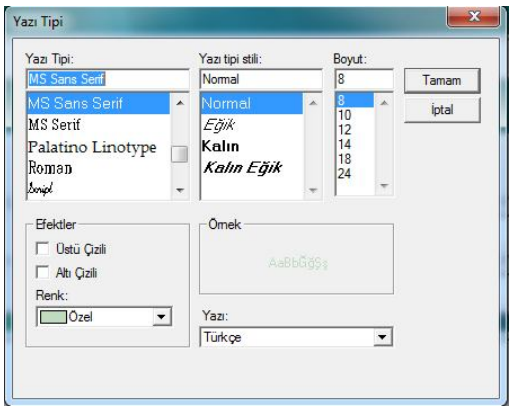
Form 5. Değerlendirme Sonuçları

Değerlendirme sonuçları formlarda bulunan "Değerlendir" düğmesi aracılığıyla gösterilmektedir. Aşağıda önce "Otomatik Değerlendirme" sonuçları için, daha sonra da "El ile Değerlendirme" sonuçları için birer örnek görünüm verilmiştir.

Şekil 4.15. "Değerlendirme Sonuçları" sayfasından bir görünüm

a. "Metni Düzenle" Düğmesi

Bu düğme, elde edilen değerlendirme sonuçlarını kaydetmeden önce Şekil 4.16'daki özellikler kullanılarak yazı formatını düzenleme imkanı sağlar.



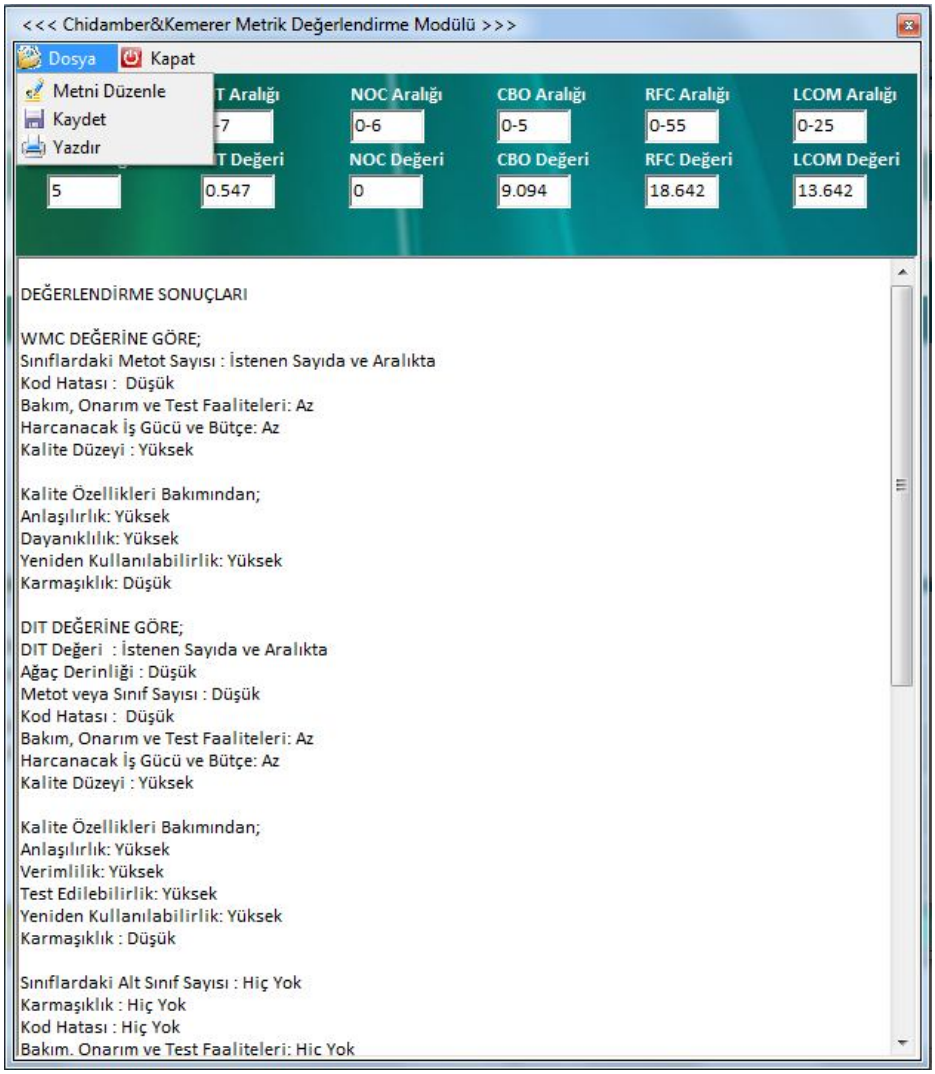
Şekil 4.16. Yazı tipi düzenleme ekranı

b. "Kaydet" Düğmesi

Elde edilen değerlendirme sonuçlarını veya raporları .doc uzantılı (word belgesi) halde kaydetme imkanı sağlar.

c. “Rapor Al” Düğmesi

Sözkonusu düğme, kullanıcının değerlendirme sonuçlarını tam olarak görmek istediği durumlarda rapor alma imkanı sağlar. Aynı zamanda elde edilen bu rapor, Şekil 4.17’de görüldüğü üzere, “Dosya” menüsü kullanılarak düzenlenebilir, kaydedilebilir ve yazdırılabilir özelliktedir.



Şekil 4.17. Raporlama sayfası

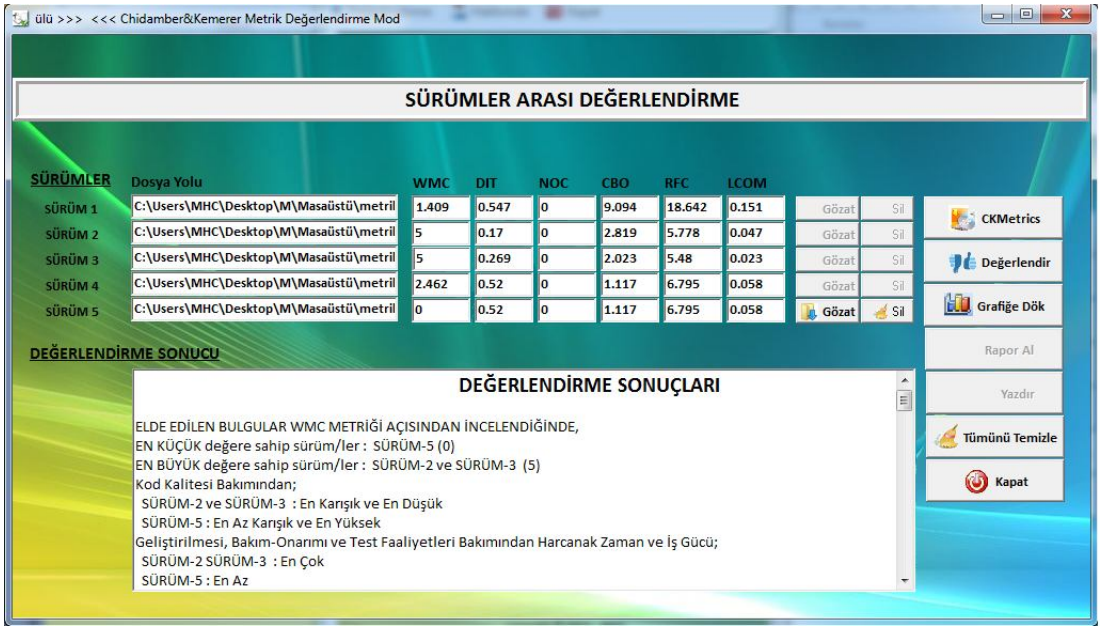
d. “Yazdır” Düğmesi

Elde edilen değerlendirme sonuçları kullanıcı tarafından bu düğme tıklanarak yazdırılabilir.

Form 6. Sürümler Arası Değerlendirme

Bu yöntemde, daha öncede açıklandığı gibi, aynı metrik grubunun ölçülecek olan yazılımın sürümleri üzerinde uygulanarak değerlendirme yapılmıaktadır. Başka bir ifadeyle, önceki yazılım sürüm/lerin metrik değerleri ve kusur bilgileri kullanılarak bir model ortaya çıkarılmaktadır. Ardından yeni sürümdeki metrik değerleri ve kusur bilgileri ile karşılaştırılarak kalite düzeyi yorumlanmaktadır. Şekil 4.18'de örnek "Sürümler Arası Değerlendirme" sayfası gösterilmektedir.

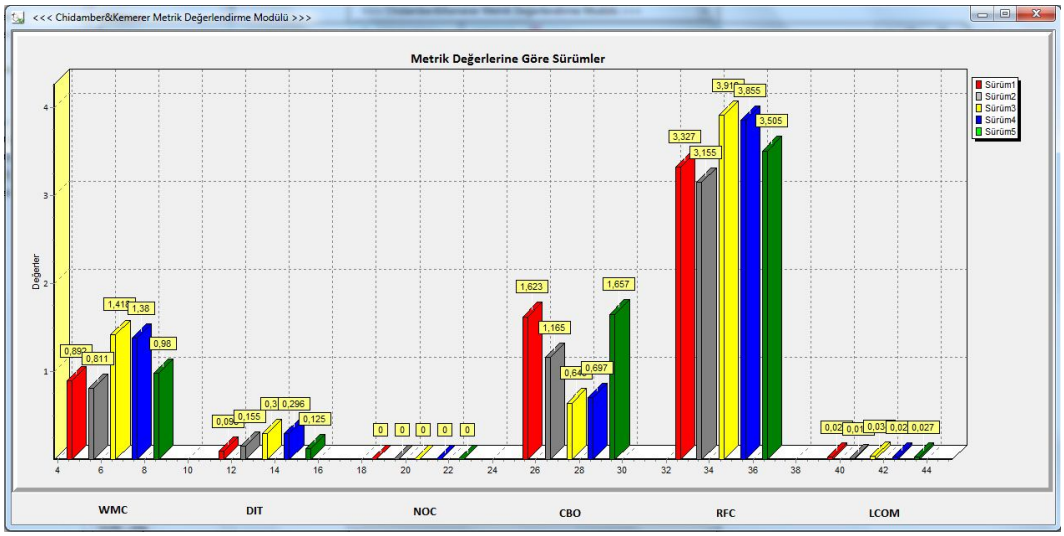
Bu form aracılığıyla "Metrics" programı ile elde edilen sürüm metrik değerleri önce ilgili alanlara aktarılır. Herhangi bir yanlışlık durumunda silinebilir. Aktarılan bu metrik değerleri kullanıcı isterse grafiğe aktarılabilir. Daha sonra değerlendirme yapılır ve değerlendirme sonuçları rapor edilir.



Şekil 4.18. "Sürümler Arası Değerlendirme" sayfasından bir görünüm

Form 7. Grafik

CKMetrics programı ile elde edilen verilerin sürümlere göre metrik değerlerini grafiksel olarak göstermek amacıyla kullanılır. Şekil 4.19'da örnek bir görünüm mevcuttur.



Şekil 4.19. Metrik değerlerine göre sürümlerin grafiksel gösterimi

Form 8. Raporlama

Bu form, daha öncede açıklandığı üzere, kullanıcının değerlendirme sonuçlarını tam olarak görmek istediği durumlarda kullanılır. Aynı zamanda elde edilen bu rapor, Şekil 4.17'de de görüldüğü üzere "Dosya" menüsü kullanılarak elde edilen metin veya rapor düzenlenebilir, kaydedilebilir ve yazdırılabilir özelliktedir.

Form 9. Tablonun Tümünü Gösterme

Bu düğme, Şekil 4.10'da görüldüğü gibi formlarda bulunan küçük tabloların büyütülmüş halini gösterir. Dolayısıyla, kullanıcı elde edilen bulguları çok daha rahat analiz etme şansını bulur.

5. SONUÇ VE ÖNERİLER

Bu tez çalışmasında, özellikle nesne yönelimli yazılım geliştirme sürecindeki test faaliyetleri (test modelleri, test teknikleri ve test seviyeleri) araştırılmıştır. Bu konularda geniş bir literatür taraması yapılarak konuyla ilgili eksiklikler giderilmeye çalışılmıştır. Ayrıca, nesne yönelimli yazılımlar temelde sınıflandırma mantığıyla geliştirildiği için, sözkonusu yazılımların sınıf metriklerini değerlendiren bir uzman modül geliştirilmiştir. Geliştirilen uzman modül aracılığı ile .jar uzantılı bir dosyanın, proje/lerin sürümlerinin metrik bulguları hatasız bir şekilde değerlendirilebilmektedir. Metrikler literatürde yaygın olarak kullanılan Chidamber ve Kemerer'in metrik kümesidir. Geliştirilen uzman modülün değerlendirilmesi için uzman sistemlerden yararlanılmıştır. Sözkonusu modülün "Sınıf veya Proje Değerlendirilmesi" kısmında uzman sistemlerin bilgi sunma kurallarından olan "Eğer-O halde" kuralı kullanılmıştır. Değerlendirmede, "*özelliğe bağlı metrik ölçümü ve yorumlanması"* yöntemine başvurulmuştur. "Sürümler Arası Değerlendirme" kısmında ise, metrik değerlendirme yöntemlerinden olan "*daha önceki sürüm veya sürümler ile yeni geliştirilen sürümün metrik bulguları karşılaştırılması"* yöntemine başvurularak geliştirilen yazılımın kalitesi hakkında yorumlar yapılabilmektedir.

Ancak, yazılım kalitesinin istenen düzeyde arttırılabilmesi için her şeyden önce mevcut kalitenin doğru biçimde ölçülebilmesi gerekmektedir. Bunun için de doğru ve açık bir şekilde belirlenmiş test stratejileri olmalıdır ve belirlenen bu test stratejileri eksiksiz bir şekilde uygulanmalıdır. Ayrıca, yazılım test faaliyetleri içerisinde, sözkonusu faaliyetleri daha iyi yönetebilmek, geliştirilen yazılımlar hakkında daha anlamlı yorumlar yapabilmek ve daha başarılı sonuçlar elde etmek amacıyla test metrikleri kullanılmalıdır. Sözkonusu metriklerin ne anlama geldiklerini bilmek ve ona göre geliştirilen yazılımların kaliteleri hakkında değerlendirme yapmak gerekir. Literatürde bu konularda bir takım eksiklikler bulunmaktadır. Buna ilaveten, bir otomasyon aracılığıyla metrikleri değerlendiren yazılım araçları olmakla birlikte, sözkonusu araçların yetenekleri konusunda da eksiklikler bulunmaktadır. Bu araçların çeşitlendirilmesi ve yeteneklerinin

arttırılması gerekmektedir. Örneğin, araçların sadece rakam göstermemesi, ayrıca metriklerin görselleştirmesi veya değerlendirilmesi de analizcilerin işini kolaylaştıracaktır. Metrik araçları yeni metriklerin kolayca uyarlanabilmesini desteklemeli ve geliştiricilerin kendi özel metrik değerlerini ayarlamasına izin vermelidir. Burada unutulmaması gereken önemli bir husus da, belirlenen metriklerin tek başına değil, bir arada değerlendirilmesi gerekliliğidir. Verilen kararlar bir metriği olumlu yönde etkilerken diğer bir metriği daha kötü hale getirebilir. Dolayısıyla birden fazla metriğin bir arada kullanılması gerekir. Bütün bunlara ilaveten belirlenen metriklerin ürettiği değerler ile aynı amaca hizmet eden başka metriklerin ürettiği değerler arasındaki ilişki ölçülerek yapılan değerlendirme yöntemi de kullanılabilir.

Sonuç olarak, bu çalışma ile nesne yönelimli test faaliyetleri konusunu açıklığa kavuşturarak nesne yönelimli tasarım metrik değerlendirilmesi ve bunun sonucu olarak da yazılım kalite değerlendirilmesi konularında bir otomasyon geliştirilmiştir. Geliştirilen yazılım aracı ile literatüre katkı sağlanması amaçlanmıştır.

KAYNAKLAR

1. Gürbüz, A., "Yazılım Test Mühendisliği", **Papatya Yayıncılık Eğitim**, İstanbul, 31,34 (2010).

2. Tiftik, N., Öztarak, H., Ercek, G. ve Özgün, S., "Sistem/yazılım geliştirme sürecinde doğrulama faaliyetleri", **III.Ulusal Yazılım Mühendisliği Sempozyumu**, Ankara, 1-2 (2007).

3. Ertemel, H.Ö., Selçuk, Y.E, Kalıpsız O., "Nesneye yönelik sistemler için bir uyum ölçütü önerisi: Comıas", **IV. Ulusal Yazılım Mühendisliği Sempozyumu,** Ankara, 1 (2009).

4. Mustafa, K. ve Khan, R.A., "Software Testing: Concept and Practices", India, **Lucknow**, 5-21, 227-228 (2007).

5. Naik, K. ve Tripathy, P., "Software Testing and Quality Assurance", **New Jersey**, USA, 7-10 (2008)

6. Demir, D., "Endüstride yazılım testi ve kalite güvencesi etkinlikleri", **I. Ulusal Yazılım Mühendisliği Sempozyumu,** İzmir, 23-25 Ekim (2003).

7. Tuna, O., "Yazılım geliştirme süreci ve mimari gösterime dayalı yazılım testi", Yüksek Lisans Tezi, **Dokuz Eylül Üniversitesi Fen Bilimleri Enstitüsü**, İzmir, 1 (2005).

8. İnternet: IEEE Standards Associatiın, "Glossary of Software Engineering Terminology", http://standards.ieee.org/findstds/standard/610.12-1990.html, (1990).

9. Smith, M.D., Robson, D.J, "Object oriented programming the problems of validation", **Software Maintenance, 1990. (ICSM '90) Proceedings. IEEE International Conference**, CH2921-5/90/0000/0272, (1990).

10. ANSI Std-1991, "Standart glossary of software engineering terminology (ANSI)", **The institute of electrical and electronics engineers inc.**, (1991).

11. Myers, G., "The Art of Software Testing", **Wiley Interscience**, ISBN: 471043281, (1979).

12. Gill, A., "Debugging haskell by observing intermediate data structures", **In Proceedings of the 4th Haskell Workshop**, Technical Report of the University of Nottingham, (2000).

13. Lewis, E. W., "Software Testing and Continuous Quality Improvement, 2nd ed.", **A CRC Pres Company**, USA, 10, 60, 117-127, 129-155, 183-185, 230-256 (2005).

14. Bourque, P. ve Dupuis, R., "Guide To The Software Engineering Body Of Knowledge", **The Institute of Electrical and Electronics Engineers**, USA, 5 (2004).

15. Davis, S. ve arkadaşları, "Software Testing Engineering With IBM Rational Functional Tester", **Pearson Plc**, USA, 54 (2010).

16. Muller, T., Graham, D., Friedenberg, D. ve Veendendal, E., "International Software Testing Qualifications Board (ISTQB)", **Foundation Level Syllabus**, USA, 10 (2007).

17. Sommerville, I., "Software Engineering", ISBN:0-201-39815-X, **Addison Wesley**, (2001).

18. Garrido, J.M., "Object Oriented Programming: From Problem Solving to Java", **Charles River Media,** USA, 19-23, 240-241 (2003).

19. Güngören, B., "UML ile Nesne Tabanlı Çözümleme ve Tasarım", **Seçkin Yayıncılık,** Ankara, 87(2005).

20. Goodman, P., "The Practical Implementation of Software Metrics", **McGraw-Hill**, New York, USA, 136 (1993).

21. Schulmeyer, G. G. ve McManus I., J., "The Handbook of Software Quality Assurance, 3rd Edition", **Prentice Hall PTR,** USA, 404 (1998).

22. Pressman, R. S., "Software Engineering: A Practitioner's Approach", 6th Ed., **Mc Graw Hill,** Singapore**,** 480-481, 492 (2005).

23. Pusala, R., "Operational Excellence Through Efficient Software Testing Metrics", **EuroSTAR 2005: European Conference on Software Testing İnfosys**, Denmark, 2-3 (2005).

24. Yurga, T., "Nesne Yönelimli Yazılım Tasarımının Testedilebilirliğini Ölçmeye Yönelik Bir Model", **II.Ulusal Yazılım Mühendisliği Sempozyumu**, Ankara, 3 (2005).

25. Xie T., Huang H., Chen X., Mei H. and Yang F., "Object Oriented Software Metrics Technology - Technical Report", **Peking University**, (2001).

26. Binder V. R., "Software Testing, Verification and Reliability", Vol. 6, **RBSC Corporation, USA, 125-252** (1996).

27. İnternet: Testingfaqs.org "Defect Tracking Tools" http://www.testingfaqs.org/t-track.html (2010).

28. İnternet: SourceForge.net, "FindBugs™ - Find Bugs in Java Programs" http://findbugs.sourceforge.net/index.html (2008).

29. İnternet: Yıldız Teknik Üniversitesi Bilgisayar Mühendisliği, "Ölçüm Programı ve Kullanımı-Metrics", http://www.ce.yildiz.edu.tr/myindex.php?id=63 (2009).

30. İnternet: SourceForge.net, "PMD" http://pmd.sourceforge.net/ (2002).

31. İnternet: Coverlipse, http://coverlipse.sourceforge.net/index.php (2005).

32. İnternet: Checksytle, http://checkstyle.sourceforge.net/ (2001).

33. İnternet: SDMetrics, http://www.sdmetrics.com/ (2009).

34. İnternet: Coverity, "Coverity Analysis" http://www.coverity.com/ (2009).

35. Kung D. ve arkadaşları, "Developing an Object-Oriented Software Testing and Maintenance Environment", ***Communications of the ACM - Special issue on object-oriented experiences and future trends*** , Newyork USA, Vol:38, Issuse:10 (1995).

36. Pezz, M., Young, M., "Testing Object Oriented Software," icse, pp.739-740, **26th International Conference on Software Engineering** (ICSE'04), (2004).

37. Bahsir, İ. ve Goel, A.L, "Testing Object Oriented Software", **NewYork**, USA, 21 (2000).

38. Berard, E.V., "Essays on Object-Oriented Software Engineering", Volume 1, **Prentice Hall**, Englewood Cliffs, New Jersey, (1993).

39. Daşdemir, Y., "Delphi 2006 Programlama", **Türkmen Kitabevi**, İstanbul, 256 (2006).

40. Pacheco, X., "Herkes için Delphi 8", **Alfa Basım Yayın Dağıtım**, İstanbul, 93,111 (2004).

41. Booch, G., "Object-Oriented Analysis and Design With Applications", Second Edition, **Benjamin/Cummings Publishing Company**, Redwood City, California, (1994).

42. Hitz, M., and Montazeri, B., "Chidamber and Kemerer's metric suite: A measurement theory perspective", IEEE Transactions on Software Engineering, Vol. 4, pp.267-271, (1996).

43. O'regan, G., "A Pratical Approach to Software Quality", **Maple-Vail Book Manufacturing Group**, Newyork, 71-77 (2002).

44. Kızılören, T., "Java ve Java Teknolojileri", **KODLAB Yayın Dağıtım Yazılım ve Eğitim Hizmetleri San. ve Tic. Ltd. Şti**, İstanbul, 128-130 (2010).

45. Kalıpsız, O. ve arkadaşları, "Sistem Analizi ve Tasarımı: Nesneye Yönelik Modelleme", **Altan Matbaacılık Toros Dağıtım**, İstanbul, 107 (2006).

46. McGregor, J. D., and Harrold, M. J., "Toward a Testing Methodology for Object-oriented Software Systems." **Proc. Workshop on Object Oriented Software Engineering Practice**, February (1992).

47. Amstel, M.,v., Brand, M., v.,d., and Lange, C., "Metrics For Analyzing The Quality Of Model Transformations", **In Falcone, G., Gu´eh´eneuc, Y., Lange, C., Porkol´ab, Z., Sahraoui, H., eds.: Proceedings of the 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering**, Paphos, Cyprus, pp.41–51, (2008).

48. Thirugnanam, M. and Swathi J.N., "Quality Metrics Tool for Object Oriented Programming", **International Journal of Computer Theory and Engineering**, Vol. 2, No. 5, pp.1793-8201, (2010).

49. Kaur, J., P.,, Verma A. and Thapar, S., "Software Quality Metrics for Object-Oriented Environments", **Proceedings of National Conference on Challenges ve Opportunities in Information Technology (COIT-2007)**, RIMT-IET, Mandi Gobindgarh, pp.13-16, (2007).

50. Mosley, D.J. ve Posey B.A., "Just Enough Software Test Automation", **Prentice Hall PTR**, USA, 10 (2002).

51. Naish, L., "A Declarative Debugging Scheme", **Journal of Functional and Logic Programming**, 44 (1997).

52. Wallace, M., Chitil, O., Brehm, T., Runciman, C., "Multiple-view Tracing for Haskell: A New Hat", **In Preliminary Proceedings of the 2001 ACM Sıgplan Haskell Workshop**, Frenze, Italy, 151 (2001).

53. Mary Jean Harrold, John D. McGregor, Kevin J. Fitzpatrick: "Incremental Testing of Object-Oriented Class Structures", **Proc. Of 14th International Conference on Software Engineering**, USA, May 11-15, 68-80 (1992).

54. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C., "A Test Strategy for Object-Oriented Systems", **Proc. of Computer Software and Applications Conference**, pp. 239 - 244, Dallas Texas, August 9-11, IEEE Computer Society, (1995).

55. Şimşek, H., Özdemir, N., "Güvenlik-Kritik Sistemlerde Yazılım Birim Testleri", **III.Ulusal Yazılım Mühendisliği Sempozyumu**, Ankara, 27-30 Eylül (2007).

56. Wu, Y., "Software Design", **Fall 2000 Lecture Notes**, George Mason University, (2001).

57. Kılıç, E. ve Öztürk, S., "Büyük Ölçekli Yazılım Projelerinde Entegrasyon Testleri", **Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu (YKGS10)**, İstanbul, 5.Oturum (2010).

58. Rogers, R. O., "Acceptance testing vz. unit testing: A developer's perspective, XP/Agile Universe", **Lecture Notes in computer Science**, ISSN 0302-9743 (Print) 1611-3349 (Online) V.3134 (2004).

59. Canna, J., "Testing Fun? Really?", **IBM developer Works Java Zone**, (2001).

60. Miller, R., Collins, C., "Accepting Testing", **XP Universe Conference**, July, (2001).

61. İnternet: Software Testing, "Yazılımda Test Süreçleri", http://softwaretesting-tr.blogspot.com/ (2011).

62. IEEE 729/610.12- 1990: IEEE Std 610.12-1990 IEEE Standart Glossary of Software Engineering Terminology – Description, (1990).

63. Beizer, B., "Software Testing Techniques", **2nd Ed. Itp Media**, New York, Van Nostrand Reinhold, (1990).

64. Raishe, T., "Black Box Testing", **Lecture Notes of College of Engineering and Computer Science**, Florida Atlantic University, (1999).

65. Keytorc, "Certified Tester Foundation Level Syllabus", **International Software Testing Qualifications Board (ISTQB)**, ISTQB Report Versiyon 2010, 40 (2010).

66. İnternet: Buzzle.com, "Software Testing Technique" http://www.buzzle.com/articles/software-testing-techniques.html (2011).

67. Fagan, M. E, "Advanced in software inspection", **IEEE Transactions on Software Engineering**, Vol. SE-12, No.7, July (1986).

68. Midtgaard, J., "Control-Flow Analysis of Functional Programs", **BRICS Report Series: RS-07-18**, ISSN 0909-0878, December (2007).

69. Kaveh, P., "A framework of the use of information in software testing", **The Faculty Of The School Of Engineering and Applied Science Of The George Washington University in Partial Satisfaction of The Requirements for The Degree of Doctor of Science**, May 16, (2010).

70. Ertemel, H.Ö., "Nesneye yönelik yazılım geliştirmede kalite ölçütlerinin incelenmesi", Yüksek Lisans Tezi, **Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü**, İstanbul, 1 (2009).

71. Kitchenham, B. ve Pfleeger, S.L., "Software Quality: The Elusive Target", **IEEE Software**, vol. 13, no. 1, pp.12-21, January (1996).

72. Baldassari, B., Robach, C. ve du Bosquet, L., "Early Metrics for Object Oriented Designs", **IEEE Testability Assessment, 2004. IWoTA 2004. Proceedings. First International Workshop**, p.62-69, (2004).

73. Jorgensen M, "Software Quality Measurement" **Department of Informatics**, **University of Oslo**, Oslo, Norway, Vol:30, Issue:12, 907-912, September (1999).

74. Carey, D., "Software Quality Intrinsic, Subjective or Relational", **Software Engineering Notes**, 21(1):74–75 (1996).

75. Lanubile F, Visaggio, G., "Evaluating predicting quality models derived from software measures: Lessons learned", **Technical report CS-TR- 3606**, Department of Computer Science, University of Maryland, (1996).

76. Jetter, A., "Assessing Software Quality Attributes with Source Code Metrics", Diploma Tezi, **University of Zurich Department of Informatics**, Zurih, 13 (2006).

77. McCall, J. A., Richards, P. K., and Walters, G. F., "Factors in software quality", **Nat'l Tech.Information Service**, 1,2 and 3, (1977).

78. Boehm, B. W., Brown, J. R., Kapsar, H., Lipow, M., McLeod, G., and Merritt, M., "Characteristics of software quality", Vol 1 of TRW series on softwaretechnology, **North-Holland**, Amsterdam, Netherlands (1978).

79. Dromey, R. G., "A model for software product quality", **Software Engineering**, IEEE Transactions on, 21(2):146 – 162, (1995).

80. Bansiya, J. ve Davis, C. G., "A hierarchical model for object-oriented design quality assessment", **IEEE Transactions on Software Engineering**, vol. 28, no. 1, pp.4-17, January (2002).

81. Beklen, A., "İleri yazılım mühendisliği (ISO/IEC 9126)", **Maltepe Üniversitesi Fen Bilimleri Enstitüsü,** Ders Notu, 1-11 (2008).

82. Erdemir, U., Tekin, U. ve Buzluca, F. "Nesneye dayalı yazılım metrikleri ve yazılım kalitesi", **Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu (YKGS08)**, İstanbul, 4 (2008).

83. İnternet: RUP (2007). Rational Unified Process., "Types of Test", http://rup.hops-fp6.org/process/workflow/test/co_tytst.htm, June 17, (2007).

84. Amstel, V., M.F., Lange, C.F.J., Brand, M.G.J., "Metrics for analyzing the quality of model transformations", In Falcone, G., Gu´eh´eneuc, Y., Lange, C., Porkol´ab, Z., Sahraoui, H., eds.: **Proceedings of the 12th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering**, Paphos, Cyprus, 41–51 (2008).

85. Çatal, Ç. ve Diri, B., "Yazılım metriklerini kullanarak düşük kaliteli/yüksek kaliteli modüllerin otomatik tespiti", **Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu (YKGS08)**, İstanbul, 1, 8-9 (2008).

86. Briand, L. C., Morasca, S., and Basili, V. R., "Property- Based Software Engineering Measurement," **IEEE Transactions on Software Engineering**, vol. 22 (1), (1996).

87. Çatal, Ç. ve Diri, B., "Software fault prediction with object oriented metrics based artifical immune recognition system", **Lecture notes in computer science 4589**, Springer-Verlag, 300-314 (2007).

88. Çatal, Ç. ve Diri, B., "A fault prediction model with limited fault data to improve test process", **Lecture notes in computer science 5089**, Springer-Verlag, 244-257 (2008).

89. Chidamber, S.R. ve Kemerer, C.F., "A metrics suite for object-oriented design," **IEEE Transactions on Software Engineering**, Vol. 20, No. 6, 482-491 (1994).

90. Brito e Abreu, F., Pereira, G., and Soursa, P., "Coupling Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems," **Proc Euromicro Conference Software Maintenance and Reeng**., pp.13-22, (2000).

91. İnternet: McCabe Software "Using Code Quality Metrics in Management of Outsourced Development and Maintenance" http://www.mccabe.com/pdf/McCabeCodeQualityMetrics-OutsourcedDev.pdf (2010).

92. Chidamber, S.R, ve Kemerer, C.F., "Towards A Metric Suite For Object-Oriented Design", Proceedings : OOPSLA '91, Phoenix, AZ, pp.197-211, (1991).

93. Nabiyev, V.V., "Yapay Zeka", **Seçkin Yayıncılık**, Ankara, 445 (2005).

94. Allahverdi, N., "Uzman Sistemler: Bir Yapay Zeka Uygulaması", **Atlas Yayıncılık**, İstanbul, 16-20 (2002).

95. Muratore, J. F. and DeMasie, M. P. "Artificial intelligence and expert systems in-flight software testing", **Digital Avionics Systems Conference**, 1991. Proceedings., IEEE/AIAA 10th, Digital Object Identifier: 10.1109/DASC.1991.177202, Los Angeles, CA , USA, 416-419 (1991).

96. Xu, Z., Khoshgoftaar, T. M., Gao, K., "Application of fuzzy expert system ın test case selection for system regression test" **In Proceedings of IEEE International Conference on Information Reuse and Integration**, IEEE Computer Society, Las Vegas, NV, August (2005).

97. Xu, Z., Khoshgoftaar, T. M., Allen, E. B., "Application of fuzzy expert systems in assessing operational risk of software", **Information ve Software Technology,** 45(7): 373-388 (2003).

98. Ramsey, C. L. ve Basili, V. R., "An evaluation of expert systems for software engineering management," Departmant Computer Science University Maryland College Park. Tech. Rep. TR 1708, **IEEE Transactions** on **Software Engineering**, pp. 747–759, September (1986).

99. Tao, Y., " Using Expert Systems to Understand Object-Oriented Behaviour", **The 26th SISCSE Technical Symposium on Computer Science Education**, (1995).

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, adı : CALP, M.Hanefi

Uyruğu : T.C.

Doğum tarihi ve yeri : 20.04.1982 Erzurum

Telefon : 0 (506) 536 42 91

e-mail : mhcalp@gazi.edu.tr - hcalp25@hotmail.com

Eğitim

Derece	Eğitim Birimi	Mezuniyet Tarihi
Yüksek lisans	Gazi Üniversitesi /Bilişim Enstitüsü	2011
Lisans	Selçuk Üniversitesi / Elektronik ve Bilgisayar Eğitimi	2006
Ön Lisans	Atatürk Üniversitesi/ Bilgisayar Programcılığı	2002

İş Deneyimi

Yıl	Yer	Görev
2006-2007	Milli Eğitim Bakanlığı/Erzurum Tortum Çok Programlı Lisesi	Bilişim Teknolojileri Öğretmeni
2007-2009	Milli Eğitim Bakanlığı/Erzincan Refahiye Çok Programlı Lisesi	Bilişim Teknolojileri Öğretmeni/Bölüm Şefi
2009	Karadeniz Teknik Üniversitesi	Araştırma Görevlisi
2009- ..	Gazi Üniversitesi / Bilişim Enstitüsü	Araştırma Görevlisi
2011	Uluslararası Sertifikalı Yazılım Kalite Eğitimi (ISTQB)	Katılımcı

Yabancı Dil

İngilizce

Hobiler

Şiir Okuma, Bilgisayar Teknolojileri, Futbol, Yüzme, Araştırma Yapma, Seyahat