BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİMDALI
BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS PROGRAMI

ÖLÇÜT TABANLI YAZILIM HATA KESTİRİM YAKLAŞIMLARININ İNCELENMESİ VE YENİ BİR YAZILIM HATA KESTİRİM ÖNERİSİ

YÜKSEK LİSANS TEZİ

HAZIRLAYAN

BEGÜM ERKAL

ANKARA - 2020

BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİMDALI
BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS PROGRAMI

ÖLÇÜT TABANLI YAZILIM HATA KESTİRİM YAKLAŞIMLARININ İNCELENMESİ VE YENİ BİR YAZILIM HATA KESTİRİM ÖNERİSİ

YÜKSEK LİSANS TEZİ

HAZIRLAYAN

BEGÜM ERKAL

TEZ DANIŞMANI

DR. ÖĞR. ÜYESİ TÜLİN ERÇELEBİ AYYILDIZ

ANKARA - 2020

BAŞKENT ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği Tezli Yüksek Lisans Programı çerçevesinde Begüm Erkal tarafından hazırlanan bu çalışma, aşağıdaki jüri tarafından Yüksek Lisans Tezi olarak kabul edilmiştir.

Tez Savunma Tarihi: 13 / 01 / 2020

Tez Adı: Ölçüt Tabanlı Yazılım Hata Kestirim Yaklaşımlarının İncelenmesi ve Yeni Bir Yazılım Hata Kestirim Önerisi

Tez Jüri Üyeleri **İmza**

Dr. Öğr. Üyesi, Emre SÜMER, Başkent Üniversitesi

Dr. Öğr. Üyesi, Tülin ERÇELEBİ AYYILDIZ, Başkent Üniversitesi

Dr. Öğr. Üyesi, Damla TOPALLI, Atılım Üniversitesi

ONAY

Prof. Dr. Faruk ELALDI

Fen Bilimleri Enstitüsü Müdürü

Tarih: … / … / 2020

BAŞKENT ÜNİVERSİTESİ
FEN BİLİMLER ENSTİTÜSÜ
YÜKSEK LİSANS TEZ ÇALIŞMASI ORİJİNALLİK RAPORU

Tarih 15 / 01 / 2020

Öğrencinin Adı, Soyadı : Begüm Erkal

Öğrencinin Numarası : 21620201

Anabilim Dalı : Bilgisayar Mühendisliği

Programı : Bilgisayar Mühendisliği Tezli Yüksek Lisans

Danışmanın Unvanı/Adı, Soyadı : Dr. Öğr. Üyesi Tülin Erçelebi Ayyıldız

Tez Başlığı : Ölçüt Tabanlı Yazılım Hata Kestirim Yaklaşımlarının İncelenmesi ve Yeni Bir Yazılım Hata Kestirim Önerisi

Yukarıda başlığı belirtilen Yüksek Lisans tez çalışmamın; Giriş, Ana Bölümler ve Sonuç Bölümünden oluşan, toplam 38 sayfalık kısmına ilişkin, 15/01/2020 tarihinde tez danışmanım tarafından Turnitin adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı % 3'dür.

Uygulanan filtrelemeler:

1. Kaynakça hariç
2. Alıntılar hariç
3. Beş (5) kelimeden daha az örtüşme içeren metin kısımları hariç

"Başkent Üniversitesi Enstitüleri Tez Çalışması Orijinallik Raporu Alınması ve Kullanılması Usul ve Esaslarını" inceledim ve bu uygulama esaslarında belirtilen azami benzerlik oranlarına tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Öğrenci İmzası:

Onay

… / 01 / 2020

Dr. Öğr. Üyesi Tülin ERÇELEBİ AYYILDIZ

TEŞEKKÜR

Çalışmanın başından sonuna kadar ilgisini ve desteğini hiçbir zaman eksik etmeyen, her ihtiyacım olduğunda vaktini ayıran ve değerli fikirleriyle bana yol gösteren tez danışmanım kıymetli hocam Dr. Öğretim Üyesi Tülin ERÇELEBİ AYYILDIZ'a,

Çalışmam boyunca bana sabır ve anlayış gösteren, hayatım boyunca desteklerini benden esirgemeyen, her zaman ve her an yanımda olan, manevi destekleri ile inancımı pekiştiren annem Sibel ERKAL ve babam Cemal ERKAL'a,

Beni her zaman destekleyen, yanımda olan çok değerli sevgili arkadaşım Dr. Elif YILMAZ'a,

Bu süreçte beni yalnız bırakmadıkları için teşekkürlerimi sunuyorum.

ÖZET

Begüm ERKAL

ÖLÇÜT TABANLI YAZILIM HATA KESTİRİM YAKLAŞIMLARININ İNCELENMESİ VE YENİ BİR YAZILIM HATA KESTİRİM ÖNERİSİ

Başkent Üniversitesi Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı

2020

Yazılım projelerinde elde edilen sonucun amacı sadece doğru çalışan bir ürün çıkarmak değildir. Gerçekleştirilen ve ortaya çıkarılan yazılımın kalitesinin değerlendirilmesi, ölçülmesi de gerekmektedir. Yazılım ne kadar kaliteli ve hatalardan arındırılmış olursa bakım onarım sürecinde maliyetler de bir o kadar azalacaktır. Yazılımın kalitesini etkileyen önemli noktalardan biri yazılımdaki hataların sayısıdır. Bu nedenle, geliştirilen yazılımlardaki hataların mümkün olduğunca erken belirlenebilmesi oldukça önem taşımaktadır. Çalışma kapsamında ölçüt tabanlı yazılım hata kestirimi yaklaşımlarından olan kaynak kod ölçütleri incelenmiş ve yazılım hata sayıları ile yazılım kalite ölçütleri arasındaki ilişki analiz edilmiştir. Bu amaçla, 25 adet açık kaynak kodlu java programlama diliyle geliştirilen oyun projesi veri seti olarak kullanılmıştır. Yazılım kalite ölçütlerinin analizinde "Understand" statik kod analiz aracı kullanılmıştır. Çalışmada projelerin hata sayılarının belirlenmesinde ise Spotbugs hata tespit aracından yararlanılmıştır. Yazılım hataları ve yazılım kalite ölçütleri arasındaki ilişkiyi çıkarabilmek için doğrusal regresyon yöntemi uygulanmıştır. Analiz sonuçlarında çıkan sonucun kestirim doğruluğu birisi-dışarıda çapraz doğrulama (Leave one out cross validation - LOOCV) ile yapılmıştır. Sonuçlar, yazılım hata sayısını tahmin etmek için yazılım kalite ölçütlerinden faydalanmanın mümkün olduğunu göstermektedir.

ANAHTAR KELİMELER: Yazılım Hata Kestirimi, Yazılım Ölçütleri, Understand, Spotbugs

ABSTRACT

Begüm ERKAL

INVESTIGATION OF METRIC BASED SOFTWARE BUG PREDICTION APPROACHES AND A NEW SOFTWARE BUG PREDICTION RECOMMENDATION

Başkent University Institute of Science and Engineering

Department of Computer Engineering

2020

The aim of the software project is not only to produce a product that works correctly. It is also necessary to evaluate the quality of the software performed and to measure the quality. The more quality and error-free the software, the lower the costs during the maintenance process. One of the important points affecting the quality of the software is the number of bugs in the software. For this reason, it is very important to detect bugs in the software developed at an early stage. In the scope of the study, source code metrics which is one of the metric based software bug estimation approaches were examined. In the scope of the study, source code metrics from metric based software bug prediction approaches were examined and the relationship between software bug numbers and software quality metrics was analyzed. For this purpose, 25 open source game projects developed with java programming languages were used as data sets. "Understand" static code analysis tool was used to analyze software quality metrics. In the study, Spotbugs bug detection tool was used to determine the bug numbers of the projects. Linear regression method was used to determine the relationship between software bugs and software quality metrics. The prediction accuracy of the results obtained from the analysis results was made with leave one out cross validation. The results show that it is possible to make use of software quality metrics to estimate the number of software bugs.

KEYWORDS : Software Bug Prediction, Software Metrics, Understand, Spotbugs

İÇİNDEKİLER

TABLOLAR LİSTESİ

Sayfa

ŞEKİLLER LİSTESİ

Sayfa

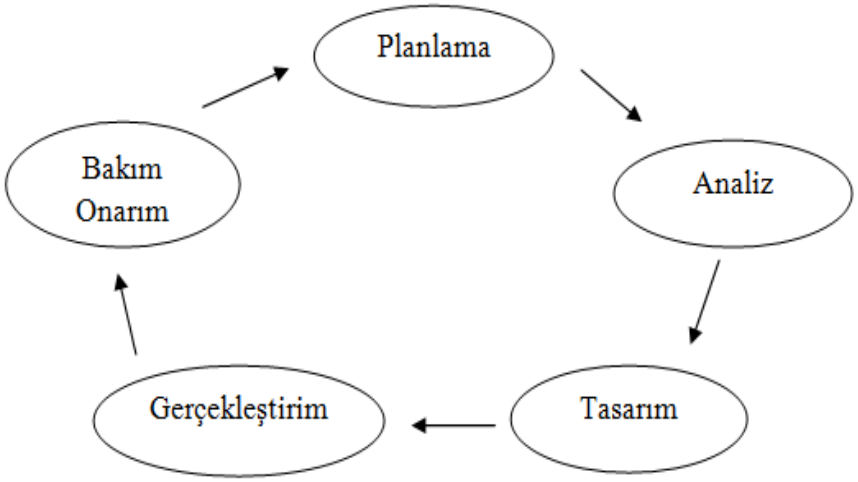
SİMGELER VE KISALTMALAR LİSTESİ

CBO	Coupling Between Objects: Sınıflar Arası Bağımlılık
CK	Chidamber and Kemerer
DIT	Depth of Inheritance Tree: Kalıtım Ağacının Derinliği
IFPUG	International Function Point Users Group
KLOC	Kilo Line of Code: Bin Adet Kod Satırı
LCOM	Lack of Cohesion of Methods: Uyum Eksikliği
LOC	Line of Code: Kod Satırı
LOOCV	Leave One Out Cross Validation
MRE	Magnitude Relative Error
NOC	Number of Children: Alt Sınıf Sayısı
NPM	Number of Public Method
NPRM	Number of Private Method
NSF	Number of Static Field: Statik Alan Sayısı
NSM	Number of Static Method: Statik Metot Sayısı
RFC	Response for a Class: Sınıfın Tetiklediği Metot Sayısı
SRS	Software Requirement Specification: Yazılım Gereksinim Dokümanı
VG	Mccabe Karmaşıklığı
WMC	Weighted Methods per Class: Sınıfın Ağırlıklı Metot Sayısı

1. GİRİŞ

Günümüzde teknoloji hızlı şekilde ilerlemekte ve yazılım dünyası da bu ilerlemelerle birlikte gelişmeler göstermektedir. Bu gelişmelere rağmen geliştiriciler tarafından ortaya çıkarılan yazılımlar bazı hatalar içerebilmektedir. Yazılımdaki hatalar, genellikle üç kategori altında toplanabilmektedir [1]; tasarım hataları; tipik olarak yazılım mimarisinin tasarımındaki hatalardır, uygulama ve teslimat hataları; yazılımcının müşteriye tesliminden sonra ortaya çıkan hatalardır. Bu hataların erken aşamada tespiti büyük rol oynamaktadır. Yazılım hataları; geliştiricilerin projelerdeki hatalı modülleri tekrar tekrar gözden geçirmesine ve bunun sonucunda geliştirilen modüller üzerinde daha fazla zaman harcamalarına neden olmaktadır.

Yazılımda hataların erken tespiti; yazılımın müşteriye teslim edilmeden önce hatasız olmasını ve yazılımın daha kaliteli olmasını sağlar. Şekil 1.1.'de gösterildiği gibi bir yazılım geliştirme yaşam döngüsü sırasıyla gösterilen adımları içerir; planlama, analiz, tasarım, gerçekleştirim ve bakım onarım.



Şekil 1.1. Yazılım Geliştirme Yaşam Döngüsü

Yazılım geliştirme yaşam döngüsünün herhangi bir aşamasında bir yazılım hatasıyla karşılaşıldığında, eğer yazılım geliştirme yaşam döngüsü Waterfall (Şelale) ise analiz ilk adıma tekrar döner. Daha kötü bir durum ise bu durumla müşteriye tesliminden sonra karşılaşılabilmesidir.

Müşteriye teslimden sonra karşılaşılan bu durumda ise aynı adım tekrarlanır. Bu nedenle, yazılım bakım maliyetleri önemli ölçüde artar. Ayrıca yazılım hatasıyla karşılaşan müşteriler açısından güven problemleri ortaya çıkabilir.

Geliştirilen yazılımların kalitesinin sağlanmasında kaynak kodunun sürekli olarak test edilmesi ve doğrulanması önemlidir [2]. Boehm [3]'e göre son kullanıcıya teslim edilecek durumda olan bir yazılımın teslim edilmesinden sonra hesaplanan maliyeti, yazılımın geliştirilmesi sırasında hesaplanan maliyetinden daha yüksektir. Yazılım test araçları, geliştirilen yazılımlardaki hataların tespitinde kullanılır. Test araçlarıyla yazılımın müşteriye teslim edilmeden önce doğruluğu sağlanır [4]. Test araçları da ortaya çıkan hataların her zaman yakalanmasında yeterli olmayabilmektedir [5].

Bu çalışmaya konu olan ve bahsedilen gereksinimler doğrultusunda, yazılım geliştiricilerinin yazılım geliştirmesi periyodunda yazılım ölçütlerine bağlı olarak hata kestiriminin yapılabilmesi amaçlanmıştır. Yapılan çalışmanın aşamaları ilerleyen bölümlerde verilmektedir.

Giriş bölümünde; geliştirilen yazılımlardaki hataların neler olduğu, hatalar tespit edilirken hangi araçlardan yararlanıldığı, hataların erken tespit edilmesinin önemi gibi bilgiler özet olarak verilmektedir.

Literatür çalışması bölümünde; yazılım hata kestirimi konusunda yapılan çalışmalar başlığı altında yazılım hatası, yazılım kalitesi ve yazılım geliştirme sürecindeki hataların nedenleri incelenmektedir.

Yürütülen çalışma bölümünde; yapılan çalışmanın detayları yer almaktadır. Ölçüt tabanlı yazılım hata kestirimlerinin neler olduğu ve kaynak kod ölçütleriyle ilgili açıklamalar bulunmaktadır. Yazılım hatası ve kalite çalışmalarında kullanılabilecek yardımcı araçlar, seçilen ölçütler ve bu ölçütlerin seçilme sebepleri bu bölümde açıklanmaktadır. Çalışmada kullanılan veri setinin detayları, kullanılan analiz araçları ve çalışmanın bulguları yer almaktadır.

Sonuç ve öneriler bölümünde; çalışma sonucunda çıkan bulgular literatürde geçenlerle kıyaslanmaktadır ve literatürde yer alan çalışmalardan farklı çıkan bulguların neler olduğuna değinilmektedir.

2. LİTERATÜR ÇALIŞMASI

Literatür çalışmasında;

Yazılım Hatalarının Kestirilmesi Alanında Yapılmış Olan Çalışmalar başlığı altında;

- Yazılım Hatası,
- Yazılım Kalitesi,
- Yazılım Geliştirme Sürecindeki Hataların Nedenleri,

konuları incelenmiştir.

Yazılım hatalarının özelliklerini anlamak için birçok çalışma yapılmıştır. Yapılan çalışmalar Bölüm 2.1'de detaylı olarak verilmiştir. Yapılan çalışmalara bakıldığında araştırmalar genellikle yazılım geliştirme, test etme ve doğrulama aşamalarında ortaya çıkan yazılım hatalarının incelendiğini göstermektedir [6].

2.1. Yazılım Hatalarının Kestirilmesi Alanında Yapılmış Çalışmalar

Literatür taramasında, yazılım hatalarının kestirilmesi alanında yapılmış araştırmalar incelenmiştir ve bu konuda yapılan çalışmaların bazıları aşağıda verilmiştir:

Lanubile et al., [7] tarafından 1995 yılında yapılan bir çalışmada; Bari Üniversitesinde geliştirilen 27 akademik projenin hataya açık bileşenlerinin belirlenmesi için diskriminant analizi, lojistik regresyon ve mantıksal sınıflandırma yaklaşımları gibi modeller karşılaştırılmıştır. Araştırmalarında "Halstead" ve "Mccabe" karmaşıklık ölçütlerini de içeren 11 ölçüt kullanmışlardır. Değerlendirme sonuçlarında; sınıflandırma oranı, kalite ve doğrulama oranına göre yaptıklarında yaklaşımların hiçbirinin kabul edilebilir sonuçlar vermediği belirtilmiştir. Hata eğilimli ve hata eğilimli olmayan modüller arasında ise ayrım yapılamadığı vurgulanmıştır.

Denaro [8] 2000 yılında yaptığı çalışmada; lojistik regresyon ve 37 ölçüt kullanarak anten konfigürasyon yazılımındaki modüllerin hata sıklığını tahmin etmiştir. R^2 değeri bu çalışmanın değerlendirme parametresi olarak kullanılmış ve ölçütler ile hata eğilimi arasında bir ilişki olduğu gösterilmiştir.

Gyimothy et al., [9] 2005 yılında yaptıkları araştırmada; Mozilla C++ dilinde yazılmış açık kaynak kodlu projeleri üzerinde lojistik regresyon, doğrusal regresyon, karar ağacı ve sinir ağları kullanarak hata kestirimi için "Chidamber ve Kemerer" 'in yaptıkları çalışmada ortaya attıkları nesne yönelimli ölçütleri doğrulama üzerinde çalışmışlardır. "BUGZILLA"

adı verilen hata takip sistemi aracılığıyla, hata veritabanı kullanıp tespit edilen hata sayılarını sağlamışlardır. Araştırmalarında hata kestiriminde ölçütlerden hangilerinin daha iyi bir kullanıma sahip olduğunun üzerine gitmişlerdir. Sınıf düzeyinde ölçütler kullanarak tamlık, doğruluk ve kesinliği değerlendirme ölçütü olarak almışlardır. CBO ölçütünün hata kestiriminde faydalı olduğunu, DIT ölçütünün ise hata tahmini için güvenilir olmadığını bildirmişlerdir.

Hata kestiriminde LCOM ölçütünün iyi olduğunun ancak sonuçlarda çıkan değerinin çok yüksek olmadığını söylemişlerdir. Aynı zamanda NOC ölçütünün de kullanılmaması gerektiğini bildirmişlerdir.

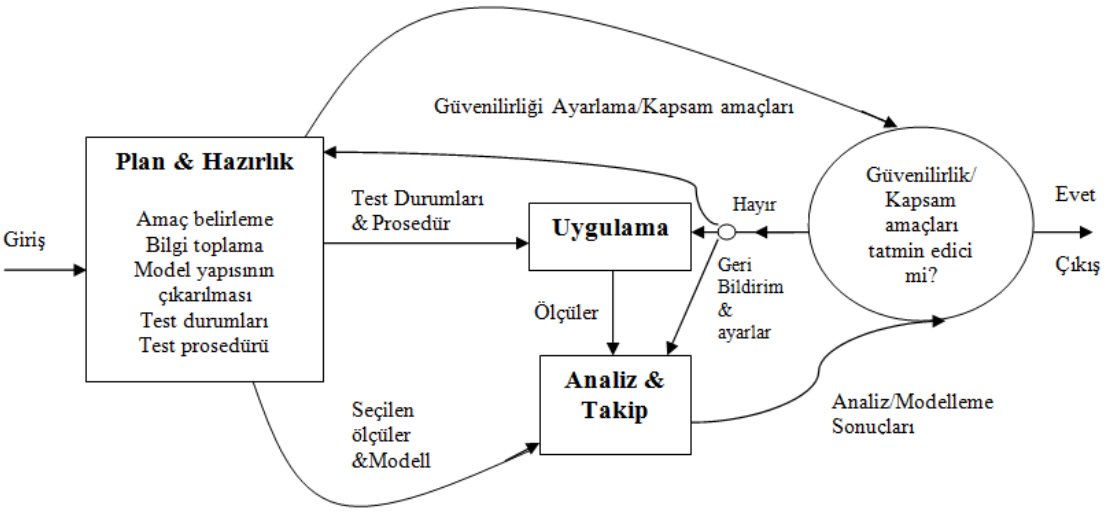
Olague et al., [10] tarafından 2007 yılında yapılan çalışmada; açık kaynaklı "Rhino" projesinin altı versiyonunda üç yazılım ölçüt paketi araştırılmıştır. "Chidamber ve Kemerer" (CK) ölçütleri, nesne yönelimli ölçüt olan "MOOD" ve kalite ölçütü olan "QMOOD" incelemişlerdir. Analizde tek değişkenli ve ikili lojistik regresyon tekniklerini kullanmışlardır. Model doğrulaması için doğruluk parametresi ve ölçümlerin etkilerini incelemek için ise "Spearmen" Korelasyonu uygulamışlardır. "Chidamber ve Kemerer" (CK) ve "QMOOD" ölçütlerinin hata kestirimi için faydalı olduğunu "MOOD" ölçütlerinin ise çok yararlı olmadığını belirtmişlerdir. Ayrıca CK ölçütlerinden en iyi performansın elde edildiğini vurgulamışlardır.

Zimmerman et al., [11] çalışmalarında veri seti olarak hazır Eclipse verilerini kullanarak seçtikleri ölçütlerle hata kestirimi yapmışlardır. Veri seti olarak, dosya ve sürüm seviyelerinden oluşan aynı zamanda altı dosyanın bulunduğu veri setini kullanmışlardır. Çalışmalarının devamında sürümlerin başlangıç ve son durumlardaki hata sayılarını elde etmişlerdir. Hata sayılarını tespit ederken "BUGZILLA" adı verilen hataların takip edilebildiği bir sistemi kullanmışlardır. "Spearman Korelasyonu" ile hesaplamalar yaparak tercih edilen hata sayıları ve seçilen karmaşıklık ölçütlerinin ilişkisini ortaya çıkarmışlardır. Ölçütler arasından toplam kod satırı (LOC) ile Mccabe karmaşıklığı (VG) ölçütlerinde 0,400 değerinin üstünde bir ilişki bulduklarını ve bu ilişkinin dosya düzeyinde olduğunu belirtmişlerdir. Bu dosyaların hataya açıklıkların bulunduğunun simgesi olduğunu söylemişlerdir. Düzey olarak paketlere bakıldığındaysa, kullanılan ölçütlerin büyük çoğunluğunun yüksek ilişki sağladığı ve en yüksek ilişkinin statik metot sayısı (NSM) ve statik alan sayısı (NSF) ölçütleri ile olduğunu söylemişlerdir. Çalışmalarının sonunda karmaşıklık ölçütleri ile hataların kestirilebileceğinin, kodlandırmalardaki karmaşıklığın artmasının kodlarda hata oranını da arttıracağının belirtisi olduğunu savunmuşlardır.

2.1.1. Yazılım hatası

Yazılıma olan bağımlılık arttıkça, yazılım kalitesi de günümüzde daha önem kazanmaktadır. Yazılım; hemen hemen her yerde ve hayatın her aşamasında kullanılmıştır. Hata gibi yazılımı etkileyen sonuçlar, müşteri memnuniyetsizliğine yol açarak yazılımın kalitesini düşürebilmektedir [12]. Bir yazılım hatası hem fazla zaman kaybı hem de maddi açıdan kayıplara neden olmaktadır.

Bir yazılım geliştiricisinin ilk geliştirdiği zamanki deneyimiyle, kendini geliştirip tecrübe kazandıktan sonraki süreçte geliştirdiği yazılımdaki hataları kestirebilme durumu aynı değildir. Bunu başarabilmek için hangi programların diğerlerinden daha çok hataya eğilimli olduğu, hangi aşamalarda daha çok hata olasılığının yüksek olduğunu bilmek gerekir. Yapılan önceki çalışmalar, genel geliştirme sürecinde %27 adam saatinin test ile tüketildiğimi göstermiştir [13]. Test sürecini iyileştirmek için hata kestirimi kullanılabilir. Örnek bir test süreci görseli Şekil 2.1. 'de verilmiştir [12]. Bu modeller; hata kestirimi, risk analizi, efor tahmini, yazılım test edilebilirliği, bakımı ve yazılım geliştirmenin erken aşamalarında güvenilirlik analizlerinde kullanılabilir. Ayrıca yazılım geliştirme yaşam döngüsünün erken aşamalarında yazılım kalitesini tahmin ederek iş riskini en aza indirgemede kullanılabilir. Bunun yalnızca müşteri memnuniyetini arttırmakla kalmayıp, hataların düzeltilme maliyetini de azaltacağı öngörülmüştür [14].

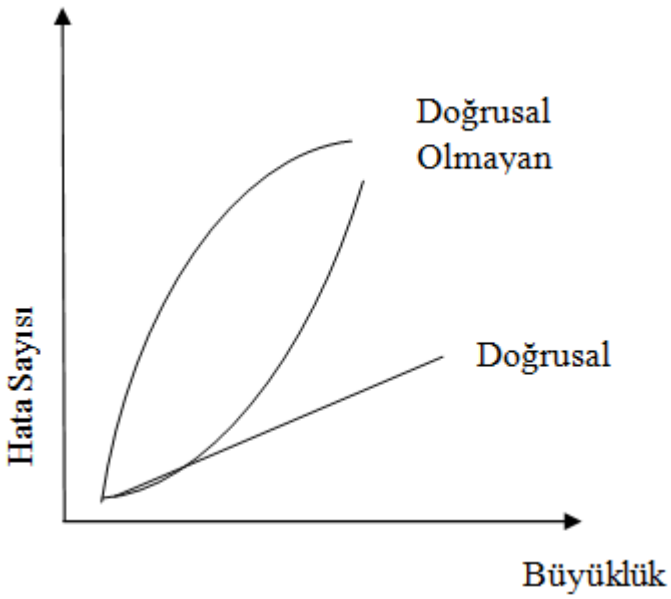


Şekil 2.1. Genel Test Süreci [14]

Khoshgoftaar et al., [15] ise yazılım testinden sonra hata düzeltme maliyetinin önemli oranda yüksek olduğunu bildirmiştir. Erken hata kestiriminin ek bir avantajı ise daha iyi

kaynak planlaması ve test planlamasıdır [16;17]. Bu sebeple zaman ve bütçe dahilinde güvenilir kalitede yazılım geliştirmenin anahtarı, hata kestirim modelleri kullanarak yazılım geliştirme yaşam döngüsünün erken safhasında hataya açık modüllerin tespit edilmesini sağlamaktır.

Yazılımlarda daha büyük modüller, daha küçük modüllerden daha fazla hataya sahip olma eğilimindedir. Bunun nedenleri arasında, daha büyük modüllerin hataların yerleşmesi için daha fazla fırsat sunması, daha büyük modüllerde hataların denetiminin ve testinin daha zor olması ve daha karmaşık bir yapıya sahip olması gibi nedenler vardır. Şekil 2.2.'de büyüklük ve hata arasındaki olası doğrusal ve doğrusal olmayan ilişkilerin örneği verilmiştir. Doğrusal bir ilişki, hata sayısının modül büyüklüğü arttıkça sabit bir şekilde artacağı anlamına gelmektedir. Doğrusal olmayan bir ilişkide ise modülün büyüklüğü arttıkça hata yoğunluğunun da arttığı belirtilir [18].



Şekil 2.2. Büyüklük ve Hata Sayısı Arasındaki olası Doğrusal ve Doğrusal Olmayan İlişkiler [18]

SANS Enstitüsü 2008 ve 2009 yılında, en yaygın ve tehlikeli 25 yazılım hatasını belirlemek için bir çalışma yapmıştır. Çalışmaya yaklaşık 30 kuruluş katkı sağlamıştır. Bunlardan bazıları; Apple, Aspect Security, CERT, Microsoft, MITRE, Oracle, Red Hat and Tata, California Üniversitesi ve Purdue Üniversitesi gibi kuruluşlardır. Çalışmadaki bu 25 güvenlik hata problemi Şekil 2.3.'de gösterildiği gibi üç alana ayrılarak sınıflandırılmıştır [19].

Şekil 2.3. Hata Problemleri [19]

SANS Enstitüsü'nün raporu; test, analiz, denetim ve kalite güvence konularında çalışan şirketler için değerli bir kaynaktır. Geliştirilen bir yazılımın dış dünyaya güvenli bir şekilde sunulmasından önce doğrulanması gereken konular için sağlam bir kontrol listesi sunmaktadır.

2.1.2. Yazılım kalitesi

Yazılım piyasası gittikçe artan bir küresel pazardır, kaliteli ürünler ve hizmetler üretilmedikçe bu pazarın başarılı olarak devam etmesi zordur. Tasarımın kalitesine yazılımın ilk geliştirilme aşamalarında dikkat edilmezse, geliştirilen yazılımın sürdürülebilir olması

risk altındadır [20]. Yazılımın tesliminden sonra hataların bulunması ve bunların düzeltilmesi genellikle proje bütçesinin büyük bir zamanını alır. Bu nedenle, teslimattan önce yapılan hata kestirimlerinin projenin başarısına kalite ve maliyet açısından önemli ölçüde katkısı bulunur.

Evett et al., [21] 1998 yılında yaptıkları çalışmada; askeri bir iletişim sistemi ve telekomünikasyon sistemi üzerindeki genetik programlamanın kalitesini tahmin etmeyi amaçlamışlardır. Çalışmalarında "Mccabe" ve "Halstead" ölçütlerinden 8 ölçüt kullanmışlardır. Genetik programlama ilk kez bu çalışmada kalite tahmini için kullanılmıştır. Çıkan performans sonucu veri setlerinde oldukça başarı göstermiştir.

Kalite konusunda çeşitli araştırmacılar tarafından farklı şekillerde yapılan kalite tanımlamaları aşağıda belirtildiği gibidir:

- Boehm et al., [22] yüksek seviyede kullanıcı memnuniyetinin, taşınabilirliğin, bakım kolaylığının, sağlamlığın ve kullanım uygunluğunun sağlanmasının kaliteyi de getireceğini söylemiştir.
- Musa [23] ise; kalitenin "düşük hata seviyesinde olması, yazılım fonksiyon gereksinimlerine bağlılığı ve yüksek güvenilirlik" birleşimi anlamına geldiğini belirtmiştir.
- Kitchenham and Pfleeger, [24] tanımında kalite ürünün teslimat sonrası kullanımının etkinliği ile ilişkilendirilir.

Yazılım ürününün kalitesi ise; amacına uygunluğu açısından tanımlanır. Her ne kadar amacın uygunluğu donanım ürünleri için kalite açısından tatmin edici tanım olsa da, yazılım ürünleri için aynı şey söz konusu değildir. Bunun sebebi ise; işlevsel olarak doğru olan bir yazılım ürünü düşünüldüğünde örneğin yazılım gereksinim dokümanı (Software Requirement Specification-SRS); belirtilen tüm fonksiyonları doğru bir şekilde yerine getirmektedir. Bu işlevsel olarak doğru olsa bile, neredeyse kullanılamaz bir kullanıcı ara yüzüne sahipse eğer kaliteli bir ürün olduğu söylenemez [25]. Bu sebepledir ki yazılım kalitesi kavramı kolay tanımlanamaz. Yazılımın çeşitli olası kalite özellikleri ve bunun için uluslararası standartları vardır. Bu standartların hata önleme ve gidermede yaklaşık etkileri Tablo 2.1.'de verilmektedir [19].

ISO/IEC 9126-1 standardına göre [26] yazılım kalitesi; içsel kalite özelliği, dışsal kalite özelliği ve kullanımdaki kalite özelliği olarak sınıflandırılır. Bu sınıflandırmaya göre yazılımın kalite özelliği ise;

- İşlevsellik
- Kullanılabilirlik
- Güvenilirlik
- Bakım Kolaylığı
- Verimlilik

üzerinden değerlendirilir.

Tablo 2.1. Standartların Hata Önleme ve Gidermeye Etkisi [19]

Standart	Hata Önlemenin Faydası	Hata Gidermenin Faydası
ISO/IEC 10181 Güvenlik Çatısı	%-25,00	%25,00
ISO 17799 Security	%-15,00	%15,00
Sarbanes-Oxley	%-12,00	%6,00
ISO/IEC 25030 Yazılım Ürün Kalite Gereksinimi	%-10,00	%5,00
ISO/IEC 9126-1 Yazılım Ürün Kalitesi	%-10,00	%5,00
IEEE 730-1998 Yazılım Kalite Güvence Planı	%-8,00	%5,00
IEEE 1061-1992 Yazılım Ölçütleri	%-7,00	%2,00
ISO 9000-9003 Kalite Yönetimi	%-6,00	%5,00
ISO 9001:2000 Kalite Yönetim Sistemi	%-4,00	%7,00
Ortalama	%-10,78	%8,33

2.1.3. Endüstride hata giderme etkisi ve yazılım kalitesi

Genel olarak uçaklar, bilgisayarlar, tıbbi cihazlar ve telefon sistemleri gibi karmaşık fiziksel cihazlar üreten endüstriler en yüksek hata giderme verimine, en iyi kalite önlemleri ve en iyi kalite tahmin kabiliyetlerine sahiptir. Bunların sağlanabilmesi üreten endüstriler için zorunluluktur çünkü kalite sıfır hataya yaklaşmadıkça bu karmaşık cihazlar çalışmayacaktır. Hata giderme etkinliği bakımından ilk 20 sektör aşağıda verildiği şekildedir:

- Hükümet- istihbarat teşkilatı
- Üretim- tıbbi cihazlar
- İmalat- uçak
- Üretim- ana bilgisayarlar
- Üretim- telekomünikasyon anahtarlama sistemleri
- Telekomünikasyon- işlemler

● İmalat- savunma silahları sistemleri

● Üretim- elektronik cihazlar ve akıllı cihazlar

● Hükümet- askeri hizmetler

● Eğlence- film ve televizyon prodüksiyonu

● İmalat- ilaç

● Ulaştırma- havayolları

● Üretim- tabletler ve kişisel bilgisayarlar

● Yazılım- ticari

● İmalat- kimyasallar ve süreç kontrolü

● Bankalar- ticari

● Bankalar- yatırım

● Sağlık hizmeti- tıbbi kayıtlar

● Yazılım- açık kaynak

● Finans- kredi birlikleri

2012 yılı itibariyle 300'den fazla sektör, önemli hacimlerde yazılım üretmektedir. Yazılım hatlarının giderilmesi bakımından dip seviyeye yakın olan gecikmeli endüstrilerden bazılarına; devlet hükümetleri, belediyeler, toptan satış zincirleri, perakende zincirleri, kamu hizmetleri ve finans sektörleri örnek verilebilir [27].

2.1.4. Yazılım geliştirme sürecindeki hataların nedenleri

Bir sistem hatası birkaç nedenden ya da bu nedenlerin birinin sonucundan doğabilmektedir. Nedenleri aşağıda belirtildiği şekilde sıralanabilir [28]:

- Şartname yanlışlığı veya eksik gereksinimler olabilir,
- Şartname, tam olarak müşterinin istediği ve ihtiyaç duyduğu şeyi belirtmeyebilir,
- Teknik özellikler, öngörülen donanım ve yazılım senaryoları göz önüne alındığında uygulanamayacak kadar karmaşık gereksinimler içerebilir,
- Program tasarımı bir hata içerebilir,
- Algoritma eksik veya yanlış olarak uygulanabilir.

Önceki sayfada bahsedildiği gibi gereksinimlerin yanlış anlaşılması, yazılım geliştirme sürecinde kodda hataların ortaya çıkmasına neden olan en yaygın sorundur. Gereksinimlerin eksik veya belirsiz olması ise geliştiricilerin tanımlanmamış gereksinimlere dayanarak yazılım uygulaması geliştirmesine ve bunun sonucunda da eksik bir uygulamanın test edilmesi sorununa neden olur.

Tablo 2.2.'de, yazılım geliştirme sırasında ve sonrasında yazılım uygulamaları için karşılaşılabilecek muhtemel hata potansiyelleri ve oranları verilmektedir. Şiddeti 1 olan hatalar yazılımın hiç çalışmaması durumunu, şiddeti 2 olan hatalar ise yazılımın bazı özelliklerinin devre dışı veya yanlış olması durumunu temsil etmektedir. Şiddeti 1 olan hataların önem derecesi şiddeti 2 olan hatalara göre daha fazladır. En sık görülen hata nedenlerinde geçen komisyon (commission) hatalarını; komisyon toplantılarında alınan talimatlardan kaynaklı olarak ortaya çıkan kodlama ve hatalı düzeltmeler olduğu düşünülmektedir. Hata potansiyellerindeki veriler, 35 yıldan fazla süredir yazılım kalitesini inceleyen IBM Yazılım Kalite Güvence grupları gibi kuruluşlar tarafından yürütülen hataları ve hata gidermelerinin uzun vadeli çalışmalarına dayanmaktadır [19].

Tablo 2.2. Hata Potansiyelleri [19]

Hata Kökeni	İşlev Puanı Başına Hatalar	Şiddeti 1 Olan Hatalar	Şiddeti 2 Olan Hatalar	En Sık Görülen Hata Nedeni
Gereksinimler	1,00	%11,00	%15,00	İhmal
Tasarım	1,25	%15,00	%20,00	İhmal
Kodlama	1,75	%70,00	%57,00	Komisyon
Dokümanlar	0,60	%1,00	%1,00	Belirsizlik
Hatalı Düzeltmeler	0,40	%3,00	%7,00	Komisyon
TOPLAM	5,00	%100,00	%100,00	İhmal

Belirtilen potansiyel hataların dışında ise çeşitli ölçütler ve ölçümlerin hata önleme ve hata giderme üzerinde yaklaşık etkileri de Tablo 2.3.'de verilmektedir. Hata önleme; hata potansiyellerinin azaltılması ile ilgili olduğundan, yüzdeler hataları düşüren yöntemler için negatif değerler göstermektedir. Verilen yüzdelerin yalnızca yaklaşık olduğu ve yalnızca genel etkililik sırasını göstermeye yaradığı belirtilmektedir. Pozitif değerli yüzdeler hata

potansiyelini artıran yöntemleri göstermektedir. IFPUG işlev puanının üst sıralarda olduğu görülmektedir, çünkü gereksinimlerdeki ve tasarımlardaki hataları ölçmek için kullanılır. Tablo 2.3'de en altında yer alan alttaki iki ölçütün, bunlar hata başına maliyet ve kod satırı ölçütü; hata önleme ve gidermede faydalı olmanın aksine zararlı ölçümler olduğu söylenmiştir. Kod satırı ölçütünün gereksinimler ya da tasarım hatalarında hata ölçütü olarak kullanılmaması gerektiğinden bahsetmişlerdir. Sebebini ise yüksek seviyeli dillerde kullanılmasının iyi sonuçlar vermemesine, düşük seviyeli dillerde kaliteyi ve verimliliği olduğundan daha iyi göstermesine bağlayarak açıklamışlardır.

Hata başına maliyet ölçütünün kaliteyi azalttığını, hata içermeyen uygulamalar için bile kullanılmaması gerektiğini ve uygulamalar için yanlış bir ölçüt adayı olduğunu söylemişlerdir [19].

Tablo 2.3. Yazılım Ölçütleri Hata Önleme ve Giderme [19]

Ölçüt	Hata Önlemenin Faydası	Hata Gidermenin Faydası
IFPUG İşlev Puanı	%-30,00	%15,00
Altı Sigma	%-25,00	%20,00
Kalite Maliyeti (COQ)	%-22,00	%15,00
Kök Neden Analizi	%-20,00	%12,00
TSP/PSP	%-20,00	%18,00
Aylık ihtiyaç oranı değişikliği	%-17,00	%5,00
Hedef-soru Ölçütleri	%-15,00	%10,00
Hata Giderme Verimliliği	%-12,00	%35,00
Use-case puanı	%-12,00	%5,00
COSMIC İşlev Puanı	%-10,00	%10,00
Çevrimsel Karmaşıklık	%-10,00	%7,00
Test Kapsamı Yüzdesi	%-10,00	%22,00
Kaçırılan Gereksinimlerin Yüzdesi	%-7,00	%3,00
Hikaye Puanı	%5,00	%-5,00
Hata Başına Maliyet	%-10,00	%-15,00
Kod Satır Sayısı (LOC)	%15,00	%-12,00
Ortalama	%-11,25	%9,06

3. YAPILAN ÇALIŞMA

3.1. Yazılım Ölçütleri

Yazılımların ölçülebilmesi ya da bu ölçümler sonucunda hesaplanmış olan değerlerine "Yazılım Ölçütleri" denilir [29]. Geliştirilmiş olan yazılımlardaki hata tespitlerinde yazılım ölçütleri kullanılabilinmektedir. Böylece yazılımda hangi kod parçalarında hataların olduğu, hangi modüllerin daha önce test edilmesi gerektiği gibi önemli gereksinimlerin sağlanacağı düşünülmektedir. Literatürde farklı sayılarda yazılım ölçütleri olduğu söylenmektedir. Hata kestirim yaklaşımlarının sınıflandırılması söylenen bu ölçütlere göre yapılmaktadır. Tablo 3.1.'de bu yaklaşımlar özetlenerek verilmiştir [29].

3.1.1. Ölçüt tabanlı yazılım hata kestirim yaklaşımları

Tablo 3.1. Ölçüt Tabanlı Hata Kestirim Yaklaşımları [29]

Tip	Gerekçe	Kimin tarafından kullanıldığı
Süreç Ölçütleri	Yazılımlardaki değişikliklerin hatalardan kaynaklanması.	Moser et al. [30]
Önceki Hata Ölçütleri	Gelecekte oluşabilecek hataların geçmiş hatalardan çıkarılabileceği.	Kim et al. [31]
Kaynak Kodu Ölçütleri	Yazılımlardaki karmaşık değişikliklerin daha büyük hatalara sebep olması.	Basili et al. [32]
Değişikliklerin Entropisi Ölçütleri	Basit değişimlerin, karmaşık değişimlere göre daha fazla hataya sebep olması.	Hassan [33]
Kod Çalkantısı Ölçütleri	Kodlamalardaki karmaşıklığın kaynak kod ölçütleriyle daha iyi kestirilebilmesi.	Novel [29]

3.1.1.1. Süreç ölçütleri

Süreç ölçütleri, yazılım geliştirme sürecine fayda sağladığı gibi ölçmenin de değerli bir yoludur. Birçok sorun türü için erken uyarı sağlamanın yanı sıra bir geliştirme sürecinin nasıl takip edileceğine dair günlük rehberlik de sağlarlar. Bir projenin ne kadar verimli ve etkili şekilde çalıştığına dair fikir verir. Proje yöneticilerine, ekip liderlerine ve tüm ekiplere daha iyi karar vermeleri açısından kullanabilecekleri odaklanmış bilgiler sunarak daha etkili olmalarına yardımcı olur. Süreç ölçütlerinin sürekli iyileştirme için önemi büyüktür [34].

Süreç ölçütlerinin geçtiği başka bir tanımlama da ise [35]; yönetim ölçütleri olarak bilindiği ve yazılımı elde etmek için kullanılan işlem özelliklerini ölçmek için kullanıldığı belirtilir. Süreç ölçütleri, nihai sistemin boyutunu tahmin etmeye ve bir projenin programa göre çalışıp çalışmadığının belirlenmesine yardımcı olmaktadır.

3.1.1.2. Önceki kusur ölçütleri

Zimmermann et al., [11] geçmiş hata düzeltmelerinin sayısının gelecekteki düzeltmelerin sayısıyla ilişkili olduğunu belirtir. Zimmermann, programlardaki ve süreçlerdeki hangi faktörlerin geleceğin öngörücüsü olduğunu görebilmek için bu ölçütü kullanmıştır. Geçmişteki hatalar ile ilgili ne kadar çok şey öğrenilirse, gelecekte de bunlardan kaçınma ve daha düşük bir maliyetle daha iyi bir yazılım oluşturulma şansının o kadar iyi olacağını söylemektedir.

3.1.1.3. Kaynak kodu ölçütleri

Literatürde de sıklıkla geçen kaynak kod ölçütleri maddeler halinde sıralanmaktadır [36];

- Chidamber ve Kemerer (CK) Ölçütleri,
- Halstead Ölçütleri,
- Kod Satırı Ölçütü (LOC),
- Mccabe Karmaşıklık Ölçütleri

1970'lerin sonunda kullanılmaya başlanan ölçütler Halstead Ölçütleri ve Mccabe Karmaşıklık Ölçütleridir. Thomas Mccabe tarafından kendi ismini vererek öne sürdüğü Mccabe Karmaşıklığı Ölçütü 1974 yılında ortaya çıkmıştır [37]. 1970 ve 1990 yılları arasında üzerinde en fazla çalışılan ölçütler olarak bilinmektedir. Yapılan çalışmalardaki sonuçlar Li and Henry, [38] ile Chidamber and Kemerer, [39] tarafından öne sürülen nesne yönelimli ölçütleri önemli bir noktaya getirmiştir.

Kod satırı ölçütü; 1960'ların sonlarında kullanılmıştır. Kod Satırının (LOC) bilinen en eski ve kolay yazılım ölçütleri arasında olduğu ve uygulamalardaki boyutların hesaplanmasında kullanıldığı söylenmektedir [40].

CK Metrik Suite başlığıyla 1994 yılında Chidamber and Kemerer (CK); altı adet sınıf tabanlı ölçüt önerisinde bulunmuştur [39]. Önerilen bu ölçütler aşağıda sıralanmaktadır:

- WMC (Sınıftaki ağırlıklı metot sayısı)
- DIT (Kalıtım ağacının derinliği)

- NOC (Alt sınıfın sayısı)
- RFC (Sınıf tarafından tetiklenen metodun sayısı)
- CBO (Sınıfların arasındaki bağımlılık)
- LCOM (Uyum eksikliği)
- **WMC (Sınıftaki ağırlıklı metot sayısı)**

Bir sınıf içerisinde yer alan metot sayısı olarak tanımlanır. WMC ölçütünün sınıf içerisindeki toplam metod sayısını gösterebilmesi için karmaşıklık değerinin bir olarak alınması gerekir. Bir sınıftaki metot sayısı arttıkça child (çocuk) sınıflar üzerindeki potansiyel etki de artmaktadır. Çünkü çocuk (child) sınıflar tanımlanan tüm metotları devralırlar. Ortalama WMC'deki bir artış hata yoğunluğunu arttırmakta ve yazılımdaki kaliteyi de düşürmektedir [39; 41].

- **DIT (Kalıtım ağacının derinliği)**

Kalıtım ağacının derinliği; yazılımda bulunan bir sınıfın sahip olduğu en üst sınıfla en alt sınıfın arasındaki mesafenin değeridir. Bir sınıf için miras alınan ağaç derinliği ne kadar çok olursa, devralınan özellikler ve yeni özellikler arasındaki etkileşime bağlı olarak davranış tahmininde bulunmak o kadar zor olur [42].

- **NOC (Alt sınıfın sayısı)**

Yazılımlarda türetilen alt sınıfların sayısını bildiren ölçüt olmakla birlikte, bu sınıfların doğrudan türetilmiş olmasına dikkat edilir. Bir sınıfın birden fazla alt sınıfa sahip olması yeniden kullanımın yüksek ve hata riskinin fazla olmasına sebep olur. Çok alt sınıfa sahip sınıflarda metotların daha fazla teste ihtiyacından dolayı bu ölçüt sınıflar için yapılan testlerde harcanan bütçeyle ilgili daha detaylı bilgi sağlamaktadır. Bu ölçüt ile yazılımın verimlilik, test edilebilirlik ve yeniden kullanılabilirlik gibi özelliklerini ölçmek mümkündür [42].

- **RFC (Sınıf tarafından tetiklenen metodun sayısı)**

Bu ölçüt, yazılım sınıfında yer alan toplam metot sayısını belirtir ve bu metot sayıları bir sınıfın çağırabildiği potansiyel olarak görülür. Bu değerin hesabında; sınıfın miras aşamasında sahip olduğu private olmayan metotları, sınıfın sahip olduğu metotlar ve sınıfın içerdiği nesnelerindeki metotları kullanılır. RFC ölçütü, sınıfın tetiklediği metot sayısı hakkında bilgi verirken, sınıfın test maliyeti hakkında da bilgi edinmede yardımcı olmaktadır. Sınıftaki tetiklenen metot sayısının artması; sınıf testinin yapılması ve hataların

tespit edilmesi bakımından zorlaşmasına neden olmaktadır. Aynı zamanda bir sınıfın çok sayıda metot çağırması sınıfın karmaşık yapısının da ne kadar yüksek derecede olduğunu belirtir [40; 42].

- **CBO (Sınıfların arasındaki bağımlılık)**

CBO ölçütü; bir sınıfın bağımlı olduğu sınıf sayısıdır [41]. CBO arttıkça, bir sınıfın tekrar kullanılabilirliğinin azalması muhtemeldir. Yüksek CBO değeri aynı zamanda modifikasyon yapıldığında yapılan testleri zorlaştırmaktadır. Genel olarak her sınıf için CBO değerleri düşük tutulmalıdır [43].

- **LCOM (Metot içi uyumsuzluk)**

LCOM ölçütü metotların birbiri ile benzerliğini ölçer. LCOM değeri ne kadar yüksekse sınıf karmaşıklığı da o kadar artmaktadır. Bu sebeple de geliştirme aşamalarında hata yapılma olasılığı yükselir. Sınıflar arasındaki uyumu yüksek tutmak için LCOM değeri düşük tutulmalıdır [43].

Kaynak kodu ölçütlerinin yazılım hatalarıyla arasında ilişki olduğunu gösteren bazı araştırmalar aşağıda yer almaktadır:

Thapaliyal and Verma, [44] yüksek WMC değerinin hata yoğunluğunu arttırdığını ve kaliteyi azalttığını belirtmiştir. Breesam [45] ağacın derinliğinin tasarımın karmaşıklığını gösterdiğini, bir çok kalıtım katmanına sahip sistemi anlamanın zor olduğunu ve yüksek DIT ölçütünün hataları arttırdığını bildirmişlerdir. Chidamber and Kemerer, [39] alt sınıf sayısının yani NOC ölçütünün yüksek olmasının genel olarak daha karmaşık, bakım yapılabilmesinin daha zor ve hataya eğilimin daha yüksek olduğu anlamına geldiğini ve bu sebeple değerinin düşük olması gerektiğini söylemişlerdir. Aynı zamanda Chidamber ve Kemerer RFC ölçütü için gönderilen bir mesajda fazla sayıda metodun tetiklenmesinin; yazılan sınıflardaki hataların tespit edilmesini ve sınıfların test edilmesinin zorlaştırdığını açıklamıştır. LCOM ölçütü için düşük uyumluluğun karmaşıklığı arttırdığını ve bu nedenle geliştirme aşamasında hata yapılmasının yükseleceğini belirtmişlerdir.

Bu bahsedilen çalışmalar dışında yapılan bazı diğer çalışmalarda ise; 1999 yılında Briand et al., [46] çalışmasında lojistik regresyon yöntemi kullanılarak CBO, RFC ve LCOM ölçütlerinin sınıfların hata eğilimini bulmada kullanılabileceği gösterilmiştir. Zhou and Leung, [47] tarafından yapılan çalışmada DIT ölçütü dışındaki CK ölçütlerinin ve kod satır sayısının hata eğilimlerini belirlemede önemli olduğu gösterilmiştir. Yu et al., [48] çalışmasında ise DIT ölçütleri dışındaki ölçütlerin önemi vurgulanmıştır. Tang et al., [49]

DIT, NOC, CBO ölçütlerinin önemsiz olduğunu raporlamışlar ve yaptıkları çalışmalarında lojistik regresyon kullanarak WMC ve RFC ölçütlerinin önemli olduğunu saptamışlardır.

Halstead ölçütleri

Halstead karmaşıklık ölçütü; bir program modülünün karmaşıklığını doğrudan kaynak kodundan ölçmek için geliştirilmiştir. Modüldeki operatör ve operandlardan doğrudan bir niceliksel karmaşıklık ölçütü belirlemek amacıyla geliştirilmiştir [50]. Hesaplanması ise aşağıda belirtildiği gibidir :

Halstead program uzunluğu Eş. 3.1'de;

$$N = N1 + N2 \tag{3.1}$$

Program hacmi Eş. 3.2'de;

$$V = N * \log^2 n \tag{3.2}$$

Program zorluğu Eş. 3.3'de;

$$D = (n1/2) * (N2/ n2) \tag{3.3}$$

Program eforu Eş. 3.4'de;

$$E = V * D \tag{3.4}$$

verilen formüller ile hesaplanmaktadır.

n1 : tekil operatörlerin sayısı

n2 : tekil operandların sayısı

N1 : toplam operatörlerin sayısı

N2 : toplam operandların sayısını belirtmektedir.

Kod satırı ölçütü (LOC)

Yazılım projeleri için en eski ölçüt “kod satırı (LOC)" dır. Bu ölçüt ilk olarak 1960'larda tanıtılmış ve ekonomik, verimlilik ve kalite çalışmaları için kullanılmıştır. Yazılım uygulamalarının ekonomisi “LOC başına dolar” kullanılarak ölçülmüştür. Verimlilik “zaman birimi başına kod satırları” cinsinden ölçülmüştür. Kalite ise, “KLOC başına hata” olarak ölçülmüş ve buradaki "K"; 1000 kod satırının simgesi olarak düşünülmüştür. Kodun LOC ölçütü üç amaç için de oldukça etkili olmuştur.

Daha yüksek seviyeli programlama dilleri oluşturuldukça, LOC ölçütü diğer ölçütlerle karşılaştırılmaya başlanmıştır. LOC ölçütlerinin gittikçe pahalı hale gelen gereksinimler ve tasarımdaki hataları ölçmede yeterli olmadığı görülmüştür. Bu problemler o kadar ciddi bir hal alınca ise 1994 yılında 10 dilde kodlanan aynı uygulamanın 10 versiyonu için hem LOC ölçütleri hem de işlev puanı ölçütleri kullanan bir çalışma sonucunda; birden fazla

programlama dili içeren çalışmalar için LOC ölçütünün yanlış sonuçlara yol açacağı anlaşılmıştır [51].

Mccabe karmaşıklık ölçütü

Ölçüt ilk olarak McCabe tarafından 1974 yılında önerilmiştir. Bu nedenle de yaygın olarak McCabe karmaşıklığı olarak bilinmektedir. Mccabe karmaşıklık ölçütü (V(G)); kontrol akışındaki yolların uzunluğunu ölçer. Akıştaki her dallanma V(G) değerini arttırır. V(G)'yi ölçmek için, genellikle kontrol akış grafiği kullanılır [37]. Mccabe karmaşıklık ölçütü hesaplanması Eş. 3.5'de verilen formül ile yapılmaktadır:

$$V(G) = E - n + p \tag{3.5}$$

E kenar sayısını, n düğüm sayısını p ise bileşen sayısını belirtmektedir.

3.1.1.4. Değişikliklerin entropisi ölçütleri

Hassan [33] çalışmasında; değişikliklerin entropisi ölçütlerinin bir zaman aralığında, sistemde dağıtılmış değişikliklerin nasıl olduğunun ve bunların ölçülmesinden ibaret olduğunu belirtir. Değişiklikler ne kadar yayılırsa, karmaşıklık da o kadar yüksek olur. Yalnızca bir dosyadaki bir değişikliğin, birçok farklı dosyayı etkileyebileceği ve değişikliğin yapılmak zorunda olduğu durumlarda geliştiricinin tümünü takip etmesi gerektiğini belirtir.

3.1.1.5. Kod çalkantısı ölçütleri

Kod çalkantısı; aynı sistemin iki sürümü arasındaki değişimi ölçmek için en yaygın kullanılan ölçütler arasındadır. Bu değişimlerin ölçümü sisteme eklenen ya da sistemden çıkarılanlara değil, matematiksel analizlerle değişikliklerin deltası hesaplanarak sağlanır. Bir sistemin içinde yapılan ufak değişikliklerin, sistemin gerekli veya gereksiz tüm bölümlerinde etkisinin olabileceği üzerine kurulu olan bir ölçüt olarak tanımlanır. Kod çalkantısı olarak tanımlanmasının sebebi ise buna dayanır [29].

3.2. Yazılım Hata ve Kalite Çalışmalarında Kullanılabilecek Yardımcı Araçlar

Bu bölümde yazılım hata ve yazılım kalitesinin tespitinde kullanılan yardımcı araçlar anlatılmıştır. Bu araçlar yazılımlarda ihtiyaç duyulan bazı ölçümleri otomatik ve daha hızlı bir şekilde gerçekleştirerek analiz çalışmalarına oldukça katkı sağlamaktadır.

- FindBugs; Maryland Üniversitesi tarafından geliştirilmiş Java kodları üzerinde çalışabilen açık kaynaklı statik kod analiz aracıdır. 300'den fazla programlama hatasını basit

analiz teknikleri kullanarak tanımlayabileceği kodlamaları tanımaktadır. Aynı zamanda günümüzde popüler olarak kullanılan Eclipse, Netbeans, Jboss gibi programlarda etkin şekilde kullanılabilmektedir [52].

- SpotBugs; FindBugs statik kod analiz aracının yeni çıkan sürümünün adıdır. Yukarıda bahsedilen aynı özelliklere sahip olmakla birlikte bu sürümde bazı güncellemeler ve geliştirmeler yapılarak analiz aracı daha iyi hale getirilmiştir [53].

- Metrics; Java projelerinde birçok yazılım ölçütünü değerlendirip bunları grafiksel olarak sunabilen raporlama aracıdır [54].

- PMD; kaynak kodu tarayarak olası hataları, kullanılmayan ve gereksiz kodlamaları raporlayabilen statik kod analiz aracıdır [55].

- Understand; analiz edilen kodla ilgili ölçütleri toplamada çok etkili bir statik kod analiz aracıdır. Bu ölçütler komut satırı çağrıları ile otomatik olarak çıkarılabilmekte, tablolara aktarılabilmekte ve grafiksel olarak görüntülenebilmektedir. Fonksiyonlar, sınıflar, değişkenler vb. kullanılanlar hakkında bunların nasıl kullanıldığı, nasıl adlandırıldığı yada nasıl etkileşime girdiği ile ilgili bilgileri hızlı şekilde görebilmeyi sağlar [56].

- CheckStyle; açık kaynak kodlu analiz aracıdır ve programcıların kodlama standardına uyan Java kodlamaları yapmasına yardımcı olan bir araçtır. Kaynak kodunun birçok yönünü kontrol edebilmektedir. Örneğin; sınıf tasarım problemleri, metot tasarım problemlerini bulabilmektedir. Ayrıca kodlama düzeni ve biçimlendirme sorunlarını da kontrol edebilme özelliğine sahiptir [57].

- Coverlipse; JUnit testlerinin kod kapsamının görselleştirilmesi için bir Eclipse eklentisidir. Özellikle ilk test geliştirmeleri için kullanışlı bir araçtır. Bunun dışında veri akış analizi gibi kullanımları da mevcuttur [58].

3.3. Seçilen Ölçütler

Önceki bölümlerde bahsedilen literatür çalışmasında da anlatıldığı gibi; kaynak kod ölçütlerinden olan CK ölçütlerinin altısı ve buna ek olarak dört ölçüt dahil edilerek toplam 10 ölçüt ile çalışma yürütülmüştür.

CK ölçütleri; WMC, DIT, NOC, RFC, CBO ve LCOM'dur. Diğer dört ölçüt ise VG; McCabe'in karmaşıklık ölçütü, FANIN, NPRM ve NPM 'dir. McCabe Karmaşıklık Ölçütü (VG); 1974 yılında McCabe tarafından önerilmiştir [37]. Yazılım hata kestirimlerinde sıklıkla karşılaşılan bir ölçüttür. FANIN ölçütü; genel anlamıyla bir yazılımdaki temel sınıf sayısına karşılık gelmektedir [59]. Yapılan bir çalışmada FANIN ölçütünün hata tahmin etmede önemli yerinin olduğu gözlemlenmiştir [60]. NPRM ölçütü; kalıtım yoluyla miras olarak alınmamış private metotların sayısını belirtir. NPM ölçütü; kalıtım yoluyla miras olarak alınmamış public (açık) metotların sayını belirtir [61]. Bu diğer ölçütler hata tahmin etmede daha çok kullanıldığı için tercih edilmiştir.

CK ölçütleri yazılımlarda modüllerin karmaşıklığının belirlenmesinde en çok kullanılan ve bilinen ölçüt kümesidir. Bu ölçüt grubunun yazarları Chidamber and Kemerer [39]; kullanıcılara nesne yönelimli tasarım karmaşıklığının yanı sıra yazılım hatalarının da tahmin edilmesinde bu ölçüt grubunun kullanılmasının yardımcı olabileceğini belirtmişlerdir. Yapılan bazı çalışmalarda CK ölçütlerinin hata tahmin etmede faydalı olacağı söylenmiştir.

Bu çalışmalar şöyle sıralanabilir:

Thapaliyal and Verma, [44] yüksek WMC ölçütünün hata yoğunluğunu arttırdığını ve kaliteyi azalttığını,

Breesam [45] yüksek DIT ölçütünün hataları arttırdığını,

Chidamber and Kemerer, [39] NOC ölçütünün, yani bir sınıftan doğrudan türetilmiş alt sayısının yüksek olmasının hem daha karmaşık bir yapıya hem de daha fazla hataya eğilimli kod yapısına sahip olmanın göstergesi olduğunu,

RFC ölçütünün, yüksek olmasının karmaşıklığı arttırdığı ve bu karmaşıklığa bağlı olarak metod çağrılmalarının tetiklenmesinden kaynaklı sınıflardaki testin ve hata ayıklamanın zorlaşacağını,

Erdemir vd., [62] LCOM ölçütünün düşük uyumlulukta karmaşıklığı arttırdığını ve bu sebeple geliştirme aşamasında hatayı arttıracağını, söylemişlerdir. CK ölçütlerinin çalışmada tercih edilmesinin nedenleri yukarıda sıralanmaktadır. Bununla birlikte diğer seçilen 4 ölçütün neden tercih edildiği ise aşağıda anlatılan çalışmalardaki gibi hata kestiriminde önemi olmasından dolayı tercih edilmiştir.

Couto et al., [63] yaptıkları çalışmalarında, kullandıkları ölçütler arasında NPM ve NPRM ölçütünün yani public (açık) metotların sayısının ve private metotların sayısının hataları kestiriminde yararlı bir ölçüt olduğunu söylemişlerdir.

Yamashita et al., [64] kullandıkları ölçütler arasından Mccabe'in karmaşıklık ölçütü V(G)'nin yaptıkları çalışma sonucunda istatistiksel olarak önemli olduğunu belirtmişlerdir.

Concas et al., [60] çalışmalarında, FANIN ölçütünün önemli bir korelasyon gösterdiği ve hata kestiriminde bu ölçütü de dikkate almanın önemli olduğu belirtmişlerdir.

3.4. Understand Statik Kod Analiz Aracı

Seçilen ölçütler ve kullanılan JAVA projeleri elde edilirken faydalanılan "Understand" kod analiz aracının ekran görüntüleri ve uygulanan adımları EK-1'de verilmektedir.

Çalışma boyunca kullanılan ölçütlerin "Understand" kod analiz aracında isimlerinin bulunmamasından dolayı "Understand" aracının internet sitesinde yer alan karşılıkları bulunmuştur. Ölçütlerin karşılıkları Tablo 3.2.'de verilmiştir. Bulunan bu karşılıklar "Understand" analiz aracının kullanımında ölçütleri seçerken kullanılmıştır. Analiz aracında ölçütlerin karşılıklarının seçilmesiyle ilgili ekran görüntüleri EK-1'de verilmektedir.

Tablo 3.2. Ölçütlerin Understand Analiz Aracındaki Karşılıkları

Ölçüt Adı	Understand Karşılığı
WMC	CountDeclMethod
DIT	MaxInheritanceTree
NOC	CountClassDerived
RFC	CountDeclMethodAll
CBO	CountClassCoupled
LCOM	PercentLackOfCohesion
VG	Cyclomatic
FAN-IN	CountClassBase
NPRM	CountDecMethodPrivate
NPM	CountDecMethodPublic

3.5. SpotBugs Hata Tespit Aracı

FindBugs statik kod analiz aracının çıkan yeni sürümüne verilen adıdır. "SpotBugs" analiz aracı kendi arayüzüne sahip olmakla birlikte bazı geliştirme ortamlarına eklenti (plugin) olarak kurulabilmektedir. Sadece "JAVA" dilinde geliştirilmiş olan hataların tespitinde kullanılabilmektedir [53].

"SpotBugs" hata tespit aracının seçilmesinin nedenleri;

- Java programlama dilinde geliştirilmiş hataları tespit etmesi,
- Eklenti entegrasyonuyla kolay kullanıma hazır olması,
- Mevcut jar, zip ya da war dosyalarının kolaylıkla çalışmasını sağlaması
- Hata örüntülerini kategoriler halinde gruplandırarak her hata önceliğine göre atama yapabilmesi [65].

"SpotBugs" aracı incelendiğinde tespit ettiği hataların yalnızca mantıksal hataları içerdiği görülmüştür. İncelenen bu hataların gerçekliği ise test edilerek doğrulanmıştır. Mantıksal hataların bulunduğu küçük modüller yazılarak analiz aracının testi gerçekleştirilmiştir. Test aşamasında örnek kodlamalar yapıldığında "SpotBugs" aracının hataları tespit ettiği ve bu hataların hangi türlerde hatalar olarak görüldüğünü açıklama şeklinde verdiği görülmüştür.

Test işlemi yapılırken yazılan bir mantık hatası ve sonsuz döngü örneği Şekil 3.1., Şekil 3.2. ve Şekil 3.3. 'de verilmiştir.

```
package comsrcodes.example;
public class HelloWorld{
   public static void main(String[] args) {
    int i;
    for(i=0; i<i; i++){
     System.out.println("Hello World");
    }
    }
}
```

Hata: i değerinin kendisiyle karşılaştırılması, yazım hatası veya mantık hatasını gösterebilir. Doğru şeylerin karşılaştırıldığından emin olun.

Şekil 3.1. Mantık Hatası Örneği

```
public class HelloWorld {
  public static void toplama(int sayi1, int sayi2){
  int sonuc = sayi1 + sayi2 ;
  System.out.println(sayi1 + " + " + sayi2 + " + " + sonuc);
  int i=0;
  while(i<10); {
  System.out.println(" i " + i);
   i++;
   }
}
public static void main(String[] args){
}
```

Hata: Comstrcodes.example.HelloWorld.toplama'da (int, int) için belirgin bir sonsuz döngü var, bu döngüyü sona erdirmenin bir yolu yok gibi görünüyor.

Şekil 3.2. Sonsuz Döngü Hatası Örneği

```
package comsrcodes.example;
public class HelloWorld{
   public static void toplama (int a, int b){
   int sonuc = a + b;
   System.out.println ( a + " + " + b + " = "  + sonuc);
   int i = 0;
   while ( i< 10); {
      if(i==0)
  System.out.println ( " i " + i );
       i++;
  }
}
}
```

Hata: Kullanışsız durum : if koşulunda i değerinin 0'a eşit olmadığı bilinir
Bu koşul daima daha önce daraltılmış olan ilgili değişkenin değeri ile aynı sonuçları verir.
Muhtemelen başka bir şey kastedildi veya bu koşul kaldırılabilir.

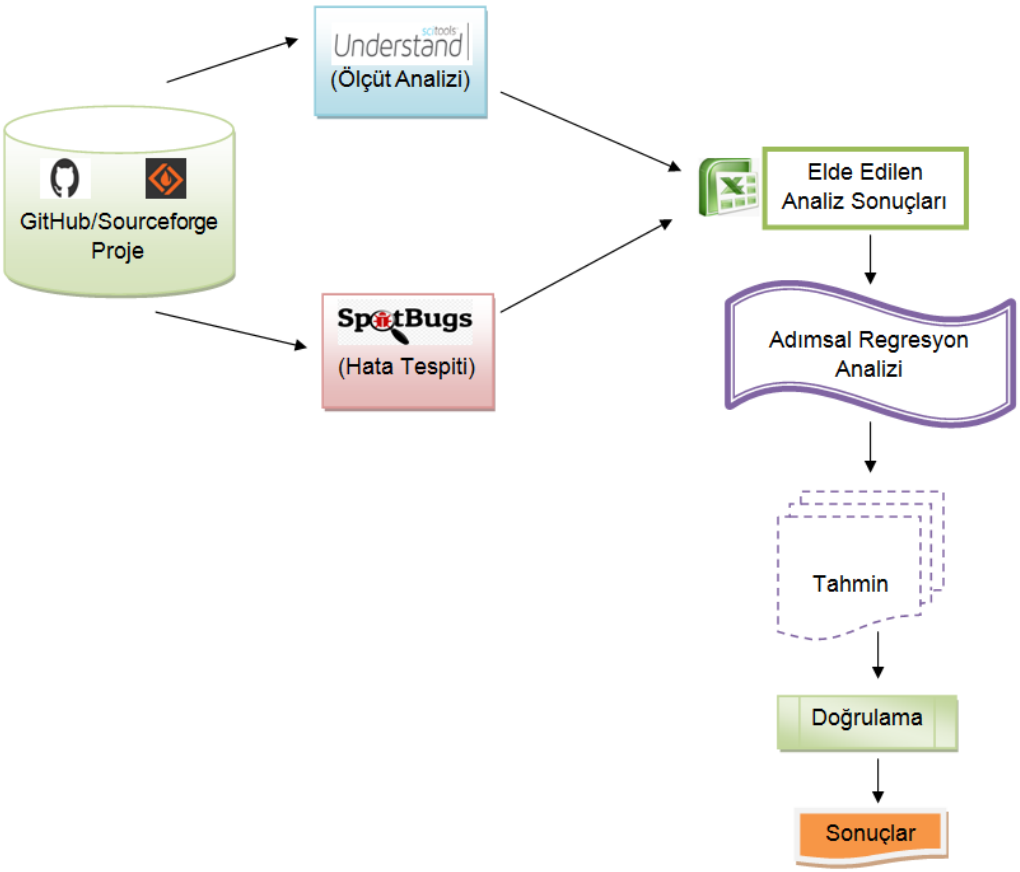
Şekil 3.3. Kullanışlı Olmayan Koşul Hatası Örneği

Seçilen projelerdeki hataların tespitinde kullanılan SpotBugs aracının ekran görüntüleri ve uygulanması adımları EK-2'de verilmektedir.

3.6. Çalışmanın Yöntemi

Yapılan çalışmada ilk olarak proje araştırması yapılmıştır. Şekil 3.4.'de görüldüğü gibi kriterleri sağlayan projeler "Github" ve "Sourceforge" sitelerinden indirilerek analiz aşamasına geçilmiştir. Analizlerde ölçüt analizi bölümünde "Understand", projelerdeki hataların tespitinde ise "SpotBugs" kod analiz aracı kullanılmıştır. Analizlerin sonucunda ölçüt analizlerinden ortalama hesaplamaları yapılarak Excel tablosuna aktarılmıştır. Bulunan hata sayıları da bu Excel tablosuna eklenerek "Adımsal Regresyon Analizi" aşamasına geçilmiştir.

"Adımsal (Stepwise) Doğrusal Regresyon Analizi" tamamlandıktan sonra LOOCV uygulanarak kestirim doğruluğu yapılmıştır. Kestirim doğruluğunda Pred(0,25) ve Pred(0,30) hesaplamalarından sonuçlar elde edilmiştir. Elde edilen sonuçların doğruluğu, araştırılan literatürle desteklenmiştir ve literatürde geçen çalışmalarla çıkan sonuçların değerlendirilmesi sonuç ve öneriler kısmında verilmiştir.



Şekil 3.4. Tez Uygulama Süreci Model

3.6.1. Projelerin seçimi

Çalışma sürecinde incelenen alanlar yazılım hatası alanında daha önceden yapılan çalışmalar olmuştur. Yapılan çalışmada veri seti olarak erişim kolaylığı bakımından 25 adet "Java" programlama diliyle yazılan açık kaynak kodlu oyun projesi tercih edilmiştir.

Kullanılan projelerde dil tercihinin "Java" olmasının sebepleri ise;

- Java dilinin güvenilirliği,
- Farklı platformlarda da çalışabilme bağımsızlığı,
- Java diliyle daha hızlı proje geliştirilebilmesi [66].

Çağımızda oyun projelerinin geliştirilmesinde görülen önemli artış ve popülariteden kaynaklı olarak oyun projeleri tercih edilmiştir. Çalışmada kullanılan projeler, "GitHUB" ve "SourceForge" sitelerinden indirilmiştir. Bu siteler günümüz geliştiricileri tarafından sıklıkla kullanılan platformlardır.

Projelerin bu platformlardan indirilme nedenleri;

- Hem daha hızlı hem de daha güvenilir olması,
- Detaylı raporlara sahiplik etmesi,
- Aynı projede senkronize biçimde birden çok kişinin çalışmasına izin verebilmesi [67].

Proje seçiminde tutarlılık açısından proje boyutlarının kod satır aralığına karşılık gelen gruplandırılmasından yararlanılmıştır. Kod satır sayılarının proje boyutlarıyla olan ilişkisi Tablo 3.3. 'de verilmektedir [68].

Tablo 3.3. LOC Aralığına Göre Proje Boyutları [68]

LOC Aralığı	Proje Boyutu
0-1000	Çok Küçük
1000-10.000	Küçük
10.000-100.000	Orta
100.000-500.000	Büyük
>500.000	Çok Büyük

"Understand" statik kod analiz aracı ile seçilen ölçütlerin analizi sağlanmıştır. Belirlenen ölçütlerin elde edilen karşılıkları seçilerek her projenin analiz sonuçları Understand aracılığıyla excel tablolarına aktarılmıştır.

Geliştirilen yazılımlardaki hataların tespitinde kullanılan "SpotBugs" aracı ise yukarıda daha önceden de bahsedildiği gibi yapılan test işlemlerinin ardından projeler üzerinde uygulanmıştır. "SpotBugs" analiz aracı "Eclipse" geliştirme ortamında plugin olarak kurulmuş ve böylelikle projelerdeki hata sayıları tespit edilmiştir.

Yazılım hatasıyla ilişkisi olan ölçütler tercih sebebi olmuştur. Yazılım hatası ve ölçütler arasındaki ilişki Tablo 3.4.' de verilmektedir. Tablo 3.4. incelendiği zaman; CK ölçütlerinin hata yoğunluğunu arttırdığını gösteren referanslar bulunmaktadır. Bu sebeple belirtilen ölçütlerin analizlerde yer alması tercih edilmiştir.

Tablo 3.4. Yazılım Hatası ve Ölçütler ile İlişkileri

Ölçüt	Hatayla Olan İlişki	Kaynak
WMC	Hatanın artması WMC değerinin yükselmesine ve kalitede azalmaya neden olur.	[43]
DIT	Hatalardaki artış yüksek DIT ölçüt değerine sebep olur.	[44]
RFC	Sınıfların test edilmesinin ve hataların tespitinin zorlaşması, fazla sayıda metodun tetiklenmesine bağlıdır.	[38]
CBO	Hataların artışı sınıfların birbirine bağımlılığının artmasına bağlıdır.	[38]
LCOM	Yazılımların geliştirilmesi sırasında hata oranının artışı, düşük uyumluluk karmaşıklığını arttırmaktadır.	[62]

25 proje içinde gerekli aynı analizler yapılmıştır ve bulunan sonuçlar tablolar halinde ilerleyen sayfalarda verilmiştir.

Yapılan çalışma için seçilen 25 proje ve kod satır sayıları Tablo 3.5.'de verilmektedir. Karşılık gelen kod satır sayılarının analizi de yine Understand statik kod aracıyla sağlanmıştır. Projelerin tamamı 10-100 KLOC aralığında olup orta ölçekli projelerden oluşmaktadır. Bu aralıklarda projeler seçilerek yapılan çalışmanın tutarlılığının sağlanması amaçlanmıştır.

Tablo 3.5. Kullanılan Projeler ve Kod Satır Sayıları

Proje No	Oyun Projesinin Adı	KLOC	Proje No	Oyun Projesinin Adı	KLOC
1	Macchiatodoppio_code	55K	14	Freelords_git	54K
2	Ysoccer_code	51K	15	Mtastrategy_code	15K
3	Sketchy_Truck_master	53K	16	Game_theory_poker_master	12K
4	Perfectday_code	24K	17	Gogui_code	50K
5	dune2themaker4j_master	20K	18	Hale_code	85K
6	Simplytrain_code	41K	19	Cyber_decker_code	16K
7	JCloisterZone_master	43K	20	NOVA_Core_master	70K
8	Capa_code	96K	21	Gomule_git	26K
9	Rails_code	74K	22	Wrath_code	97K
10	Antiyoy_master	53K	23	Toz_code	74K
11	Freemars_code	79K	24	Linkinland_code	15K
12	Lobby_code	15K	25	PuzzleGame_master	42K
13	Nestorlab_code	24K			

Çalışma sürecinde kullanılan projelerin isimleri, kod satır sayıları, web sayfaları ve projelere son erişim tarihleri ile ilgili detaylar Tablo 3.6.'de verilmektedir.

Tablo 3.6. Kullanılan Projeler, Kod Satır Sayıları, Web Sayfaları ve Son Erişim

Proje No	Oyun Projesinin Adı	KLOC	Web Sayfası	Son Erişim Tarihi
1	macchiatodoppio-code	55K	https://sourceforge.net/projects/macchiatodoppio/	Eylül 19
2	ysoccer-code	51K	https://sourceforge.net/projects/ysoccer/	Eylül 19
3	Sketchy-Truck-master	53K	https://github.com/sebnil/Sketchy-Truck	Eylül 19
4	perfectday-code	24K	https://sourceforge.net/projects/perfectday/	Eylül 19
5	dune2themaker4j-master	20K	https://github.com/Fundynamic/dune2themaker4j	Eylül 19
6	simplytrain-code	41K	https://sourceforge.net/projects/simplytrain/	Eylül 19
7	JCloisterZone-master	43K	https://github.com/farin/JCloisterZone	Eylül 19
8	capa-code	96K	https://sourceforge.net/p/capa/code/HEAD/tarball	Eylül 19
9	rails-code	74K	https://sourceforge.net/p/rails/code/ci/master/tarball	Eylül 19
10	Antiyoy-master	53K	https://github.com/yiotro/Antiyoy	Eylül 19
11	freemars-code	79K	https://sourceforge.net/projects/freemars/	Eylül 19
12	lobby-code	15K	https://sourceforge.net/projects/lobby/	Eylül 19
13	nestorlab-code	24K	https://sourceforge.net/projects/nestorlab/	Eylül 19
14	freelords-git	54K	https://sourceforge.net/projects/freelords/	Eylül 19
15	mtastrategy-code	15K	https://sourceforge.net/p/mtastrategy/code/HEAD/tarball	Eylül 19
16	game-theory-poker-master	12K	https://github.com/adamsmith/game-theory-poker	Eylül 19
17	gogui-code	50K	https://sourceforge.net/projects/gogui/	Eylül 19
18	hale-code	85K	https://sourceforge.net/projects/hale/	Eylül 19
19	cyber-decker-code	16K	https://sourceforge.net/projects/cyber-decker/	Eylül 19
20	NOVA-Core-master	70K	https://github.com/NOVA-Team/NOVA-Core	Ekim 19
21	gomule-git	26K	https://sourceforge.net/projects/gomule/	Ekim 19
22	wrath-code	97K	https://sourceforge.net/projects/wrath/	Ekim 19
23	toz-code	74K	https://sourceforge.net/p/toz/code/HEAD/tarball	Ekim 19
24	linkinland-code	15K	https://sourceforge.net/projects/linkinland/	Ekim 19
25	PuzzleGame-master	42K	https://github.com/victordibia/PuzzleGame	Ekim 19

3.6.2. Korelasyon ve regresyon analizi

Korelasyon; istatiksel olarak değişkenler arasındaki ilişkileri belirlemek için kullanılır. Korelasyon katsayısı +1,00 ve -1,00 aralığında değişim gösterir. +1,00 korelasyon katsayısı; iki değişken arasında pozitif bir lineer ilişki olduğunu belirtirken -1,00 katsayısı iki değişkenin negatif anlamda bir ilişki gösterdiğini belirtir. En çok kullanılan korelasyon katsayıları Spearman korelasyon katsayısı ve Pearson korelasyon katsayısıdır. [69].

Regresyon analizi; bağımlı bir değişken ile bir veya daha fazla bağımsız değişken arasındaki ilişkilerin tahmininde kullanılan bir dizi istatistiksel yöntemdir. Değişkenler arasındaki ilişkinin gücünü değerlendirmek ve aralarındaki ilişkiyi modellemek için

kullanılır. Regresyon analizi doğrusal, çoklu doğrusal ve doğrusal olmayan gibi çeşitli varyasyonları içerir. En yaygın modeller basit doğrusal ve çoklu doğrusaldır. Doğrusal olmayan regresyon analizi, bağımlı ve bağımsız değişkenlerin doğrusal olmayan bir ilişki gösterdiği daha karmaşık veri kümeleri için yaygın olarak kullanılır [69].

Çalışmada yapılan korelasyon analizinde; hata sayıları ile ölçütlerin arasındaki korelasyon ilişkisi "Pearson Korelasyonu" ile elde edilmiştir ve sonuçlara bakıldığında LCOM ölçütü ile 0,602, NOC ölçütü ile ise 0,468 değerinde ilişki olduğu gözlemlenmiştir.

3.6.3. Bağıl hata (Magnitude Relative Error-MRE)

Bağıl hata; önerilen model tarafından tahmin edilen değerler ile gerçekte tahmin edilen değerler arasındaki fark hesabından yola çıkarak kullanılan bir ölçüdür [70]. Literatürde görülen belirli bir doğruluk seviyesi için gerçekleştirilen gözlemlerin oranı Pred(x) kestirim düzeyi olarak tanımlanır [71]. Bağıl hata (MRE) ve Pred(x) ve eşitlikleri Eş. 3.6 ve Eş. 3.7'de formülleriyle birlikte verilmektedir.

$$MRE = \frac{|Egerçek - Etahmin|}{Egerçek} \tag{3.6}$$

$$\text{Pred (x)} = \frac{\text{k}}{\text{N}} \tag{3.7}$$

Bağıl hata değerinin "x" değerinden düşük olduğu projelerin sayısı, Eş. 3.7'deki "k" değerini vermektedir. Yazılan "N" değeri; eşitlikte de verildiği gibi proje sayılarının toplamını ifade eder. Pred değer hesaplamasında kestirim modelleri yaratma durumlarında sıklıkla kullanılan "x" değerleri 0,25 ve 0,30 değerleridir [71;72].

Kabul edilen bağıl hatanın (MRE) değerinin Conte et al., [71] tarafından 0,25 değerinden düşük ve Pred(0,25) hesaplamasının ise 0,75 değerine eşit ya da bu değerden fazla olması gerektiği belirtilir. Pred(0,30) değerininse Tate and Verner, [72] tarafından 0,70 değerinden yüksek olması gerektiği söylenir.

Yukarıda da bahsedildiği gibi gerçek hata sayısından tahmin edilen hata sayısı çıkarıldığında elde edilen değerin gerçek hata sayısına bölümüyle bağıl hata (MRE) değerleri elde edilmiştir.

3.7. Analiz ve Bulgular

Eclipse geliştirme ortamına kurulan projelerin hata sayılarının tespitinde yukarıda da bahsedildiği gibi SpotBugs aracı kullanılarak projelerdeki hata sayıları tespit edilmiştir ve her bir proje için hata sayıları belirlenmiştir. Tespit edilen bu hata sayıları Tablo 3.7.'da sunulmaktadır.

Tablo 3.7. Projelerde Tespit Edilen Hata Sayısı

Proje No	Hata Sayısı	Proje No	Hata Sayısı
1	61	14	18
2	41	15	18
3	58	16	24
4	56	17	16
5	59	18	37
6	38	19	46
7	51	20	20
8	40	21	16
9	96	22	38
10	42	23	25
11	28	24	30
12	49	25	27
13	12		

Projelerin analizinde bölüm 3.6'da da bahsedildiği gibi Understand aracı kullanılmıştır. Projelerdeki ölçüt değerleri toplanarak sınıf sayısını bölünmüştür ve bölümünden elde edilen değerler her bir projenin ölçüt değerlerinin ortalamasının hesabında kullanılmıştır. Bu hesaplamadan elde edilen değerler Tablo 3.8. 'de verilmektedir.

Tablo 3.8. Projelerde Kullanılan Ölçütlerin Hesaplanan Ortalama Değerleri

PROJE NO	WMC	DIT	RFC	NOC	CBO	LCOM	FANIN	VG	NPRM	NPM
1	7,38	1,93	4,38	0,59	7,79	31,99	1,30	0,83	3,54	0,28
2	7,68	2,46	3,55	0,79	8,84	15,24	1,05	0,64	1,78	0,32
3	9,33	2,31	6,97	0,52	2,68	35,83	1,34	0,23	5,50	0,72
4	4,21	1,50	7,98	0,43	6,52	33,92	1,21	1,45	6,99	0,85
5	10,87	1,75	7,47	0,61	5,07	35,24	1,28	0,38	7,23	0,16
6	9,62	1,17	4,71	0,11	7,34	21,67	1,60	0,62	3,80	0,70
7	10,09	1,82	5,51	0,44	6,36	34,89	1,34	0,56	4,81	0,56
8	8,34	1,50	4,35	0,24	8,09	27,53	1,49	0,75	3,53	0,43
9	25,77	1,65	10,32	0,43	14,12	51,37	1,43	0,32	8,29	0,83
10	13,33	1,85	7,43	0,78	4,83	24,72	1,05	0,79	4,65	1,92
11	9,63	1,89	5,16	0,35	6,11	23,86	1,30	0,54	3,86	1,23
12	15,23	1,21	7,51	0,13	9,53	33,47	1,41	0,81	6,42	0,61
13	7,72	1,12	4,43	0,09	6,03	16,54	1,58	0,30	3,23	1,19
14	4,62	1,29	3,20	0,02	8,05	26,73	1,25	0,60	2,77	0,42
15	6,78	1,59	4,36	0,10	5,13	26,55	1,26	0,61	3,83	0,29
16	12,52	1,10	3,38	0,10	8,14	27,48	1,19	1,16	2,63	0,65
17	12,53	1,32	5,43	0,24	7,65	17,01	1,34	0,27	3,83	1,58
18	13,77	1,49	6,97	0,17	9,74	31,68	1,51	0,56	5,61	0,83
19	19,88	1,02	4,65	0,02	5,25	47,45	1,00	0,92	1,54	0,06
20	9,45	1,87	5,49	0,43	3,99	27,92	1,31	0,16	4,96	0,35
21	17,53	1,10	5,56	0,10	6,72	22,34	1,52	0,81	4,82	0,63
22	23,61	1,57	11,58	0,38	7,05	37,89	1,33	0,63	8,27	1,20
23	16,15	1,66	8,04	0,15	7,06	30,05	1,85	0,92	6,94	0,18
24	28,45	1,66	13,24	0,16	3,46	50,90	1,48	0,94	9,73	3,04
25	13,37	2,68	7,58	0,54	5,32	31,57	1,20	0,76	6,46	0,49

Yapılan çalışma için seçilen projelerde "Minitab" istatistik aracı kullanılarak hata sayılarıyla ölçütlerin hesaplanan ortalama değerleri arasındaki ilişki çıkarılmıştır. "Adımsal (Stepwise) Doğrusal Regresyon" analizi uygulanarak seçilen ölçütler ve projelerdeki hata sayıları arasındaki ilişki çıkarılmıştır. Analiz sonucu Tablo 3.9. 'da verilmektedir. Tabloya bakıldığında R^2 değerinin yüksek olarak görüldüğü yedinci adım seçilerek, Eş. 3.8'de hata sayısı için bulunan denklem verilmektedir.

$$\textbf{Hata Sayısı} = (\text{LCOM} * 2{,}01) + (\text{NOC} * 78) + (\text{CBO} * 2{,}94) + (\text{WMC} * -0{,}99) + (\text{DIT} * -16{,}4) + (\text{FANIN} * 29) + (\text{NPRM} * -2{,}1) - 58{,}33 \quad (3.8)$$

Tablo 3.9. Adımsal Doğrusal Regresyonun Sonucu

Adım	1	2	3	4	5	6	7
Sabit	0,7379	-13,128	-36,953	-38,7191	-20,6953	-41,4338	-58,33
LCOM	1,21	1,25	1,22	1,65	1,71	1,79	2,01
NOC		40,6	44,1	41	61	69,7	78
CBO			3,45	3,77	3,37	3,17	2,94
WMC				-1	-1,05	-1,18	-0,99
DIT					-14	-16,2	-16,4
FANIN						16,5	29
NPRM							-2,1
S	15,7	12,7	9,72	8,61	7,9	7,52	7,14
R-Sq	36,21	60,01	77,62	83,27	86,63	88,52	90,22
R-Sq(adj)	33,43	56,38	74,43	79,93	83,11	84,7	86,19

Tablo 3.9.'a bakıldığı zaman, elde edilen sonuçlarda en güçlü değişkenden itibaren diğer değişkenlerin de eklenmesiyle sonuçların yansıtıldığı görülmektedir. En güçlü değişkenlerin LCOM, NOC, CBO, WMC, DIT, FANIN ve NPRM değişkenlerinin olduğu ve bu değişkenlere bakıldığında anlamlı şekilde hata sayılarının kestirilebildiği söylenebilir.

3.7.1. Kestirim doğruluğu (Prediction accuracy)

Yapılan çalışmadaki 25 proje için regresyon analizinde çıkan denklem eşitliğinde ölçüt değerleri yerine konulduğunda tüm projelerden kestirilmiş olan hata sayıları bulunmuştur. Tablo 3.10.'da "Spotbugs" analiz aracıyla elde edilen gerçek hata sayıları ile birlikte denklemde yerine konularak hesaplandığında ortaya çıkan hata sayıları verilmektedir.

Tablo 3.10. Gerçek Hata Sayıları ve Kestirilmiş Hata Sayıları

Proje No	Hata Sayısı	Kestirilmiş Hata Sayısı	Proje No	Hata Sayısı	Kestirilmiş Hata Sayısı
1	61	66,20	14	18	25,32
2	41	38,67	15	18	13,62
3	58	42,31	16	24	27,18
4	56	54,20	17	16	13,83
5	59	57,46	18	37	41,18
6	38	25,09	19	46	43,39
7	51	53,73	20	20	30,61
8	40	42,45	21	16	12,69
9	96	91,46	22	38	40,27
10	42	43,54	23	25	30,39
11	28	23,95	24	30	33,72
12	49	39,58	25	27	26,93
13	12	12,68			

MRE değerleri hesaplanırken; gerçek hata sayısından tahmin edilen hata sayıları çıkarılarak, elde edilen değerin hesaplanan gerçek hata sayılarına bölünmesiyle sağlanmıştır. Tablo 3.11.'de bağıl hata (MRE) değerleri hesaplanarak verilmiştir.

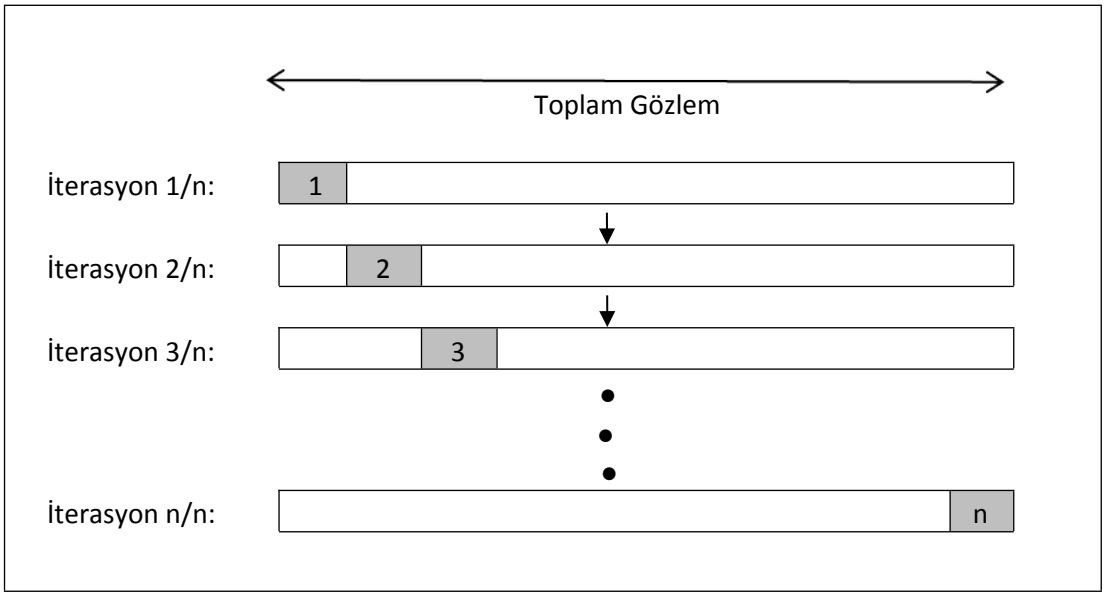
Tablo 3.11. Hesaplanan Bağıl Hata (MRE)

Proje No	Bağıl Hata	Proje No	Bağıl Hata
1	0,08	14	0,40
2	0,05	15	0,24
3	0,27	16	0,13
4	0,03	17	0,13
5	0,02	18	0,11
6	0,33	19	0,05
7	0,05	20	0,53
8	0,06	21	0,20
9	0,04	22	0,05
10	0,03	23	0,21
11	0,14	24	0,12
12	0,19	25	0,00
13	0,05		
Kestirimin Doğruluğu Pred(0,25) = 0,84 Pred(0,30) = 0,88			

Conte et al., [71]' nin de belirttiği gibi kestirim başarısının iyi olduğu; Pred(0,25) değeri için elde edilen değerin 0,84 geldiği ve bu değerin 0,75'den büyük olduğu, Tate and Verner, [72]'in belirttiği gibi elde edilen değerin Pred(0,30) için 0,88 geldiği ve bu değerin de 0,70'den büyük olduğu görülmektedir.Verilen referanslardan yola çıkılarak kestirimin başarılı olduğu desteklenmektedir.

3.7.2. Birisi-dışarıda çapraz doğrulama (Leave one out cross validation - LOOCV)

Birisi-dışarıda çapraz doğrulama yönteminde, n adet gözlemin bulunduğu veri setinde, veri seti n adet parçaya ayrılır ve her iterasyonda veri setinden birisi çıkarılarak geriye kalan gözlemlerle eğitilmesi sağlanır. Bu süreç tüm gözlemler için tekrarlanır [73]. Birisi dışarıda çapraz doğrulama için örnek bir görsel Şekil 3.5.'de verilmektedir [74].



Şekil 3.5. LOOCV Görsel Örneği [74]

Çalışmadaki 25 proje için birisi-dışarıda çapraz doğrulama (LOOCV) kullanılarak bağıl hata(MRE) değerleri hesaplanmış ve oradan Pred(0,25) ile Pred(0,30) için elde edilen değerlerin hesaplaması yapılmıştır. LOOCV yapılırken önce ilk proje çıkarılarak kalan 24 proje için tekrar " Adımsal (Stepwise) Doğrusal Regresyon" analizi uygulanmıştır. Analiz sonucunda çıkan denklem ilk proje için yerine konularak kestirilen hata sayısı hesaplanmıştır. Bu süreç 25. projeye kadar aynı şekilde devam ettirildikten sonra tekrar

MRE hesaplamaları yapılmıştır. MRE hesaplamalarından da Pred(0,25) ve Pred(0,30) değerleri elde edilmiştir.

Uygulanan "Adımsal (Stepwise) Doğrusal Regresyon" sonucunda çıkan denklem ilk projede yerine konularak LOOCV ile kestirilen hata sayısı hesaplanmıştır. İlk proje çıkarıldığında gelen denklem Eş. 3.9'da verilmektedir. İlk projenin çıkarılmasıyla hesaplanan analiz Tablo 3.12. 'de verilmektedir. Tabloya bakıldığında R^2 değerinin yüksek olarak görüldüğü yedinci adım seçilmiştir.

LOOCV ile yapılan hesaplamaların sonucu Tablo 3.13.'de verilmektedir. Verilen tabloda 0,25'den küçük MRE değerleri kutu içine alınarak verilmiştir.

Tablo 3.12. LOOCV ile İlk Projenin Çıkarılmasıyla Elde Edilen Adımsal Doğrusal Regresyonun Sonucu

Adım	1	2	3	4	5	6	7
Sabit	0,2952	-12,518	-36,149	-38,608	-20,731	-42,101	-64,28
LCOM	1,2	1,24	1,21	1,65	1,71	1,81	2,11
NOC		38,2	42,6	40,8	61,1	70,5	83
CBO			3,39	3,76	3,38	3,19	2,97
WMC				-0,99	-1,05	-1,21	-1,06
DIT					-14	-16,4	-17
FANIN						16,9	34
NPRM							-2,6
S	15,3	12,8	9,85	8,83	8,11	7,71	7,14
R-Sq	37,68	58,73	76,62	82,15	85,73	87,81	90,16
R-Sq(adj)	34,84	54,8	73,11	78,39	81,77	83,51	85,86

$$\text{LHS} = (\text{LCOM} * 2{,}11) + (\text{NOC} * 83) + (\text{CBO} * 2{,}97) + (\text{WMC} * -1{,}06) + (\text{DIT} * -17) + (\text{FANIN} * 34) + (\text{NPRM} * -2{,}6) \quad (3.9)$$

LHS : LOOCV ile hata sayısını temsil etmektedir.

Tablo 3.13. LOCCV ile 25 Proje için Gerçek Hata Sayısı, Kestirilen Hata Sayısı ve MRE

Proje No	Hata Sayısı	LOOCV ile Kestirilmiş Hata Sayısı	LOOCV ile MRE	Proje No	Hata Sayısı	LOOCV ile Kestirilmiş Hata Sayısı	LOOCV ile MRE
1	61	69,69	0,142	11	28	24,53	0,123
2	41	37,67	0,081	12	49	38,56	0,213
3	58	33,82	0,416	13	12	13,70	0,141
4	56	53,49	0,044	14	18	34,63	0,924
5	59	56,79	0,037	15	18	13,57	0,245
6	38	19,01	0,499	16	24	29,57	0,232
7	51	20,75	0,593	17	16	13,81	0,137
8	40	41,17	0,029	18	37	42,73	0,154
9	96	83,78	0,127	19	46	34,65	0,246
10	42	46,06	0,096	20	20	32,32	0,615
LOOCV ile Kestirimin Doğruluğu Pred(0,25) = 0,72 Pred(0,30) = 0,76				21	16	12,30	0,230
				22	38	45,22	0,190
				23	25	32,48	0,299
				24	30	37,46	0,248
				25	27	43,83	0,623

Pred(0,25) hesaplanırken MRE değeri 0,25'den küçük olan değerlerin sayısının toplam proje sayısına bölünmesiyle 0,72 değeri, Pred(0,30) hesaplanırken ise MRE değeri 0,30'dan küçük olan değerlerin sayısının yine toplam proje sayısına bölünmesi sonucunda 0,76 değeri elde edilmiştir.

3.8. Tartışma

İlgili çalışmalara bakıldığında, Gyimothy et al., [9] çalışmalarında CK ölçütlerini kullanmışlardır ve hata kestiriminde LCOM ve CBO ölçütleri daha iyi sonuç vermiştir. NOC ölçütünün güvenilmezliğinden bahsederek bu ölçütün hata kestiriminde kullanılmaması gerektiğini savunmuşlardır. Zimmerman et al., [11] tarafından yapılan çalışmaya bakıldığında Mccabe karmaşıklı V(G) ölçütünün yüksek ilişki gösterdiği belirtilmiştir. Ve fazla karmaşık yapıdaki kodlamaların daha çok hataya sahip olduğunun, hataları kestirebilmek için karmaşıklık ölçütlerini kullanmanın yararından bahsetmişlerdir.

Gyimothy et al., [9] yaptıkları çalışmada NOC ölçütünün hata kestirimi için güvenilmezliğinden bahsederken, yapılan bu çalışmada ise NOC ölçütünün de hata

kestiriminde rolü olduğu çalışılan 25 proje ile görülmüştür. Gyimothy et al.çalışmalarında farklı bulgular gözlemlemesinin sebepleri arasında; seçilen programlama dilinin farklılığı kullanılan hata takip sisteminin farklılığından kaynaklı olduğu düşünülmektedir.

Couto et al., [66] tarafından yapılan çalışmada, kullandıkları ölçütler arasından NPM ve NPRM ölçütünün yani public (açık) metotların sayısının ve private metotların sayısının hataları kestiriminde yararlı bir ölçüt olduğunu söylemişlerdir. Yapılan bu çalışmada ise 25 proje için NPM ölçütünün hata kestiriminde öneminin olmadığı görülmüştür. Farklı bulgular gözlemlenme sebebinin ise seçilen veri setinin farklılığı ve hata takip sisteminden kaynaklı olduğu söylenebilir.

4. SONUÇ VE ÖNERİLER

Yapılan çalışmada yazılım projelerindeki hata sayıları ile yazılım kalitesi arasında olan ilişkiyi çıkarabilmek amacıyla bir hata kestirim önerisi sunulmaktadır. Bu amaçla 25 tane orta ölçekli oyun projesi incelenmiştir ve bu projeler açık kaynak kodlu projeler olup, nesne yönelimlidir. İncelenen projelerdeki yazılım ölçütleri analizinde "Understand" statik kod analiz aracı, yazılım hatalarının bulunmasında ise "SpotBugs" kod analiz aracı kullanılmıştır. 25 proje için birisi-dışarıda çapraz doğrulama (LOOCV) yöntemi kullanılarak bağıl hata (MRE) değerleri hesaplanmıştır. Bu hesaplamadan sonra Pred(0,25) ve Pred(0,30) için elde edilen değerler sırasıyla 0,72 ve 0,76 şeklinde çıkmıştır. LOOCV yöntemi uygulanırken önce ilk proje çıkarılmıştır ve kalan 24 projeye " Adımsal (Stepwise) Doğrusal Regresyon" analizi uygulanmıştır, bu süreç 25. proje gelene kadar ve 25. projede dahil tekrarlanmıştır. Analiz sonuçlarında çıkan denklemler sırayla yapılan hesaplamalarda yerlerine konulmuştur.

Elde edilen pred değerlerine bakıldığında Pred (0,25) değerinin 0,72 olarak hesaplandığı ve literatürdeki Pred(0,25) değerine çok yakın olduğu, Pred(0,30) değerinin ise 0,76 olarak hesaplandığı ve bu değerin ise literatürde geçen değeri desteklediği görülmektedir. Yapılan çalışmanın, literatürdeki kestirim doğruluğunun kabul edilir seviyede olduğu, Pred(0,30) değerinin eşik değerini sağlamasından anlaşılmaktadır.

Çalışmada veri seti olarak açık kaynak kodlu projelerin tercih edilip farklı ölçütler üzerinden analizinin sağlanmasının sebebi, literatürde geçen çalışmalar incelendiğinde kullanılan hazır veri setlerinde hata kestirimiyle ilişkisi olan her ölçüte yer verilmemesinden kaynaklı olması söylenebilir.

Çalışma sonucunda elde edilen bulgulara genel olarak bakıldığı zaman; CK ölçütleri arasından RFC ölçütü dışında diğer ölçütlerin hata kestirimiyle ilişkisi olduğu, kullanılan diğer dört ölçüt arasından ise FANIN ve NPRM ölçütlerinin hata kestiriminde etkili olduğu gözlemlenmiştir. Hata kestiriminde en güçlü değişkenlerin LCOM, NOC, CBO, WMC, DIT, FANIN ve NPRM ölçütleri olduğu ve bu ölçütlerin hata kestirimi üzerinde öneminin olduğu gözlemlenmiştir.

Bu çalışmada kullanılan proje sayısı 25 ile sınırlı olup proje sayısı arttırılarak gelecekte yapılabilecek çeşitli projeler veya çalışmalar için kestirim önerisinin doğruluğunu arttırmak hedeflenmektedir.

KAYNAKLAR

[1] G. Tassey, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology, USA, Gaithersburg, Technical Report, September 15, 2019. [Online]. Available: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf

[2] C. Ghezzi, M. Jazayeri, D. Mandrioli, *Fundamentals of Software Engineering,* NJ: Prentice-Hall 1991.

[3] B. W. Boehm, *Software Engineering Economics,* NJ: Prentice-Hall, 1981.

[4] G. J. Myers, *Art of Software Testing*, New York: Wiley, 1979.

[5] Y. Singh, *Software Testing*, UK: Cambridge University Press, 2012.

[6] A. Endres, "An analysis of errors and their causes in system programs," *ACM SIGPLAN Notices International Conference on Reliable Software,* vol.10, no.6, pp. 327-336, 1975, doi: 10.1145/800027.808455.

[7] F. Lanubile, A. Lonigro and G. Vissagio, "Comparing models for identifying fault-prone software components," in *Proc. 7th International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, USA, 1995, pp. 312-319.

[8] G. Denaro, " Estimating software fault-proneness for tuning testing activities," in *Proc. of Twenty-second International Conference on Software Engineering*, New York, NY, USA, 2000, pp. 704-706.

[9] T. Gyimothy, R. Ferenc and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering,* vol.31, no.10, pp. 897-910, 2005, doi: 10.1109/TSE.2005.112.

[10] H. M. Olague, L. H. Etzkorn, S. Gholston and S. Quattlebatum, "Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed

using highly iterative or agile software development processes," *IEEE Transactions on Software Engineering,* vol.33, no.6, pp. 402-419, 2007, doi: 10.1109/TSE.2007.1015.

[11] T. Zimmermann, R. Premraj and A. Zeller, "Predicting defects for eclipse," in *Proc. 3th Third International Workshop on Predictor Models in Software Engineering,* Washington, DC, USA, 2007, pp. 9.

[12] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.

[13] J. C. Laprie and K. Kanoun, *Reliability and System Reliability,* Handbook of Software Reliability Engineering, Hightstown, NJ, USA, pp. 27-69, 1996.

[14] K. Punitha and S. Chitra, "Software defect prediction using software metrics a survey," in *Proc. of 2013 International Conference on Information Communication and Embedded Systems (ICICES),* Chennai, India, 2013, pp. 55-558.

[15] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan and N. Goel, "Early quality prediction: a case study in telecommunications," *IEEE Software,* vol.13, no.1, pp. 65-71, 1996, doi: 10.1109/52.476287.

[16] T. M. Khosgoftaar and J. C. Munson, "Predicting software development errors using software complexity metrics," *Journal On Selected Areas In Communications,* vol.8, no. 2, pp. 253-261, 1990, doi: 10.1109/49.46879.

[17] X. Yuan, T. M. Khoshgoftaar, E. B. Allen and K. Ganesan, "An application of fuzzy clustering to software quality prediction," in *Proc. 3th IEEE Symposium on Application-Specific Systems and Software Engineering Technology,* March 24-25, 2000.

[18] K. EL EMAM, *The ROI from software quality,*1th Edition, Auerbach Publications, 2005.

[19] C. Jones, *Software engineering best practices: lessons from successful projects in the top companies,* 1st Edition, The McGraw-Hill Companies, 2010.

[20] J. Sanders and E. Curran, *Software Quality,* Addison Wesley Longman, 1994.

[21] M. Evett, T. Khoshgoftar, P. Chien and E. Allen, "GP-based software quality prediction," in *Proc. 3th Annual Conference on Genetic Programming*, 1998.

[22] B. W. Boehm, J. R. Brown, H. Kaspar, M. G. Lipow, G. Mcleod and M. Merritt, *Characteristics of Software Quality,* North-Holland, Amsterdam, 1978.

[23] J. D. Musa, *Software reliability engineering*, 2nd Edition, Author House, Bloomington, 2004.

[24] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target," *IEEE Software,* vol.13, no.1, pp. 12-21, 1996, doi: doi: 10.1109/52.476281.

[25] B. Singh and S. P. Kannojia, "A model for software product quality prediction," *Journal of Software Engineering and Applications,* vol.5, no.6, pp. 395-401, 2012, doi: 10.4236/jsea.2012.56046.

[26] *ISO/IEC 9126-1: Information Technology - Software Product Quality - Part 1: Quality Model,* ISO/IEC JTC1/SC7/WG6, 1999.

[27] J. Capers, "Software Defect Origins and Removal Methods" https://www.ifpug.org/content/documents/JonesSoftwareDefectOriginsAndRemovalMethodsDraft5.pdf (Accessed: September 2019).

[28] S. L. Pfleeger, *Software Engineering Theory & Practice,* 1st Edition, Prentice Hall, Inc, 2001.

[29] M. D'ambros, M. Lanza and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Software Engineering,* vol. 17, no.4, pp. 531-577, 2012, doi: 10.1007/s10664-011-9173-9.

[30] R. Moser, W. Pedrycz and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, " in *Proc. 30th International*

Conference on Software Engineering, New York, NY, USA, 10-18 May., 2008, pp. 181-190.

[31] S. Kim, T. Zimmermann, E. J. Whitehead and A. Zeller, "Predicting faults from cached history," in *Proc. 29th International Conference on Software Engineering,* Minneapolis, MN, USA, May. 20-26, 2007.

[32] V. R. Basili, L. C. Briand and W.L. Melo, "A validation of object oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering,* vol. 22, no.11, pp. 751-761, 1996, doi: 10.1109/32.544352.

[33] A. E. Hassan, "Predicting Faults Using the Complexity of Code Changes," in *Proc. 31st International Conference on Software Engineering,* Vancouver, BC, Canada, May. 2009.

[34] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 35th International Conference on Software Engineering (ICSE),* San Francisco, CA, USA, May. 20-22, 2013.

[35] G. Singh, D. Singh and V. Singh, "A study of software metrics," *International Journal of Computational Engineering & Management,* vol.11, 2011.

[36] A.S. Nunez-Varela, H. Perez-Gonzalez, F.E. Martinez and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study, " *Journal of Systems and Software,* vol.128, pp. 164-197, 2017, doi: 10.1016/j.jss.2017.03.044.

[37] T. J. Mccabe, "A complexity measure," *IEEE Transactions on Software Engineering,* vol.SE-2, no.4, pp. 308-320, 1976, doi: 10.1109/TSE.1976.233837.

[38] W. Li and S. Henry, "Object-Oriented metrics that predict maintainability," *Journal of Systems and Software - Special issue on object-oriented software,* vol.23, no.2, pp. 111-122, 1993, doi: 10.1016/0164-1212(93)90077-B.

[39] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering,*vol.20, no.6, pp. 476-493, 1994, doi: 10.1109/32.295895.

[40] N. E. Fenton and M. Neil, "Software metrics: successes, failures and new directions," *Journal of Systems and Software - Special issue on invited articles on top systems and software engineering scholars,* vol. 47, no. 2-3, pp. 149-157, 1999, doi: 10.1016/S0164-1212(99)00035-7.

[41] L. Zhao and J. H. Hayes, "Predicting classes in need of refactoring: An application of static metrics," in *Proc. 2nd International PROMISE Workshop*, Philadelphia, Pennsylvania USA, 2006.

[42] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Transactions on Software Engineering,* vol.29, no.4, pp. 297-310, 2003, doi: 10.1109/TSE.2003.1191795.

[43] R. S. Pressman, *Software engineering: A practitioner's approach,* 5th Edition, New York: McGraw-Hill, 2000.

[44] M. P. Thapaliyal and G. Verma, "Software defects and object oriented metrics – An empirical analysis," *International Journal of Computer Applications,* vol.9 , no.5, 2010, doi: 10.5120/1379-1859.

[45] K. M. Breesam, "Metrics for object oriented design focusing on class inheritance metrics," in *Proc. 2nd International Conference on Dependability of Computer Systems,* Szklarska, Poland, June 14-16, 2007.

[46] L. C. Briand, J. Wust, S. V. İkonomovski and H. Lounis, "Investigating quality factors in object-oriented designs: An industrial case study," in *Proc. of the 1999 International Conference on Software Engineering,* Los Angeles, CA, USA, May. 22, 1999.

[47] Y. Zhou and H. Leung, "Analysis of object-oriented design metrics for predicting high and low severity faults," *IEEE Transactions on Software Engineering,* vol.32, no.10, pp. 771-789, 2006, doi: 10.1109/TSE.2006.102.

[48] P. Yu, T. Systa and H. Müller, "Predicting fault-proneness using OO metrics. An industrial case study," in *Proc. 6th European Conference on Software Maintenance and Reengineering,* Budapest, Hungary, March 13, 2002.

[49] M. Tang, M. Kao and M. Chen, "An empirical study on object-oriented metrics," in *Proc. 6th International Software Metrics Symposium*, Boca Raton, FL, USA, Nov. 4-6, 1999.

[50] H. F. Li and W. K. William, "An empirical study of software metrics," *IEEE Transactions on Software Engineering*, vol.13, no.6, pp. 697-708, 1987, doi: 10.1109/TSE.1987.233475.

[51] C. Jones, *A short history of the lines of code (LoC) metric,* Providence: Capers Jones & Associates LLC, 2012.

[52] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler and J. Penix, "Using static analysis to find bugs," vol. 25, no.5, pp. 22-29, 2008, doi: 10.1109/MS.2008.130.

[53] "SpotBugs" https://spotbugs.github.io (Accessed: April,2019).

[54] "Metrics" https://sourceforge.net/projects/metrics/ (Accessed: May,2019).

[55] "PMD" https://pmd.github.io/ (Accessed: April,2019).

[56] "Understand" https://scitools.com/feature/metrics/ (Accessed: February, 2019).

[57] "Checkstyle" https://checkstyle.sourceforge.io/ (Accessed: April, 2019).

[58] "Coverlipse" https://sourceforge.net/projects/coverlipse/ (Accessed: April, 2019).

[59] S. Herbold, A. Trautsch and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Transactions on Software Engineering*, vol.44, no.9, pp.811-833, 2017, doi: 10.1109/TSE.2017.2724538.

[60] G. Concas, M. Marchesi, A. Murgia and R. Tonelli, "An empirical study of social networks metrics in object-oriented software," *Advances in Software Engineering - Special issue on new generation of software metrics*, vol.2010, no.4, 2010 doi: 10.1155/2010/729826.

[61] C. Ficarra and F. Vicente, *Advanced Research and Trends in New Technologies, Software, Human-computer Interaction, and Communicability,* IGI Global, 2013.

[62] U. Erdemir, U. Tekin ve F. Buzluca, "Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi," Ekim 2008. [Online]. Available : http://softwaresuccess.org

[63] C. Couto, C. Silva, M. T. Valente, R. Bigonha and N. Anquetil, "Uncovering causal relationships between software metrics and bugs," in *Proc. 16th European Conference on Software Maintenance and Reengineering,* Szeged, Hungary, March 27-30, 2012.

[64] K. Yamashita, C. Huang, M. Nagappan, Y. Kamei, A. Mockus, A. E. Hassan and N. Ubayashi, "Thresholds for size and complexity metrics: A case study from the perspective of defect density," in *Proc. of the 2016 IEEE International Conference on Software Quality, Reliability and Security,* Vienna, Austria, Aug. 1-3, 2016.

[65] "SpotBugs" https://spotbugs.readthedocs.io/en/stable/index.html (Accessed: October, 2019).

[66] J. Farrell, *Java Programming, Course Technology,* 6th Edition, , Boston, 2012.

[67] Y. Perez-Riverol, L. Gatto, T. Sachsenberg, J. Uszkoreit, F. V. Leprevost, ... and J. A. Vizcaino, "Ten simple rules for taking advantage of git and github," *PLOS Computational Biology,* vol. 12, 2016, doi: 10.1371/journal.pcbi.1004947.

[68] "Sonarqube" https://www.sonarqube.org/ (Accessed: September, 2019).

[69] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers,* 6nd Edition, John Wiley & Sons, Inc, 2010.

[70] R. D. Baker, "A Hybrid Approach to Expert and Model Based Effort Estimation", M.S. thesis, Dept. College of Engineering and Mineral Resources, West Virginia University, Morgantown, 2007.

[71] S. D. Conte, H. E. Dunsmore and V. Y. Shen, *Software Engineering Metrics and Models,* The Benjamin/Cummings Publishing Company, Inc, 2003

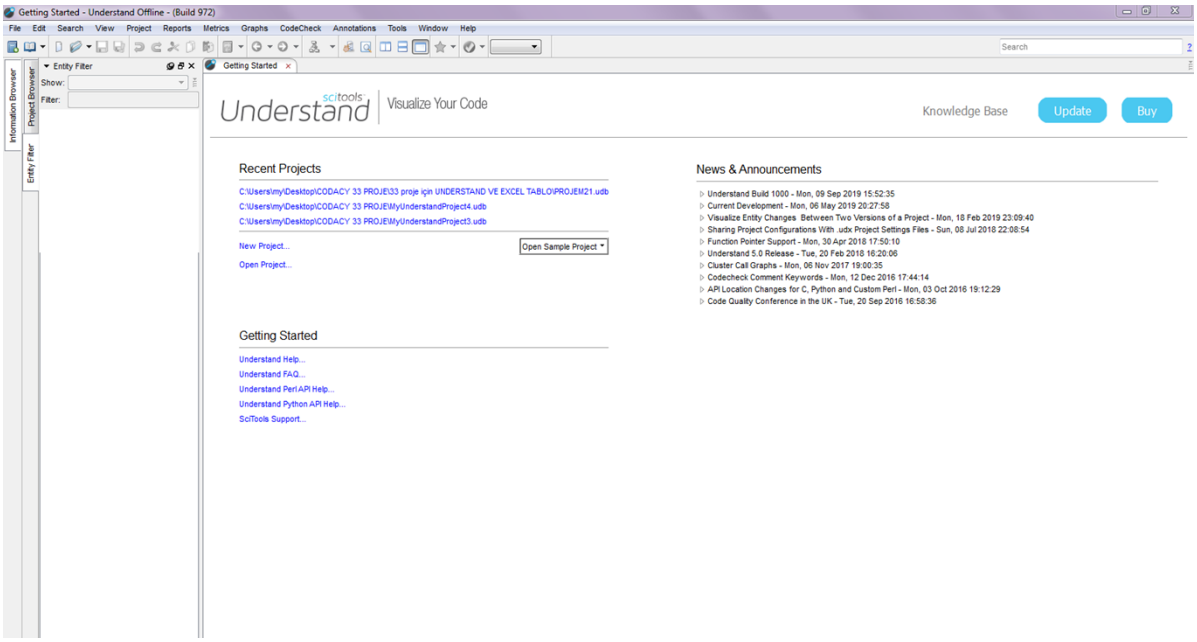
[72] G. Tate and J. Verner, *The Economics of Information Systems and Software*, Software Costing in Practice, 1991.

[73] T. Wong, "Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation," *Pattern Recognition*, vol.48, no.9, pp. 2839-2846, 2015, doi: 10.1016/j.patcog.2015.03.009.

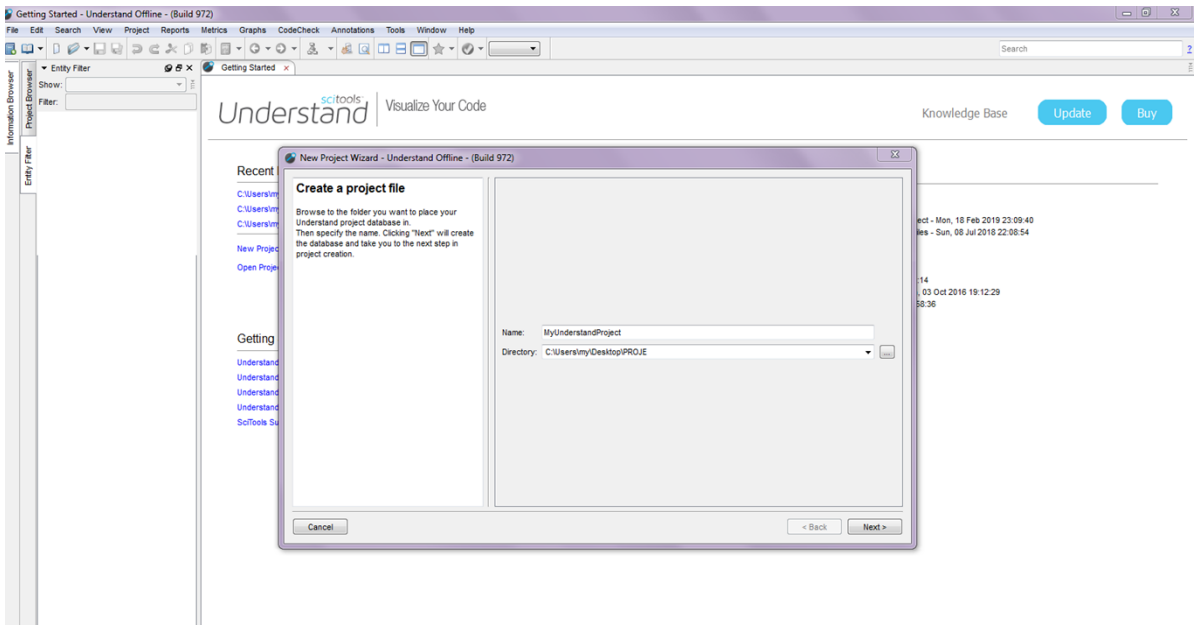
[74] "K fold and other cross-validation techniques", https://medium.com/datadriveninvestor/k-fold-and-other-cross-validation-techniques-6c03a2563f1e (Accessed: November, 2019).

EKLER

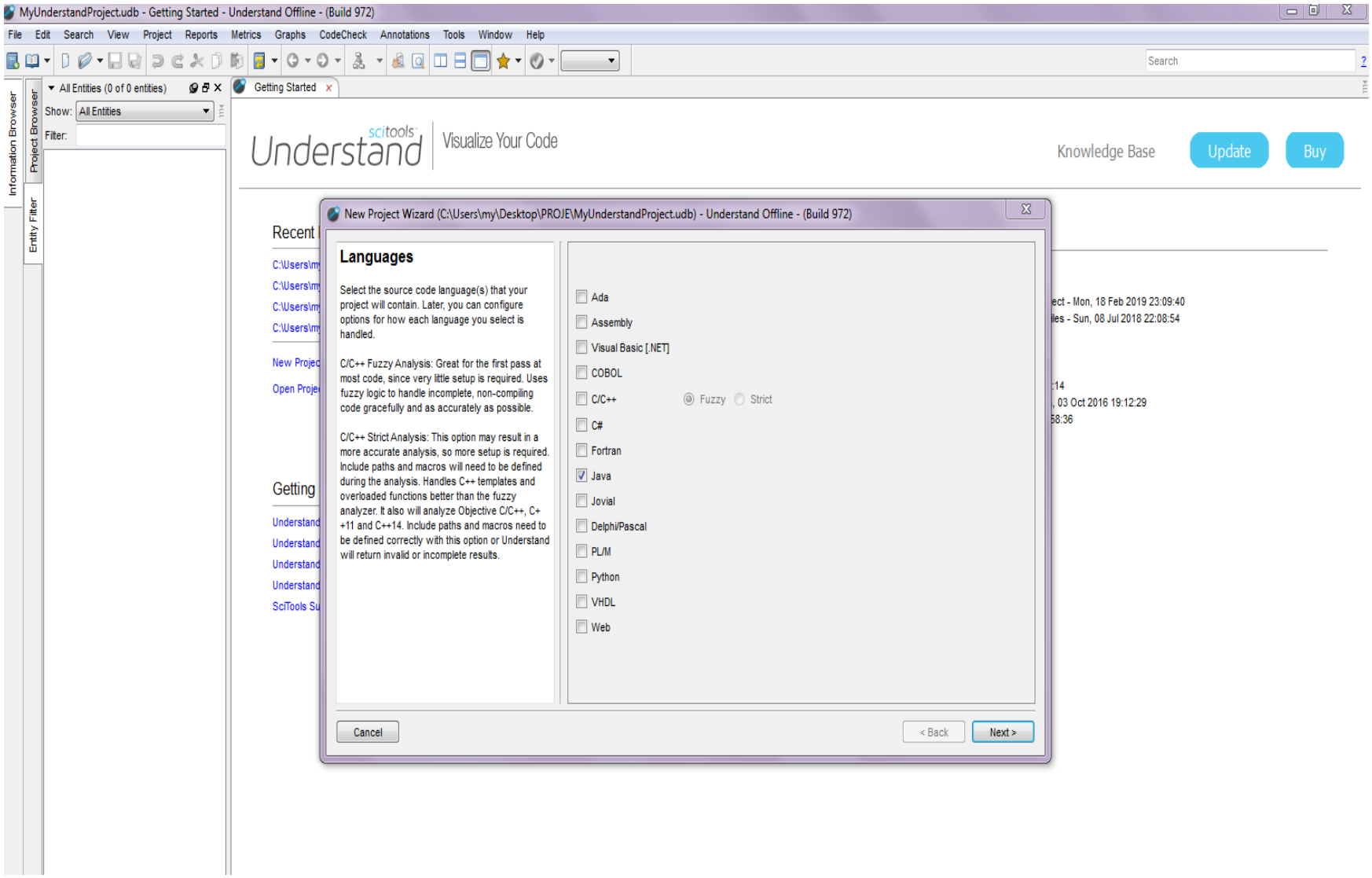
EK 1 : UNDERSTAND ARACI EKRAN GÖRÜNTÜLERİ



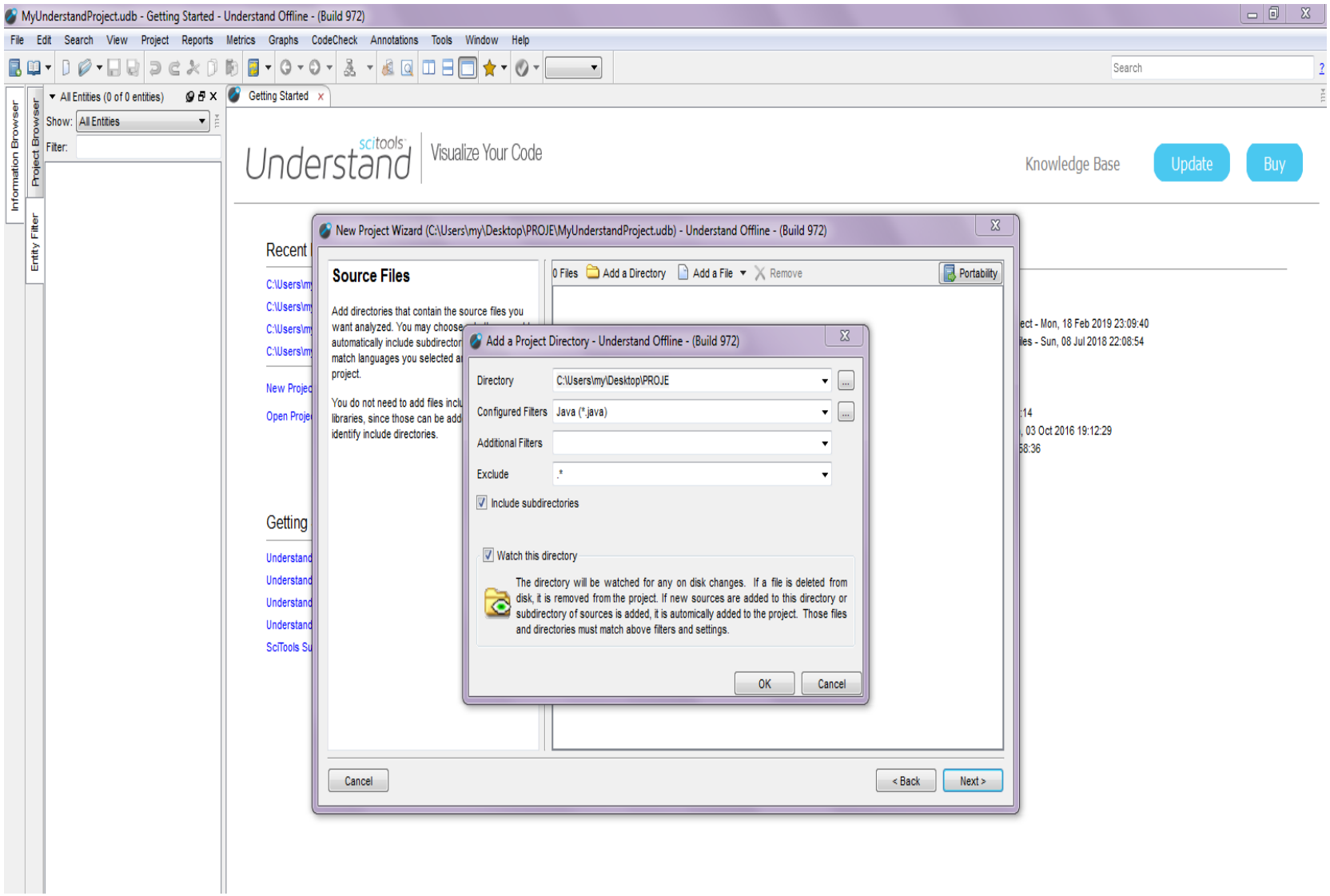
Görsel 1-1 Understand Aracı İlk Ekranı



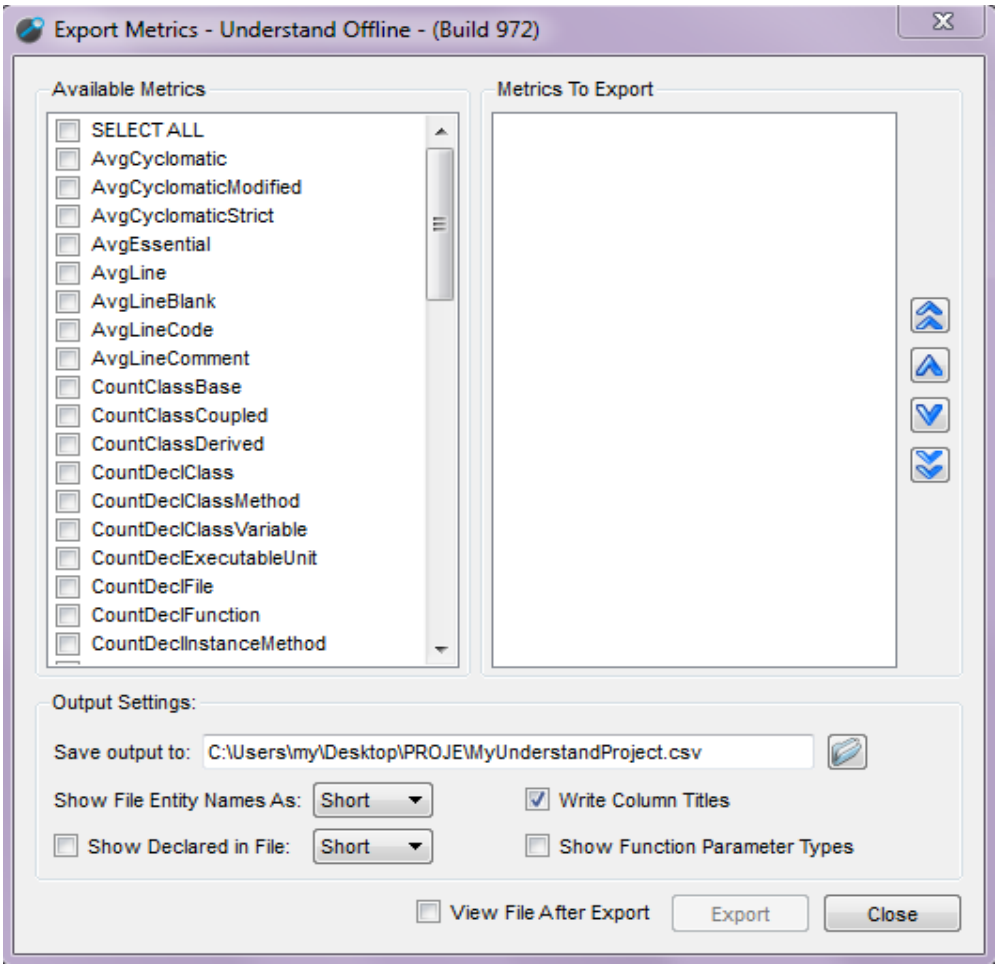
Görsel 1-2 Understand Proje Oluşturma Ekranı



Görsel 1-3 Understand ile Dil Seçimi



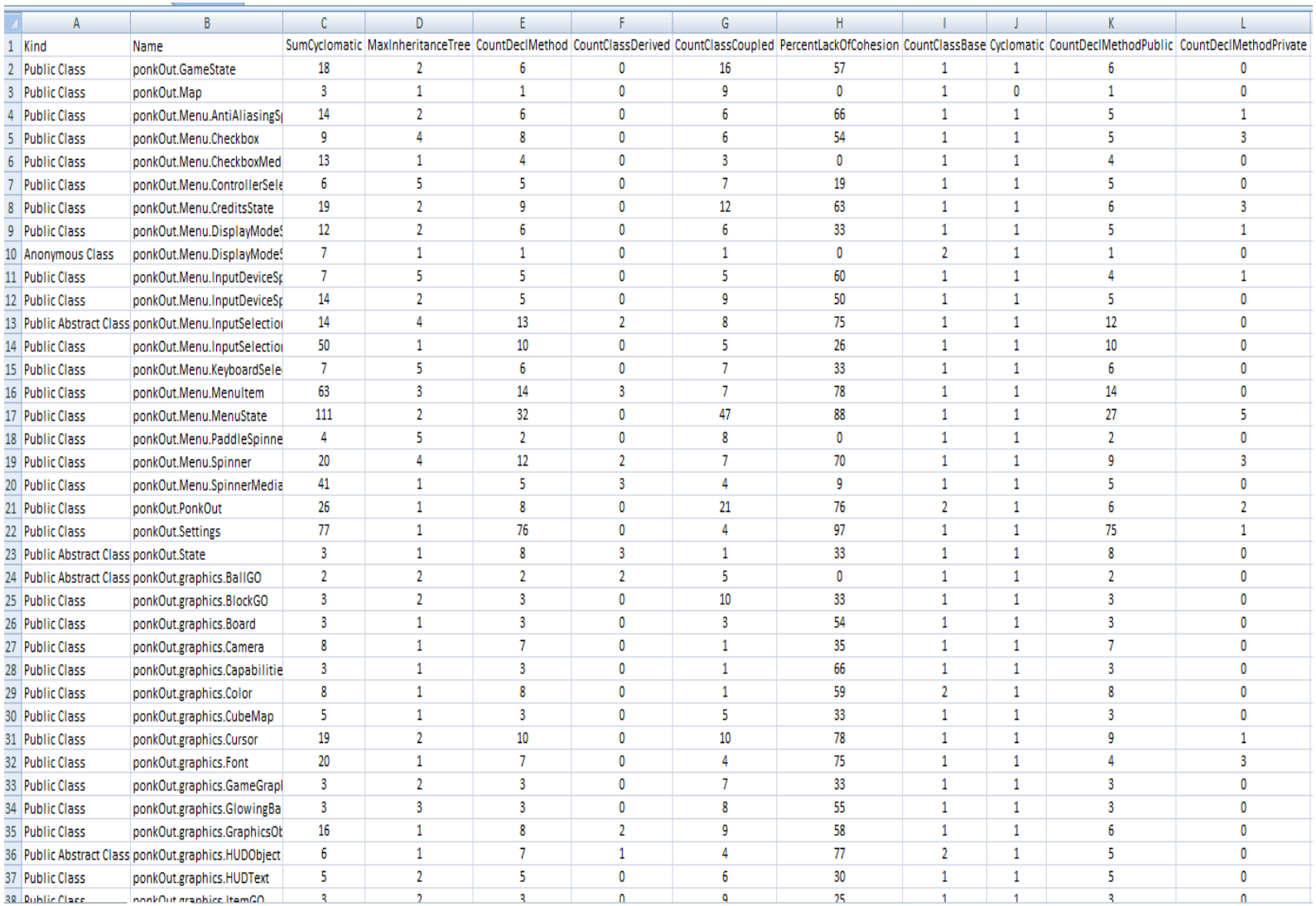
Görsel 1- 4 Understand ile Projenin Adresinin Seçilmesi Ekranı



Görsel 1-5 Understand Proje Analizinde Kullanılacak Ölçütlerin Seçilmesi Ekranı

Classes:	584
Files:	357
Program Units:	2777
Lines:	51531
Lines Blank:	7310
Lines Code:	42390
Lines Comment:	2104
Lines Inactive:	0
Executable Statements:	8675
Declarative Statements:	8993
Ratio Comment/Code:	0.05

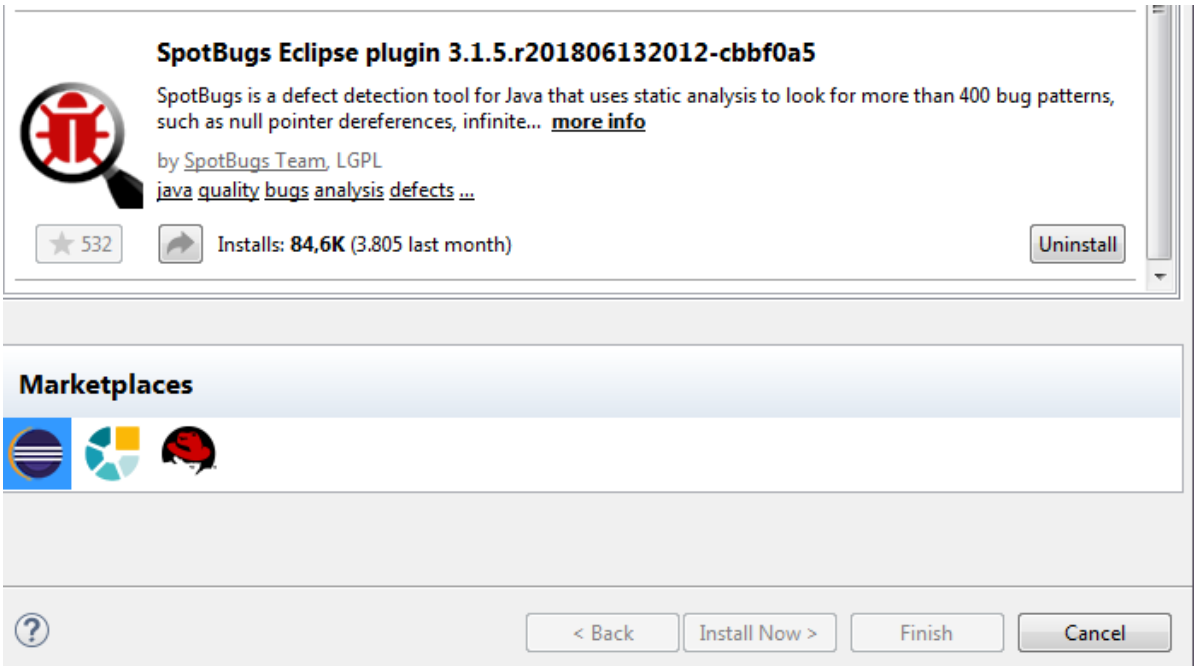
Görsel 1-6 Understand Seçilen Bir Proje Örneğinin Yapısal Özellikleri



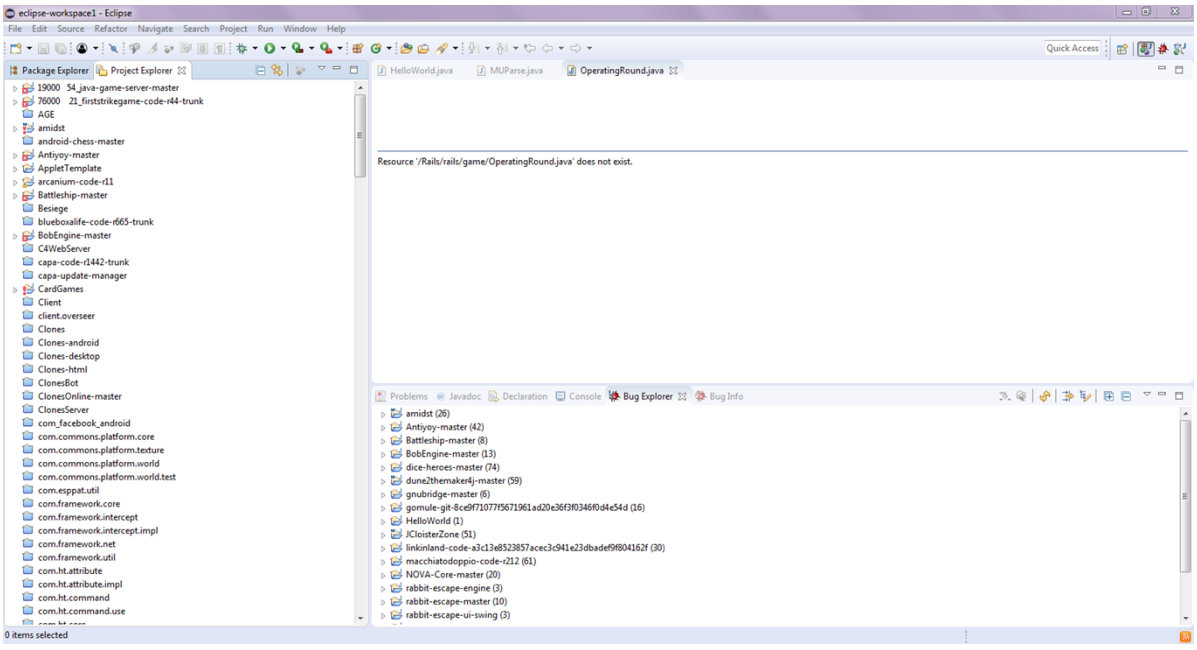
	A	B	C	D	E	F	G	H	I	J	K	L
1	Kind	Name	SumCyclomatic	MaxInheritanceTree	CountDeclMethod	CountClassDerived	CountClassCoupled	PercentLackOfCohesion	CountClassBase	Cyclomatic	CountDeclMethodPublic	CountDeclMethodPrivate
2	Public Class	ponkOut.GameState	18	2	6	0	16	57	1	1	6	0
3	Public Class	ponkOut.Map	3	1	1	0	9	0	1	0	1	0
4	Public Class	ponkOut.Menu.AntiAliasingS	14	2	6	0	6	66	1	1	5	1
5	Public Class	ponkOut.Menu.Checkbox	9	4	8	0	6	54	1	1	5	3
6	Public Class	ponkOut.Menu.CheckboxMed	13	1	4	0	3	0	1	1	4	0
7	Public Class	ponkOut.Menu.ControllerSele	6	5	5	0	7	19	1	1	5	0
8	Public Class	ponkOut.Menu.CreditsState	19	2	9	0	12	63	1	1	6	3
9	Public Class	ponkOut.Menu.DisplayMode$	12	2	6	0	6	33	1	1	5	1
10	Anonymous Class	ponkOut.Menu.DisplayMode$	7	1	1	0	1	0	2	1	1	0
11	Public Class	ponkOut.Menu.InputDeviceSp	7	5	5	0	5	60	1	1	4	1
12	Public Class	ponkOut.Menu.InputDeviceSp	14	2	5	0	9	50	1	1	5	0
13	Public Abstract Class	ponkOut.Menu.InputSelectio	14	4	13	2	8	75	1	1	12	0
14	Public Class	ponkOut.Menu.InputSelectio	50	1	10	0	5	26	1	1	10	0
15	Public Class	ponkOut.Menu.KeyboardSele	7	5	6	0	7	33	1	1	6	0
16	Public Class	ponkOut.Menu.MenuItem	63	3	14	3	7	78	1	1	14	0
17	Public Class	ponkOut.Menu.MenuState	111	2	32	0	47	88	1	1	27	5
18	Public Class	ponkOut.Menu.PaddleSpinne	4	5	2	0	8	0	1	1	2	0
19	Public Class	ponkOut.Menu.Spinner	20	4	12	2	7	70	1	1	9	3
20	Public Class	ponkOut.Menu.SpinnerMedia	41	1	5	3	4	9	1	1	5	0
21	Public Class	ponkOut.PonkOut	26	1	8	0	21	76	2	1	6	2
22	Public Class	ponkOut.Settings	77	1	76	0	4	97	1	1	75	1
23	Public Abstract Class	ponkOut.State	3	1	8	3	1	33	1	1	8	0
24	Public Abstract Class	ponkOut.graphics.BallGO	2	2	2	2	5	0	1	1	2	0
25	Public Class	ponkOut.graphics.BlockGO	3	2	3	0	10	33	1	1	3	0
26	Public Class	ponkOut.graphics.Board	3	1	3	0	3	54	1	1	3	0
27	Public Class	ponkOut.graphics.Camera	8	1	7	0	1	35	1	1	7	0
28	Public Class	ponkOut.graphics.Capabilitie	3	1	3	0	1	66	1	1	3	0
29	Public Class	ponkOut.graphics.Color	8	1	8	0	1	59	2	1	8	0
30	Public Class	ponkOut.graphics.CubeMap	5	1	3	0	5	33	1	1	3	0
31	Public Class	ponkOut.graphics.Cursor	19	2	10	0	10	78	1	1	9	1
32	Public Class	ponkOut.graphics.Font	20	1	7	0	4	75	1	1	4	3
33	Public Class	ponkOut.graphics.GameGrapl	3	2	3	0	7	33	1	1	3	0
34	Public Class	ponkOut.graphics.GlowingBa	3	3	3	0	8	55	1	1	3	0
35	Public Class	ponkOut.graphics.GraphicsOb	16	1	8	2	9	58	1	1	6	0
36	Public Abstract Class	ponkOut.graphics.HUDObject	6	1	7	1	4	77	2	1	5	0
37	Public Class	ponkOut.graphics.HUDText	5	2	5	0	6	30	1	1	5	0
38	Public Class	ponkOut.graphics.ItemGO	3	2	3	0	9	25	1	1	3	0

Görsel 1-7 Understand Proje Analizinde Kullanılan Ölçütlerin Excel'e Aktarılma Görüntüsü

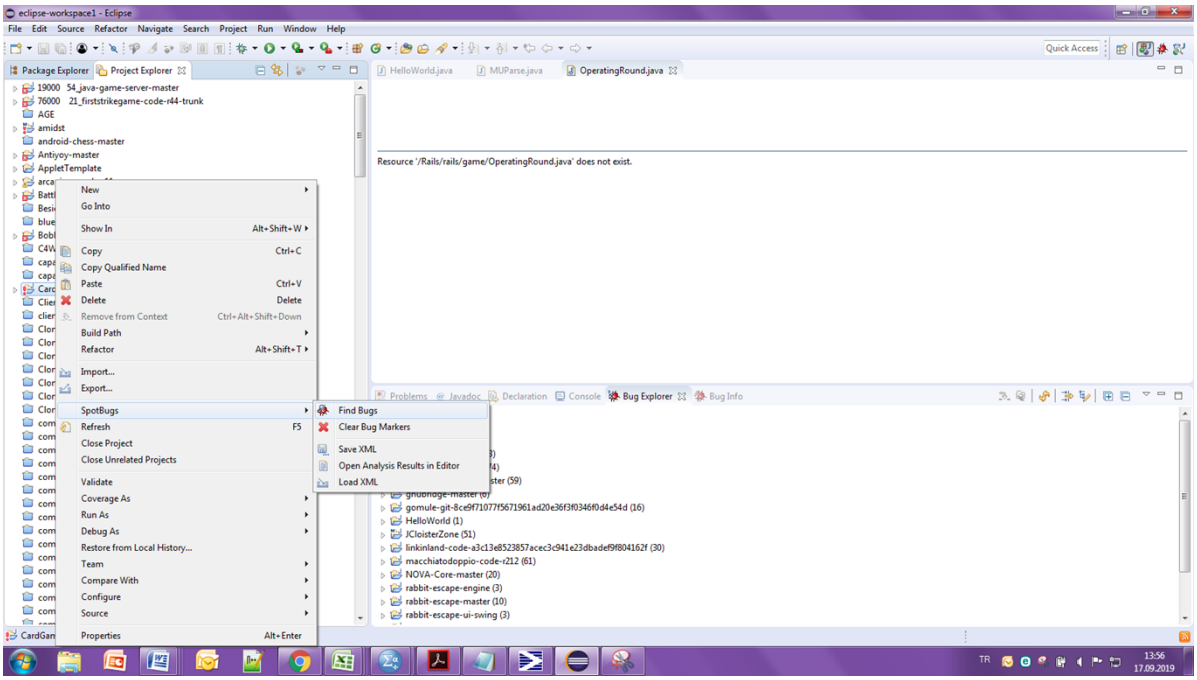
EK 2 : SPOTBUGS ARACI EKRAN GÖRÜNTÜLERİ



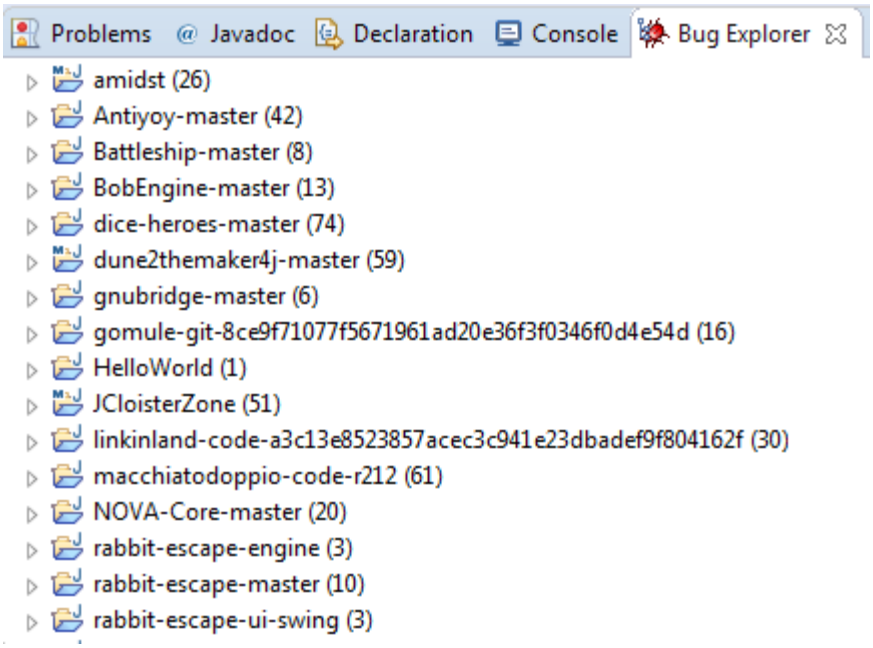
Görsel 2-1 SpotBugs Aracının Eclipse Ortamına Plugin Olarak Kurulması Ekranı



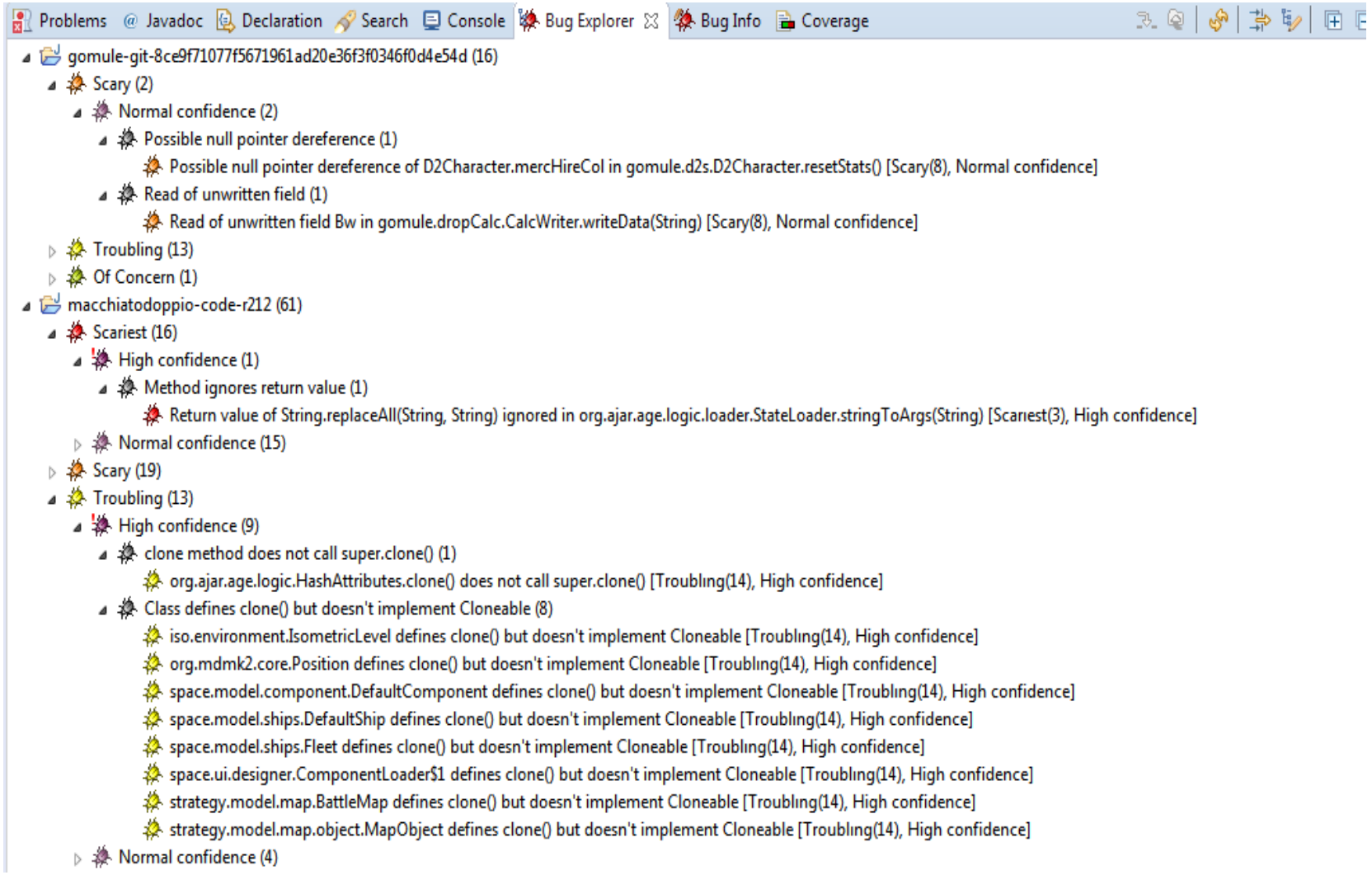
Görsel 2-2 SpotBugs Aracına Projelerin Yüklenmesi



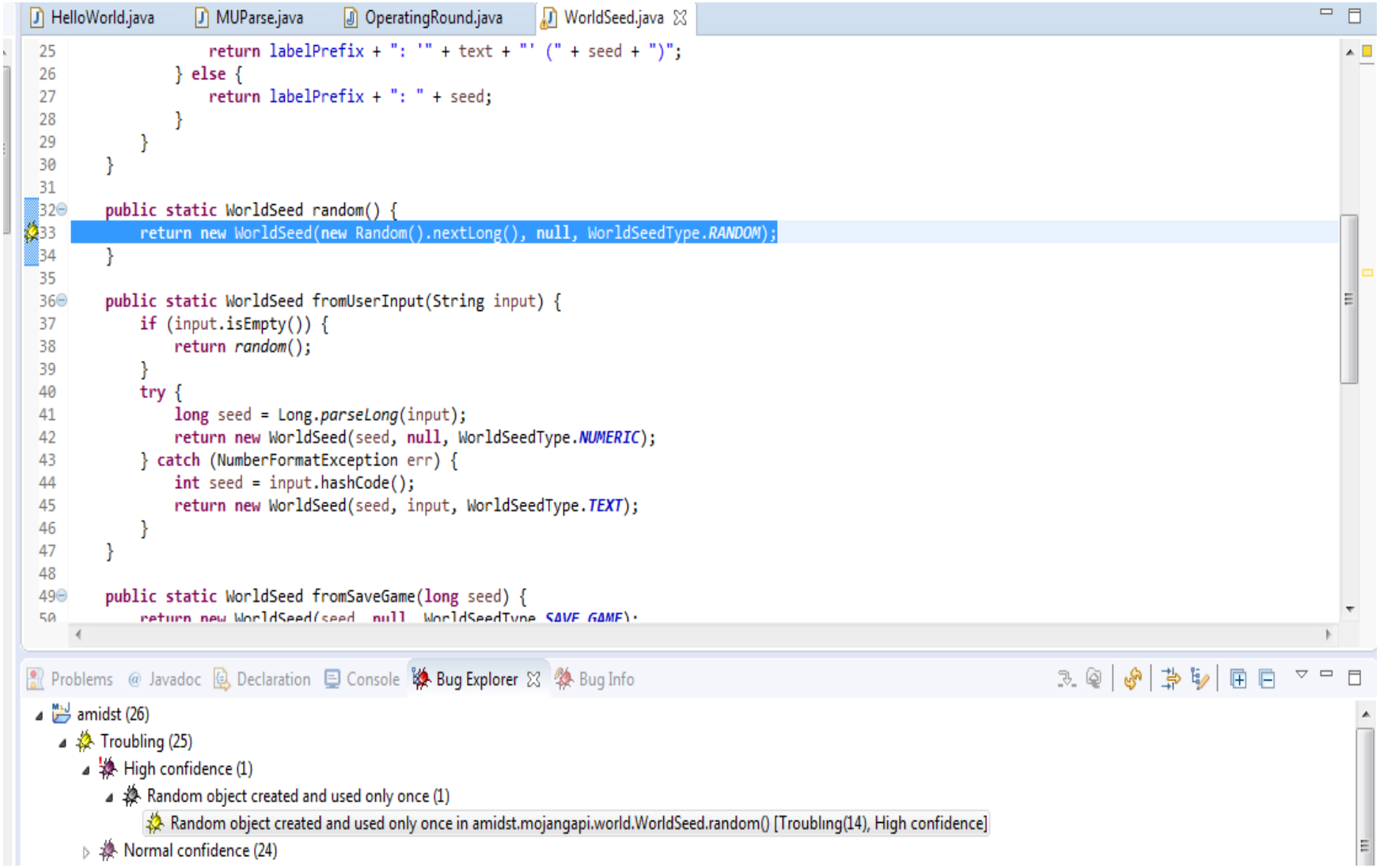
Görsel 2-3 SpotBugs Aracı ile Analizin Başlatılması Ekranı



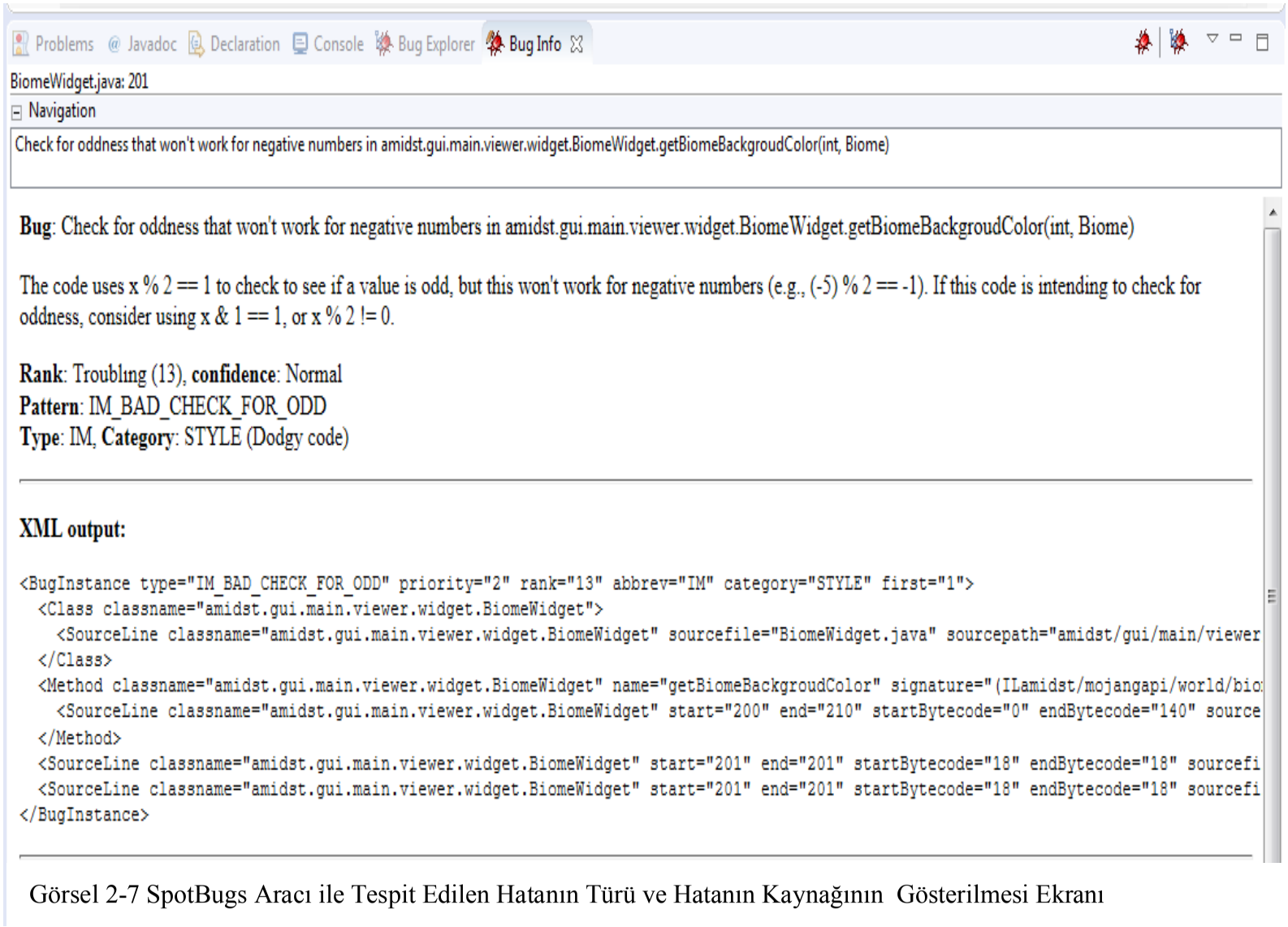
Görsel 2-4 SpotBugs Aracı ile Analizi Yapılan Projelerdeki Hata Sayılarının Tespit Edilmesi Ekranı



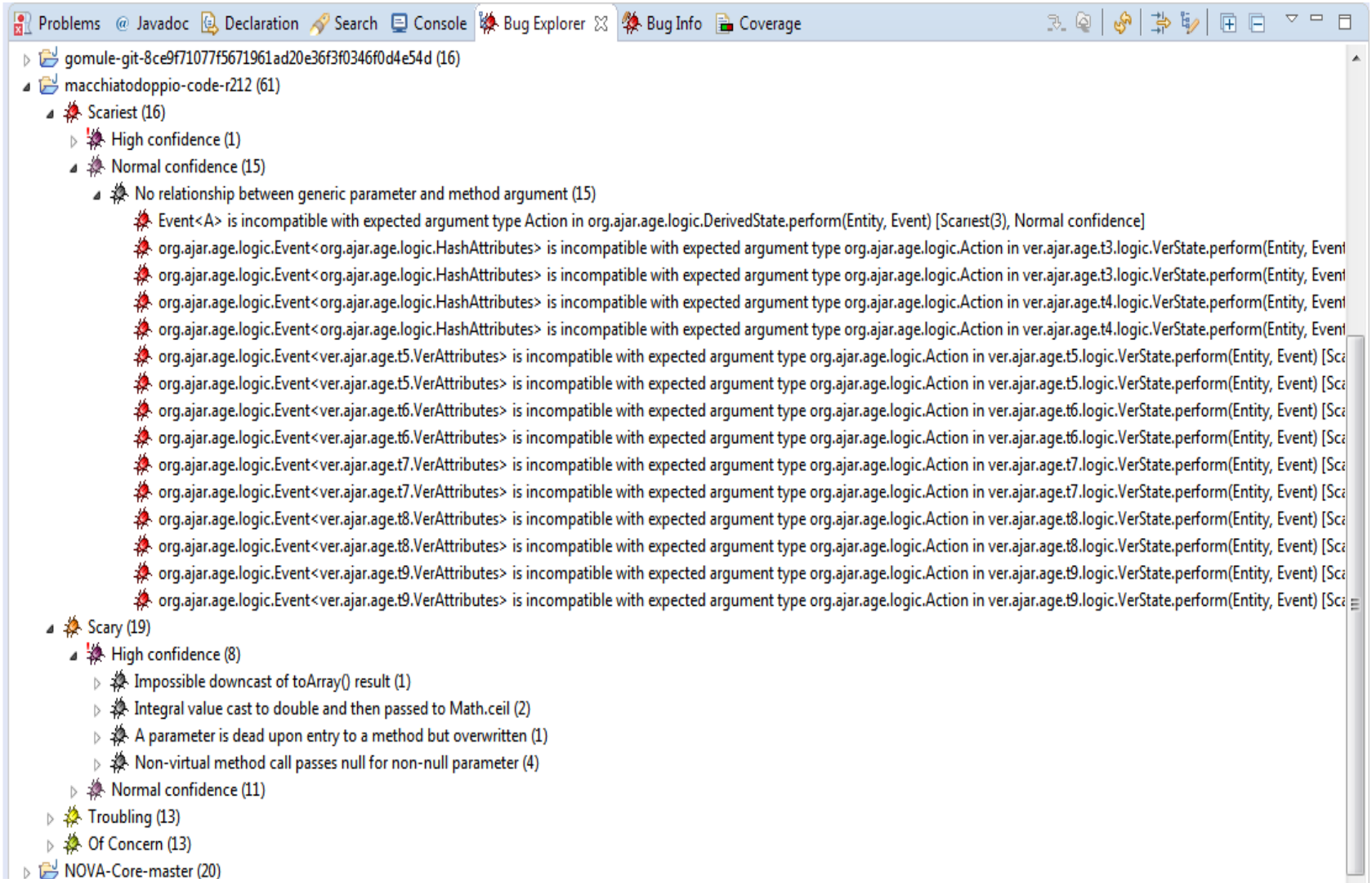
Görsel 2-5 SpotBugs Aracı ile Analizi Yapılan Projelerdeki Hata Sayılarının



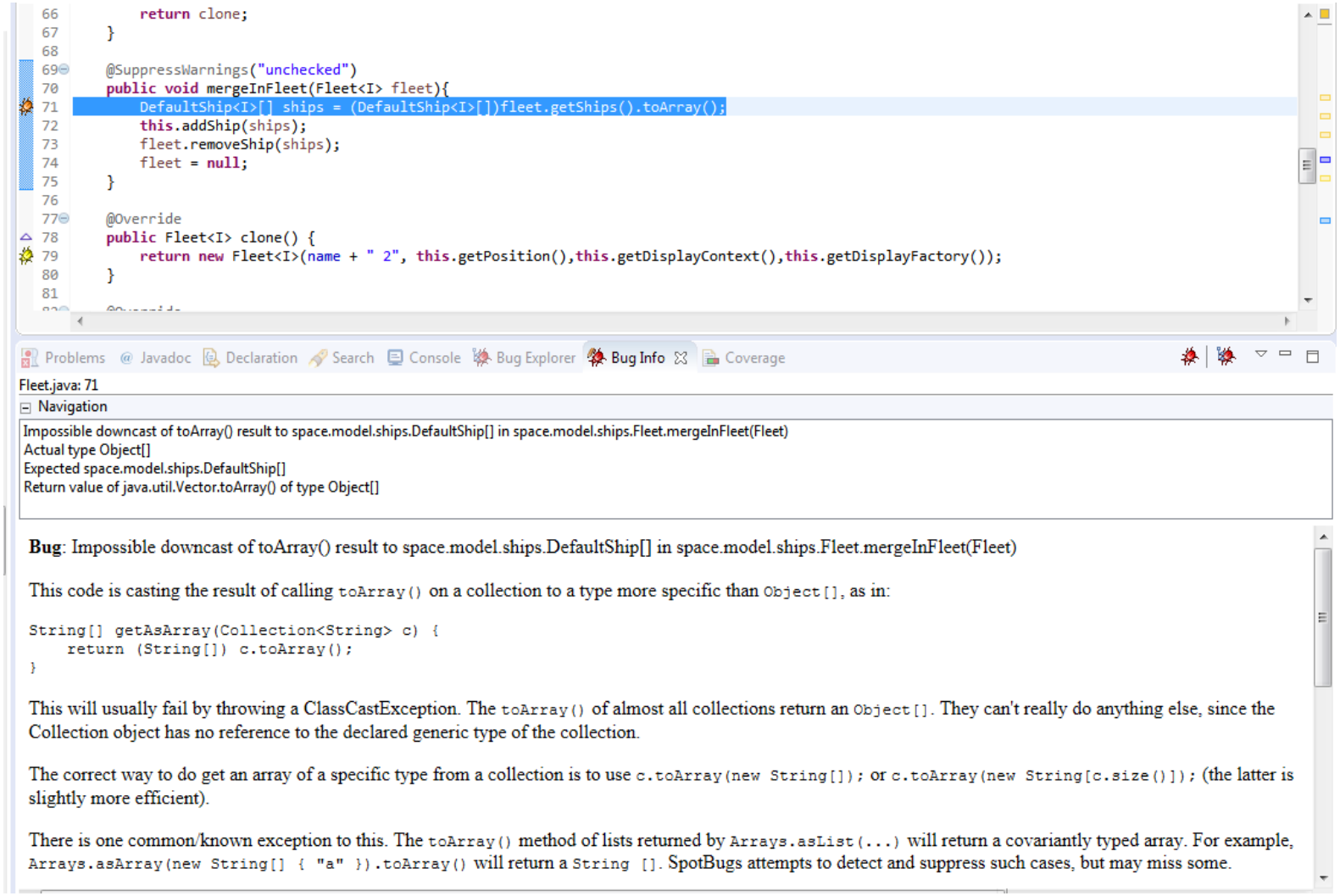
Görsel 2-6 SpotBugs Aracı ile Tespit Edilen Hatanın Kodlamadaki Yerinin Tespiti Ekranı



Görsel 2-7 SpotBugs Aracı ile Tespit Edilen Hatanın Türü ve Hatanın Kaynağının Gösterilmesi Ekranı



Görsel 2-8 SpotBugs Aracı ile Tespit Edilen Hataların Detaylı Sonuçları



Görsel 2-9 SpotBugs Aracı ile Tespit Edilen Örnek Bir Hatanın Sebebinin Detaylandırılması