T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YAZILIM KALİTE METRİKLERİ İLE TEST EFORU ARASINDAKİ İLİŞKİNİN BELİRLENİP TARİHSEL VERİNİN OLUŞTURULMASI

YÜKSEK LİSANS TEZİ

Nurhan YAĞCI

Enstitü Anabilim Dalı	**:**	**BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ**
Tez Danışmanı	**:**	**Yrd. Doç. Dr. Kürşat AYAN**

Haziran 2013

T.C.
SAKARYA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YAZILIM KALİTE METRİKLERİ İLE TEST EFORU ARASINDAKİ İLİŞKİNİN BELİRLENİP TARİHSEL VERİNİN OLUŞTURULMASI

YÜKSEK LİSANS TEZİ

Nurhan YAĞCI

Enstitü Anabilim Dalı : **BİLGİSAYAR VE BİLİŞİM MÜHENDİSLİĞİ**

Bu tez 20 / 06 /2013 tarihinde aşağıdaki jüri tarafından Oybirliği ile kabul edilmiştir.

Yrd.Doç.Dr. Kerset Ayan	Yrd.Doç.Dr. Nilüfer Yurtay	Yrd.Doç.Dr. Mümtaz İpek
Jüri Başkanı	**Üye**	**Üye**

ÖNSÖZ

Geçmişte yazılım test süreci yazılım yaşam döngüsü içinde yer alan süreçlerden en çok ihmal edileni olsa da, günümüzde yaşanan tecrübeler sayesinde bu sürece verilen önem artmaktadır. Test sürecinin planlanması aşamasında ise en önemli konulardan biri kaynak planlanması yapılması için test efor tahminlemesinin yapılması aşamasıdır. Doğru yapılmış bir test efor tahminleme sürecinden sonra gerçekleştirilecek test süreci ortaya çıkacak yazılım ürünlerin daha az hata içermesini sağlayacaktır.

Tez çalışmam boyunca bana rehberlik eden, tavsiyeleri ve eleştirileriyle beni yönlendiren danışmanım Yrd. Doç Dr. Kürşat AYAN' a çok teşekkür ederim.

Savaş ÖZTÜRK' e tezimi hazırlarken bana verdiği yol gösterici fikirler için teşekkür ederim.

Tez çalışmamda bana gereken desteği ve zamanı sunan TÜBİTAK BİLGEM Yazılım Test ve Kalite Değerlendirme Birimine ve birim yöneticilerine teşekkür ederim.

Yüksek lisans eğitimim boyunca bana destek veren eşim Alper Yücel YAĞCI' ya ve tüm eğitim hayatım boyunca her zaman beni maddi ve manevi destekleyen aileme çok teşekkür ederim.

İÇİNDEKİLER

SİMGELER VE KISALTMALAR LİSTESİ

Sn	: Saniye
T.D.	: Test Durumu
IEEE	: The Institute of Electrical and Electronics Engineer
ILF	: İç mantıksal dosyalar
EIF	: Dış arayüz dosyaları
EI	: Dış girdiler
EO	: Dış çıktılar
EQ	: Dış sorgular
DIN	: Düzenlenmemiş işlev noktası
TED	: Toplam etki derecesi
DDE	: Değer düzenleme etkeni
TDIN	: Toplam düzenlenen işlev noktaları
TDI	: Toplam etki derecesi
De	: Fonksiyona bağlı sistemler için ağırlıklandırma etkeni
Kd	: Kullanıcı değeri
Ky	: Kullanım yoğunluğu
A	: Arabirim olma
K	: Karmaşıklık
T	: Tekdüzelik
De	: Fonksiyona bağlı sistemler için ağırlıklandırma etkeni
TNf	: Fonksiyona atanan test noktası sayısı
İNf	: Fonksiyona atanan işlev noktası sayısı
Kd	: Ağırlıklandırılmış dinamik kalite özelliklerinin değeri

TN	: Sisteme verilen toplam test noktası değeri
TTN	: Her bir fonksiyona verilen test noktalarının toplam değeri
İN	: Atanan işlev noktası sayılarının toplamı
OE	: Ortamsal etkenin hesaplanması
TTS	: Temel test saatlerinin hesaplanması
DKDA	: Düzeltilmemiş kullanım durumu ağırlıkları
DAA	: Düzeltilmemiş aktör ağırlıkları
DKDN	: Düzeltilmemiş kullanım durumu noktası
TÇEÇ	: Teknik ve Çevresel Etken Çarpanı
DKD	: Düzeltilmiş kullanıcı durumu
TDK	: Türk Dil Kurumu
Avg_v(G)	: Ortalama çevrimsel karmaşıklık
v(G)	: Çevrimsel karmaşıklık
ev(G)	: Esas karmaşıklık
iv(G)	: Modül tasarım karmaşıklığı
Branches	: Dallar
Call Pairs	: Çağrı çiftleri
ed(G)	: Esas sıkışıklık
Edge count	: Kenar sayısı
ev(G)>4	: Esas karmaşıklık kontrolü
id(G)	: Tasarım sıkışıklığı
Lines_with_Nodes	: Düğüm içeren satır sayısı
Norm_v(G)	: Normalize edilmiş çevrimsel karmaşıklık
MNT_SEV	: Bakım yapma zorluğu
Param_ Count	: Parametre sayısı
pv(G)	: Patolojik karmaşıklık
SLOC	: Sadece kod satır sayısı
vd(G)	: Çevrimsel sıkışıklık
vg>10||ev(G)>4	: Esas karmaşıklık ve çevrimsel karmaşıklık kontrolü

DIT	: Kalıtım ağacı derinliği
Lack_Cohesion	: Bütünlük kaybı
Max_ev(G)	: En büyük esas karmaşıklık
Max_v(G)	: En büyük çevrimsel karmaşıklık
Parent count	: Türetildiği sınıf sayısı
RFC	: Bir sınıf için çağrı sayısı
sum_v(G)	: Toplam çevrimsel karmaşıklık
Mv(g)	: Metodun v(g) metrik değeri
Km	: Metodun kapsam yüzdesi
Clv(g)	: Sınıfın v(G) metrik değerleri
Kc	: Sınıfın kapsam yüzdesi

ŞEKİLLER LİSTESİ

TABLOLAR LİSTESİ

ÖZET

Anahtar kelimeler: Yazılım Test Eforu Tahminleme, Yazılım Kalite Metrikleri

Günümüzde gelişen teknoloji ile beraber, yazılım ürünleri birçok alanda insan hayatının içine girmiş durumdadır. Dolayısıyla, bu ürünlerde meydana gelebilecek en küçük hata insan hayatını çok olumsuz etkilemektedir. Ortaya çıkabilecek hatanın maliyeti, kullanılan yazılımın önemine göre değişse; bu maliyet, de en iyi ihtimalle para, kötü ihtimaller söz konusu olduğunda ise insan hayatına zarar verecek boyutlara kadar ulaşabilir.

Yazılım testi; olası hataların erken evrelerde, ürün kullanıma girmeden bulunmasını sağlamak amacıyla yapılır. Yazılımların günümüzdeki etkilerinin büyüklüğü göz önüne alınırsa, yazılım testinin önemi net bir şekilde görülebilir. Daha iyi test süreçleri için, yazılım firmalarının kaynaklarını planlamaları ve test efor süreleri için tahminleme yapmaları gerekmektedir. Yapılacak bu tahminlemeler kullanılarak kaynak ve zaman ayarlamaları yapılacağı için tahminlemenin doğruluğu çok önemlidir. Ancak şimdiye kadar önerilen ve kullanılan yöntemler, ya çok kişisel kararlara bağlı ya da çok fazla efor gerektiren yöntemler olmaları sebebiyle kullanılamamaktadır. Bu çalışmada, yazılım kalite metrikleri ile test efor süresi arasında bir ilişkinin bulunması ve daha sonraki proje tahminlemelerinin bu ilişkiye göre yapılması yöntemi önerilmektedir. Böylelikle, nesnel test efor tahminleme yöntemi, kabul edilebilir sürelerde yapılabilinecektir.

FINDING THE RELATIONSHIP BETWEEN SOFTWARE TESTING EFFORT AND SOFTWARE QUALITY METRICS, GENERATING HISTORICAL DATA

SUMMARY

Key Words: Testing Effort Estimation, Software Quality Metric

Nowadays, by the development of software technology, software products are taken very big part in human life. So even a little failure in software products, can cause very bad consequences. For avoiding failures, software test process must be done properly.

The purpose of software test process is to find the possible failures before the software products are used. Considering the big role which software products have in human life, the important of the software test can understood clearly. For better software test process, software companies have to schedule and plan their sources and have to estimate the test effort accurate as possible. The accuracy of the estimation is very important for project success, because source and time planning is going to be actualizing according to this estimation. But so far the methods which have been used to estimate the test effort are too subjective or required too many efforts. We propose a new method for test effort estimation. Proposed method is about finding the relationship between software quality metrics and test effort execution then making the estimation according to this relationship. In this manner, objective test effort estimation is going to be possible in acceptable time.

BÖLÜM 1. GİRİŞ - YAZILIM TESTİ

Yazılım Testi; bir sistemin/yazılımın belirlenmiş isterleri yerine getirip getirmediğini belirlemek veya beklenen ile gerçek sonuçlar arasındaki farkları tespit etmek amacıyla gerçekleştirilen bir dizi faaliyettir. Yazılım testinin çeşitli tanımlamaları mevcuttur;

- Bir programın davranışını dinamik yöntemlerle, sonsuz bir küme içinden belirli sayıda seçilen test durumlarını kullanarak, beklenen davranışa uymadığı durumları bulma işlemidir [1].
- Yazılım testi; yazılımın yapması için tasarlandığı işlemleri yaptığından, yapması beklenmeyenleri yapmadığından emin olmak için yapılan bir süreç ya da süreçler serisidir [2].
- Test, test edilen yazılımın kalitesi artırmak ve ölçmek amacıyla test yazılımlarını kullanıp gerekli değişiklikleri yapan mühendislik eşzamanlı yaşam döngüsü sürecidir [3].

Yazılım testinin önemi yazılım projelerinde gün geçtikçe artmaktadır. Eskiden yazılım geliştiriciler kendi kodları test ediyorken günümüzde birçok yazılım geliştiren firma bağımsız test ekibi modelini benimsemektedir.

Bu modelde test ekibi oluşturulur. Yazılım testi ile ilgili raporları; test ekibi test ekibinden sorumlu olan test yöneticisine, test yöneticisi ile proje yöneticisine iletir. Bu modelde yazılım yöneticisi ile test yöneticisi aynı seviyededir [4].

1.1. Yazılım Yaşam Döngüsü

Yazılımın gerçeklenmeye başlamasına karar verilmesinden, kullanılmayacak duruma gelmesine kadar geçen süre yazılım yaşam döngüsü olarak isimlendirilmektedir.

Birçok farklı yazılım yaşam döngüsü geliştirme modelleri bulunmaktadır. Bu tez kapsamında şelale modelinden faydalanılacaktır.

1.1.1. Şelale modeli

Yazılım projelerinde en yaygın kullanılan modeldir. Gereksinimlerin belirlenmesi, tasarım, gerçekleme, test ve bakım döngüsünden oluşur.

Gereksinimlerin belirlenmesi; bu aşamada yazılımın gereksinimleri belirlenir ve dokümante edilir.

Bu gereksinimler aşağıdaki tiplerde hazırlanır.

Müşteri gereksinimleri; müşterinin geliştirecek olan yazılımdan beklentilerini ana hatlarıyla anlatıldığı dokümandır.

Sistem gereksinimleri; müşteri gereksinimlerin detaylandırılmasından oluşan yazılımın bir bütün halinde sistem olarak ele alındığı dokümandır.

Yazılım gereksinimleri; sistem gereksinimlerinin detaylandırılmasında oluşur. Detaylandırılma yapılırken yazılım alt modüllere bölünür ve gereksinimler modül bazında yazılır.

Tasarım; yazılım belirlenen isterlere göre tasarlanır.

Gerçekleme; yazılım kodlanır.

Test; gerçeklenen yazılım belirlenen gereklere göre test edilir.

Bakım; bu aşamada yazılım yaşam döngüsü sonunda gelen geri bildirimlere göre ya da değişen isteklere göre bu döngü tekrar işletilebilir.

1.2. Yazılım Testinin Önemi

Yazılım testinin öneminin anlaşılabilmesi için, yazılım testinin düzgün bir şekilde yapılmadığı projelerde ortaya çıkan sorunların incelenmesi faydalı olacaktır. Dünyadaki en önemli yazılım felaketlerinden bazıları aşağıda verilmiştir [5].

1.2.1. Sovyet uyarı sistemi hatası

Sovyet erken uyarı sisteminde bulunan bir yazılım hatası sonucunda 1983 yılında neredeyse 3. dünya savaşı meydana gelecekti. Rusların sistemi Amerika Birleşik Devletleri'nin beşli balistik füzelerini fırlattığını söyledi. Hatanın uydu tetikleyicilerinin bulutlardan yansıyan güneş ışınlarını toplayıp füze olarak algılanmasını önleyen yazılımda bulunan hata yüzünden ortaya çıktığı anlaşıldı.

1.2.2. AT&T ağ çökmesi

1990 yılında, bir anahtarda meydana gelen ufak bir mekanik arızası sonucu anahtar merkezi kapatıldı. Merkez tekrar açıldığında, diğer merkezleri etkileyen "Kapatın ve Tekrar Başlatın" mesajı gönderdi. Bu durum 75 milyon telefon aramasının cevapsız duruma düşmesine sebep oldu.

Sorunun çok karmaşık bir yazılım güncellenmesinde eklenen tek bir kod satırındaki hatadan dolayı oluştuğu anlaşıldı.

1.2.3. Ariane 5 patlaması

1996 yılında, Avrupa'nın en yeni ve insansız uydusu Ariane 5 kalkıştan sadece saniyeler sonra kendi kendini havaya uçurdu. Avrupa Uzay Ajansı, Ariane 5'in geliştirme maliyetinin 8 milyar dolardan daha fazla tuttuğunu belirtti. Kendi kendini

yok etme yazılımının 64 bitlik bir sayıyı 16 bitlik bir alana çevirmeye çalışırken tetiklendiği ortaya çıktı.

Belirtilen yazılım felaketleri ve daha birçoğu düzgün bir yazılım test sürecinden geçse idi önlenebilirdi. Yazılımlarda çıkan hatalar maddi zararlara yol açmasının yanı sıra, zaman zaman can kayıplarına mal olabilecek zararlara da yol açabilmektedirler. Günümüzde yazılımın yer almadığı alanlar yok denecek kadar azdır ve kullanılan yazılımların isterlerine uygun olup olmadığının kontrolü ise ancak test edilerek yapılabilir.

1.3. Yazılım Test Süreci

Yazılım test süreci yazılım yaşam döngüsü süreçlerinden biridir. Bu süreçte çeşitli tiplerde testler ile yazılım sınanır ve bu sınanmalardan geçen yazılımlar kullanıma alınır.

1.3.1. Yazılım test tipleri

Yazılım test tipi olarak; birim, yazılım, sistem ve kabul test tipleri sayılabilir.

Birim testleri**;** yazılım geliştiricileri tarafından yapılan beyaz kutu testleridir. Yazılım kodunda bulunan her bir metodu işlemini doğru yaptığından emin olunması için, her koda karşılık gelen test kodu yazılır ve çalıştırılır.

Yazılım testleri**;** yazılım gereksinimleri doğrulamak üzere test ekibi tarafından yapılan kara kutu testleridir. Bu yöntemde kodlar görüntülenmez, test ekibi kapalı bir yazılım sistemi üzerinde testlerini yapar.

Sistem testleri**;** sistem gereksinimleri doğrulamak üzere test ekibi tarafından yapılan kara kutu testleridir. Bu yöntemde, yazılım ve yazılımın kurulacağı donanım beraber test edilir.

Kabul testleri**;** müşteri gereksinimleri doğrulamak üzere müşteri ve test ekibi tarafından yapılan kara kutu testleridir. Kabul testleri doğrulandıktan sonra yazılım müşteri tarafından kabul edilir.

1.3.2. Yazılım test süreci adımları

Yazılım test sürecinde test planlama, tasarım, koşturma, hata yönetimi ve raporlama adımları mevcuttur. Testler bu süreçteki adımlara uygun olarak işletilir.

Test planlama**;** test süresince yapılacak işlemlerin planlandığı aşamadır. Test amaçları, stratejisi, ortamı ve takvimi bu aşamada belirlenir.

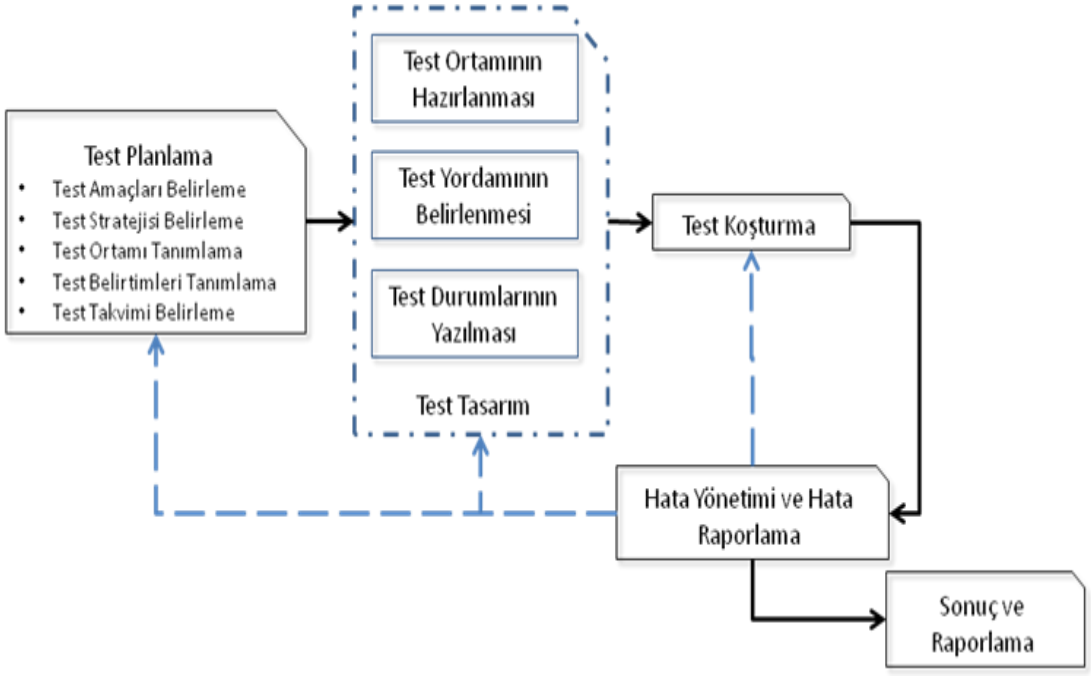
Test tasarım**;** test ortamının hazırlanması, test yordamların ve test durumların yazılması aşamasıdır.

Test koşturma**;** bu aşamada test tasarım aşamasında tasarlanan ortamda, test yordamları ve durumları koşturulur.

Hata yönetimi ve hata raporlama**;** test koşturması sonucunda bulunan hataların raporlanması işlemidir. Raporlanan hatalar yazılım ekibi ile paylaşılır. Hiçbir hata düzeltilmeden kapatılmaz.

Aşağıdaki standartlar baz alınarak oluşturulmuş olan yazılım test süreci adımları Şekil 1.1.' de gösterilmektedir.

- IEEE STD 1059-1993 Software Verification and Validation Plans [6]
- IEEE STD 1012-1998 Software Verification and Validation [7]
- IEEE STD 829-1998 Software Test Documentation [8]
- IEEE STD 1012a-1998 Supplement to IEEE Standard for Software Verification and Validation [9]
- IEEE STD 1028-1997 Software Reviews [10]



Şekil 1.1. Test süreci.

BÖLÜM 2. YAZILIM TEST EFOR TAHMİNLEME

Yazılım test efor tahminleme test için ayrılacak kaynak ve zamanın tahminlenmesidir. Bu güne kadar test efor tahminlemesi için birçok yöntem kullanılmıştır. Kullanılan yöntemler, artı ve eksi yönleriyle birlikte anlatılmaktadır.

2.1. Yazılım Test Efor Tahminlemenin Önemi

Yazılım testinin önemin gün geçtikçe kavranması ile birlikte yapılacak test çalışmalarının planlanması da önem kazanmıştır. Test edilecek çalışmada gerekecek kaynak ve zamanın belirlenebilmesi projenin başarısı için çok önemlidir. Çünkü yazılımın testi için yeterli kaynaklar ayarlanamazsa test süreci başarısız geçer ve bu durum yazılımın yeterli test edilememesine, hataların bulunamamasına ve dolayısıyla projenin başarısız olmasına sebep olabilir.

2.2. Kullanılan Yöntemler

Yazılım test efor tahmini yapılırken günümüze kadar çeşitli yöntemler kullanılmıştır. Bu yöntemlerden öne çıkanları aşağıda detaylı olarak açıklanmaktadır.

2.2.1. Yazılım geliştirme eforundan yararlanma

Bu yöntem kolay olmasından dolayı sıklıkla kullanılmaktadır. Bu yöntem ile test eforu hesaplaması yapılırken yazılımı geliştirme için harcanan eforun belirli bir yüzdesi alınır. Yüzdenin ne kadar olacağı proje yöneticisinin alacağı karara göre değişir.

Örnek bir hesaplama;

A yazılımı 6 kişi tarafından 5 ayda geliştirilmiş olsun.
Yazılım Geliştirme Eforu = Kişi Sayısı × Süre= 6 × 5 = 30 adam ay
Yazılım test eforu için yüzde %20 olarak alınırsa;
Yazılım Test Eforu = Yazılım Geliştirme Eforu × Belirlenen Yüzde = 30 × %20 = 6 adam ay olarak hesaplanır.

Bu durumda yazılım test eforu için 6 adam aylık bir tahminleme yapılabilir.

Yöntemin artıları;

- Hızlı, pratik, kolay hesaplanabilir bir yöntemdir.
- Ekstra bir efor harcanmasına gerek kalmadan kolaylıkla hesaplama yapılmasını sağlar.

Eksiklikler;

- Yazılım test efor tahminlemesini çok basite alan bir yöntemdir.
- Yazılım test etme ve yazılım geliştirme birbirlerinden ayrı uzmanlık alanlarına ihtiyaç duyan konulardır.
- Yazılım geliştiriciler ve yazılım testçilerinin yaptıkları işler birbirlerinden çok farklıdır.

2.2.2. İşlev noktaları analizi

Bu yöntem proje büyüklüğünü belirtmek amacıyla geliştirilmiştir. Bu yöntem geliştirilirken asıl amaç hem yazılım geliştiricilerin hem de kullanıcıların işlevsel isterlerini ortak bir dille anlatmasını sağlayabilmekti [11].

1979'da IBM'den Allan Albrecht tarafından Kaliforniya'da bir konferansta işlev noktaları fikri ortaya atıldı [12].

İşlev noktaları saat, kilo, ton, derece gibi günlük yaşamda kullanılan bir yapay ölçümdür. Bununla birlikte işlev noktaları bir uygulama sisteminin veya belirli bir

modülün işlevselliğine ve karmaşıklığına odaklanır. Değeri 1000 olan işlev nokta uygulaması, değeri 500 olan işlev nokta uygulamasına göre daha büyük ve daha karmaşıktır.

İşlev noktaları çözümlemesinin güzel yanı teknolojiden bağımsız olmasıdır. Daha özel olarak işlevsellik ve teknoloji birbirinden ayrı tutulur. Böylece farklı programlama dillerini veya teknoloji ortamlarını kullanan veya kullanmayan farklı uygulamaları karşılaştırabilme imkânı sunar. Örneğin COBOL'da yazılan bir uygulama ile Java'da geliştirilen bir diğer uygulamayı karşılaştırabilinir.

İşlev noktaları çözümlemesi güvenilirdir. İşlev noktaları çözümlemesinde deneyimli ve aynı nitelikleri olan iki kişi yapılan çözümleme sonucunda işlev nokta sayılarını aynı sayıda veya kabul edilebilir bir hata payı ile elde ederler. İşlev nokta çözümlemesini ciddi bir şekilde öğrenmek isteyenlerin sertifikalı olması önerilir.

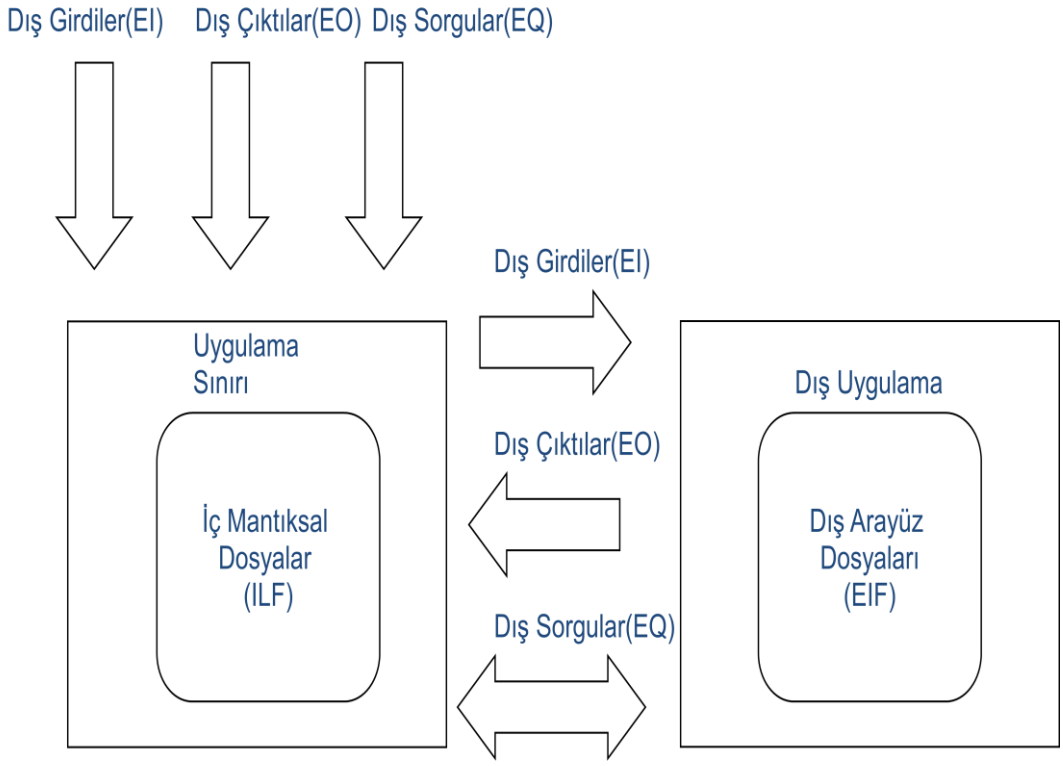
Birden fazla işlev nokta kurumu olmasına rağmen, IFPUG (Uluslararası İşlev Nokta Kullanıcıları Grubu) ve UFPUG (Birleşik Krallık İşlev Nokta Kullanıcıları Grubu) gibi kar amacı olmayan, kuralları, standartları ve yetkilendirmesi olan iki temel kuruluş vardır.

İşlev noktalarının sayımında anahtar nokta kullanıcının gereksinimlerini iyi anlamaktan geçer. Projenin başında, işlev nokta çözümlemesi projenin kapsamı çerçevesinde yürütülür. Kullanıcının gereksinimleri doğrultusunda daha ayrıntılı çözümleme, tasarım ve çözümleme aşamalarında yapılabilir.

İşlev nokta çözümlemesi proje yaşam döngüsünün çeşitli aşamalarında yürütülebilir. Örneğin; proje kapsamı tanımında, kestirim ve proje planının geliştirilmesi için kullanılabilir. Çözümleme ve tasarım aşamasında ise işlev noktaları ilerleyişin yönetilmesi ve raporlanması ayrıca kapsamın izlenmesinde kullanılabilir.

Ek olarak projenin yaşama geçirilmesinden sonra tüm işlevselliğin ulaştırıldığının saptanmasında da faydalı olabilir. Bu bilginin yakalanarak veritabanında tutulması ile ve diğer yazılım ölçümleri ile birleştirilerek gelecekteki projelerin kestirilmesi,

kalite sınaması, yeni yöntemlerin, araçların, teknolojilerin tanıtılması açısından faydalı olabilir. İşlev noktaları analizinin grafiksel gösterimi Şekil 2.1.' de verilmektedir.



Şekil 2.1. İşlev noktaları analizi.

İç mantıksal dosyalar (Internal logical file, ILF); uygulama sınırları ile birlikte verilerin saklandığı mantıksal bir dosyadır. ILF' nin karmaşıklığı, veri elemanlarının sayısına göre az, orta ve yüksek olarak sınıflandırılır. Veri elemanları müşterinin adı, numarası, adresi, telefon numarası vs. olabilir.

Dış arayüz dosyaları (External interface file, EIF); iç mantıksal dosyaya benzer. Bununla birlikte EIF başka bir uygulama sistemi tarafından kontrol edilen bir arayüz dosyasıdır. EIF' nin karmaşıklığı, ILF' de kullanılan aynı ölçüt ile saptanır.

Dış girdiler (External input, EI); uygulamanın dışından uygulamanın sınırları içine doğru olan süreçleri ve işlenebilir verileri gösterir. Veri genellikle uygulamaya içine eklenebilir, silinebilir veya güncellenebilir. En yaygın örnek kullanıcının bilgi girişi için kullandığı klavye ve faredir.

Dış çıktılar (External output, EO); verinin uygulama sınırları içinden dışarı çıkmasına izin veren süreç veya işlemlerdir.

Dış çıktıların örnekleri raporları, doğrulama mesajları, hesaplanan toplam değerler, çizgeler, çizelgeleri içerir. Veriler ekrana, yazıcıya veya diğer uygulamalara gidebilir. Dış çıktıların sayısı sayıldıktan sonra, karmaşıklıklarına göre derecelendirilirler.

Dış sorgular (External inquiry, EQ); elde edilen veri için iki yönlü iç dosyalardan dışarı veya dışarıdan uygulama içerisine girdilerin ve çıktıların birleşimini içeren bir süreç veya işlemdir. Dış sorgular dosyada saklanan veriyi değiştirmez veya güncellemez. Sadece bilgiyi okurlar.

Düzenlenmemiş işlev noktası (Unadjusted function point, DIN); tüm iç mantıksal dosyalar, dış arayüz dosyaları, dış girdiler, dış çıktılar ve dış sorgular sayıldıktan sonra ve karmaşıklıkları oranlandırıldıktan sonra düzeltilmemiş işlev noktası (Unadjusted function point, DIN) sayısı saptanır.

Örneğin; bir uygulama sisteminin gözden geçirildikten sonra aşağıdaki değerler saptanmıştır.

- ILF: 3 Basit, 2 Orta, 1 Karmaşık
- EIF: 2 Orta
- EI: 3 Basit, 5 Orta, 4 Karmaşık
- EO: 4 Basit, 2 Orta, 1 Karmaşık
- EQ: 2 Basit, 5 Orta, 3 Karmaşık

DIN' ın hesaplanması; bu aşamada kullanılan değerler Tablo 2.1.' de gösterilmektedir.

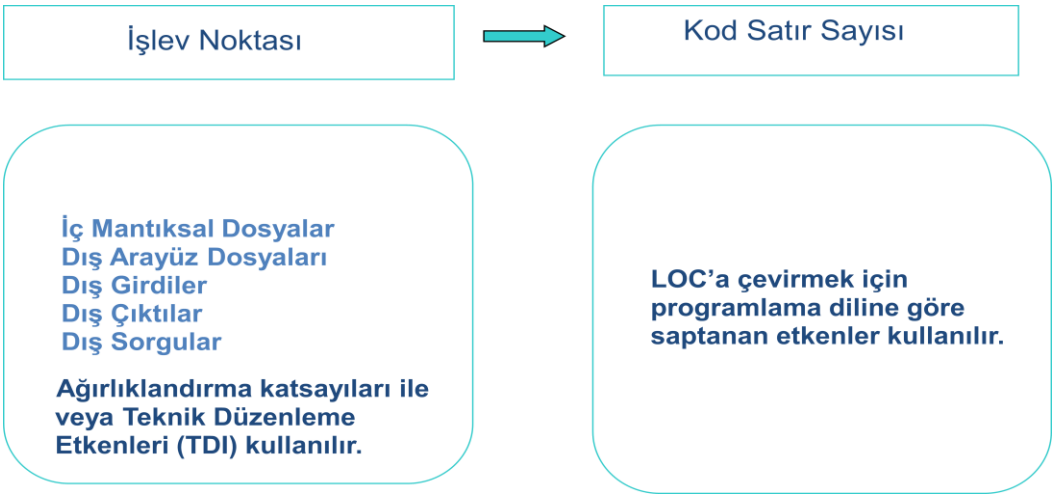
DIN değeri eşitlik (2.1) ile hesaplanır.

Tablo 2.1. DIN' ın hesaplanması

Karmaşıklık				
	Düşük	**Orta**	**Yüksek**	**Toplam**
İç mantıksal dosyalar(ILF)	3x7=21	2x10=20	1x15=15	56
Dış arayüz dosyaları(EIF)	_x5=_	2x7=14	_x10=_	14
Dış girdiler(EI)	3x3=9	5x4=20	4x6=24	53
Dış çıktılar(EO)	4x4=16	2x5=10	1x7=7	33
Dış sorgular(EQ)	2x3=6	5x4=20	3x6=18	44
Toplam düzeltilmemiş noktalar				200

$$DIN = İç\ Mantıksal\ Dosyalar x\ K(1)\ +\ Dış\ Arayüz\ Dosyaları\ x\ K(2)\ +\ Dış\ Girdilerx\ K(3)\ +\ Dış\ Çıktılarx\ K(4)\ +\ Dış\ Sorgularx\ K(5) \quad \textbf{(2.1)}$$

Yazılım büyüklük kestirimi; işlev noktası hesaplamasından kod satır sayısına çevrim Şekil 2.2.' de gösterilmektedir.



Şekil 2.2. İşlev noktasından kod satır sayısı dönüştürme.

Kullanıcının gereksinimleri doğrultusunda geliştirme sürecinin başlarında kolay bir şekilde hesaplanabilir. Geliştirme dilinden bağımsızdır. Genelde ilk aşamalarda işlev noktaları çözümlemesi kullanılıp daha sonra kod satır sayısına geçilir.

Teknik ayarlama etkenleri;

- Sistem güvenilir yedekleme ve kurtarma gerektiriyor mu?
- Veri iletişimi gerekiyor mu?
- Dağıtık fonksiyon var mı?
- Başarım kritik mi?
- Sistem çok kullanılan bir işletim ortamında mı çalışacak?
- Sistem çevrimiçi veri girişi gerektiriyor mu?
- Çevrimiçi veri girişi, giriş işlemlerinin birden fazla ekran ya da işlem üzerinden almasını mı gerektiriyor?
- Ana dosyalar çevrimiçi mi güncelleniyor?
- Girdiler, çıktılar, dosyalar ve sorgular karmaşık mı?
- Kod yeniden kullanılabilir olarak mı tasarlanmış?
- İç süreç karmaşık mı?
- Dönüşüm ve kurulum tasarım içerisinde mi?
- Uygulama değişik kuruluşlarda birden fazla kurulum gerektirecek şekilde mi tasarlanmış?
- Uygulama kullanıcı tarafından kolaylıkla kullanmayı ve değiştirmek üzere mi tasarlanmış?

Her bir genel sistem karakteristiği için 0’dan 5’ e kadar kullanılan ölçek Tablo 2.2.’ de gösterilmektedir.

Tablo 2.2.' de kullanılan değerlere göre alınan cevaplar ağırlıklandırılır ve toplam değer elde edilir.

Bu etkenler ağırlıklandırılırken uzman personel desteğine ihtiyaç vardır. Alınacak sonuçlar her bir yazılım projesinde farklı olabilir.

Tablo 2.2. Etki derecesi-Etki tanımı.

Etki derecesi	**Etkinin tanımı**
1	Önemsiz etki
2	Az etkili
3	Orta düzeyde etkili
4	Önemli düzeyde etkili
5	Güçlü etki

Toplam etki derecesi (TED) hesaplaması eşitlik (2.2)' de verilmiştir.

$$TED = \sum_{i=[1,14]} Cevap_i \qquad \textbf{(2.2)}$$

Burada Cevap; her bir teknik ayarlama etkeni için alınan değerleri göstermektedir.

Değer düzenleme etkeni (DDE) hesaplaması eşitlik (2.3)' de verilmiştir.

$$DDE = (TED \times 0.01) + 0.65 \qquad \textbf{(2.3)}$$

Toplam düzenlenen işlev noktaları (TDIN) hesaplaması eşitlik (2.4)' de verilmiştir.

$$TDIN = DDE \times DIN \qquad \textbf{(2.4)}$$

Genel sistem karakteristiklerinin etki derecesi değerleri Tablo 2.3.' de verilmektedir.

Hesaplamalar bu değerlere göre yapılmaktadır.

Tablo 2.3. Genel sistem karakteristiği

Genel sistem karakteristiği	**Etkinin derecesi**
Veri iletişimi	3
Dağıtık veri işleme	2
Başarım	4
Çok kullanılan bir işletim ortamı	3
İşlem oranı	3
Çevrimiçi veri girişi	4
Uç kullanıcı verimliliği	4
Çevrimiçi güncelleme	3
Karmaşık işlemler	3
Yeniden kullanılabilirlik	2
Kurulum kolaylığı	3
İşlemsel kolaylık	3
Birden fazla kurulum	1
Uygulamanın değiştirilebilirliği	2
Toplam etki derecesi (TDI)	40

Hesaplamalar;

Örnek DDE hesaplaması eşitlik (2.5)' de verilmiştir.

$$DDE = (40 \times 0,01) + 0,65 = 1,05 \quad \textbf{(2.5)}$$

Örnek TDIN hesaplaması eşitlik (2.6)' da verilmiştir.

$$TDIN = 200 \times 1,05 = 210 \quad \textbf{(2.6)}$$

İşlev noktasından kod satır sayısına çevrim; işlev noktasından kod satır sayısına çevrimin hesaplanma örneği Tablo 2.4.' de gösterilmektedir.

Tablo 2.4. İşlev noktasından kod satır sayısına çevrim

Dil	**Her bir işlev noktasına karşılık ortalama kaynak kod satır sayısı**	**210 işlev noktasına sahip bir uygulama için ortalama kaynak kod satır sayısı**
Access	38	$(210 \times 38)=7980$
Basics	107	22470
C	128	26880
C++	53	11130
Cobol	107	22470
Delphi	29	6090
Java	53	11130
Makine Dili	640	134440
Visual Basic 5	29	6090

Kod satır sayısı hesaplaması eşitlik (2.7)' de verilmiştir.

$$Kod\ Satır\ Sayısı = \left(İşlev\ Noktaları\ Sayısı\right) \times \left(Programlama\ Dili\ Kaynak\ Kod\ Satır\ Sayısı\right) \quad \textbf{(2.7)}$$

İşlev noktasından test efor tahminleme; İşlev noktası tahminleme asıl olarak proje büyüklük tahminlemede kullanır.

Test eforu tahminlemesi yaparken ise proje büyüklüğünün belirlenen bir yüzdesi alınır.

Eksiklikler;

- Bu yöntem ile düzgün bir tahminleme yapabilmek için geliştirilecek yazılım ile ilgili çok fazla bilgiye sahip olmak gerekmektedir.
- Sonuca ulaşabilmek için uzman bir ekip tarafından çok uzun süre çalışılması gerekmektedir.
- Günümüz rekabet koşullarında sadece tahminleme yapılması için bu kadar uzun süre ayırabilmek mümkün gözükmemektedir.

2.2.3. Test noktası analizi

Test noktası analizi yöntemi, sistem ve kabul testleri için gerekli test eforunu objektif olarak tahminlemede kullanılabilecek yöntemlerden biridir [13].

Test noktası analizi yapılırken üç nokta üzerinde durulur;

- Büyüklük: Büyüklük bilgisi işlev noktası analizinden alınır.
- Strateji: Test kalitesi için gerekli olan önem ve test stratejisine göre test edilecek sistemlerin önemine göre belirlenir.
- Verimlilik: İki önemli yönü vardır; çevre ve verimlilik rakamları. Çevresel etkenler, çevre etkenlerinin (kullanılan test araçları, test yazılımları vb.) proje büyüklüğünü ne derece etkilediğini tarif eder. Verimlik rakamları ise bilgiye(projede çalışan uzaman testçi sayısı vb.) dayalıdır.

Fonksiyona bağlı sistemler için ağırlıklandırma; fonksiyona bağlı sistemle içi ağırlıklandırma, aşağıda tanımlanan değerler baz alınarak yapılmaktadır.

Kullanıcı değeri; kullanıcının sistemdeki diğer metotlara kıyasla seçilen metoda verdiği değeri gösterir.

- Düşük(3)
- Orta(6)
- Yüksek(12) olarak sınıflandırılır.

Kullanım yoğunluğu; kullanıcının belirlenen metodu hangi sıklıkla kullanacağı ve kullanacak grubun büyüklüğü ile ilgilenir.

- Düşük(2)
- Orta(4)
- Yüksek(12) olarak sınıflandırılır.

Arabirim olma; belirlenen fonksiyonda yapılacak değişikliklerin sistemin diğer kısımlarını ne derecede etkilediğini göstermede kullanılır.

Karmaşıklık; fonksiyonun algoritmasına göre karmaşıklık hesaplanır.

- 5 koşuldan daha fazla koşul içermiyorsa (3)
- 6-11 koşul içeriyorsa (6)
- 11 koşuldan daha fazla koşul içeriyorsa (12) olarak bölümlendirilir.

Tekdüzelik; sistemin ne kadar tekrar kullanılabilir olduğu ile ilgilenir. Tekdüzelik değeri eğer kod tekrarları, kullanılmayan fonksiyonlar vb. durumlar var ise 0.6, diğer durumlarda 1 olarak belirlenir.

Fonksiyona bağlı sistemler için ağırlıklandırma etkeni (De) hesaplaması eşitlik (2.8)' de verilmiştir.

$$De = \left(\left(Kd + Ky + A + K\right)/16\right) \times T \tag{2.8}$$

Burada Kd; kullanıcı değerini, Ky; kullanım yoğunluğunu, A; arabirim olmayı, K; karmaşıklığı ve T; tekdüzeliği göstermektedir.

Dinamik kalite özellikleri; test noktası analizi yapılırken 4 ölçülebilir kalite kısıtı ele alınır;

- Uygunluk
- Güvenlik

- Kullanılabilirlik
- Etkinlik

Oylama yapılırken; 0, 3, 4, 5, 6 notlarından biri derecelendirmeye göre verilir.

Dinamik kalite özelliklerinin değeri hesaplanırken; her bir dinamik için değerler 4'e bölünür ve sonrasında ağırlıklandırma etkeni ile çarpılır. (Her bir karakteristik için 0.02 kabul edilir).

Fonksiyona atanan test noktası sayısı (TNf) hesaplaması eşitlik (2.9)' da verilmiştir.

$$TNf = \dot{I}Nf \times De \times Kd \qquad \textbf{(2.9)}$$

Burada İNf; fonksiyona atanan işlev noktası sayısını göstermektedir.

Statik test noktaları; ISO 9126'ya göre belirlenen kalite karakteristiklerini kullanarak bir kontrol listesi oluşturulur [14].

Bu kontrol listesine göre yapılacak testler yapılırsa, her bir kontrolün testi için 16 puan eklenir.

Toplam test noktası; test noktalarının toplamı (TN) hesaplaması eşitlik (2.10)' da verilmiştir.

$$TN = TTN + \left(\dot{I}N \times Kd\right) / 500 \qquad \textbf{(2.10)}$$

Burada TTN; her bir fonksiyona verilen test noktalarının toplam değerini, İN; atanan işlev noktası sayılarının toplamını (en düşük değeri 500) göstermektedir.

Verimlilik/Yetkinlik etkenlerinin hesaplanması; Verimlilik/Yetkinlik etkenleri, her bir test noktasının test edilmesi için gereken test saatlerinin sayısıdır. Bu değer testi

gerçekleştirecek takımın deneyimini, bilgisini, uzmanlığını ve gerçekleme yeteneğini ölçer.

Verimlilik etkeni projeden projeye ve kurumdan kuruma değişkenlik gösterir. Örneğin, eğer test takımında birçok uzman test mühendisi varsa verimlilik artar. Verimlilik etkeninin yüksek olması test süresinin uzayacağı anlamına gelir.

Ortamsal etkenin hesaplanması; test saatlerinin sayısı sadece test takımının yeteneği ile hesaplanması aynı zamanda çeşitli kaynakların kullanıldığı ortamdan da etkilenir.

Temel test saatlerinin hesaplanması; temel test saatleri (TTS) hesaplaması eşitlik (2.11)' de verilmiştir.

$$TTS = TN \times Verimlilik / Yetkinlik\ Etkeni \times OE \quad \textbf{(2.11)}$$

Burada OE; ortamsal etkeni göstermektedir

Eksiklikler;

- Fonksiyona bağlı sistemler için ağırlıklandırma, dinamik kalite özellikleri gibi hesaplamalarda kullanılan etkenlerinin hesaplanması nesnel olmayan yöntemlerle yapılmaktadır.
- Başta işlev noktası analizine ihtiyaç duymaktadır.
- Çok fazla analiz edilecek etkene ihtiyaç duymaktadır.
- Çok fazla zaman gerektirmektedir.

2.2.4. Kullanım durumu noktası analizi

Kullanım durumları, hedeflenen bir amaç ile ilgili sistemin dış aktörleri ve sistem arasındaki olası etkileşim sıralamalarını tanımlar [15]. Kullanım durumları en temel anlamı ile kullanıcının sistemden ne istediğinin temel bir şekilde gösterimidir. Kullanım durumu noktası analizi yaklaşımında her bir kullanım durumu için her bir

senaryo ve aykırı durumu, test durumuna girdi oluşturur. İsterler zamanla netleştikçe tahminleme de revize edilerek netleştirilir [16].

Bu yaklaşımının kullanılabilmesi için kullanım durumu analizi çalışmalarının yapılması gerekmektedir.

Aktör belirleme ve ağırlıklandırma; Sistemde bulunacak aktörler belirlenmelidir. Bu bize düzeltilmemiş aktör ağırlıklarını verecektir. Aktörler sistemle etkileşimi olan sistem dışındakilerdir. (Örneğin; son kullanıcılar, diğer programlar vb.) Aktörler 3 gruba ayrılır; basit, ortalama ve karışık.

Aktör sınıflandırması geliştirme eforu ve test eforu hesaplanırken farklılık gösterir. Son kullanıcılar basit aktörlerdir, yaptıkları hareketler çeşitli programlar yardımıyla kolaylıkla otomatize edilebilir. Ortalama aktörler ise sistem bazı protokoller aracılığıyla etkileşime girerler. Karışık aktör ise sistem ile bir alt seviye yazılım aracılığıyla haberleşen ayrı sistemlerdir.

Aktör derecelendirmesinde kullanılan hesaplama değerleri Tablo 2.5.' de verilmektedir. Derecelendirme hesaplaması bu değerler kullanılarak yapılacaktır.

Tablo 2.5. Aktör derecelendirme

Tip	**Tanım**	**Etki derecesi**
Basit	Kullanıcı arayüzü	1
Ortalama	Etkileşimli ya da protokol üzerinden arayüzlü	2
Karışık	API/Düşük seviye etkileşim	3

Bu değerlerin toplamı düzeltilmemiş aktör ağırlıklarını (DAA) vermektedir.

Kullanım durumu belirleme ve ağırlıklandırma; Sistemdeki kullanım durumlarının sayısı belirlenir. Kullanım durumları içerdikleri işlemlerin ve senaryoların sayısına göre ağırlıklandırılır. Bu ağırlıklandırmada kullanılacak değerler Tablo 2.6.' da gösterilmektedir.

Tablo 2.6. Kullanım durumu derecelendirme

Tip	**Tanım**	**Etki derecesi**
Basit	<=3	1
Ortalama	4-7	2
Karışık	>7	3

Bu değerlerin toplamı düzeltilmemiş kullanım durumu ağırlıklarını (DKDA) vermektedir. Düzeltilmemiş kullanım durumu noktası hesaplama; Düzeltilmemiş kullanım durumu ağırlıkları (DKDN) hesaplaması eşitlik (2.12)' de verilmiştir.

$$DKDN = DAA + DKDA \quad \textbf{(2.12)}$$

Teknik ve çevresel etkenlerin hesaplanması; bir test projesinde görülebilecek teknik ve çevresel etkenler aşağıdaki tabloda verilmektedir. Hesaplama yapılırken; her bir etken için değerler ağırlıklandırılır ve ağırlıklandırılan değerler belirlenen değer ile çarpılarak sonuç bulunur. Tüm etkenlerin sonuçları toplanarak, teknik ve çevresel etken çarpanı (TÇEÇ) hesaplanır. Teknik ve çevresel etkenlerin hesaplanmasında kullanılacak değerler Tablo 2.7.' de gösterilmektedir.

Tablo 2.7. Teknik ve çevresel etkenlerin hesaplanması

Etken	**Tanım**	**Belirlenen değer**
T1	Test araçları	5
T2	Belgelenmiş girdiler	5
T3	Geliştirme ortamı	2
T4	Test ortamı	3
T5	Tekrar kullanılabilecek test yazılımları	3
T6	Dağıtık sistemler	4
T7	Performans hedefleri	2
T8	Güvenlik özellikleri	4
T9	Karışık arayüz yapısı	5

Düzeltilmiş kullanım durumu noktası hesaplama; düzeltilmiş kullanım durumu ağırlıkları (DKD) hesaplaması eşitlik (2.13)' de verilmiştir.

$$DKD = DKDN \times [0.65 + (0.01 \times TÇEÇ)] \quad \textbf{(2.13)}$$

Sonuç; son nokta olarak elde edilen düzeltilmiş kullanıcı durumu değerini, bir çevirme etkeni ile çarparak kullanım durumu noktası değeri elde edilmektedir. Bu çevirme etkeni dil ve teknoloji kombinasyonuna göre gerekli olan test eforu adam/saat bazında göstermektedir. Kurumların bu kombinasyona uygun çevirme etkenlerini belirlemeleri gerekir.

Efor hesaplaması eşitlik (2.14)' de verilmiştir.

$$Efor = Düzeltilmiş\ kullanıcı\ durumu \times Çevirme\ etkeni \quad \textbf{(2.14)}$$

Eksiklikler;

- Her sistem için kullanım durumlarının tasarımı yapılmamaktadır.
- Kullanım durumlarının tasarımının tüm senaryoları kapsayacak şekilde detaylı yapılamamaktadır.
- Kullanım durumu, aktörler, teknik ve çevresel etkenlerinin hesaplanması nesnel olmayan yöntemlerle yapılmaktadır.
- Son adımda belirlenen çevirme etkeninin nasıl belirleneceği belirtilmemektedir.

BÖLÜM 3. YAZILIM KALİTESİ

Yazılım tasarımı kalitesi yazılımın tasarıma uygunluğunun kalitesini ölçer. "Amaca Uygunluk" ifadesi ile tariflenebilir.

- "Belirtilen ya da kastedilen ihtiyaçların karşılanması yeteneğini taşıyan bir ürünün ya da hizmetin karakteristiklerinin ve özelliklerinin toplamıdır [17]."
- "Yazılım ürünün belirtilen isterleri karşılama yeteneğidir [18]."
- "Sistem, bileşen ya da sürecin belirtilen isterleri karşılama derecesidir [19]."
- "Sistem, bileşen ya da sürecin müşteri ya da kullanıcı ihtiyaçlarını ya da beklentilerini karşılama derecesidir [19]."

3.1. Yazılım Kalite Metrikleri

"Metrik" Türk Dil Kurumuna (TDK) göre "matematik ölçümlü" anlamına gelmektedir [20]. Çalışmada kullanılan anlamı ile metrik yapılan ölçümlerde kullanılan ölçütlerdir. Bu ölçütlerin nasıl alınacağı ile ilgili kıstaslar belirli ve tanımlı olmalıdır. Örneğin, bir yazılımın kod satır sayısı ölçümü alındığında; "kod satır sayısı" metriğinin değeri alınmış olur.

Niçin ölçmeye ihtiyaç duyarız:

Ölçüm alma işleminin gerekliliği ile ilgili çeşitli araştırmalar yapılmıştır.

"Konuştuğun şeyleri ölçebilir ve sayılarla açıklayabilirsen o konu hakkında bir şeyler biliyorsun demektir ancak eğer bunu yapamıyorsan bilgin yetersizdir ve tatmin edici değildir. " Lord Kelvin, 1889

“Ölçemediğinizi yönetemezsiniz” W. Edwars Deming
Ölçüm almak bize öncelikle üç konuda fayda sağlar;

- Anlama; Ölçümler alarak projenin durumunu anlayabiliriz.
- Kontrol etme; Belirlediğimiz metrik sınırları koyarak kontrol edebiliriz.
- Geliştirme; Belirlediğimiz sınırlara uyarak geliştirilen projenin kalitesini artırabiliriz.

3.2. Yazılım Kalite Metrik Grupları

Yazılım metrikleri aşağıda belirtilen üç kategoride değerlendirilmektedir [21].

- Ürün metrikleri: ürünün büyüklük, karmaşıklık karakteristiklerini ortaya koyan metriklerdir.
- Süreç metrikleri: yazılım geliştirme ve bakım süreçlerini iyileştirmek amacıyla kullanılan metriklerdir. Bulunan hata sayıları, her işlem için cevap süreleri bu metriklere örnek olarak verilebilir.
- Proje metrikleri: proje karakteristiklerini ve uygulamasını tarif eden metriklerdir. Yazılım mühendisi sayıları, proje bütçesi bu metriklere örnek verilebilir.

Yazılım kalite metrikleri yazılım metriklerinin alt kümesidir, yazılım metriklerinin kaliteye bakan kısmı ile ilgilenir. İki tiptir;

- Yazılım kalite süreç metrikleri
- Yazılım kalite ürün metrikleri

Çalışmamızda yazılım kalite ürün metriklerinden faydalanacağız.

3.3. Yazılım Kalite Ürün Metriklerinin Tarihçesi

Yazılım metrikleriyle ilgili ilk kitap [22] 1976 yılına kadar basılmamış olsa da aktif yazılım kalite metriklerinin tarihi 1960’ lara dayanır. Daha sonraları kod satır sayısı

metriği hem programcının üretkenliği (aylık yazdığı kod satır sayısı vb.) hem de yazılımın kalitesini (kod satır sayısı başına düşen hata vb.) ölçmek amacıyla kullanılmaya başlandı. Diğer bir deyişle kod satır sayısı metriği programın büyüklüğünü göstermede bir ölçüt olarak kullanıldı. İlk kaynak tahminleme modelleri de [23-24] satır sayısı veya ilişkili diğer metrikleri tahminleme girdileri olarak kullandı.

Kod satır sayısı metriğinin programın karmaşıklığını göstermede yetersiz kaldığının anlaşılması ile beraber farklı metrik oluşturma çalışmaları hızlandı. Özellikler farklı programlama dillerinin ortaya çıkması ile birlikte farklı metriklere olan ihtiyaç daha da arttı. 1970'lerde birçok yeni metrik ortaya atıldı. Bu çalışmalardan bir kısmı yazılımın karmaşıklığı ile [25-26] bir kısmı ise yazılımın büyüklüğü ile ilgili metrikleri [27-28] ortaya attı. Bu metriklerin programlama dili bağımsız olarak ölçülebilinmesi amaçlanmaktaydı.

Nesne yönelimli dillerin kullanılmaya başlanması ile birlikte sınıf bazlı metrikler ön plana çıkmaya başladı. Bu konuda en yaygın kullanılan metrikler Chidamber ve Kemerer [29] tarafından geliştirilmiş sınıf bazlı metriklerdir.

Daha sonraları bu metrikler kullanılarak çeşitli araştırmalar yapılmıştır. Çalışmamızda satır sayısı metrikleri, McCabe çevrimsel karmaşıklık metrikleri ve türevleri ayrıca Chidamber ve Kemerer metriklerinden seçilenler kullanılmıştır.

3.4. Kullanılan Metrikler

Yazılım kalite ölçümlerinde çeşitli metrikler kullanılmaktadır. Bu çalışmada kullanılacak olan metrikler metot ve sınıf bazlı olmak üzere iki ayrı bazda ele alınacaktır.

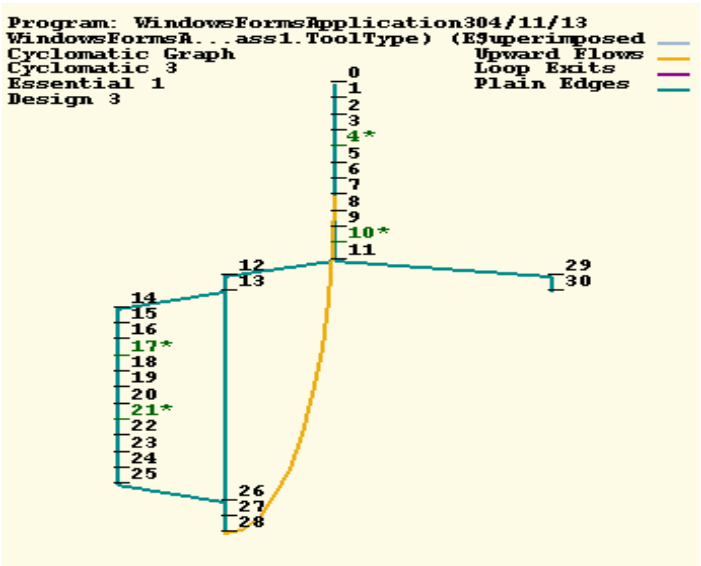
3.4.1. Metot bazlı metrikler

Bu metriklerle alınan ölçümler metot bazlı olarak alınmaktadır. Akış diyagramı gösterilecek olan metodun kod gösterimi Şekil 3.1.' de verilmektedir.

```
internal void createCheckBox(CheckedListBox cbMTypes, ToolType tType)
{
    cbMTypes.Items.Clear();
    int i = 0;

    //Create Checkbox
    foreach (var enums in Enum.GetNames(typeof (MeasurementType)))
        //if tool type is McCabe then add checkbox
    {
        if (tType == ToolType.McCabe)
        {

            cbMTypes.Items.Add(enums);

            cbMTypes.SetItemChecked(i, true);

            i++;
        }
    }
}
```

Şekil 3.1. Örnek metot kodu

Metod akış diyagramının grafiksel gösterimi Şekil 3.2.' de sunulmaktadır.



Şekil 3.2. Metot akış diyagramı

Çevrimsel karmaşıklık - (v(G)); Thomas J. McCabe tarafından 1976 yılında ortaya atılmış bir metriktir [25]. Çevrimsel karmaşıklık bir kod parçası içerisinde yer alan

“if” ve “for” gibi yapıların çokluğu ile doğrudan ilişkilidir. Bir yazılım ne kadar karmaşıksa o kadar hataya açıktır ve test edilmesi de zordur.

Çevrimsel karmaşıklık için McCabe eşik değerini 10 olarak belirlemiş ve bu değerin üzerinde karmaşıklığa sahip modülleri test edilebilirlik açısından riskli bulmuştur [30].

Aşağıdakilerden her biri karmaşıklığı bir artırır:

- “if” ifadesi (programa yeni bir dal ekler)
- “for” ve “while” döngüleri gibi çevrim yapıları
- Bir “switch” bloğundaki her “case” parçası
- Bir “try” bloğundaki her “catch”(...) parçası
- Önişlemciler (#if, #ifdef, #ifndef, #elif)

Örnekte verilen metot için çevrimsel karmaşıklık 3 olarak hesaplanır.

Esas karmaşıklık - (ev(G)); bir kod parçası içerisinde yapısal olmayan kod miktarını saptayabilmek için kullanılır [25]. Esas karmaşıklık, tamamen yapısal olmayan bir program için çevrimsel karmaşıklığa eşittir. McCabe esas karmaşıklık için eşik değerini 4 olarak verse de ideali 1'e indirgemektir [30].

Yapısal olan bir kod, küçük bir değişiklik ile tamamen yapısallığını kaybedebilir.

Esas karmaşıklığı genellikle aşağıdaki durumlar artırır:

- Bir fonksiyonda birden fazla yerde ya da arada bir yerde “return” kullanmak
- “break” ve “continue” kullanmak
- Bir koşul bloğu içerisinde “try/catch” kullanmak
- “and” (&) ve “or” (||) operatörleri

Örnekteki metot için esas karmaşıklık değeri 1’dir.

Modül tasarım karmaşıklığı - (iv(G)); bir metodun başka metotlara yaptığı çağrı sayısı ile orantılıdır [31]. Çağrı yapılmayan kod elimine edildiği ortaya çıkan kodun karmaşıklığı modül tasarım karmaşıklığıdır. En fazla çevrimsel karmaşıklık kadar olabilir. McCabe eşik değeri 7 olarak belirlenmiştir [30].

Örnekteki metot için esas karmaşıklık değeri 3'tür.

Dallar – (Branches); akış diyagramındaki başlangıç çizgisi ve herhangi bir karar da çıkan çizgi sayılarının toplamıdır.

Örnekteki metot için dallar değeri 5'tir.

Çağrı çiftleri – (Call Pairs); metot içinde diğer metotlara yapılan çağrı sayısına eşittir.

Örnekteki metot için çağrı çiftleri değeri 4'tür.

Esas sıkışıklık – (ed(G)); esas karmaşıklığın 1 eksiğinin çevrimsel karmaşıklığının 1 eksiğine bölünmesi ile bulunur. McCabe eşik değeri 0.40 olarak belirlenmiştir [30].

Esas sıkışıklık değeri (ed(G)) hesaplaması eşitlik (3.1)' de verilmiştir.

$$ed(G) = (ev(G) - 1) / (v(G) - 1) \quad \textbf{(3.1)}$$

Örnekteki metot için esas sıkışıklık değeri 0'dır.

Kenar sayısı – (Edge count); akış diyagramındaki çizgi aralık sayılarının toplamıdır.

Örnekteki metot için dallar değeri 32' dir.

Esas karmaşıklık kontrolü – (ev(G)>4); esas karmaşıklık değeri eşik değer olan 4 değerinden büyük ise "Pozitif" değil ise "Negatif" kabul edilir.

Örnekteki metot için değeri "Negatif"'' tir.

Tasarım sıkışıklığı – (id(G)); modül tasarım karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur. McCabe eşik değeri 0.70 olarak belirlenmiştir [30].

Tasarım sıkışıklığı değeri (id(G)) hesaplaması eşitlik (3.2)' de verilmiştir.

$$id(G) = iv(G) \ / \ v(G) \quad \textbf{(3.2)}$$

Örnekteki metot için esas sıkışıklık değeri 1'dir.

Düğüm içeren satır sayısı – (Lines_with_Nodes); metot içersindeki düğüm içeren satır sayısı değeridir. McCabe eşik değeri 30 olarak belirlenmiştir [30].

Örnekteki metot için değeri 13' tür.

Bakım yapma zorluğu – (MNT_SEV); esas karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur. McCabe eşik değeri 1 olarak belirlenmiştir [30].

Bakım yapma zorluğu değeri (MNT_SEV) hesaplaması eşitlik (3.3)' de verilmiştir.

$$MNT\_SEV = ev(G) \ / \ v(G) \quad \textbf{(3.3)}$$

Örnekteki metot için esas sıkışıklık değeri 0.33' tür.

Normalize edilmiş çevrimsel karmaşıklık – (Norm_v(G)); çevrimsel karmaşıklığı metot satır sayısına bölünmesi ile bulunur. McCabe eşik değeri 0.28 olarak belirlenmiştir [30].

Normalize edilmiş çevrimsel karmaşıklık (Norm_v(G)) hesaplaması eşitlik (3.4)' de verilmiştir.

$$Norm\_v(G) = v(G) \ / \ Satir\ Sayisi \quad \textbf{(3.4)}$$

Örnekteki metot için değeri 0.19’ dur.
Parametre sayısı – (Param_Count); metodun aldığı parametre sayısıdır. McCabe eşik değeri 5 olarak belirlenmiştir [30].

Örnekteki metot için değeri 2’ dir.

Patolojik karmaşıklık – (pv(G)); metodun azaltılmış diyagramının çevrimsel karmaşıklığıdır [25]. McCabe eşik değeri 2 olarak belirlenmiştir [30].

Örnekteki metot için değeri 1’ dir.

Sadece kod satır sayısı – (SLOC); metodun sadece koddan oluşan satırlarının sayısıdır. McCabe eşik değeri 30 olarak belirlenmiştir [30].

Örnekteki metot için değeri 13’ tür.

Çevrimsel sıkışıklık – (vd(G)); çevrimsel karmaşıklığın metot kod satır sayısı ve metot karışık kod satır sayısının toplamına bölünmesi ile bulunur. McCabe eşik değeri 0.14 olarak belirlenmiştir [30].

Çevrimsel sıkışıklık (vd(G)) hesaplaması eşitlik (3.5)’ de verilmiştir.

$$vd(G) = v(G) \ / \ (Kod\ Satir\ Sayisi + Karisik\ Kod\ Satir\ Sayisi) \quad \textbf{(3.5)}$$

Örnekteki metot için değeri 0.23’ dür.

Esas karmaşıklık ve çevrimsel karmaşıklık kontrolü – (vg>10||ev(G)>4); esas karmaşıklık değeri eşik değer olan 4 değerinden büyük ise ya da çevrimsel karmaşıklık değeri eşik değer olan 10 değerinden büyük ise “Pozitif” diğer durumlarda “Negatif” kabul edilir.

Örnekteki metot için değeri “Negatif”’ tir.

3.4.2. Sınıf Bazlı Metrikler

Bu metriklerle alınan ölçümler sınıf bazlı olarak alınmaktadır. Bu metrikler nesneye yönelimliğinin gösteriminde kullanılır. Nesneye yönelimli metriklerde belirlenen sınır değerleri aşmanın projedeki hata sayısını artırdığını gösteren birçok çalışma yapılmıştır [32-34].

Ortalama çevrimsel karmaşıklık - (Avg_v(G)); çevrimsel karmaşıklık metriği Thomas J. McCabe tarafından 1976 yılında ortaya atılmış bir metriktir [25]. Çevrimsel karmaşıklık bir kod parçası içerisinde yer alan "if" ve "for" gibi yapıların çokluğu ile doğrudan ilişkilidir. Bir yazılım ne kadar karmaşıksa o kadar hataya açıktır ve test edilmesi de zordur. Nesneye yönelimli programlamada ortalama çevrimsel karmaşıklık hesaplanırken sınıf içinde bulunan metotların çevrimsel karmaşıklığı toplanır ve metot sayısına bölünür.

Dallar – (Branches); akış diyagramındaki başlangıç çizgisi ve herhangi bir karar da çıkan çizgi sayılarının sınıf bazında toplamıdır.

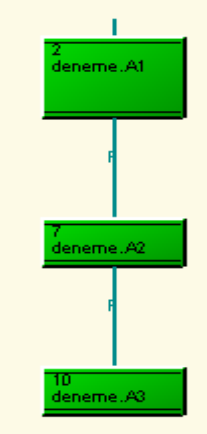
Kenar sayısı – (Edge count); akış diyagramındaki çizgi aralık sayılarının sınıf bazında toplamıdır.

Düğüm sayısı – (Node count); akış diyagramındaki düğüm sayılarının sınıf bazında toplamıdır.

Düğüm içeren satır sayısı – (Lines_with_Nodes); metot içersinde yer alan düğüm içeren satır sayılarının sınıf bazında toplamıdır.
Parametre sayısı – (Param_Count); bu değer; metodun aldığı parametre sayılarının sınıf bazında toplamıdır.

Kalıtım ağacı derinliği – (DIT); kalıtım ağacı derinliği sınıfın kalıtım ağacı köküne olan uzaklığıdır. Kalıtım ağacındaki derinlik arttıkça sınıfın davranışını tahmin

etmek ve sınıfı yönetmek zorlaşır. Derin kalıtım ağaçları tasarım karmaşıklığına sebep olur [29].
Örnek olarak, A3 sınıfı A2 sınıfından, A2 sınıfı da A1 sınıfından türüyorsa; oluşacak olan kalıtım ağacı Şekil 3.3.' de gösterilmektedir.



Şekil 3.3. Kalıtım ağacı örneği

- A1 sınıfı derinlik derecesi; "1",
- A2 sınıfı derinlik derecesi; "2",
- A3 sınıfı derinlik derecesi; "3" olarak kabul edilir.

Esas karmaşıklık - (ev(G)); bir kod parçası içerisinde yapısal olmayan kod miktarını saptayabilmek için kullanılır [25]. Esas karmaşıklık, tamamen yapısal olmayan bir program için çevrimsel karmaşıklığa eşittir. Nesneye yönelimli programlamada esas karmaşıklık hesaplanırken sınıf içinde bulunan metotların esas karmaşıklığı toplanır ve metot sayısına bölünür.

Modül tasarım karmaşıklığı - (iv(G)); modül tasarım karmaşıklığı, bir metodun başka metotlara yaptığı çağrı sayısı ile orantılıdır [31]. Çağrı yapılmayan kod elimine edildiği ortaya çıkan kodun karmaşıklığı modül tasarım karmaşıklığıdır. En fazla çevrimsel karmaşıklık kadar olabilir.

Nesneye yönelimli programlamada esas karmaşıklık hesaplanırken sınıf içinde bulunan metotların modül tasarım karmaşıklığı toplanır ve metot sayısına bölünür.

Tasarım sıkışıklığı – (id(G)); modül tasarım karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur. Nesneye yönelimli programlamada tasarım sıkışıklığı ortalama modül tasarım karmaşıklığı, ortalama çevrimsel karmaşıklık değerine bölünür.

Bütünlük kaybı - (Lack_Cohesion); metotların birbiri ile benzerliğini ölçer. Sınıfın uyumluluğunun düşük olması, sınıfın 2 veya daha fazla alt parçaya bölünmesi gerektiğini gösterir.

Düşük uyumluluk karmaşıklığı artırır, bu nedenle geliştirme aşamasında hata yapılma ihtimali yükselir. Ayrıca metotlar arasındaki ilişkisizliklerin ölçüsü ve sınıfların tasarımındaki kusurların belirlenmesinde de yardımcı olabilir [29].

En büyük esas karmaşıklık - (Max_ev(G)); içerisinde yer alan metotlar içerisinde, esas karmaşıklığı en yüksek olan metodun esas karmaşıklık değeridir.
En büyük çevrimsel karmaşıklık - (Max_v(G))

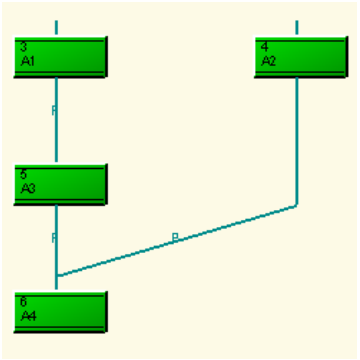
Max_v(G) metriği sınıf içerisinde yer alan metotlar içerisinde, çevrimsel karmaşıklığı en yüksek olan metodun çevrimsel karmaşıklık değeridir.

Bakım yapma zorluğu – (MNT_SEV); ortalama esas karmaşıklığın ortalama çevrimsel karmaşıklığa bölünmesi ile bulunur.

Normalize edilmiş çevrimsel karmaşıklık – (Norm_v(G)); ortalama çevrimsel karmaşıklığın toplam satır sayısına bölünmesi ile bulunur.

Türetildiği sınıf sayısı – (Parent count); bir sınıf için türetildiği sınıf sayısıdır. Sınıf hiyerarşi yapısının karmaşıklığını göstermekte kullanılır.

A3 sınıfı A1 sınıfından ve A4 sınıfı hem A3 hem A2 sınıfından türüyorsa oluşacak olan hiyerarşi gösterimi Şekil 3.4.’ de yer almaktadır.



Şekil 3.4. Türetildiği sınıf sayısı gösterimi

Bu durumda;

- A1 sınıfı türetildiği sınıf sayısı; “0”,
- A2 sınıfı türetildiği sınıf sayısı; “0”,
- A3 sınıfı türetildiği sınıf sayısı; “1”
- A4 sınıfı türetildiği sınıf sayısı; “2” olarak kabul edilir.

Patolojik karmaşıklık – (pv(G)); metodun azaltılmış diyagramının çevrimsel karmaşıklığıdır [25].

Nesneye yönelimli programlamada patolojik karmaşıklık hesaplanırken sınıf içinde bulunan metotların patolojik karmaşıklığı toplanır ve metot sayısına bölünür.

Bir sınıf için çağrı sayısı – (RFC); verilen sınıftan bir sınıfın metotları çağrıldığında, bu sınıfın tetikleyebileceği tüm metotların sayısıdır. Bu metrik sınıfın test maliyeti hakkında da fikir verir. Bir mesajın çok sayıda metodun çağrılmasını tetiklemesi, sınıfın testinin ve hata ayıklamasının zorlaşması demektir. Bir sınıftan fazla sayıda metodun çağrılması, sınıfın karmaşıklığının yüksek olduğunun işaretçisidir [29].

Toplam çevrimsel karmaşıklık - (sum_v(G)); sınıf içerisinde yer alan metotların çevrimsel karmaşıklığının toplamına eşittir.
Çevrimsel sıkışıklık – (vd(G)); ortalama çevrimsel karmaşıklığın, ortalama metot kod satır sayısı ve metot karışık kod satır sayısının toplamına bölünmesi ile bulunur.

BÖLÜM 4. YAZILIM KALİTE METRİKLERİ ve YAZILIM TEST EFORU ARASINDAKİ İLİŞKİ

Yazılım kalite metrikleri yazılan kodun büyüklüğü (satır sayısı metrikleri), karmaşıklığı (karmaşıklık metrikleri) ve nesneye yönelimliliği konularında inceleme yapmamıza yardımcı olurlar. Bazı metriklerin test eforu üzerinde etkisi olduğu bilinmektedir. Örneğin çevrimsel karmaşıklık metriğinin yüksek olmasının test eforunu artırdığına dair çalışmalar yapılmıştır [25]. Aynı zamanda kod satır sayısı metriğinden de yola çıkılarak test efor tahminleme çalışmaları da yapılmaktadır. Ancak burada test eforu ile karmaşıklık ve satır sayısı metriklerinin test eforu arasındaki ilişkiyi formüle edilmiş bir şekilde gösteren bir bağıntı bulunmamaktadır.

Ayrıca karmaşıklık ve kod satır sayısı metriklerinin yanı sıra diğer ölçülen metrikler de yazılım karmaşıklığı, büyüklüğü ve mimarisi hakkında önemli ipuçları sergilemektedirler. Bu metriklerinin test edilecek yazılımın test efor süresi üzerinde önemli etkileri olduğu düşünülmektedir.

Bu çalışmayla, kullandığımız tüm metrikler arasındaki ilişki geçmiş proje tecrübelerinden yararlanılarak formüle edilecek ve bir ortak veri havuzu oluşturulacaktır. Daha sonraki test edilecek yazılım projelerinde, bu veri havuzundaki bilgilerden yararlanılarak test efor tahminlemesi kolaylıkla yapılabilecektir.

Bu tahminleme yöntemi sayesinde test efor tahminini kolay, hızlı, güvenilir ve nesnel bir yaklaşımla yapabilmek mümkün olacaktır.

Bu çalışma iki aşamalı olarak gerçekleştirilecektir.

- İlk aşamada seçtiğimiz örnek bir proje üzerinden deneysel bir çalışma yapılacak.
- İkinci aşamada ise yapılan bu deneysel çalışmadan elde edilen sonuçların şu anda elde bulunan verilerle uyumlu olup olmadığı kontrol edilip, sonuçta test eforu ile tahminleme yapmada bir bağıntı oluşturulacaktır.

4.1. Çalışma - 1

Bu çalışmada kullanılan yazılımın test durumları belirlendikten sonra; bu test durumlarının kod üzerinde kapsadığı kısmın kod kalite metrik ölçümleri alınmıştır. Aynı zaman test durumu bazında test efor süreleri alınmıştır. Kalite metrikleri ile test efor süreleri arasında ilişki gösterilmiştir.

4.1.1. Kullanılan yazılım hakkında bilgi

Kullanılan yazılım, yazılım geliştirme yaşam döngüsüne uygun olarak geliştirilmiştir.

Yazılım isterleri kullanılarak test durumları dokümanı oluşturulmuştur. Oluşturulan test durum dokümanında bulunan test durumlarından bu çalışmada kullanılacak olanlar seçilmiştir. Seçim yapılırken birbirlerinden bağımsız test durumlarına seçilmesine, farkın anlaşılabilmesi açısından önem verilmiştir.

Seçilen test durumları; T1, T2, T3, T4, T5, T6, T7, T8, T9 olarak kullanılmıştır.

Bu çalışmada ilk aşamada kullanılacak yazılım kalite metrikleri belirlenmiştir. Yazılım kalite metrikleri iki bazda ele alınmaktadır. Kullanılan yazılımın bu iki bazda ölçümleri alınmıştır.

Metot Bazlı Metrikler; Branches, Call_Pairs, ed(G), Edge_Count, ev(G), ev(G)>4, id(G), iv(G), Lines_with_Nodes, MNT_SEV, Norm_v(G), Param_Count, pv(G), SLOC, vd(G), v(G), vg>10||ev(G)>4 *

(*Metot bazlı metriklerle ilgili detaylı bilgi Bölüm 3.4.1’ de verilmiştir.)

Sınıf Bazlı Metrikler; Avg_v(G), Branches, DIT, Edge_Count, ev(G), id(G), iv(G), Lack_Cohesion, Lines_w_Nodes, Max_ev(G), Max_v(G), MNT_SEV, Node_Count, Norm_v(G), Param_Count, Parent_Count, pv(G), RFC, Sum_v(G), vd(G)*

(*Sınıf bazlı metriklerle ilgili detaylı bilgi Bölüm 3.4.2’ de verilmiştir.)

4.1.2. Kapsam hesaplama

Seçilen test durumları program üzerinden ayrı ayrı koşturularak her bir test durumunun kod üzerinde kapsadığı yerler belirlendi ve bu liste bir sonraki çalışmada kullanılmak üzere saklandı. T1 test durumu için metot kapsama oranları Tablo 4.1.’ de gösterilmektedir. Kapsam oranı yüzde olarak verilmektedir.

Tablo 4.1. T1 test durumu için metot kapsama oranları

Test Durum Numarası	**Metot adı**	**Kapsama oranı (Yüzde olarak)**
T1	AdminPanel_windows.CommonWorks.arrangeToolTip	85.71
T1	AdminPanel_windows.Forms.FrmIntroduction.FrmIntroduction	100
T1	AdminPanel_windows.Forms.FrmIntroduction.FillComboDb	100
	……	

4.1.3. Metrik değeri hesaplaması

Elde bulunan metrik değer tablosu ve kapsama oranı bilgileri kullanılarak her bir test durumunun metot ve sınıf bazlı metrik değerleri hesaplanacaktır.

4.1.4. Metot bazlı hesaplama

Örnek bir hesaplama gösterimi;

Tn test durumu olsun. Tn test durumunun M1, M3, M3 metotlarından oluştuğu ve bu metotları Tablo 4.2.'de verilen oranlarda kapsadığı varsayılsın.

Tablo 4.2. Test durumu - Metot kapsama oranı örneği

Test durumu numarası	**Metot adı**	**Kapsama oranı (yüzde olarak)**
Tn	M1	%85.71
Tn	M2	%100
Tn	M3	%76

M1, M2 ve M3 metotlarının çevrimsel karmaşıklık(v(G)) değerleri Tablo 4.3.' de verilmektedir.

Tablo 4.3. Metot - v(G) metrik değeri

Metot adı	**v(G)**
M1	5
M2	12
M3	3

Tn test durumu için v(G) metriği hesaplaması yapılırken; her bir metodun metrik değeri ve kapsama oranı çarpılır. Sonra bu değerler toplanarak Tn test durumu için v(G) metriği değeri bulunur.

v(G) metriğinin Tn test durumu için hesaplanması eşitlik (4.1)' de gösterilmektedir.

$$Tn_{v(g)} = \sum_{i=1..m} M_{iv(g)} \times Km_i \qquad \textbf{(4.1)}$$

Burada $M_{v(g)}$, metodun v(G) metrik değerini, Km; metodun kapsam yüzdesini göstermektedir.

Örnek için hesaplama yapmak gerekirse aşağıdaki sonuç elde edilir.

$$Tn_{v(g)} = (M_{1v(g)} \times Km_1) + (M_{2v(g)} \times Km_2) + (M_{3v(g)} \times Km_3)$$
$$= (5 \times 85.71/100) + (12 \times 100/100) + (3 \times 76/100) = 18.5655$$

Eşitlik (4.1), tüm test durumlarında metot bazlı tüm metrikler için uygulandığında Tablo 4.4.' deki değerler elde edilir. Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.4. Test durumları - Metot metrikleri

T.D.	Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
T1	59	145	44	779	0	6	27	1941	0	0	28	39	776	23	5	28	28
T2	76	165	54	830	0	7	29	2102	0	0	32	48	825	25	6	31	32
T3	69	156	50	803	0	7	28	2029	0	0	30	43	799	23	5	28	30
T4	74	156	54	751	0	8	31	1834	0	0	33	48	746	27	6	38	33
T5	77	162	55	763	0	7	30	1876	0	0	32	49	756	26	6	34	32
T6	80	164	57	768	0	8	31	1890	0	0	33	50	760	26	6	35	33
T7	130	222	88	1035	1.25	10	42	2638	0.88	1.28	49	78	1025	34	8	55	45
T8	67	127	47	648	0	6	25	1576	0	0	26	43	645	21	5	36	26
T9	97	212	70	1031	0.44	10	40	2664	0.44	0.73	44	64	1027	34	8	52	42

4.1.5. Sınıf bazlı hesaplama

Örnek bir hesaplama gösterimi;

Tn test durumu olsun. Tn test durumunun C1, C2, C3 sınıflarından oluştuğu ve bu sınıfların Tablo 4.5.' de verilen oranlarda kapsadığı varsayılsın.

Tablo 4.5. Test durumları - Sınıf kapsama oranı örneği

Test durum numarası	**Sınıf adı**	**Kapsama oranı (yüzde olarak)**
Tn	Cl1	%15.71
Tn	Cl2	%40
Tn	Cl3	%36

Cl1, Cl2, Cl3 sınıfları için çevrimsel karmaşıklık(v(G)) değerleri Tablo 4.6.' da verilmektedir.

Tablo 4.6. Sınıf - v(G) metrik

Sınıf adı	**v(G)**
Cl1	14
Cl2	12
Cl3	13

Tn test durumu için v(G) metriği hesaplaması yapılırken; her bir sınıfın metrik değeri ve kapsama oranı çarpılır. Sonra bu değerler toplanarak Tn test durumu için v(G) metriği değeri bulunur.

v(G) metriğinin Tn test durumu için hesaplanması eşitlik (4.2)' de gösterilmektedir.

$$Tn_{v(g)} = \sum_{i=1...m} Cl_{iv(g)} \times Kc_i \qquad \textbf{(4.2)}$$

Burada $Cl_{v(g)}$; sınıfın v(G) metrik değerini, Kc; sınıfın kapsam yüzdesini göstermektedir.

Örnek için hesaplama yapmak gerekirse aşağıdaki sonuç elde edilir.

$Tn_{v(g)} = (Cl_{1v(g)} \times K_1) + (Cl_{2v(g)} \times K_2) + (Cl_{3v(g)} \times K_3)$

$= (14 \times 15.71/100) + (13 \times 40/100) + (12 \times 36/100) = 11.6794$

Eşitlik (4.2) tüm test durumlarında sınıf bazlı tüm metrikler için uygulandığında Tablo 4.7.' deki değerler elde edilir. Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.7. Test Durumları – Sınıf Metrikleri

T.D.	Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_ Count	ev(G)	iv(G)	Lines_w_ Nodes	Lack_ Cohesion	MNT_ SEV	Max_ ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_ Count	pv(G)	RFC	Sum_v(G)
T1	53	78	7	55	25	1154	29	45	433	2727	20	32	6	120	1157	26	27	28	481	737
T2	60	88	8	61	28	1353	32	51	502	3056	23	35	7	140	1356	30	31	32	546	839
T3	56	83	8	57	27	1234	30	48	458	2867	21	33	6	130	1237	28	28	30	496	766
T4	62	89	9	64	30	1275	34	52	503	3208	24	37	7	136	1280	31	35	33	658	954
T5	61	88	9	63	29	1219	33	51	478	3144	24	36	7	137	1223	31	33	32	610	895
T6	62	90	9	65	30	1246	34	53	488	3230	24	37	7	142	1250	31	34	33	620	912
T7	93	140	11	86	41	2021	50	81	782	4293	32	86	9	221	2018	41	52	45	943	1425
T8	51	76	6	49	25	1238	28	45	497	2541	19	38	5	100	1240	22	37	26	664	929
T9	85	127	11	80	38	1874	45	74	720	4069	30	76	9	199	1872	38	50	42	818	1262

4.1.6. Test ekibinin eforunun hesaplanması

Aynı zamanda 4 kişilik deneyimli test mühendislerinden oluşan bir test ekibi çalışmanın bu Bölümüne katıldı. İlk aşamada seçilen test durumları test mühendisleri tarafından ayrı ayrı koşturuldu.

Her bir test durumunun koşturulma süresi kaydedildi. Daha sonra test durumu bazında elde edilen bu değerlerin aritmetik ortalaması alındı. Bu yöntemin uygulanmasındaki amaç insan faktörünü minimize edebilmektir.

Test süresi değerleri Tablo 4.8.' de gösterilmektedir.

Tablo 4.8. Test durumu - Test koşturma eforu(adam/sn)

T.D.	Test mühendisi 1	Test mühendisi 2	Test mühendisi 3	Test mühendisi 4	Aritmetik ortalama
T1	60	54	60	63	59.25
T2	31	25	29	44	32.25
T3	71	69	55	57	63
T4	26	67	53	64	52.5
T5	34	30	22	52	34.5
T6	48	49	56	66	54.75
T7	90	98	101	143	108
T8	31	22	34	36	30.75
T9	65	81	67	72	71.25
Toplam	456	495	477	597	506.25

4.1.7. Sonuç

Alınan tüm sonuçlar metrik sonuçları normalize edilip karşılaştırıldığında elde edilen değerler Tablo 4.9.' da gösterilmektedir. Bu tabloda değerler büyükten küçüğe doğru sıralanmaktadır.

Tablo 4.9. Test durumları sıralama kıyaslamaları

Metot bazlı metrik sıralaması	Sınıf bazlı metrik sıralaması	Test efor bazlı sıralama
T7	T7	T7
T9	T9	T9
T6	T6	T3
T4	T4	T1
T5	T5	T6
T3	T3	T4
T2	T2	T5
T1	T1	T2
T8	T8	T8

Tablo 4.9.' da görüldüğü gibi, sıralamalarda bulunan alt ve üst sıralamaları değişkenlik göstermemektedir. Orta kısımlarda bulunan test durumlarının test eforları birbirlerine yakın olduğu için göz ardı edilebilir seviyede yer almaktadır.

Bu çalışma ile yazılım kalite metrikleri ile test eforu arasında bir bağıntı olup olmadığı anlaşılmak istenmiştir. Öncelikle test durumları seçilmiş ve seçilen test durumları için kod kapsam oranı elde edilmiştir. Aynı zamanda seçilen metrikler için yazılımın metot ve sınıf metrik ölçümleri alınmıştır.

Yaklaşımın doğruluğunu kontrol edebilmek için ayrı bir test ekibi de ölçümleri alınan test durumlarını koşturup eforlarını kaydetmişlerdir. Son aşamada test durumlarının metrik ölçümleri ve test efor süreleri kıyaslandığında bu ölçümlerin sıralamasının ve oranlamasının birbirlerine yakın değerler içerdiği görülmüştür.

İkinci çalışma, birinci çalışmada kullanılan ve önemini kanıtlanılan yazılım kalite metrikleri kullanılarak; 3 ayrı yazılım projesinde, yazılım projesi bazında alınan ölçüm ve test süresi bilgilerinden yaralanılarak yapılacaktır.

4.2. Çalışma - 2

Bu çalışma kapsamında 3 ayrı yazılım projesini yazılım kalite metrikleri ve harcanan test eforları kıyaslanacaktır. Bu çalışmada geliştirilmiş 3 ayrı projenin kodları alınmıştır.

Proje 1: Bu proje toplam 3426 kod satır sayısına sahiptir.
Proje 2: Bu proje toplam 17697 kod satır sayısına sahiptir.
Proje 3: Bu proje toplam 40765 kod satır sayısına sahiptir.

4.2.1. Proje 1 ölçüm tablosu

Proje 1'in, birinci çalışmada belirlenen metriklere göre proje metrik ölçümleri metot bazlı ve sınıf yönelimli bazlı olmak üzere alınmıştır.

Proje 1 için metot bazlı ölçümler; Tablo 4.10.' da verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.10. Proje 1 metot metrik ölçümleri

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
753	910	496	2832	3	52.1	222	8213	0	4.67	249	454	3056	149	44	177	235

Proje 1 için sınıf bazlı ölçümler; Tablo 4.11.'de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.11. Proje 1 sınıf metrik ölçümleri

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_ Count	ev(G)	iv(G)	Lines_w_ Nodes	Lack_ Cohesion	MNT_ SEV	Max_ ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_ Count	pv(G)	RFC	Sum_v(G)
55.76	84.13	5.97	48	25.44	805.77	29.16	50.53	301	2400	17.63	41	5	99	803.99	22	17.74	27	382	496

4.2.2. Proje 2 ölçüm tablosu

Proje 2'nin birinci çalışmada belirlenen metriklere göre proje metrik ölçümleri, metot bazlı ve sınıf yönelimli bazlı olmak üzere alınmıştır.

Proje 2 için metot bazlı ölçümler; Tablo 4.12.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.12. Proje 2 metot metrik ölçümleri

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge_Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
5408	8181	3812	17697	47	865.66	2085	75462	43	74.98	2526	3415	19049	1825	584	1448	2154

Proje 2 için sınıf bazlı ölçümler; Tablo 4.13.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.13. Proje 2 sınıf metrik ölçümleri

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_Count	ev(G)	iv(G)	Lines_w_Nodes	Lack_Cohesion	MNT_SEV	Max_ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_Count	pv(G)	RFC	Sum_v(G)
675	882	189	644	446	14306	479	640	3580	9805	391	594	124	1058	14537	192	413	453	2617	3811

4.2.3. Proje 3 ölçüm tablosu

Proje 3'ün, birinci çalışmada belirlenen metriklere göre proje metrik ölçümleri metot bazlı ve sınıf yönelimli bazlı olmak üzere alınmıştır.

Proje 3 için metot bazlı ölçümler; Tablo 4.14.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.14. Proje 3 metot metrik ölçümleri

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
39373	27702	21859	40765	866	6693	6546	195360	835	1318	15472	17570	36743	5657	5407	8359	7114

Proje 3 için sınıf bazlı ölçümler; Tablo 4.15.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.15. Proje 3 sınıf metrik ölçümleri

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_ Count	ev(G)	iv(G)	Lines_w_ Nodes	Lack_ Cohesion	MNT_ SEV	Max_ ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_ Count	pv(G)	RFC	Sum_v(G)
1315	2165	585	730	551.3	11843	1074	1015	2180	20263	537	1826	521	2742	11552	116	302	602	10178	13533

Alınan değerler kıyaslama yapılabilmesi için normalize edildiğinde aşağıdaki değerlere erişilmektedir. Normalize değerler kıyas için önem taşımaktadır.

4.2.4. Proje 1 normalize ölçüm tablosu

Proje 1 için metot bazlı normalize edilmiş ölçümler Tablo 4.16.' da verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.16. Proje 1 metot bazlı normalize edilmiş ölçümler

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge_Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
1.653	2.473	1.895	4.620	0.327	0.6845	2.5147	2.9433	0	0.3340	1.3646	2.1176	5.1930	1.9505	0.736	1.7728	2.4729

Proje 1 için sınıf bazlı normalize edilmiş ölçümler Tablo 4.17.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.17. Proje 1 sınıf bazlı normalize edilmiş ölçümler

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_Count	ev(G)	iv(G)	Lines_w_Nodes	Lack_Cohesion	MNT_SEV	Max_ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_Count	pv(G)	RFC	Sum_v(G)
2.725	2.687	0.765	3.375	2.487	2.9893	1.842	2.962	4.966	7.3918	1.863	1.665	0.79	2.539	2.9895	6.666	2.420	2.495	2.898	2.7802

4.2.5. Proje 2 normalize ölçüm tablosu

Proje 2 için metot bazlı normalize edilmiş ölçümler Tablo 4.18.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.18. Proje 2 metot bazlı normalize edilmiş ölçümler

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge_Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
11.878	22.23	14.56	28.87	5.13	11.373	23.555	27.043	4.89	5.3636	13.843	15.928	32.369	23.919	9.685	14.503	22.66

Proje 2 için sınıf bazlı normalize edilmiş ölçümler Tablo 4.19.' da verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.19. Proje 2 sınıf bazlı normalize edilmiş ölçümler

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_ Count	ev(G)	iv(G)	Lines_w_ Nodes	Lack_ Cohesion	MNT_ SEV	Max_ ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_ Count	pv(G)	RFC	Sum_v(G)
32.98	28.16	24.27	45.28	43.60	53.073	30.27	37.54	59.06	30.198	41.33	24.13	19	27.13	54.053	58.18	56.31	41.86	19.86	21.362

4.2.6. Proje 3 normalize ölçüm tablosu

Proje 3 için metot bazlı normalize edilmiş ölçümler Tablo 4.20.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.1'de verilmiştir.

Tablo 4.20. Proje 3 metot bazlı normalize edilmiş ölçümler

Branches	Call_Pairs	v(G)	SLOC	vgb10or evgb4	vd(G)	id(G)	Edge_Count	evgb4	ed(G)	ev(G)	iv(G)	Lines_w_Nodes	MNT_SEV	Norm_v(G)	Param_Count	pv(G)
86.46	75.291	83.536	66.507	94.5	87.941	73.93	70.012	95.1	94.302	84.792	81.953	62.437	74.13	89.57	83.72	74.86

Proje 3 için sınıf bazlı normalize edilmiş ölçümler Tablo 4.21.' de verilmiştir.

Kullanılan metrikler ile ilgili açıklamalar Bölüm 3.4.2'de verilmiştir.

Tablo 4.21. Proje 3 sınıf bazlı normalize edilmiş ölçümler

Avg_v(G)	Branches	vd(G)	DIT	id(G)	Edge_ Count	ev(G)	iv(G)	Lines_w_ Nodes	Lack_ Cohesion	MNT_ SEV	Max_ ev(G)	Norm_v(G)	Max_v(G)	Node_Count	Parent_Count	Param_ Count	pv(G)	RFC	Sum_v(G)
64.29	69.14	74.95	51.33	53.90	43.937	67.88	59.49	35.97	62.409	56.80	74.19	80	70.32	42.956	35.15	41.26	55.63	77.24	75.857

4.2.7. Sonuç

Tüm normalize edilmiş metrik değerleri toplandığında aşağıdaki kıyaslama tablosu oluşmaktadır. Tüm projelerin metot ve sınıf bazlı metrik ölçümleri Tablo 4.22.' de gösterilmektedir.

Tablo 4.22. Proje metrik kıyaslama tablosu

Proje adı	**Metot metrikleri toplamı**	**Sınıf metrikleri toplamı**
Proje 1	17850.73	5773.08
Proje 2	144678.2	56510
Proje 3	437639.8	84947.5

Kodları alınan projelerin testi için harcanan efor aşağıdaki şekildedir. Tüm projeler için harcanan test eforu ve eforun normalize edilmiş değerleri Tablo 4.23.' de gösterilmektedir.

Tablo 4.23. Proje test efor gösterimi

Proje adı	**Test eforu (adam/saat)**	**Test eforu normalize değerler**
Proje 1	4 saat	% 2.68
Proje 2	55 saat	% 36.91
Proje 3	90 saat	% 60.40

Alınan tüm sonuçlar metrik sonuçları normalize edilip karşılaştırıldığında elde edilen sonuçlar Tablo 4.24.' de gösterilmiştir. Burada, metot bazlı metrik sıralaması, sınıf bazlı metrik sıralaması ve test efor bazlı sıralamanın normalize edilmiş değerleri yer almaktadır.

Tablo 4.24. Test durumları sıralama kıyaslamaları

Proje adı	**Metot bazlı metrik sıralaması**	**Sınıf bazlı metrik sıralaması**	**Test efor bazlı sıralama**
Proje 1	% 1.94	% 2.954	% 2.68
Proje 2	% 16.93	% 37.17	% 36.91
Proje 3	% 81.12	% 59.86	% 60.40

Elde edilen sonuçlara göre test efor tahminlemede sınıf bazlı metrikleri doğruluğunun yüksek olduğu görüldü.

BÖLÜM 5. SONUÇLAR VE ÖNERİLER

Yazılım kalite metrikleri ile test eforu arasında bulunan ilişkinin bulunması amacıyla iki ayrı çalışma yapılmıştır.

İlk çalışmada bir projenin test durumları bazında kapsadığı kodların kalite metrik ölçümleri metot ve sınıf bazlı olarak alınmıştır. Sonraki aşamada ise seçilen test durumları bir test ekibi tarafından koşturulmuş ve bu eforların aritmetik ortalaması alınmıştır. Son aşama olarak yapılan kıyaslamada metriklerin hem metot hem nesne bazında test eforu ile ilişkili olduğu görülmüştür.

İkinci çalışmada ise ilk çalışmada kullanılan metrikler tekrar ele alınmıştır. 3 ayrı büyüklüklerde seçilen yazılım projeleri için proje toplamında kullandığımız metriklerin metot ve sınıf bazlı olarak ölçümleri alınmıştır. Sonraki aşamada ise alınan değerler normalize edilerek karşılaştırılabilecek formata getirilmiştir. Normalize edilen değerler toplanarak her bir proje için metot ve sınıf bazlı olarak toplam metrik ölçüm değeri hesaplanmıştır. Aynı zamanda metrik ölçümleri alınan yazılımların test eforu bilgisi alınmıştır. Son aşamada kıyaslama yapmak için toplam metrik ölçüm değerleri ile test efor bilgileri karşılaştırılmıştır.

Bu karşılaştırma sonucunda metot bazlı metriklerinin sıralaması incelendiğinde; metot bazlı metrik sıralaması ile test eforu sıralamasının aynı olduğu ancak orantılamanın çok farklı sonuçlar ortaya çıkardığı görülmüştür. Metot bazlı metriklerinin test efor süresi belirlemede yetersiz kaldığı anlaşılmıştır.

Sınıf bazlı metriklerin sıralaması incelendiğinde; sınıf bazlı metrik sıralaması ile test eforu sıralamasının aynı olduğu aynı zamanda orantılamanın da çok benzer olduğu

ufak farklılıkların göz ardı edilebilecek kadar küçük olduğu görülmüştür. Sınıf bazlı metriklerinin test efor süresi belirlemede faydalı olduğu anlaşılmıştır.
Sınıf bazlı metrikler kullanılarak tarihsel veri tutmanın anlamlı olduğu yapılan çalışmalar sonucunda anlaşılmıştır. Test eforu ve sınıf metrik değerleri Tablo 5.1.' de gösterilmektedir.

Tablo 5.1. Test eforu – Sınıf metrikleri toplamı

Proje adı	**Test eforu (adam/saat)**	**Sınıf metrikleri toplamı**
Proje 1	4 saat	5773.08
Proje 2	55 saat	56510
Proje 3	90 saat	84947.5
Toplam	149 saat	147230.6

Tablo 5.1.' de bulunan değerler baz alındığında; 1 saatlik bir test faaliyetinde 147230.6 / 149 yani yaklaşık 988 toplam sınıf bazlı metrik ölçümlü koda tekabül eden yazılımın parçasının test edilebildiği görüldü.

Bu çalışma sonucu olarak; yazılım kalite metriklerinin test eforu ile ilişkili olduğu ortaya çıkarıldı. Metot bazlı metrik ölçümlerinin projeler arası test eforu tahminlemede sıralamayı belirleme açısından faydalı olacağı görüldü. Sınıf bazlı metriklerin ise direkt zaman tahmini yapılmasını sağladığı görüldü. Tutulan bu verilerin sonraki yazılım projelerinde test eforunun hesaplanmasında kullanılmasının test eforunu tahminlemesini nesnel, hızlı ve doğru olarak yapmasını sağlayacağı görüldü. Test efor tahminlemesi yapılırken sunulan çalışmadan faydalanılması önerilmektedir.

KAYNAKLAR

[1] IEEE Standart Glossary of Software Engineering Terminology, IEEE The Institute of Electrical and Electronics Engineer, New York, 1990.

[2] MYERS, G., The Art of Software Testing; John Willey & Sons, Inc: New Jersey, pp. 8, 2004.

[3] CRAIG, R. ve JASKIEL, S., Systematic Software Testing.; Artech House Publishers: Boston, pp. 4, 2002.

[4] KANER, C., FALK J., NGUYEN, H.Q., Testing Computer Software; Wiley Computer Publishing, New York, pp. 344, 1999.

[5] http://www.zdnet.com/the-top-10-it-disasters-of-all-time-3039290976/, Erişim Tarihi: 04.02.2013

[6] IEEE STD 1059-1993, IEEE. Software Verification and Validation Plans, New York, 1993.

[7] IEEE STD 1012-1998, IEEE. Software Verification and Validation, New York, 1998.

[8] IEEE STD 829-1998, IEEE. Software Test Documentation, New York, 1998.

[9] IEEE STD 1012a-1998, IEEE. Supplement to IEEE Standard for Software Verification and Validation, New York, 1998.

[10] IEEE STD 1028-1997, IEEE. Software Reviews, New York, 1997.

[11] VICKERS P., An introduction to function point analysis; Northumbria University School: NewCastle, pp. 5-12, 1998.

[12] ALBRECHT, A., Measuring Application Development Productivity: IBM Corporation, Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium, California, USA, pp. 83-92, 1979.

[13] CISA E., DEKKERS T., Testpointanalysis: a method for test estimation; Shaker Publishing BV: The Netherlands, pp. 1, 1999.

[14] ISO/IEC 9126, ISO, Software Engineering Product Quality, Cenevre, 2001.

[15] COCKBURN A., Writing Effective Use Cases; Addison-Wesley Longman: Boston, pp. 15, 2000.

[16] NAGESWARAN, S., Test Effort Estimation Using Use Case Points, Quality Week 2001, San Francisco, pp. 3-4, 2001.

[17] ISO 8402, Quality magement and quallty assurance -- Vocabulary, Cenevre, 1994.

[18] DOD-STD-2168, Military Standard Defense System Software Quality Program, Washington D.C, 1979.

[19] IEEE Standard 610.12, IEEE Standard Glossary of Software Engineering Terminology, New York, 1990.

[20] www.tdk.gov.tr, Erişim Tarihi: 02 01 2013.

[21] KAN, S., Metrics and Models in Software Quality Engineering, 2/E; Addison-Wesley Professional: Boston, pp. 85, 2002.

[22] GILB, T., Software Metrics; Winthrop Publishers: Cambridge, pp.1, 1976.

[23] BOEHM, B., Software Engineering Economics; Prentice-Hall: New York, pp.1 , 1981.

[24] PUTNAM, L., A general empirical solution to the macro software sizing and estimating problem; IEEE Trans Soft: New York, pp. 1, 1978.

[25] McCABE, T., A Complexity Measure; IEEE Transactions On Software Engineering: NewYork, Vol. SE-2, NO.4, 1976.

[26] HALSTEAD, M., Elements of Software Science; Elseiver Science Inc.: North Holland, 1977.

[27] ALBRECHT, A.,Measuring Application Development Productiviy; Proceedings of IBM Applications Development joint SHARE/GUIDE symposium: Monterey, s. 83-92, 1979.

[28] SYMONS, C., Software Sizing & Estimating: Mark II Function Point Analysis; John Wiley&Sons: London, pp.1, 1991.

[29] CHIDAMBER, S., KEMERER, C., A Metrics Suite for Object Oriented Design; IEEE Transactions on Software Engineering: New York, VOL. 20, No:6, pp. 476-493, 1994

[30] http://www.mccabe.com/pdf/McCabe%20IQ%20Metrics.pdf, Erişim Tarihi: 02.01.2013.

[31] MCCABE, T., BUTLER, C., Design Complexity Measurement and Testing; Communications of the ACM: New York, VOL. 32, No:12, pp. 1415-1425, 1989.

[32] VICTOR, B., LIONEL B., MELO W., A Validation of Object-Oriented Design Metrics as Quality Indicators; IEEE Transactions On Software Engineering: New York, VOL 22, No:10, pp. 751-761, 1996

[33] SUBRAMANYAM R, KRISHNAN M, Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects; IEEE Transactions On Software Engineering: New York, VOL 29, No:4, pp. 297-308, 2003.

[34] GYIMOTHY T., FERENC R., SIKET I., Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction; IEEE Transactions On Software Engineering: New York, VOL 31, No:10, pp. 897-910, 2005

EKLER

Ek- A

Metot bazlı metrikler ve sınır değerleri bilgisi Tablo A.1. ve Tablo A.2.' de listelenmektedir.

Tablo A.1. Metot metrikleri

Metrik adı	**Açıklaması**	**Sınır/Beklenen değeri**
Çevrimsel karmaşıklık - (v(G))	Çevrimsel Karmaşıklık bir kod parçası içerisinde yer alan if, for gibi yapıların çokluğu ile doğrudan ilişkilidir.	14
Esas karmaşıklık - (ev(G))	Esas karmaşıklık, bir kod parçası içerisinde yapısal olmayan kod miktarını saptayabilmek için kullanılır.	4
Modül tasarım karmaşıklığı - (iv(G))	Çağrı yapılmayan kod elimine edildiği ortaya çıkan kodun karmaşıklığı modül tasarım karmaşıklığıdır.	7
Dallar – (Branches)	Akış diyagramındaki başlangıç çizgisi ve herhangi bir karar da çıkan çizgi sayılarının toplamıdır.	19
Çağrı çiftleri – (Call Pairs)	Metot içinde diğer metotlara yapılan çağrı sayısına eşittir.	100
Esas sıkışıklık – (ed(G))	Esas sıkışıklık değeri; esas karmaşıklığın 1 eksiğinin çevrimsel karmaşıklığının 1 eksiğine bölünmesi ile bulunur.	0.4
Kenar sayısı – (Edge count)	Akış diyagramındaki çizgi aralık sayılarının toplamıdır.	100
Esas karmaşıklık kontrolü – (ev(G)>4)	Esas karmaşıklık değeri eşik değer olan 4 değerinden büyük ise "Pozitif" değil ise "Negatif" kabul edilir.	Negatif

Tablo A.2. Metot metrikleri (Devamı)

Tasarım sıkışıklığı – (id(G))	Modül sıkışıklığı değeri; modül tasarım karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur.	0.7
Düğüm içeren satır sayısı – (Lines_with_Nodes)	Metot içersindeki düğüm içeren satır sayısı değeridir.	30
Bakım yapma zorluğu – (MNT_SEV)	Bakım yapma zorluğu değeri; esas karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur.	1
Normalize edilmiş çevrimsel karmaşıklık – (Norm_v(G))	Bu değer; çevrimsel karmaşıklığı metot satır sayısına bölünmesi ile bulunur.	0.28
Parametre sayısı – (Param_Count)	Bu değer; metodun aldığı parametre sayısıdır.	5
Patolojik karmaşıklık – (pv(G))	Bu değer; metodun azaltılmış diyagramının çevrimsel karmaşıklığıdır.	2
Sadece kod satır sayısı – (SLOC)	Bu değer; metodun sadece koddan oluşan satırlarının sayısıdır	30
Çevrimsel sıkışıklık – (vd(G))	Bu değer; çevrimsel karmaşıklığın metot kod satır sayısı ve metot karışık kod satır sayısının toplamına bölünmesi ile bulunur.	0.14
Esas karmaşıklık ve çevrimsel karmaşıklık kontrolü – (vg>10||ev(G)>4)	Esas karmaşıklık değeri eşik değer olan 4 değerinden büyük ise ya da çevrimsel karmaşıklık değeri eşik değer olan 10 değerinden büyük ise “Pozitif” diğer durumlarda “Negatif” kabul edilir.	Negatif

Ek- B

Sınıf bazlı metrikler ve sınır değerleri bilgisi Tablo B.1. ve Tablo B.2.' de listelenmektedir.

Tablo B.1. Sınıf metrikleri

Metrik adı	**Açıklama**	**Sınır/Beklenen değer**
Ortalama çevrimsel karmaşıklık - (Avg_v(G))	Nesneye yönelimli programlamada ortalama çevrimsel karmaşıklık hesaplanırken sınıf içinde bulunan metotların çevrimsel karmaşıklığı toplanır ve metot sayısına bölünür.	10
Dallar – (Branches)	Akış diyagramındaki başlangıç çizgisi ve herhangi bir karar da çıkan çizgi sayılarının toplamıdır.	19
Kenar sayısı – (Edge Count)	Akış diyagramındaki çizgi aralık sayılarının toplamıdır.	100
Düğüm sayısı – (Node Count)	Akış diyagramındaki düğüm sayılarının sınıf bazında toplamıdır.	100
Düğüm içeren satır sayısı – (Lines_with_Nodes)	Metot içersindeki düğüm içeren satır sayılarının sınıf bazında toplamıdır.	30
Parametre sayısı – (Param_Count)	Bu değer; metodun aldığı parametre sayılarının sınıf bazında toplamıdır.	5
Kalıtım ağacı derinliği – (DIT)	Kalıtım Ağacı Derinliği sınıfın kalıtım ağacı köküne olan uzaklığıdır.	7
Esas karmaşıklık - (ev(G))	Nesneye yönelimli programlamada esas karmaşıklık hesaplanırken sınıf içinde bulunan metotların esas karmaşıklığı toplanır ve metot sayısına bölünür.	4
Tasarım sıkışıklığı – (id(G))	Modül sıkışıklığı değeri; modül tasarım karmaşıklığın çevrimsel karmaşıklığa bölünmesi ile bulunur.	0.7
Modül tasarım karmaşıklığı - (iv(G))	Nesneye yönelimli programlamada esas karmaşıklık hesaplanırken sınıf içinde bulunan metotların modül tasarım karmaşıklığı toplanır ve metot sayısına bölünür.	7
Bütünlük kaybı - (Lack_Cohesion)	Bütünlük kaybı metriği metotların birbiri ile benzerliğini ölçer. Sınıfın uyumluluğunun düşük olması, sınıfın 2 veya daha fazla alt parçaya bölünmesi gerektiğini gösterir.	75

Tablo B.2. Sınıf metrikleri (Devamı)

En büyük esas karmaşıklık - (Max_ev(G))	Max_ev(G) metriği sınıf içerisinde yer alan metotlar içerisinde, esas karmaşıklığı en yüksek olan metodun esas karmaşıklık değeridir.	4
En büyük çevrimsel karmaşıklık - (Max_v(G))	Max_v(G) metriği sınıf içerisinde yer alan metotlar içerisinde, çevrimsel karmaşıklığı en yüksek olan metodun çevrimsel karmaşıklık değeridir.	10
Bakım yapma zorluğu – (MNT_SEV)	Bakım yapma zorluğu değeri; ortalama esas karmaşıklığın, ortalama çevrimsel karmaşıklığa bölünmesi ile bulunur.	1
Normalize edilmiş çevrimsel karmaşıklık – (Norm_v(G))	Bu değer; ortalama çevrimsel karmaşıklığın, toplam satır sayısına bölünmesi ile bulunur.	0.28
Türetildiği sınıf sayısı – (Parent count)	Bir sınıf için türetildiği sınıf sayısıdır. Sınıf hiyerarşi yapısının karmaşıklığını göstermekte kullanılır.	
Patolojik karmaşıklık – (pv(G))	Nesneye yönelimli programlamada patolojik karmaşıklık hesaplanırken sınıf içinde bulunan metotların patolojik karmaşıklığı toplanır ve metot sayısına bölünür.	2
Bir sınıf için çağrı sayısı – (RFC)	Bir Sınıf İçin Çağrı Sayısı; verilen sınıftan bir sınıfın metotları çağrıldığında, bu sınıfın tetikleyebileceği tüm metotların sayısıdır.	100
Toplam Çevrimsel karmaşıklık - (sum_v(G))	Sınıf içerisinde yer alan metotların çevrimsel karmaşıklığının toplamına eşittir.	70
Çevrimsel sıkışıklık – (vd(G))	Bu değer; ortalama çevrimsel karmaşıklığın, ortalama metot kod satır sayısı ve metot karışık kod satır sayısının toplamına bölünmesi ile bulunur.	0.14

ÖZGEÇMİŞ

Nurhan Yağcı, 01.01.1985'de Samsun'da doğdu. İlk, orta ve lise eğitimini İstanbul'da tamamladı. 2002 yılında Habire Yahşi Yabancı Dil Ağırlıklı Lisesi, Fen-Matematik bölümünden mezun oldu. Aynı yıl İstanbul Üniversitesi Bilgisayar Mühendisliği bölümünü kazandı ve 2006 yılında mezun oldu. 2006-2009 yılların arasında Uyumsoft yazılım firmasında yazılım uzmanı olarak çalıştır. 2009 yılından itibaren TÜBİTAK Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezinde Araştırmacı olarak görev yapmaktadır.