T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

MOBİL PLATFORMLARDA YAPISAL TASARIM ÖRÜNTÜLERİNİN YAZILIM KALİTESİ ÜZERİNE ETKİSİNİN İNCELENMESİ

Furkan ÖZBAY

YÜKSEK LİSANS TEZİ
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Programı

Danışman
Prof. Dr. Oya KALIPSIZ

Kasım, 2022

T.C.
YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

MOBİL PLATFORMLARDA YAPISAL TASARIM ÖRÜNTÜLERİNİN YAZILIM KALİTESİ ÜZERİNE ETKİSİNİN İNCELENMESİ

Furkan ÖZBAY tarafından hazırlanan tez çalışması 18.11.2022 tarihinde aşağıdaki jüri tarafından Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı Bilgisayar Mühendisliği Programı **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Prof. Dr. Oya KALIPSIZ
Yıldız Teknik Üniversitesi
Danışman

Jüri Üyeleri

Prof. Dr. Oya KALIPSIZ, Danışman
Yıldız Teknik Üniversitesi _____

Doç. Dr. Mehmet Sıddık AKTAŞ, Üye
Yıldız Teknik Üniversitesi _____

Prof. Dr. Selim AKYOKUŞ, Üye
İstanbul Medipol Üniversitesi _____

Danışmanım Prof. Dr. Oya KALIPSIZ sorumluluğunda tarafımca hazırlanan Mobil Platformlarda Yapısal Tasarım Örüntülerinin Yazılım Kalitesi Üzerine Etkisinin İncelenmesi başlıklı çalışmada veri toplama ve veri kullanımında gerekli yasal izinleri aldığımı, diğer kaynaklardan aldığım bilgileri ana metin ve referanslarda eksiksiz gösterdiğimi, araştırma verilerine ve sonuçlarına ilişkin çarpıtma ve/veya sahtecilik yapmadığımı, çalışmam süresince bilimsel araştırma ve etik ilkelerine uygun davrandığımı beyan ederim. Beyanımın aksinin ispatı halinde her türlü yasal sonucu kabul ederim.

Furkan ÖZBAY

İmza

Aileme

TEŞEKKÜR

Tez sürecim boyunca bana destek olan aileme ve tez danışmanım Prof. Dr. Oya KALIPSIZ'a içten duygularımla teşekkür ederim.

Furkan ÖZBAY

İÇİNDEKİLER

KISALTMA LİSTESİ

A	Abstractness
AHF	Attribute Hiding Factor
AIF	Attribute Inheritance Factor
ATFD	Access to Foreign Data
Ca	Afferent Coupling
CBO	Coupling Between Object
CC	Cyclomatic Complexity
CDISP	Coupling Dispersion
Ce	Efferent Coupling
CINT	Coupling Intensity
CK	Chidamber-Kemerer
D	Normalized Distance From the Main Sequence
DAC	Data Abstraction Coupling
DIT	Depth of Inheritance Tree
FDP	Foreign Data Providers
FURPS	Functionality Usability Reliability Performance Supportability
I	Instability
IBM	International Business Machines
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IOS	iPhone Operating System
ISO	International Organization for Standardization
LAA	Locality of Attribute Accesses

LCC	Loose Class Cohesion
LCOM	Lack of Cohesion in Methods
LND	Loop Nesting Depth
LOC	Lines of Code
MHF	Method Hiding Factor
MIF	Method Inheritance Factor
MND	Maximum Nesting Depth
MOOD	Metrics for Object Oriented Design
MPC	Message Passing Coupling
MVC	Model View Controller
MVI	Model View Intent
MVP	Model View Presenter
MVVM	Model View View-Model
NOA	Number of Attributes
NOAC	Number of Accessor Methods
NOAM	Number of Added Methods
NOAV	Number of Accessed Variables
NOC	Number of Children
NOL	Number of Loops
NOM	Model View View-Model
NOO	Number of Operations
NOOM	Number of Overridden Methods
NOPA	Number of Public Attributes
QMOOD	Quality Model for Object Oriented Design
RFC	Response For a Class
SIZE1	Number of Semicolons of Class
SIZE2	Number of Attributes and Methods of Class
SPICE	Software Process Improvement and Capability Determination
SQuaRE	Software Product Quality Requirements and Evaluation

TSC	Tight Class Cohesion
VIPER	View Interactor Presenter Entity Router
WMC	Weighted Methods per Class
WOC	Weight of Class

ŞEKİL LİSTESİ

TABLO LİSTESİ

ÖZET

Mobil Platformlarda Yapısal Tasarım Örüntülerinin Yazılım Kalitesi Üzerine Etkisinin İncelenmesi

Furkan ÖZBAY

Bilgisayar Mühendisliği Anabilim Dalı
Yüksek Lisans Tezi

Danışman: Prof. Dr. Oya KALIPSIZ

Son yıllarda mobil platformların yeteneklerinin ve popülerliklerinin artmasıyla mobil uygulama geliştirmeye verilen önem ve ayrılan bütçe artmıştır. Mobil uygulamalar da diğer yazılımlar gibi doğası gereği değişime ve gelişime açıktır ve bu nedenle mobil uygulamalar yaşam döngüleri boyunca birçok değişikliğe uğramaktadır. Bu değişimlere; yeni özellikler, hata düzeltmeleri, performans iyileştirmeleri gibi örnekler verilebilir. Yapılan değişikliklere yazılımın hızlı ve kolay bir şekilde uyum sağlayabilmesi için yazılımın kaliteli olması gerekmektedir. Yazılım kalitesinden yoksun yazılımların hem şirketler hem de ülke ekonomileri üzerinde yıkıcı etkileri olabilmektedir. Yazılımın kalitesinin bu kadar önemli olması; hazırlanan yazılımların belirli bir kalitenin üzerinde olması için büyük ve titiz çabalar gösterilmesi gerektiğini göstermiştir.

Yazılım kalitesi iki ana başlık altında incelenebilir. Bunlardan birincisi geliştirme evresi kalitesi, ikincisi ise operasyonel evredeki kalitedir. Yazılım kalitesini iyi seviyelerde tutabilmek için özellikle geliştirme evresindeki kaliteyi sağlamak gerekir. Bunu sağlamak için ilk aşama olarak yazılım geliştirme sürecini iyileştirmek gerekir. Geliştirme sürecinin önemli faktörlerinden biri olan tasarım aşamasının ana bölümü yazılımın tasarımına karar vermektir. Bu noktada tasarıma büyük etki yapan yapısal tasarım kalıplarının seçimi önem arz etmektedir. İkinci aşama olarak ise yazılım kodunun kalitesinin, yazılım kod kalite metrikleri veya nitelikleri gibi somut, bağımsız, tekrarlanabilir ve güvenilir ölçüm yöntemleri ile ölçülmesi, değerlendirilmesi ve gerekli aksiyonların alınması gerekir.

Bu tez çalışmasında geliştirme gereksinimleri alt başlığında; yazılım kod kalitesi ve yazılım kod kalite metrikleri, Android platformunda kullanılan yapısal tasarım örüntüleri araştırılmış ve yazılım kod kalite metrikleri kullanılarak bir Android uygulamasının farklı iki yapısal tasarım örüntüsü ile yazılmış versiyonlarının yazılım kod kalitesinin farkı gözlemlenmiştir.

Anahtar Kelimeler: Yazılım kod kalitesi, kod kalite metrikleri, yapısal tasarım örüntüleri

ABSTRACT

Investigation of the Effect of Architectural Design Patterns on Software Quality in Mobile Platforms

Furkan ÖZBAY

Department of Computer Engineering
Master of Science Thesis

Supervisor: Prof. Dr. Oya KALIPSIZ

In recent years, with the increase in the capabilities and popularity of mobile platforms, the importance given to mobile application development and the allocated budget have increased. Mobile applications, like other software, are open to change and development by nature, and therefore mobile applications undergo many changes throughout their lifecycle. Examples of these changes include new functionality, bug fixes, and performance enhancements. In order for the software to adapt to the changes made quickly and easily, the software must be of high quality. Software that lacks software quality can have devastating effects on both companies and national economies. The software's quality is crucial, and it has been demonstrated that in order for it to be above a specific standard, tremendous and careful efforts must be taken.

Software quality can be examined under two main headings. The first is the quality of development requirements, and the second is the quality of operational requirements. In order to keep the software quality at good levels, it is necessary to ensure the quality especially in the development phase. In order to achieve this, it is necessary to improve the software development process as a first step. The main part of the design phase, which is one of the important factors of the development process, is to decide on the design of the software. At this point, the selection of structural design patterns, which have a great impact on the design, is important. As a second step, the quality of the software code should be measured and evaluated with concrete, independent, repeatable and reliable measurement methods such as software code quality metrics or attributes, and necessary actions should be taken.

In this thesis, under the development requirements heading; software code quality and software code quality metrics, architectural design patterns used in the Android platform were investigated and the difference in software code quality of versions of an Android application written with two different structural design patterns was observed using software code quality metrics.

Keywords: Software code quality, code quality metrics, architectural design patterns

1
GİRİŞ

1.1 Literatür Özeti

Her geçen gün ilerleyen mobil teknolojiler sonucunda mobil platformlar ve uygulamalarının önemi gün geçtikçe artmaktadır. Bunun sonucunda yazılım camiası yapılan yazılımların kaliteli, maliyetlerin düşük ve bakımlarının kolay yapılması için önemli ölçüde çaba sarf eder. Yazılım kalitesinin eksik olduğu yazılımlar araştırmaya [1] göre 2020 yılında düşük kaliteli yazılımların Amerika Birleşik Devletleri ekonomisine toplam maliyeti yaklaşık 2,08 trilyon dolardır. Aynı çalışmaya göre 2020 yılında Amerika Birleşik Devletleri'nde 1,6 trilyon doların bilişim sistemlerine ve bunun da yaklaşık %75'inin, yaklaşık 1,2 trilyon dolar, de eski sistemlerin bakımına harcandığı bulunmuştur. Bu miktarın da üçte ikisinin israf olarak kabul edilmiş ve ortaya yaklaşık 800 milyar dolar gibi bir israf çıkmıştır. Yine aynı çalışmaya göre 1,31 trilyon dolarlık da teknik borçlanma maliyeti hesaplanmıştır. Fakat bu gelecek borçlara dahil olduğu için hesaplamaya katılmamıştır. Bununla beraber kalitesi düşük yazılımların maddi zararlardan hariç itibar kaybı gibi manevi zararları da olabilmektedir. Bir firmanın düşük kaliteli bir yazılımı piyasaya sürmesi ve kullanıcıları mağdur etmesi hem müşterilerin hem de yatırımcıların firmaya olan güvenini azaltmaktadır. Yazılım kalitesinin bu denli önemli olduğu ve gitgide de artan öneminden dolayı hazırlanan yazılımların belirli bir kalitenin üzerinde olması, hataların projenin geliştirilme aşamasında bulunmasına ve çözülmesine verilen önem haliyle artmıştır.

Yazılımların kod kalitesine mühendis gözüyle bakabilmek için ölçüm araçlarına ve karşılaştırma yapabilmek içinse belirlenen metriklere gereksinim vardır. Bu metrik gereksinimleri için yazılım metrikleri kullanılır [2]. Bu metriklerin yazılım kalitesi ile ilişkisi, önemi ve yazılımların kalitesinin daha az maliyetli ve tutarlı bir şekilde ölçülmesi gibi konular üzerinde yapılan çalışma ortaya koymuştur [3]. Yazılım kalite modelleri kaliteyi etkileyen etmenleri belirlemek ve bu etmenleri ölçen bir dizi metrik oluşturmak böylece yazılımın kalitesini değerlendirmeye yardımcı olacak verileri toplamak için oluşturulan modellerdir. Yazılım kalite modellerine örnek

olarak ISO/IEC 9126-1 [4] kalite modeli örnek verilebilir. Bu yazılım kalitesi modeli uluslararası bir standart olup McCall [5] ve Boehm [6] modellerini temel almıştır [7].

Bir yazılımın yanlış veya optimum olmayan bir şekilde yazılması; o yazılımın geliştirme safhasında veya daha ileri bir zamanda hataların bulunmasına, kaynak kodu yazımının uzamasına ve haliyle de projenin uzamasına böylece zaman maliyetinin artmasına, kaynak kodunun bakım yapılabilirliğinin azalmasına ve son olarak da anlaşılabilirliğin azalmasına neden olabilir. Örneğin, anlaşılabilirliğin azalması proje açısından büyük bir olumsuz etkisi vardır. Yazılım kaynak kodunun anlaşılabilirliği az olduğunda yanlış özellik eklemelerine, yanlış hata düzeltmelerine ve projeye yeni katılan yazılım geliştiricilerinin projeyi geç veya yanlış anlamasına kadar varan birçok kötü sonuçlara yol açabilir. Bu tarz sorunları önleyebilmek ve yazılımın gelen taleplere ve beklentilere noksansız ve hızlıca cevap verebilmesi, kaliteli bir yazılım için vazgeçilmez bir öğedir. Yazılımların hayat döngüsüne baktığımızda taleplerin ve hata düzeltmelerinin sadece geliştirme safhasında değil, ileriki safhalarda da yeri vardır ve bu neredeyse geliştirme esnasındaki önemle eş değerdir. Bu nedenle, yazılımların geliştirilme sürecinde başlangıçtan itibaren atılacak adımların hali hazırda bulunan istek ve taleplerin karşılanması dışında daha sonraki safhalarda istenilebilecek değişikliklerin yazılımı nasıl evrimleştireceği öngörülüp ince elenip sık dokunulması gerekir. Tüm bunların sağlanabilmesi amacıyla yazılım modüllerinin aynı proje içinde veya benzer fonksiyonları sağlayan farklı projelerde tekrar kullanılabilmesi, kolayca muadilleriyle değiştirilebilmesi gibi kolaylıklar sağlaması beklenir.

Yazılım geliştirme süreçlerinde tasarım örüntülerinin kullanılması geliştiricilerin hem yükünü azaltmış olup hem de geliştiriciler, yanlış mimari kurgulama sonucu doğabilecek sorunlardan ve maliyetlerden de kaçınmış olurlar. Günümüzde; aktif olarak birçok tasarım örüntüsü kullanılmaktadır. Bu tasarım kalıplarının Android platformunda sıklıkla kullanılanların ve özellikle bu tez çalışmasının da konusu olan yapısal veya mimari tasarım örüntülerinin çeşitli açılardan karşılaştırılmaları hakkında karşılaştırma ve yazılım kalitesine olan etkilerine ilişkin birçok çalışma vardır. Bu çalışmalara örnek olarak [8–24] gibi çalışmalar örnek gösterilebilir. Bu çalışmalara ek olarak; nesneye yönelik yazılımlarda tasarım kalıplarının yazılım kalitesi ve bakım yapılabilirliği inceleyen çalışmada [25] var olan bir yazılıma farklı tasarım kalıpları uygulanarak elde edilen sonuçlar incelenmiş, yazılımların evrimleşme sürecinde tasarım örüntülerinin yazılım kalitesi üzerindeki etkilerini inceleyen çalışmada [26] yazılımların versiyonları arasındaki kullanılan tasarım kalıplarını bularak yazılıma olan etkisini incelenmiş, mobil platformlarda kaliteli kod geliştirilmesi ve maliyetin azaltılması için çalışmada [27] da IOS platformu için tasarım kalıpları yazılım kalitesi başlığında incelenmiş ve maliyet azaltımı için önerilerde bulunup son olarak da literatüre ek bir yazılım tasarım örüntüsü kazandırmıştır.

1.2 Tezin Amacı

Yazılımların günden güne değişen ve artan sorumluluklarını yerine getirirken hızlı, az maliyet isteyen, güvenli, stabil ve hatasız olması gibi gereksinimleri de karşılaması gerekir. Bu tarz gereksinimleri karşılayamayan yazılımlarda, yazılımların geliştirme süreci uzayabilir ve geliştirici ücreti gibi maliyet yüklerinin yanında projenin iptaline varan yükler getirebilir. Geliştirme süreci bittikten sonra başlayan bakım yapılması evresinde de hem geliştirici için hem de iş veren için zaman kaybı, maliyet yükü gibi olumsuzluklar oluşturabilir. Bir diğer önemli nokta ise yazılımdaki hatalardır, bu hatalar sonucu oluşabilecek büyük kayıplar meydana gelebilir. Bu hatalar yazılımın karmaşık olması ve bu nedenle yanlış kod parçaları yazımı, yanlış tasarım sonucu öngörülemeyen hatalı kod parçası çağırımları gibi nedenlerden meydana gelebilir ve bu hatalar bir fabrikada yanlış üretim, yanlış hesaplama ile zarar etme gibi görece telafi edilebilecek durumları meydana getirebilirken, askeri alanda ise yanlış hesaplama sonucu insan hayatına mâl olabilecek ve telafi edilemeyecek sonuçlara kadar varan istenmeyen durumları meydana getirebilir. Bu tarz istenmeyen durumları önleyebilmek adına yazılımların geliştirilme evresi çok önemlidir. Çünkü geliştirilme evresinde bulunan tasarım, hesaplama gibi her türlü hatalar kolaylıkla düzeltilebilir ve yukarıda belirtilen kötü senaryolara sebebiyet vermez. Yazılım geliştirme evresindeki amaç yazılımın kalitesini artırmak böylece daha kaliteli bir yazılım ürünü ortaya koymaktır.

Bu tezde, Android mobil platformundaki popüler iki yapısal tasarım örüntüsü ile yazılmış iki yazılım proje kodu metrik tabanlı yöntemle incelenerek yapısal tasarım örüntülerinin yazılım kalitesine etkisi incelenmiş, metrik sonuçları paylaşılmış ve yorumlanmıştır.

1.3 Hipotez

Tez kapsamında, yapısal tasarım örüntülerinin yazılım kod kalitesini; proje, paket, sınıf ve metot seviyesindeki metrikler bakımından etkilediği hipotezi ortaya atılmaktadır.

2
YAZILIM KALİTESİ

Kalite hayatımızın her yerinde karşımıza çıkabilecek bir olgudur. Birçok tanımı yapılabilse de genel olarak herkesin aklındaki kalite tanımı birbirine yakındır. Bir üründen, hizmetten veya servisten beklentimizin karşılanıp karşılanamaması veya ne kadar karşılandığı aslında bir kalite göstergesidir. Yazılımlar da bir ürün, servis olduğundan yazılımların da bir kalitesi vardır ve bu kalite, beklentileri ne kadar karşıladığının göstergesidir. Burada özellikle göstergesi ifadesi kullanılmıştır çünkü üründen beklenilen beklentiler olmaması ve ürünün bir özellik sunmaması veya sunduğu özellikleri başarılı bir biçimde yapmaması o ürünü kaliteli bir ürün sınıfına sokmaz.

Kalitenin tanımı veya göstergesi yukarıdaki gibi başka birçok şekilde yapılmaktadır. Örneğin Philip Crosby'e göre kalite, yönetim tarafından belirlenen belirli gereksinimlere uyum olarak ifade edilirken Edward Demings'e göre iyi kalite, müşteriye uygun bir kalite standardı ile öngörülebilir bir tekdüzelik ve güvenilirlik derecesi anlamına gelir. Bir başka yaygın kabul gören tanım ise; kalite, performansın beklentileri karşılama derecesidir. American Society for Quality kurumuna göre ise; kalite, özellikle gereksinimlere uygunluk ve müşterileri tatmin etme derecesine göre, mal ve hizmetlerde mükemmelliği ifade eder [28].

Bir yazılım bütün kalitesinden söz edildiğinde hem geliştirme gereksinimlerini hem de operasyonel gereksinimleri karşılayabilme kabiliyetlerinden bahsetmek gerekir çünkü bir yazılım hem geliştirme gereksinimlerini hem de operasyonel gereksinimleri karşılayabildiği kadar kaliteli bir yazılımdır. Kod kalitesi bakımından yüksek olan bir yazılımın kullanılabilirliği düşük olabilir.

Yazılımların kalitelerini belirtmek için belirli standartlar, kalite modelleri, kalite metrikleri ve kalite nitelikleri vardır. Böylece uluslarası ortak kullanılabilen herkes tarafından kabul gören kavramlar sayesinde kalite somut bir şekilde ifade edilebilir.

2.1 Yazılım Kalite Standartları ve Modelleri

Kalite herkes için benzer anlamı taşısa da farklı koşullar altında bu anlam daha fazla farklılaşabilmektedir. Bu nedenle yazılımların kalitesinin ortak bir standartta buluşturmak adına bazı organizasyonlar tarafından yazılım kalite standartları oluşturulmuştur. Bu standartlar, kalite modelleri, kalite nitelikleri ve kalite metrikleri içerebilir.

Yazılım kalite modelleri ise kaliteyi etkileyen faktörleri bulmak ve bu faktörleri ölçen metrikleri oluşturmak böylece yazılımların kalitesini belirlemeye yarayan modellerdir [29]. Bu modellere örnek olarak ISO/IEC 9126 kalite standardında [30] tanımlanan ISO/IEC 9126-1 [4] kalite modeli örnek verilebilir. Bu kalite modeli uluslararası bir yazılım kalite modeli olup McCall ve Boehm yazılım kalite modellerini baz almıştır [7]

Aşağıda çokça bilinen bazı standartlar ve kalite modelleri bölümler halinde açıklanmıştır.

2.1.1 ISO 9126 Standartı

ISO/IEC 9126 kalite standardı [30] diğer kalite standartları gibi kalitedeki farklılaşmayı ve insan etkisini azaltmak amacıyla oluşturulmuştur. Bu standart ilk olarak 1991 yılında duyurulup 2001 yılında güncellenip son olarak 2011 yılında ISO/IEC 25010 standartıyla [31] değiştirilmiştir. İçerdiği kalite modeli 6 kalite ve 26 alt kalite faktörü içermektedir. ISO/IEC 9126 standartı 4 parçadan oluşmaktadır:

- Kalite Modeli
- Harici Metrikler
- Dahili Metrikler
- Kullanım Kalitesi Metrikleri

Kalite modeli bölümünde, kaliteye yönelik çeşitli yaklaşımlar arasındaki ilişkileri tanımlayan ve yazılım ürünlerinin kalite özelliklerini ve alt özelliklerini ifade eden kalite modeli çerçevesi tanımlanır.

Harici metrikler bölümünde, kalite modeli bölümünde bahsedilen harici metriklerin tanımlanması yapılmaktadır. Ayrıca bu kalite metriklerinin nasıl uygulanacağına dair bir açıklama, her bir alt karakter için temel bir metrik seti ve yazılım ürünü yaşam döngüsü sırasında metriklerin nasıl uygulanacağına dair bir örnek içerir.

Dahili metrikler bölümünde, kalite modeli bölümünde bahsedilen dahili metrikler ve alt karakteristiklerin açıklaması yapılmaktadır.

Kullanım kalitesi metrikler bölümünde kullanıcı için toplu kalite etkilerini ölçen metriklerin tanımlaması yapılmaktadır. Kullanım kalitesi metrikleri nihai ürünün gerçek kullanım şartlarında kullanıldığında ölçülebilir.

ISO/IEC 9126-1 kalite modeli; işlevsellik (functionality), güvenilirlik (reliability), kullanılabilirlik (usability), verimlilik (efficiency), taşınabilirlik (portability) ve bakım yapılabilirlik veya sürdürülebilirlik (maintainability) olmak üzere 6 ana kalite faktörleri veya kalite nitelikleri ve alt başlıklarını tanımlar. Bu altı kalite faktörleri ve alt başlıkları aşağıda tanımlanmıştır.

2.1.1.1 İşlevsellik

İşlevsellik, bir yazılımın yerine getirmesi beklenilen tüm fonksiyonları yerine getirebilme becerisidir. Örneğin bir veritabanı uygulaması verileri kayıt etme, silme ve güncelleme işlemlerini yapabileceğini söylüyorsa bu işlemleri gerçekten sağlamalıdır. Bu kalite faktörünün uygunluk (suitability), doğruluk (accuracy), birlikte çalışabilirlik (interoperability), güvenlik (security), işlevsel uyumluluk (functional compilance) olmak üzere 5 alt kalite faktörü vardır.

- Uygunluk: Yazılım işlevlerinin spesifikasyona uygunluğunu belirtir.
- Doğruluk: Yazılımın doğru şekilde işlediğini ve doğru sonuçlar verdiğini ifade eder.
- Birlikte Çalışabilirlik: Yazılımın bileşeninlerinin tekil ya da çoğul olarak diğer bileşenler veya sistemlerle etkileşime girme yeteneğini belirtir.
- Güvenlik: Yazılımın yetkisiz erişimlere karşı olan tutumunu belirtir.
- İşlevsel Uyumluluk: Yazılımın işlevsellik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.1.2 Güvenilirlik

Güvenilirlik, bir yazılımın belirli koşullar altında belirli bir süre kadar görevini icra edebilme kapasitesini belirtir. Örneğin yazılımın ihtiyaç duyduğu kaynağa erişimi belirli süre kısıtlandıysa veya kaynak belirli bir süre erişilemez duruma gelip daha sonra tekrar erişilebilir hale gelince yazılımın işleyişine devam etmesi beklenir. Güvenilirlik kalite niteliğinin olgunluk (maturity), hata toleransı (fault tolerance),

kurtarılabilirlik (recoverability), güvenilirlik uyumluluğu (reliability compilance) olmak üzere 4 alt kalite niteliği vardır.

- Olgunluk: Bir yazılımın ne denli hata verdiğini belirtir.
- Hata Toleransı: Yazılımın ortaya çıkan bir hatadan sonra işleyişine devam edebilmesidir.
- Kurtarılabilirlik: Yazılımın oluşan hatadan sonra kendisini normal akışına devam edebilecek hale getirmesidir.
- Güvenilirlik Uyumluluğu: Yazılımın güvenilirlik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.1.3 Kullanılabilirlik

Kullanılabilirlik kalite niteliği bir yazılımın kullanıcıları tarafından öğrenilmesi, anlaşılması ve kullanılmasının kolaylığını gösterir. Kullanılabilirlik kalite niteliğinin anlaşılabilirlik (understandability), öğrenilebilirlik (learnability), çalışabilirlik (operability), çekicilik (attractiveness) ve uyumluluk (compilance) olmak üzere 5 alt kalite niteliği bulunmaktadır.

- Anlaşılabilirlik: Bir yazılımın işlevlerinin bölümler halinde veya bütün olarak kullanıcılar tarafından kavranabilirlik derecesini belirtir.
- Öğrenilebilirlik: Bir yazılımın işlevlerinin ne kadar öğrenilebilir olduğunu ifade eder.
- Çalışabilirlik: Bir yazılımın kullanıcıları tarafından belirli bir ortamda kolayca çalıştırılabilme derecesini belirtir [7].
- Çekicilik: Bir yazılımın kullanıcıyı cezbedilme derecesini belirtir.
- Kullanılabilirlik Uyumluluğu: Yazılımın kullanılabilirlik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.1.4 Verimlilik

Verimlilik kalite niteliği bir yazılımın verilen ortamdaki ihtiyaç duyup kullandığı kaynakları efektik olarak kullanma derecesidir. Nasıl satın alınan ürünlerin fiyat/performans oranı varsa aynı şekilde yazılımların da kaynak/performans oranı vardır. Örneğin bir mesajlaşma uygulaması depolama, ağ ve zaman gibi

kaynakları kullanma miktarı fazla olmasına rağmen yeterli veya beklenen performansı veremiyorsa o uygulamanın verimlilik açısından düşük olduğu söylenebilir. Verimlilik kalite faktörünün zamansal davranış (time behaviour), kaynak kullanımı (resource utilization) ve verimlilik uyumluluğu (efficiency compilance) olmak üzere 3 alt kalite faktörü vardır.

- Zamansal Davranış: Yazılımın yapılan işleme verdiği çalıştırma ve yanıt süresi ayrıca çıktı oranını ifade eder.
- Kaynak Kullanımı: Yazılımın çalışmak ve fonksiyonlarını yerine getirmek için kullandığı kaynak miktarını belirtir.
- Verimlilik Uyumluluğu: Yazılımın verimlilik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.1.5 Bakım Yapılabilirlik

Yazılımların doğası gereği özelliklerinin değişimi, artması veya kaldırılması gibi işlev yaşam döngüleri vardır. Bu değişimlerin sebepleri olarak ise aşağıdaki nedenler sıralanabilir.

- Bazı işlevler, bağımlı oldukları kaynaklara göre zaman içinde çalışmaz hale gelmesi veya müşteriler tarafından artık farklı bir biçimde yapılması istenmesi.
- Maddi kaygıların neticesiyle daha fazla özellik eklenerek yeni müşterilere ulaşmak veya var olan müşteriler tarafından yeni özellikler istenmesi.
- Var olan işlevlerin artık yerine getirilemez hale gelmesi nedeniyle işlevin kaldırılması.
- Yazılımın farklı ortamlarda çalışmasının desteklenmesi.

Bakım yapılabilirlik kalite niteliği veya diğer bir deyişle sürdürülebilirlik kalite niteliği, bir yazılımın içerdiği hataların bulunması, bulunan hataların çözümlenmesi veya yazılımın özelliklerini artırmak ya da farklı ortamlara uyum sağlamak için modifiye edilmesindeki kolaylık derecesi ile alakalıdır. Bakım yapılabilirlik kalite niteliğinin analiz edilebilirlik (analyzability), değiştirilebilirlik (changeability), kararlılık (stability), test edilebilirlik (testability) ve sürdürülebilirlik uyumluluğu (maintainability compilance) olmak üzere 5 alt kalite faktörü vardır. Bu kalite faktörleri aşağıda listelenmiştir.

- Analiz Edilebilirlik: Yazılımdaki hataların sebebini bulunabilme kolaylığının derecesidir.
- Değiştirilebilirlik: Yazılımda yapılmak istenilen değişikliklerin yapılmasındaki kolaylık derecesidir.
- Kararlılık: Yazılımın yapılan değişikliklerden dolayı olumsuz olarak etiklenmesinin duyarlılık derecesidir.
- Test Edilebilirlik: Yazılımın test edilebilme derecesidir. Bir yazılım, onu oluşturan parçaları kendi başlarına test edilebiliyorsa, yapılan testler dizgesel olarak belirtilip tekrarlanabiliyorsa ve yapılan testlerin çıktıları gözlemlenebiliyorsa test edilebilen bir yazılım olarak tanımlanır.
- Bakım Yapılabilirlik Uyumluluğu: Yazılımın bakım yapılabilirlik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.1.6 Taşınabilirlik

Yazılımlar zamanla farklı ortamlara veya farklı platformlarda çalışmak durumunda kalabilir. Bu nedenle yazılımların farklı sistemlere uyarlanabilir olması önemlidir.

Taşınabilirlik kalite faktörü yazılımların farklı ortamlara veya platformlara geçişinin kolaylık derecesini belirtir. Taşınabilirlik kalite faktörünün; uyarlanabilirlik (adaptability), kurulabilirlik (installability), birlikte var olabilme (co-existence), değiştirilebilirlik (changeability), taşınabilirlik uyumluluğu (portability compilance) olmak üzere 5 alt kalite faktörü vardır. Bu kalite faktörleri aşağıda listelenmiştir.

- Uyarlanabilirlik: Yazılımın farklı sistemlerin gereksinimlerini karşılayabilme derecesidir.
- Kurulabilirlik: Yazılımın farklı ortama veya platforma kurulma kolaylığı becerisidir.
- Birlikte Var Olabilme: Yazılımın, çalıştığı ortamdaki başka yazılımlar ile çalışabilme becersidir.
- Değiştirilebilirlik: Belirtilen yazılımın belirtilen ortamda başka bir yazılım ile değiştirilebilme becerisidir. Örneğin yazılımın yeni versiyonu ile eskisinin yer değiştirmesi.
- Taşınabilirlik Uyumluluğu: Yazılımın taşınabilirlik açısından yönergelere veya düzenlemelere uyumlu olmasını belirtir.

2.1.2 ISO/IEC 25010 Standartı

ISO/IEC 9126 kalite standardı 2011 yılında ISO/IEC 25010 kalite standardıyla değiştirilmiştir ve SQuaRE (Software product Quality Requirements and Evaluation) olarak da bilinir. ISO/IEC 9126 standartının modernize edilmiş bir halidir ve içerdiği kalite modeline ekstra olarak güvenlik ve uygunluk kalite faktörleri eklenmiştir. Kalite modelinde toplam 8 kalite ve 31 alt kalite faktörü vardır.

2.1.3 ISO/IEC 15504 Standartı

ISO/IEC 15504 standartı yazılımların süreçlerini iyileştirmeyi ve yetenek düzeylerinin belirlenmesini hedefler. Diğer bir adı olan SPICE (Software Process Improvement and Capability dEtermination) olarak da bilinir. İyi yazılım mühendisliği için gereken ana gayeleri tanımlar. Yazılımı elde etmekten başlayıp sağlama, geliştirme, işletim ve destek süreçlerine kadar olan süreçlerin etkinliğini hedefler ve her bir süreç için bir yetenek seviyesi tanımlar. Bu yetenek seviyeleri ve açıklamaları Tablo 2.1'de verilmiştir.

Tablo 2.1 ISO/IEC 15504 yetenek seviyeleri ve açıklamaları

Seviye	Açıklama
0	Tamamlanmamış süreç
1	Yürütülen süreç
2	Yönetilen süreç
3	Yerleşmiş süreç
4	Tahmin edilebilir süreç
5	Optimum süreç

Bu standart, süreç bölümü ve yetenek bölümü olmak üzere iki ana bölümden oluşan bir model içermektedir. Modelin süreçleri 5 kategoriye ayrılır.

- Müşteri-Tedarikçi
- Mühendislik
- Destek
- Yönetim
- Organizasyon

2.1.4 McCall Yazılım Kalite Modeli

McCall yazılım kalite modeli [5] 1977 yılında Jim McCall ve ekip arkadaşları tarafından oluşturulmuştur ve General Electrics Modeli olarak da bilinir [32]. Jim

McCall bu modeli Amerika Birleşik Devletleri Hava Kuvvetleri için kullanıcılar ve geliştiriciler arasında köprü olması amacıyla kullanıcı bakış açısını geliştirici öncelikleriyle eşlemeye çalışarak geliştirmiştir. McCall ve Boehm modelleri kalite modellerinin genel amaçlarını ve sorunlarını gösterir [33].

Yazılımların kalitesini somut olarak ölçmek için kullanılan yazılım kalite metriklerinin tanımlanması zordur ve bunu yapan ilk çalışmalardan birisi de McCall yazılım kalite modelidir. Jim McCall; bakım yapılabilirlik, esneklik, test edilebilirlik, yeniden kullanılabilirlik, taşınabilirlik, birlikte çalışılabilirlik, doğruluk, güvenilirlik, kullanılabilirlik, bütünlük, verimlilik olmak üzere toplamda 11 adet kalite niteliğini ürün revizyonu, ürüm uyum süreci ve ürün operasyonu olmak üzere 3 alt başlıkta incelemiştir. Bu modelin literatüre katkısı, harici kalite faktörleri ile ürün kalite kriterleri arasındaki ilişkileri değerlendirmesidir. Fakat, modelin dezavantajları da bulunmaktadır. Örneğin, işlevsellik kalite faktörünün mevcut olmaması ve birçok metriğin öznel olması gibi önemli dezavantajlar sayılabilir.

2.1.4.1 Ürün Operasyonu

Ürün operasyonu (product operation) alt başlığında, yazılımın gereksinimlerinin karşılama derecesini ve son kullanıcıyı etkileyen kalite faktörlerini tanımlar.

- Doğruluk: Yazılımın bir ortamdan başka bir ortama geçebilme yeteneğidir.
- Güvenilirlik: Yazılımın parçalarının hata vermeden kullanılabilme yeteneğidir.
- Kullanılabilirlik: Yazılımın parçalarının beraberce kolay bir biçimde çalışabilmesidir.
- Bütünlük: Yazılımın bir ortamdan başka bir ortama geçebilme yeteneğidir.
- Verimlilik: Yazılımın parçalarının farklı bağlamlarda kullanılabilme kolaylığıdır.

2.1.4.2 Ürün Uyum Süreci

Ürün uyum süreci (product transition) alt başlığında, ürünün yeni ortamlara uyumuyla alakalı kalite faktörleri tanımlanır.

- Taşınabilirlik: Yazılımın bir ortamdan başka bir ortama geçebilme yeteneğidir.
- Yeniden Kullanılabilirlik: Yazılımın parçalarının farklı bağlamlarda kullanılabilme kolaylığıdır.

- Birlikte Çalışılabilirlik: Yazılımın parçalarının beraberce kolay bir biçimde çalışabilmesidir.

2.1.4.3 Ürün Revizyonu

Ürün revizyonu (product revision) alt başlığında, yazılım ürününü değiştirme yeteneğini etkileyen kalite faktörleri tanımlanır. Temel olarak bu faktörler değişime hızlı ve kolay uyum sağlamak ve validasyonlar ile alakalıdır. Bu faktörler aşağıda sıralanmıştır.

- Bakım Yapılabilirlik: Hataların bulunup çözülmesi yeteneğidir.
- Esneklik: Gerekli değişiklikleri yapabilme yeteneğidir.
- Test Edilebilirlik: Gereksinimleri doğrulayabilme yeteneğidir.

2.1.5 Boehm Modeli

Boehm modeli [6], yazılımın kalitesini otomatik ve nicel olarak değerlendirmek için McCall modelini temel alan hiyerarşik bir modeldir. Boehm modelinin en önemli katkılarından birisi yazılımın bakım yapılabilirlik derecesini ifade eden kalite özellikleri ve metrikleri tanımlamasıdır.

Boehm modelinde en üst katmanda kullanışlılık (as-is utility), bakım yapılabilirlik (maintainability), taşınabilirlik (portability) olmak üzere 3 temel gereksinime ve 7 alt kalite faktörüne ayrılmıştır [33]. Modelin dezavantajı ise Boehm'in metriklerin ölçümü için bir yöntem belirtmemesidir.

2.1.6 FURPS Modeli

FURPS modeli [34] Robert Grady tarafından 1992 yılında sunulmuş ve daha sonra da IBM Rational Software [35, 36] tarafından yeni tasarım kıstasları ve geliştirme gereksinimleri eklenerek FURPS+ ile geliştirilmiş bir kalite modelidir [37]. FURPS modeli adını içerdiği kalite faktörlerinin baş harflerinin birleşiminden alır. Bu kalite özellikleri aşağıda listelenmiştir.

- İşlevsellik (Functionality)
- Kullanılabilirlik (Usability)
- Güvenilirlik (Reliability)

- Performans (Performance)
- Desteklenebilirlik (Supportability)

FURPS modelinin dezavantajı işe taşınabilirlik (portability) kalite faktörünün göz ardı etmesidir [38].

2.1.7 Dromey Modeli

Dromey kalite modeli [39] ISO 9126 kalite modelini [4] baz alan ve genişleten bir modeldir. Kalite ölçütünün her ürün için farklı olduğunu ve kalitenin modellenmesi için daha dinamik bir fikrin farklı sistemlere uygulanabilecek kadar geniş olması gerektiğini ifade eder [37]. Dromey modeli, doğruluk (correctness), dahili (internal), içeriksel (contextual), açıklayıcı (descriptive) olmak üzere 4 başlığa ayrılmıştır.

2.1.7.1 Doğruluk

Yazılım gereksinimlerini doğru bir biçimde karşılamasını ölçer.

2.1.7.2 Dahili

Yazılımın amacına uygun kullanılabilme derecesidir.

2.1.7.3 İçeriksel

Yazılımın kullanımındaki dış etmenlerle alakalı özelliklerdir.

2.1.7.4 Açıklayıcı

Yazılımın açıklanabilirlik derecesini belirtir.

2.1.8 QMOOD Modeli

QMOOD (Quality Model for Object Oriented Design, Nesneye Yönelik Tasarım İçin Kalite Modeli) kalite modeli [40] katmanlı yapıya sahip bir kalite modelidir ve nesneye yönelik programlama mantığıyla yazılmış yazılımların kalitesini ölçmeyi hedefler. Diğer birçok kalite modeli üst düzey kalite faktörleri ile tanımlanan kalite metrikleri arasındaki bağlantıyı kurmaz. QMOOD kalite modeli ise diğer birçok modele nazaran kendi kalite hesaplama yöntemi vardır ve yazılım kalite metrikleri ile üst düzeydeki kalite faktörleri ile bağlantısını modele dahil etmiştir.

QMOOD modeli hiyerarşik olan 4 katmandan oluşur.

- Tasarım Kalitesi Nitelikleri (Design Quality Attributes)
- Nesne Tabanlı Tasarım Özellikleri (Object-Oriented Design Properties)
- Nesne Tabanlı Tasarım Metrikleri (Object-Oriented Design Metrics)
- Nesne Tabanlı Tasarım Bileşenleri (Object-Oriented Design Components)

2.1.8.1 Tasarım Kalitesi Nitelikleri

Proje kapsamında bilginin verileceği ve asıl elde edilmek istenen verileri barındıran katmandır. ISO/IEC 9126 [30] standartındaki kalite faktörleri baz alınmıştır. QMOOD toplam 6 kalite faktörüne sahiptir.

- Etkinlik (Effectiveness): Tasarımın istenilen işlevi ve davranışı başarma derecesidir [40].
- Genişletilebilirlik (Extendibility): Tasarımda var olan ve kullanılan özelliklerin yeni gereksinimlere uyumlu olma becerisidir [40].
- Esneklik (Flexibility): Tasarımın yapılacak olan değişikliklere olan uyum becerisidir [40].
- İşlevsellik (Functionality): Tasarımda sınıfa atanan sorumluluklardır [40].
- Yeniden Kullanılabilirlik (Reusability): Var olan tasarım parçalarının fazla efor gerektirmeden yeni probleme uygulanabilme becerisidir [40].
- Anlaşılabilirlik (Understandability): Tasarımın kolay öğrebilebilmesi ve kavranabilme derecesidir. Kod karmaşıklığı ile direkt olarak bağlantılıdır [40].

2.1.8.2 Nesne Tabanlı Tasarım Özellikleri

Yazılım kodundaki, değişken, metot gibi yazılım birimlerinin arasındaki bağlantıları değerlendirerek çıkarımlar yapılan özelliklerdir. Bu özellikler aşağıda listelenmiştir.

- Tasarımın Boyutu (Design Size): Tasarımdaki sınıf sayılarını ölçer.
- Hiyerarşiler (Hierarchies): Tasarımdaki ata-çocuk hiyerarşi sayısını belirtir ve tasarımdaki ata sınıfların sayısı ölçer.

- Soyutlama (Abstraction): Tasarımı genelleme ve özelleştirme açısından ölçer. Bir sınıftan bir veya daha fazla sınıf türemişse bu soyutlama özelliğinin bir göstergesidir.
- Kapsülleme (Encapsulation): Sınıfın özelliklerinin ve metotlarının erişiminin kısıtlandırılmasını belirtir.
- Bağlantı veya Bağıntı (Coupling): Sınıfın başka sınıflardaki değişkenleri veya metotları kullanması durumuyla meydana gelen bağımlılığı ifade eder.
- Kohezyon (Cohesion): Tasarımdaki sınıfların iç özelliklerinin ve metotlarının içsel uyumunu belirtir.
- Kompozisyon (Composition): Tasarımdaki parça-bütün ilişkilerini inceleyerek sahiplik, parçası olma gibi kavramları ölçer.
- Kalıtım (Inheritance): Tasarımdaki sınıfların ata-çocuk hiyerarşisindeki seviyeleri ölçer.
- Poliformizm (Polymorphism): Çalışma zamanında tipleri eşleşen objelerin birbirleri yerine kullanılabilen bileşenlerin sayısını belirtir.
- Mesajlaşma (Messaging): Herkes tarafından erişilebilen (public) metotların sayısını belirtir.
- Karmaşıklık (Complexity): Anlaşılması ve bakım yapılabilirliğin zorluk derecesini belirtir.

2.1.8.3 Nesne Tabanlı Tasarım Metrikleri

QMOOD metrikleri yazılımın planlanma aşamasında ölçülüp bulunabilirler ve bir üst düzey olan tasarım özelliklerini elde etmek için kullanılırlar. QMOOD metrikleri ile ilgili daha detaylı bilgi Bölüm 3'te verilecektir.

2.1.8.4 Nesne Tabanlı Tasarım Elemanları

Nesne tabanlı programlama mantığındaki kök elemanlar olan sınıflar, değişkenler, metotlar gibi parçalar ve aralarındaki ilişkiyi inceleyen modelin en alt katmanıdır.

3
YAZILIM KOD KALİTESİ METRİKLERİ

Yazılım kod kalitesini somut ve bilimsel bir şekilde analiz etmek ve hesaplamak için hesaplanabilir, güvenilir, aynı girdiler için aynı sonucu veren formüllere ihtiyaç duyulmaktadır. Yazılım kod kalite metrikleri de bu formüllere dayanarak kod kalitesinin ölçülmesi ve çıkarılan sonuçların yorumlanması olarak ifade edilebilir. Metrikler üst seviye olarak bilinen kalite faktörlerinin tam ve elle tutulabilir halini ifade eder ve faktörler arası ilişkilere ışık tutar. Literatürde çok sayıda kod kalite metriği vardır, ancak bu çalışmada en bilinen kalite metrikleri ve özniteliklerinden bazıları ele alınmıştır. Bu çalışmada metrikler, [41]'deki gibi proje düzeyi, paket düzeyi, sınıf düzeyi ve metot düzeyi olarak dört kategoriye ayrılmıştır. Ancak sınıflandırma kişiden kişiye veya çalışmadan çalışmaya farklılık gösterebilir. Örneğin bazı çalışmalar sınıf ve metot seviyesi olarak ayırmayı tercih ederken bazı çalışmalar ise sınıf, metot gibi yazılım seviyesi olmadan sadece metrikleri oluşturan kişilere göre başlıklandırmışlardır.

3.1 Proje Seviyesi Metrikler

Proje düzeyinde yazılım kalitesini değerlendirmek ve göstermek için kullanılabilecek çok sayıda kod kalite metriği vardır. Bu metrikler, yazılımın genel kalite izlenimlerini gösterir. Bu metriklerden bazıları aşağıda açıklanmıştır.

3.1.1 MOOD Metrik Seti

MOOD metrikleri [42], nesne yönelimli projelerin genel kalitesini ifade etmek için tasarlanmıştır. MOOD metrik seti 6 metrikten oluşur.

3.1.1.1 MHF (Method Hiding Factor, Metot Gizleme Faktörü)

MHF metriği, dışarıdan erişilebilir (public, visible) metotların sayısını ölçer. Dışarıdan erişilebilir metotlar, sınıf işlevselliğinin bir ölçüsüdür. Sınıfın işlevselliğini artırmak

bu metriği azaltır. Dengeyi sağlamak için arayüzler (interface) tanımlanıp genel çağırımlar bu metotlar üzerinden yapılıp detay işlemler ise dışarıdan erişilemeyen (private, invisible) metotlarda yapılmalıdır. Düşük MHF değeri yetersiz soyutlamanın işaretiyken yüksek MHF değeri ise düşük işlevselliğin işaretidir. Ayrıca yapılan çalışmalara [43, 44] göre MHF metriğinin artması düşük hata oranı sağlarken hataların bulunup çözülmesi eforunu azaltır ve kaliteyi artırır [45].

3.1.1.2 AHF (Attribute Hiding Factor, Nitelik Gizleme Faktörü)

AHF metriği dışarıdan erişlilebilir (public, visible) niteliklerin ya da değişkenlerin sayısını ölçen yüzdelik bir değerdir. İdeal olarak tüm değişkenlerin dışarıdan erişilemeyen (private, invisible) olması yani değerin %100 olmasıdır [45].

3.1.1.3 MIF (Method Inheritance Factor, Metot Kalıtım Faktörü)

MIF metriği üst sınıflardan miras alınan metotların toplam metot sayısına oranıdır. Yüksek MIF değeri, nesneye yönelik programlamanın kalıtım paradigmasının aşırı kullanımı anlamına gelirken az çıkan MIF değeri ise az kullanımı veya metot ezilmesi veya metodun geçersiz kılınması (method overriding) işleminin çokça yapıldığını gösterir.

3.1.1.4 AIF (Attribute Inheritance Factor, Nitelik Kalıtım Faktörü)

MIF metriğine benzeyen AIF metriği ise üst sınıflardan kalıtım yoluyla miras alınan niteliklerin ya da değişkenlerin toplam değişken miktarına oranıdır. Yüksek veya düşük olması MIF metriğindeki gibi sonuçları verir.

3.1.1.5 PF (Polymorphism Factor, Polimorfizm Faktörü)

Polimorfizm sözcüğü çeşitli biçimler almak anlamına gelmeyi betimler. PF metriği, sınıfın kalıtım yoluyla aldığı metotların ezilme veya geçersiz kılınması (overriding) derecesini ölçer. Tüm sınıflardaki geçersiz kılınan yöntemlerin toplam sayısının, tüm sınıflardaki yeni yöntemlerin sayısının tüm sınıflardaki alt öğelerin sayısıyla çarpılmasının sonucuna bölünmesiyle hesaplanır. PF metriğinin yüksek olması yazılım kodunun kompleks olabileceğinin ve anlaşılması zor olabileceğinin işaretidir.

3.1.1.6 CF (Coupling Factor, Bağlılık Faktörü)

CF metriği sınıflar arası bağlılık (bağlaşım, bağımlılık, bağlama) derecesini ölçer. Bir X sınıfı, Y sınıfının metotlarına veya değişkenlerine erişiyorsa X sınıfı Y sınıfına bağlıdır

yargısı çıkarılabilir. Eğer hiçbir sınıf birbirlerinin değişkenlerine veya metotlarına erişmiyorsa CF metriği %0 iken bütün sınıflar erişiyorsa %100'dür.

3.1.2 QMOOD Metrikleri

QMOOD [40], genişletilebilirlik ve esneklik gibi nesne yönelimli tasarım kalite özelliklerini değerlendirmek ve bunu kapsülleme ve bağlılık gibi nesne yönelimli tasarım özellikleri aracılığıyla ifade etmek için iyi tanımlanmış ve doğrulanmış bir model oluşturan kapsamlı bir kalite modelidir. Bu özellikleri elde etmek için tasarım sınıfı boyutu metriği ve yöntem sayısı metrikleri gibi bir metrik listesi kullanılır. Bu metrikler ve açıklamaları aşağıda verilmiştir.

3.1.2.1 Tasarımdaki Sınıf Sayısı (Design Size in Classes, DSC)

Tasarımdaki sınıfların sayısını ölçen bir metriktir.

3.1.2.2 Hiyerarşi Sayısı (Number of Hierarchies, NOH)

Tasarımdaki sınıf hiyerarşi sayısını ölçen bir metriktir.

3.1.2.3 Ortalama Ata Sınıf Sayısı (Average Number of Ancestors, ANA)

Tasarımdaki ortalama ata sınıf sayısını ölçen bir metriktir.

3.1.2.4 Veri Erişim Metriği (Data Access Metric, DAM)

Bir sınıf için dışarıdan erişilebilir olmayan değişkenlerin tüm değişkenlere oranını ölçen bir metriktir.

3.1.2.5 Direkt Sınıf Bağımlılığı (Direct Class Coupling, DCC)

Sınıf ile doğrudan ilişkili olan sınıfların sayısını ölçen bir metriktir.

3.1.2.6 Sınıf Metotları Arasındaki Kohezyon (Cohesion Among Methods of Class, CAM)

Sınıf metotlarının aldığı parametrelere göre ilişkililik düzeyini ölçen bir metriktir.

3.1.2.7 Kümelenme Ölçütü (Measure of Aggregation, MOA)

Kullanıcı tarafından belirlenmiş veri deklerasyonlarını ölçen bir metriktir.

3.1.2.8 İşlevsel Soyutlama Ölçütü (Measure of Functional Abstraction, MFA)

Sınıfın miras aldığı metotların sayısının sınıfın üye metotları tarafından erişilen toplam yöntem sayısına oranını ölçen bir metriktir.

3.1.2.9 Çok Biçimli Metot Sayısı (Number of Polymorphic Methods, NOP)

Çok biçimli davranış gösterebilen metot sayısını ölçen bir metriktir.

3.1.2.10 Sınıfsal Arayüz Boyutu (Class Interface Size, CIS)

Sınıfın içerdiği dışarıdan erişilebilen (public) metotların sayısını ölçen bir metriktir.

3.1.2.11 Metot Sayısı (Number of Methods, NOM)

Sınıfın içerdiği bütün (public, private) metotların sayısını ölçen bir metriktir.

3.2 Paket Seviyesi Metrikler

Paket (package), birbirleriyle bağlantılı olan somut veya soyut sınıflar, arayüzler, alt paketler ve diğer öğelerden oluşan bir koleksiyondur. Paket seviyesi metrikler, yazılım paketlerinin kalitesini ölçmek için kullanılan bir dizi kalite ölçüsüdür. Paket seviyesi metriklerine örnek olarak Robert C. Martin tarafından sunulan metrikler [46] örnek verilebilir.

- A (Abstractness, Soyutluk)
- I (Instability, Değişkenlik)
- Ce (Efferent Coupling, Dışarı Bağlılık)
- Ca (Afferent Coupling, İçeri Bağlılık)
- D (Normalized distance from the main sequence, Ana diziden normalleştirilmiş mesafe)

3.2.1 Soyutluk

Paket içindeki soyut yapıların toplam paket boyutuna oranını belirtir.

3.2.2 Dışarı Bağlılık

Paket içerisindeki yapıların başka paketlerdeki yapılara erişme sayısını belirtir.

3.2.3 İçeri Bağlılık

Paket içerisindeki yapıların başka paketlerden erişilme sayısını belirtir.

3.2.4 Değişkenlik

Paket içerisindeki yapıların değiştirilebilmeye yatkınlık oranını belirtir. Bir paketin ne çok değiştirilebilir ne de hiç değiştirilemez olması gerekir ve ideal değer olarak 0.3 veya 0.7 olması beklenir [47]. Metrik aşağıdaki formül ile hesaplanır.

$$I = \frac{Ce}{Ce + Ca} \tag{3.1}$$

3.2.5 Ana diziden normalleştirilmiş mesafe

Paketin ideal soyutlama ve değiştirilebilme çizgisine uzaklığını belirtir. Bu metrik için iki istenmeyen durum mevcuttur. Bunlardan ilki soyutluk metriğinin ve değişkenlik metriklerinin sıfır olma durumu iken diğeri ise ikisinin de bir olma durumudur. Metrik değerinin olabildiğince küçük olması beklenir ve aşağıdaki formül ile hesaplanır.

$$D = |A + I - 1| \tag{3.2}$$

3.3 Sınıf Seviyesi Metrikler

Sınıf düzeyinde yazılım kalitesini ölçmek ve göstermek için kullanılabilecek çok sayıda ölçüm vardır. Chidamber-Kemerer metrikleri [48] en iyi bilinenleridir. Kısaltma olarak CK metrikleri olarak bilinir. Diğer metrikler arasında Lorenz-Kidd metrikleri [49], Li-Henry metrikleri [50], Lanza-Marinescu metrikleri [51] ve Bieman-Kang metrikleri [52] bulunmaktadır.

3.3.1 CK Metrikleri

1994 yılında Chidamber ve Kemerer tarafından tanıtılan nesne yönelimli sistemler için bir dizi metriktir. Metrikler altı tanedir ve aşağıda listelenmiştir.

- WMC (Weighted Methods per Class, Sınıf Başı Ağırlıklı Metotlar)
- DIT (Depth of Inheritance Tree, Kalıtım Ağacı Derinliği)
- NOC (Number of Children, Çocuk Sayısı)

- CBO (Coupling Between Object Classes, Sınıf Nesneleri Arasındaki Bağlılık)
- RFC (Response For a Class, Sınıf İçin Cevap)
- LCOM (Lack of Cohesion in Methods, Metotlardaki Uyum Eksikliği)

3.3.1.1 WMC

WMC metriği, bir sınıf için ağırlıklı metotlar anlamına gelmektedir. Bir S sınıfı için;

$$WMC(S) = c_1 + ... + c_n \tag{3.3}$$

her c_i, S sınıfının ilgili indeksindeki metodunun karmaşıklığı olmak üzere WMC değeri hesaplanır.

Metot karmaşıklığı için belirlenen standart bir yöntem olmamakla beraber genellikle McCabe Cyclomatic Complexity ölçüm yöntemi tercih edilir.

3.3.1.2 DIT

DIT metriği, bir S sınıfı için kalıtım ağacının derinliğini yani S sınıfından S sınıfının ata sınıflarına kadar olan maksimum uzunluğu belirtir.

Yüksek DIT değerleri, ata sınıfların sayısını artırdığından dolayı kodun anlaşılabilirliğini düşürür. Çünkü ata sınıf sayısı arttıkça hem miras yoluyla elde edilen değişken ve metot sayısı artar hem de metotların ezilme (override) yöntemi ile davranışları değişebileceğinden çalışma zamanındaki davranışlarını tahmin etmek güçleşir. Anlaşılabilirliğin düşmesiyle beraber bakım yapılabilirlik de düşebilir. Fakat öte yandan yüksek kalıtım yüksek yeniden kullanılabilirlik sağlayabilir ve test edilebilirlik açısından avantaj haline dönüşebilir [53].

3.3.1.3 NOC

NOC metriği bir S sınıfının ilk seviyeden sahip olduğu alt sınıf sayılarını belirten sınıf seviyesi metriktir.

NOC metriğinin yüksek olması sınıftaki bir hatanın birçok alt sınıfa aktarılabileceğinden dolayı tehlikelidir. Bu nedenle ölçüm araçlarında eşik seviyesi düşüktür.

3.3.1.4 CBO

CBO metriği birbirlerine bağlı olan sınıf sayısını gösterir. Eğer bir C sınıfı D sınıfının değişkenlerini veya metotlarını kullanıyorsa bu iki sınıfın bağlı olduğunu gösterir. Aynı sınıfın değişkenlerini veya metotlarını birden fazla kullanmak önemsizdir. Önemli olan erişimin bulunup bulunmadığıdır. Bir metot polimorfikse (metot ezme (override) veya aşırı yüklemeler (overload) nedeniyle), metotun bulunduğu tüm sınıflar toplam sayıma dahil edilir [48].

Yüksek CBO değeri istenilen bir durum değildir. Çünkü sınıflar arasındaki fazla bağlantı modüler tasarımı engeller, değiştirilebilme (replecability) azalır ve başka bir yazılım kodunda kullanımı da zorlaşır. Ayrıca yazılım koduna yapılacak bir bakım, yazılım kodunun bakımla ilgisi olmayan kısımlarının da değiştirilmesini gerektirebilir ve bu nedenle bakım yapılabilirlik derecesi düşebilir.

3.3.1.5 RFC

RFC metriği bir S sınıfına ait olan veya S sınıfının metotları tarafından çağrılan metot kümesini belirtir.

RFC metriği özellikle sınıfa ait olmayan metotları barındırdığından, sınıfın diğer sınıflarla olan iletişimini de gösterir. Yüksek RFC değerine sahip olan sınıflar daha karmaşık ve anlaşılması daha zordur. Test etme ve hata ayıklama işlemlerinin maliyeti yüksektir [54].

3.3.1.6 LCOM

LCOM metriği, sınıfın metotları arasındaki kohezyon eksikliğini ölçer. İdeal bir sınıfta LCOM metrik değerinin düşük olması beklenir çünkü düşük kohezyon tasarımda karmaşıklığa yol açar [53].

LCOM metriği hesaplanırken sınıftaki metotların ikişer kombinasyonları alınır ve nesne referansları paylaşıyorlarsa LCOM değeri 1 azalır, eğer herhangi bir nesne değişkeni referansı paylaşmıyorlarsa LCOM değeri 1 artar. Böylece metotlar arası kohezyon ölçülmüş ve sınıfın metotlarının birbirleriyle ne kadar uyumlu olarak çalıştığı ifade edilmiş olur.

3.3.2 Lorenz-Kidd Metrikleri

Lorenz ve Kidd sınıf kalitesini ölçmek için 3 başlık altında incelenebilen bir dizi metrik sundu. Metriklerin toplandığı 3 başlık ise aşağıda verilmiştir.

- Dahili (Internal)
- Harici (External)
- Boyut (Size)

Bu kategorideki bazı metrikler ve açıklamaları, aşağıda verilmiştir.

3.3.2.1 Nitelik Sayısı (Number of Attributes, NOA)

Sınıfa ait toplam değişken sayısını gösterir.

3.3.2.2 Operasyon Sayısı (Number of Operations, NOO)

Sınıfa ait toplam operasyon sayısını gösterir.

3.3.2.3 Eklenen Metot Sayısı (Number of Added Methods, NOAM)

Sınıfa eklenen metot sayısını gösterir.

3.3.2.4 Ezilen Metot Sayısı (Number of Overridden Methods, NOOM)

Bir üst sınıftan miras alınan ve sınıf içinde ezilen veya geçersiz kılınan (overridden) metot sayısını gösterir.

3.3.3 Li-Henry Metrikleri

1993 yılında Li ve Henry bir dizi metrik önerdiler ve metrikleriyle iki sistemin bakım yapılabilirlik eforunu gözlemlediler. Gözlemin sonunda aşağıdaki metrikleri tanımladılar.

3.3.3.1 SIZE1

Sınıftaki toplam noktalı virgül sayısıdır. Sınıftaki toplam metot veya prosedür çağırımı sayısını ifade eder.

3.3.3.2 SIZE2

Sınıfın toplam değişken ve metot sayısını gösterir. NOA ve NOM metriklerinin toplamı olarak ifade edilebilir.

3.3.3.3 Mesaj Geçiş Bağlantısı (Message Passing Coupling, MPC)

Sınıfın metotları içinde başka sınıflara ait metotları çağıran metot sayısıdır. Böylece sınıfın başka sınıflara olan bağımlılığını ölçer.

3.3.3.4 Veri Soyutlama Bağlılığı (Data Abstraction Coupling, DAC)

Sınıftaki soyut veri tiplerinin sayısıdır. Fazla soyut veri tiplerinin neden olduğu karmaşıklığı ifade eder. DAC metriği, nesne tabanlı tasarımın sınıflar arasındaki bağlantısıyla ilgilidir, çünkü sınıfın yeniden kullanılabilme seviyesi, sürdürülebilirlik ve test edilebilirlik maliyeti, sınıflar arasındaki bağlantılara bağlıdır.

3.3.3.5 Metot Sayısı (Number of Methods, NOM)

Sınıftaki toplam yerel metot sayısını gösterir. NOM metriğine kalıtım özelliğiyle üst sınıflardan gelen metotlar dahil edilmez.

3.3.4 Lanza-Marinescu Metrikleri

Lanza ve Marinescu [51]'de yazılım tasarım kusurlarını analiz etmiş ve bunlara uyumsuzluk (disharmonies) adını vermişlerdir. Toplamda 11 tasarım kusuru tanımlayıp bunlar için metrik tabanlı bir model önermişlerdir. Bu modelin sınıf bazlı metrikleri aşağıda listelenmiştir.

- Dış Veriye Erişim (Access to Foreign Data, ATFD)
- Herkese Açık Değişken Sayısı (Number of Public Attributes, NOPA)
- Erişen Metot Sayısı (Number of Accessor Methods, NOAC)
- Sınıfın Ağırlığı (Weight of a Class, WOC)

3.3.4.1 Dış Veriye Erişim Metriği

Birbiriyle bağımsız sınıfların değişkenlerine ya direkt ya da dolaylı olarak erişilmesini ölçer. Yüksek değer olması sınıf için düşük kohezyonu ve sınıflar arası yüksek bağlantıyı gösterir.

3.3.4.2 Erişilen Değişken Sayısı Metriği

Bir metot ya da fonksiyon tarafından erişilen değişken sayısını belirtir.

3.3.4.3 Herkese Açık Değişken Sayısı Metriği

NOPA metriği, sınıftaki herkes tarafından erişilebilen değişkenleri ya da alanların sayısını belirtir.

3.3.4.4 Erişen Metot Sayısı Metriği

Sınıftaki değişkenleri getiren veya değiştiren (getter ve setter) metotların sayısıdır.

3.3.4.5 Bağlılık Dağılımı Metriği

Bir sınıf içindeki toplam metot çağırımları bağımlığın sınıfın içinde mi kaldığını yoksa sınıfın bağımlılığının diğer sınıflara doğru mu olduğunu belirtir. Eğer sınıftaki çağırılan metotların çoğu o sınıfa aitse CDISP değeri düşük olup bağlılık sınıf için kalmıştır çıkarımı yapılabilir.

3.3.4.6 Sınıfın Ağırlığı Metriği

Sınıftaki erişen metotlar, yapıcı metotlar hariç diğer metotların sayısının toplam metot sayısına oranıdır. Metriğin düşük değeri sınıf değişkenlerinin metotlarından daha fazla sergilendiği anlamına gelir ve düşük kapsülleme göstergesidir [55].

3.3.5 Bieman-Kang Metrikleri

Bieman ve Kang, 1995 yılında metot çiftlerinin doğrudan ve dolaylı bağlantılarına dayanan iki sınıf uyumu ölçütü tanımladılar. Bu metrikler aşağıda verilmiştir.

- Sıkı Sınıf Kohezyonu (Tight Class Cohesion, TSC)
- Gevşek Sınıf Kohezyonu (Loose Class Cohesion, LCC)

3.3.5.1 Sıkı Sınıf Kohezyonu

Sınıfın en az bir ortak değişkenine doğrudan erişen metot çiftlerinin sayısını belirtir. Çünkü eğer iki metot da aynı değişkene erişiyorsa bu iki metot birbirlerine bu değişken üzerinden bağlıdır çıkarımı yapılır. Yüksek TCC değeri sınıfın bileşenlerine ayrılmasının zorluğunu belirtirken düşük değeri ise sınıfın birçok alakasız işlemi yapmaya çalıştığının göstergesi olabilir.

3.3.5.2 Gevşek Sınıf Kohezyonu

TCC metriği gibi sınıftaki metotların birbirlerine değişken üzerinden bağlı olması durumuyla sınıf kohezyonunu ölçer. TCC metriğindeki gibi değişkenlere doğrudan şartına ek olarak dolaylı olarak erişimi de kapsar.

3.4 Metot Seviyesi Metrikler

Metot seviyesi metrikleri, metota özgü kaliteyi ölçer ve genel bir yargı çıkarımını sağlar. Bu metriklerden bazıları aşağıda verilmiştir.

- Kod Satır Sayısı (Lines of Code, LOC)
- McCabe Döngüsel Karmaşıklığı (McCabe Cyclomatic Complexity, CC)
- Maksimum Gruplanma Derinliği (Maximum Nesting Depth, MND)
- Döngü Derinliği (Loop Nesting Depth, LND)
- Koşul Derinliği (Condition Nesting Depth, CND)
- Döngü Sayısı (Number of Loops, NOL)
- Değişken Erişimlerinin Yerelliği (Locality of Attribute Accesses, LAA)
- Yabancı Veri Sağlayıcıları (Foreign Data Providers, FDP)
- Erişilen Değişken Sayısı (Number of Accessed Variables, NOAV)
- Bağlılık Yoğunluğu (Coupling Intensity, CINT)
- Bağlılık Dağılımı (Coupling Dispersion, CDISP)

3.4.1 Kod Satır Sayısı

Metodun içerdiği satır sayısını belirtir. Satır sayısı hesaplanırken koddaki yorumlar da eklenir çünkü kodun değiştirilmesi durumunda yorumlar da değişmek zorundadır bu nedenle bakım maliyetine yorumlar da katılır. Metriğin fazla olması bakım maliyetini artırır.

3.4.2 McCabe Döngüsel Karmaşıklığı

Döngüsel veya diğer adıyla çevrimsel olarak bilinen bu karmaşıklık metriği, yazılım kodundaki akışın farklılaşmasını ölçer. Döngüler, koşullu ifadeler (if-else, switch) gibi akışı dallandıran komutlar döngüsel karmaşıklığı artırır.

Yüksek döngüsel karmaşıklık metriği; yazılım kodunun karmaşıklığını, dolayısıyla bakım yapılabilitesinin ve test edilebilirliğinin düşüklüğünü gösterir [56].

3.4.3 Maksimum Gruplanma Derinliği

Lanza ve Marinescu tarafından sunulan [51] maksimum gruplanma derinliği metriği, metodda bulunan kontrol yapılarının en derin olanının derinlik seviyesini belirtir. Metriğin fazla olması okunabilirliğin ve anlaşılabilirliğin az, karmaşıklığın ise fazla olduğunu gösterir.

3.4.3.1 Döngü Derinliği

Döngü derinliği metriği, bir metodda bulunan en derin iç içe kullanılan döngünün derinliğini belirtir. Metriğin fazla olması okunabilirliğin ve anlaşılabilirliğin az, karmaşıklığın ise fazla olduğunu gösterir.

3.4.4 Koşul Derinliği

Koşul derinliği metriği, bir metodda bulunan en derin iç içe kullanılan koşullu ifadenin derinliğini ifade eder. Metriğin fazla olması okunabilirliğin ve anlaşılabilirliğin az, karmaşıklığın ise fazla olduğunu gösterir.

3.4.5 Döngü Sayısı

Döngü sayısı metriği, bir metodda bulunan döngülerin toplam sayısını belirtir. Metriğin fazla olması; okunabilirliğin,, değiştirilebilirliğin ve anlaşılabilirliğin az, karmaşıklığın ise fazla olduğunu gösterir.

3.4.6 Değişken Erişimlerinin Yerelliği

Lanza ve Marinescu tarafından sunulan [51] değişken erişimlerinin yerelliği metriği, bir metodun içinden erişilen tüm değişkenlerin metodun bulunduğu sınıfa ait olma oranını belirtir. Örneğin "S" sınıfındaki bir "M" metodunun, eriştiği toplam beş değişkeninin üçü S sınıfına ait, geri kalanlar farklı sınıfa veya sınıflara aitse, M metodu için bu metrik değeri 0.6 olacaktır.

3.4.7 Yabancı Veri Sağlayıcıları

Lanza ve Marinescu tarafından sunulan [51] yabancı veri erişimi metriği, bir metodun içinden erişilen ve o sınıfa ait olmayan verilerin kaynak (sınıf) çeşitliliğini belirtir.

Örneğin "S" sınıfındaki bir "M" metodunun, eriştiği toplam 3 değişkeninin ikisi farklı sınıfa veya sınıflara aitse, M metodu için bu metrik değeri 2 olacaktır.

3.4.8 Erişilen Değişken Sayısı

Lanza ve Marinescu tarafından sunulan [51] erişilen değişken sayısı metriği, bir metot içerisinden erişilen toplam değişken sayısını belirtir.

3.4.9 Bağlılık Yoğunluğu

Lanza ve Marinescu tarafından sunulan [51] bağlılık yoğunluğu metriği, bir metot içerisinden çağrımı yapılan metotların farklı sınıflara ait olma oranını belirtir.

3.4.10 Bağlılık Dağılımı

Lanza ve Marinescu tarafından sunulan [51] bağlılık dağılımı metriği, bir metot içerisinden çağrımı yapılan farklı sınıfa ait olan metotların kaynak (sınıf) çeşitliliğini belirtir.

4
TASARIM ÖRÜNTÜLERİ

Tasarım örüntüleri ya da tasarım kalıpları, bir programın veya bilgisayar sisteminin yazılım mimarisi, yazılım öğelerini, bu öğelerin dışarıdan görülebilen özelliklerini ve bunlar arasındaki ilişkileri içeren sistemin yapısı veya yapılarıdır [57]. Tasarım örüntüleri başarısı kanıtlanmış, belirli ihtiyaçlara karşılık veren tasarımların tekrar kullanılabilmesini sağlar. Bu başarısı kanıtlanmış, birçok geliştirici tarafından kullanılan ve benimsenen teknikler, tasarım örüntüleri olarak soyutlanması sayesinde yeniden kullanılabilir hale gelir. Tasarım örüntüleri, direkt projeye dahil edilip çalıştırılabilecek algoritmalar gibi bitmiş kod parçaları değildir. Çünkü tasarım örüntüleri hesaplama problemlerini çözmek yerine belirli tasarım sorunlara nasıl çözüm yaklaşımında bulunabileceği hakkında yol gösterir.

4.1 Tasarım Örüntüleri Çeşitleri

Tasarım kalıpları farklı kişilere göre birçok farklı alt başlığa ayrılabilir. Bu çalışmada ise 3 alt başlık altında olan kabul edilmiştir [58].

- Yaratımsal (Creational)
- Davranışsal (Behavioral)
- Yapısal (Structural)

4.1.1 Yaratımsal Tasarım Örüntüleri

Nesne tabanlı programlamadaki en temel kavramlardan biri olan obje kavramının yaratımı çok önemlidir. Bir sınıftan, belirli koşullarda farklı biçimlerde objeler oluşturulmak istenebilir, bu nedenle objelerin ihtiyaçlara göre yaratılması gerekir. Basit bir şekilde obje oluşturup modifiye etmek tasarıma karmaşıklık katabilir veya tasarımda bazı problemlere yol açabilir.

Yaratımsal tasarım kalıpları, bir sınıftan oluşturulacak objelerin oluşumuyla ilgili problemleri ele alıp belirli yöntemler önererek obje yaratım işindeki bu problemlerin kaldırılmasına ve tasarımın basit kalmasını sağlar. Böylece geliştirici nesneleri yaratırken hangi şekilde ve nereden yaratıldığı ile ilgili fazla kafa yormadan geliştirmesini yapabilir. Bu aynı zamanda bakım maliyetini de düşürür.

Bu kategorideki bilinen bazı tasarım kalıpları aşağıda listelenmiştir.

- Fabrika Metodu (Factory Method)
- Soyut Fabrika Metodu (Abstract Factory)
- Yapıcı (Builder)
- Prototip (Prototype)
- Tek Nesne (Singleton)

4.1.2 Davranışsal Tasarım Örüntüleri

Nesneye yönelik programlamadaki temel kavramlardan bir diğeri ise objelerin arasındaki etkileşimdir. Nesneler doğası gereği sürekli etkileşim halinde oldukları için aralarındaki etkileşimin sorunsuz, kolay ve tekrarlanabilir olması, yazılımın geliştirilme ve bakım aşamasında yapılacak değişikliğin veya eklenecek özelliğin rahat bir şekilde uygulanabilmesini sağlar. İyi tasarlanmamış nesne etkileşimleri, yazılım kodunu kompleksleştirip geliştirme ve bakım maliyetleri artırabileceği gibi bunun yanında potansiyel hatalara da davetiye çıkarır.

Davranışsal tasarım kalıpları, belirli işlevleri daha kolay ve esnek yerine getirebilmek adına nesnelere yüklenecek olan sorumlulukların planlanması, dağıtımı gibi işlemlerde yol gösterici yöntemler önerir.

Bu kategorideki bilinen bazı tasarım kalıpları aşağıda sıralanmıştır.

- Sorumluluk Zinciri (Chain of Resposibility)
- Komut (Command)
- Yineleyici (Iterator)
- Aracı (Mediator)
- Hatıra (Memento)

- Gözlemci (Observer)
- Durum (State)
- Strateji (Strategy)
- Şablon (Template)
- Ziyaretçi (Visitor)

4.1.3 Yapısal Tasarım Örüntüleri

Yazılım kaynak kodu geliştirirken yapılacak ilk işlemlerden birisi yazılımda kurulmak istenen bileşenler arasındaki yapının veya yapıların nasıl olması gerektiğini planlamaktır. Bileşenler arasındaki yapı ne kadar esnek ve tekrar kullanılabilir olursa geliştirme ve bakım maliyetleri de o kadar azalır.

Yapısal tasarım kalıpları, yazılım bileşenlerinin geniş ölçekte kullanımının esnek, tekrar kullanılabilir ve verimli bir biçimde nasıl yapılabileceğine ilişkin yöntemler gösterir.

Bu kategorideki temel olarak bilinen tasarım kalıpları aşağıda listelenmiştir.

- Adaptör (Adapter)
- Köprü (Bridge)
- Kompozit (Composite)
- Dekoratör (Decorator)
- Cephe (Facade)
- Sineksiklet (Flyweight)
- Vekil (Proxy)

Bu kalıplar dışında, bu tez çalışmasında da kullanılan ve birçok kişi tarafından mimarisel tasarım kalıbı alt başlığında da gösterilebilecek yazılım projesinin en dış katmanındaki yapısal tasarım kalıbı olarak betimlenebilecek tasarım kalıpları vardır. Bu tasarım kalıpları yazılım kodundaki ana ve alt bileşenlerin nasıl birbirleriyle uyumlu bir halde etkileşime geçeceğini belirler. Bu mimarisel tasarım kalıpları içinden Android platformuna yönelik yazılım projeleri geliştirilirken kullanılan bazı popüler tasarım kalıpları olarak aşağıdakiler gösterilebilir [59].

- MVC (Model View Controller, Model Arayüz Kontrolcü)
- MVP (Model View Presenter, Model Arayüz Sunucu)
- MVVM (Model View View-Model, Model Arayüz Arayüz-Modeli)
- MVI (Model View Intent, Model Arayüz Amaç)
- VIPER (View Interactor Presenter Entity Router, Arayüz Etkileşimci Sunucu Varlık Yönlendirici)

4.1.3.1 MVC

MVC; model, arayüz ve kontrolcü parçalarından oluşan bir tasarım kalıbıdır. Buradaki amaç sorumlulukların birbirlerinden ayrılması prensibini uygulamak ve her bir parçanın kendi sorumlu olduğu işlevi yönetmesi gerektiğidir. Böylece bir parçada birden çok iş yükü olmayacak ve parça başına geliştirme ve bakım maliyeti azalacaktır. Kalıbın genel yapısı Şekil 4.1'de gösterilmiştir.

Şekil 4.1 MVC yapısı

- Model, uygulamadaki veri katmanını sembolize eder ve uygulamada kullanılacak verilerin sağlanmasından sorumlu katmandır.
- Arayüz, kullanıcıya gösterilmesi istenilen şeyleri gösteren ve kullanıcı etkileşimlerini bildirmek adına kontrolcü ile iletişime geçen katmandır.
- Kontrolcü, iş akışını yönetmekle görevli katmandır.

MVC kalıbının çeşitli şekilde uygulanış biçimleri vardır. Bu biçimlerden en bilinenleri modelin pasif ya da aktif oluşuna göre şekillenen uygulanışlardır. Pasif modelde, kontrolcü bileşeni modeli günceller ve arayüze de modelin güncellendiğinin haberini verir. Böylece arayüz, model bileşeninden veriyi alır. Aktif modelde ise model bileşeni, aktif rol oynayarak gözlemci tasarım kalıbının yardımıyla arayüze verinin güncellendiğini kendisi haber verir [60].

Küçük ölçekli projelerde, hem öğrenilmesi ve pratiğe dökmesi kolay olması hem de yapısı gereği hızlı geliştirmeye olanak vermesi nedeniyle, MVC tasarım kalıbının seçilmesi iyi sonuç verebilir. Fakat proje ölçeği büyüdükçe iş mantığı kodlarının artması ve sorumlulukların iyi ayrılamaması nedeniyle bakım yapılabilirlik azalır.

Bir Android yazılım projesi için, MVC kalıbının avantajları olarak

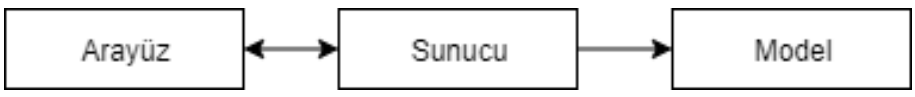
- Proje geliştirme sürecini hızlandırması ve bakım yapılabilirlik maliyetini azaltması
- Birden fazla geliştiricinin aynı anda daha rahat çalışabilmesine olanak tanıması
- Test yapılabilirliği artırması
- Hata ayıklama sürecini kolaylaştırması

maddeleri örnek verilebilirken dezavatajları olarak aşağıdaki maddeler verilebilir.

- Arayüz ve model bileşenleri birbirleriyle sıkı bağımlıdır.
- Arayüz hem model hem de kontrolcü bileşenine bağımlıdır.
- Arayüzde veri görüntülenmesinin nasıl ve hangi bileşen tarafından yapılacağı net olmadığı için karmaşıklığı artırır, sorumlulukların ayrılması prensibi ihlal edilmiş olur.

4.1.3.2 MVP

MVP; model, arayüz ve sunucu adlı parçalardan oluşan bir tasarım kalıbıdır. Amaç MVC kalıbında olduğu gibi sorumlulukların dağıtılıp tek bir parçaya çok görev vermemektir. MVC kalıbından farklı olarak, kontrolcü yerine sunucu adlı parçanın gelmesi ve bileşenler arasındaki etkileşiminin değişmesi söylenebilir. Kalıbın genel yapısı Şekil 4.2'de gösterilmiştir.



Şekil 4.2 MVP yapısı

- Model, MVC kalıbındaki gibi uygulamadaki veri katmanını sembolize eder ve veri işlerinden sorumludur.

- Arayüz, kullanıcıya gösterilmesi istenilen verileri gösteren ve sunucu ile kullanıcı etkileşimleri hakkında iletişime geçen katmandır.
- Sunucu, uygulamadaki genel işleyiş mantığının bulunduğu katmandır ve uygulamanın genel akışını yönetmekle görevlidir. Modelden veri alınması ve kullanıcı arayüzüne iletilmesi veya kullanıcıyı farklı ekranlara yönlendirmek için gerekli tetiklemeleri yapmak gibi görevleri vardır.

MVP tasarım kalıbında sunucu katmanı, kullanıcı arayüzüne olan erişimi soyutlanmış bir katman üzerindendir. Arayüz kullanıcının yaptığı değişiklikleri kendi oluşturduğu sunucuya iletir, sunucu ise yine soyut arayüz katmanı üzerinden arayüz metotlarını çağırıp arayüz ile etkileşime girer. Böylece sunucu verinin nasıl gösterileceğini kendi belirler ve arayüz üzerinden bu sorumluluğu almış olur. Bu avantajla beraber sunucu ve arayüz arasında direkt bağlantı yerine soyut sınıflar (interface) ile dolaylı bir bağlantı olmuş olup test esnasında arayüz katmanı rahat bir şekilde değiştirilebilmektedir.

MVP tasarım kalıbının MVC'ye göre avantajı olarak

- Arayüzün model ile olan bağlantısını kaldırır.
- Arayüzde görüntülenecek olan verinin nasıl görüntüleneceğine dair sorumluluğu arayüzden alır.
- Daha iyi bir sorumluluk ayrımı ilkesi sağlar.
- Test edilebilirlik daha fazladır.

maddeleri örnek verilebilirken, dezavantajları olarak ise aşağıdaki maddeler verilebilir.

- Büyük ölçekli projeler için iyi bir kalıp olsa da küçük ölçekli projelerde birçok sınıf ve soyut sınıf oluşturulup bakım yapılması gerekliliği geliştiriciye ekstra yük getirir.
- Arayüzdeki veri gösterim mantığının sunucuya alınmasıyla beraber sunucu katmanındaki sınıflardaki kod sayısı artar ve sunucu her şeyi bilen bir yapıya dönüşmeye başlayabilir. Bunun önüne geçmek için ise kod mümkün olduğunca bölünmeli ve sınıfın tek bir sorumluluğu olmalıdır.

4.1.3.3 MVVM

MVVM; model, arayüz ve arayüz modeli parçalarından oluşan yapısal tasarım kalıbıdır [59, 61]. İlk bakışta MVP tasarım kalıbına benzerliği dikkat çekse de daha çok kullanıcı arayüzleri için olay yönelimli programlama felsefesiyle MVC ve MVP kalıplarından ayrılır. MVVM'in çıkış amaçları arasında MVC'de kontrolcüde ve MVP'de sunucuda bulunan yoğunluğun indirgenmesi vardır.

MVVM tasarım kalıbı Android projelerinde "LiveData" yapısı [62] ve özellikle "Databinding" kütüphanesiyle [63] beraber çok sık bir biçimde kullanılır. Bu kütüphaneler kalıbın belirttiği arayüz ve arayüz modeli arasındaki veri yönetimini kolay bir şekilde yapabilmeyi sağlar. Kalıbın genel yapısı Şekil 4.3'de gösterilmiştir.

Şekil 4.3 MVVM yapısı

MVVM kalıbındaki bileşenler aşağıdaki maddeler halinde özetlenebilir.

- Model, diğer kalıplarda olduğu gibi burada da uygulamada gösterilecek veri işlerinden sorumludur. Veritabanı veya servis sorgulamaları bu katmanda yapılır.
- Arayüz, istenilen bilgileri gösteren ve kullanıcı etkileşimlerini arayüz modeline bildiren katmandır.
- Arayüz modeli ise verileri model katmanından alıp akışlar (stream) halinde arayüze bildiren katmandır.

Kalıbın avantajları olarak

- Model katmanındaki değişiklikler arayüz modelinden akışlar halinde bildirildiği için arayüz katmanının MVP kalıbındaki presenter katmanına olan referansı gibi arayüz modeli katmanına bir referans tutmasına gerek olmaması
- Referanslar olmadığı için bu referansları soyutlaştıran soyut sınıflara (kontrat arayüzleri) gerek olmaması

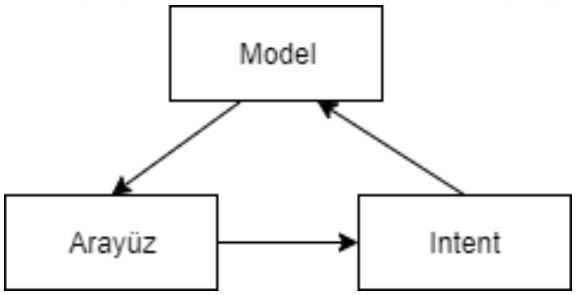
maddeleri örnek verilebilirken dezavantajları olarak aşağıdaki maddeler örnek verilebilir.

- Küçük ölçekli projelerde uygulanması zahmetlidir.
- Projede yapılan büyük değişiklikler hem arayüzde hem de arayüz modelinde yapılacak birçok değişikliğe sebep olacağından maliyet fazla olur.
- Yeniden kullanılabilirlik az ve kod tekrarı fazla olabilir.
- Geliştiricilerin tecrübesiz olması kalıbın yanlış uygulanışlarına ve teknik borçlanmalara sebep olabilir.

4.1.3.4 MVI

MVVM; model, arayüz ve amaç katmanlarından oluşan bir tasarım kalıbıdır. MVC, MVP ve MVVM kalıpları gibi MVI tasarım kalıbı da kodun bakım yapılması kolay, test edilebilir ve esnek olması için tasarlanmıştır.

Android projelerinde durum yönetimi ve arayüzün işlenişi proje ölçeği büyüdükçe karmaşık ve yönetilmesi zor bir hâl alabilir. MVI kalıbındaki durum yönetimi fikri buna çözüm getirir. Bununla birlikte; MVP için iki yönlü bağlantı (bidirectional coupling) veya arayüz durumlarını elle yeniden oluşturma ihtiyacı, MVVM için tek bir veri kaynağının olmaması gibi sorunları gidermiştir. Kalıbın genel yapısı Şekil 4.4'te gösterilmiştir.



Şekil 4.4 MVI yapısı

MVI kalıbındaki katmanlar aşağıdaki maddeler halinde özetlenebilir.

- Model, diğer katmanlardaki gibi uygulamadaki verilerden sorumlu olan ve veriyi sağlayan tek katmandır.
- Arayüz ise kullanıcının etkileşime girdiği, kullanıcı aksiyonlarının tanımlandığı ve model tarafından belirtilen durumların gösterildiği katmandır.
- Amaç katmanı ise kullanıcı ya da uygulama tarafından alınacak aksiyonları temsil eden katmandır.

MVI kalıbının diğer kalıplardan en büyük farkı gerçekleşebilecek olan aksiyonları tespit edip her aksiyon için gerekli sınıfları oluşturmasıdır. Ayrıca bu aksiyonlar arası tek yönlü bir akış vardır. Böylece uygulamanın genel işleyişi netleşir, uygulamadaki durumlar arası geçişler anlaşılır olur ve kodda yapılacak akış değişiklikleri kolaylaşır.

Kalıbın avantajları olarak

- Uygulamadaki akışlar önceden belirlendiği için anlaşılabilirlik ve bakım yapılabilirlik yüksektir.
- Tek yönlü ve döngüsel akış olması nedeniyle uygulamada beklenmeyen durumların olmasının olasılığı düşüktür.
- Modelin değiştirilemez olması hem iş parçacığı güvenliği (thread safety) hem de veri güvenilirliği sağlar.

gibi maddeler örnek verilebilirken dezavantajlar olarak aşağıdaki maddeler örnek verilebilir.

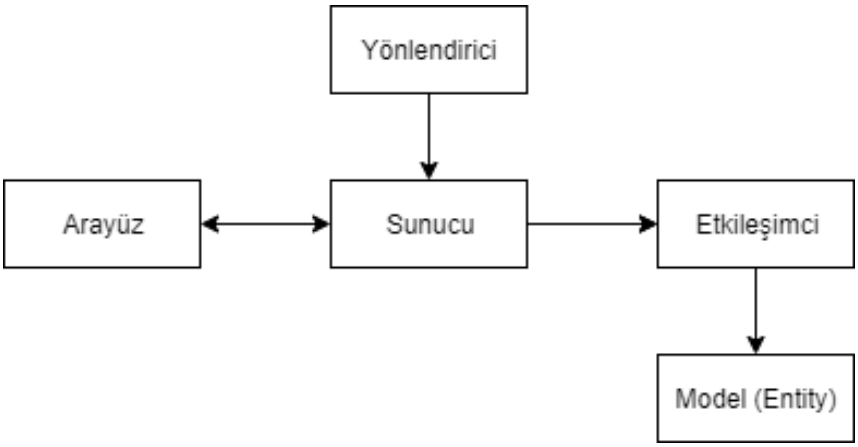
- Uygulamadaki her durum için sınıfların oluşturulması aşırı sınıf sayısına neden olur.
- Uygulamadaki akışlar parçalanabilir değilse uygulanması zorlaşır.
- Uygulamanın analizi iyi yapılmayıp akışlar ve durumlar iyi belirlenmezse sorunlara yol açabilir.

4.1.3.5 VIPER

VIPER tasarım kalıbı; arayüz, etkileşimci, sunucu, varlık ve yönlendirici bileşenlerinden oluşur. Kalıbın amacı, sorumlulukları fazla parçaya ayırarak daha iyi bir biçimde dağıtmaktır. Bu sayede her bir sınıfın üzerine düşen sorumluluk azaldığı için sınıftaki kod yoğunlunu azalacak, kodun anlaşılabilirliği ve bakım yapılabilirliği artacaktır. Kalıbın genel yapısı Şekil 4.5'te gösterilmiştir.

VIPER kalıbındaki katmanlar aşağıdaki maddeler halinde özetlenebilir.

- Arayüz katmanı kullanıcının etkileşime girdiği ve kullanıcıya verilerin gösterildiği katmandır.
- Etkileşimci, iş mantığını bulunduran katman olup sunucu katmanı ve uygulama verileri arasında köprü görevi görür ve sunucunun üstündeki sorumluluğu alır.



Şekil 4.5 VIPER yapısı

- Sunucu, kullanıcı arayüzünden gelen istekleri işler. Bu isteklerin sonucu ya interactor katmanından veri istenir ya da kullanıcı farklı ekranlara yönlendirilir.
- Varlık katmanı, uygulamadaki verileri ve veri işlerini temsil eden katmandır.
- Yönlendirici katmanı, uygulamadaki sayfa geçişlerinin yapıldığı katmandır.

Android projelerinden VIPER mimarisi kullanılırken yönlendirici katmanı dahil edilmeyebilir. Bunun yerinde Android platformunda dahili olarak bulunan yönlendirme mekanizmaları kullanılır. Böylece tasarımdaki karmaşıklık ve sınıf sayısı azaltılabilir.

Kalıbın avantajları olarak

- Yazılım kodunu birçok parçaya böldüğünden anlaşılabilirlik ve bakım yapılabilirlik fazladır.
- Sorumluluklar daha fazla ayrıldığından geliştiriciler aynı anda birbirlerinden bağımsız olarak daha rahat çalışabilir.
- Sınıflarda kod yoğunlukları azdır.
- Her bir parça ayrı olduğundan test edilebilirlik fazladır.

gibi maddeler örnek verilebilirken dezavantajlar olarak aşağıdaki maddeler örnek verilebilir.

- Daha fazla katman olması oluşturulacak ve bakımı yapılacak daha fazla sınıf anlamına gelir. Bu da zaman maliyetini artırır.

- Uygulamanın iyi analiz edilmesi gerekir aksi halde tasarım gereğinden fazla karmaşıklaşır.
- Hızlı geliştirmeye elverişli değildir.
- Tecrübesiz yazılımcılar için öğrenilmesi ve uygulanması zordur.

5
YÖNTEM VE BULGULAR

Yazılım projeleri için maliyetleri ve riskleri düşük seviyede tutmanın birçok yolu vardır. Aşağıdaki maddeler bu yollara örnek olarak verilebilir.

- Geliştirme süresini kısaltmak
- Bakım yapılabilirliği artırmak
- Hataları en aza indirmek
- Güvenilirliği artırmak

Bu tür süreçlerin maliyet ve risk açısından büyük etkisi vardır. Bu noktada kod kalitesini yüksek tutmak ve uygun yazılım mimarisi kalıplarını seçmek yazılım projeleri için çok önemli görevlerdir.

Yapılan çalışmada MVP (Model-View-Presenter) ve MVVM (Model-View-ViewModel) adlı iki farklı mimari yazılım kalıbı ile yazılmış örnek Android projesi için Bölüm 3'te açıklanan metriklerin ölçümleri yapılıp sonuçları gözlemlenmiştir. Her gözlem bölümündeki metrik sonuçları yorumlanmıştır.

5.1 Projenin Seçilmesi

Çalışmayı gerçekleştirebilmek için bir uygulamanın işlev ve nitelik farkı minimum düzeyde olan iki farklı mimari tasarım kalıbıyla Android platformu için yazılmış bir yazılım kaynak koduna ihtiyaç duyulmuştur. Her iki versiyonun arasındaki farkın minimum düzeyde olması, tasarım kalıplarının arasındaki saf farkı ve kaliteye olan bireysel ve net etkileri için önemlidir.

Tasarım kalıplarının çok katı kuralları olmadığı için uygulanması da kişiden kişiye veya projeden projeye göre değişkenlik gösterebilir. Tecrübesiz bir geliştirici bir

yazılım projesi için uyguladığı tasarım kalıbını uygun bir biçimde uygulayamabilir ve bu proje ilerledikçe ortaya çıkabilir. Özellikle ilerleyen evrelerde ortaya çıkan bu sorunların düzeltilmesi çok maliyetli olabilir. Aynı şekilde tasarım kalıplarının yanlış biçimde uygulanması, eğer projenin tanımı, gereksinimleri iyi algılanıp analiz edilmediyse tecrübeli bir geliştiricinin de başına gelebilir. Buradaki yanlış uygulama aslında en iyi senaryonun dışına çıkılması, böylece bileşenlerin yeterli veya düzgün bir şekilde tanımlanmaması ve bileşenlerin birbirleriyle iletişime geçmesindeki hatalardan oluşur. Bu tarz sorunların en aza indirgenmesi için yazılan kaynak kodunun bir veya daha fazla bilgili kişiler tarafından incelenmesi, eğer varsa sorunların veya daha iyi uygulanabilirliklerin belirtilmesi ve mümkünse yazan kişi tarafından bu isteklerin uygulanması istenir.

Yukarıdaki kriterler göz önüne alınarak bu tez çalışmasında kullanılmak ve incelenmek üzere Android platformu için iki farklı mimari tasarım kalıbıyla yazılmış projeler araştırılmıştır. Yazılım kaynak kodlarının herkese açık bir şekilde paylaşılması ve isteyen kişilerin sorunları belirtip isteyen kişilerin düzeltme isteklerini belirtebileceği projelerden olması yapılacak çalışmanın kalitesini artıracağından seçilecek projenin Github [64] platformu üzerinden alınmasının daha iyi olacağı yargısına ulaşılmıştır. Bazı bulunan aday projeler elendikten sonra bu tez çalışmasında kullanılacak olan yapılacaklar (To-Do) projesi [65] seçilmiştir. Bu proje, Google tarafından Android uygulama geliştiricileri için Android platformunda mimari kalıpların nasıl uygulanması gerektiğine dair bir rehber niteliğinde oluşturulmuştur. Bu projenin birden fazla versiyonu olmasına rağmen bu çalışmada sadece MVP ve MVVM versiyonları dikkate alınmıştır. Ayrıca projenin Google tarafından rehber niteliğinde paylaşılması da mimari tasarım kalıplarının iyi bir biçimde uygulandığından ve kaliteye olan etkilerinin daha net bir biçimde ölçüleceğinin göstergesi olarak düşünülmüştür.

5.2 Ortamın Seçilmesi

Yazılım kodlarını hızlı bir şekilde okumak, analiz etmek ve çalıştırmak için entegre bir geliştirme ortamına ihtiyaç duyulmuştur. Bu geliştirme ortamlarının Android platformunda en bilinenleri ve kullanılanları Eclipse [66] ve Android Studio [67]'dur. Bu çalışma için; popülerliği, kullanım kolaylığı, yetenekleri ve metrik ölçüm eklentilerine sahip olması nedeniyle Android Studio seçilmiştir.

5.3 Metriklerin ve Metrik Ölçüm Araçlarının Seçilmesi

Bu tez çalışmasında, yazılım kod kalitesi metrikleri bölümünde verilen birçok metrik dikkate alınıp ölçülmüştür. Metriklerin ölçümü için kolaylık olması açısından entegre çalışma ortamı uygulaması üzerinde çalışabilecek metrik araçları seçilmeye çalışılmıştır. Metrik ölçümleri için bir ölçüm aracı yeterli olsa da hem ölçüm araçlarının hatalı ölçüm yapması riskine karşın hem de ölçümlerin birkaç farklı yolla yapılabilmesinden kaynaklı, çıkan metrik sonuçlarının doğruluğunu ve tutarlılığını görmek amacıyla birden fazla ölçüm aracının kullanılmasının daha iyi olacağı kanaatine varılmıştır. Yapılan çalışmada kullanılan metrik araçları aşağıda listelenmiştir.

- MetricsTree
- MetricsReloaded
- JaSoMe

5.3.1 MetricsTree

MetricsTree, Java kodunun nicel özelliklerini değerlendirmek için bir entegre geliştirme ortamı (IDE) eklentisidir. Proje, paket, sınıf ve yöntem seviyelerinde en yaygın metrik setlerini ölçer. Daha önce bahsedilen yazılım kalite ölçütlerini değerlendirmek için kullanılan araçlardan biridir. Sonuçları direkt olarak IDE üzerinde göstermesi ve referans eşik değerlerini vermesi açısından avantajlıdır. Aracın dikkat edilmesi gerek noktası ise test sınıflarını hesaplamaya dahil etmemesidir.

5.3.2 MetricsReloaded

MetricsReloaded eklentisi, MetricsTree gibi yazılım kod kalitesi metriklerinin ölçülmesinde ve değerlendirilmesinde yardımcı olan bir entegre geliştirme ortamı (IDE) eklentisidir. Sonuçları direkt olarak IDE üzerinde göstermesi ve referans eşik değerlerini vermesi açısından avantajlıdır.

5.3.3 JaSoMe

JaSoMe (Java Source Metrics) programı haricen çalıştırılan bir yazılım kod kalitesi ölçüm aracıdır. Sonuçları bir XML dosyası olarak belirlenen lokasyona çıkartır.

5.4 Metriklerin Ölçümü

Ölçüm araçları seçim bölümünde bahsedilen üç metrik ölçüm aracıyla, projenin [65] her iki versiyonu da proje düzeyinde, paket düzeyinde, sınıf düzeyinde ve yöntem düzeyinde metrikler açısından incelenmiştir. Bu araçlar ile metrikleri hesaplamak için herhangi bir ek yapılandırma, dosya veya programa ihtiyaç duyulmamıştır. Ölçüm sonuçları için ana araç olarak MetricsTree aracı kullanılmış olup diğer araçlar sadece sonuçları kontrol amaçlı kullanılmıştır.

İlk adım olarak, proje düzeyindeki metrikler incelenmiştir. Üç ölçüm aracı, sürümlerin kök klasörleri seçilerek çalıştırılmıştır. Çıkan metrik sonuçları incelenip ve kaydedilmiştir. Farklı araç sonuç metrikleri karşılaştırılıp çıkan bazı ufak farklılar tespit edilmiş bu farklılıkların nedenleri araştırılmıştır.

Bir sonraki adımda, paket düzeyinde metrikler incelenmiştir. Her bir paket için tüm metrik araçları ile ölçüm yapılıp sonuçlar karşılaştırılmıştır.

Üçüncü adım olarak her bir paket için benzer işleve sahip olan sınıflar seçilip bu sınıflar üzerinde sınıf seviyesi metrikler kullanılarak ölçüm yapılmıştır.

Dördüncü adım olarak, benzer işleve sahip olan belirli sayıda metot seçilip metot seviyesi metrikler kullanılarak bu metotların ölçümleri yapılmıştır.

5.5 Sonuçların Karşılaştırılması ve Yorumlanması

Metrik sonuçları proje, paket, sınıf ve metot seviyesi olarak toplam 4 aşamada karşılaştırılıp yorumlanmıştır.

5.5.1 Proje Seviyesi Ölçümlerin Karşılaştırılması

Proje seviyesi metrikleri olarak, MOOD metrikleri, sayısal değer elde edilebilen ve proje geneli skorlar verdiği için QMOOD kalite metrikleriyle hesaplanan QMOOD kalite nitelikleri, ve tasarım boyutları ile alakalı metrikler ölçülmüştür. Ölçüm sonuçları Tablo 5.1'de verilmiştir.

Tablo 5.1 Proje seviyesi ölçüm sonuçları

Metrik / Kalite Niteliği	MVP	MVVM
Etkinlik	2.07	1.88
Genişletilebilirlik	1.09	0.56
Esneklik	3.66	3.03

Tablo 5.1 Proje seviyesi ölçüm sonuçları (devamı)

İşlevsellik	2.81	2.32
Yeniden Kullanılabilirlik	2.24	1.51
Anlaşılabilirlik	-3.81	-3.68
Nitelik Gizleme Faktörü	%90.61	%71.58
Nitelik Kalıtım Faktörü	%91.83	%89.99
Bağlılık Faktörü	%4.43	%5.51
Metot Gizleme Faktörü	%21.28	%29.02
Metot Kalıtım Faktörü	%2.24	%8.01
Çok Biçimlilik Faktörü	%128.76	%119.29
Soyut Sınıf Sayısı	1	2
Somut Sınıf Sayısı	59	63
Arayüz (Interface) Sayısı	19	8
Statik Sınıf Sayısı	13	4

Proje düzeyindeki metriklerin sonucu incelendiğinde aşağıdaki sonuçlara ulaşılmıştır.

- MVP tasarım kalıbında bulunan "BasePresenter", "BaseView" ve her pakette bulunan kontrat arayüzlerinin uygulanması (implement) sınıfı veya sınıfları birer alt tip yaptığı için bir kalıtım işlemidir. Kalıtım paradigması, sınıfların değişkenlerini ve metotlarını alt sınıflara aktararak yeniden kullanılabilirliği ve genişletilebilirliği artıran bir nesne tabanlı programlama paradigmasıdır. İki versiyon arasındaki etkinlik, genişletilebilirlik, esneklik ve yeniden kullanılabilirlik faktörleri metriklerinin farkları ise kalıtımın etkisi olarak düşünülmektedir [68–70].

- İşlevsellik kalite niteliği; tasarımın boyutuna, hiyerarşisine, polimorfizmine bağlıdır [40]. Bu tasarım özelliklerinin MVP tasarım kalıbıyla yazılmış versiyonda daha yüksek değere sahip olduğu görülmektedir.

- Anlaşılabilirlik kalite niteliği; soyut yapıların kullanımı, tasarım boyutu, bağımlılık, karmaşıklık ile ters orantılı iken kapsülleme, kohezyon, bakım yapılabilirlik ve test edilebilirlikle de doğru orantılıdır [40, 71–73]. MVP tasarım kalıbındaki kontrat arayüzlerinin kullanımı, hem tasarım boyutunu büyütürken hem de soyutlamayı arttırdığı için kodun anlaşılabilirliğini azaltmış ve dolayısıyla anlaşılabilirlik metriğinin düşmesine neden olmuştur [40].

- Metot kalıtım faktörü ve bağlılık faktörünün MVVM kalıbı ile yazılmış versiyonda daha fazla çıkmasının nedeni "databinding" özelliğinin

kullanımından kaynaklıdır. Bu durum test edilebilirliği artırırken diğer yandan değiştirilebilirliği azaltır [8].

5.5.2 Paket Seviyesi Ölçümlerin Karşılaştırılması

Paket seviyesi metrikleri olarak Robert C. Martin tarafından sunulan metriklerin her iki versiyon için kaynak kodundaki tüm paketlerin ölçümü yapılmıştır.

Ölçümleri daha iyi değerlendirebilmek adına MetricsTree adlı ölçüm aracının paket seviyesi metrikler için verdiği referans değer aralıkları Tablo 5.2'de verilmiştir.

Tablo 5.2 Paket seviyesi metriklerin referans değer aralıkları

Metrik	Değer Aralığı
Ce	[0,7)
Ca	[0, 8)
I	[0, 1]
A	[0, 1]
D	[0, 0.5]

İlk aşamada yazılım kodlarını barındıran ve daha üst paketlerde bu sınıfların bulunmadığı hiyerarşideki en alt paket olan "todoapp" paketinin için ölçümleri yapılmıştır. Bu sonuçlar Tablo 5.3'te verilmiştir.

Tablo 5.3 "Todoapp" paketinin ölçüm sonuçları

Metrik	MVP	MVVM
Ce	4	7
Ca	20	9
I	0.16	0.43
A	0.66	0.2
D	0.16	0.36

Daha sonraki aşamada, yazılımdaki "addedittask" adlı paket için paket seviyesi metrik ölçümleri yapılmıştır. Sonuçlar Tablo 5.4'te verilmiştir.

Tablo 5.4 "Addedittask" paketinin ölçüm sonuçları

Metrik	MVP	MVVM
Ce	8	5
Ca	3	3

Tablo 5.4 "Addedittask" paketinin ölçüm sonuçları (devamı)

I	0.72	0.62
A	0.5	0.25
D	0.22	0.12

Bir sonraki aşama olarak "data" adlı paketin her MVP ve MVVM versiyonları için paket seviyesi metrik ölçümleri yapılmıştır ve bu sonuçlar Tablo 5.5'te verilmiştir.

Tablo 5.5 "Data" paketinin ölçüm sonuçları

Metrik	**MVP**	**MVVM**
Ce	1	1
Ca	18	16
I	0.05	0.05
A	0.0	0.0
D	0.94	0.94

Sonraki aşamada "statistics" paketi için paket seviyesi metrik ölçümleri yapılmıştır ve bu sonuçlar Tablo 5.6'da verilmiştir.

Tablo 5.6 "Statistics" paketinin ölçüm sonuçları

Metrik	**MVP**	**MVVM**
Ce	9	5
Ca	1	1
I	0.9	0.83
A	0.5	0.0
D	0.4	0.16

Diğer aşamada, "taskdetail" paketi ölçülmüş ve sonuçlar Tablo 5.7'de verilmiştir.

Tablo 5.7 "Taskdetail" paketinin ölçüm sonuçları

Metrik	**MVP**	**MVVM**
Ce	9	7
Ca	1	2
I	0.9	0.7
A	0.5	0.25
D	0.4	0.02

Bir sonraki aşamada ise "tasks" paketinin paket seviyesi metrik sonuçları yapılmış olup bu sonuçlar Tablo 5.8'de verilmiştir.

Tablo 5.8 "Tasks" paketinin ölçüm sonuçları

Metrik	MVP	MVVM
Ce	18	19
Ca	0	1
I	1.0	0.95
A	0.4	0.22
D	0.4	0.17

Paket seviyesinde ölçümlerin son aşaması olarak "util" paketinin ölçümleri yapılmış olup bu sonuçlar Tablo 5.9'da verilmiştir.

Tablo 5.9 "Util" paketinin ölçüm sonuçları

Metrik	MVP	MVVM
Ce	0	0
Ca	8	11
I	0.0	0.0
A	0.0	0.0
D	1.0	1.0

Tüm paket sonuçları incelendiğinde aşağıdaki sonuçlara ulaşılmıştır.

- Ce metriği paketlerin dışa bağımlılığını belirten bir metriktir. Bu metriğin, "addedittask", "taskdetail", "statistics" gibi uygulamanın ana işlevlerini içeren sınıfları barındıran paketlerin MVP tasarım kalıbını kullanan versiyonunda daha yüksek çıktığını ve bundan dolayı da MVP versiyonundaki bu paketlerin, diğer kod paketlerine daha fazla erişme eğiliminde olduğunu ve bu nedenle değişme olasılığının daha yüksek olduğunu gösterir [46, 47].
- I metrik değeri, paketin değişkenliğini belirten bir metriktir. MVP versiyonunda daha yüksek değere sahip olduğu için MVP tasarım kalıbındaki paketler daha değişken veya kararsızdır [46, 47].
- A metriği paketlerdeki soyut yapıları belirten bir metriktir. MVP versiyonundaki paketlerin kontrat arayüzleri gibi soyut yapıları içermesi bu metriği yüksek çıkarmıştır [46, 47].

- D metriği paketlerin ideal soyutlama-değişkenlik çizgisine olan uzaklığını belirten bir metriktir ve küçük olması beklenir. MVVM versiyonundaki paketlerin ideal çizgiye olan uzaklıklarının daha az olduğunun ve daha iyi bir soyutlama-kararsızlık ilişkisi gösterdiği gözlemlenmiştir [46, 47].

5.5.3 Sınıf Seviyesi Ölçümlerin Karşılaştırılması

Sınıf seviyesi ölçümler için Bölüm 3'te açıklanan sınıf seviyesi metrikleri kullanılmıştır. Bu metrikler her iki versiyon için projenin ana işlevlerini yapan kodların barındığı sınıflar için ölçülmüştür. Tasarım kalıplarının yapısı gereği bir tasarım kalıbında bulunan sınıf diğer tasarım kalıbında bulunmayabilir, bu nedenle eğer aynı ada sahip sınıflar yoksa benzer işlevleri yerine getiren sınıflar karşılaştırılmaya çalışılmıştır.

5.5.3.1 Görev Listeleme İşlevinden Sorumlu Sınıflar

Uygulamanın ana işlevlerinden biri olan görevleri listeleme, filtreleme gibi işlevlerdir. İlk aşamada bu işlevin ana ekranını temsil eden "TasksActivity" sınıfının ölçümü yapılmıştır ve sonuçlar Tablo 5.10'da verilmiştir.

Tablo 5.10 "TasksActivity" sınıfının ölçüm sonuçları

Metrik	MVP	MVVM
WMC	12	20
DIT	8	10
CBO	12	17
RFC	30	44
LCOM	2	3
NOC	0	0
NOA	128	126
NOO	665	666
NOOM	3	4
NOAM	2	8
SIZE2	682	684
NOM	5	12
MPC	28	43
DAC	3	3
ATFD	1	3
NOPA	0	0
NOAC	0	0

Tablo 5.10 "TasksActivity" sınıfının ölçüm sonuçları (devamı)

WOC	1.0	1.0
TCC	0.4	0.10

Yine aynı ekrana bağlı, iç sayfa olan "TasksFragment" sınıfının ölçümü yapılmıştır ve sonuçlar Tablo 5.11'de verilmiştir.

Tablo 5.11 "TasksFragment" sınıfının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	36	21
DIT	2	2
CBO	18	14
RFC	74	49
LCOM	6	3
NOC	0	0
NOA	230	58
NOO	201	191
NOOM	6	6
NOAM	21	7
SIZE2	284	238
NOM	28	14
MPC	27	17
DAC	7	4
ATFD	0	1
NOPA	0	0
NOAC	0	1
WOC	1.0	0.92
TCC	0.07	0.48

En son aşamada, bu sayfaların iş mantığını yürüten, "TasksPresenter", "TasksViewModel" sınıfları ölçülmüş, sonuçları Tablo 5.12'de verilmiştir.

Tablo 5.12 "TasksPresenter" ve "TasksViewModel" sınıflarının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	38	31
DIT	1	2

Tablo 5.12 "TasksPresenter" ve "TasksViewModel" sınıflarının ölçüm sonuçları (devamı)

CBO	9	15
RFC	39	30
LCOM	1	2
NOC	0	0
NOA	4	13
NOO	37	31
NOOM	0	0
NOAM	14	11
SIZE2	41	44
NOM	15	12
MPC	35	48
DAC	3	7
ATFD	2	3
NOPA	0	6
NOAC	2	1
WOC	0.85	0.90
TCC	0.57	0.2

5.5.3.2 Görev Detayı İşlevinden Sorumlu Sınıflar

Uygulamada görevlerin detayını gösterme işlevi bulunmaktadır. Bu işlevin ana ekranını temsil eden "TaskDetailActivity" sınıfının ölçüm sonuçları 5.13'te verilmiştir.

Tablo 5.13 "TaskDetailActivity" sınıfının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	3	14
DIT	8	10
CBO	8	13
RFC	19	39
LCOM	2	3
NOC	0	0
NOA	126	128
NOO	662	663
NOOM	2	4
NOAM	0	5

Tablo 5.13 "TaskDetailActivity" sınıfının ölçüm sonuçları (devamı)

SIZE2	677	683
NOM	2	9
MPC	17	40
DAC	1	2
ATFD	2	8
NOPA	0	0
NOAC	0	0
WOC	1.0	1.0
TCC	0.0	0.08

İç sayfa olan, "TaskDetailFragment" sınıfının ölçüm sonuçları Tablo 5.14'te verilmiştir.

Tablo 5.14 "TaskDetailFragment" sınıfının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	24	12
DIT	2	2
CBO	11	9
RFC	48	32
LCOM	7	2
NOC	0	0
NOA	61	58
NOO	216	187
NOOM	5	6
NOAM	14	4
SIZE2	267	234
NOM	19	10
MPC	47	22
DAC	4	3
ATFD	5	3
NOPA	0	0
NOAC	0	1
WOC	1.0	0.9
TCC	0.18	0.48

En son aşama olarak, bu sayfaların iş mantığını yürüten, MVP için "TaskDetailPresenter", MVVM için "TasksDetailViewModel" sınıflarının ölçümü yapılıp sonuçları Tablo 5.15'te verilmiştir.

Tablo 5.15 "TaskDetailPresenter" ve "TasksDetailViewModel" sınıflarının ölçüm sonuçları

Metrik	MVP	MVVM
WMC	18	7
DIT	1	3
CBO	6	5
RFC	29	9
LCOM	1	1
NOC	0	0
NOA	3	9
NOO	25	40
NOOM	0	1
NOAM	7	3
SIZE2	28	49
NOM	8	5
MPC	33	4
DAC	3	1
ATFD	1	0
NOPA	0	0
NOAC	0	1
WOC	1.0	0.75
TCC	0.71	1.0

5.5.3.3 Görev Ekleme ve Düzenleme İşlevinden Sorumlu Sınıflar

Uygulama, işlevlerinden biri olan kullanıcının görev ekleme ve düzenlemesine olanak tanır. İlk aşamada bu işlevin ana ekranını temsil eden AddEditTaskActivity" sınıfının ölçümü yapılmıştır ve sonuçları Tablo 5.16'da verilmiştir.

Tablo 5.16 "AddEditTaskActivity" sınıfının ölçüm sonuçları

Metrik	MVP	MVVM
WMC	9	10
DIT	8	10
CBO	12	13

Tablo 5.16 "AddEditTaskActivity" sınıfının ölçüm sonuçları (devamı)

RFC	32	38
LCOM	3	4
NOC	0	0
NOA	129	127
NOO	665	660
NOOM	3	3
NOAM	2	4
SIZE2	683	679
NOM	5	7
MPC	29	36
DAC	3	2
ATFD	5	5
NOPA	0	0
NOAC	0	0
WOC	1.0	1.0
TCC	0.20	0.04

Yine aynı ekrana bağlı, iç sayfa olan "AddEditTaskFragment" sınıfının ölçümü yapılmıştır ve Tablo 5.17'de verilmiştir.

Tablo 5.17 "AddEditTaskFragment" sınıfının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	11	15
DIT	2	2
CBO	10	10
RFC	29	33
LCOM	4	2
NOC	0	0
NOA	59	58
NOO	201	187
NOOM	3	4
NOAM	7	5
SIZE2	250	234
NOM	11	10
MPC	22	30
DAC	3	4

Tablo 5.17 "AddEditTaskFragment" sınıfının ölçüm sonuçları (devamı)

ATFD	5	1
NOPA	0	0
NOAC	0	0
WOC	1.0	1.0
TCC	0.24	0.44

Bir sonraki aşamada ise bu sayfaların iş mantığını yürüten, MVP tasarım kalıbıyla yazılmış versiyonundaki "AddEditTaskPresenter", MVVM tasarım kalıbıyla yazılmış versiyonundaki "AddEditTaskViewModel" sınıflarının ölçümü yapılıp sonuçları Tablo 5.18'de verilmiştir.

Tablo 5.18 "AddEditTaskPresenter" ve "AddEditTaskViewModel" sınıflarının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	18	19
DIT	1	1
CBO	7	8
RFC	25	29
LCOM	1	1
NOC	0	0
NOA	4	10
NOO	28	26
NOOM	0	0
NOAM	9	11
SIZE2	32	36
NOM	10	12
MPC	24	27
DAC	3	6
ATFD	1	2
NOPA	0	4
NOAC	1	1
WOC	0.88	0.90
TCC	0.36	0.21

5.5.3.4 İstatistik İşlevinden Sorumlu Sınıflar

Uygulama, aktif veya tamamlanmış görevlerin sayısının gösterilmesi işlevine sahiptir. Bu işlevin kaynak kodlarının bulunduğu StatisticsActivity, StatisticsFragment ve MVP tasarım kalıbıyla yazılmış versiyonundaki StatisticsPresenter, MVVM tasarım kalıbıyla yazılmış versiyonundaki StatisticsViewModel sınıfları için metrik ölçümleri yapılmıştır ve sonuçlar her sınıf için ayrı tablolarda verilmiştir.

İlk olarak istatistik işlevinin ana ekranı olan StatisticsActivity sınıfının MVP ve MVVM versiyonları için sınıf seviyesi metriklerle ölçümleri yapılmış olup sonuçlar Tablo 5.19'da verilmiştir.

Tablo 5.19 "StatisticsActivity" sınıfının ölçüm sonuçları

Metrik	**MVP**	**MVVM**
WMC	9	15
DIT	8	10
CBO	11	12
RFC	23	32
LCOM	1	1
NOC	0	0
NOA	126	125
NOO	663	659
NOOM	2	2
NOAM	1	5
SIZE2	678	676
NOM	3	7
MPC	23	36
DAC	1	2
ATFD	1	1
NOPA	0	0
NOAC	0	0
WOC	1.0	1.0
TCC	1.0	0.14

Aynı ekrana bağlı, "StatisticsFragment" sınıfının ölçümü yapılmıştır ve Tablo 5.20'de verilmiştir.

Tablo 5.20 "StatisticsFragment" sınıfının ölçüm sonuçları

Metrik	MVP	MVVM
WMC	11	6
DIT	2	2
CBO	8	5
RFC	18	13
LCOM	4	3
NOC	0	0
NOA	57	56
NOO	197	183
NOOM	2	3
NOAM	6	3
SIZE2	244	228
NOM	8	6
MPC	20	7
DAC	2	2
ATFD	3	1
NOPA	0	0
NOAC	0	1
WOC	1.0	0.83
TCC	0.25	0.26

Bir sonraki aşamada ise bu sayfaların iş mantığını yürüten, MVP versiyonunda "StatisticsPresenter", MVVM versiyonundaki "StatisticsViewModel" sınıflarının ölçümü yapılıp sonuçları Tablo 5.21'de verilmiştir.

Tablo 5.21 "StatisticsPresenter" ve "StatisticsViewModel" sınıflarının ölçüm sonuçları

Metrik	MVP	MVVM
WMC	8	10
DIT	1	2
CBO	7	7
RFC	8	14
LCOM	1	1
NOC	0	0
NOA	2	7
NOO	16	26
NOOM	0	0

Tablo 5.21 "StatisticsPresenter" ve "StatisticsViewModel" sınıflarının ölçüm sonuçları (devamı)

NOAM	2	6
SIZE2	18	33
NOM	3	7
MPC	7	10
DAC	2	3
ATFD	1	2
NOPA	0	1
NOAC	0	0
WOC	1.0	1.0
TCC	0.0	0.46

Daha rahat yorum yapabilmek adına sonuçların aritmetik ortalamaları alınmış ve Tablo 5.22'de verilmiştir.

Tablo 5.22 Sınıf seviyesi metriklerinin aritmetik ortalamaları

Metrik	**MVP**	**MVVM**
WMC	16.4166	15
DIT	3.6666	4.66
CBO	9.9166	10.66
RFC	31.16	30.16
LCOM	2.75	2.16
NOC	0	0
NOA	77.41	49.25
NOO	298	293.25
NOOM	2.16	2.75
NOAM	7.08	6
SIZE2	323.66	318.66
NOM	9.75	9.25
MPC	26	26.66
DAC	2.916	3.25
ATFD	2.25	2.41
NOPA	0	0.83
NOAC	0.25	0.5
WOC	0.97	0.93
TCC	0.326	0.324

Elde edilen bu sonuçlara göre aşağıdaki sonuçlara ulaşılmıştır.

- CBO metriği incelendiğinde sınıfların, diğer sınıflara bağımlı olmasının MVVM'de yüksek olduğu [2, 48, 53] ve bunun da veri "databinding" özelliğinden kaynaklı olduğu düşünülmektedir [8].
- WMC, RFC, LCOM, NOAM, SIZE2, NOO, NOAM, NOM, WOC gibi sınıfın boyutu ve karmaşıklığı ile alakalı metriklerin [48–51] MVP ile yazılmış versiyonda daha yüksek çıktığı ve bu nedenle proje seviyesinde de yapılan MVVM tasarım kalıbının daha anlaşılabilir olduğu çıkarımı sınıf seviyesinde de yapılmıştır [53].
- RFC, WMC, NOO, DIT metriklerine bakılarak test edilebilirliğin MVVM tasarım kalıbıyla yazılmış versiyonda daha iyi olduğu görülmüştür [53, 73].
- Anlaşılabilirlik ve test edilebilirlikten yola çıkılarak bakım yapılabilirliğin de MVVM'de daha iyi olduğu çıkarımı yapılmıştır [74, 75].

5.5.4 Metot Seviyesi Ölçümlerin Karşılaştırılması

Uygulamadaki iş akışının sağlandığı metotların bir kısmı, Bölüm 3'te verilen metot seviyesi metrikler ile ölçümleri yapılmıştır.

İlk olarak "TasksPresenter" ve "TasksViewModel" sınıflarında bulunan ve görevlerin yüklenmesini sağlayan "loadTask" metodunun ölçümleri yapılmıştır ve sonuçlar Tablo 5.23'te verilmiştir.

Tablo 5.23 "LoadTask" metodu için ölçüm sonuçları

Metrik	MVP	MVVM
CND	1	1
LND	1	1
CC	14	12
NOL	1	1
LOC	64	59
LAA	1.0	1.0
FDP	0	0
NOAV	1	1
MND	5	5
CINT	3	2
CDISP	0.66	1.0

İkinci aşama olarak "TaskDetailPresenter" veya "TaskDetailViewModel" sınıflarında bulunan ve görevlerin silinmesini sağlayan "deleteTask" metodunun ölçümleri yapılıp sonuçları Tablo 5.24'te verilmiştir.

Tablo 5.24 "DeleteTask" metodu için ölçüm sonuçları

Metrik	MVP	MVVM
CND	1	1
LND	0	0
CC	2	2
NOL	0	0
LOC	9	9
LAA	1.0	1.0
FDP	0	0
NOAV	3	1
MND	1	1
CINT	4	2
CDISP	0.75	1.0

Üçüncü aşama olarak "TaskDetailPresenter" veya "TaskDetailViewModel" sınıflarında bulunan ve görevlerin güncellenmesini sağlayan "editTask" metodunun ölçümleri yapılıp sonuçları Tablo 5.25'te verilmiştir.

Tablo 5.25 "EditTask" metodu için ölçüm sonuçları

Metrik	MVP	MVVM
CND	1	1
LND	0	0
CC	2	2
NOL	0	0
LOC	8	5
LAA	1.0	1.0
FDP	0	0
NOAV	2	1
MND	1	1
CINT	3	1
CDISP	0.66	1.0

Dördüncü aşama olarak "AddEditTaskPresenter" ve "AddEditTaskViewModel" sınıflarında bulunan ve görevlerin oluşturulmasını sağlayan "createTask" metodunun metrik ölçümleri yapılıp sonuçları Tablo 5.26'da verilmiştir.

Tablo 5.26 "CreateTask" metodu için ölçüm sonuçları

Metrik	**MVP**	**MVVM**
CND	1	1
LND	0	0
CC	2	2
NOL	0	0
LOC	9	9
LAA	1.0	0.75
FDP	0	1
NOAV	5	7
MND	1	1
CINT	4	4
CDISP	0.75	0.8

Beşinci aşama olarak "AddEditTaskPresenter" ve "AddEditTaskViewModel" sınıflarında bulunan ve verilerin yüklemesi bittikten sonra çalışan "onTaskLoaded" metodunun metrik ölçümleri yapılmıştır. Sonuçlar Tablo 5.27'de verilmiştir. MVP ve MVVM'deki LOC yani satır sayısı metriği aynı çıkmasına rağmen MVVM'deki yorum satırı sayısı daha fazla olduğu için gerçek kod satır sayısı MVP'ye göre daha azdır.

Tablo 5.27 "OnTaskLoaded" metodu için ölçüm sonuçları

Metrik	**MVP**	**MVVM**
CND	1	0
LND	0	0
CC	2	1
NOL	0	0
LOC	9	9
LAA	0.5	0.66
FDP	1	1
NOAV	3	5
MND	1	1
CINT	5	4
CDISP	0.4	0.75

Altıncı aşama olarak "StatisticsPresenter" ve "StatisticsViewModel" sınıflarında bulunan ve istatistik verilerinin yüklenmesini sağlayan "loadStatistics" metodunun metrik ölçümleri yapılmıştır. Sonuçlar Tablo 5.28'de verilmiştir. MVVM'deki metot iki ayrı metoda ayrıldığı için tek metot olarak birleştirilip ölçüm yapılmıştır.

Tablo 5.28 "LoadStatistics" metodu için ölçüm sonuçları

Metrik	**MVP**	**MVVM**
CND	1	1
LND	1	1
CC	6	4
NOL	1	1
LOC	41	40
LAA	1.0	1.0
FDP	0	0
NOAV	1	1
MND	4	4
CINT	3	2
CDISP	0.66	1.0

Daha rahat yorum yapabilmek adına sonuçların aritmetik ortalamaları alınmış ve Tablo 5.29'da verilmiştir.

Tablo 5.29 Tüm metotların metriklerinin aritmetik ortalaması

Metrik	**MVP**	**MVVM**
CND	1	0.83
LND	0.33	0.33
CC	4.66	3.83
NOL	0.33	0.33
LOC	23.33	21.83
LAA	0.916	0.901
FDP	0.166	0.33
NOAV	2.5	2.66
MND	2.16	2.16
CINT	3.66	2.5
CDISP	0.646	0.92

Elde edilen bu bilgilere göre aşağıdaki sonuçlara ulaşılmıştır.

- MVVM kalıbında bulunan metotların LOC metriğinin daha düşük olması, daha az kodun olduğu anlamına gelmektedir. Bu da anlaşılabilirlik ve bakım yapılabilirlik açısından avantajdır [74].

- CDISP ve FDP metriklerinin MVP tasarım kalıbı ile yazılmış versiyonda daha düşük olması nedeniyle değişkenlere erişimin veya metot çağırımlarının daha çok sınıf içinde veya daha az sınıfa bağımlı olduğu görülmüştür [51].
- CINT metriğine göre MVP kalıbı ile yazılmış versiyonda bir metot içerisinden yapılan çağırımların sayısının MVVM kalıbı ile yazılmış versiyondaki çağırımların sayısına göre daha fazladır [51].
- MVP tasarım kalıbı ile yazılmış versiyonun karmaşıklık (CC) ve CND metriğinin yüksek olması, daha fazla veya daha derin döngü ve koşullu ifade gibi kontrol mekanizmalarına sahip olduğu anlamına gelir. Bu da anlaşılabilirlik, bakım yapılabilirlik, değiştirilebilirlik ve test edilebilirliği düşürür [56, 76, 77].

6
SONUÇ VE ÖNERİLER

6.1 Sonuç

Yazılımlar için geliştirme süreçleri, yaşam döngüleri boyunca çok önemlidir. Geliştirme sürecindeki atılan adımlar ne kadar ileriye dönük ve iyi olursa hem geliştirme hem de ilerleyen zamanlarda gerçekleşecek süreçlerdeki işlemlerin, tamamlanma süresi kısalır ve yapılması kolaylaşır.

Bu tez çalışmasında mimari yazılım örüntüleri ile yazılım kalitesinin ilişkisi Android platformundaki bir proje ile incelenmiştir. Yapılan bu çalışmada Android platformu için tasarlanan ve iki farklı tasarım kalıbıyla yazılan yapılacaklar uygulamasının [65] proje, paket, sınıf ve metot seviyesi metrikleri ile ölçülmüş, incelenmiş ve karşılaştırılmıştır. Görülmektedir ki farklı mimari tasarım kalıplarının yazılım kalitesi üzerinde farklı etkileri bulunmaktadır. Bir mimari tasarım kalıbı bütün ihtiyaçlara çözüm olamamakta ve çalışmada kullanılan tüm metriklerin sonucunu iyileştirememektedir.

6.2 Öneriler

Her projenin doğası gereği farklı ihtiyaçları olabilmektedir. Bu nedenle, yazılımlarda mimarisel tasarım kalıbı seçerken hem projenin hem de geliştiricilerin ihtiyacına göre mimarisel tasarım kalıbı bulunmalı ve örnek metrik sonuçları incelenmelidir çünkü bu ufak da olsa bir fikir verebilir.

Bir mimarisel tasarım kalıbı bir projeye çok iyi uyarken bir başka projeye iyi uymayabilir ve bazı sorunlar meydana getirebilir. Ayrıca tasarım kalıplarının da bir öğrenme eğrisi bulunduğundan geliştiricilerin öğrenip uyum sağlayabilmesi, doğru ve efektif bir şekilde uygulayabilmesi de gerekmetedir. Aksi taktirde proje ihtiyaçlarına iyi cevaplar verebilen bir mimarisel tasarım kalıbı kötü uygulandığı için beklenen etkiyi veremeyecektir.

6.3 Gelecek Çalışmalar

Mimarisel tasarım kalıplarının yazılım kalitesine etkisinin ölçüldüğü metriklerin çoğaltılması, karşılaştırılan tasarım kalıplarının çoğaltılıp farklı mimarisel tasarım kalıplarının kaliteye olan etkisinin incelenmesi, ayrıca ölçüm ve karşılaştırılma işlemlerinin otomasyonunun yapılması, böylece metriklerin tek tek karşılaştırılması yapılıp zaman harcanmadan kolaylıkla karşılaştırılma yapılması planlanmaktadır.

KAYNAKÇA

[1] H. Krasner, "The cost of poor software quality in the us: A 2020 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021.

[2] E. Ural, T. Umut, B. Feza, "Nesneye dayalı yazılım metrikleri ve yazılım kalitesi," *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu*, 2008.

[3] E. B. Belachew, F. A. Gobena, S. T. Nigatu, "Analysis of software quality using software metrics," *International Journal on Computational Science & Applications (IJCSA)*, vol. 8, no. 4/5, 2018.

[4] ISO/IEC, *ISO/IEC 9126-1. Software engineering – Product quality*. ISO/IEC, 2001.

[5] J. A. McCall, P. K. Richards, G. F. Walters, "Factors in software quality. volume i. concepts and definitions of software quality," GENERAL ELECTRIC CO SUNNYVALE CA, Tech. Rep., 1977.

[6] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. MacLeod, M. Merritt, *Characteristics of software quality*, 1978.

[7] *Iso 9126 software quality characteristics, an overview of the iso 9126–1 software quality model definition*. [Online]. Available: `http://www.sqa.net/iso9126.html` (visited on 04/25/2022).

[8] T. Lou *et al.*, "A comparison of android native app architecture mvc, mvp and mvvm," *Master's Thesis, Eindhoven: Eindhoven University of Technology*, 2016.

[9] G. E. Krasner, S. T. Pope, *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[10] Y. Zhang, Y. Luo, "An architecture and implement model for model-view-presenter pattern," in *2010 3rd international conference on computer science and information technology*, IEEE, vol. 8, 2010, pp. 532–536.

[11] J. Kouraklis, "Mvvm as design pattern," in *MVVM in Delphi*, Springer, 2016, pp. 1–12.

[12] N. Akhtar, S. Ghafoor, *Analysis of architectural patterns for android development*, 2021.

[13] K. Chauhan, S. Kumar, D. Sethia, M. N. Alam, "Performance analysis of kotlin coroutines on android in a model-view-intent architecture pattern," in *2021 2nd International Conference for Emerging Technology (INCET)*, IEEE, 2021, pp. 1–6.

[14] D. DOBREAN, L. DIOSŞAN, "A comparative study of software architectures in mobile applications.," *Studia Universitatis Babes-Bolyai, Informatica*, vol. 64, no. 2, 2019.

[15] V. Humeniuk, *Android architecture comparison: Mvp vs. viper*, 2019.

[16] Y. Haravy, S. Surto, "Modern approaches to android app architecture: Mvvm, mvp, viper," 2018.

[17] B. Wisnuadhi, G. Munawar, U. Wahyu, "Performance comparison of native android application on mvp and mvvm," in *International Seminar of Science and Applied Technology (ISSAT 2020)*, Atlantis Press, 2020, pp. 276–282.

[18] F. E. Shahbudin, F.-F. Chua, "Design patterns for developing high efficiency mobile application," *Journal of Information Technology & Software Engineering*, vol. 3, no. 3, p. 1, 2013.

[19] S. Magamadov, "Design patterns for mobile devices-a comparative study of mobile design patterns using android," M.S. thesis, 2020.

[20] A. Ampatzoglou, A. Kritikos, G. Kakarontzas, I. Stamelos, "An empirical investigation on the reusability of design patterns and software packages," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2265–2283, 2011.

[21] N.-L. Hsueh, P.-H. Chu, W. Chu, "A quantitative approach for evaluating the quality of design patterns," *Journal of Systems and Software*, vol. 81, no. 8, pp. 1430–1439, 2008.

[22] J. Aichberger, "Mining software repositories for the effects of design patterns on software quality," 2020.

[23] P. Pradhan, A. K. Dwivedi, S. K. Rath, "Impact of design patterns on quantitative assessment of quality parameters," in *2015 Second International Conference on Advances in Computing and Communication Engineering*, IEEE, 2015, pp. 577–582.

[24] A. Daoudi, G. ElBoussaidi, N. Moha, S. Kpodjedo, "An exploratory study of mvc-based architectural patterns in android apps," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1711–1720.

[25] T. Türk, "The effect of software design patterns on object-oriented software quality and maintainability," M.S. thesis, Middle East Technical University, 2009.

[26] M. İ. Akalın, "Yazılımın evrimleşme sürecinde tasarım örüntülerinin yazılım kalitesi üzerindeki etkilerinin incelenmesi," M.S. thesis, Trakya Üniversitesi Fen Bilimleri Enstitüsü, 2014.

[27] N. G. Alp, "Mobil platformlarda kaliteli kod geliştirilmesi ve maliyetin azaltılması," M.S. thesis, Beykent University, 2019.

[28] T. R. Chandrupatla, "Quality concepts," *Quality and reliability in Engineering*, vol. 5, pp. 50 271–8, 2009.

[29] I. 8402, *Quality vocabulary*, 1994.

[30] International Standard Organization (ISO), *Software Product Evaluation - Quality Characteristics and Guidelines for their Use. ISO/IEC IS 9126*. Geneva, Switzerland, 1991.

[31] ISO/IEC 25010, "ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models," Tech. Rep., 2011.

[32] *Mccall yazılım kalite modeli*. [Online]. Available: https://brtcntopcu.medium.com/mccall-yazilim-kalite-modeli-9dda06dd9660 (visited on 04/23/2022).

[33] *Software quality attributes*. [Online]. Available: https://www.sqa.net/softwarequalityattributes.html (visited on 05/03/2022).

[34] R. B. Grady, *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992.

[35] I. Jacobson, *The unified software development process*. Pearson Education India, 1999.

[36] P. Kruchten, *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[37] R. E. Al-Qutaish, "Quality models in software engineering literature: An analytical and comparative study," *Journal of American Science*, vol. 6, no. 3, pp. 166–175, 2010.

[38] T. A. Al-Rawashdeh, F. M. Al'azzeh, S. M. Al-Qatawneh, "Evaluation of erp systems quality model using analytic hierarchy process (ahp) technique," *Journal of Software Engineering and Applications*, vol. 2014, 2014.

[39] R. Dromey, "Cornering the chimera [software quality]," *IEEE Software*, vol. 13, no. 1, pp. 33–43, 1996. DOI: 10.1109/52.476284.

[40] J. Bansiya, C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[41] b333vv, *Metricstree/readme.md at master · b333vv/metricstree*, 2021. [Online]. Available: https://github.com/b333vv/metricstree (visited on 04/25/2022).

[42] F. B. Abreu, R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proceedings of the 4th international conference on software quality*, vol. 186, 1994.

[43] S. C. Misra, V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *International Conference on Computational Science and Its Applications*, Springer, 2003, pp. 724–732.

[44] F. B. e Abreu, W. Melo, "Evaluating the impact of object-oriented design on software quality," in *Proceedings of the 3rd international software metrics symposium*, IEEE, 1996, pp. 90–99.

[45] *Mood and mood2 metrics*. [Online]. Available: https://www.aivosto.com/project/help/pm-oo-mood.html (visited on 05/04/2022).

[46] R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, no. 1, pp. 151–170, 1994.

[47] *Object-oriented metrics by robert martin*. [Online]. Available: https://kariera.future-processing.pl/blog/object-oriented-metrics-by-robert-martin (visited on 06/10/2022).

[48] S. R. Chidamber, C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[49] M. Lorenz, J. Kidd, *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[50] W. Li, S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.

[51] M. Lanza, R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[52] J. M. Bieman, B.-K. Kang, "Cohesion and reuse in an object-oriented system," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 259–262, 1995.

[53] S. K. Dubey, A. Sharma, *et al.*, "Comparison study and review on object-oriented metrics," *Global Journal of Computer Science and Technology*, 2012.

[54] *Chidamber & kemerer object-oriented metrics suite*. [Online]. Available: https://www.aivosto.com/project/help/pm-oo-ck.html (visited on 05/04/2022).

[55] *Weight of a class*. [Online]. Available: https://dartcodemetrics.dev/docs/metrics/weight-of-class (visited on 06/17/2022).

[56] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[57] L. Bass, P. Clements, R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[58] *Design patterns*. [Online]. Available: https://refactoring.guru/design-patterns (visited on 05/25/2022).

[59] *Understanding structural design patterns*. [Online]. Available: https://lickability.com/blog/structural-design-patterns/ (visited on 05/10/2022).

[60] *Android architecture patterns part 1: Model-view-controller*. [Online]. Available: https://medium.com/upday-devs/android-architecture-patterns-part-1-model-view-controller-3baecef5f2b6 (visited on 05/10/2022).

[61] *Design patterns by tutorials: Mvvm*. [Online]. Available: https://www.raywenderlich.com/34-design-patterns-by-tutorials-mvvm (visited on 05/10/2022).

[62] *Livedata overview*. [Online]. Available: https://developer.android.com/topic/libraries/architecture/livedata (visited on 06/10/2022).

[63] *Data binding library*. [Online]. Available: https://developer.android.com/topic/libraries/data-binding (visited on 06/10/2022).

[64] *Github*. [Online]. Available: https://www.github.com (visited on 05/12/2022).

[65] *Android architecture blueprints*. [Online]. Available: https://www.github.com/android/architecture-samples (visited on 05/17/2022).

[66] *Enabling open innovation & collaboration | the eclipse foundation*. [Online]. Available: https://www.eclipse.org (visited on 05/17/2022).

[67] *Android studio*. [Online]. Available: https://developer.android.com/studio (visited on 05/17/2022).

[68] N. S. Gill, S. Sikka, "Inheritance hierarchy based reuse & reusability metrics in oosd," *International Journal on Computer Science and Engineering*, vol. 3, no. 6, pp. 2300–2309, 2011.

[69] *Inheritance (object-oriented programming)*. [Online]. Available: https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming) (visited on 05/10/2022).

[70] P. Li-Thiao-Té, J. Kennedy, J. Owens, "Assessing inheritance for the multiple descendant redefinition problem in oo systems," in *OOIS'97*, M. E. Orlowska, R. Zicari, Eds., London: Springer London, 1998, pp. 197–210, ISBN: 978-1-4471-1525-0.

[71] Y. Tashtoush, M. Al-Maolegi, B. Arkok, "The correlation among software complexity metrics with case study," *arXiv preprint arXiv:1408.4523*, 2014.

[72] N. Zighed, N. Bounour, A.-D. Seriai, "Comparative analysis of object-oriented software maintainability prediction models," *Foundations of Computing and Decision Sciences*, vol. 43, no. 4, pp. 359–374, 2018.

[73] P. R. Suri, H. Singhani, "Object oriented software testability (ooste) metrics analysis," *Int. J. Comput. Appl. Technol. Res*, vol. 4, no. 5, pp. 359–367, 2015.

[74] I. Heitlager, T. Kuipers, J. Visser, "A practical model for measuring maintainability," in *6th international conference on the quality of information and communications technology (QUATIC 2007)*, IEEE, 2007, pp. 30–39.

[75] C. Chen, M. Shoga, B. Li, B. Boehm, "Assessing software understandability in systems by leveraging fuzzy method and linguistic analysis," *Procedia Computer Science*, vol. 153, pp. 17–26, 2019.

[76] A. H. Watson, D. R. Wallace, T. J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*, 235. US Department of Commerce, Technology Administration, National Institute of ..., 1996, vol. 500.

[77] G. K. Gill, C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

TEZDEN ÜRETİLMİŞ YAYINLAR

Konferans Bildirisi

1. F. Özbay, O. Kalıpsız, "Investigation of the Effect of Architectural Design Patterns on Software Quality in Mobile Platforms," International Korkut Ata Scientific Researches Conference, 2022, pp. 830-836.