İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE DAYALI YAZILIMLARDA SINAMA VE BAKIM ÖNCELİKLİ SINIFLARIN BELİRLENMESİ

YÜKSEK LİSANS TEZİ

Sinan ESKİ

Anabilim Dalı : Bilgisayar Mühendisliği

Programı : Bilgisayar Mühendisliği

Tez Danışmanı: Yrd. Doç. Dr. Feza BUZLUCA

Haziran 2011

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE DAYALI YAZILIMLARDA SINAMA VE BAKIM ÖNCELİKLİ SINIFLARIN BELİRLENMESİ

YÜKSEK LİSANS TEZİ
Sinan ESKİ
(504081529)

Tezin Enstitüye Verildiği Tarih : 06 Mayıs 2011
Tezin Savunulduğu Tarih : 01 Haziran 2011

Tez Danışmanı : Yrd. Doç. Dr. Feza BUZLUCA (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Nadia ERDOĞAN (İTÜ)
Yrd. Doç. Dr. Yunus Emre SELÇUK (YTÜ)

Haziran 2011

Aileme,

ÖNSÖZ

Tez çalışmalarım süresince yardımları ve katkıları ile her zaman yol gösteren tez danışmanım Yrd. Doç. Dr. Feza Buzluca başta olmak üzere, fikirlerinden her zaman yararlandığım ve desteklerini hiç esirgemeyen çalışma arkadaşlarım Ural Erdemir ve Umut Tekin'e; tez ve makale yazım sürecinde düzeltmeleriyle büyük emeği geçen ve hep yanımda olan Akif Kutlu ve Engin Dumlu'ya, hayatımın her döneminde olduğu gibi yüksek lisans eğitimim boyunca destek olan aileme sonsuz teşekkür eder, saygılarımı sunarım.

Haziran 2011

Sinan Eski
Bilgisayar Mühendisi

İÇİNDEKİLER

Sayfa

ÖNSÖZ ..... **v**
İÇİNDEKİLER ..... **vii**
KISALTMALAR ..... **ix**
ÇİZELGE LİSTESİ ..... **xi**
ŞEKİL LİSTESİ ..... **xiii**
ÖZET ..... **xv**
SUMMARY ..... **xvii**
1. GİRİŞ **1**
2. LİTERATÜR ÖZETİ **7**
3. YAZILIM KALİTESİ VE SÜRÜM DENETİM SİSTEMLERİ **13**
3.1 Yazılım Kalitesi 13
3.2 Metrikler 16
3.3 Sürüm Denetim Sistemleri 16
3.4 Yazılım Sınama Süreci 20
4. ÖNCELİKLİ SINANMASI GEREKEN SINIFLARIN METRİKLKER YARDIMIYLA BELİRLENMESİ **25**
4.1 Sistem Mimarisi 25
4.2 Hipotez ve Kullanılan Metrikler 27
4.3 Değişimlerin Sınıflandırılması 29
4.4 Doğrulama Yöntemi 32
5. DENEYSEL SONUÇLAR **35**
5.1 İncelenen Projeler 35
5.1.1 JFreeChart 35
5.1.2 Yari 35
5.1.3 UCDetector 36
5.2 Değişim Eğilimli Sınıfları Belirleme Yöntemleri 36
5.2.1 Rasgele seçim yöntemi 36
5.2.2 Karar ağaçları yöntemi 37
5.2.3 Metrik kombinasyonu yöntemi 40
5.2.3.1 Deneysel çalışmalar 42
JFreeChart 43
Yari 45
UCDetector 47
5.2.3.2 Sonuçlarının değerlendirilmesi 48
5.2.3.3 İyileştirme çalışmaları 49
5.2.3.4 Değişim maliyeti hesaplamada farklı yaklaşımlar 52
5.2.4 Yöntemlerin değerlendirilmesi 55
6. SONUÇ VE İLERİKİ ÇALIŞMALAR **57**
KAYNAKLAR **59**

KISALTMALAR

ISO : International Organization for Standardization
IEC : International Electrotechnical Commission
CMMI : Capability Maturity Model Integration
VCS : Version Control System
SCM : Software Configuration Management
SVN : Subversion
CVS : Concurrent Version System
GIT : Global Information Tracker
SWEBOK : Guide to the Software Engineering Body of Knowledge
GRASP : General Responsibility Assignment Software Patterns (Principles)
GOF : Gang of Four
AST : Abstract Syntax Tree
CK : Chidamber and Kemerer Metrics Suite
QMOOD : Quality Model for Object-Oriented Design
WMC : Weighted Methods per Class
LOC : Line of Code
CAM : Cohesion Among Methods
CBO : Coupling Between Object Classes
RFC : Response For a Class
LCOM : Lack of Cohesion in Methods
NOM : Number of Methods
NOF : Number of Fields
WEKA : Waikato Environment for Knowledge Analysis

ÇİZELGE LİSTESİ

Sayfa

Çizelge 4.1 : Yapısal değişimler. 31
Çizelge 5.1 : JFreeChart projesinin sürüm bilgileri. 35
Çizelge 5.2 : Yari projesinin revisyon bilgileri. 36
Çizelge 5.3 : UCDetector projesinin revizyon bilgileri. 36
Çizelge 5.4 : Rasgele seçim yöntemi sonuçları. 37
Çizelge 5.5 : Karar ağaçları yöntemi eğitim setinde kullanılan proje aralıkları. 38
Çizelge 5.6 : Karar ağaçları yöntemi sınama sonuçları. 39
Çizelge 5.7 : Metrik harf gösterimi 43
Çizelge 5.8 : JFreeChart projesinin Pearson's korelasyonu 43
Çizelge 5.9 : JFreeChart projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı 44
Çizelge 5.10 : Yari projesinin Pearson's korelasyonu 45
Çizelge 5.11 : Yari projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı 46
Çizelge 5.12 : UCDetector projesinin Pearson's korelasyonu 47
Çizelge 5.13 : UCDetector projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı 48
Çizelge 5.14 : JFreeChart projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%) 50
Çizelge 5.15 : Yari projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%) 51
Çizelge 5.16 : UCDetector projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%) 52
Çizelge 5.17 : Yapısal değişimlere farklı ağılıkların atanması. 53
Çizelge 5.18 : JFreeChart projesinin farklı değişim ağırlıkları için örtüşme oranları. 54

ŞEKİL LİSTESİ

Sayfa

Şekil 3.1 : Merkezi sürüm denetim sistemlerinin yapısı. 19
Şekil 3.2 : Dağıtık sürüm denetim sistemlerinin yapısı. 20
Şekil 4.1 : Sistem mimarisi 26
Şekil 4.2 : Doğrulama yönteminin akış şeması. 33
Şekil 4.3 : Proje sürümlerinin zaman aralıklarına bölünmesi. 34
Şekil 5.1 : Projelerin ilk aralıkları için eğitim süreci sonrası oluşan ağaç yapıları. .. 39
Şekil 5.2 : Sınıfların metrik değerlerinin normalizasyonu 41
Şekil 5.3 : Normalize değerler kullanılarak sınıf metrik sıralamaları. 42

NESNEYE DAYALI YAZILIMLARDA SINAMA VE BAKIM ÖNCELİKLİ SINIFLARIN BELİRLENMESİ

ÖZET

Günümüzde, toplam yazılım geliştirme maliyetlerinin yüzde ellisinden fazlasını yazılım bakım maliyetleri kapsamaktadır. Yazılım sınamaları ise bakım maliyetlerinin düşürülmesinde önemli bir role sahiptir. Yazılımın daha çok ve öncelikli olarak sınanması gereken kritik kısımlarının belirlenmesiyle, yapılacak sınamaların maliyetlerinin ve sürelerinin azaltılması sağlanabilir.

Genel olarak yazılımların kaliteli kısımlarında hata çıkma olasılığı azdır ve bu kısımlarda çıkan hataların düzeltilmesi kolaydır; diğer yandan yazılımın kalitesiz ve karmaşık kısımlarında ise durum bunun tam tersidir. Yazılımlardaki kalitesiz kısımların, bir yazılım için, bakım sürecindeki en kiritik kısımlar olduğu açıktır. Yazılım kalitesinin doğru biçimde ölçülmesi yazılımın iyi ve kötü kısımlarını ortaya çıkartacaktır.

Ölçme kavram olarak varlıkların özelliklerinin sayısallaştırılmasıdır. Yazılım metrikleri ise yazılımların kalitesini ölçmede kullanılabilecek en temel ölçme araçlardır; dolayısı ile yazılım metrikleri kullanılarak, kritik yani daha çok ve öncelikli olarak sınanması; bakım ve yazılımın gözden geçirilmesi sırasında öncelikli olarak değerlendirilmesi gereken kısımlar belirlenebilir. Proje yöneticileri sınama için ayıracakları kaynakları daha verimli bir biçimde ayarlayabilir; yazılım geliştiricileri yazılımın düşük kaliteli bu kısımlarından başlayarak yazılımın iyileştirilmesini sağlayabilirler.

Bu çalışma kapsamında nesneye dayalı metrikler ile nesneye dayalı yazılımların değişimleri arasındaki ilişki araştırılmıştır. Sonuçların çıkarılmasında, açık kaynak kodlu yazılımlardaki değişimler, yazılımların gelişim süreçleri boyunca ardışıl sürümleri analiz edilerek incelenmiştir. Elde edilen deneysel sonuçlar düşük kaliteli yazılım kısımlarının yazılım geliştirme ve bakımı süresince sıklıkla değiştiğini göstermiştir. Bu ilişki dikkate alınarak, bu çalışmada yazılımlardaki öncelikle sınanması gereken kritik kısımların ve değişime eğilimli sınıfların belirlenmesi için sistematik bir yöntem önerilmiştir, diğer yöntemlerle karşılaştırılmıştır ve sonuçları doğrulanmıştır. Açık kaynak kodlu projeler üzerine yapılan deneysel sonuçlar göstermiştir ki geliştirilen yöntemle ilerleyen sürümlerde en çok değişime uğrayacak kısımlar %80'e yakın oranda belirlenebilmektedir.

PREDICTION OF CLASSES HAVING PRIORITY AT OBJECT ORIENTED SOFTWARE TEST AND MAINTENANCE

SUMMARY

Software maintenance cost is typically more than fifty percent of the cost of the total software life cycle and software testing plays a critical role in reducing it. Determining the critical parts of a software system is an important issue, because they are the best place to start testing in order to reduce cost and duration of tests. Software quality is an important key factor to determine critical parts since high quality parts of software are less error prone and easy to maintain. As object oriented software metrics give important evidence about design quality, they can help software engineers to choose critical parts, which should be tested firstly and intensely. Also these parts should be considered for refactoring. In this paper, we present an empirical study about the relation between object oriented metrics and changes in software. In order to obtain the results, we analyze modifications in software across the historical sequence of open source projects. Empirical results of the study indicate that the low-level quality parts of a software change frequently during the development and management process. Using this relation, we propose a method that can be used to estimate change-prone classes and to determine parts which should be tested first and more deeply. We compare our method with random selection and decision tree approaches. Our metric-based method that can be applied at the last development phase of the software to predict critical classes, which tend to change in the future. These parts are also the low-level quality parts of the software. Therefore, they possibly need refactoring. Experimental results show that this approach can find up to 80 percent of the change-prone parts.

1. GİRİŞ

Yazılımların boyutlarının büyümesi ve karmaşıklıklarının artması ile birlikte yazılım üretim maliyetleri, geliştirme zamanları ve hata oranları da artmaktadır. Yazılım projeleri üzerine yapılan araştırmalar, yazılım projelerinin bakım maliyetlerinin projenin toplam maliyetinin %50'sinden fazla olduğunu [1] ve projelerin %47'sinde beklenenden daha fazla bakım masrafı ortaya çıktığını [2] göstermektedir. Yazılım projelerinin ve bakım masraflarının büyümesi ile yazılımlarda kalite kavramı, hem akademik hem de endüstüriyel alanda en çok çalışılan ve tartışılan konulardan biri halini almıştır. Amaç, gelişmeye ve değişimlere açık, hata oranları düşük, tekrar kullanılabilir kaliteli yazılımlar üretmektir. Çünkü hızla artan yazılım maliyetlerini azaltmak; ancak yüksek kalitede yazılımların geliştirilmesiyle veya mevcut yazılımların kalitelerinin arttırılmasıyla sağlanabilir [3].

Yazılım için kalite kavramı; kişiler tarafından farklı algılanan öznel bir kavram olarak karşımıza çıkmasına rağmen, kaliteli yazılım açısından kabul görmüş bazı yaklaşımlar mevcuttur. Bunlar içindeki en önemli örnekler GRASP yazılım prensipleri [4] ve GOF tasarım kalıplarıdır [5]. Yazılımların kalitesini sadece algı ile ifade etmek mümkün değildir; bu nedenle yazılımların kalitesinin ölçülebilmesi yani sayısallaştırılabilmesi gerekir. Yazılım kalitesinin nicel olarak ölçülebilmesi için yazılım metrikleri kullanılmaktadır. Yazılım metrikleri en basit haliyle yazılımın ya da uygulamanın verilerinden çıkarılan ölçüm değerleri olarak ifade edilebilir. Kaynak koddan çıkarılan satır ve metot sayıları basit metriklere örnek olarak verilebilir. Nesneye dayalı yazılımlarda, yazılımın temel bileşeni olan sınıfların yapısını ve birbirleri arasındaki ilişkiyi ifade etmek için uyum, bağımlılık ve karmaşıklık gibi üst seviye metrikler, literatürde CK metrik kümesinde [6] ve QMOOD kalite modelinde [7] tanımlanmıştır. Yazılımların metrik ölçüm değerlerinin yorumlanması, o yazılımın kalitesi hakkında çıkarsamalar yapılmasına ve buna bağlı olarak daha doğru kararlar verilmesine olanak sağlar.

Yazılım kalitesinin sağlanmasının ve sürdürülebilmesinin temel adımlarından bir diğeri, yazılımların doğru çalışıp çalışmadığının belirlendiği yazılım sınamalarıdır.

Genel olarak yazılım projeleri birbirini izleyen analiz, tasarım, kodlama, sınama ve ürünün oluşturulması süreçlerinden oluşur. Yazılım geliştirmenin adımları bu şekilde ardışıl olarak birbirini izlese de yazılım sınamaları, yazılım kalitesinin çok önemli bir parçasıdır ve ideal bir geliştirme süreci içerisinde, yazılım projelerinin tüm geliştirme sürecine yayılmaktadır. Sınama sürecinin yazılımın her evresinde yer aldığı ve her aşamaya karşılık bir sınama adımının karşılık geldiği bir yazılım geliştirme modeli: V Modelidir [8]. Bu modelde, yazılım sınama süreci; projenin geliştirilmesinin ilk aşamalarından itibaren, her aşamada yapılacak sınamaların tasarımı ve hazırlıkların yapılması ile uygulanmaya başlar. Planlanmış olan bu sınamalar, projenin gelişim süresince yürütülerek yazılımın kalitesi ve istenen özellikleri karşılayıp karşılamadığı kontrol edilmiş olur. Bu sayede kaynak kodda ya da tasarımda ortaya çıkabilecek olası hatalar mümkün olan en erken fazda belirlenir.

Yazılımlardaki hataların bulunma zamanları, bunların düzeltilmesi için harcanan zaman ve bakım maliyetlerini doğrudan etkilemektedir. Bir yazılım, ürün haline dönüştükten sonra bulunan hataların düzeltilme maliyeti oldukça yüksek iken, yazılımın geliştirilmesi süresince yapılan sınamalar ile hataların erken belirlenmesi bu maliyetleri oldukça azaltmaktadır. Boehm and Basili, yaptıkları araştırmada [9], bir problemin yazılım teslim edildikten sonra bulunmasının ve düzeltilmesinin, gereksinim analizi veya tasarım sırasında bulunmasından yüz kat daha fazla olduğunu göstermişlerdir. Bu nedenle yazılım sınamalarına ayrılan bütçenin geri kazanımı oldukça yüksektir.

Yazılım geliştirme maliyetinin yarısından fazlasını yazılım bakım maliyetleri oluşturmaktadır. Yazılım sınamaları, yazılımın kaynak kodundaki ya da tasarımındaki hataların yazılımın erken fazlarında bulunması ve bakım maliyetlerinin düşürülmesinde büyük öneme sahiptir. Ayrıca yazılımlar; yeni özellik ekleme, kalite iyileştirme ve hata düzeltme gibi farklı sebeplerle gelişim süreleri boyunca sürekli değişime uğramaktadır. Her bir değişimden sonra yazılımın bütünlüğünü ve kararlılığını korumak için yazılım sınamaların yapılması gereklidir. Ancak, yazılım sınamaları, yazılım hayat döngüsünün en çok zamana ve kaynağa ihtiyaç duyan faaliyetidir [10, 11].

Zamanı ve kaynakları daha etkin kullanabilmek için yazılım geliştiricilerin, sınama uzmanlarının ve yöneticilerin yazılımlarda öncelikli olarak sınanması gereken kritik kısımlarını belirlemesi gereklidir. Bir yazılım projesindeki tüm birimlerin ve tüm

işlevselliklerin sınanması, çok maliyetlidir bazen de zaman ve maliyet açısından mümkün değildir. Ayrıca yazılımda, kalitesi yüksek ve istikrarlı kısımlarının uzun süre sınanması kaynakların verimsiz kullanılmasına neden olur; çünkü bu zaman ve kaynaklar kalitesi daha düşük kısımların daha fazla sınanması ve yeniden düzenlenmesi için kullanılabilir. Özellikle, kısıtlı zaman ve kaynak altında hangi kısımların öncelikli sınanması gerektiği yönünde yapılacak öneriler, kaynakların ve zamanın daha etkin kullanılmasına yardımcı olur. Yazılımın ilerleyen sürümlerinde hangi kısımların değiştiğinin belirlenmesi, bu nedenle önemlidir. Bu sayede sınama ve gözden geçirme işlemleri sırasında hangi kısımlara öncelik verilmesi gerektiği bulunabilir ve yazılım etkin bir şekilde sınanabilir. Ayrıca yazılımdaki farklı kısımları değişim eğimliliklerine göre değerlendirmek; yazılımdaki hangi kısımların istikrarlı ve yeniden kullanılabilir olduğunu hangi kısımların ise tekrar gözden geçirilmesi gerektiğini belirlemeyi sağlar. Bu sayede yazılımın daha kolay anlaşılmasına, kaynakların daha etkin kullanılmasına ve kalitenin artırılmasına yardımcı olur.

Yazılımların kaliteli kısımlarında hata çıkma olasılığı diğer kısımlara göre daha azdır; ayrıca iyi bir tasarımda bu kısımlar eklenecek yeni özelliklerden az etkilenirler. Diğer taraftan kalite özellikleri daha düşük olan sınıflar hataya açıktır ve gelen değişiklik isteklerinden çok etkilenir. Dolayısı ile yazılımlardaki düşük kaliteli kısımlar yazılımların kritik kısımlarıdır ve yazılımın diğer kısımlarına göre daha değişim eğilimlidir. Bu nedenle, değişim eğilimli kısımlara sınamada öncelik verilmeli ve daha çok sınanmalıdır. Yazılım metrikleri, yazılımlardaki yüksek ve düşük kaliteli kısımların bulunması için kullanılabilir. Bu konudaki en önemli noktalardan biri metriklerin birlikte değerlendirilmesinin gerektiğidir. Çünkü metrikler tek başına çok fazla bir bilgi vermemektedir, yazılımlar hakkında daha anlamlı bilgiler ancak metriklerin birlikte değerlendirilmesiyle sağlanabilir. Hangi metriklerin, değişim eğilimli sınıfları bulmak için daha etkin olduğunu belirlemek; metrikleri birlikte değerlendirerek başarımı artırmaya çalışmak tez kapsamında incelenecektir.

Bu çalışmada nesneye dayalı yazılım metrikleri kullanılarak, yazılımlardaki kritik kısımların ve değişime eğilimli sınıflarının bulunması için metrik tabanlı bir yöntem önerilmiştir. Çalışmada, nesneye dayalı metrikler ile yazılım değişimlerinin arasındaki ilişkinin çıkarılması için, açık kaynak kodlu projelerin ardışıl sürümleri

incelenmiştir. Geliştirilen yöntem diğer yöntemlerle karşılaştırılmıştır ve deneysel sonuçlar ile bu yöntemi kullanarak ilerleyen sürümlerde en çok değişime uğracak kısımların %80'e yakın oranda belirlenebildiğini gösterilmiştir.

Bu çalışmada; yöntemin doğrulanması için Java programlama dili ile geliştirilmiş, farklı büyüklükte ve farklı türde açık kaynak kodlu üç proje incelenmiştir. Ayrıca, yazılım kaynak kodundan metriklerin ve değişimlerin çıkarılması için bir program geliştirilmiştir. Tez kapsamında, QMOOD [7] kalite modeli ve CK [6] metrik kümesi ile genel metriklerden seçilen metrikler kullanılmıştır. Ayrıca sonuçların doğrulanabilmesi için, yazılımların kaynak kodundaki "yapısal değişimleri" analiz eden ve yazılımın gerçekten değişen kısımlarını bulan bir değişim analiz yöntemi geliştirilmiştir. Yazılımların sürümleri arası değişimleri ölçmek için "değişim maliyeti" kavramı tanımlanmıştır ve bu kavram ardışıl iki sürüm arasındaki farkların ölçülmesi için kullanılmıştır. Değişim maliyeti hesaplanırken, yazılımlardaki değişimlerin yazılım mimarisini ve yapısını etkileme miktarını hesaba katmak için değişim türleri tanımlanmış ve bunlara ağırlıklar verilmiştir. Bu sayede, yorumlarda yapılan düzenlemeler gibi yazılım mimarisini etkilemeyen "yapısal olmayan değişimler" hesaba katılmaz iken, yazılımın birçok yerini etkileyebilecek bir değişim olan metod imzasındaki değişimin etkisi daha ön plana çıkartılmıştır.

Önerilen yöntemin değerlendirilmesi için, incelenen yazılımların, yeterli olgunluğa eriştiği kabul edilen, erken fazlardaki bir sürümü başlangıç noktası olarak seçilmiştir. Belirlenen bu sürüm için değişime eğilimli olan sınıflar, nesneye dayalı metrikler ve önerilen metot kullanılarak belirlenmiştir. Değişim analizi yöntemi ile yazılımın ardışıl sürümleri incelenmiş sınıflardaki değişimler çıkarılmış ve sınıfların değişim maliyetleri hesaplanmıştır. Değişim eğilimli olarak belirlenen sınıflar ile değişim analizi sonrası gerçekten değişen sınıflar arasındaki ilişkiler araştırılmıştır. Elde edilen sonuçlar, değişime eğilimli olarak belirlenen sınıfların çoğunun yazılımın sonraki sürümlerinde değiştiğini ve bu sınıfların yüksek değişim maliyetine sahip olduğunu göstermiştir.

Bu sonuçlara göre, önerilen yöntemin yazılımın son sürümlerinde kritik sınıflarının belirlenmesinde kullanılabileceği söylebilir. Bu sınıflar olgunlaşmamış ve kararsız olabileceğinden öncelikli olarak ve daha çok sınanmalıdır. Ayrıca bu sınıflar bakım ve gözden geçirme sırasında öncelikli olarak değerlendirilmelidir. Sınıfların yapısal değişimlerini analiz ederek önerdiğimiz sürümler arasındaki değişimleri bulma

yöntemi, nesneye dayalı yazılımlarda değişim eğilimli sınıfları belirlemek için yapılacak diğer çalışmalarda da sonuçları sınamak için kullanılabilir.

Yazının planı şu şekildedir. Literatürde daha önce yapılmış olan çalışmaların incelemesi Bölüm 2.'de yapılmıştır. Bölüm 3.'de çalışmada kullanılan yazılım kalitesi, metrikler, sürüm denetim sistemleri ve yazılım sınama süreci kavramları açıklanmıştır. Tez kapsamında geliştirilen sistemin mimarisi, kullanılan metrikler ve bunların değişim eğilimli sınıflarla ilişkisi ile ilgili hipotezler; değişim analizinde değişimlerin nasıl ele alındığı ve kullanılan doğrulama yöntemi Bölüm 4.'te anlatılmıştır. Değişim eğilimli dolayısı ile sınama ve bakım sırasında öncelik verilmesi gereken kritik sınıfların belirlenmesinde kullanılan ve önerilen yöntemler, deneysel çalışmalarda kullanılan örnek yazılımlar hakkındaki bilgiler, bunlar üzerinde yapılan incelemeler ve elde edilen sonuçlar Bölüm 5.'de verilmiştir. Bölüm 6.'daki sonuçlar ve ilerideki çalışmalar yazının son bölümünü oluşturmaktadır.

2. LİTERATÜR ÖZETİ

Nesneye dayalı metrikler ile yazılım kalitesi arasındaki ilişki birçok araştırmacı tarafından incelenmiştir.

Bazı araştırmacılar, sistem tasarımının değişebilirliğe ve değişimin yayılmasına olan etkisini araştırmışlardır. M.Chaumun ve diğerleri yaptıkları araştırmada [12] sistem tasarımının değişebilirliğe etkisini incelemek için bir değişim etkisi modeli oluşturmuşlardır. Yazılımın bir yerinde yapılan değişimin, yazılımdaki diğer hangi kısımları etkilediğini bulmanın; yazılımın değişimden sonra da kararlı ve doğru çalışmasını sağlamak için önemli olduğunu vurgulamışlardır. Bu nedenle; ana odakları, sistemin bir değişimi nasıl karşılayacağını ve sistemin değişebilirliğinin değişimleri ne kadar soğurabildiğidir. Yazılım projeleri birbiri ile etkileşimleri olan yazılım bileşenlerinden (sınıf) oluşmaktadır ve tasarım açısından sınıflar arası ilişkiler, sınıf içi değişimlerden daha çok etkiye sahiptir. Geliştirilen model dilden bağımsız olarak tasarım üzerine dayandırılmıştır. Değişimleri üç ana bölümde ele almışlardır: niteliklerin eklenmesi, silinmesi, tipinin ve etki alanının değişmesi; metotların eklenmesi, silinmesi, gerçeklemesinin, imzasının ve etki alanının değişmesi; sınıfların eklenmesi, silinmesi ve yapısının değişmesi. Sınıflar arası ilişkileri; bir sınıfın diğer bir sınıfa ait referansa sahip olması (association), bir sınıfın tanımının diğer bir sınıfın nesnesinin olmasına bağlılığı (aggregation), türeme (inheritance) ve bir sınıfın diğerini tetiklemesi (invocation) olarak dört şekilde ve sınıf içi değişikliklerin etkisi tanımlamak için sanal bir bağlantı (local) tanımlamışlardır. Faklı tipteki değişimler farklı sınıf kümelerini etkiler. Örneğin, bir değişkenin üzerinde yapılan değişim ona bağlı olan sınıfları etkilerken, eğer bu değişiklik bir ata sınıfta yapılırsa en azından türeyen tüm sınıfları etkiler. Ayrıca değişim yapıldığı bağlantının tipine görede bu etki değişmektedir. Örneğin bir sınıfın metodu public iken protected yapılırsa, bu değişiklikten bu metodu çağıran tüm sınıflar etkilenirken, türeyen sınıflar etkilenmez. Verilen bağlantılara göre değişimden etkilenecek olan sınıf kümeleri çıkarılmıştır. Çalışmanın deneysel kısmında, C++ ile yazılmış ve 1044 sınıfa sahip endüstriyel bir uygulamada, sadece

metodun parametre değişimine bağlı olan değişimin etkisini araştırmışlardır. Analiz kısmındaki performans problemlerinden dolayı uygulamanın tüm sınıfları yerine örneklenen kümeler üzerinde çalışmışlardır. Örneklemede üç grup oluşturulmuştur: 1,2 metodu olan sınıflar, 3 ile 29 arasında metodu olan sınıflar ve 30'dan fazla metodu olan sınıflar. Deneysel calışmada sınıfın ağırlıklı metot sayısı (WMC) metriği, sınıfın metot sayısına eşit olacak şekilde kullanılmıştır ve metodun paremetre değişimine bağlı olan değişim etkisi ile sınıfın WMC metriği arasındaki korelasyon araştırılmıştır. Elde edilen sonuçlara göre korelasyon en yüksek 0.55 olarak bulunmuştur ve düşük olduğu ifade edilmiştir.

Bazı araştırmacılar, yeni eklenen bir özelliğin ve var olan bir özelliğin değiştirilmesinin sınıfları etkileme olasılıklarını dikkate alarak değişim eğilimli sınıfları olasılıksal yaklaşımlarla belirleme üzerine çalışmışlardır. Tsantalis ve diğerleri bu konuda yaptıkları çalışmada [13]**,** bu yaklaşımla, nesneye dayalı sistemlerde verilen bir değişime karşılık diğer sınıfların değişim olasılıklarını belirlemeye yönelik bir yöntem önermişlerdir. Bir sınıf değiştiğinde, ondan etkilenen sınıfların da güncellenmesi gerekmektedir. Önerilen yöntemin uygulabilmesi için yazılımın yeterli sayıda sürümüne ihtiyaç duyulmaktadır. Bir sınıfın iç değişimin olasılığı bilgisi manuel olarak toplanmıştır. Sınıfların birbirini etkileme olasılıklarının hesaplanması, bu bilgiye oldukça bağımlıdır. Sharafat ve Tahvildari'nin makalesinde [14] yazılımların, Tsantalis ve diğerlerin önerdiği yönteme yakın olarak, kaynak kodunun geçmiş sürümleri ve UML diyagramlarındaki bağımlılıkları inceleyerek, ilerleyen sürümlerde hangi sınıfların değişeceğini olasıksal olarak hesaplamaktadır. Sınıfların iç değişimleri ile bir sınıftaki değişimin diğer bir sınıfı etkileme olasılığını çıkarmaktadırlar. İlk olarak bir sınıfların kendinden kaynaklanan değişim ihtimalini (P_{ic}) hesaplamaktadır, daha sonra UML diyagramlarından alınan bağımlılıklar göz önüne alınarak bir sınıftaki (i) değişimin diğer sınıfa (j) yayılma olasılığı (α_{ji}) çıkarılmaktadır. Buradan yola çıkarak bir sınıfın toplam değişim olasılığı (P_{is}) bulunmaktadır. Bir diğer çalısmada [15], [13]'de kullanılan yöntemin çok sınırlı bir versiyonunu kullanarak bir teknik sunmuşlardır; ancak gerçekleme ya da benzeri diğer yöntemlerle karşılaştırma vermemişlerdir.

Bir çalışmada [16], yazılımın önceki sürümleri, sürüm denetim sistemleri üzerinden incelenerek kaynak kodun bir yerinde yapılan değişimin bağlantılı olduğu diğer kısımlarda değişimlere neden olabilceği için, bu kısımları geliştiricilere önermek için

bir yöntem geliştirilmiştir. Ancak bu yöntemlerin uygulanması için ardışıl yazılım sürümlerine ihtiyaç duyulmaktadır.

L.Li ve A. Offut kullanıcı tarafından tanımlanarak girilen bir değişimin etkisini belirlemek için bağımlılık graflarına dayalı bir sistem önermiştir [17].

Yazılımın gelişim sürecindeki metriklerin değişimlerini analiz eden, bunlardan projelerin ilerleyen aşamalarındaki kalite değişimi hakkında bilgi çıkarmak üzerine yapılmış çalışmalar vardır.

Young Lee ve diğerleri tarafından yapılan çalışmada [18] fan-in/out bağımlılık ve uyum metrikleri ile yazılımın gelişim davranışlarını anlamak için açık kaynak kodlu JFreeChart yazılımını analiz edilmiştir. Yazılım metriklerini ve kalite özelliklerini çıkarmak için JamTool adında bir yazılım geliştirmişlerdir. Yaptıkları deneysel çalışmada incelenen yazılımdaki sınıf sayısı artışı ile bağımlılık ve uyum metrikleri arasındaki ilişkiyi incelemişlerdir. Ayrıca eklenen ve silinen sınıfların yazılımın bağımlılık ve uyum metrikleri üzerindeki değişimini araştırmışlardır. Silinen sınıfların düşük kalite özelliklerine sahip olduklarını bu nedenle bağımlılıklarının yüksek, uyumlarının düşük olmasını; eklenen sınıfların ise daha kaliteli olacaklarını bu nedenle bağımlılıklarının düşük, uyumlarının yüksek olmasını beklemektedirler. JFreeChart yazılımının ardışıl sürümlerindeki sınıf sayıları ve ortalama fan-in/out bağımlılık metrikleri ve uyum metriklerini çıkararak incelemişlerdir. Elde edilen sonuçlar bağımlılık metriklerinin sınıf sayısı ile ilişkili olduğunu göstermiştir. Uyum metriği için ise negatif korelasyon bulunmuştur. Ayrıca yeni eklenen sınıfların, silinen sınıflara göre fan-in metrik değerleri yüksek, fan-out metrik değerleri ise daha düşüktür. Bu da tekrar kullanılabilirlik için istenen bir durumdur. Metriklerin yazılımın değişim döngüsündeki durumları incelenerek yazılımın geçtiği aşamalar hakkında fikir sahibi olunabilir.

Çevik (agile) geliştirme yöntemi ile geliştirilen yedi proje, nesneye dayalı metriklerinin bu projelerdeki değişimlerini ve farklı proje içeriklerinin kaynak koddan nasıl etkilendiğini belirlemek için incelenmiştir [19]. Metriklerin kullanılmasının, çevik geliştirme yöntemi ile geliştirme yapan grupların yazılımı anlamalarını kolaylaştıracağını, iyileştirmelerin ve gelişimin sağlanması için günlük işlemlerini belirlemede yardımcı olacağını ileri sürmektedirler. Metrikler ile yazılımda hataları ve tasarım kusurları bulma üzerine çalışmalar yapılmıştır. Yapılan

bir çalışmada nesneye dayalı metrikler kullanılarak tasarım hatalarının bulunması ve düzeltilmesi için bir çatı [20] önerilmiştir. Yazılımlarda hata eğilimli sınıfların bulunması çalışılan bir diğer konudur. [21] ve [22]'de yapılan çalışmalarda CK metrikleri ile yazılımın hata eğilimli sınıfların bulunması incelenmiştir.

Sürüm denetim sistemleri ve yazılım sürümleri incelenerek yazılımlarda yapılan hataları ve ilerleyen sürümlerde hata çıkma ihtimali olan modülleri belirlemek üzerine yapılan araştırmalar vardır [23-25].

Kastro ve Bener, yazılımın yeni sürümündeki hata sayısını eski sürümlerine göre olan değişiklikleri dikkate alarak tahmin etmek için, yapay sinir ağlarını kullanan bir yöntem [23] önermişlerdir. Yazılıma eklenen değişiklikler: Eklenen bir özellik, algoritma değişikliği, hata ayıklama olabilir. Bunların yanı sıra kodun boyutundaki hacimsel değişiklikleri de dikkate alarak, sürümde çıkabilecek hata sayısını doğru olarak tahmin etmeye çalışmışlardır. Yapay sinir ağlarında kullanılmak üzere iki farklı eğitim seti tanımlamışlardır. İlk set, bir sürümdeki düzeltilen hata sayısı, eklenen özellik sayısı, tamamlanan değişiklik sayısı ve boyut olarak önceki sürümle farkıdır. İkinci set ise; eklenen satır sayısı, silinen satır sayısı, değiştirilen satır sayısı ve boyut olarak önceki sürümle olan farktan oluşmaktadır. Bu veriler kullanılarak incelenen yazılımların önceki sürümlerinde yapay sinir ağları eğitilmiş ve incelenen sürümde çıkacak hata sayısı belirlenmeye çalışılmıştır.

Brilliant ve Knight tarafından N-sürümlü yazılımlar üzerine yapılan bir araştırmada [24] bir program birbirinden bağımsız olarak, ayrı isterlerle yirmi yedi defa yazılmıştır. Yapılan hataların istatiksel olarak birbirinden bağımsızlığını araştırmışlardır. Ancak geliştiriciler tafından yapılan bazı mantıksal hataların benzer olduğunu bulmuşlardır. Yazılım geliştiriliken, farklı sürümler için kullanılan programlama dili gibi bazı farklılıkların, aralarında korelasyon olan hataları azaltmada önemli bir etkisinin olmadığını söylemişlerdir.

Mikrosoft Research tarafından desteklenen, Nagappan ve Ball tarafından 2005 yılında kod çalkalanma (code churn) ölçülerini kullanarak yapılan araştırmada [25]; relatif olarak kullanılan ve birbirleri ile korelasyonu olan ölçümlerin, hatalı kısımları belirlemede mutlak ölçümlerden daha iyi sonuç verdiği ortaya koyulmuştur. Değişime uğramış yazılımın bileşenleri üzerinde, yapılan değişimleri çıkararak kod çalkalanma ölçümlerini hesaplamışlardır. Windows 2003 işletim sistemi yazılımı

üzerinde yapılan çalışmada, hatalı olan ve olmayan çalıştırılabilir (binary) dosyaları belirlemede %89'luk başarıma ulaşmışlardır.

Yazılım sürüm depolarından elde edilen bilgileri analiz etme ve görselleştirme yöntemleri kullanarak, yazılımlardaki örtük bağımlılıkları ve tasarım hatalarını bulmak için çalışmalar yapılmıştır. Bayer ve Noack [26], yazılımın sürümlerinde birlikte değişen kısımları tesbit etmek ve önceki kümeleme yöntemlerinden farklı bir kümeleme yöntemi ortaya koymak için sürüm denetim sistemlerinden yararlanarak bir yöntem önermiştir ve yazılımda sıklıkta birlikte değişen kısımları graf ile göstermiştir. Yazılım bileşenleri olarak yazılımdaki dosyalar temel alınmıştır. Birlikte sıklıkla değişen dosyalar graf üzerinde birbirine yakın, az değişenler ise uzaktır. Yapılan bir başka çalışmada; yazılım sürümlerinden yapılan analizle, yazılımdaki dosyaların ve modüllerin değişimlerini görselleştirerek tasarım hatalarının ve mantıksal bağımlılıkların bulunmasına yardımcı olmak için Evolution Radar [27] geliştirilmitir. Bu görselleştirme yöntemi, sistemin yapısındaki modül bağımlılıkları anlamada ve onları dosya düzeyinde incelemede; bir modüldeki değişimin başka modüllerdeki örtük etkisini anlamada; bu bilgilerden yola çıkarak tasarım kusurlarını bulmakta kullanılabilir. Bu çalışmalarda kaynak kod analizi yerine, sürüm depolarından alınan bilgiler kullanılmıştır; yazılımların temel bileşeni olarak dosyalar dikkate alınmıştır ve yapısal değişimler yerine, dosyalar üzerindeki tüm değişimler kullanılmıştır.

Yazılım sürümleri üzerine yapılan bir diğer çalışma alanı, yazılım sürümlerindeki bilgilerin ve sürümler arasındaki farkların görselleştirilmesidir.

Voinea ve Talea tarafından yapılan çalışmada [28]; sürüm denetim sistemlerinden ham veriyi alıp bunları işleyebilen ve bazı özellikleri görselleştirebilen bir yazılım oluşturulmuştur. Bu sayede sürüm denetim sistemlerininden alınan verilerin görsel olarak daha anlaşılır hale getirilmesi amaçlanmıştır. Yazılımdaki geliştiriciler, onların kod üzerine etkileri, projeden ayrılmalarının riski; yazılımın üst seviye bloklarının belirlenmesi, yazılımın bakılabilirliği hakkında fikir yürütülebilmesini sağlamaktadır. Benzeri şekilde yazılım anlaşılırlığını artırmak ve yapılan değişiklikleri takip edebilmek için geliştirilmiş başka görsel araçlar da [29, 30, 31] vardır. Ancak bu araçlar, yapısal değişimlerden ziyade metin tabanlı olarak kullanıcıya verilen karşılaştırılmaların, renklendirilerek görselleştirilmesini sağlamaktadırlar.

Bu çalışmada, sınıfların metrik değerlerini analiz ederek değişime eğilimli sınıfların belirlenmesi için bir yöntem önerilmiştir. Metodun doğrulanması için seçilen sınama projelerinin farklı zaman aralıklarında değişim maliyetleri çıkarılmıştır. Yapılan deneyler sırasında; yazılımların değişim bilgileri, yazılımların ardışıl sürümlerinden ya da projelerde kullanılan sürüm denetleme sistemlerinden elde edilmiştir. Çalışmada yazılım sürümlerinin incelenmesi sadece yöntemin doğrulaması için kullanılmıştır; geliştirilen metot yazılımın tek ve tercihen son sürümü üzerinde kullanılabilir.

3. YAZILIM KALİTESİ VE SÜRÜM DENETİM SİSTEMLERİ

Bu bölümde yazılım kalitesi, yazılımın kalitesinin ölçülmesinde kullanılan metrikler, yazılım geliştirme sürecinin bir parçası olan sürüm denetim sistemleri ve yazılım sınama süreçleri hakkında genel bilgiler verilmiştir.

3.1 Yazılım Kalitesi

Yazılım kalitesi denildiğinde ilk akla gelen müşterinin gereksinimleri eksiksiz olarak karşılayan, belirlenen bütçe ve istenen zamanda tamamlanan, değişim isteklerinin kolaylıkla yapılabildiği yazılımlar akla gelmektedir. Yazılım kalitesini farklı açılardan incelemek mümkündür: Ürün olarak yazılım kalitesi, geliştirme süreci ve yazılımın iç özelliklerinin kalitesidir. Dolayısıyla yazılım kalitesi; yazılımı kullananların, yazılımı geliştirenlerin veya yöneticilerinin bakış açısıyla değerlendirilebilir. Müşteriler, yazılımları bir ürün olarak görür ve genel olarak kullandıkları programların gereksinimlerini karşılayan, kolay kullanılabilir, güvenilir ve yeterli performansa sahip olmasını ister. Diğer taraftan yazılım yöneticileri; yazılımların, zaman ve maliyet açısından beklenen hedeflere uygun olarak tamamlanmasını ister. Yazılım geliştiricileri ise yazılımın, kabul edilebilir düzeyde hatayla ürün haline dönüşmesini, bakımının kolay yapılabilir olmasını ve tekrar kullanılabilir olmasını ister.

Yazılımların kalitelerinin arttırılması amacıyla tanımlanmış standartların, yöntemlerin ve araçların büyük çoğunluğu, yazılımların müşteriye yansıyan kalite özellikleri ve yazılım geliştirme süreçleri üzerine yoğunlaşmaktadır. Yazılım geliştirme sürecinin iyileştirilmesi üzerine yoğunlaşan modellerden en bilineni Bütünleşik Yetenek Olgunluk Modeli'dir(CMMI) [32]. CMMI, firmaların yazılım planlama, geliştirme ve yapılandırma gibi yazılım süreçlerinin olgunluğunu değerlendirmek için geliştirilmiş bir modeldir. Uluslararası Standartlaştırma Organizasyonu (ISO) yazılımda kalite ile ilgili olarak, ISO/IEC 9126 [33] olarak adlandırılan yazılım ürünlerinin kalitesi anlatan ve sınıflandıran uluslararası bir standart yayınlamıştır.

ISO/IEC 9126 standardı kaliteyi dört temel kısım altında incelemektedir. Bu kısımlar şu şekilde ifade edilebilir [3, 33, 34]:

- **Kalite Modeli (ISO/IEC 9126-1):** Birinci bölüm kalite modeli olup kaliteyi tanımlamak için altı ana kısımdan oluşur ve daha çok yazılımı, kullanıcıların gözünden ve ürün bazlı olarak elle almaktadır. Ana kısımlar geniş kapsamlı olduklarından, her bir grubun altında, alt sınıflandırma yapılmıştır.
    - **İşlevsellik:** Yazılımın belirli şartlar altında, tanımlanan ihtiyaçları karşılamak için bir araya gelmiş olan özellikleri olarak tanımlanmaktadır. Uygunluk, doğruluk, birlikte çalışabilirlik, uyum ve güvenlik konuları bu kategori altında incelenmektedir.
    - **Güvenilirlik:** Yazılımın belirlenen şartlar altında, düzgün çalışma halini muhafaza edebilmesi olarak tanımlanmaktadır. Olgunluk, hata toleransı ve geri kurtarma konuları bu kategori altında incelenmektedir.
    - **Kullanılabilirlik:** Yazılımın kullanım kolaylığı sağlamak için kolay öğrenilebilme, anlaşılabilme özelliklerini ifade etmektedir. Öğrenebilme, anlaşılabilirlik, işletilebilirlik ve kullanıcı etkileşimi konuları bu kategori altında incelenmektedir.
    - **Verimlilik:** Yazılımın ihtiyaç duyulan ölçüde yeterli performansla çalışabilme becerisi olarak tanımlanmaktadır. Zaman ve kaynak kullanımı konuları bu kategori altında incelenmektedir.
    - **Bakılabilirlik:** Yazılımın yeni isteklere uyum sağlama, değişiklik veya düzeltme yapmaya cevap verme yeteneği olarak tanımlanmaktadır. Değiştirilebilirlik, sınanabilirlik, analiz edilebilirlik ve bağışıklılık konuları bu kategori altında incelenmektedir. Burada tanımlanan sınanabilirlik, yazılımın sınanabilmesi, yapılan değişikliklerin doğrulanabilmesi kabiliyetidir.
    - **Taşınabilirlik:** Yazılımın farklı çalışma ortamlarına uyum sağlayabilme yeteneği olarak tanımlanmaktadır. Adaptasyon yeteneği, yüklenebilirlik özellikleri**,** ortam değiştirme imkânı ve diğer yazılımlarla uyum konuları bu kategori altında incelenmektedir.

- **Harici Metrikler (ISO/IEC 9126-2):** Harici metrik, uygulamanın karaktertistiğini ve özelliklerini ölçmek için ölçüm metodudur. Bunlar, yazılımın sınama ya da ilerleyen dönemlerinde toplanan ölçümlerdir.
- **Dahili Metrikler (ISO/IEC 9126-3):** Dahili metrikler, yazılım tasarım ve kodlama safhalarında yazılım kalitesini doğrudan ya da dolaylı yoldan ölçmeye çalışır.
- **Kullanımda Kalite (ISO/IEC 9126-4):** Yazılım geliştirilmesi yapılan ortamda kullanıcıların, geliştirilen yazılımı kullanmalarını gözlemleyerek ve bilgi toplayarak gerekli ölçümlerin yapılmasını hedefler.

Yukarıda bahsedilen standart ve modeller daha çok yazılımların ürün olarak değerlendirilmesi ya da geliştirme süreçleri ile ilgilenmektedir. Yazılım geliştiricilerin gözünden baktığımızda ise yazılımın iç özelliklerinin kalitesi ön plana çıkmaktadır. Bunun için yazılımın tasarım aşamasından başlayarak, geliştirme süreci boyunca, kaynak kod gibi yazılım dökümanlarının kalitesinin ölçülebilmesi ve buna bağlı olarak geliştirilebilmesi gerekir. Günümüzde; yazılımlar, yaygın olarak, nesneye dayalı programlama dilleri kullanılarak geliştirilmektedir. Nesneye dayalı tasarımlarda ise yazılımların en temel bileşeni sınıflardır ve sınıflar arası bağımlılıklar, sınıfların ve sahip olduğu metotların uyumu, karmaşıklığı, sınıfların büyüklüğü en temel kalite bileşenleridir. Bunların nicel olarak ölçülebilmesi için yazılım metrikleri tanımlanmıştır. Bu ölçümler sayesinde geliştiriciler, sınıfların kod boyutunu, sınıfların yaptıkları işin kendi içerisinde uyumunu, sınıfların birbirleri ile olan bağlılıklarını anlayabilirler ve bazı kararları vermek için bu ölçümleri kullanabilirler. Bu noktada temel olarak duyulan ihtiyaç yazılım metriklerinin verdiği değerlerin neyi ifade ettiğinin bilinmesi ve yorumlanabilmesidir.

Kaliteli yazılımların üretilebilmesinde yazılım sınamaları büyük önem taşımaktadır. Bugün kabul gören yaklaşım sınamaların yazılım geliştirme sürecinin tüm aşamalarına yayılması yönündedir. Bu çalışmada birden fazla üst düzey kalite metriğinin birlikte kullanılarak yazılımda kritik, değişime açık, olabilecek sınıfların belirlenmesi; bu metrikleri kullanıcıdan soyutlayarak yapılmaktadır. Bu sayede geliştiriciler, özellikle iteratif çalışmalarda, kaynak kodu sınaması yapacakları zaman hangi sınıfların öncelikli olarak sınanmasına ya da bakım aşamasında bu sınıfların öncelikli olarak gözden geçirilmesine karar verebilirler.

3.2 Metrikler

Yazılımlarda sürekli kaliteyi sağlama için kullanılan yöntemlerin başında kalitenin ölçülmesi gelir. Bu sayede yazılımarın anlaşılması, onlar hakkında bilgi sahibi olunması ve bazı çıkarsamaların yapılması sağlanır. Yazılım metrikleri, en basit anlamda, kalitenin ölçülmesi için kullanılan ve genellikle tersine mühendislik yöntemleri ile kaynak kod gibi yazılım dökümanların doğrudan ya da hesaplanarak çıkarılabilen veya yazılımların çalışma zamanında elde edilen sayısal değerlerdir. Yazılım metrikleri; proje büyüklüğü, karmaşıklığı ve kalite gibi faktörlerin belirlenmesine yardımcı olur. Bu sayede ileriye dönük daha doğru kararların alınmasına olanak sağlar. Bu veriler; yazılımın bütçe, sınama ve bakım maliyetlerinin belirlenmesinde, yazılımdaki kritik kısımların bulunmasında, tasarım hatalarının tesbit edilmesinde kullanılabilir.

Yazılım metrikleri statik olarak yazılım kaynak kodu ve belgeleri üzerinden çıkarılabileceği gibi, yazılımların çalışma anında toplanan verilerle dinamik olarak da elde edilebilir. Yazılımı çalıştırmadan ve çalışması sırasında ölçülen bu metrikler, sırasıyla statik ve dinamik yazılım metrikleri olarak adlandırılırlar. Bu çalışmada statik metrikler, tersine mühendislik yöntemleri ile yazılım kaynak kodundan çıkarılarak kullanılmıştır.

Metriklerin ürettikleri veri tipleri ve aralıkları değişiklik göstermektedir. Bazı metrikler $[0,+\infty)$ aralığında tam sayı değer alırken, bazı metrikler $[0,1]$ aralığında gerçel sayı değer almaktadır. Metrikleri değerlendirebilmek için, metriklerin aldıkları değerlerin ne anlama geldiğini bilmek oldukça önemlidir. Ayrıca daha anlamlı sonuçlar elde edebilmek için, birden fazla metriğin bir arada değerlendirilmesi gerekmektedir.

Literatürde yaygın olarak kabul görmüş nesneye dayalı yazılım metrik kümesinden Chidamber & Kemerer metrik kümesi [6] ve QMOOD kalite modeli [7] kullanılmıştır. Kullanılan metriklerin açıklamaları ve hesaplama yöntemleri Bölüm 4.2 Hipotez ve Kullanılan Metrikler kısmında anlatılmıştır.

3.3 Sürüm Denetim Sistemleri

Sürümler, yazılım geliştirme sürecindeki aşamalarıdır. Bugün gerek açık kaynak kodlu olsun gerek ticari kapalı kodlu yazılımlar olsun; yazılımların geliştirilmesi

süresince sürüm denetim sistemleri (VCS) kullanılmaktadır. Yazılım konfigurasyon yönetimi (Software Configuration Management) sürecinde bu tip sistemler kullanılmaktadır. Bu sistemler, geliştirilen projenin herhangi bir dizin veya dosyasında yapılan değişiklikleri; değişimi yapan kullanıcın bilgileri, değişimin yapılma zamanı ve nedeni gibi bilgilerle beraber, değişimin yapıldığı dosyalara sürüm numaraları atayarak tutulmasını sağlar.

İyi bir sürüm denetim sisteminde aşağıdaki özelliklerin olması beklenir [35]:

- **Yedekleme ve Kurtarma (Backup and Restore):** Dosyalar düzenleme yapıldıktan sonra saklanabilmelidir ve istenildiği zaman daha önceki bir tarihteki ya da numarası verilen sürüme geri dönülebilmelidir.
- **Senkronizasyon (Synchronization):** Kullanılıcılar dosyaları paylaşabilmelidir ve son sürümle senkronize edebilmelidir.
- **Geri alma (Short/Long Term Undo):** Kullanılıcılar, kısa dönem geri alma işleminde, son yapılan değişiklikleri iptal ederek en son çalışan sürüme geri dönebilmelidir. Uzun dönem geri almada ise kullanıcılar çok daha uzun zaman önce yapılan değişikleri görebilmeli ve o dönemki sürümlere geri dönebilmelidir.
- **Değişiklik Takibi (Track Changes):** Kullanıcılar yapılan değişikliğin nedenini sisteme ekleyebilmelidirler. Değişikliğin kimin tafından yapıldığı bilgisi de tutulmalıdır.
- **Dal açma ve Birleştirme (Branching and Merging):** Geliştirilen kodun bir kopyası ayrı bir dal altına kopyalanır ve geliştirmenin bir kısmı ana deliştime dalının dışında burada geliştirilebilir. Daha sonra farklı dallarda yapılan değişimler, ana dal ile birleştirilebilmelidir.

Bu sistemler sayesinde, kaynak kodların birden fazla geliştirici tarafından daha kolay geliştirilmesi; geliştirme ve yayınlanmış sürümün bakım kısımlarının birbirinden ayrı olarak yönetilebilmesi sağlanmaktadır. Yine bu sistemler sayesinde geliştirme sürecinin tarihsel gelişimi kayıt altında tutulmaktadır ve süreç ile ilgili birçok bilgi saklanmaktadır.

Sürüm denetim sistemleri temel olarak istemci-sunucu mimarisine benzer bir yapıda çalışırlar. Yazılım sürümlerinin saklandığı depolar, sunucular üzerinde bulunur.

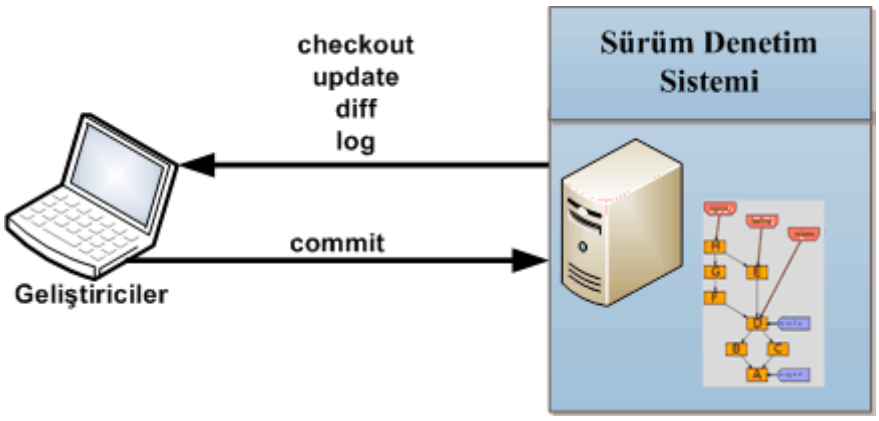
Geliştiriciler kullanılan sürüm denetim sistemine göre, uygun bir protokol ile bu sunucuya bağlanarak gerekli işlemleri yapabilirler.

Bu sistemler, merkezi bir yapıda olabilecekleri gibi dağıtık yapıda da olabilirler. Merkezi yapıda olanlara merkezi sürüm denetim sistemleri (Centralized SCM) denilmektedir, bu sistemlerin yapısı Şekil 3.1'de gösterilmiştir. Merkezi yapıdaki sürüm denetim sistemlerine SVN [36], CVS [37] ve SourceSafe [38] örnek olarak verilebilir. Özellikle açık kaynak kodlu yazılım geliştiricileri tarafından SVN, CVS en çok tercih edilen sürüm denetim sistemleridir. SVN, CVS'e göre daha sonra geliştirilmiştir ve CVS'e göre bazı artı özelliklere sahiptir. SVN sayesinde, depo içersindeki tüm dizin ağacı izlenir, tüm dizin ve dosyaların sürüm takibi yapılır; oysa CVS'de sadece dosyarın geçmişi izlenebilir. CVS'de dosyaların kopyalanması, yeniden adlandırılması ve başka bir dizine taşınması gibi işlemler takip edilemez; SVN'de ise dosya üzerinde yapılan tüm kopyalama, taşınma, yeniden adlandırma ve silme işlemlerine sürüm numaraları atanır. SVN'de atanan sürüm numaraları revisyon (revision) olarak adlandırılmaktadır. Merkezi sürüm denetim sistemlerinde yapılan temel işlemler şu şekildedir:

- **Teslim-alma (Check-out):** Sürüm denetim sisteminde tutulan projenin verilen bir sürümünün çalışma dizinine çekilmesini sağlamak için kullanılır.
- **Teslim-etme (Commit):** Dosyalarda düzenleme yapıldıktan sonra, yapılan değişikliklerin sürüm deposuna saklanması için gönderilmesi işlemidir. Burada birlikte değişen dosyalar birlikte teslim edilmiş olur ve geliştiriciler yaptıkları değişiklerin nedenlerini açıklamak için yorum ekleyebilirler.
- **Güncelleme (Update):** Geliştiricinin üzerinde çalıştığı dal ile onun sürüm deposunda bulunan son halinin senkronize edilmesidir. Bu işlemler ile kullanıcının sunucudan almamış olduğu değişiklikleri alması sağlanır.
- **Karşılaştırma (diff):** Sürüm denetim sisteminde tutulan iki sürüm ya da çalışılan proje ile sürüm denetim sistemindeki hali karşılaştırılabilir. Karşılaştırma işlemi metin tabanlı olarak yapılmaktadır ve dosyalar üzerinde yapılan tüm düzenlemeler değişim olarak görülmektedir. Eklenen, silinen ve değişen satırlar bu karşılaştırma işleminde farklı şekillerde gösterilmektedir.
- **Değişim Logları (log):** Bir dosyanın, dizinin ya da projenin bir sürümünden diğerine geçerken yapılan değişimlerin kim tarafından yapıldığı, yapılan

işlemlerin türü (ekleme, silme, düzeltme), hangi tarihte ve neden yapıldığı gibi bilgiler kullanıcıya gösterilir.

Bunun haricinde yeni bir dal açma (branch), dalları birleştirme (merge) ve etiketleme (tag) gibi işlemler de yapılabilmektedir.

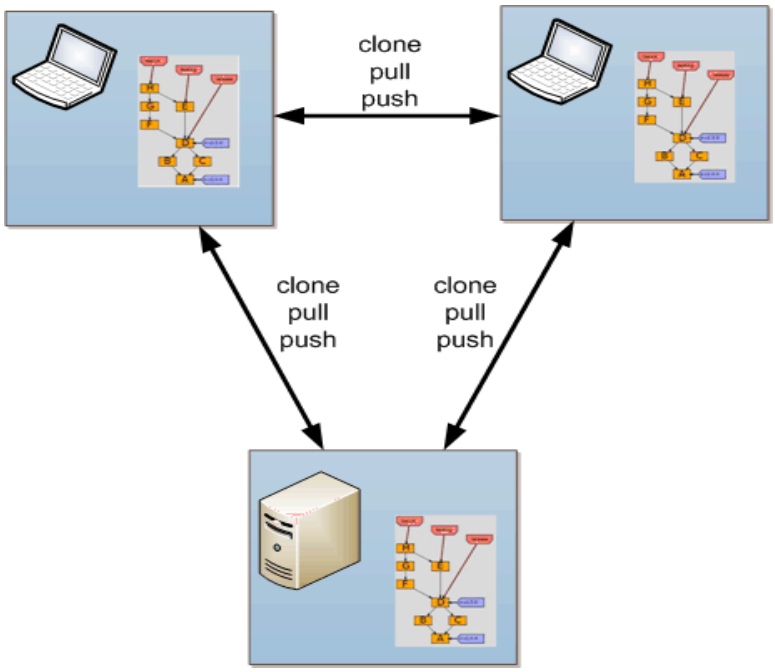


Şekil 3.1 : Merkezi sürüm denetim sistemlerinin yapısı.

Dağıtık yapıda olanlara dağıtık sürüm denetim sistemleri ya da merkezi olmayan yazılım konfigurasyon sistemleri (Decentralized SCM) denilmektedir, bu sistemlerin yapısı Şekil 3.2'de gösterilmiştir. Bu sistemlerde sürümleri tutan sunucular arasında sürüm deposunu alma (clone), değişimleri diğer sunucudan alma (pull), değişimleri diğer sunucuya gönderme (push) gibi işlemler de yapılmaktadır. Onun haricinde her bir sunucuya bağlı istemciler merkezi sürüm denetimlerinde olduğu gibi temel işlemleri yapabilirler. Özellikle büyük açık kaynak projelerin geliştirilmesinde dağıtık mimarideki sürüm denetim sistemleri kullanılmaktadır. Bunun en güzel örneği GIT [39] kullanılarak geliştirilen Linux kernelidir.

Sürüm denetim sistemlerinde, kaynak kodun tarihsel gelişimi, değişimi; bu değişimlerin kim tarafından ve ne zaman tutulduğu ile ilgili bilgiler saklanmaktadır. Sürüm denetim sistemlerine yapılan teslim-etme işlemleri kodun gelişme adımlarını detaylı olarak görmemizi sağlar. Hangi kaynak kod dosyalarının birlikte değiştiği, kaynak kod dosyalarında ne tür değişikliklerin yapıldığı; kişilerin kodun üzerine etkileri gibi bilgilere ulaşılabilir. Sistemdeki farklı sürümler karşılaştırılabilir ve eski yapılanlar incelenebilir. Bu sayede geliştirilen yazılımın hangi aşamadan hangi aşamaya geldiği gözlemlenebilir, zaman yönetimi ve yapılan işelerin takibi için kolaylıklar sağlar. Sürüm denetim sistemlerinde, bir dosyanın farklı sürümleri karşılaştırma metin tabanlı olarak yapılmaktadır. Bu karşılaştırmada; eklenenen, silinen veya değiştirilen satır sayısı farklı şekilde ifade edilerek gösterilmektedir.

Ancak bazı değişimler bu karşılaştırma araçları ile bulunamamaktadır. Örneğin, bir dosyadaki bir fonksiyon bir sürümden diğerine geçerken aynı dosyada önceki konumundan uzak bir yere taşınırsa, bu fonksiyon eski sürümden silinmiş ve yeniden eklenmiş olarak görülmektedir.



Şekil 3.2 : Dağıtık sürüm denetim sistemlerinin yapısı.

3.4 Yazılım Sınama Süreci

SWEBOK, tarafından yazılım sınaması için verilen tanım şu şekildedir: Yazılımın, sınırlı sayıda ve seçilmiş sınama senaryoları kullanılarak beklenilen davranışı gösterdiğinin yazılımın çalıştırılması ile dinamik olarak doğrulanmasıdır [40]. Yazılım sınama süreçleri, yazılımın ilk aşamalarından başlayarak gelişim süresince yazılımın var olan hatalarını ortaya çıkarmak, bunların düzeltilmesini sağlayarak ortaya konulacak olan ürünün hatasız ve isterlere uygun şekilde çalıştığını gösterme amacını içeren faaliyetlerdir. Ürün olarak teslim edilen sürümde sıfır hata olması pek mümkün değildir; ama önemli olan kritik hataların teslim edilen üründen arındırılmış olmasıdır.

Yazılım sınamalarının yapılması yazılım kalitesini artırmak, bakım ve gözden geçirme maliyetlerini azaltmak için gereklidir. Ayrıca, hataların bulunma zamanları ile düzeltilme maliyetleri arasında ilişki vardır;.sınamalar, yazılımdaki hataların erken fazlarda bulunmasını, bu sayede düzeltilme maliyetlerinin azalmasını sağlar.

Yazılım sınama süreci genel olarak sınama hazırlık süreci, dinamik sınama süreci ve sınamaların sonlandırılması kısımlarından oluşur [41]. Sınama hazırlık sürecinde; gereksinim analizi, yazılıma ait tasarım aşamalarında hangi sınamaların yapılacağı, sınamaların ne şekilde sonlandırılacağı ve dinamik sınamaların nasıl uygulanacağı ile ilgili planlar hazırlanır. Dinamik sınama süreci, kodlamanın bitmesine yakın bir dönemde başlar, yazılımın türüne ve özelliklerine göre uygulanacak yöntemler farklılık gösterebilir ve aşağıdaki kısımlardan oluşur [8, 41]:

- **Birim Sınama (Unit Testing) :** Birim sınamaları, hataların erken bulunup düzeltilebilmesi için oldukça büyük öneme sahiptir ve dinamik sınama sürecinin ilk aşamasıdır. Kaynak kodunun iç yapısı ve tasarım biçiminin bilinmesi gerektiğinden geliştiriciler tarafından yapılır. Sınama gereksinimleri çıkarıldıktan sonra sınama planlaması yapılır. Ardından, yazılımın alt birimleri ve bunların bileşenlerinden oluşan parçaların hangi sırayla ve nasıl sınanacağı belirlenir. Manuel olarak veya bazı sınama araçları kullanılarak ya da sınama programları yazarak otomatik olarak yapılır. Otomatik olarak yazılması, sınama işlemlerinin tekrar edilebilmesi açısından önem taşımaktadır. Bu sınamalarda, yazılmış olan kaynak kodun içerdiği alt bileşenler olan sınıfların, rutinlerin, kullanılan veri yapılarının doğruluğu sınanmış olur. Bu sayede ileriki sınama süreçleri öncesi, birimlerin düzgün çalışması sınanmış olur. Birim sınamaları sırasında, birimlerin performansı ölçülebilir ve bellek analizi de yapılabilir. Geliştirilmiş olan yazılımın mimarisi bu sınamaların yapılma zorluğunu belirleyen unsurların başında gelmektedir.
- **Tümleştirme Sınaması (Integration Testing) :** Bir uygulamanın altındaki bileşenler modüller, bağımsız uygulamalar, istemci/sunucu uygulamaları şeklinde olabilir. Tümleştirme sınaması ile bu bileşenlerinin beraberce uyum içinde çalışıp çalışmadığını doğrulama için yapılır. Özellikle istemci/sunucu uygulamaları ve dağıtık sistemlerde çok kullanılmaktadır.
- **Artımsal Tümleştirme Sınaması (Incremental Integration Testing**): Uygulamaya yeni bileşenler eklendikçe tümleştirme sınamalarının yapılmasıdır. Bu sayede yeni eklenen ve kendi başına diğer sınamalardan geçmiş olan modülün, diğer modüllerle başarılı şekilde çalışması sınanmış olur ve uygulamaya eklenmesi sağlanır.

- **İşlevsellik Sınaması (Functional Testing) :** Uygulamanın beklenen işlevleri yerine getirip getirmediğinin ilgili dökümanlara göre sınama uzmanları tarafından yapılır.
- **Regresyon Sınaması (Regression Testing) :** Projeye yeni bir özellik eklendiğinde ya da düzeltmeler yapıldığında, yazılımın diğer kısımlarında herhangi bir sorunun olup olmadığının kontrol edilmesidir ve uygulama geliştirme döneminin sonlarına doğru yapılır. Özellike herhangi bir şekilde birbirine bağlı modüller yazılımın başka taraflarında yapılan değişikliklerden etkilenebilir. Regresyon sınamaları ile, düzenlenen kısımlara bağlı olarak yeni hataların oluşup oluşmadığı sınanmış olur.
- **Zorlanım – Performans Sınaması (Performance Testing) :** Zorlanım ve performans sınamaları genel olarak yük sınaması olarak da bilinmektedir. Uygulamanın ağır koşullar altında çalışabilirliğinin sınanmasıdır. Beklenenin üzerindeki istek altında uygulamanın iş çıkarabilme yeteneği sınanmış olur.
- **Kullanıcı Kabul Sınaması (User Acceptance Testing) :** Son kullanıcı veya müşteri isteklerine dayanan son sınama işlemidir. Kullanıcıların, uygulamayı "kabul" etmeden önce, söz konusu uygulamanın gereksinimlerini ne ölçüde karşılayıp karşılamadığını belirlenir. Bu aşama başarıyla geçildiğinde uygulama devreye alınmaya hazır demektir.
- **Beyaz Kutu Sınama Tekniği (White Box Testing Technic) :** Projenin kaynak ve derlenmiş kodunun sınanmasıdır. Yazılımların kaynak kodundaki akış denetimleri, koşullar, hata yakalama mekanizmaları gibi iç özellikleri sınanmış olur.
- **Kara Kutu Sınama Tekniği (Black Box Testing Technic) :** Kara kutu sınamaları; yazılımın iç özellikleri hakkında bilgi sahibi olmaya gerek duymadan, ona dışarıdan kapalı bir kutu gibi bakıp, gereksinimlere cevap verip vermediğinin sınanmasıdır.

Yazılım sınama sürecinde bulunan hataların düzeltilmesinin ardından, sınama hazırlık sürecindeki sınama sonlandırılma kriterlerine göre yazılım sınamaları sonlandırılır.

Bu çalışma, daha çok, yazılım birim sınamaları ve beyaz kutu sınamaları gibi yazılımın kaynak kodunun doğrulandığı sınamalarda öncelikli ve daha çok sınanması

gereken sınıfların belirlenmesini hedeflemektedir. Ayrıca, yazılım gözden geçirme sürecinde, hangi sınıfların ilk olarak ele alınması gerektiğini belirlemede kullanılacaktır. Yazılım hakkında daha kolay fikir sahibi olmaları konusunda, yazılım grubuna sonradan katılan ya da yazılıma dışarıdan destek veren kişilerin, yazılımı incelemeye hangi sınıflardan başlamaları gerektiğini belirlemede yardımcı olacaktır.

4. ÖNCELİKLİ SINANMASI GEREKEN SINIFLARIN METRİKLER YARDIMIYLA BELİRLENMESİ

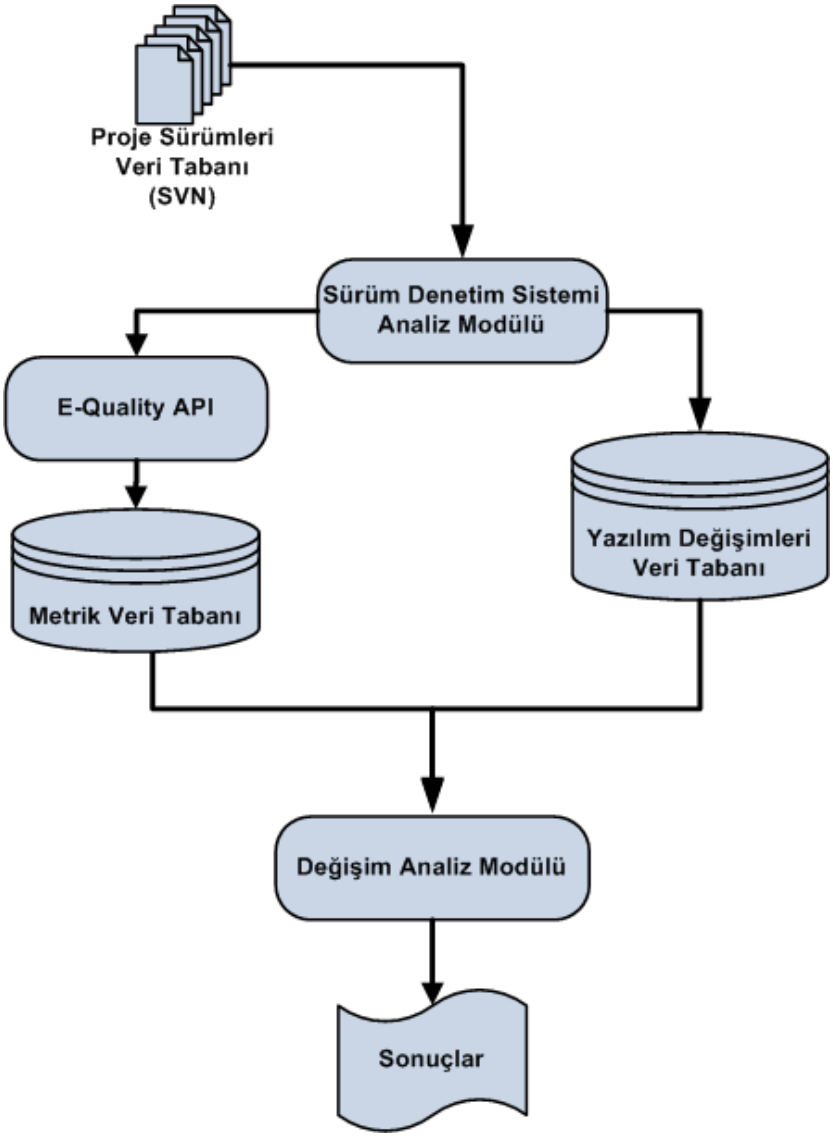
Bu bölümde tez kapsamında yapılan çalışmada kullanılan metrikler, bu metrikler ile değişim arasındaki ilişki için ortaya konan hipotezler ve değişim analizinde kullanılan yapısal değişim türleri ve değişim maliyeti hesaplama yöntemleri anlatılmaktadır.

4.1 Sistem Mimarisi

Sistem genel olarak, sürüm depolarından bilgi toplayan ve yapısal değişimleri çıkaran; metrikleri hesaplayan ve bu değişim ve metrik bilgilerini kullanarak gerekli analizleri yapan ve istenen sonuçların çıkarılmasını sağlayan alt modüllerden oluşmaktadır. Sistem mimarisi Şekil 4.1'de gösterilmiştir.

Sürüm denetim sistemi analiz modülü, sürüm depolarındaki revisyon bilgilerini çıkarmak ve analiz etmek için gerçeklenmiştir. Bu modül sürüm depolarına bağlanarak ordan değişim bilgilerini çıkarır ve bellek tabanlı bir veritabanına kaydeder. Sürüm deposu olarak SVN depoları ya da projelerin sürümleri toplanarak oluşturulan sürüm veritabanı kullanılmıştır. SVN sürüm deposundan alınan bilgiler: geliştirici bilgileri, hangi sınıfların birlikte değiştiği, değişim tarihleri, eklenen yorumlar ve kaynak koddur. Oluşturulan sürüm veritabanından ise projelerin farklı sürümlerinin kaynak kodlarına erişilmektedir. Çalışmada değişim analizi ve metriklerin hesaplanması kaynak kod üzerinden yapılmaktadır. Diğer alınan bilgiler, isim değişiklikleri, sınıfların taşınması gibi çıkarılan bilgilerin doğrulanması için kullanılmıştır. Bu bilgilerin çıkarılan veriler içerisinde olmaması durumunda kaynak kod karşılaştırılarak doğrulama yapılmıştır. Bu modül elde edilen bilgileri kullanarak ardışıl sürümlerindeki farkları çıkararak değişim veritabanına kayıt etmektedir. Burada yazılımdaki tüm yapısal değişimler çıkarılmakta ve kaydedilmektedir. Bu modül ayrıca, kendi laboratuvarımızda yazılım analizi için geliştirilmiş olan E-Quality API'sini [42] kullanarak yazılımdaki nesneye dayalı metrik bilgilerini çıkarmaktadır. Çıkarılan bu metrik bilgileri de veritabanında saklanmaktadır. Son

olarak değişim analizi modülü, yazılımın yaşam döngüsündeki tüm farksal değişimleri, kullanıcı tarafından verilen ağırlıklar ile istenen hesaplama bilgilerini kullanarak ve çıkarılan metrikleri analiz ederek istenen ölçüm sonuçların elde edilmesini sağlamaktadır.



Şekil 4.1 : Sistem mimarisi

Değişimlerin analizi kaynak kod üzerinden, soyut sentaks ağaçları (AST) kullanılarak tersine mühendislik yöntemi ile elde edilen yazılım modelleri üzerinden yapılmıştır. Öncelikli olarak kaynak kod soyut sentaks ağacı üzerinden her bir sürüm için yazılım modeli çıkarılmıştır; daha sonra ardışıl yazılım modelleri incelenerek aralarındaki yapısal farklar elde edilmiştir. Değişimlere verilen ağırlıklar ve kullanılacak denklemler göz önünde bulundurularak, sınıfların değişim maliyetleri hesaplanmıştır.

4.2 Hipotez ve Kullanılan Metrikler

Değişime eğilimli sınıfların belirlenmesi için CK metrik kümesinden, QMOOD kalite modelinden ve genel metriklerden toplam sekiz metrik seçilmiştir. Seçilen metriklerin açıklamaları ve bu metriklerin sınıfların değişim eğilimi ile ilgisi aşağıda verilmiştir.

Sınıfın Ağırlıklı Metot Sayısı - Weighted Methods per Class (WMC): Bir sınıfın metotlarının ağırlıklı toplamı olarak tanımlanmıştır [6]. WMC, sınıf karmaşıklığının bir ölçüsüdür ve bu çalışmada metodların karmaşıklıklarının toplamıdır. Bir sınıfın n adet metodu varsa ve c_i sınıfın i'inci metodun karmaşıklığı ise WMC aşağıdaki denklemdeki gibi hesaplanır (4.1).

$$\mathrm{WMC} = \sum_{i=1}^{n} c_i \qquad \textbf{(4.1)}$$

Tüm metotların karmaşıklığının bir kabul edildiği durumda metot sayısına eşittir. Bu çalışmada metodların karmaşıklığı için, bir metodun kendi içinde kaç farklı yol izleyerek yürütülebileceğini gösteren, McCabe çevrimsel karmaşıklığı [43] kullanılmıştır. Daha karmaşık sınıflar daha hata eğilimlidir [44], sınanmaları ve analiz edilmeleri daha zordur. Metot sayısı çok olan sınıfların uygulamaya özgü olma ihtimali yüksek olduğundan tekrar kullanılabilirlikleri daha düşüktür. Ayrıca; karmaşık sınıflar, hata düzeltme ve iyileştirme gibi değişimlerden çok etkileneceğinden değişime daha çok eğilimlidirler.

Sınıfın Satır Sayısı – Line of Code (LOC): Bir sınıfın gövdesindeki ve metotlarındaki boş ve yorum olmayan satırların toplam sayısıdır. Sınıfın büyüklüğünün ölçüsüdür ve dolaylı olarak karmaşıklığı ile ilgilidir. Büyük sınıfların daha fazla değişim eğilimli olması beklenmektedir.

Nesne Sınıfları Arasındaki Bağımlılık - Coupling Between Object Classes (CBO): Bir sınıfın bağlı olduğu sınıfların sayısıdır. Bir sınıfın niteliklerini ve metotlarını kullandığı sınıflar ile bu sınıfın niteliklerini ve metotlarını kullanan sınıflar dikkate alınarak hesaplanır. Türeme ilişkileri hesaplamanın dşında tutulur. Bağımlılık ölçüsü bir sınıfın tekrar kullanılabilirliğinin ve sınanabilirliğinin bir göstergesidir. Sınıflar arası yüksek bağımlılık modüler bir tasarımı olmusuz etkiler ve tekrar kullanılabilirliği düşürür. Bir sınıf başkalarına çok bağlı ise, o sınıflarda yapılan değişimler sınıfın kendisinde de düzenlemeye gidilmesine neden olabilir. Ayrıca,

bağımlılık yazılım parçalarının daha kapsamlı sınanmasını gerektirir. Dolayısı ile sınama maliyetlerini de artırır.

Metotların Uyumluğu - Lack of Cohesion in Methods (LCOM): Sınıfın metotlarının birbirleri ile ne kadar uyumlu olduğunun ölçüsüdür.

Literatürde bu metriğin farklı hesaplama şekilleri olmakla birlikte [3], bu çalışmada Chidamber ve Kemerer'in [6]'daki tanımı kullanılmıştır.

M_i , bir sınıfın i'inci metodu ve $\{I_i\}$ bu metodun eriştiği niteliklerin kümesidir. Bu şekilde, tüm metodların eriştiği nitelik kümeleri çıkarılır. Bu durumda, hiçbir nitelik değişkeni ortak olmayan metot çiftleri kümesi P ve en az bir ortak nitelik değişkeni olan metot çiflerinin kümesi Q olsun. Bu durumda LCOM aşağıdaki şekilde hesaplanır (4.2).

$$\mathrm{LCOM} = \begin{cases} |P| - |Q| & \text{eğer } |P| > |Q| \\ 0 & \text{eğer } |P| \le |Q| \end{cases} \qquad \textbf{(4.2)}$$

Uyumluluğun düşük olaması bu sınıfın alt parçalara ayrılabileceğini de gösterir. Çünkü düşük uyumluluk bir sınıfın birden fazla görevi yerine getirmek için gerçeklendiğini ifade etmektedir. Bu görevlerden birinde hata düzeltme ya da yeni özellik eklemek için yapılan değişiklikler bu sınıfı etkileyecektir. Ayrıca, uyumun düşük olması kodun anlaşılabilirliğini düşürmektedir ve sınıfların daha hata eğilimli olmasına neden olmaktadır [44] .

Sınıfın Tetiklediği Metot Sayısı - Response For a Class (RFC): Verilen sınıftan bir nesnenin metotları çağrıldığında, bu nesnenin tetikleyebileceği tüm metotların sayısıdır. RFC metrik değerinin yüksek olması o sınıfın karmaşıklığının yüksek olduğunu; dolayısı ile hata çıkma olasılığının da fazla olduğunu gösterir. Bu sınıfların sınanması ve hataların düzeltilmesi daha zordur. Bu metrik için hipotezimiz; bu metrik değerinin yüksek olduğu sınıfların, yazılımın ilerleyen dönemlerinde, değişimlerden etkilenme ihtimalinin yüksek olduğudur. Bu nedenle bu sınıflar öncelikli olarak ve daha çok sınanması gereken sınıflar arasında yer almalıdır.

Metotlar Arası Uyumluluk - Cohesion Among Methods (CAM): Uyumun bir ölçüsüdür. Metotlar ile nitelikler arasındaki ilişkiler yerine metotların parametre tiplerini dikkate alarak hesaplanır. Metotların imzaları arasındaki benzerliğin bir ölçüsüdür. Tam tanımı Bansiya'nın metrik kümesi çalışmasında yer almamaktadır.

Literatürde metotların imzalarına bakarak uyumluluk ölçen farklı metrik tanımları bulunmaktadır. Bunlar temelde metotların parametre kullanım matrisi üzerinde çalışırlar ve imzalardaki uyumluluğu tam uyumlu olmaya yakınlıkla kıyaslayarak ölçerler. Metotlar ve tüm parametreler numaralandırılır, i. metodun imzasında, j. parametre kullanılıyorsa, matriste Mij 1 aksi halde 0 olur. LCOM metriği gibi bu metrik de sınıfların değişime eğilimleri ile ilişkilidir.

Nitelik Sayısı - Number of Fields (NOF): Sınıfın niteliklerinin sayısıdır. Bir sınıfın sahip olduğu nitelik sayısı fazla ise değişim eğilimi fazladır.

Metot Sayısı - Number of Methods (NOM): Sınıfın metotlarının sayısıdır. Sınıfta tanımlı metot sayısı, tüm metotların karmaşıklığının bir birim kabul edildiği durumdaki WMC metriği ile aynı değer ifade eder. Sınıfın metot sayısı sınıfın karmaşıklığının ve büyüklüğünün bir göstergesidir. Bir sınıfın sahip olduğu metot sayısı fazla ise değişim eğilimi fazladır.

WMC, CBO, LCOM ve RFC metrikleri CK metrik kümesinden [6]; CAM ve NOM metrikleri QMOOD kalite modelinden [7] seçilmiştir.

4.3 Değişimlerin Sınıflandırılması

Önerilen yöntemin ve incelenen yöntemlerin değerlendirilmesi için, seçilen tasarım metrikleri ile yazılımdaki değişimler arasındaki ilişkiler incelenmiştir. Bu çalışmada, yazılımlardaki değişimleri tasarımsal düzeyde incelemek için, kaynak kod analizine dayalı bir yöntem izlenmiştir ve değişim sınıfları oluşturulmuştur. Bu sınıfların belirlenmesinde; nitelik, metot, sınıf gibi nesneye dayalı yazılım bileşenleri, bunlar arasındaki ilişkiler ve kaynak kod üzerindeki olası değişim türleri göz önünde bulundurulmuştur. Ayrıca, bu değişim sınıflarına ağırlıklar verilmesi ile hesaplanan değişim maliyeti kavramı tanımlanmıştır.

Yazılımın mimari değişimleri, yazılımın gelişimsel süresince kaynak kodu incelenerek çıkarılmıştır. Yazılımın kaynak kodundaki değişimler: Yapısal değişimler ve yapısal olmayan değişimler olarak iki kısıma ayrılmıştır. Yapısal değişikler yazılımın derlenen çalıştırılabilir kodunu değiştirirken, yapısal olmayan değişimler çalıştırılabilir kodunu değiştirmemektedir. Yorum ve boşluk ekleme, silme, değiştirme; kod hizalama veya bir metodun sınıfın içerisinde yerinin değiştirilmesi yapısal olmayan değişimlere örnek olarak verilebilir ve bu tür

değişimler değişim maliyeti hesaplanırken ihmal edilmiştir. Nesneye dayalı yazılımların temel bileşeni sınıflardır. Sınıflar üzerinde ekleme, silme, taşıma ve sınıfın içinde yapılabilecek değişimleri hesaplayabilmek için; sınıfların nitelik, metot gibi sahip oldukları alt bileşenler belirlenmiştir. Dolayısı ile yapısal değişimler belirlenirken, nesneye dayalı yazılımlardaki alt bileşenler ve bunlar üzerinde yapılabilecek işlemler ile bunların aralarındaki ilişkiler dikkate alınmıştır.

İsim değiştirme ve sınıfın taşınması işlemleri yapısal değişim olarak değerlendirilmiştir. Değişim analizinde bu operasyonların belirlenmesi, hesaplamanın sağlıklı yapılabilmesi için önem taşımaktadır. Bu işlemlerin belirlenemediği durumda; bu işlemlerin yapıldığı yazılım birimlerinin operasyondan önceki hali silinmiş, yeni hali eklenmiş gibi bir etkiye sahip olmaktadır. Kaynak kod analizinde: Sınıflar sahip oldukları metotlar, nitelikler, türedikleri sınıflar, gerçekledikleri arayüzler, onlardan türeyen sınıflar; metotlar ise sahip oldukları parametreler, eriştikleri nitelikler, çağırdıkları ve onları çağıran metotlar göz önüne alınarak analiz edilmiştir. Çıkan sonuçların doğrulanması kaynak kodların karşılaştırılmasıyla yapılmıştır.

Farklı türdeki yapısal değişimler yazlımların tasarımları üzerinde farklı etkiye sahiptir. Örneğin sınıfın bir metodu içerisindeki yerel değişimler programın diğer kısımlarını etkilemezken, metodun parametre listesindeki bir değişim farklı kısımları etkileyebilir. Farklı değişim türlerinin ayırt edilebilmesi için değişimlere, yazılım tasarımına etkilerine göre farklı ağırlıklar verebiliriz. Tez kapsamında, yazılımlardaki olası değişimler incelenerek belirlenen yapısal değişimlerin kısaltmaları, açıklamaları, ağırlıkları ve çalışmada genel olarak kullanılan ağırlıklar (KA) Çizelge 4.1'de verilmiştir.

Burada tanımlanan değişim işlemleri ve onlara verilen ağırlıklar kullanılarak, nesneye dayalı yazılımların alt bileşenlerinin değişim maliyeti hesaplanır. Bu maliyetler: Nitelik ekleme maliyeti (Field Addition Cost-FAC), nitelik silme maliyeti (Field Deletion Cost-FDC), nitelik değişim maliyeti (Field Change Cost-FCC); metot ekleme maliyeti (Method Addition Cost-MAC), metot silme maliyeti (Method Deletion Cost MDC), metot değiştirme maliyeti (Method Change Cost-MCC), sınıf ekleme maliyeti (Class Addition Cost-CAC), sınıf silme maliyeti (Class Deletion Cost-CDC), sınıf değiştime maliyeti (Class Change Cost-CCC) olarak tanımlanmıştır.

Çizelge 4.1 : Yapısal değişimler.

Değişim Türü	Açıklama	Ağırlık	KA
MA	Bir sınıfa bir metot eklenmesi sabiti	W_{MA}	0
MD	Bir sınıftan bir metodun silinmesi sabiti	W_{MD}	0
FA	Bir sınıfa bir nitelik eklenmesi	W_{FA}	1
FD	Bir sınıftan bir niteliğin silinmesi	W_{FD}	1
CA	Bir pakete bir sınıfın eklenmesi sabiti	W_{CA}	0
CD	Bir paketten bir sınıfın silinmesi sabiti	W_{CD}	0
R	Sınıf veya metodun isminin değiştirilmesi	W_{R}	1
M	Bir sınıfın başka bir pakete taşınması	W_{M}	1
MC	Görünürlük değişimi (public/private/protected)	W_{MC}	1
FTC	Nitelik tipinin değiştirilmesi	W_{FTC}	1
SCC	Sınıfın türediği sınıfların değişimi	W_{SCC}	1
IIC	Gerçeklenen arayüzün değiştirilmesi	W_{IIC}	1
MLC	Metodun yerel değişiklikleri sabiti	W_{MLC}	0
MRTC	Metodun dönüş tipinin değişmesi	W_{MRTC}	1
MPC	Metodun parametre değişimi	W_{MPC}	1
MLOCC	Metodun yerel operasyon ve kontrol değişiklikleri	W_{MLOCC}	1
MAAF	Metoda nitelik erişimi eklenmesi	W_{MAAF}	1
MDAF	Metoddan nitelik erişimi silinmesi	W_{MDAF}	1
MCAF	Metodun eriştiği niteliğin değiştirilmesi	W_{MCAF}	1
MACM	Metoda metot çağrısı eklenmesi	W_{MACM}	1
MDCM	Metoddan metot çağrısı silinmesi	W_{MDCM}	1
MCCM	Metodun çağırdığı metodun değiştirilmesi	W_{MCCM}	1

Değişim maliyetlerinin hesaplanması ile ilgili geliştirilen denklemler aşağıda verilmiştir (4.3).

$$
\left.
\begin{aligned}
\mathrm{FDC} &= W_{\mathrm{FD}} \\
\mathrm{FAC} &= W_{\mathrm{FA}} \\
\mathrm{FCC} &= W_{\mathrm{MC}} + W_{\mathrm{FTC}} \\
\mathrm{MDC} &= W_{\mathrm{MD}} + \sum W_{\mathrm{MDAF}} + \sum W_{\mathrm{MDCM}} \\
\mathrm{MAC} &= W_{\mathrm{MA}} + \sum W_{\mathrm{MAAF}} + \sum W_{\mathrm{MACM}} \\
\mathrm{MCC} &= W_{\mathrm{MLC}} + W_{\mathrm{MLOCC}} + W_{\mathrm{R}} + W_{\mathrm{MC}} + W_{\mathrm{MRTC}} + W_{\mathrm{MPC}} \\
&\quad + \sum W_{\mathrm{MAAF}} + \sum W_{\mathrm{MDAF}} + \sum W_{\mathrm{MCAF}} \\
&\quad + \sum W_{\mathrm{MACM}} + \sum W_{\mathrm{MDCM}} + \sum W_{\mathrm{MCCM}} \\
\mathrm{CDC} &= W_{\mathrm{CD}} + \sum \mathrm{FDC} + \sum \mathrm{MDC} \\
\mathrm{CAC} &= W_{\mathrm{CA}} + \sum \mathrm{FAC} + \sum \mathrm{MAC} \\
\mathrm{CCC} &= \sum \mathrm{FAC} + \sum \mathrm{FDC} + \sum \mathrm{FCC} + \sum \mathrm{MAC} \\
&\quad + \sum \mathrm{MDC} + \sum \mathrm{MCC} + W_{\mathrm{R}} + W_{\mathrm{M}} + W_{\mathrm{MC}} + W_{\mathrm{SCC}} + W_{\mathrm{IIC}}
\end{aligned}
\right\} \quad \textbf{(4.3)}
$$

Hesaplamalar sırasında, değişim ağırlıkları sadece ilgili değişim varsa ve hesaplamaya katılması seçilmişse hesaplamaya katılmıştır. Bu çalışmada değişim maliyeti hesaplanırken yapısal değişim ağırlıkları MLC, MA, MD, CA ve CD

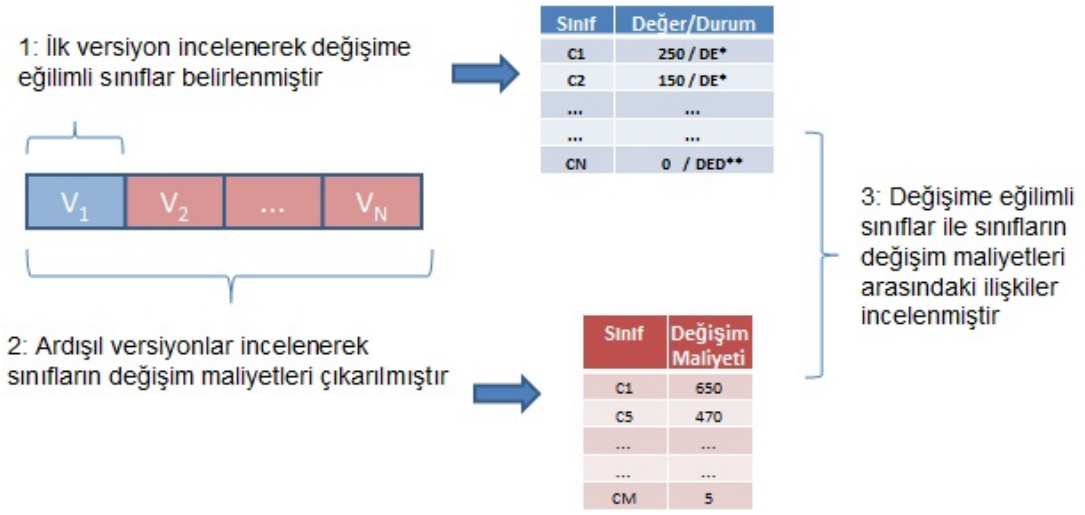
sabitleri hariç olmak üzere diğerlerine "1" verilmiştir. Metot ve sınıfların hesaplanmasında sabitlere değer atayarak sadece onlar kullanılabildiği gibi, metot ve sınıfların sahip oldukları veya eriştikleri nitelik ve metotların ağırlıkları toplamı da dikkate alınabilir. İstenirse metot ya da sınıf ekleme, silme işlemlerine ekstra ağırlık vermek için bu sabitlere sıfırdan farklı ağırlık verilerek de kullanılabilir. Değişim maliyeti hesaplama, bu şekilde esnek yapıda tutulmuştur. Metot ekleme, silme ve yerel değişim için sabitler kullanılması yerine metodun eriştiği nitelik ve çağırdığı metotların ağırlıkları ile yerel değişimler için kontrol, döngü ve metot içi diğer değişikliklere verilen diğer ağırlıkların toplamı kullanılmıştır. Benzeri şekilde sınıflar için de ekleme, silme ve değişim maliyetleri hesaplanırken sahip oldukları nitelik ve metotların değişim maliyetleri toplamı hesaba katılmıştır. O nedenle burada kullanılan sabitlerin değerleri metot ve sınıflar için "0" olması tercih edilmiştir. Değişim zorluğunu hesaba katmak için metot değişim maliyetleri hesaplanırken metot ekleme, silme veya yerel değişimine sabit bir katsayı vermek yerine, erişilen nitelikler ve cağrılan metotların durumu dikkate alınmıştır. Benzer şekilde sınıf silme maliyeti hesaplanırken, sabit bir katsayı yerine sınıf içerisindeki niteliklerin ve metotların değişim maliyetlerinin toplamı kullanılmıştır. Sınıfların ilk büyüklüklerine, nitelik ve metot maliyetleri toplamına olan bağımlılığı ortadan kaldırmak için sınıf ekleme maliyeti ihmal edilmiştir. Aksi halde değişim maliyetlerinin hesaplanması sınıfların ilk sahip oldukları nitelik ve metot sayısına bağlı olacaktır.

Bu denklemler iki sürüm arasındaki değişim maliyetlerinin hesaplanması içindir. Yazılımın ardışıl sürüm serisi analiz edilirken, değişim maliyetleri birbirini takip eden ardışıl sürüm çiftleri arasındaki değişim maliyetlerinin kümülatif toplamı olarak hesaplanmaktadır. Değişim maliyet analizi sonrası sınıfların değişim maliyetlerine göre sıralı bir liste oluşturulmuştur. Bu listenin en üsteki %10'u "en çok değişen sınıf" olarak işaretlenmiştir.

4.4 Doğrulama Yöntemi

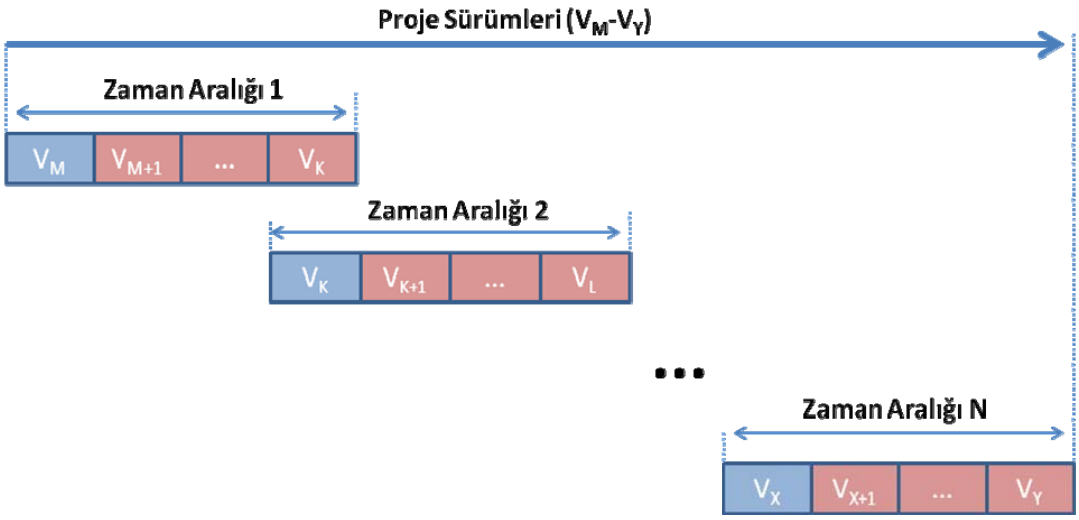
Bu çalışmadaki amaç yazılımların son sürümlerinde sınama ve bakım öncelikli sınıfların belirlenmesine yardımcı olmaktır. Ancak, doğrulama yapılabilmesi için yazılımın olgunluğa eriştiği düşünülen bir sürümünden başlanarak, takip eden ardışıl sürümleri incelenmiştir. İncelenen yazılımların seçilen ilk sürümleri belirli bir

olgunluğa eriştiği kabul edilerek, sınanması gereken ve kritik sınıfların belirlenmesi için kullanılacak olan sürümler olarak belirlenmiştir. Dolayısı ile değişim eğilimli sınıfların belirlenmesi, yazılımların seçilen ilk sürümü üzerinde yapılmıştır. Bu sürüm üzerinde; kullanılan yönteme göre, sınıflar ya "değişim eğilimli (DE)", "değişim eğilimli değil (DED)" şeklinde etiketlenmiş ya da değişim eğilim maliyetlerine göre sıralı bir liste oluşturacak şekilde çıkarılmıştır. Ardından, yazılımların ardışıl sürümleri üzerinde değişim analizi yapılarak sınıfların ilerleyen sürümlerindeki gerçek değişim maliyetleri çıkarılmıştır. Değişim eğilimli olarak belirlenen sınıflar ile sınıfların gerçek değişim maliyetleri arasındaki ilişkiler incelenerek, gerekli ölçümler ve karşılaştırmalar yapılmıştır. Doğrulama yönteminin akış şeması Şekil 4.2'de yapılmıştır.



Şekil 4.2 : Doğrulama yönteminin akış şeması.

Sonuçların doğru şekilde ölçülmesini etkileyebileyecek iki durum vardır. İlk olarak yazılımlarda çalışan geliştirici sayısına ve gelen değişim isteğine bağlı olarak farklı zamanlarda yazılımların farklı kısımları değişir. İkincisi de sınıfların metrik değerleri de geliştirme süresince değişim gösterecektir. Bu problemleri çözmek ve doğru ölçüm yapabilmek için yazılımlar, yeteri kadar revisyon ya da sürüm içerecek şekilde farklı zaman aralıklarına bölünerek incelenmiştir. Değişime eğilimli sınıflar incelenen yazılımın herbir zaman aralığındaki ilk sürümü için çıkarılırken, zaman aralığındaki tüm sürümler incelenerek sınıfların gerçek değişim maliyetleri çıkarılmıştır. Projenin sürümlerinin örtüşen zaman aralıklarına bölümünü Şekil 4.3'te gösterilmiştir.



Şekil 4.3 : Proje sürümlerinin zaman aralıklarına bölünmesi.

Değişim analizinde kullanılan yazılım revisyonları, projenin SVN deposundaki teslimlerden; sürümler ise yazılımın indirilen sürümlerinden oluşturulan sürüm veritabanı kullanılarak incelenmiştir. Yukarıda anlatılan şekilde, projeler için farklı gelişim aralıkları oluşturulmuştur.

5. DENEYSEL SONUÇLAR

Açık kaynak kodlu üç yazılım projesi bu çalışmada önerilen yöntemin doğrulanması için analiz edilmiştir. Bu bölümde analiz çalışmalarında kullanılan projeler tanıtılmıştır.

5.1 İncelenen Projeler

5.1.1 JFreeChart

JFreeChart açık kaynak kodlu güçlü bir çizim kütüphanesidir [45]. Uzun dönemli bir projedir ve incelemek için yeteri kadar sürümü vardır. Object Refinery Limited firması ve açık kaynak kod komitesi tarafından birlikte geliştirilmektedir. Projede, kullanıcılar tarafından isteklerin ve hataların raporlandığı beş geliştiricisi görülmektedir.

Bu çalışmada 0.9.0 ile 0.9.15 arasındaki ardışıl onaltı sürümü incelenmiştir. Bu sürümler [0.9.0-0.9.5], [0-9.5-0.9.10] ve [0.9.10-0.9.15] olmak üzere üç zaman aralığına bölünmüştür. Bu zaman aralıkları I1, I2 ve I3 olarak adlandırılmıştır; herbir zaman aralığında altı sürüm yer almaktadır. Projenin sürümler hakkındaki özet bilgiler Çizelge 5.1'de verilmiştir.

Çizelge 5.1 : JFreeChart projesinin sürüm bilgileri.

JFreeChart Sürümleri	Tarih	Sınıf sayısı
0.9.0	07/06/2002	196
0.9.5	06/02/2003	456
0.9.10	25/07/2003	581
0.9.15	28/11/2003	693

5.1.2 Yari

Yari Eclipse tabanlı arayüz projelerin gözlenmesi, debug edilmesi, incelenmesi için geliştirilmiş bir araçtır. Eclipse plugin projesidir [46]. Projenin tek geliştirici vardır. Deneysel çalışmalar sırasında Yari projesinin revisyon bilgileri projenin SVN deposundaki *trunk/net.sf.yari* dizininden alınmıştır. Projenin incelenecek olan ardışıl revisyonları I1 ve I2 olmak üzere iki zaman aralığına bölünmüştür. İlk zaman aralığı

[40-203] arasındaki ardışıl 48 revisyonu, ikinci zaman aralığı [203-301] arasındaki 23 revisyonu içermektedir. Projenin özet bilgisi Çizelge 5.2'de verilmiştir.

Çizelge 5.2 : Yari projesinin revisyon bilgileri.

Yari'nin Revisyon Bilgileri	Tarih	Sınıf sayısı
40	20/04/2009	38
203	14/12/2009	47
301	18/08/2010	69

5.1.3 UCDetector

UCDetector (Unnecessary Code Detector) Java tabanlı projelerde kullanılmayan kod parçalarını bulmak için geliştirilmiş bir Eclipse plugin projesidir [47]. Projenin tek geliştirici vardır.

Deneysel çalışmalar sırasında UCDetector projesinin revisyonları projenin SVN deposundaki *trunk/org.ucdetector* dizininden alınmıştır. Projenin revisyonları [246-537], [537-1170] ve [1170-1435] olmak üzere üç zaman aralığına bölünmüştür. İlk zaman aralığı, I1, 159 ardışıl revisyon; ikinci zaman aralığı, I2, 436 ardışıl revisyon ve son zaman aralığı, I3, 210 ardışıl revizyon içermektedir. Projenin revisyonları ile ilgili özet bilgiler Çizelge 5.3'te verilmiştir.

Çizelge 5.3 : UCDetector projesinin revizyon bilgileri.

UCDetector Revisyon Bilgileri	Tarih	Sınıf sayısı
246	07/09/2008	47
537	15/12/2008	62
1170	31/12/2009	58
1435	20/08/2010	77

5.2 Değişim Eğilimli Sınıfları Belirleme Yöntemleri

Değişim eğilimli sınıfların belirlenmesi için önerilen ve denenen yöntemler bu bölüm altında anlatılacaktır.

5.2.1 Rasgele seçim yöntemi

Rasgele seçim yöntemi, proje hakkında bilgi sahibi olmadan, projedeki sınıfların %10'unun öncelikli sınanması gereken sınıf olarak rasgele seçilmesidir. Bu yöntem, kullanılacak diğer yöntemlere referans oluşturması amacıyla gerçeklenmiştir. Diğer yöntemlerin başarımları değerlendirilirken, iyileştirmenin olup olmadığı bu sayede gözlemlenebilecektir.

Rasgele seçim yönteminde, en çok değişecek sınıflar rasgele olarak belirlenmiş ve bunlar gerçekten çok değişen sınıflarla karşılaştırılarak başarımı çıkarılmıştır. Seçme işlemi, herbir proje aralığı için 1000 kere tekrarlanmış, doğru olarak belirlenen sınıf sayılarının minimum, maksimum ve ortalama değerleri aşağıdaki Çizelge 5.4'de verilmiştir. Rasgele olarak değişim eğilimli sınıfları seçme işlemi herbir aralığın ilk sürümünde yapılmış; en çok değişen sınıflar ise aralığın tamamındaki ardışıl sürümler incelenerek, sınıfların değişim maliyetlerine göre oluşturulan listenin en üstteki %10'u olarak çıkarılmıştır.

Çizelge 5.4 : Rasgele seçim yöntemi sonuçları.

Proje	Aralık	Minimum	Maksimum	Ortamala
JFreeChart	I1	0	7	2.04
	I2	0	14	5.56
	I3	1	15	6.31
Yari	I1	0	3	0.44
	I2	0	3	0.57
UCDetector	I1	0	3	0.55
	I2	0	4	0.79
	I3	0	4	0.64

Elde edilen sonuçlara göre bu yöntemin, JfreeChart projesinin I1 aralığında değişim eğilimli sınıfları ortalama doğru belirleme oranı %10.2 iken, I2 aralığı için %12.09 ve I3 aralığı için %10.69'dir. Yari projesinde I1 aralığı için ortalama başarım oranı %11 ve I2 aralığı için %11.4'tür. UCDetector projesinde ise ortalama başarım oranı I1 aralığı için %11 iken, I2 aralığında %11.29 ve I3 aralığında %10.67'dir. Bu sonuçlar, diğer yöntemlerin başarım oranlarının iyileştirme sağlayıp sağlamadığını anlamak için referans olarak kullanılmıştır.

5.2.2 Karar ağaçları yöntemi

Karar ağaçları yöntemi uygulanırken, her bir projenin seçilen ilk aralığı eğitim seti olarak kullanılmıştır. Eğitim setinde kullanılan bu aralıkların ilk sürümüü için projenin metrikleri çıkarılmıştır. Daha sonra aralıkların tamamındaki ardışıl sürümler analiz edilerek sınıfların değişim maliyetleri çıkarılmış ve sınıfların değişim maliyetlerine göre sınıflara "değişim eğilimli (DE)" ve "değişim eğilimli değil (DED)" olmak üzere etiket verilmiştir. Herbir aralıktaki sınıfların değişim maliyeti en yüksek olan %10'u DE olarak işaretlenmiştir. Eğitim setinde kullanılan proje aralıkları, DE ve DED olarak etiketlenen sınıf sayıları Çizelge 5.5'te verilmiştir.

Çizelge 5.5 : Karar ağaçları yöntemi eğitim setinde kullanılan proje aralıkları.

Proje	Aralık	DE	DED
JFreeChart	I1 (0.9.0-0.9.5)	20	176
Yari	I1 (40-203)	4	34
UCDetector	I1 (245-537)	5	42

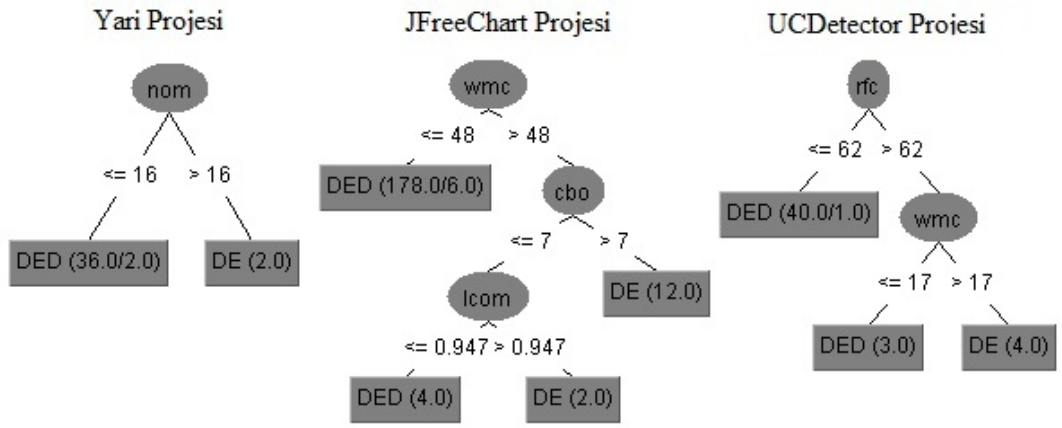
Karar ağaçları, yaygın olarak kullanılan bir sınıflandırma yöntemidir. Bu çalışmada sınıfların metrik değerlerine göre, onları DE ve DED olmak üzere iki sınıfa ayırmak için kullanılmıştır. Ağaç formunda bir yapıya sahiptir. Yaprak düğümler nesnelerin sınıflarını gösterir, ara düğümler ise o nesnelerin özniteliklerine göre karşılaştırmalarını sağlar. Eğitim aşamasında ara düğümlerin nesneleri karşılaştıracağı kurallar belirlenir. Bir nesnenin hangi sınıfa ait olduğunun belirlenmesi için kök düğümden itibaren, ara düğümlerdeki kurallar, nesnenin öz nitelikleri ile karşılaştırılır ve ağaç üzerinde uygun yol izlenerek yaprak düğümlere; dolayısı ile nesnenin hangi sınıfa ait olduğu bilgisine ulaşılır.

Bu çalışmada, karar ağaçlarının oluşturulmasında Waikato Üniversitesinde veri madenciliği uygulamarı için açık kaynaklı olarak geliştirilen WEKA [48] yazılımı tarafından desteklenen J48 algoritmasının [49] C4.5 gerçeklemesi kullanılmıştır. J48 algoritması karar ağaçlarının eğitim süreci sonunda oluşan karar ağaçları yapısı Şekil 5.1'de verilmiştir. J48 algoritması karar ağaçlarındaki aşırı-uyumu (over-fitting) engellemek için budama işlemi yapmaktadır; dolayısı ile boyut olarak daha küçük ve karmaşıklığı daha az ağaçlar oluşmaktadır [50].

Yari projesi için eğitim aşaması sonucunda oluşan karar ağacı sadece NOM metriğini kullanmaktadır. Metrik değeri 16'dan büyük olan sınıflar DE, küçük eşit olanlar ise DED olarak sınıflandırılacaktır. Diğer projeler için de üretilen ağaçların karşılaştırma değerleri Şekil 5.1'de görülebilir.

Deneysel çalışmalarda kullanılan projelerin diğer zaman aralıkları yöntemin başarımını ölçmek için kullanılmıştır. Sınama setindeki en çok değişen sınıflar algoritmanın yürütülebilmesi için DE olarak etiketlenirken, diğer sınıflar DED olarak etiketlenmiştir.

Karar ağaçları ile yapılan deneme sonuçları Çizelge 5.6'da verilmiştir. Çizelgedeki, DE ve DED sütunları sınama setindeki en çok değişen ve değişmeyen sınıfları ifade etmektedir. Her bir proje için DE ve DED kümelerindeki doğru pozitif (TP – true positive), yanlış pozitif (FP – false positive) değerleri gösterilmiştir.



Şekil 5.1 : Projelerin ilk aralıkları için eğitim süreci sonrası oluşan ağaç yapıları.

Bir kümedeki değerin hangi kümedeki ile eşleştiğinin gösterildiği karşılaştırma matrisi (confusion matrix) yerine, burada iki sınıf olduğu için çizelgede DE sınıfı olarak bulunup, gerçekten çok değişenler sınıfında yer alanlar çizelgeki DE sütununda doğru sayısı (DS), DE sınıfı olarak bulunup DED sınıfında yer alanlar ise DED sütununda yanlış sayısı (YS) olarak gösterilmiştir. Aynı durum DED sınıfı için de geçerlidir; eğer belilenen ve gerçek değerler aynı kümede ise DED sütununda DS, farklı kümedelerse DE sütununda YS olarak ifade edilmiştir.

Çizelge 5.6 : Karar ağaçları yöntemi sınama sonuçları.

Proje	Aralık	DE				DED			
		DS	YS	TP	FP	DS	YS	TP	FP
JFreeChart	I2	14	32	0.304	0.020	402	8	0.980	0.696
	I3	21	38	0.356	0.015	514	8	0.985	0.664
Yari	I2	1	4	0.200	0.024	41	1	0.976	0.800
UCDetector	I2	4	3	0.571	0.145	47	8	0.855	0.429
	I3	5	1	0.833	0.173	43	9	0.827	0.167

DE sütunundaki DS ile DED sütunundaki YS toplamı, karar ağaçları tarafından DE olarak belirlenmiş sınıf sayısını gösterirken; DED sütunundaki DS ile DE sütunundaki YS toplamı, karar ağaçları tarafından DED olarak bulunan sınıf sayısını göstermektedir. Sonuçları incelediğimizde karar ağaçları yöntemine göre Yari projesi I2 aralığı için 2 sınıf DE olarak bulunmuş ve bunların 1 tanesi en çok değişen sınıflar kümesinde yer alırken, diğer yer almamaktadır. İncelenen bu aralık için 5 adet sınıf en çok değişen sınıflar kümesinde yer alırken karar ağaçları ile bunların sadece 1 tanesi bulunabilmiştir, 4 tanesi ise DED olarak bulunmuştur. Karar ağaçları tarafından doğru olarak DE olarak bulunan sınıf sayısının, gerçekten çok değişen sınıf sayısında oranı doğru pozitif değeri (TP) ile de 0.2 olarak ifade edilmektedir.

JFreeChart projesinin sınama için incelenen ilk aralığı için, I2, değişime eğilimli olarak 22 sınıfın belirlendiği; bunların 14 tanesinin en çok değişen sınıflar arasında yer alırken 8 tanesi bu kümede yer almamaktadır. Aynı projenin I3 aralığı için 29 sınıf DE olarak belirlenmiştir ve bunların 21 tanesi en çok değişen sınıftır. Değişim eğilimli olarak belirlenen sınıflar için doğru pozitif değerleri I2 ve I3 aralıkları için sıra ile 0.304 ve 0.356'dır.

UCDetector projesinde ise değişim eğilimli olarak belirlenen sınıflar için doğru pozitif değerleri I2 ve I3 aralıkları için sıra ile 0.571 ve 0.833'dür. Bu proje için en çok değişen sınıfları bulma oranı diğer projelere göre yüksek çıkmıştır; ancak DE olarak belirlenen sınıf sayısı 12'dir, en çok değişen sınıf sayısı ise 7. Bulunan 12 sınıfın 4 tanesi en değişen sınıfları kümesinde yer almıştır.

Karar ağaçları yönteminin başarımını değerlendirirken yöntem tarafından DE olarak belirlenen sınıf sayısını da dikkate almak gerekir. Dolayısı ile UCDetector projesinde doğru olarak belirlenen DE sayısının gerçekte çok değişen sınıf sayısına oranı yüksek olmasına rağmen, algoritma tarafından DE olarak belirlenen sınıf sayısı da fazladır.

Elde edilen sonuçlar göstermektedir ki, karar ağaçları yöntemi, rasgele seçim yöntemine göre daha yüksek sonuçlar vermektedir. Ancak, bu sonuçlar bu yöntemin kullanılması için yeterli değildir. Çünkü doğru pozitif (TP) oranına göre başarımın yüksek olduğu yerlerde, yöntem tarafından değişim eğilimli olarak belirlenen sınıf sayısı en çok değişen ve DE olarak işaretlenmiş sınıf sayısından çok fazla çıkmaktadır. Ayrıca bu yöntem, sınama öncelikli sınıfların belirlenmesi için yazılımların geçmişinde yeterli sayıda sürüme ihtiyaç duymaktadır.

5.2.3 Metrik kombinasyonu yöntemi

Tez kapsamında geliştirilen metrik kombinasyonu yöntemi temel olarak iki kısımdan oluşmaktadır: sınıf metrik sıra listesinin (RL – Rank List). oluşturulması ve sınıf metrik kombinasyonu sıra listesinin (CRL – Combined Rank List) oluşturulması.

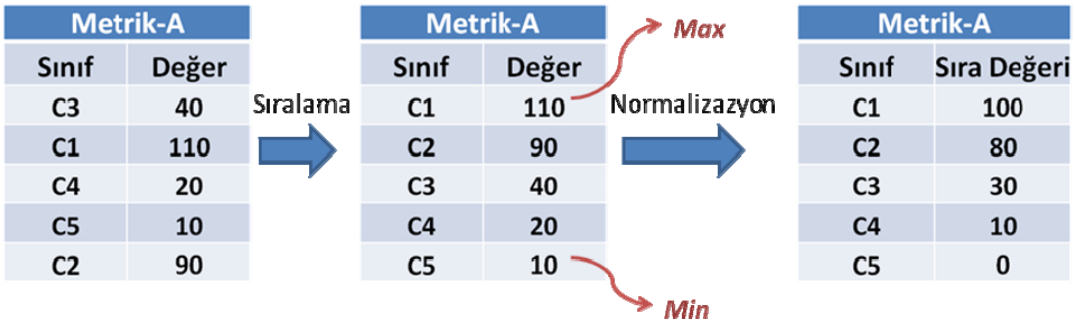
Sınıf metrik sıra listesinin oluşturulması için ilk olarak projenin seçilen bir ilk sürümü için sınıfların metrik değerleri çıkarılır. CAM metriği hariç diğer metrikler için, yüksek metrik değerleri üstte olacak şekilde sıralı sınıf listesi hazırlanır. CAM için hazırlanan sınıf listesinde, en düşük metrik değerine sahip olan sınıf en üstte

olur; çünkü diğer metriklerin aksine CAM değeri, daha iyi tasarlanmış ve iyi uyumlu sınıflar için yüksek değer alır.

Farklı metrikler farklı aralıklarda değer alırlar: CAM metriği [0,1] arasında gerçel sayı değer alırken; LOC, NOM gibi metrikler [0, +∞) arasında tam sayı değer alır. Bu nedenle, hesaplamalarda kullanılmak üzere, metrik değerleri min-max normalizasyon yöntemi kullanılarak normalize edilir. Normalize edildikten sonra metrikler 0 ile 100 arasında bir sıra numarası değerine sahip olurlar. Min-max normalizasyon yönteminde kullanılan denklem aşağıda verilmiştir (5.1).

$$V_{norm} = \left(\frac{V_{org} - Min}{Max - Min} \right) \times (NewMax - NewMin) + NewMin \qquad \textbf{(5.1)}$$
$$NewMax = 100; NewMin = 0$$

Denklemdeki *Min* ve *Max* değerleri o metrik için incelenen yazılım sürümündeki sınıfların almış olduğu en küçük ve en yüksek değerlerdir. V_{org} değeri, ilgili metriğin orjinal değerini, V_{norm} da metriğin sıra listesinde kullanılacak olan normalize edilmiş değerini göstermektedir. Örnek olarak Şekil 5.2'de beş sınıf için bulunan metrikler, bunların sıralanması ve normalize edilmiş metrik değerleri verilmiştir.

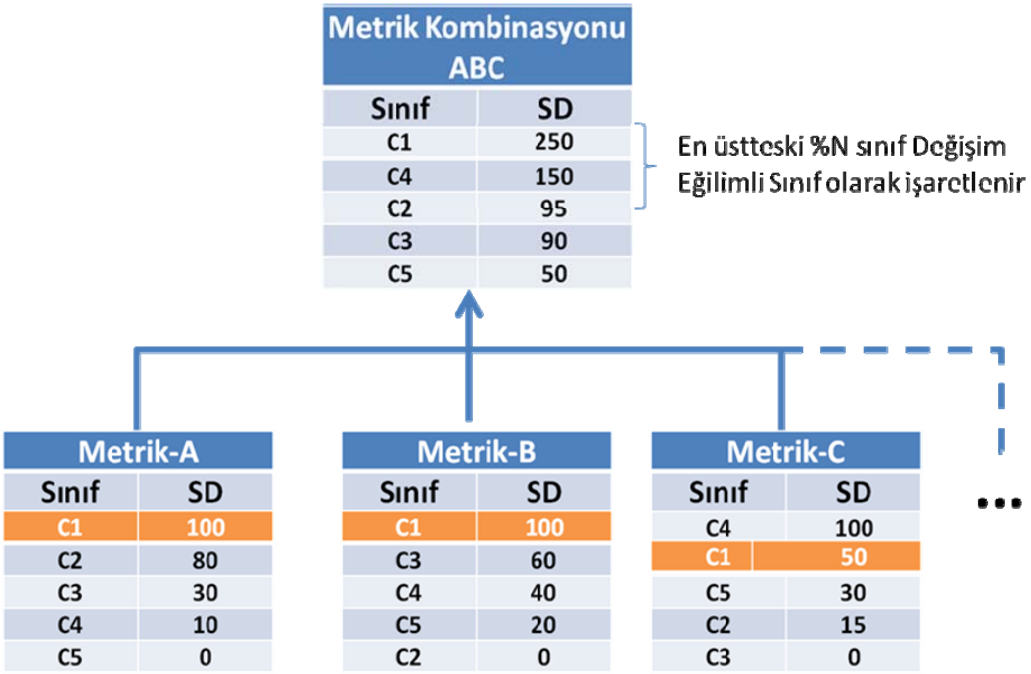


Şekil 5.2 : Sınıfların metrik değerlerinin normalizasyonu.

Metrik sıra listeleri oluşturulduktan sonra, farklı metriklere ait RL'lerin birleşimi kullanılarak sınıf metrik kombinasyonu sıra listesi oluşturulur. CRL'deki sınıfların metrik sıra değerleri, bu listeyi oluşturan RL'lerdeki metrik değerlerinin toplamıdır ve CRL'deki sınıflar en yüksek metrik sıra değerine sahip sınıf üstte olacak şekilde sıralıdır.

Metrik kombinasyonu sıra listesi oluşturulurken, CAM için daha önce bahsedilen faklı davranışı ve iyi tasarlanmış sınıflarda yüksek değere sahip olması nedeniyle, V_{norm} değeri yerine hesaplamalarda CAM için *NewMax*-V_{norm} kullanılmıştır.

Şekil 5.3'te, C1 sınıfı A ve B metrik sıralama listelerinde 100 değerine, C metrik sıralama listesinde 50 değerine sahiptir. Metrik kombinasyonu sıra listesinde ise bunların toplamı olan 250 değerine sahip olduğu görülmektedir.



Şekil 5.3 : Normalize değerler kullanılarak sınıf metrik sıralamaları.

Oluşturulan CRL'lerin en üstündeki %N'inde yer alan sınıflar "değişim eğilimli sınıflar (CPC – Change Prone Classes)" olarak belirlenir. Burada önemli olan sınıfların değişimleri ile korelasyonu yüksek olan uygun metrik kombinasyonunun bulunmasıdır. Tez kapsamında, değişim eğilimli sınıfları belirlemek için üst seviye sekiz nesneye dayalı metrik ve onların farklı kombinasyonları kullanılmıştır. Değişim eğilimli olarak belirlenen sınıflar ile projenin gerçekten değişen sınıfları arasındaki ilişkiler incelenmiştir. Kullanılan metriklerin ve metrik kombinasyonlarının başarımları, gerçek yazılım projeleri üzerinde çıkarılmıştır.

5.2.3.1 Deneysel çalışmalar

Değişim eğilimli sınıfların belirlenmesinde, Bölüm 4.2'de daha önce bahsedilen sekiz metrik kullanılmıştır. Gösterimi basitleştirmek için her metriğe kısaltma olarak bir harf atanmıştır ve atanan bu harfler Çizelge 5.7'de verilmiştir. Metrik kombinasyonlarını göstermek için harflerin birleşimi kullanılmıştır. Örneğin A, kombinasyon metrik sıra listesi oluşturulurken sadece WMC metrik sıra listesinin

kullanıldığını gösterirken; ABC ise WMC, LOC ve CAM metrik sıra listelerinin birlikte kullanıldığını gösterir.

Çizelge 5.7 : Metrik harf gösterimi.

WMC	LOC	CAM	CBO	LCOM	RFC	NOM	NOF
A	B	C	D	E	F	G	H

Bu metrikler ile her projenin incelenen zaman aralığındaki ilk sürümü için metrik kombinasyonu sıra listeleri oluşturulmuştur. Listelerin en üstünde yer alan sınıfların %10'u değişim eğilimli sınıf olarak işaretlenmiştir. En çok değişen sınıfları bulmak için her projenin incelenen zaman aralığındaki ardışıl sürümleri incelenmiştir. Bu sürümler üzerinde yapılan değişim analizi sonucunda sınıfların değişim maliyetlerine göre sıralı bir liste hazırlanmıştır ve listedeki sınıfların en üstteki %10'u en çok değişen sınıf olarak seçilmiştir.

JFreeChart

Projenin herbir zaman aralığı için CRL'ler hazırlanmış, değişim eğilimli sınıflar belirlenmiş; sınıfların değişim maliyetleri hesaplanarak en çok değişen sınıflar çıkarılmıştır. Sınıfların CRL'lerdeki metrik sıra değeri ile gerçek değişim maliyetleri arasındaki korelasyon, Pearson's korelasyon yöntemi kullanılarak hesaplanmıştır ve sonuçlar Çizelge 5.8'de verilmiştir.

Çizelge 5.8 : JFreeChart projesinin Pearson's korelasyonu

CRL	I1	I2	I3
A	0.7219	0.6121	0.7591
B	0.7626	0.5121	0.6451
C	0.4094	0.3227	0.3061
D	0.6072	0.3410	0.3405
E	0.4006	0.3776	0.3635
F	0.7018	0.5197	0.5249
G	0.7141	0.6226	0.6997
H	0.5825	0.4910	0.6501
ABCDEFGH	0.7274	0.5661	0.6165
ABFG	0.7898	0.6437	0.7201
ABG	0.7861	0.6402	0.7445
ABDG	0.8193	0.6401	0.7242
ABF	0.78	0.6032	0.6893

Elde edilen sonuçlara göre sınıfların; WMC (A), LOC (B), RFC (F) ve NOM (G) metrik sıra değerleri ile değişim maliyetleri arasındaki korelasyonun yüksek olduğu görülmektedir. Ayrıca kullanılan metriklerin olası tüm kombinasyonları için CRL'ler

hazırlanmış ve sınıfların bu listelerdeki metrik sıra değerleri ile değişim maliyetleri arasındaki korelasyon sonuçları çıkarılmıştır Korelasyon değerleri yüksek olan CRL'ler için sonuçlar Çizelge 5.8'de verilmiştir. Tek başlarına kullanıldıklarında korelasyon değerleri yüksek olan metriklerin, birleşimleri kullanılarak oluşturulan CRL'lerin korelasyon değerlerinin genel olarak daha yüksek olduğu görülmüştür.

Değişime eğilimli sınıflar ile en çok değişen sınıfların örtüşme oranı (HR – Hit Ratio) Çizelge 5.9'da verilmiştir. Korelasyonları yüksek olan WMC, LOC, RFC ve NOM metriklerinin kombinasyonları ile oluşturulan ABFG'ye göre belirlenen değişime eğilimli sınıfların örtüşme oranlarının yüksek olduğu görülmektedir. Bu oranlar I1, I2 ve I3 aralığı için %70, %64 ve %60'tır.

Çizelge 5.9 : JFreeChart projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı

	I1			I2			I3		
CPC	HR	Cost	CR	HR	Cost	CR	HR	Cost	CR
A	70	5361	48.55	56	6085	42.52	59	5306	64.93
B	65	5154	46.67	56	6427	44.91	55	4908	60.00
C	45	3323	30.00	36	4844	33.85	47	4591	56.18
D	35	3324	30.01	36	4080	28.51	33	3707	45.37
E	40	4061	36.77	44	5565	38.89	40	4170	51.03
F	45	4071	36.86	60	5730	40.04	41	4231	52.78
G	60	4953	44.85	42	5457	38.13	52	4739	58.00
H	55	4677	42.35	31	4598	32.13	43	4213	51.56
ABCDEFGH	55	4897	44.34	62	6856	47.91	59	5136	62.86
ABFG	70	5343	48.38	64	6707	46.87	60	5402	66.11
ABG	70	5235	47.40	56	6426	44.90	54	5001	61.20
ABDG	70	5265	47.68	58	6499	45.42	57	5322	65.13
ABF	70	5235	47.40	62	6475	45.24	64	5470	66.94

Değişime eğilimli sınıflar olarak belirlenen sınıfların her bir aralık için değişim maliyetleri ve o aralıktaki tüm sınıfların toplam değişim maliyetlerine oranı (CR – Cost Ratio) Çizelge 5.9'da verilmiştir. Buradaki amaç, değişim eğilimli olarak seçilen ve o versiyondaki sınıfların %10'undan oluşan sınıfların değişim maliyetlerinin, yazılımın o versiyonundaki değişikliklerin ne kadarını kapladığını görmektir. Buradan, JFreeChart projesi için elde edilen sonuçları incelediğimizde, bu sınıflarda yapılan değişimlikerin maliyetlerinin I1, I2 ve I3 aralıkları için sırasıyla, yaklaşık olarak %48, %47 ve %67 oranında olduğunu görmekteyiz. Dolayısı ile, yazılımdaki değişimlerin büyük miktarının, yazılımdaki sınıfların bazıları üzerinde gerçekleştiğini ifade edebiliriz. Ayrıca, projede, değişimle korelasyonu yüksek olan

metriklere göre elde edilen örtüşme ve değişim maliyetlerinin de yüksek olduğu olduğu görülmektedir.

Yari

Pearson korelasyonu; HR ve CPC'lerin toplam değişim maliyeti ile bu maliyetlerin projenin toplam değişim maliyetine oranları (CR) sırasıyla Çizelge 5.10 ve Çizelge 5.11'de verilmiştir.

Elde edilen sonuçlara göre LOC, CBO, RFC ve NOM metriklerine ve onların kombinasyonlarına göre hazırlanan CRL'lerin korelasyonunun yüksek olduğu görülmektedir.

Çizelge 5.10 : Yari projesinin Pearson's korelasyonu

CRL	I1	I2
A	0.5113	0.3858
B	0.7981	0.8440
C	0.4352	0.4251
D	0.9038	0.9251
E	0.2081	0.4298
F	0.7386	0.6856
G	0.7440	0.7139
H	0.3051	0.7438
ABCDEFGH	0.7197	0.7710
BDFG	0.8994	0.8861
BDG	0.8981	0.8920
BDH	0.7736	0.9008
ABF	0.8004	0.7572

WMC metriğinin korelasyonu düşük olmasına rağmen, bu metrik kullanılarak oluşturulan metrik kombinasyonu sıra listesi (A) ile belirlenen değişim eğilimli sınıfların örtüşme oranlarının, korelasyonu yüksek olan metriklerin (LOC, CBO) sonuçlarına benzer şekilde yüksek olduğu görülmektedir. RFC metriğinin değişimle korelasyonu yüksek çıkmasına rağmen, örtüşme oranları sonucu düşüktür; ancak değişim oranı sonucuna bakıldığında %51.72 olduğu ve diğer sonuçlara yakın olduğu görülmektedir. Buradan projede en çok değişen olarak bulunan %10 sınıfın, bazılarının değişim maliyetlerinin bu sınıflar arasındaki diğer sınıflardan çok yüksek olduğunu ve bu değişim maliyetleri yüksek sınıfların RFC metriği kullanılarak belirlenebildiğini söyleyebiliriz.

Bu projede, metrik kombinasyonu sıra listeleri oluşturulurken, bazı metriklerin tek başına kullanılması diğer metriklerle beraber kullanılmaları kadar yüksek sonuç

vermiştir. Projede, değişimle korelasyonu en yüksek çıkan CBO metriğine göre elde edilen örtüşme oranları ilk aralık için %50 iken, ikinci aralıkta %80'dür. BDH kombinasyon sıra listesinde de aynı sonuçlar elde edilmiştir.

Yari projesinde, değişim eğilimli olarak belirlenen sınıfların değişim maliyetlerinin, o versiyondaki toplam değişim maliyetine oranları I1 ve I2 aralıkları için sırası ile yaklaşık olarak %74 ve %65'dir. Bu da göstermektedir ki, değişimlerin büyük çoğunluğu, projedeki sınıfların %10'u üzerinde yapılmıştır.

Çizelge 5.11 : Yari projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı

	I1			I2		
CPC	HR	Cost	CR	HR	Cost	CR
A	50	345	73.09	60	523	58.04
B	50	345	73.09	60	523	58.04
C	50	329	69.70	20	421	44.50
D	50	345	73.09	80	589	65.37
E	25	209	44.30	60	522	57.94
F	50	340	72.03	40	466	51.72
G	50	329	69.70	40	463	51.39
H	25	189	40.04	40	464	51.50
ABCDEFGH	50	345	73.09	60	536	59.49
BDFG	50	349	73.94	60	546	60.60
BDG	50	346	73.30	60	536	59.49
BDH	50	345	73.09	80	589	65.37
ABF	50	345	73.09	40	509	56.50

Genel olarak değişim maliyeti oranları ve örtüşme oranları, metriklerin değişimlerle olan korelasyonu ile bağlantılıdır. Ancak, Yari projesinde bazı sınıfların değişim maliyetlerinin, en çok değişen sınıflar olarak bulunan sınıfların içinde oranları dahi değişim oranları çok yüksektir. O nedenle bazı metriklerin, için örtüşme oranlarına bakıldığında bu ilişkiye uymadıkları görülmüştür. Projenin I2 aralığında; sınıfların, değişim analizi ile elde edilen, değişim maliyetlerini incelediğimizde en çok değişen sınıfların ve maliyetlerinin:

1. *net.sf.yari.ui* paketindeki *InspectorView* sınıfı için 377
2. *net.sf.yari.ui.provider* paketindeki *PropsContentProvider* sınıfı için 80
3. *net.sf.yari.ui.util.dialogcreator* paketindeki *DialogCreator* sınıfı için 66
4. *net.sf.yari.ui.provider* paketindeki *YariEditingSupport* sınıfı için 65
5. *net.sf.yari.ui. provider* paketindeki *MemberContentProvider* sınıfı için 35

olduğu görülmektedir. Bu da tesbitimizin doğruluğunu göstermektedir. *InspectorView* sınıfı 377'lik değişim puanı ile diğer sınıflardan çok daha yüksek bir değişim maliyetine sahiptir ve yazılımın I2 aralığındaki 23 revisyonun 12'sinde değişime uğramıştır.

UCDetector

Pearson's korelasyonu sonuçları Çizelge 5.12'de verilirken değişime eğilimli olarak belirlenen sınıfların her bir aralıktaki örtüşme miktarları (HR), bu sınıfların toplam değişme maliyetleri ve projenin toplam değişme maliyetlerine oranı (CR) Çizelge 5.13'te gösterilmiştir.

Çizelge 5.12 : UCDetector projesinin Pearson's korelasyonu

CRL	I1	I2	I3
A	0.6657	0.5734	0.5407
B	0.6557	0.6805	0.6689
C	0.3278	0.3476	0.2960
D	0.6380	0.3880	0.5831
E	0.3961	0.3127	0.3992
F	0.6722	0.4962	0.4816
G	0.1688	0.0879	0.1496
H	0.4310	0.4101	0.2852
ABCDEFGH	0.6476	0.5467	0.5637
ABDF	0.7344	0.6161	0.6417
ABF	0.7412	0.6523	0.6232

Elde edilen sonuçlara göre, WMC, LOC ve RFC metriklerinin tek tek kullanılmasıyla ve kombinasyonları ile elde edilen CRL'lerin, sınıfların değişim maliyetleri ile olan korelasyonu yüksektir. Değişimle daha çok ilişkili olarak WMC, LOC onlardan sonra RFC ve CBO metriğinin geldiği görülmektedir. Bunun yanında CAM, NOM, LCOM ve NOF metriklerine göre oluşturulan CRL'lerin korelasyonu düşüktür.

Diğer projelerden faklı olarak NOM metriğinin korelasyonu bu proje için çok düşüktür. Pearson korelasyon sonuçlarını göz önüne alarak, değişim eğilimli olarak belirlenen sınıfların, en çok değişen sınıflarla olan örtüşme oranını ve değişim eğilimli sınıfların değişim maliyetini incelersek. Korelasyonu yüksek olarak bulunan metrik kombinasyon listelerinin, örtüşme ve maliyet oranlarının da yüksek olduğu; korelasyonu düşük olanlar için bu oranların da düşük olduğu görülmektedir. Örneğin, LOC için elde edilen örtüşme oranları I1, I2 ve I3 için sırası ile %40, %57 ve %83'dür. Korelasyon değeri yüksek olan dört metriğin kombinasyonu olan ABDF

için bu sonuçlar %80, %57 ve %67'dir. Diğer taraftan NOM metriğine göre elde edilen sonuçlar aynı aralıklar için %20, %0 ve %33'dür.

Değişim oranlarına bakıldığında, projedeki sınıfların %10'ununu oluşturan değişim eğilimli sınıfların, projenin o versiyonundaki toplam değişime oranı I1, I2 ve I3 için ABDF kombinasyonuna göre %50.6, %40.15 ve %31.84; ABF kombinasyonuna göre %45.65, %45.42 ve %35.09'dur. İlk iki aralık için, projedeki değişimlerin yarısına yakını sınıfların %10'u üzerinde gerçeklenirken son aralıkta bu oran daha düşüktür. Buradan, bu projedeki değişimlerin sınıflara diğer incelenen projelere göre daha dağılmış olduğu söylenebilir.

Çizelge 5.13 : UCDetector projesinin örtüşme oranı, CPC'lerin değişim maliyeti ve toplam proje maliyetine oranı

	I1			I2			I3		
CPC	HR	Cost	CR	HR	Cost	CR	HR	Cost	CR
A	40	917	42.93	57	1793	44.80	83	2590	37.79
B	40	648	30.34	57	1922	48.02	83	2590	37.79
C	20	306	14.33	43	1753	43.80	33	1467	21.40
D	40	881	41.25	43	1480	36.98	67	1988	29.00
E	20	330	15.45	29	825	20.61	50	2152	31.40
F	60	976	45.69	29	1281	32.01	50	1887	27.53
G	20	447	20.93	0	401	7.00	33	1016	14.82
H	40	632	29.59	14	1302	32.53	50	1705	24.86
ABCDEFGH	60	1023	47.89	43	1516	37.88	67	2018	29.44
ABDF	80	1081	50.60	57	1607	40.15	67	2182	31.84
ABF	60	975	45.65	57	1818	45.42	67	2405	35.09

5.2.3.2 Sonuçlarının değerlendirilmesi

Deneysel çalışmalar sırasında temel olarak üç sorunun cevabı aranmıştır.

1. Sınıfların metrik değerleri ile değişim eğilimlilikleri arasında bir ilişki var mıdır?
2. Değişim eğilimli sınıfların belirlenmesinde hangi metrikler daha anlamlıdır?
3. Değişim eğilimli sınıfların belirlenmesinde uygun metrik kombinasyonları kullanılarak daha doğru sonuçlar alınabilir mi?

Deneysel çalışmalar sırasında görülmüştür ki sınıfların metrik değerleri ile değişim eğilimlikleri arasında bir ilişki bulunmaktadır. Ancak, hangi metriklerin daha ilişkili olduğu yazılımlara göre farklılık gösterebilmektedir. Örneğin, JFreeChart projesinde CBO metriğinin korelasyonu Yari projesine göre daha küçükken, NOM metriğinin

korelasyonu daha yüksektir. Projelerde genel olarak WMC, LOC başta olmak üzere RFC'nin değişim eğilimli sınıfları belirlemede daha iyi sonuç verdiği ve ortak olarak kullanılabileceği söylenebilir. Tüm projelerde ortak olarak; CAM, LCOM metriklerinin hem korelasyon değerleri hem de örtüşme ve değişim maliyet oranı gibi diğer ölçüm sonuçları, diğer metriklerin sonuçlarından düşük çıkmıştır.

Farklı projeleri ve bunların incelenen farklı zaman aralıklarını göz önüne aldığımızda, metriklerin kombinasyonunu kullanmanın daha başarılı sonuçlar verdiğini söyleyebiliriz. Metrikler tek kullanıldıklarında, projenin farklı sürümlerinde elde edilen sonuçlar arasındaki değişim, metriklerin birlikte kullanılmasından daha fazladır. Metriklerin birlikte kullanıldığında farkı aralıklarda sonuçların değişme oranı düşük dolayısı ile daha istikrarlıdır.

Buradan çıkan en temel sonuç her projenin kendine özgü doğası olduğudur; projenin geçmiş bilgisini incelemek onu anlamakta yardımcı olabilir. Eğer, proje sürüm denetim sistemi kullanıyorsa ve yeteri kadar revisyona ya da sürüme sahipse bizim yöntemimiz ile hangi metriklerin ve kombinasyonların o proje için kullanılabilir olduğunu çıkartılabilmektedir. İncelenen projeler kapsamındaki gözlemlerimize göre; WMC, LOC ve RFC metrikleri ve onların kombinasyonları iyi sonuç vermekte ve birlikte kullanılabilmektedir.

5.2.3.3 İyileştirme çalışmaları

İyileştirme çalışmalarında; değişim eğilimli sınıflar belirlemek için metrikleri eşit ağırlıklarla almak yerine, herbir metrik için farklı ağırlık kullanılmıştır. Metrik kombinasyon sıra listeleri, metriklerin bu ağırlıkları dikkate alarak oluşturulmuştur. Deneysel çalışmalarda incelenen her proje ve zaman aralığı için farklı metrik ağırlıkları ile sonuçlar çıkarılarak; ortak olarak kullanılabilecek metrik ağırlıklarının olup olmadığı araştırılmıştır. Çalışmada, metriklere 1 ile 3 arasında ağırlıklar vererek; olası tüm kombinasyonlar için sonuçlar çıkarılmıştır. Ölçümlerde değişim eğilimli olarak belirlenen sınıflar ile en çok değişen olarak etiketlenen sınıfların örtüşme oranları çıkarılmıştır.

JFreeChart projesi ve deneysel çalışmalarda kullanılan I1, I2 ve I3 aralığı için elde edilen ölçüm sonuçları ve metrik kombinasyonu listesi oluşturulurken kullanılan metrikler ve ağırlıkları Çizelge 5.14'de verilmiştir. Bu tabloda "-" ile gösterilen metrik değerleri, o metriğin metrik kombinasyonuna katılmadığını göstermektedir.

Sadece sayısal değeri olan metrikler verilen ağırlıkları kullanılarak metrik kombinasyonu sıra listesi oluşturulmuştur. Aynı durum diğer projeler için yapılan incelemeler için de geçerlidir.

Çizelge 5.14 : JFreeChart projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%).

WMC	LOC	CAM	CBO	LCOM	RFC	NOM	NOF	I1	I2	I3
1	-	1	2	-	1	-	1	70	67	66
3	2	3	3	1	1	-	-	70	64	64
3	3	2	2	-	2	1	-	70	64	64
3	3	1	-	1	-	-	-	70	62	64
1	1	2	2	-	3	-	-	65	73	64
2	1	2	3	1	3	2	-	65	73	62
1	-	2	2	-	3	-	-	65	73	62
-	-	3	1	-	3	-	-	65	73	62
1	2	1	1	-	1	-	-	65	64	66
1	1	2	3	-	2	-	1	65	69	66
3	1	3	3	-	3	-	-	65	69	66
3	-	2	3	-	-	-	1	70	67	66

JFreeChart projesinde, farklı ağırlıklar kullanılarak elde edilen sonuçların yüksek örtüşme oranı değerlerine sahip olanları yukarıdaki çizelgede verilmiştir. Bu sonuçlar, metriklerin ağırlıksız kullanılmasına göre çok farklılık göstermemiştir. Bazı aralıklarda metriklere ağırlık vermek yüksek sonuç vermiştir. Örneğin I2 aralığı için deneysel çalışmalarda en yüksek %64 değer ABFG metrik kombinasyonu kullanılarak elde edilirken, metriklere ağırlık verilmesi ile bu oranın %73'lere kadar çıktığı görülmüştür. Ancak I1 aralığında bu ağırlıklar için daha düşük sonuç elde edilirken, I3 aralığında yakın sonuçlar elde edilmiştir. Ağırlıkların kullanıldığı kombinasyonlarda da, korelasyonu yüksek olarak bulunmuş olan WMC, LOC ve RFC metrikleri sıklıkla yer almıştır. Dikkat çeken nokta CAM metriğinin korelasyonunun diğer metriklere göre çok düşük olmasına rağmen, çizelgede verilen tüm ağırlıklı metrik kombinasyonlarında bulunmasıdır. Buradan bu uyum metriğinin sınıf değişimi ile korelasyonunun düşük olmasından dolayı; ancak ağırlığı artırıldığı zaman etkisini gösterdiğini söyleyebiliriz.

Yari projesi'nin I1 ve I2 aralığı için elde edilen ölçüm sonuçları ve metrik ağırlıkları Çizelge 5.15'te verilmiştir. Metriklere verilen ağırlıklarla oluşan metrik kombinasyonu sıra listeleri ile değişim eğilimli olarak belirlenen sınıfların, en çok değişen sınıflar içinde yer alma oranları tüm sonuçlarda I1 için %50, I2 için %80 olarak bulunmuştur. Yari projesinin incelenen çekirdek paketi, diğer projelere göre

daha küçüktür. Bu projede, bazı sınıfların değişim maliyetleri diğerlerine oranla oldukça yüksektir. Yüksek değişime uğrayan bu sınıflar bulunabilirken, onlara göre değişim maliyeti daha düşük olanlar bulunamamaktadır. Dolayısı ile bu sonuçların ağırlıksız sonuçlara göre çok bir iyileştirme sağlamadığını söyleyebiliriz.

Çizelge 5.15 : Yari projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%)

WMC	LOC	CAM	CBO	LCOM	RFC	NOM	NOF	I1	I2
1	-	1	2	-	1	-	1	50	80
1	2	2	3	-	1	-	1	50	80
-	2	2	3	1	1	-	-	50	80
-	-	2	3	-	1	-	2	50	80
3	1	2	3	-	2	-	3	50	80
3	-	2	3	-	2	-	3	50	80
2	-	2	3	-	2	-	3	50	80
-	1	2	3	1	2	-	1	50	80
-	1	2	3	1	1	-	1	50	80
-	2	2	3	-	1	-	1	50	80

Burada verilen sonuçlarda korelasyonu yüksek olan CAM, LOC, CBO ve RFC metrikleri ağırlıklar kullanılarak oluşturulan kombinasyon listelerinde çoğunlukla yer almaktadır.

UCDetector projesi'nin I1, I2 ve I3 aralığı için elde edilen ölçüm sonuçları ve metrik ağırlıkları Çizelge 5.16'da verilmiştir. UCDetector projesinde deneysel çalışmalarada ağırlıksız olarak kullanılan ABDF metrik kombinasyonuna göre elde edilen örtüşme oran sonuçları: I1, I2 ve I3 aralıkları için %80, %57 ve %67'dir. Çizelge 5.16'da verilen metriklerin ağırlıklı değerlendirilmesine göre verilen sonuçların en üstteki dört sonuç sıra ile aynı aralıklar için %100, %43 ve %67 değerini vermiştir. Buradan ağırlıkları dikkate alarak yapılan deneylerde I1 aralığında yüksek değer elde edilirken, I2'de daha düşük bir sonuç bulunduğunu I3'te ise aynı başarıma sahip olduğu görülmektedir. Ayrıca, benzeri durum, incelenen projenin kendi aralıklarında da gözükmektedir. Bir aralık için yüksek değer veren metrik ağırlıkları projenin diğer incelenen aralıklarında düşük değerler vermektedir.

İyileştirme çalışmalarında tüm projeler arasında ve projelerin incelenen aralıklarında metrik kombinasyon listeleri oluşturulurken, metriklere ağırlık vererek değişim eğilimli sınıflar belirlenmiş ve genel olarak yüksek değer veren bir ağırlık seti bulunabilir mi araştırılmıştır.

Çizelge 5.16 : UCDetector projesi için farklı metrik ağılıkları kullanılması ile elde edilen örtüşme oranları (%).

WMC	LOC	CAM	CBO	LCOM	RFC	NOM	NOF	I1	I2	I3
-	-	3	3	3	3	2	-	100	43	67
-	1	2	1	2	1	1	-	100	43	67
-	-	2	2	2	2	1	-	100	43	67
-	1	2	2	3	3	2	-	100	43	67
2	1	1	1	-	1	-	-	60	71	83
3	2	1	-	-	1	-	-	60	71	67
3	-	-	1	-	1	-	-	60	71	67
2	1	-	1	-	1	-	-	60	71	83
3	2	3	2	-	2	-	-	60	71	83
2	1	-	1	-	1	-	-	60	71	83
-	3	-	1	2	2	1	-	80	57	83
1	2	1	2	3	3	-	-	80	57	83

Elde edilen sonuçlar genel olarak değerlendirildiğinde metriklere ağırlık vererek kullanılarak oluşturulan bazı kombinasyonların, projelerin bazı aralıklarında; deneysel çalışmalardaki metriklerin ağırlıksız olarak kullanılması ile değişim eğilimli sınıfların bulunmasından daha yüksek değer verdiği görülmektedir. Buna karşılık aynı projenin diğer aralıklarında ise daha düşük sonuçlar verdiği görülmektedir. Projeler arasında değişim eğilimli sınıfları belirlemede yüksek değer veren ortak bir metrik seti olmadığı sonuçlardan görülmektedir. Aynı şekilde tek bir projenin incelenen farklı zaman aralıkları için de genel olarak daha iyi sonuç veren bir metrik ağırlık seti de bulunmamaktadır; dolayısı ile metriklere ağırlık verilerek değişim eğilimli sınıfların belirlenmesinde ağırlıklarla ilgili bir genelleştirilemenin yapılamayacağını söyleyebiliriz.

5.2.3.4 Değişim maliyeti hesaplamada farklı yaklaşımlar

Deneysel çalışmalar sırasında sınıf ekleme hariç tüm ağırlıklar dikkate alınmıştır ve bu şekilde her değişim, doğrulama yöntemindeki analizde hesaba katılmıştır. Bu kısımda ise, değişim analizinde sınıfların değişim maliyetlerini hesaplarken farklı ağırlıklar kullanılmıştır ve bazı ek değişim maliyetlerinin hesaplamaya katılması engellenmiştir. Değişimlerin farklı hesaplanması ve farklı ağırlıkların kullanılmasıyla üç grup oluşturulmuştur: Grup 1, Grup 2 ve Grup3. Grup 1'de, deneysel çalışmalar sırasında kullanılan değişim hesaplama yöntemleri ve değişim ağırlıkları kullanılmıştır. Çizelge 5.17'de değişimlerin farklı hesaplanması sırasında Grup 1, Grup 2 ve Grup 3'de kullanılan ağırlıklar sırayla KA1 , KA2 ve KA3 sütunlarında verilmiştir. Grup 1'de tüm değişimlere, sabitler hariç, 1 değişim değeri verilken,

Grup 2'de tüm değişimlere, sabitler, taşınma (M) ve isim değişimi (R) ağırlıkları hariç, 1 değişim değeri atanmıştır. Grup 3'de ise Grup 2'dekinden farklı olarak değişimlerin yazılımın tasarımı etkileme miktarları göz önüne alarak farklı ağırlıklar atanmıştır.

Çizelge 5.17 : Yapısal değişimlere farklı ağılıkların atanması.

Değişim Türü	Açıklama	KA1	KA2	KA3
MA	Bir sınıfa bir metot eklenmesi sabiti	0	0	0
MD	Bir sınıftan bir metodun silinmesi sabiti	0	0	0
FA	Bir sınıfa bir nitelik eklenmesi	1	0	0
FD	Bir sınıftan bir niteliğin silinmesi	1	1	1
CA	Bir pakete bir sınıfın eklenmesi sabiti	0	0	0
CD	Bir paketten bir sınıfın silinmesi sabiti	0	0	0
R	Sınıf veya metodun isminin değiştirilmesi	1	0	0
M	Bir sınıfın başka bir pakete taşınması	1	0	0
MC	Görünürlük değişimi (public/private/protected)	1	1	1
FTC	Nitelik tipinin değiştirilmesi	1	1	1
SCC	Sınıfın türediği sınıfların değişimi	1	1	20
IIC	Gerçeklenen arayüzün değiştirilmesi	1	1	10
MLC	Metodun yerel değişiklikleri sabiti	0	0	0
MRTC	Metodun dönüş tipinin değişmesi	1	1	3
MPC	Metodun parametre değişimi	1	1	5
MLOCC	Metodun yerel operasyon ve kontrol değişiklikleri	1	1	1
MAAF	Metoda nitelik erişimi eklenmesi	1	1	1
MDAF	Metoddan nitelik erişimi silinmesi	1	1	1
MCAF	Metodun eriştiği niteliğin değiştirilmesi	1	1	1
MACM	Metoda metot çağrısı eklenmesi	1	1	1
MDCM	Metoddan metot çağrısı silinmesi	1	1	1
MCCM	Metodun çağırdığı metodun değiştirilmesi	1	1	1

Grup 1'de değişim maliyetleri hesaplamalarında, sınıfların sahip oldukları ilk nitelik ve metot sayısından etkilenmemesi için sınıf ekleme maliyeti (CAC) ihmal edilmiştir. Grup 2 ve Grup 3'te de değişim maliyeti hesaplamaların, sınıfların sahip oldukları ilk nitelik ve metot sayısından etkilenmemesi için sınıf ekleme maliyeti (CAC) ihmal edilmiştir; ayrıca nitelik ekleme ve metot ekleme yeni bir özellik ekleme olarak düşünülerek nitelik, metot ekleme maliyetleri de (FAC, MAC) hesaba katılmamıştır. Değişim maliyeti hesaplama denklemerinde bu maliyetlerin nasıl hesaplandığı ve nerelerde kullanıldığı görülebilir (4.3).

Grup 1, Grup 2 ve Grup 3'deki maliyet hesaplama yöntemleri ve kullanılan ağırlıklara göre, JFreeChart projesinin üç zaman aralığı için sınıfların değişim

maliyetleri analiz edilmiştir. Değişim eğilimli olarak belirlenen sınıfların, en çok değişen sınıflarla örtüşme oranları hesaplanmıştır ve sonuçları Çizelge 5.18'de verilmiştir.

Çizelge 5.18 : JFreeChart projesinin farklı değişim ağırlıkları için örtüşme oranları.

	Grup 1			Grup 2			Grup 3		
CPC	I1	I2	I3	I1	I2	I3	I1	I2	I3
A	70	56	59	80	53	66	75	49	67
B	65	56	55	75	56	59	75	53	62
C	45	36	47	45	31	50	50	31	50
D	35	36	33	40	35	33	40	33	36
E	40	44	40	45	38	40	50	36	41
F	45	60	41	50	56	47	50	56	47
G	60	42	52	65	36	53	70	36	56
H	55	31	43	65	27	45	65	24	47
ABCDEFGH	55	62	59	65	58	64	70	58	66
ABFG	70	64	60	85	60	66	80	60	67
ABG	70	56	54	75	53	59	80	51	60
ABDG	70	58	57	80	58	64	75	53	66
ABF	70	62	64	75	60	71	75	60	71

Deneysel çalışmalarda ve buradaki Grup 1'de sınıf ekleme maliyeti hariç (CAC) tüm değişimler dikkate alınırken, Grup 2 ve Grup 3 için buna ek olarak metot ekleme ve nitelik ekleme maliyetleri hesaplamaya katılmamıştır. Çünkü, bu değişimlerin yeni özellik ekleme için yapılan değişimler olduğu kabulu yapılmıştır. Bu varsayım altında Grup 2 ve Grup 3 için elde edilen sonuçların Grup 1'dekine göre daha iyi sonuçlar verdiği görülmektedir. Nitelik ve metot eklemeyi, yeni özellik ekleme olarak ve diğer değişimlerin iyileştirme ve düzeltmeye yönelik çalışmalar olarak değerlendirdiğimizde; önerdiğimiz metrik kombinasyonu yönteminin, iyileştirme ve düzeltme değişimlerinin yapılması gereken sınıfları bulmada daha iyi sonuçlar verdiğini söyleyebiliriz. Grup 3'de, Grup 2'den faklı olarak değişim ağırlıklarına farklı değerler atanmıştır ve bazı değişim ağırlıklarına yüksek değerler verilmiştir; ancak elde edilen örtüşme oran sonuçları Grup 2'dekinden daha düşük çıkmıştır.

Değişim eğilimli sınıfları belirlemede kullanılan metrik kombinasyonu sıra listelerinin başarım sıralarının, Grup 1, Grup 2 ve Grup 3 için elde edilen sonuçlarda benzer olduğu görülmektedir. Örneğin bu kısımdaki gruplarda ABFG, ABF metrik kombinasyonları I1, I2 ve I3 aralıklarında genel olarak yüksek sonuç verirken, C, D ve E metriklerinin tek başına kullanıldığı kombinasyon sıra listeleri için elde edilen

sonuçlarda genel olarak daha düşüktür ve başarım sıraları deneylerin yapıldığı gruplarda değişmemektedir.

5.2.4 Yöntemlerin değerlendirilmesi

Rasgele şeçme yönteminde, yazılım hakkında bilgi sahibi olmadan en çok değişecek sınıfları belirlenmesinde sınıfların %10'u rasgele seçilerek bunların projenin incelenen sürümünde gerçekten en çok değişen sınıfları bulmadaki başarımı diğer, yöntemlere referans olması için çıkarılmıştır. Karar ağaçları yönteminde yazılımların ilk zaman aralıkları eğitim amaçlı olarak karar ağaçlarının oluşturulmasında, diğer aralıkları ise yöntemin değişim eğilimli sınıfları belirleme başarımını ölçmek için sınama amaçlı kullanılmıştır. Elde edilen sonuçların, rasgele seçme yöntemine göre daha iyi sonuçlar verdiği görülmüştür; ancak yöntemin kullanılabilmesi elde edilen başarımlar yetersiz olarak değerlendirilebilir. Yari projesi için doğru pozitif oranı 0.2 iken, JFreeChart projesinin I2 aralığı için 0.304, I3 aralığı için 0.356'dır. UCDetector projesinde ise bu oran I2 aralığı için 0.571, I3 aralığı için 0.833'dür ve bu proje için değişim eğilimli sınıfları belirleme oranı iyi olarak görülmesine rağmen değişim eğilimli olarak bulunan sınıf sayısı en çok değişen sınıf sayısından çok fazladır. Ayrıca, bu yöntemin uygulanabilmesi için eğitim aşamasında yazılımın yeterli sayıda sürümüne ihtiyaç duyulmaktadır. Metriklerin birlikte kullanılmasına dayanan önerdiğimiz yöntemin elde edilen başarımı ve incelenen projelerin farklı aralıklarındaki istikrarı ve başarımı çok daha yüksektir. Metriklere farklı katsayılar atanarak, genel bir metrik ağırlık seti bulunmaya çalışılmıştır; incelenen yazılımların farklı aralıklarda ve projelerin arasında genel olarak kullanılanılabilecek ortak bir metrik ağırlık seti bulunamamıştır.

Yazılımdaki kalitesi düşük kısımların yazılımın ilerleyen sürümlerinde değişmesi beklenmektedir; ancak gelen değişim istekleri, projede çalışan kişi sayısına bağlı olarak yazılımların farklı kısımları etkileneceğinden, yazılımlar farklı zaman aralıklarında ve yeteri kadar sürüme sahip olacak şekilde incelenmiştir. Ancak, yazılım içinde kullanılmayan ya da istikrarlı hale gelmiş kod parçalarınında kalitesi düşük olup yazılımın ilerleyen fazlarında hiç değişmeyebilir ya da çok az değişebilir. Bu kısımlar yazılımların gözden geçirilmesi yapılırken, yazılımı bilen geliştiriciler tarafından eğer, kullanılmayan kod parçaları ise yazılımdan temizlenebilir; istikrarlı hale gelmiş, yeteri kadar sınama yapılmış sınıflar ise ihmal edilebilir. Deneysel

çalışmalarda, yazılımlar belirli bir olgunluğa eriştiği sürümlerden başlanarak incelenmiştir; ancak o sürüm öncesinde yeni eklenmiş olan sınıfların metrik değerleri düşük olabilir; ancak bu sınıflar ilerleyen kısımlarda çok değişebilir. Sınama ve bakım öncelikli sınıfların belirlenmesi için yapılan analizler sonrasında; yazılımın incelenen sürümüne yakın zamanda eklenmiş olan bu sınıflar analiz tarafından bulunamamaktadır. Bu durumda olan sınıflar varsa bunlara da sınamalarda öncelik verilmesi gerekmektedir.

6. SONUÇ VE İLERİKİ ÇALIŞMALAR

Bu çalışmada, nesneye dayalı metrikleri kullanarak değişim eğilimli sınıfların yazılımların tek bir sürümüne bakarak belirlenmesi için bir yöntem önerilmiştir ve diğer yöntemlerle karşılaştırması yapılmıştır.

Metriklerin birlikte kullanılmasına dayanan önerdiğimiz yöntem temel olarak yazılımda kalite nitelikleri düşük olan kısımları, metrikleri birlikte kullanarak belirlenmesini sağlamaktadır. Bu kısımların uyumları düşük, karmaşıklıkları, boyutları ve bağımlılıkları fazladır. Bu kısımların, oralarda hata çıkma olasılığı fazla olduğundan, hata düzeltme faatliyetlerinden; ayrıca yeni bir özellik ekleme ve iyileştirme çalışmalarından fazla etkilenecekleri, dolayısı ile yazılımın ilerleyen sürümlerinde değişime uğrayacakları ileri sürülmüştür.

Bu çalışmada, nesneye dayalı yazılımlardaki değişimleri yapısal olarak incelemeye ve farklı değişimleri, etkilerine göre farklı ağırlıklandırmaya dayalı bir yazılım değişim analiz yöntemi, yazılımlardaki gerçek değişimleri bulmak için geliştirilmiştir. Ayrıca, yazılım alt bileşenlerindeki değişim miktarlarını ifade edebilmek için değişim maliyeti kavramı tanımlanmıştır. Yazılımlardaki yapısal değişimler analiz edilerek çıkarılmış ve metriklerle aralarındaki ilişkiler bulunmuştur. Önerilen yaklaşımın doğrulanması ve diğer yöntemlerin de başarımlarının ölçülebilmesi için açık kaynak kodlu üç yazılım incelenmiştir. Deneysel çalışmalarda, önerdiğimiz yöntemin elde edilen başarımı ve incelenen projelerin farklı aralıklarındaki istikrarı karar ağaçları ve rasgele seçim yöntemine göre daha yüksektir. Yazılımın o sürümündeki en çok değişen sınıfların belirlenmesindeki %80'lere varan başarım elde edilmiştir. Ayrıca sınıflardaki değişimlerin büyük bir kesimi, sınıfların düşük bir oranı üzerinde yapıldığı görülmüştür. Yari projesinin ilk zaman aralığında sınıfların %10'unu oluşturan değişim eğilimli olarak belirlenen sınıflar, o sürümdeki toplam değişimin %74'ünü kapsamaktadır; bu oran UCDetector projesinin incelenen üçüncü zaman aralığı hariç diğer projeler için %45'den fazladır.

Elde edilen sonuçlar, uygun nesneye dayalı metriklerin seçilmesi ile değişime eğilimli sınıfların başarılı bir şekilde belirlenebildiği göstermiştir. Sınama maliyetlerini ve zamanını azaltmak için projeler son fazındayken, sınama yapılmadan önce, bu yöntem kullanılarak yazılımların kritik kısımların belirlenebileceği gösterilmiştir. Bakım maliyetlerinin yüksekliği ve yazılım geliştirmenin en önemli fazı olduğu göz önünde bulundurulduğunda, önerilen yöntem ile değişim eğilimli sınıflar belirlenerek, bu sınıflara sınamalarda öncelik verilmesi ve daha fazla zaman ayrılması ile bakım maliyetleri azaltılabilir. Bununla birlikte kaynak kodun gözden geçirilmesi işlemlerinde, belirlenen bu sınıflara öncelik verilmelidir. Bu öncelikli sınıfların belirlenmesi, yazılımı inceleyen kişilerin yazılımdaki kritik sınıfları belirlemelerini sağlayarak, kodun daha kolay incelenmesine yardımcı olacaktır.

Benzer yöntem nesneye dayalı metrikler kullanılarak elde edilen listelere ters yaklaşım uygulanarak, yazılımlardaki kararlı ve tekrar kullanılabilir sınıfların belirlenmesinde kullanılabilir. Ayrıca önerilen değişim analizi ve doğrulama yöntemi, nesneye dayalı diğer veya yeni tanımlanacak olan metriklerin de değişim eğilimli sınıfların belirlenmesindeki başarımı hesaplamak için kullanılabilir. İlerleyen çalışmalarda, literatürdeki diğer metriklerin de değişimlerle olan ilişkilerinin araştırılması hedeflenmektedir. Değişim eğilimli sınıfları belirlemek için, yapay sinir ağları ve naive bayes gibi diğer makine öğrenmesi yöntemlerinin araştırılması düşünülmektedir. Geliştirilen değişim analiz yöntemi, yazılımların tarihsel bilgilerini incelemede ve elde edilen verilerle yazılım hakkında çıkarsamalarda bulunulması için kullanılması amaçlanmaktadır.

KAYNAKLAR

[1] **Turver, R.. J. and Malcolm, M.**, 1994: An Early Impact Analysis Technique for Software Maintenance. *The Journal of software Maintenance, Research and Practice*, 18(12):35-52, January-February.

[2] Tata Consultancy Services, 2007: IT Projects: Experience Certainty. Independent Market Research Report. *Dynamic Markets Limited*, August.

[3] **Erdemir, U., Tekin, U., Buzluca, B.,** 2008: Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi. *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu, YKGS 2008*. İstanbul, Türkiye.

[4] **Larman, C.,** *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development,* (3rd ed.). New Jersey: Prentice Hall. ISBN 0-13-148906-2.

[5] **Gamma, E., Helm, R., Johnson, R., and Vlissides, J.,** 1995: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, ISBN 0-201-63361-2

[6] **Chidamber, S.R., Kemerer, C.F.,** 1994: A Metrics Suite for Object- Oriented Design. *IEEE Transaction on Software Engineering*, vol. 20, no. 6, pp. 476-493.

[7] **Bansiya, J., Davis, C.,** 2002: A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transaction on Software Engineering*, vol. 28, no. 1, pp. 4-17, January.

[8] **Tiftik, N., Öztarak, H., Ercek, G., Özgün, S.,** 2007: Sistem/Yazılım Geliştirme Sürecinde Doğrulama Faaliyetleri. *III. Ulusal Yazılım Mühendisliği Sempozyumu, UYMS*. Ankara.

[9] **Boehm, B., Basili, V.,** 2001: Software Defect Reduction Top 10 List. *IEEE Computer, IEEE Computer Society*, Vol. 34, No. 1, pp. 135-137, January.

[10] **Song, O., Sheppard, M., Cartwright, M., and Mair, C.,** 2006: Software Defect Association Mining and Defect Correction Effort Prediction. *IEEE Transactions on Software Engineering*, 32(2), 69–82. doi:10.1109/TSE.2006.1599417.

[11] **Fenton, N., and Ohlsson, N.,** 2000: Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8), 797–814. doi:10.1109/32.879815.

[12] **Chaumun, M., Kabaili, H., Keller, R., and Lustman, F.,** 2002: Change Impact Model for Changeability Assessment in Object-Oriented Software Systems. *Science of Computer Programming*, 45(2–3):155–174.

[13] **Tsantalis, N., Chatzigeorgiou, A., and Stephanides, G.,** 2005: Predicting the Probability of Change in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 31(7):601–614.

[14] **Sharafat, A. R., and Tahvildari, L.,** 2007: A Probabilistic Approach to Predict Changes in Object-Oriented Software Systems. *IEEE CSMR'07*.

[15] **Xia, F.,** 2004: A Change Impact Dependency Measure for Predicting the Maintainability of Source Code. *In Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 23–24.

[16] **Hassan, A. E., Holt, R. C.,** 2004: Predicting Change Propagation in Software Systems. *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 284–293.

[17] **Lee, M., Offutt, J., Alexander, R. T.,** 2000: Algorithmic Analysis of the Impacts of Changes to Object-Oriented Software. *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, p.61, July 30-August 03.

[18] **Lee, Y., Yang, J., Chang, K. H.,** 2007: Metrics and Evolution in Open Source Software, *Seventh International Conference on Quality Software, QSIC 2007*.

[19] **Sato, D., Goldman, A., and Kon, F.,** 2007: Tracking the Evolution of Object-Oriented Quality Metrics on Agile Projects. *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming*. June 18-22. Como, Italy

[20] **Tahvildari, L., Kontogiannis, K.,** 2004: Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 16(4–5):331– 361.

[21] **Basili, V. R., Briand, L. C., Melo, W. L.,** 1996: A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, pages 751-761, October.

[22] **Gyimothy, T., Ferenc, R., Siket, I.,** 2005: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transaction on Software Engineering*, v.31 n.10, p.897-910, October.

[23] **Kastro, Y., Bener, A. B.,** 2008: A defect prediction method for software versioning. *Proc. Software Qual. J.*, 16: 543-562. DOI: 10.1007/s11219-008-9053-8

[24] **Brilliant, S., Knight J., Leveson, N.,** 1990: Analysis of Faults in an N version Software Experiment, *IEEE Transactions on Software Engineering*, Vol. 16, No.2.

[25] **Nagappan, N., Ball, T.,** 2005: Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the 27th International Conference on Software Engineering, ICSE*. May 15-21. St. Louis, MO, USA.

[26] **Bayer, D., Noack, A.,** 2005: Clustering Software Artifacts Based On Frequent Changes. *Proceedings of the 13th International Workshop on Program Comprehension*, p.259-268, May 15-16.

[27] **D'Ambros, M., Lanza, M., Lungu, M.,** 2009: Visualizing Co-Change Information with the Evolution Radar. *IEEE Transactions on Software Engineering* , Vol. 35, No. 5.

[28] **Voinea, L., Telea, A.,** 2009: Visual Quarying and Analysis of Large Software Repositories. *Empirical Software Engineering*. Volume 14, Issue 3, p.316-340. June.

[29] **Stephen, G. E., Joseph, L. S., Eric, E. S. Jr.,** 1992: Seesoft-A Tool for Visualizing Line Oriented Software Statistics, *IEEE Transactions on Software Engineering*, v.18 n.11, p.957-968, November.

[30] **Froehlich, J., Dourish, P.,** 2004: Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. *Proceedings of the International Conference on Software Engineering, ICSE.* Edinburgh, UK.

[31] **Voinea L., Telea, A., Wijk, J.J.,** 2005: CVSscan: Visualization of Code Evolution. *Proceedings of the 2005 ACM symposium on Software Visualization*, May 14-15. St. Louis, Missouri.

[32] Software Engineering Institute, CMMI, <*http://www.sei.cmu.edu/cmmi/*> alındığı tarih 28.03.2011

[33] International Organization for Standarization, "ISO Standard 9126: Software Engineering – Product Quality, parts 1, 2 and 3", International Organization for Standarization, Geneve, 2001 (part 1), 2003 (parts 2 and 3).

[34] **Beklen, A., 2008:** İleri Yazılım Mühendisliği (ISO/IEC 9126), Maltepe Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Anabilim Dalı. from http://www.alibeklen.com/wp-content/uploads/2008/11/iso_iec9126.pdf

[35] A Visual Guide to Version Control: <*http://betterexplained.com/articles/a-visual-guide-to-version-control/*>, alındığı tarih 24.04.2011

[36] **Sussman, B.C., Fitzpatrick, B.W., Pilato, C.M.,** Version Control with Subversion, from http://svnbook.red-bean.com/en/1.5/svn-book.pdf,

[37] Version Management with CVS for CVS 1.11.21, 2005. Free Software Foundation Inc., from http://ximbiot.com/cvs/manual/

[38] <*http://msdn.microsoft.com/en-us/library/3h0544kx(v=vs.80).aspx*>, alındığı tarih 28.04.2011.

[39] <*http://git-scm.com/*>, alındığı tarih 28.04.2011

[40] SWEBOK, Guide to the Software Engineering Body of Knowledge, 2004 Version

[41] Yazılım Test Süreçleri: <*http://www.bidb.itu.edu.tr/?d=1064*> alındığı tarih 16.04.2010

[42] **Erdemir, U.,** 2009: "Doktora Tez İlerleme Raporu-1", İTÜ Bilişim Enstitüsü.

[43] **McCabe, T. J**., 1976: A Complexity Measure. *IEEE Transaction on Software Engineering*, vol.2, no.4, pp.308-320.

[44] **Gyimothy, T., Ferenc, R., and Siket, I**., 2005: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction, *IEEE Transaction on Software Engineering*.

[45] <*http://www.jfree.org/jfreechart*>, alındığı tarih 12.10.2010

[46] <*http://sourcefourge.net/projects/yari*>, alındığı tarih 07.12.2010

[47] <*http://sourcefourge.net/projects/ucdetector*> alındığı tarih 05.10.2010

[48] <*www.cs.waikato.ac.nz/~ml/weka/*> alındığı tarih 04.04.2011

[49] **Quinlan, R.,** 1993: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo, CA, p.302.

[50] **Drazin, S., and Montag, M.,** Decision Tree Analysis using Weka, University of Miami. from http://www.samdrazin.com/classes/een548/project2 report.pdf

ÖZGEÇMİŞ

Ad Soyad: **Sinan Eski**

Doğum Yeri ve Tarihi: **Kastamonu - 1985**

Adres: **Kemer Mah. Şehit Cengiz Topel Cad. No:14 D:2 Bahçeköy – Sarıyer/İstanbul**

Lisans Üniversite: İstanbul Teknik Üniversitesi Bilgisayar Mühendisliği

Yayın Listesi:

- Eski, S., Buzluca, F., 2011: An Empirical Study on Metrics and Software Evolution To Predict Change Prone Classes. *The Testing: Academic and Industrial Conference - Practice and Research Techniques, TAICPART*, March 25, 2011 Berlin, Germany.