**T.C.**
**YILDIZ TEKNİK ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**

# YAZILIM KOD KALİTESİNİN İYİLEŞTİRİLMESİNDE YENİ YAKLAŞIMLAR

**Özge MUTLU**

YÜKSEK LİSANS TEZİ
Bilgisayar Mühendisliği Anabilim Dalı
Bilgisayar Mühendisliği Programı

Danışman
Prof. Dr. Oya KALIPSIZ

Mayıs, 2021

**T.C.**
**YILDIZ TEKNİK ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**

# YAZILIM KOD KALİTESİNİN İYİLEŞTİRİLMESİNDE YENİ YAKLAŞIMLAR

Özge MUTLU tarafından hazırlanan tez çalışması 21.05.2021 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 Yazılım Kod Kalitesinin İyileştirilmesinde Yeni Yaklaşımlar 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.

Özge MUTLU

İmza

*Aileme...*

# TEŞEKKÜR

Tez çalışmalarım boyunca destek ve yol göstericiliği için kıymetli tez danışmanım sayın Prof. Dr. Oya KALIPSIZ ‘a en kalbi duygularımla teşekkür ederim. Yüksek lisans sürecim boyunca hep yanımda olan başta Aytaç Avcı olmak üzere desteklerini esirgemeyen Elçin Güveyi ve Bahar Hacıimam ‘ a çok teşekkür ederim. Beni bu güne getiren aileme , her zaman yanımda oldukları , beni destekledikleri ve sevdikleri için ne kadar teşükkür etsem azdır. Bu tezi sevgili anne ve babama ithaf ediyorum. . .

Özge MUTLU

# İÇİNDEKİLER

# KISALTMA LİSTESİ

| | |
|---|---|
| BCS | Basic Control Structures |
| CBO | Coupling Between Object Classes |
| CK | Chidamber & Kemerer |
| CMMI | Capability Maturity Model Integration |
| CWU | The Unit Of Cognitive Weight |
| DB | Database |
| DIT | Depth of Inheritance Tree |
| EAL | Evaluated Assurance Level |
| ELOC | Effective Line Of Code |
| IDE | Integrated Development Environment |
| IEC | The International Electrotechnical Commission |
| ISO | International Organization for Standardization |
| KLOC | Thousand Line Of Code |
| LCOM | Lack of Cohesion in Methods |
| LOC | Lines Of Code |
| MI | Maintainability Index |
| NOC | Number of Children |
| NOM | Number Of Methods |
| RFC | Response for a Class |
| RTPA | Real Time Process Algebra |
| SDLC | Software Development Life Cycle |
| SLOC | Source Lines of Code |
| SP | Stored Procesdures |

| | |
|---|---|
| SQL | Structured Query Language |
| UML | Unified Modelling Language |
| WMC | Weighted Method per Class |

# ŞEKİL LİSTESİ

# TABLO LİSTESİ

# ÖZET

## Yazılım Kod Kalitesinin İyileştirilmesinde Yeni Yaklaşımlar

Özge MUTLU

Bilgisayar Mühendisliği Anabilim Dalı
Yüksek Lisans Tezi

Danışman: Prof. Dr. Oya KALIPSIZ

Teknolojinin hızla gelişmesi ve bu gelişmeyle beraber artan müşteri talepleri yazılım geliştirme sektöründeki baskıyı günden güne arttırmaktadır. Geliştirilen ürünü mümkün olduğunca çabuk piyasaya sürmek ortaya çıkan bu rekabet için büyük önem arz etmektedir. Oluşan bu zaman kısıtlaması ve artan taleplerle beraber hızla büyüyen projeler yazılan kodun kalitesinde düşüşlere ve oluşabilecek hataların artışına neden olmaktadır. Bu sorunların önüne geçebilmek için tüm geliştirme süreci dikkatli şekilde gözlemlenmeli , belirli standart ve ölçütlerle kod değerlendirmeleri yapılmalıdır.

Yazılımı değerlendirmek için kullanılan ölçütlerin hedef üzerinde büyük bir etkisi vardır, bu nedenle yazılımın gereksinimlerine göre hangi ölçütlerin daha verimli olacağına alınacak kararlar oldukça önemlidir. Ancak belirli bir hedef için çok sayıda ölçüm ,değerlendirmesi zor olabilecek farklı aralıklarda çeşitli değerler vereceğinden çok kafa karıştırıcı hale gelebilir. Hedefimiz tüm bu yönleri kapsayan bir ölçüt bulup, başarılı bir yazılıma doğru iyi bir adım atmaktır. Bu yazıda amacımız, iki nedenden ötürü tasarım aşamasında yazılımın karmaşıklığını ölçmektir: Karmaşıklık, yazılımın kalitesini yeniden kullanılabilirlik, anlaşılabilirlik ve bakım maliyeti gibi pek çok açıdan etkileyen önemli bir faktördür. Tasarım aşamasında karmaşıklığın ölçülmesi, bu aşamanın yeniden tasarım ve sürdürülebilirlik maliyetini ve çabasını azaltmadaki katkısı nedeniyle, kalitede birçok avantaj sağlayabilir.

Bu çalışmada yazılım kalitesi ve metrikleri incelenmiş bu metriklerden biri olan kod karmaşıklığının farklı hesaplamaları ele alınmış ,uygulanmış ve nasıl daha verimli hale getirilebilir sorusunun cevapları aranmıştır. Uygulamalar nesneye

yönelik programlama dillerindeki geniş kalite araçlarına sahip olmayan veri tabanı nesnelerinde yapılmıştır. Sonuç olarak karmaşıklığın bu nesnelerde ölçülebilir,kontrol edilebilir olduğu ispatlanmış ve mevcut karmaşıklık hesaplamalarından daha detaylı ve genişletilebilir bir modelin başarıyla kullanılabileceği görülmüştür.

**Anahtar Kelimeler:** Kod kalitesi, yazılım kalitesi, kod karmaşıklığı, sürdürülebilirlik

# ABSTRACT

## New Approaches to Improving Software Code Quality

Özge MUTLU

Department of Computer Engineering
Master of Science Thesis

Advisor: Prof. Dr. Oya KALIPSIZ

The rapid development of technology and the increasing customer demands with this development increase the pressure in the software development sector day by day. It is of great importance to launch the developed product as quickly as possible for this emerging competition. With this time constraint and increasing demands, rapidly growing projects cause a decrease in the quality of the written code and an increase in the errors that may occur. In order to avoid these problems, the entire development process should be carefully observed, and code evaluations should be made with certain standards and criteria.

The metrics used to evaluate the software have a large impact on the target, so it is very important to make decisions about which metrics will be more efficient based on the needs of the software. However, it can be very confusing as a large number of measurements for a given target will yield various values in different ranges that may be difficult to assess. Our goal is to find a benchmark that covers all these aspects and take a good step towards a successful software. In this article, our aim is to measure the complexity of the software at the design stage for two reasons: Complexity is an important factor that affects the quality of the software in many aspects such as reusability, understandability and maintenance cost. Measuring complexity during the design phase can yield many benefits in quality, as this phase contributes to reducing the cost and effort of redesign and sustainability.

In this study, software quality and metrics were examined, and different calculations of code complexity, which is one of these metrics, were discussed, applied and the answers to the question of how to make it more efficient were sought. Applications are

made in database objects that do not have the extensive quality tools in object-oriented programming languages. As a result, it has been proven that the complexity is measurable and controllable in these objects, and it has been seen that a more detailed and extensible model can be used successfully than the existing complexity calculations.

**Keywords:** Code quality, software quality, code complexity, sustainability

# 1
# GİRİŞ

## 1.1 Literatür Özeti

Yazılımın kalitesini iyileştirmek, sürdürülebilirliğini ve bakım kolaylığını arttırmak bunlarla beraber maliyetini düşürmek yazılım endüstrisinin önemli hedefleridir. Teknolojinin hızla gelişmesiyle ortaya çıkan daha kapsamlı, büyük ölçekteki yazılım projeleri ve beraberinde gelen sorunlar bu hedeflerin önemini bizlere açıkça göstermektedir. 2000 yılında toplam yazılım satışı yaklaşık 180 milyar dolara ulaşmıştır bu rakamın 2020 yılında 500 milyar dolara yaklaştığı tahmin edilmektedir. Gelişen bu pazar ve artan taleplerle birlikte yazılımların boyutları binlerce satır kodla değil, milyonlarca satır kodla ölçülür duruma gelmiştir. Hızla büyüyen bu yazılımlarla birlikte artan karmaşıklık kod kalitesinin belirli bir seviyede tutulmasını oldukça zorlaştırmaktadır [1].Kalitesiz yazılımların, Tricentis firmasının 2017 yılında yaptığı araştırmaya göre küresel ekonomiye verdiği zarar 1.7 Trilyon dolar civarındadır [2]. Ayrıca şirketler tarafından itibar kaybına neden olmaktadır. Zararın bu denli büyük olması karşılaşılabilecek hataların projenin erken safhalarında tespit edilmesine ve çözülmesine dolayısıyla yazılım kod kalitesine verilen önemi arttırmıştır.

Yazılım kod kalitesinin bir disiplin içerisinde tanımlanabilmesi için ölçme araçlarına, sağlıklı karşılaştırmaların yapılabilmesi için tanımlı referans noktalarına ihtiyaç vardır. Bu noktadaki ihtiyaçlar için yazılım metrikleri kullanılır [2]. Yazılım metrikleri ve kalite arasındaki ilişkiyi gösteren pek çok çalışma yapılmıştır. Bunlardan biri Denizbank'ın yazılım ihtiyaçlarına cevap veren Intertech firmasında yürütülen projeler üzerinde yapılmıştır. Çalışmada yazılım metriklerinden biri olan kod satır sayısı(Line Of Code) arttıkça bir diğer metrik olan karmaşıklığın da beraberinde arttığı ve bunlara bağlı olarak karşılaşılan hata sayısının yükseldiği gözlemlenmiştir [3].

IEEE Standart Bilgisayar Sözlüğü karmaşıklığı şu şekilde tanımlar: Bir sistemin veya bileşenin anlaşılması ve doğrulanması zor bir tasarım veya uygulamaya sahip olma derecesidir [4].Basili ise karmaşıklığı, belirli bir görevi yerine getirmek için bir yazılım parçasıyla etkileşim halindeyken bir sistem tarafından harcanan

kaynakların bir ölçüsü olarak tanımlar [5].Etkileşimli sistem bir program ise, karmaşıklık, kodlama, hata ayıklama, test etme veya yazılımı değiştirme gibi görevleri gerçekleştirmenin zorluğu olarak tanımlanır [6].Karmaşıklık, anlaşılabilirliğin bir ölçüsüdür ve anlaşılabilirliğin olmaması hatalara yol açar. Karmaşık bir sistemin belirlenmesi daha zor, tasarlanması daha zor, uygulaması daha zor, doğrulanması daha zor, çalıştırması daha zor, değiştirilmesi riskli ve / veya davranışını tahmin etmek daha zor olabilir. Karmaşıklık yalnızca insanın anlaşılabilirliğini etkilemez aynı zamanda "makinenin anlaşılabilirliği" söz konusudur. Örneğin, C gibi basit diller için statik analiz araçları, C ++ gibi daha karmaşık diller için olan araçlardan daha güçlüdür [7].Koddaki bu karmaşıklığın yazılım geliştirme sürecine olan etkilerini inceleyen bir çalışmada yazılım karmaşıklığıyla bakım maliyetleri arasındaki ilişki incelenmiştir. Windows, Debian ve Linux işletim sistemlerinde toplanan bazı veriler kullanılmıştır. Geliştirme maliyetlerini ve bakım maliyetlerini tahmin etmek için Yapıcı Maliyet Modeli (COCOMO) II maliyet tahmin modelinden yararlanılmıştır. Çalışmada işletim sistemlerinin yıllar içerisinde çıkan sürümlerinin kod satır sayıları ve karmaşıkları karşılaştırılmış ve bu değerlere göre bakım maliyeti hesaplanmıştır. İşletim sistemleri geliştikçe kod satır sayısının ve karmaşıklığın artmasıyla ilişkili geliştirme ve bakım maliyetlerinin arttığı gözlemlenmiştir [7].

Karmaşıklığı ölçmek ve kontrol altında tutmak tüm bu sorunların önüne geçebilmek için oldukça önemlidir. Bu kapsamda bir geliştiricinin yazılımı anlaması ve sürdürmesinin zorluğu olarak bilinen yapısal karmaşıklığın sistem ve modül düzeyinde ölçülmesi için kullanılan metotlar ve mevcut karmaşıklık metrikleri incelenmiştir [8]. Otomatik bir ölçüm ve değerlendirme sistemi kurmak bunu kolaylaştırır. Bu konuda yapılan araştırmalardan birinde, .Net içerisinde yazılmış tüm dillerdeki kodların derlenmesi sonucunda oluşan ortak çıktının oluşturulduğu dil olan İntermediate Language (IL) kullanılmıştır. Bu özelliği ile dilden nispeten bir bağımsızlık sağlayan bir karmaşıklık ölçümü çalışması yapılmıştır. Öncelikle metrikleri hesaplamak için programların akış grafiği oluşturulmuştur. Bu grafikler C# kodu incelenerek karşılaştırılmış, yeniden yapılandırılmış akış grafiklerinin ideal sonuçlara çok yakın olduğunu doğrulanmıştır. Elde edilen verilerden McCabe cyclomatic complexity metodu kullanılarak karmaşıklık hesabı yapılmıştır [9].

Karmaşıklık metriği ayrıca yapılacak olan testlere rehber niteliğindedir. Bir programdaki tüm yolları test etmek genellikle imkansızdır, çünkü programın sonsuz veya daha fazla sayıda yol içermesi mümkün olabilir. Daha yüksek hata oranlarına sahip yolları seçebilmek adına karmaşıklık metriği kullanılabilir. Ayrıca çıkan total karmaşıklığa göre test maliyetleri daha isabetli verilebilir. Yapılan testlerden biri regresyon testidir. Uygulanan programda değişiklik yapıldıktan sonra regresyon testi yapılır. Bu, değişikliklerin değişiklikten önce düzgün çalışan herhangi bir şeyi

etkileyip etkilemediğini belirlemek için mevcut test takımlarını değiştirilen koda göre yeniden çalıştırarak veya gerektiğinde yeni test senaryoları yazarak yapılabilir. Yapılan çalışmada yapılacak regresyon testi için her bir senaryonun akışı çıkarılmış ardından her bir yola ait karmaşıklık hesaplanarak ağırlıklandırılmıştır. Ağırlığı fazla yani karmaşıklığı fazla olan yollara daha fazla test maliyeti ayrılmıştır [10].

## 1.2 Tezin Amacı

Yazılım sektöründe hızla gelişen ve değişen ihtiyaçlara cevap vermeye çalışırken güvenilir, uygun maliyetli, standartlara uygun ve hatasız yazılıma sahip olmak büyük önem arz etmektedir. Çünkü oluşacak hataların etki alanı geniş olabileceğinden büyük sorunlara yol açabilir bu da insan hayatını tehlikeye atmak, iletişimi ve iletimi durdurmak veya engellemek, hatalı üretim, müşteri kaybı ya da iflas gibi olumsuzluklara neden olabilmektedir. Tüm bu sorunlardan kaçınmak için test bir yöntemdir. Test, yazılımın işlevselliği ve kalite özellikleri hakkında bilgi edinme ve hataları belirleme yöntemi olarak bilinir. Bununla birlikte, test aşamasında karşılaşılan / keşfedilen hataların çoğu geç bulgulardır. Çünkü üretim ortamında hatta öncesinde testte bulunan her hata SDLC sürecinin önceki aşamalardan yeniden çalıştırılmasına neden olur ve sırasıyla yeniden test edilmelidir. Bu durum, ciddi bir zaman ve maliyet kaybı demektir. Yazılımla ilgili hataların erken tespiti ve bu hataların geliştirme aşamasında tespit edilerek giderilmesi önemli tasarruflar sağlar. Amaç yazılım geliştirme sırasında hataları önleyerek ve düzelterek yazılım kalitesini iyileştirmektir.

Kaliteli bir yazılımın bakımı kolaylıkla yapılabilir olmalı, kolay anlaşılabilmeli, iyi yapılandırılmış olmalı ve geliştirmeye açık olmalıdır. Bunların yanı sıra kod satırı başına hata oranı ve yaşam döngüsü boyunca meydana gelen ve alınan hataların oranının en az seviyede olması gerekir. ISO 9126 kalite standardında yazılım bakım kolaylığı için, yazılımın güncellenebilir ve sürdürülebilir olması denilmektedir. Bahsedilen bu özellikler; hata düzeltmeleri, yazılımda yapılacak iyileştirmeler ve gelen taleplere yönelik yazılımın geliştirilmesidir. Yazılım ürünü bakım kolaylığı açısından istenilen seviyede değilse yapılacak olan bu değişikler oldukça masraflı hale gelmektedir [11].

Karmaşıklık, yazılımı okumayı ve anlamayı ve dolayısıyla değiştirmeyi zorlaştırır. Çıkan hataların temel nedenleri arasındadır. Tüm bunlar göz önüne alındığında karmaşıklığın yazılım kalitesinin tersi olduğunu söyleyebiliriz. Bu nedenle karmaşıklığı ölçmek ve belirli sınırlar arasında tutabilmek önemli rol oynamaktadır.

Bu tezde, yazılım kalitesi ve bu kalitenin temel bileşenleri üzerinde durulmuştur.

Yazılım metrikleri ve bu metriklerden biri olan karmaşıklık metriğinin önemini, yazılım kalitesini iyileştirmedeki rolünü ve tasarım aşamasında karmaşıklığı ölçmenin avantajlarını gösterdikten sonra yazılım karmaşıklığını ölçen modeller incelenmiştir. Amaç bu modellerin avantajları ve dezavantajları hakkında çıkarım yapıp hepsini kapsayan genel bir model oluşturmaktır. Elde edilen sonuçlar ve öneriler verilmektedir.

## 1.3 Hipotez

Tez kapsamında, yazılım kod kalitesini iyileştirmede yazılım metriklerinden biri olan karmaşıklık metriğinin etkili olduğu hipotezi ortaya atılmaktadır.

# 2
# YAZILIM KALİTESİ VE KALİTENİN DEĞERLENDİRİLMESİ

## 2.1 Yazılım Kalitesi

Yazılım kalitesi, kalitenin yazılıma entegre edilmesini sağlamak için planlanmış faaliyetler dizisidir. Tanımını yapacak olursak kalite, yazılımın veya sürecin belirtilen müşteri veya kullanıcı ihtiyaçlarını, beklentilerini ve gereksinimleri karşılama derecesidir. Bir müşterinin beklentisinin sıfır olması, özelliği sıfır olan bir ürünün kaliteli bir ürün olduğu anlamına gelmez. Yüksek kaliteli bir ürün, birçok kalite faktörüyle ilişkilendirilen bir üründür. Bunlar, gereksinim spesifikasyonunda açıklanabilir veya geliştiricinin önemli gördüğü ancak müşteri tarafından dikkate alınmayan ve dolayısıyla gereksinim spesifikasyonuna dahil edilmeyen kalite faktörleri olabilir. Kalite, bir ürün veya hizmetin, örneğin spesifikasyonlara uygunluk gibi, verilen ihtiyaçları karşılama kabiliyetine dayanan özellik ve özelliklerinin toplamıdır.

Yazılım kalitesi; teslimat, maliyet ve kalite gereksinimlerini karşılayan yazılımlar üretmeyi amaçlar. İşlevsel ve işlevsel olmayan olarak kategorize edilebilir. İşlevsel yazılım kalitesi, erken aşamalarda tanımlanan yazılım özelliklerini tanımlar. Bir yazılım projesinin ilk aşamalarında müşteri gereksinimleri toplanır ve yazılım kalitesi üzerine etkileri nedeniyle dikkatle ele alınmalıdır. İşlevsel olmayan yazılım kalitesi ise yazılım hizmetlerini destekleyen özellikleri içerir. Belirli bir geliştirmenin nasıl çalıştığı, hangi özellik ve özelliklere sahip olduğuyla ilgilidir.

Yazılım geliştirme sürecini kontrol etmede, yönetmede ve iyileştirmede kilit bir unsur, yazılım ölçümüdür. Analiz, tasarım ve kodlama aşamaları boyunca ürün geliştirmenin çıktıları, önceden belirlenmiş kriterlere göre doğrulanabilmeleri için ölçülmeli, gözlemlenmeli ve yönetilmelidir. Yazılım ölçümleri yapılarak ürün arızalarının önüne geçilebilir. Bu ölçümler geliştiricileri ve mühendisleri geliştirme hataları konusunda uyarır ve böylece ürün sürümünün hem öncesinde hem de ilk aşamalarında hataları ve kusurları önler ve ayrıca yazılım geliştirmenin izlenmesine yardımcı olur [12].

Bir yazılım sistemi için kalite değerlendirmesi yapılırken aşağıda sıralanan maddelerden faydalanılabilir.

- Kalite Özellikleri
- Kalite Modeli
- Standartlar
- Yazılım Metrikleri

### 2.1.1 Kalite Özellikleri

ISO 9126 standardında (ISO, 2001) yazılım kalitesi alt özellikleri bulunan 6 ana kalite özelliğine sahiptir. Bunlar; işlevsellik,güvenilirlik, kullanılabilirlik, verimlilik, taşınabilirlik ve sürdürülebilirlik.Bunlar dahili ve harici olarak gruplanabilir. Harici olanlar yani müşterinin yazılımdan beklediği dış kalite özellikleri ;İşlevsellik, güvenilirlik, kullanılabilirlik, verimlilik iken dahili olanlar yani geliştiricilerim bir yazılımda dikkate aldıkları kalite özellikleri ; sürdürebilirlik, taşınabilirlik, yeniden kullanılabilirlik ve test edilebilirliktir.Bu 6 ana kalite özelliği ve alt başlıkları aşağıda sıralanmıştır.

#### 2.1.1.1 İşlevsellik (Functionality)

Yazılımın, belirtilen koşullar altında kullanıldığında ihtiyaca göre işlevleri ve hizmetleri sağlaması gerektiği anlamına gelir. İşlevselliğin altındaki alt özellikler şunlardır:

- Uygunluk(Suitability): Yazılımın işlevlerinin uygunluğunu ifade eder.
- Doğruluk(Accurateness) : Fonksiyonların doğruluğunu ifade eder.
- Birlikte Çalışabilirlik(Interoperability) : Bir yazılım bileşeninin diğer bileşenler ve sistemlerle etkileşim kurma yeteneğini ifade eder.
- Uyma(Compliance) : Yazılımın uyumlu olma yeteneğini ifade eder.
- Güvenlik(Security) : Yazılıma yetkisiz erişimin olmamasını ifade eder.
- Olgunluk(Maturity) : Yazılımın arıza sıklığını ifade eder [13].

#### 2.1.1.2 Güvenilirlik (Reliability)

Güvenilirlik, yazılımın, belirli koşullar altında kullanıldığında, belirli bir hata toleransı seviyesini sürdürme yeteneği ve bu süre içinde arıza üretme olasılığıdır. Güvenilirlik aşağıdaki alt özelliklere ayrılmıştır:

- Hata Toleransı(Fault Tolerance) : Yazılımın hataya karşı dayanıklılığını, hatadan kurtarma yeteneğini ifade eder.
- Kurtarılabilirlik (Recoverability) : Yazılımın sistem, veri ve ağ bağlantıları dahil olmak üzere tam çalışabilir duruma geri getirilebilmesini ifade eder.
- Anlaşılabilirlik(Understandability) : Kolay anlaşılabilir olmasını ifade eder [13].

#### 2.1.1.3 Kullanılabilirlik (Usability)

Yazılımın belirli koşullar altında kullanıldığında anlaşılması, öğrenilmesi, kullanılması, yapılandırılması ve yürütülmesi yeteneğidir.Kullanılabilirliğin alt özellikleri aşağıdaki gibi tanımlanmıştır.

- Öğrenilebilirlik(Learnability): Yazılımın ve kullanımının kolay öğrenilebilmesini ifade eder.
- Çalışırlık(Operability) : Belirli bir ortamda ve bir kullanıcı tarafından kolayca çalıştırılabilmesini ifade eder [13].

#### 2.1.1.4 Verimlilik (Efficiency)

Bu özellik, bir yazılımın kullanılan kaynakların miktarına göre uygun performans sağlama yeteneğini ifade eder. Verimliliğin alt özellikleri aşağıdaki gibi tanımlanmıştır.

- Zaman Davranışı (Time Behavior) : Yazılımın yapılan işleme yanıt verme süresini ifade eder.
- Kaynak Davranışı(Resource Behavior) : Yazılımın kullandığı kaynak miktarını yani bellek, cpu, disk ve ağ kullanımını ifade eder.
- Analiz Edilebilirlik(Analyzability) : Yazılımda bir hatanın temel nedeninin belirleme kolaylığını ifade eder [13].

#### 2.1.1.5 Sürdürülebilirlik (Maintainability)

Yazılımın değiştirilme yeteneğini açıklar. Sürdürülebilirliğin alt özellikleri aşağıdaki gibi tanımlanmıştır.

- Değiştirilebilirlik(Changeability) : Bir yazılımın / sistemin kolay değiştirilebilir olmasını ifade eder.
- Kararlılık(Stability) : Sistem değişikliklerinin neden olabileceği ve olumsuz etkisi olan bir değişiklik karşısında sistemin değişikliğe duyarlılığını ifade eder.
- Test Edilebilirlik(Testability) : Bir yazılımın ya da sistemin test edilme kolaylığını ifade eder.
- Uyarlanabilirlik(Adaptability) : Yazılımın / sistemin yeni ortamlarda , işletim sistemlerinde çalışabilir olmasını ifade eder [13].

#### 2.1.1.6 Taşınabilirlik (Portability)

Bu özellik, yazılımın bir ortamdan diğerine gerektiğinde çok az değişiklik yapılarak aktarılabilmesi olarak tanımlanmaktadır. Taşınabilirliğin alt özellikleri aşağıdaki gibi tanımlanmıştır.

- Kurulabilirlik(Installability) : Yazılımın kolayca kurulabilir olmasını ifade eder.
- Uyarlanabilirlik (Conformance) : Yazılımın farklı belirli platformlara uyarlanıp uyarlanamayacağını ifade eder.
- Değiştirilebilirlik(Replaceability) : Yazılımın bileşenlerinin değiştirilebilir olma yeteneğini ifade eder [13].

### 2.1.2 Yazılım Kalitesi Modelleri

ISO standardı 8402'de (ISO, 1994), bir yazılım kalite modeli şu şekilde tanımlanmaktadır: "Kalite gereksinimlerini belirlemek ve kaliteyi değerlendirmek için temel sağlayan özellikler kümesi ve bunlar arasındaki ilişkiler".

Yazılım kalite modelleri, yazılım kalitesinin değerlendirilmesi için kaliteye ilişkin farklı görüşleri birleştirir , ortak kabul edilebilir bir kalite çerçevesini tanımlayarak belirli bağlamlara göre uyarlanabilir ve ölçülebilir bir temel sağlar.Özellikle, bu erken modellerde vurgulanan kalite özelliklerinin, büyük ölçüde tasarımcının bakış açısına dayandığı ileri sürülmüştür. Bununla birlikte, kullanıcı / müşteri ve geliştiren şirket gibi başka paydaşlar da vardır.

Yazılım kalitesi çeşitli standardizasyon çabalarına konu olmuştur. Bunlardan en çok bilinenleri McCall ve Boehm modelleridir.

#### 2.1.2.1 McCall Modeli

İlk kalite modeli, 1976'da McCall ve Joseph tarafından önerildi. Yazılım kalite faktörü çerçevesi sundular ve kalite özelliklerini üç gruba ayırdılar.

**Ürün İşlemi:**Sistemin hızlı bir şekilde anlaşılma, verimli bir şekilde çalıştırılma ve kullanıcı tarafından istenen sonuçları sağlayabilme, yani doğruluk, güvenilirlik, verimlilik, bütünlük ve kullanılabilirlik gibi nitelikleri içeren beceridir.

**Ürün Revizyonu:**Hata düzeltme ve sistem adaptasyonu ile ilgilidir. Bu özellik genellikle yazılımın en maliyetli kısmı olarak kabul edilir ve sürdürülebilirlik, esneklik ve test edilebilirlik gibi özellikleri içerir.

**Ürün Geçişi:**Hızla değişen donanımla birlikte dağıtılmış işlemeyi ifade eder ve taşınabilirlik, yeniden kullanılabilirlik ve birlikte çalışabilirlik gibi özellikleri içerir [14].

Modelin temelini oluşturan ana faktörler aşağıdaki gibi sıralanmıştır.

- Doğruluk (Correctness)
- Güvenilirlik (Reliability)
- Sürdürülebilirlik (Maintainability)
- Test edilebilirlik (Testability)
- Taşınabilirlik (Portability)

Bu faktörlerden doğruluk ve güvenilirlik; sistemin hızlı anlaşılma, verimli çalıştırma ve kullanıcı tarafından istenen sonuçları sağlayabilme becerisidir. Sürdürülebilirlik ve test edilebilirlik ; hata düzeltme ve sistem adaptasyonu ile ilgilidir. Bu özellik genellikle yazılımın en maliyetli kısmı olarak kabul edilir. Taşınabilirlik ise Hızla değişen donanımla birlikte çalışabilirlik özelliğini devam ettirebilmesidir [15].

McCall modelinin arkasındaki ana fikir, dış kalite faktörleri ve ürün kalite kriterleri arasındaki ilişkileri değerlendirmektir. Dış kalite, müşteriler tarafından ölçülen kalitedir; iç kalite, programcılar tarafından ölçülen kalitedir.Bu modelin en büyük avantajı, kalite özellikleri arasında yaratılan ilişkidir; ancak ana dezavantaj, yazılım ürününün işlevsellik yönünü içermemesidir [16].

#### 2.1.2.2 Boehm Modeli

Boehm, bir yazılım ürününün Sürdürülebilirliğine vurgu yaparak McCall'ın modeline bazı özellikler ekledi. Kullanıcıların ihtiyaçlarını içeren hiyerarşik bir yapı şeklinde olduğu için McCall modeline benzer. Boehm'in modeli , sistemi kullanması beklenen kullanıcı türlerini göz önünde bulundurur. Genel olarak program, taşınabilirlik, kullanışlılık ve bakım kolaylığı/sürdürülebilirlik olarak ayrılmıştır. Kullanışlılık ayrıca güvenilirlik, verimlilik ve insan mühendisliği olarak ayrılmıştır. Sürdürülebilirlik ise test edilebilirlik, anlaşılabilirlik ve değiştirilebilirlik olarak ayrılır. Bununla birlikte, Boehm'in modeli, bu özellikleri ölçmek için metodolojiyi detaylandırmamaktadır [17].

#### 2.1.2.3 FURPS Modeli

FURPS, adını oluşturan beş özelliği dikkate alır: İşlevsellik (Functionality), Kullanılabilirlik (Usability), Güvenilirlik (Reliability), Performans (Performance) ve Desteklenebilirlik (Supportability). Robert Grady tarafından önerilen model, özellikleri iki farklı gereksinim kategorisine ayırır: İşlevsel gereksinimler: Girdi ve beklenen çıktı ile tanımlanır. İşlevsel olmayan gereksinimler: Kullanılabilirlik, Güvenilirlik, Performans ve Desteklenebilirlik. FURPS modelinin bir dezavantajı, özellikle yazılım tabanlı sistemler için uygulama geliştirme için önemli bir kriter olabilecek taşınabilirlik yönünü dikkate almamasıdır [17].

#### 2.1.2.4 Dromey Modeli

Yazılımların sahip olduğu özellikleri 4 kategoriye ayırmıştır ve bu özellikler, yazılımın kalitesini değerlendirmek için kullanılır.

- Doğruluk (Correctness): Yazılımı, bazı temel ilkelerin ihlal edilip edilmediğini değerlendirir.
- Dahili (Internal): Yazılımın amaçlanan kullanımına göre ne kadar iyi uygulandığını ölçer.
- Bağlamsal (Contextual): Yazılımın kullanımıyla ilgili ve bununla ilgili dış etkilerle ilgilenir.
- Tanımlayıcı (Descriptive): Yazılımın açıklayıcılığını ölçer.

**Tablo 2.1** Modelleri arasındaki kalite faktörleri karşılaştırması

| Faktör | McCall | Boehm | Dromey | FURPS | ISO / IEC 25010 |
|---|---|---|---|---|---|
| Maintainability | * | * | * | | * |
| Flexibility | * | * | | | |
| Testability | * | | | | |
| Correctness | * | * | | | |
| Efficiency | * | * | * | | * |
| Reliability | * | * | * | * | * |
| Integrity | * | * | | | |
| Usability | * | * | * | * | * |
| Portability | * | * | * | | * |
| Reusability | * | * | * | | |
| Interoperability | * | | | | |
| Human Engineering | | | | | |
| Understandability | | | | | * |
| Modifiability | | | | | * |
| Functionality | | | * | * | * |
| Performance | | | | * | |
| Supportability | | | | * | |
| Security | | | | | * |

### 2.1.3 Standartlar

Kalite yönetimi kavramı, farklı kişiler için, farklı rollerde, farklı durumlarda ve farklı zamanlarda farklı anlamlar taşıdığından özünde anlaşılmazdır. Bu anlaşılmazlığın ortadan kalkması ve ortak bir dil oluşturmak için bir dizi standart geliştirilmiştir.Geliştirilen bu standartların en önemlileri alt başlıklar şeklinde detaylandırılmıştır.

#### 2.1.3.1 ISO 9001

Bu, herhangi bir iş kolundaki herhangi bir kuruluş için geçerli genel bir standarttır. Bir dizi işlem sürecini tanımlar ve bu işlem süreçlerinin tasarlanması, belgelenmesi, uygulanması, izlenmesi ve sürekli iyileştirilmesini önerir [18].ISO 9001 kapsamında yer alan süreçler aşağıdaki şekilde sıralanmıştır.

- Kalite Yönetim Süreci
- Kaynak Yönetimi Süreci
- Düzenleyici Araştırma Süreci
- Pazar Araştırma Süreci

- Ürün Tasarım Süreci
- Satın alma süreci
- Üretim süreci
- Hizmet Sağlama Süreci
- Ürün Koruma Süreci
- Müşteri İhtiyaçları Değerlendirme Süreci
- Müşteri İletişim Süreci
- İç İletişim Süreci
- Belge Kontrol Süreci
- Kayıt Tutma Süreci
- Planlama Süreci
- Eğitim süreci
- İç Denetim Süreci
- Yönetim İnceleme Süreci
- İzleme ve Ölçme Süreci
- Uygunsuzluk Yönetim Süreci
- Sürekli İyileştirme Süreci

#### 2.1.3.2 ISO / IEC 9126

Bu standart, yazılım için bir kalite modeli ve modeli desteklemek için bir dizi ölçüm içerir [19]. Bu özellikler şunları içerir: İşlevsellik, Güvenilirlik, Kullanılabilirlik, Verimlilik, Sürdürülebilirlik ve Taşınabilirlik. Bu ana özelliklere ait kısa tanımlar aşağıdaki şekildedir.

**İşlevsellik:**Yazılımın belirlenen gereksinim ihtiyaçlarına göre gerekli fonksiyonları sağlayabilmesidir.

**Güvenilirlik:** Yazılımın, performans seviyesine önem verebilmesidir.

**Kullanılabilirlik:** Yazılımın kullanılabilme kolaylığıdır.

**Verimlilik:** Yazılımın çalışması esnasında fiziksel kaynakları etkin biçimde kullanabilmesidir.

**Sürdürülebilirlik:** Yazılımda değişiklik yapabilme kolaylığıdır.

**Taşınabilirlik:** Yazılımın kolay kurulabilmesi, kurulduğu ortama kolay uyum sağlayabilmesi durumudur [20].

#### 2.1.3.3 ISO / IEC 25010

2011 yılında, bu standart ISO 9126'nın yerini almıştır. Modernize edilmiş bir dizi kalite özniteliği içerir ve özellikle güvenlikle ilgili özniteliklere daha fazla önem verilmiştir [21].

#### 2.1.3.4 ISO / IEC 15504

Bazen SPICE olarak da anılan bu standart, yazılım edinme, geliştirme, işletim, tedarik, bakım ve destekle ilgili geniş bir süreçler dizisini kapsar.

#### 2.1.3.5 CMM / CMMI

Carnegie Mellon University SEI (Software Engineering Institute) tarafından oluşturulmuştur. CMM terimi, Yetenek Olgunluk Modeli (Capability Maturity Model) anlamına gelir ve yazılım geliştirme süreçlerini belirtir. Bir geliştirme organizasyonunun, sahip olduğu süreçlere bağlı olarak bir dizi olgunluk seviyesinden birine ait olarak tanımlanabileceği şekilde yapılandırılmıştır [22].

#### 2.1.3.6 ISO 27001

Bu model, özellikle bilgi güvenliği süreçlerinin sertifikasyonu için tasarlanmıştır ve bilgi güvenliğiyle ilgili bir dizi standardın parçasıdır [23] . ISO / IEC 25010, ISO / IEC 15504 ve CMM / CMMI gibi, ISO 27001 de süreç merkezlidir. Bu, modellerin, süreç doğruysa sonucun tatmin edici olacağına dair temel varsayıma dayandığı anlamına gelir [24].

#### 2.1.3.7 Ortak Kriterler (ISO / IEC 15408)

Ortak kriterler hem ISO 27001 gibi güvenliğe yöneliktir hem de ISO 27001'den farklı olarak, sadece süreç gereksinimlerini değil ürün gereksinimlerini de içerir.

Yedi adet değerlendirme güvence düzeyi (EAL'ler) tanımlanmıştır. Bunların en düşük olanı, EAL1, doğru çalıştığını görmek için işlevsel olarak test gerektirirken , EAL7 kaynak kodun hepsinin doğrulamasını zorunlu kılar.

- EAL1: İşlevsel olarak test edilmiştir.
- EAL2: Yapısal olarak test edilmiştir.
- EAL3: Metodik olarak test edilmiş ve kontrol edilmiştir.
- EAL4: Metodik olarak tasarlanmış, test edilmiş ve gözden geçirilmiştir.
- EAL5: Yarı biçimli olarak tasarlanmış ve test edilmiştir.
- EAL6: Tasarım yarı biçimsel olarak doğrulanmış ve test edilmiştir.
- EAL7: Tasarım resmi olarak doğrulanmış ve test edilmiştir.

### 2.1.4 Yazılım Kalite Metrikleri

Yazılım metrikleri, tüm geliştirme süreci boyunca verimli bir yönetim için yararlı ve ilgili bilgiler elde etmek amacıyla yazılım geliştirme sürecinin ve ürünlerinin farklı niteliklerinin ölçülmesi ve bu ölçümlerin yorumlanması olarak tanımlanabilir [25]. Geliştirmenin tüm paydaşları üzerinde bu metriklerin etkisini açıkça görebiliriz. Paydaşları yöneticiler, geliştiriciler ve müşteriler olarak kabaca ayırabiliriz.

#### 2.1.4.1 Yazılım Metriklerinin Yöneticiler Üzerinde Etkisi

Yöneticilerin amacı sürecin maliyetine göre en büyük geliri sağlayacak faaliyetlere odaklanmaktır. Yazılım üretiminde kullandığımız zamanı ve emeği ölçebilirsek, sadece toplam projenin maliyetini değil, farklı parçaların bütüne nasıl katkıda bulunduğunu da anlayabiliriz. Bir maliyet veya çaba ölçüsüne sahip olduğumuzda, projenin üretkenliği hakkında bir fikir edinmek için bunu projenin boyutu ile yorumlayabiliriz. Ayrıca edindiğimiz bilgileri gelecek projeleri tahminlemede kullanabiliriz. Metrikler gereken çabanın sayısallaştırılmasını sağlamaktadır böylece yöneticiler için yol gösterici olabilmektedir [25].

#### 2.1.4.2 Yazılım Metriklerinin Geliştiriciler Üzerinde Etkisi

Yazılım metriklerinin en önemli avantajı, yazılım sürecinin farklı niteliklerinin nesnel olarak ölçülebilir hale getirmesi yani sayılarla ifade edilebilmesidir. Böylece kod gereksinimleri objektif olarak değerlendirilebilir olur. Karmaşıklık, sürdürülebilirlik ,

bağımlılık gibi bir çok başlık altında kabul edilebilir sınırlar içerinse olup olmadığında bakılır. Çıkan sonuçlara göre uygun aksiyon alınmasını sağlar. Ayrıca ürünlerin ve süreçlerin özelliklerinin ölçülmesi, kalite standartlarının ve kalite hedeflerinin karşılanıp karşılanmadığını söyleyebilir. Örneğin, yazılımın karmaşıklığı için bir sınır değer belirlediysek , test aşamasında bunu kolaylıkla kontrol edebiliriz. Planlanan kalite kontrol testlerinin olması gereken detayı hakkında bilgi sahibi olmamızı sağlar.

Yazılım metrikleri yalnızca geliştirme sürecine geriye dönük bir bakış olarak yazılımın güvenliği , bakımının kolaylığı , kalitesi hakkında bilgi vermekle kalmaz gelecekteki ürün ve süreçlerin özelliklerini tahminlemede de kullanılır. Böylece yapılan zaman ve maliyet tahminleri daha isabetli sonuçlar verir [25].

#### 2.1.4.3 Yazılım Metriklerinin Müşteriler Üzerinde Etkisi

Yazılım metrikleri, üretilen sistemi kullanacak kişiler için açık bir ilgi alanıdır. Kalite, maliyet, zaman tüketimi vb. konularda bilgi sahibi olmalarını, böylece farklı ürünleri veya firmaları birbirleri ile karşılaştırabilmelerini sağlar. Böylece daha rasyonel ve sağlam temellere dayanan bir seçim yapabilirler. Ayrıca, bir yazılım ölçüm programından elde edilen bilgiler, kullanıcı tarafından başka bir şirketin geliştirilen bir projeyi veya üretilen ürünü değerlendirmek için kullanılabilir. Ölçümler daha sonra bir yazılım sisteminin kalitesini değerlendirmek için kullanılabilir. Soru, sistemin sahip olmasını istediğimiz güvenilirliğe, sürdürülebilirliğe, kullanılabilirliğe, verimliliğe vb. sahip olup olmadığıdır. Yazılım metrikleri, bir projenin maliyetine göre umduğumuz kadar iyi performans gösterip göstermediğini açıklayabilir, üretkenliği çok düşük olan projeleri keşfetmek için de kullanılabilir [25].

## 2.2 Yazılım Geliştirme Yaşam Döngüsünde Yazılım Kalitesi

SDLC, yazılımın tasarlanmasından piyasaya sürülmesine kadar geçen süreyi ifade eder . Bu yaşam döngüsü çeşitli aşamalara bölünmüştür: Fizibilite çalışması, Gereksinimlerin analizi ve belirtilmesi, Tasarım, Uygulama veya Kodlama, Test, İşlemler ve Bakım . Bu aşamalar kullanılan SDLC modeline bağlı olarak değişebilir . SDLC, yazılım sistemleri oluşturma veya sürdürme süreciyle ilgilidir . Bir yazılım geliştirme organizasyonunun bir yazılım ürününü başarılı bir şekilde geliştirmek için kullanması gereken tüm geliştirme sürecini temsil eder. Modern SDLC'nin geleneksel ve çevik olmak üzere iki ana kategorisi vardır . Çevik SDLC yöntemleri, yaşam döngüsünü kısaltmayı, hata oranlarını en aza indirmeyi, müşteri memnuniyetini artırmayı ve geliştirme süreci sırasında gelişen iş gereksinimlerini karşılamayı amaçlamaktadır. Yazılım kalitesi, yazılım tasarımı, kodlama / uygulama, kaynak

kod kontrolü, kod incelemeleri, konfigürasyon ve hem geliştirme hem de kullanıcı tarafında test ve ayrıca değişikliklerin yönetimi ve piyasa sürümünün dahil olduğu tüm SDLC'yi kapsar. Verimli olabilmek için kalite faaliyetleri , yazılım yaşam döngüsünün her aşamasında takip edilmelidir. Her adım için sürecin sonunda ortaya çıkan çıktının ve nihai olarak ürünün kalite standartlarını sağladığından emin olunmalıdır.

# 3
# YAZILIM METRİKLERİ

## 3.1 Yazılım Metriklerinin Sınıflandırılması

Yazılımın farklı aşamaları için metrikler aşağıdaki şekilde gruplandırılmıştır.

- Süreç Metrikleri (yazılım geliştirme sürecinin ölçüleri)
- Ürün Metrikleri (yazılım ürünü özelliklerinin ölçümleri)
- Kaynak Metrikleri (yazılım kaynaklarının ölçüleri)

### 3.1.1 Süreç Metrikleri

Yazılım süreç özelliklerinin ölçümüyle ilgili yazılım metrikleridir. Geliştiricinin her hangi bir aşamada hesaplayabileceği bu ölçümlerle, süre ve maliyet tahminlerinin yanı sıra yürütülen sürecin kalitesi ve verimliliği hakkında bilgi sahibi olunabilir.

Süreç metriklerine örnek olarak Kaynak Kod Satırı Sayısı (SLOC) ölçüsü verilebilir.Bu sayı, o kodu geliştirmek için harcanan çabanın bir göstergesi olabilir.

### 3.1.2 Ürün Metrikleri

Ürünün ölçümü sadece müşteriye teslim edilebilir ürünü değil, aynı zamanda geliştirme sürecinde yapılan tüm faaliyetleri, örneğin dokümantasyon ve prototipleri de içerir. Ürün ölçümleri, ürünün harici özelliklerinin veya dahili özelliklerinin ölçüsüdür.

Ürünün harici özellikleri, gerçek ortamda gösterdiği performansının bir ölçüsüdür. Dahili ürün özelliklerine örnek olarak da şunlar verilebilir: yazılımın boyutu, doğruluğu, karmaşıklığı, hataları ve test edilebilirliği.

### 3.1.3 Kaynak Metrikleri

Bu ölçümler, yazılım projesi için ihtiyaç duyulan kaynakların tahmini için kullanılır. Kaynakları içerisine geliştiricileri alabileceğimiz fiziksel kaynaklar , bilgisayarları , malzemeleri içeren araçlar ve yöntemler olarak sınıflandırabiliriz [26].

## 3.2 Yazılım Metrikleri Bileşenleri

Yazılım metrikleri, yazılımın her ölçülebilir varlığına uygulanabilir. Bu ölçümler, yazılım geliştirme yaşam döngüsü boyunca uygulanabilir. Bir yazılım projesinin başlangıcında, örneğin maliyet tahminini ve kaynak gereksinimlerini belirlemek için kullanılabilirler.

Geliştirme aşamasında ölçülen ölçüm sonuçları ile geliştiriciler, kodlarında gerekli düzenlemeleri yaparak örneğin karmaşıklık sınırını aşan bir kod bloğunu yeniden tasarlayarak bakım maliyetlerini düşürebilir ve sürdürülebilirdiği arttırabilirler.

Kontrol aşamasında yapılacak ölçümler sonucu ise yürütülecek test süreçlerine rehber olabilir. Güvenli metrik sınırları dışında olan ya da bu sınırlara yakın olan bölümler daha detaylı test edilebilir böylece testlerden alınan verimlilik artabilir [26]. Metrik sonuçlarının kullanılabileceği başlıca alanlar aşağıdaki gibi sıralanabilir:

- Geliştirme maliyeti tahminleri
- Geliştirme eforu tahminleri
- Mevcut yazılım kalitesinin durumu
- Yazılımın sürdürülebilirlik durumu
- Yazılımın karmaşıklık analizi
- Yazılımın hata analizi
- Yazılımın testi

Ayrıca yazılım kalite ölçümü, ölçümler aracılığıyla da mümkündür. Yazılım kalitesinin objektif olarak yorumlanması ve belirlenmesi için özelliklerinin sayısallaştırılması gerekmektedir. Bunun için çeşitli metrik kümeleri kullanılmaktadır. Bu metrik kümeleri daha çok nesneye yönelik programlama dillerinde kullanılan sınıf ve metot seviyesi olmak üzere ikiye ayrılırlar [27].

### 3.2.1 Sınıf Seviyesindeki Yazılım Kalite Metrikleri

Chidamber-Kemerer tarafından 1994 yılında ortaya konan metrik kümesidir.CK olarak kısaltılır. Bu yaklaşıma göre hesaplanan metrikler alt başlıklarda sıralanmıştır.

#### 3.2.1.1 WMC (Weighted Method per Class)

WCM, bir sınıftaki ağırlıklı metot sayısıdır. McCabe IQ aracı içerisinde, eşik seviyesi 14 olarak gösterilmiştir [27].

#### 3.2.1.2 DIT (Depth of Inheritance Tree)

Kalıtım ağacının derinliğidir yani DIT, bir sınıftan kalıtım ağacındaki kök elemana olan en uzak yolun mesafesidir. McCabe IQ aracı içerisinde, eşik seviyesi 7 olarak gösterilmiştir [27].

#### 3.2.1.3 RFC (Response for a Class)

Bir sınıf içerisinde yer alan , sınıfın tetiklediği metot sayısı ile kalıtımdan gelen metot sayısı toplanarak hesaplanır. McCabe IQ aracı içerisinde, eşik seviyesi 100 olarak gösterilmiştir [27].

#### 3.2.1.4 NOC (Number of Children)

Alt sınıf sayısıdır.Sınıfın çocuklarının sayısı ya da bir sınıftan türeyen sınıfların sayısı olarak tanımlanabilir. McCabe IQ aracı içerisinde, eşik seviyesi 3 olarak gösterilmiştir [27].

#### 3.2.1.5 NOM (Number Of Methods)

Bir sınıftaki metot sayısıdır.

#### 3.2.1.6 CBO (Coupling between Object Classes)

Nesne sınıfları arası bağımlılıktır. Sınıflar arasında kalıtım yokken bir sınıf içindeki özellik ya da metodların diğer sınıfta kullanılması sınıflar arasına bağlımlılık olduğunu gösterir . Yani, A sınıfı B sınıfının özelliklerini (attribute) veya operasyonlarını kullanıyorsa, A sınıfının B sınıfına bağlı (coupled) olduğu söylenir. CBO, bir sınıfın kalıtım dışında bağlı olduğu farklı sınıfların sayısıdır, verimliliği ve yeniden kullanılabilirliği ölçmede kullanılır McCabe IQ aracı içerisinde, eşik seviyesi 2 olarak gösterilmiştir [27, 28].

#### 3.2.1.7 LCOM (Lack of Cohesion in Methods)

Metodların uyumluluğudur.Her özellik (attribute) için o özelliği kullanan metotların yüzdesi belirlenip ortalaması alınır ve bu değer 100'den çıkarılır. Elde edilen değer, LCOM değeridir. Sınıf içi kohezyon yüksek olması gerekirken, sınıflar arası bağlılık (coupling) düşük olmalıdır. McCabe IQ aracında, eşik seviyesi 75 olarak gösterilmiştir. Bazı kaynaklarda LOCM-CK ve LCOM-HS olarak ikiye ayrılır. İkisi de metodların arasındaki uyum eksikliğini açıklar. LCOM-CK , Chidamber & Kemerer tarafından önerilmiş LCOM-HS ise Henderson-Sellers tarafından önerilmiştir [27, 28].

### 3.2.2 Metot Seviyesindeki Yazılım Kalite Metrikleri

#### 3.2.2.1 Kaynak Kodun Satır Sayısı ( Lines Of Code - LOC)

Kaynak kod satırı (SLOC), kaynak kodun satır sayısını sayarak bir programın boyutunu ölçmek için en eski, en yaygın kullanılan ve en basit metriktir. Bu metrik başlangıçta bir projenin adam-saatlerini tahmin etmek için geliştirilmiştir. LOC genellikle şu şekilde temsil edilir: Kod Satırları (LOC), yorumlar ve boş satırlar dahil her satırı sayar. Kilo Kod Satırları (KLOC), LOC'nin 1000'e bölünmesidir. Etkili Kod Satırları (ELOC), parantez boşlukları ve yorumlar hariç olmak üzere etkin kod satırını tahmin eder. LOC hesaplandıktan sonra, aşağıdaki özellikleri ölçebiliriz:

- Verimlilik = LOC / Kişi ayları

- Kalite = Kusurlar / LOC

- Maliyet = $ / LOC

LOC için bazı öneriler:

- Dosya uzunluğu 4 ila 400 program satırı olmalıdır.
- Fonksiyon uzunluğu 4 ila 40 program satırı olmalıdır.
- Bir dosyanın en az yüzde 30'u ve en fazla yüzde 75'i yorum olmalıdır.

**Avantajlar:** LOC, program çabasının bir göstergesi olarak faydalı olacak şekilde teorize edilmiştir. Yaygın olarak kullanılır ve kabul edilir ve proje tamamlandığında kolayca ölçülür.

**Dezavantajlar:**Yürütülebilir olmayan satırları da içerdiğinden karmaşıklık belirlemede kullanılamaz ayrıca LOC, dile bağlı olduğu için güçlü bir ölçüt olarak kabul edilmez: iki farklı dilde yazılmış iki özdeş işlev SLOC kullanılarak

karşılaştırılamaz, LOC tahmini yalnızca uygulama yapıldığında sayılabilir ve bu da yaşam döngüsünün başlarında tahmin etmenin avantajlarını götürür [8, 29].

#### 3.2.2.2 Sürdürülebilirlik (Maintainability) Endeksi

Döngüsel karmaşıklık, halstead hacmi kombinasyonuna dayalı olarak, kaynak kod satırlarının sayısı ve yorum satırlarının sayısı ile normalleştirilmiş bir programın ne kadar bakım yapılabilir olduğuna ilişkin bir puan hesaplar.

Özetle bir modülün bakımının karmaşıklığını ölçen metriktir. Böylece bakımının ne kolaylıkta yapılacağını gösterir. MI şeklinde kısaltılır.

Hesaplamayı yaparken satır sayısı , cyclomatic complexity ve halsread completiy gibi değerleri kullanır [8, 29].

Halstead Complexity (V),McCabe Cyclomatic Complexity (G) ve kaynak kod satır sayısı (LOC) olmak üzere hesaplamada kullanılan formül aşağıda verilmiştir.

$$MI = 171 - 5.2 * ln(v) - 0.23 * (G)\breve{}162 * ln(LOC) \tag{3.1}$$

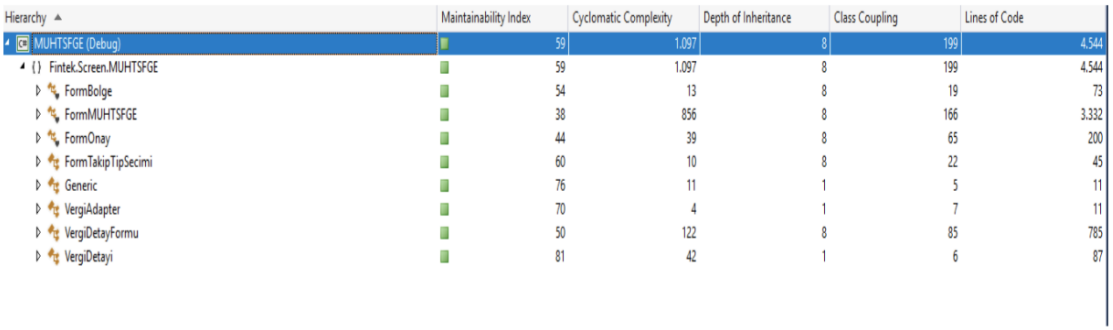
### 3.2.3 Karmaşıklık Metriği

Karmaşıklık ölçütleri çoğunlukla esneklik, hata hassasiyeti, anlaşılabilirlik ve yeniden kullanılabilirlik kalite niteliklerini ölçmek için kullanılır. Karmaşıklık metriği ile ilgili detaylı açıklama ve tanımlamalar BÖLÜM 4'de detaylı olarak açıklanmıştır.

## 3.3 Mevcut Yazılım Metrikleri Araçları

Piyasada bir dizi yazılım ölçüm aracı bulunmaktadır. Bu ölçümlerin temel amacı, yazılım geliştirme, sürdürme ve yönetme sürecinin kalitesini yükseltmektir. Bazıları açık diğerleri lisanslı araçlardır. Her birinde bakılması gereken temel unsurlar dil desteği, platform desteği, lisans fiyatları, destek ölçütleri, kullanılabilirlik ve lisans türüdür [30].

Nesneye yönelik programlama dillerinde kullanılan IDE ler bu metriklerin bir raporunu otomatik olarak bize verebilmektedir. Örneğin , C# projelerinde code metrics ler visiual studio sayesinde otomatik olarak hesaplanabilmekte ve rapor olarak izlenebilmektedir. Localimizde çalıştığımız proje için visual studio üzerinden Analyze ve calculate code metrics dediğimizde aşağıdaki gibi bir döküm elde ederiz.Aynı şekilde java projeleri için de eclipse üzerinde kullanılan plug-in ler mevcuttur.



| Hierarchy | Maintainability Index | Cyclomatic Complexity | Depth of Inheritance | Class Coupling | Lines of Code |
|---|---|---|---|---|---|
| MUHTSFGE (Debug) | 59 | 1.097 | 8 | 199 | 4.544 |
| Fintek.Screen.MUHTSFGE | 59 | 1.097 | 8 | 199 | 4.544 |
| FormBolge | 54 | 13 | 8 | 19 | 73 |
| FormMUHTSFGE | 38 | 856 | 8 | 166 | 3.332 |
| FormOnay | 44 | 39 | 8 | 65 | 200 |
| FormTakipTipSecimi | 60 | 10 | 8 | 22 | 45 |
| Generic | 76 | 11 | 1 | 5 | 11 |
| VergiAdapter | 70 | 4 | 1 | 7 | 11 |
| VergiDetayFormu | 50 | 122 | 8 | 85 | 785 |
| VergiDetayi | 81 | 42 | 1 | 6 | 87 |

**Şekil 3.1** Örnek proje kod analiz raporu

Bu ölçümlerin daha sağlıklı yapılabilmesi için bulgulardan arındırılması ve genel anlamda kalitenin arttırılması için IDE lere eklendi olarak eklenen ve kullanımı oldukça yaygın olan bir diğer araç “SonarQube” dür.Kullanımı kısaca aşağıdaki şekildedir.

Visual Studio’nun herhangi bir sürümde aşağıdaki adımlar yapılarak external tool eklenir.

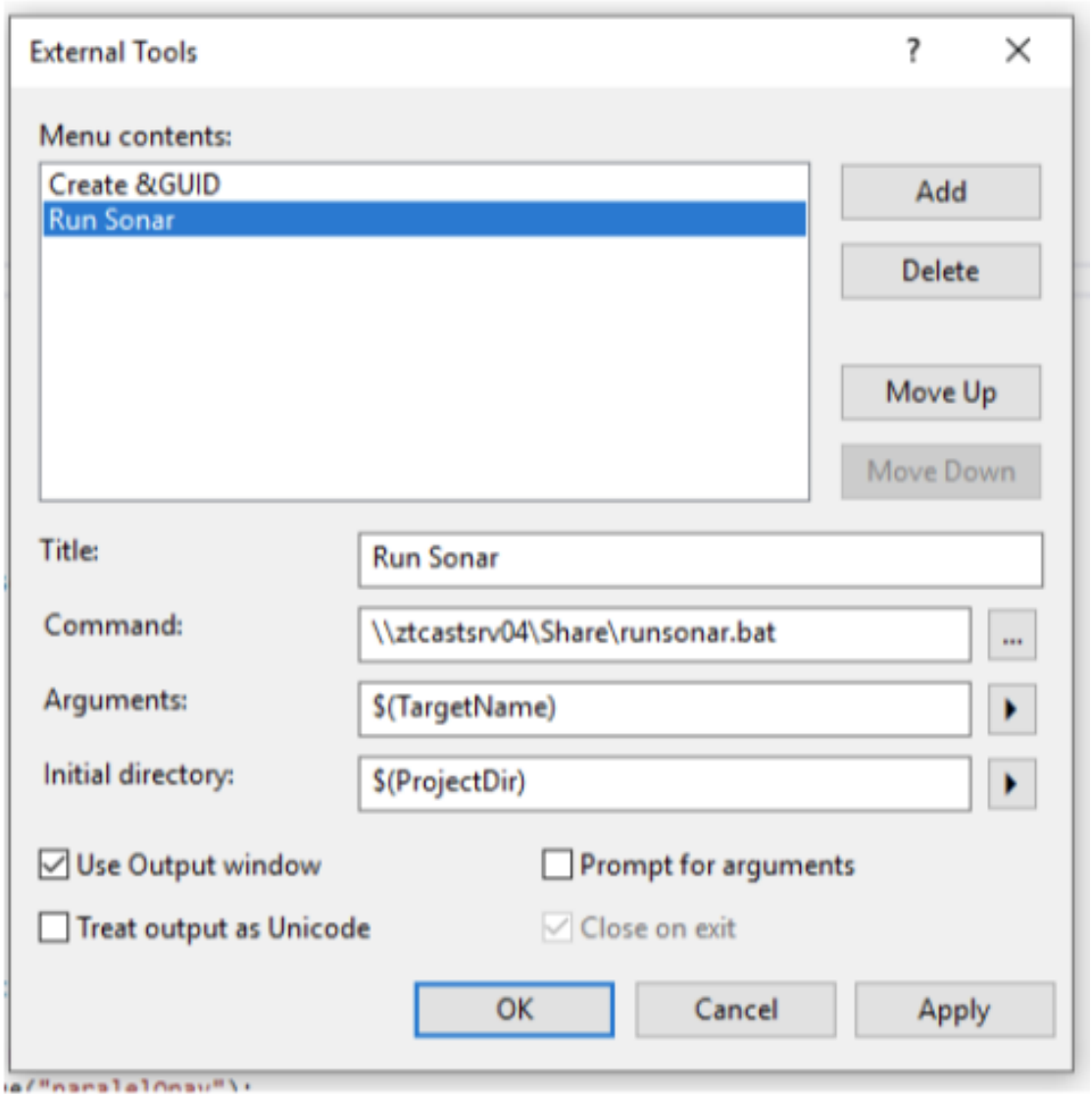
- Tools > External Tools menüsü açılır, açılan pancerede Şekil 3.2 gibi parametreler girilir.
- Daha sonra analiz edilecek proje visual studio’da açılarak Tools > Run Sonar seçilir ve analiz başlatılır. Analiz sonuçlandığında output penceresinde bulgular görüntülenir. Bu pencerede tespit edilen bulgular satır ve sütun numaraları, bulgu kodu ve açıklaması ile birlikte gösterilir.Tespit edilen bulguların üzerine çift tıklandığında, bulguya sebebiyet veren satır adreslenir.

Piyasada kullanılan diğer yazılım ölçüm araçlarının bazıları ise aşağıdaki şekilde sıralanmıştır.

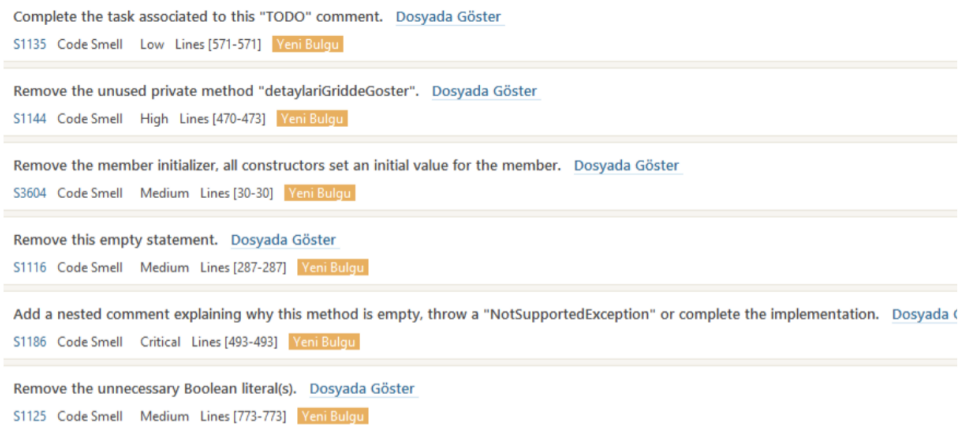
**Understand:** Statik kod analizi aracıdır. Scientific Tools INC. tarafından geliştirilen yazılım, kaynak kodunu analiz ederek kod hakkında; detaylı metrik raporu, akış diyagramı grafikleri, detaylı bağımlılık analizi gibi önemli bilgileri çıktı olarak sunmaktadır .

**OOMeter:** Alghamdi ve arkadaşları tarafından geliştirilen deneysel bir yazılım ölçüm aracıdır. Java / C # kaynak kodunu ve UML modellerini kabul eder ve çeşitli ölçümleri hesaplar.

**Semmle:** Eclipse eklentisidir. Nesneye yönelik kod için SQL benzeri bir sorgulama dili sağlar, böylelikle hataların aranmasına, kod ölçümlerinin ölçülmesine vb. izin verir.




**Şekil 3.2** SonarQube kullanımı parametre girişi




**Şekil 3.3** SonarQube analiz çıktısı örnek görüntü

**VizzAnalyzer:** Kaliteli bir analiz aracıdır. Yazılım kodunu ve diğer tasarım spesifikasyonlarının yanı sıra dokümantasyonu okur ve bir dizi kalite analizi gerçekleştirir.

**Eclipse Metrics Plug-in 1.3.6:** Frank Sauer tarafından hazırlanan Eclipse IDE için açık kaynaklı bir ölçüm hesaplama ve bağımlılık analiz eklentisidir. Çeşitli ölçümleri ölçer ve paket ve tür bağımlılıklarındaki döngüleri tespit eder.

**Eclipse Metrics Plug-in 3.4:** Lance Walton tarafından hazırlanan açık kaynak bir uygulamadır. Derleme döngüleri sırasında çeşitli ölçümleri hesaplar ve metriklerin aralık ihlalleri konusunda uyarır.

**Dependency Finder (Bağımlılık Bulucu):** Derlenmiş Java kodunu analiz etmek için kullanılan açık kaynak kodlu bir araçtır. Çekirdeği, bağımlılık grafiklerini çıkaran ve bunları yararlı bilgiler için kullanan bir bağımlılık analizi uygulamasıdır.

**Chidamber & Kemerer Java Metrics:** Açık kaynaklı bir komut satırı aracıdır. Derlenmiş Java dosyalarının bayt kodunu işleyerek C&K nesnesine yönelik ölçümleri hesaplar.

**CCCC:** Açık kaynaklı bir komut satırı aracıdır. C ++ ve Java dosyalarını analiz eder ve Kod Satırları ve Chidamber & Kemerer ve Henry & Kafura 5 tarafından önerilen ölçümler dahil olmak üzere çeşitli ölçümler hakkında raporlar oluşturur.

**Analyst4j:** Eclipse platformunu temel alır ve Java programları için arama, ölçümler, analiz kalitesi ve rapor oluşturma özelliklerini içerir [31].

**Tablo 3.1** Yazılım araçları ve metrikleri

| Yazılım Metrik Araçları | Yazılım Metrikleri | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | CBO | DIT | LCOM-CK | LCOM-HS | NOC | NOM | RFC | WMC |
| Analyst4j | * | * | * | | * | * | * | * |
| CCCC | * | * | | | * | * | | |
| Chidamber & Kemerer Java Metrics | * | * | * | | * | * | * | |
| Dependency Finder | | * | | | * | * | | |
| Eclipse Metrics Plug-in 1.3.6 | | * | | * | * | * | | * |
| Eclipse Metrics Plug-in 3.4 | | | * | * | * | | | * |
| OOMeter | * | * | * | | | | | |
| Semmle | | * | * | * | * | * | * | |
| Understand | * | * | * | | * | * | | |
| VizzAnalyzer | * | * | * | | * | * | * | * |

## 3.4 Yazılım Kalite Faktörleri Ve Yazılım Metrikleri Arasındaki İlişki

Yazılım kalitesi varsayım modeli ve ISO / IEC 25000 standardı altında, yazılım kalite faktörleri ve kriterlerinin kombinasyonunu buldu. Bu bölümde, yazılım kalite

kriterleri ile yazılım kalite ölçütleri arasındaki ilişkiyi açıklamalıdır. Her faktör için bir kriter listesi geliştirmenin dört nedeni vardır.

- Metrikler, faktörlerin daha eksiksiz, somut bir tanımını sunar.
- Faktörler arasında ortak olan metrikler, faktörler arasındaki karşılıklı ilişkiyi göstermeye yardımcı olur.
- Metrikler, denetim ve gözden geçirme ölçütlerinin daha kolay geliştirilmesine izin verir.
- Metrikler, önceden tanımlanmış kabul edilebilir bir standarda kadar olmayabilecek kalite faktörleri alanını saptamamıza izin verir.

# 4
# KARMAŞIKLIK METRİK TANIMI VE KARMAŞIKLIĞIN ÖLÇÜLMESİ

## 4.1 Karmaşıklık Metrik Tanımı

Kod karmaşıklığı genellikle bir yazılımın kusurlarına, sürdürülebilirlik sorunlarına ve taşınabilirliğine yol açan ana faktördür. Herhangi bir yazılım ölçüsü, aşağıda sıralanan bir veya daha fazla kod karmaşıklığı faktörünü ölçebiliyorsa, karmaşıklık ölçütü olarak adlandırılabilir. Bu faktörler aşağıdaki şekilde sıralanmıştır.

- Programı anlamanın zorluğu.
- Hataları düzeltmenin ve kodu korumanın zorluğu.
- Kodu başkalarına açıklamanın zorluğu.
- Programı belirlenmiş bazı kurallara göre güncellemenin zorluğu.
- Tasarıma göre program yazma iş yükü
- Programlar yürütülürken gerekli kaynakların mevcudiyeti.

Yazılım karmaşıklığını 4 ana başlıkta inceleyebiliriz.

**Teorik Karmaşıklık:**Çözülecek problemin zaman ve mekan açısından karmaşıklığıdır.

**Kullanım Karmaşıklığı:**Yazılımın kullanım zorluğudur.

**Yapısal Karmaşıklık:**Bir geliştiricinin yazılımı anlaması ve sürdürmesinin zorluğudur.

**Organizasyonel Karmaşıklık:**Yazılımın geliştirilmesi ve bakımı konusunda geliştirme ekibinin koordine etmesi ve işbirliği yapması zorluğudur.

Teorik ve organizasyonel karmaşıklık tüm organizasyonu ilgilendirirken yazılım geliştirme sürecinde özellikle üzerinde durulması gereken karmaşıklık türleri kullanım karmaşıklığı ve yapısal karmaşıklıktır.

Kullanım karmaşıklığı, yazılımı kullanacak olan kişilerin karşılaştığı zorluklardır. Bir şey çok karmaşık görünüyorsa insanlar onu kullanmak ya da satın almak istemeyeceklerdir. Müşteriye zaman kaybı, fazladan gösterilen çaba ve maliyet olarak geri dönüşleri olur. Müşteri memnuniyeti ile direk olarak bağlantılı olduğundan ,yazılım firmaları için oldukça önem arz etmektedir

Yapısal yazılım karmaşıklığı, yazılımın doğası gereği oluşan ve karşılaşılan karmaşıklıktır. Karmaşıklık ölçülerek sistemin hangi bölümlerinin şu an ya da gelecekte bakım sorunlarına potansiyel kaynak olduğu tespit edilebilir , neyin nasıl değiştirileceği belirlenebilir. Yazılım yapısal karmaşıklığını ölçmek için oluşturulan yaklaşımlar sistemi bir bütün olarak değerlendiren ve modüler olarak değerlendirenler olmak üzere ikiye ayrılırlar.

### 4.1.1 Sistem Düzeyinde Karmaşıklık Ölçme

Boyut bir yazılım sisteminin karmaşıklığının ilk göstergesidir. Düşük seviyeli ancak hesaplaması kolay bir ölçüdür. Bunun için genellikle kod satır sayılarının sayılmasıyla elde edilen LOC ve işlev noktaları kullanılır. Bunlara alternatif kullanımı pek yaygın olmayan ABC metriği hakkında da bilgi verilmiştir.

#### 4.1.1.1 Kod Satır Sayısı (LOC)

Kod satırlarının sayısının (LOC) sayılmasıyla ölçülür. Otomatik araçlarla yapılabilen hızlı bir yöntem olmasından dolayı yazılım geliştirme şirketleri arasında oldukça yaygındır. Bununla birlikte, kod satırlarını saymak tamamen nicel bir ölçüdür ve algoritmik karmaşıklığı hiç dikkate almaz. Bu nedenle programın işlevselliğini ve maliyetleri hakkında net bilgi veremez.

Satır kodlarını kullanmanın bir başka kısıtlaması da dil bağımlılığıdır. Tüm prosedürel dilleri kapsayan bir kod satırı ölçüsü için ulusal veya uluslararası bir standardı yoktur. Yazılım ürünlerini kod satırı ölçüleriyle karşılaştırabilmek için aynı dilde kodlanmaları gerekir. Örneğin, karma dil ürünü ile COBOL'da kodlanmış başka bir ürün arasındaki kod satırlarını sayarak yapılan karşılaştırma, doğru değildir. Ayrıca, bireysel programlama stilinden kaynaklanan boyut varyasyonları da olabilir. IBM tarafından yürütülen küçük kontrollü bir çalışma, kod büyüklüğünün programcıların stilleri ve şartnamelerin ne istediğine ilişkin yorumlarına bağlı olarak değiştiğini göstermiştir (Jones, 1996). Kod satırlarının nasıl ölçüleceğine dair bir standart olmadığından, endüstri verilerini karşılaştırma yeteneği sınırlıdır [32].

Ayrıca kod Satırları ölçütleri, dil seviyesi yükseldikçe paradoksal olarak geriye doğru

hareket eder, böylece en güçlü ve gelişmiş diller, daha ilkel düşük seviyeli dillerden daha az üretken görünür. Bunun nedeni, kodun boyutu küçüldüğünde daha düşük bir üretkenlik oluşturan üretkenlik denklemidir. Bu da yanıltıcı sonuçlara sebebiyet vermektedir.

Son olarak kod satırları, yalnızca uygulama kodlandıktan sonra sayılabildiğinden ölçüm sonuçlarını kontrol altında tutma ve dinamik olarak çözüm bulma imkanı vermez. Bu da yapılacak değişikliklerin riskini ve etkileşimini artırır [33].

#### 4.1.1.2 İşlev Noktaları (Function Points)

Yazılım ölçümleri, yazılım ölçümünün nicel ve nitel yönlerinden bahseder. İşlev noktaları, temel nicel ölçütlerden biri olarak sınıflandırılır. Projenin üretici bakış açısına sahip olan kod satırlarını saymanın aksine function point yöntemi daha çok kullanıcının bakış açısına odaklanmıştır. Amacı, kullanılan uygulama veya teknolojilerden bağımsız olarak eşdeğer sonuçlar vermektir.

Yöntem projenin girdi ve çıktı sayılarının tahmin edilebildiğinde bunların kullanılarak geliştirilen sisteme ait bir işlev puanı hesaplanabileceğini ve sonuçların satır sayısına dönüştürülebileceğini söyler. Bu çevrimi yapabilmek için öncelikle işlevsel noktaların bileşenlerini ve bu bileşenlerin karmaşıklıklarının ağırlıklarını bilmek gerekir. 5 ayrı bileşen vardır.

**Dış Girdiler:** Dışarıdan içeriye doğru olan süreçleri ifade eder.

**Dış Çıktılar:** İçeriden dışarıya doğru olan süreçleri ifade eder.

**Dış Sorgular:** Kullanıcı isteği doğrultusunda atılan , bilgileri okuyan sorgulardır.

**İç Mantıksal Dosyalar:** Verilerin saklandığı dosyalardır

**Dış Arayüz Dosyaları:** Paylaşılan dosylardır.

**Tablo 4.1** Bileşenlerin karmaşıklık ağırlıkları

| Bileşen | Basit Karmaşık | Orta Karmaşık | Çok Karmaşık |
|---|---|---|---|
| Dış Girdiler | 3 | 5 | 6. |
| Dış Çıktılar | 4. | 6 | 7 |
| Dış Sorgular | 3 | 5 | 6 |
| İç Mantıksal Dosyalar | 7 | 13 | 15 |
| Dış Arayüz Dosyaları | 5 | 9 | 10 |

Bileşenler ağırlıklarına ve kullanılan programlama diline denk gelen önceden belirlenmiş LOC Katsayısı ile çarpıldığında sonuç satır sayısına dönüştürülmüş olur.

**Tablo 4.2** Bazı programlama dillerinin LOC katsayıları

| Language | QSM SLOC/FP Data | | | |
|---|---|---|---|---|
| C++ * | 50 | 53 | 25 | 80 |
| C * | 97 | 99 | 39 | 333 |
| C# * | 54 | 59 | 29 | 70 |
| Java * | 53 | 53 | 14 | 134 |
| JavaScript * | 47 | 53 | 31 | 63 |
| HTML * | 34 | 40 | 14 | 48 |
| J2EE * | 46 | 49 | 15 | 67 |
| COBOL * | 61 | 55 | 23 | 297 |

İşlevsel noktaların gerçeğe yakın sonuç verebilmesi için bileşenlerin iyi çıkarılması ve karmaşıklık ağırlığı gibi sonuca direk etki eden faktörlerin iyi belirlenmesi gerekir [34].

#### 4.1.1.3 ABC Metriği (Assignment, Branch, Condition)

ABC metriği, 1997'de Jerry Fitzpatrick tarafından yazılım boyutunu ölçmek için tasarlanan bu metrik kod satır sayısından yola çıkarak yazılım boyutunu bulan SLOC ve benzeri diğer ölçütlerin yarattığı dezavantajları azaltmaya çalışmıştır.

ABC kısaltması yazılım dillerinin temel üç bileşeninden gelmektedir: atamalar(assignment) , koşullar (branch) ve koşullar (condition). ABC değeri ise bu bileşen sayılarının sıralı üçlüsü olarak yazıldığı vektördür. Ör. ABC = <5, 10, 4> olarak yazıldığında atama sayısının 5, dallanma sayısının 10 ve koşul sayısının 4 olduğu anlaşılır.

Bu metrik ile yazılımın boyutunu ölçmek için aşağıdaki gibi bir formül geliştirilmiştir [33].

$$|ABC| = sqrt((A*A)+(B*B)+(C*C)) \tag{4.1}$$

Avantajlar;

- Hesaplaması kolay bir metriktir.
- Bir programcının programlama tarzından bağımsızdır.
- İster paket, sınıf, herhangi bir modül veya kod parçası için ABC ölçümü almak mümkündür.

Dezavantajlar;

- ABC ölçüsü bir şekilde bir programın gerçek çalışma boyutunu yansıtır, ancak programın gerçek uzunluğunu yansıtmaz. Bir program herhangi bir görevi yerine getirmiyorsa, ancak programda birkaç satır kod mevcutsa, 0 boyut değerini verebilir.

- ABC metriği, boyut ölçümü için yaygın olarak benimsenmemiştir.

### 4.1.2 Modül Düzeyinde Karmaşıklığı Ölçme

Bir yazılım geliştiricisinin amacı, tek bir yazılım modülünü anlamak veya değiştirmek için gereken çaba hakkında bir fikir edinmekse, bu modülün diğer modüllerle nasıl ilişkili olduğunu bilmek genellikle yeterli olmaz. Bu nedenle, yazılım modüllerinin iç karmaşıklığını anlamak için, öncelikle kontrol akışı yapılarına veya veri akışı yapısına dayalı olarak daha ayrıntılı yazılım ölçütleri önerilmiştir.

Bu türden önerilen ilk ölçütlerden biri McCabe'nin döngüsel karmaşıklığıydı [McCabe1976]. Kontrol akışı karmaşıklığını işlevler, yöntemler veya prosedürler düzeyinde ölçer. Siklomatik karmaşıklık ne kadar yüksekse, geliştiricinin kodu anlaması ve değiştirmesi o kadar zor olacaktır.

Modül düzeyinde karmaşıklık ölçütlerinin bir diğer örneği olarak Halstead, bir programı uygulamak için gereken çabayı program hacminin ve programın zorluğunun ürünü olarak tanımlamıştır.

Sistem düzeyinde bir yapısal karmaşıklık bulmak için , modül düzeyinde bulunan karmaşıklık değerlerinin bir bütünü yansıtacak şekilde bir araya getirilmesi gerekir. Bunun en yaygın yolu , merkezi eğilim ölçülerini (ortalama, standart sapma, ağırlıklı ortalama, medyan gibi) kullanmaktır.

## 4.2 Karmaşıklık Metriğinin Kaynağı

Ürün yaşam döngüsünün her aşamasında farklı karmaşıklık türleri ortaya çıkabilir. Bu nedenle, karmaşıklık doğal karmaşıklık olan gereksinimler aşamasında yaratılan sorunun karmaşıklığı ve sorunun karmaşıklığına eklenen karmaşıklık olan çözümün karmaşıklığı (ek karmaşıklık olarak da adlandırılır) şeklinde sınıflandırılabilir. Çözümün karmaşıklığı ihtiyaç aşamasını takip eden geliştirme aşamalarında, öncelikle tasarım ve kodlama aşamalarında eklenir.

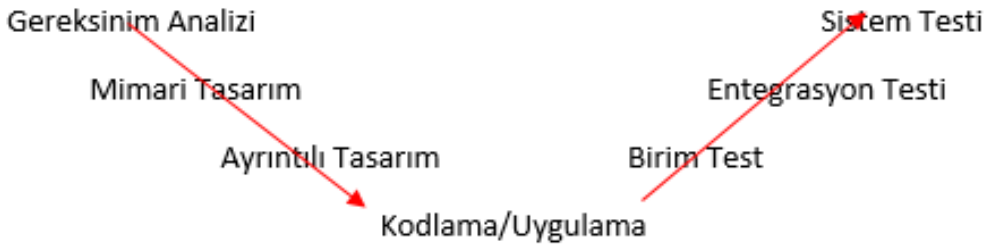
Karmaşıklığın projeye nasıl ve ne zaman getirildiğini ölçebildiğimizde, yazılım karmaşıklığını kontrol edebilmek ve azaltabilmek için ürün yaşam döngüsünün hangi aşamasına odaklanmamız gerektiğini de bilebiliriz.Bu yaklaşımın avantajı,

geliştirmenin farklı aşamalarında karmaşıklığın nasıl ölçüleceğine odaklanarak, daha sonraki süreçte projeye ne kadar karmaşıklık ekleneceğini tahmin etmek için modeller oluşturabilmemizdir. Tez çalışmamızda karmaşıklığın geliştirme süreci boyunca hesaplanabilmesi üzerinde durulmuştur.

## 4.3 Kalite Ve Karmaşıklık Arasındaki İlişki

Yazılımın doğası gereği her zaman içerisinde bir miktar karmaşıklık bulundurur ,ancak bu seviye kontrol altında tutulmalıdır. Karmaşıklık arttıkça yazılımın anlaşılabilirliği, sürdürebilirliği yani kalitesi düşer. Karmaşıklığın kalitenin önüne geçtiği noktada yazılım kullanılamaz hale gelir.

Yazılım kalitesi için gerekli olan temel unsurları veren "Kalite için V-Modeli" adlı bir model bulunmaktadır. Amacı, yazılım geliştirme sırasında ortaya çıkan karmaşıklığı basitleştirmektir. Bu model, geliştiricilerin yüksek kalitede yazılımları etkili bir şekilde geliştirmelerine yardımcı olacak özellikler içerir. Bu modelin birincisi ve en etkili kullanımı, uygulamaya ve teste geçmeden sorunu kolayca bulabilmesidir. Kalite için V-Model şekil 2'de gösterildiği gibidir. Bu modelde, geliştirmenin bir sonraki aşamasına geçmeden önce her bir aşamanın kalitesi gözden geçirilmelidir. Kalite için V-Modeli, hangi yazılım kalite sürecinin uyması gerektiğine dair bir özet sunsa da, böyle bir modeli uygulamak için gereken ayrıntıları ve desteği sağlamaz.




**Şekil 4.1** V Model

Diyagramın sol tarafı, farklı geliştirme aşamalarını gösterirken ve sağ tarafı, yazılım için farklı test türlerini gösterir.

V modelin avantajları ve dezavantajlarını aşağıdaki şekilde sıralayabiliriz.

Avantajlar;

- Proaktif bir kusur izleme söz konusudur, yani kusurlar erken aşamalarda bulunur, hatta uygulama test edilmeden önce geliştirme aşamasında olabilir.

- Kusurlar erken aşamalarda bulunacağı için kusurun giderilmesi maliyetini düşürür.
- Kusurun aşağı doğru akışını engeller.
- V-Modeli organizasyondan ve projeden bağımsız olduğu için V-Modeli farklı projelere uyarlanabilir [35].

Dezavantajlar;

- V modelinin en büyük dezavantajı az esnek olmasıdır. Yani ortada herhangi bir değişiklik olursa, yalnızca gereksinim belgelerinin değil test belgelerinin de güncellenmesi gerekir.
- Her aşamada gözden geçirme gerektirdiğinden kısa vadeli projeler için önerilmemektedir. Yine bu nedenden dolayı uygulamak için kaynağa ihtiyaç vardır. Bu yönüyle küçük şirketler için pek kullanışlı değildir [35].

## 4.4 Karmaşıklık Ölçümünde Kullanılan Modeller

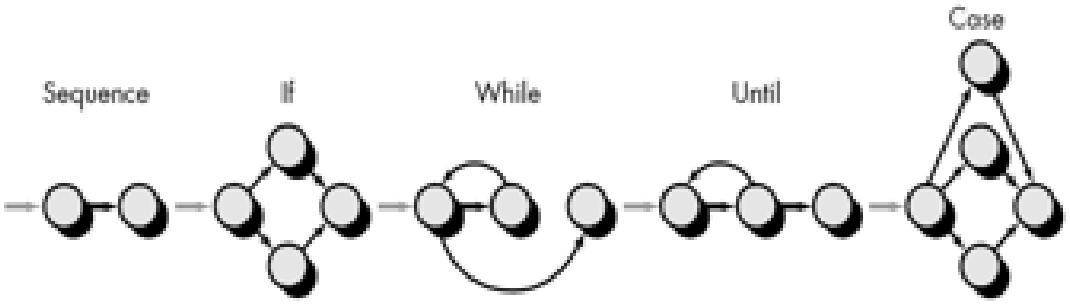
### 4.4.1 Cyclomatic Complexity

Cyclomatic kelimesi, temel döngülerin sayısından gelir. Cyclomatic Complexity bir yazılım metriğidir.1976 yılında Thomas J.McCabe ,Sr. Tarafından geliştirilmiştir ve bir programın karmaşıklığını belirtmek için kullanılır. Bir programın kaynak kodu aracılığı ile , program boyutundan bağımsız olarak daha çok bilgi /kontrol akışına dayalı bir şekilde programın karmaşıklığının nicel ölçümlerini sağlar [36].

Kontrol akış diyagramlarını kullanarak, bir programın hangi modüllerinin test edilmesi ve sürdürülmesinin zor olduğunu kontrol etmek için matematiksel bir teknik formül olarak McCabe tarafından bulunan bu metrik aynı zamanda temel yol test tekniği olarak da bilinir. Temel yol yöntemi tekniği ile bir programın mantıksal ölçümleri yapılabilir ve bu ölçümlere ilişkin test senaryoları seti hazırlanabilir. Bu senaryolar sayesinde , test sırasında her program satırın en az bir kere yürütülmesi sağlanır [36].

Kontrol akışı, bir programı düğümler ve kenardan oluşan bir grafik olarak gösterir. Grafikte, düğümler işleme görevlerini temsil ederken, kenarlar düğümler arasındaki kontrol akışını temsil eder. Karmaşıklığı bu akış üzerinde birbirinden bağımsız yolların sayısı olarak özetleyebiliriz. Bağımsız yol, daha önce başka herhangi bir yolda geçilmemiş en az bir kenarı olan yol olarak tanımlanır.

Kontrol akışı grafiği, aşağıdaki ayrıntılı şekil 1'de gösterilen gösterimleri kullanarak bir programın mantıksal kontrol akışını göstermek için kullanılır [37].




**Şekil 4.2** Kontrol akış grafiği bileşenleri

Program küçükse Cyclomatic Complexity manuel olarak hesaplanabilir. Program çok karmaşıksa, daha fazla akış grafiği içerdiğinden otomatik araçlar kullanılması gerekir. Karmaşıklık sayısına bağlı olarak, ekip, önlem için alınması gereken eylemler hakkında karar verebilir.

Manuel hesaplanması durumunda, kaynak koddan türetilen kontrol akış grafiğine göre hesaplama yapılan McCabe'nin önerdiği aşağıdaki 3 yöntem kullanılır. Bu yöntemlerde cyclomatic complexity, v (G) olarak adlandırılır ve burada v, siklomatik sayıdır ve G, grafiği tanımlar.

1- Bölge Sayısı

2- Kapalı yolların sayısı + 1

3- Düğüm Sayısı - Kenar Sayısı +2

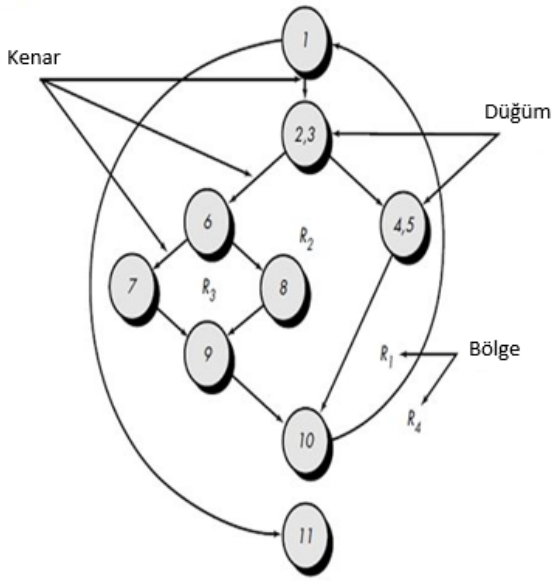
**Bölge Sayısı:** Bir modülün kontrol akış grafiğinin bölgelerinin sayısıdır. Şekil 4.3' deki grafikte 4 bölge görmekteyiz. Karmaşıklık dörde eşittir. v (G) = 4'tür [37].

**Kapalı yolların sayısı + 1:** Döngüsel karmaşıklığı hesaplamanın ikinci yöntemi, kontrol akış grafiğindeki kapalı yolların sayısı + 1'e eşittir, şekil 4.3'de bulunan kapalı yollara göre karmaşıklık değeri v (G) = 3 + 1 yani 4'tür [37].

**Düğüm Sayısı - Kenar Sayısı +2:** McCabe tarafından siklomatik karmaşıklığı hesaplamak için kullanılan üçüncü yöntem aşağıda verilmiştir. Bu yöntemde karmaşıklığı hesaplamak için akış grafiğinin kenar sayısı, kontrol düğüm sayısı ve çıkış noktaları kullanılmıştır [37].

e: edge'lerin yani node'lari birbirine baglayan oklarin sayisi

n: node'larin sayisi




**Şekil 4.3** Kontrol akış grafiği

p: exit point'lerin sayisi

$$V(G) = E \check{} N + 2P \tag{4.2}$$

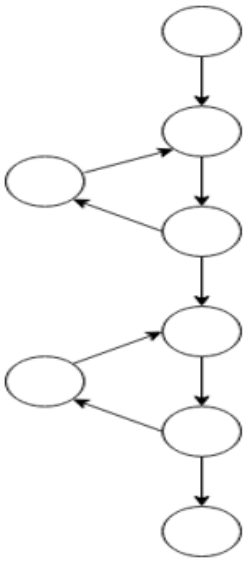
Kullanımı bir örnek ile aşağıdaki şekilde gösterilmiştir.

```
int m, n = 0;

while (m<3)
{
     i++;
}

while (n<3)
{
     j++;
}
```

v (G) = E – N + 2

v (G) = 9 – 8 + 2

v (G) = 3



**Şekil 4.4** Kod bloğunun grafı

#### 4.4.1.1 Cyclomatic Complexity Kullanmanın Avantajları

Beyaz Kutu için yararlı olan bir yazılım ölçüsü ve Yazılım geliştirmede yapılandırılmış testtir. Esas olarak bir programın karmaşıklığını değerlendirmek için kullanılır. Karar noktaları daha fazlaysa, programın karmaşıklığı daha fazladır. Programın karmaşıklık numarası yüksekse, hata olasılığı yüksektir ve bakım ve sorun giderme için daha fazla maliyet ve zaman gerektirir.

**Avantajları:**

- Test sürecine rehberlik edebilir. Cyclomatic complexity, Kalite Güvence ekibinin testinin seviyesini belirlemede etkili olabilir. Karmaşıklık katsayısı yüksekse kod parçasının / işlevselliğin hataya açık olduğu ve derinlemesine bir test gerektirdiği sonucuna varabiliriz.
- Yazılımcıların yazdığı birim testlerin yeterli olup olmadığı konusunda bilgi verir, çünkü bir metodun tüm yolları açığa çıkarır. Açığa çıkarılan yollara daha fazla odaklanmamıza yardımcı olur. Geliştiriciler, tüm yolların en az bir kez test edildiğinden emin olabilirler.
- Yazılan kodun ne kadar efektif, basit ve sürdürülebilir olduğu hakkında bilgi verir.
- Kompleks olan kod parçacıklarını göstereceğinden, bu noktalarda iyileştirme yapıp kodun anlaşılabilirliğini artırmak mümkündür.
- Bir kalite ölçütü olarak kullanılabilir.

- Uygulaması kolaydır.
- Karmaşıklığı sürekli olarak ölçmek, karmaşıklıktaki eğilimleri veya ani artışları yakalayabilir - örneğin, ekipteki yeni bir geliştirici aşırı karmaşık kod üretiyor olabilir ve sorun henüz küçükken çözmenize yardımcı olabilir. Daha fazla kod yazıldıkça, test edildikçe ve üretime itildikçe, yeniden düzenleme daha da zor hale gelir.
- Ekip içi kod kontrol toplantılarında geliştiricilerin kodları basitleştirmesine , düzenlemesine yardımcı olabilir.
- Yazılım bakım maliyetlerinin düşürülmesini sağlar. Karmaşıklığın azalması anlaşılabilirliği arttıracağından , yazılım ekibinde görev değişiklikleri olsa bile yeni gelecek olan yazılımcının anlamasını kolaylaştıracak ve yazılımın bakımı ve geliştirilmesi sırasında kolaylık sağlayacaktır. Bu da zaman ve maliyet açısından tasarruf sağlar.
- Yazılım sağlamlığını iyileştirmede yardımcı olur. Daha basit kodun okunması, anlaşılması ve test edilmesi daha kolay olduğundan, değişikliklere karşı dayanıklı olma ve hata koşullarından beklenmedik şekilde etkilenme olasılığı çok daha düşüktür [38].

#### 4.4.1.2 Cyclomatic Complexity Kullanmanın Dezavantajları

- Verilerin karmaşıklığı değil, programların kontrol karmaşıklığının ölçüsüdür.
- Bunda, iç içe geçmiş koşullu yapıları, iç içe olmayan yapılara göre anlamak daha zordur.
- Basit karşılaştırmalar ve karar yapıları olması durumunda yanıltıcı bir rakam verebilir.

### 4.4.2 Halstead Metrıcs

Maurice Howard Halstead tarafından 1977'de tanıtılan bu yöntem, bileşik karmaşıklık ölçüsü olarak sınıflandırılan yani hem fonksiyonelliği hem yazılımı içeren karmaşıklık ölçüm yöntemlerinden biridir. Bu ölçümü yaparken kod içerisinde aritmetik ve mantıksal işlemleri gerçekleştiren anahtar kelimeler olan operatörlerin yanı sıra bu işlemlerde kullanılan sayılar ve değişkenlere verilen ad olan operandları kullanır. Bunların sayılarını aritmetik olarak toplayarak ,yazılımların karmaşıklığını ölçtüğünü ve kodlar hakkında sayısal bilgi elde ettiğini savunur [39].

Halstead operatör ve operandları kullanarak aşağıdaki metrikleri çıkarır.

n1= Benzersiz operatör sayısı

n2= Benzersiz operand sayısı

N1= Toplam operatör sayısı

N2= Toplam operand sayısı

$$N = N1 + N(Length) \tag{4.3}$$

$$n = n1 + n2(vocabulary) \tag{4.4}$$

Yukarıdaki uzunluk ve sözcük kapasitesi Halstead' ın diğer metriklerinin hesaplanmasında girdi olarak rol oynar. Diğer metrikler ise aşağıda sıralanmıştır.

**Program Uzunluğu (Length /N):** Yazılan programda kullanılan toplam operatör ve operand sayısının toplamıdır. Kabul edilebilir üst sınır olarak 300 kabul edilir. 300 den büyük uzunluk değerine sahip bir kod bloğu gözden geçirilmeli gerekiyorsa yeniden tasarlanmalıdır. 500 den büyük uzunluk değerine sahip bir kod bloğu ise mutlaka yeniden tasarlanmalıdır [40].

$$N = N1 + N2 \tag{4.5}$$

**Program Hacmi (Volume/ V):** Programın toplam boyutu ile orantılıdır. Programı depolamak için gereken hafıza alanıdır; bu yüzden bitlerle temsil edilir. En az 20, en fazla 1000 olmalıdır. 1000 den büyük hacim değerine sahip bir kod bloğu yeniden tasarlanması gerekir [40].

$$V = N * log2n \tag{4.6}$$

**Program Zorluğu (Difficulty):** Programın hata toleransını gösterir ve en fazla 50 olmalıdır. 50' den büyük değerlere sahip bir kod bloğu yeniden tasarlanması gerekir [40, 41].

$$D = (n1/2) * (N2/n2) \tag{4.7}$$

**Program Eforu (Effort /E):** Programın anlaşılma çabasını gösterir. 500.000' den büyük değerlere sahip bir kod bloğu az kaliteli olarak değerlendirilebilir ve yeniden tasarlanması gerekebilir [40, 41].

$$E = D * V \tag{4.8}$$

**Program Zaman Eforu(T):**Programlama için gereken süre (T) Programı oluşturmak için makul bir süre olarak gösterilir. Programın anlaşılma çabasını saniye cinsinden gösterir. 5.000' den, yani 1 saat 30 dakikadan büyük değerlere sahip bir kod bloğu az kaliteli olarak değerlendirilebilir ve yeniden tasarlanması gerekebilir [40, 41].

$$T = (E/B) \tag{4.9}$$

Burada B, 5 ile 20 arasında değişen yüksek sayıdır. Bununla birlikte, Halstead'in ilk deneylerinde en iyi sonuçları verdiği için B genellikle 18 olarak alınır.

**Program Hata(Bug) Sayısı(B):** Uygulamadaki hata sayısıdır. Teslim edilen hataların sayısı, yazılım ürününün genel karmaşıklığı ile ilişkilidir [40, 41]. Şu şekilde hesaplanabilir:

$$B = V/3000 \tag{4.10}$$

#### 4.4.2.1 Halstead Metrıcs Kullanmanın Avantajları

Halstead metrics kullanmanın avantajları aşağıdaki şekilde sıralanmıştır.

- Hesaplaması basittir.
- Programların genel kalitesini ölçer.
- Hata oranını tahmin eder.
- Bakım çabasını öngörür.
- Programlama yapısının tam analizini gerektirmez.
- Herhangi bir programlama dili için kullanılabilir [41, 42].

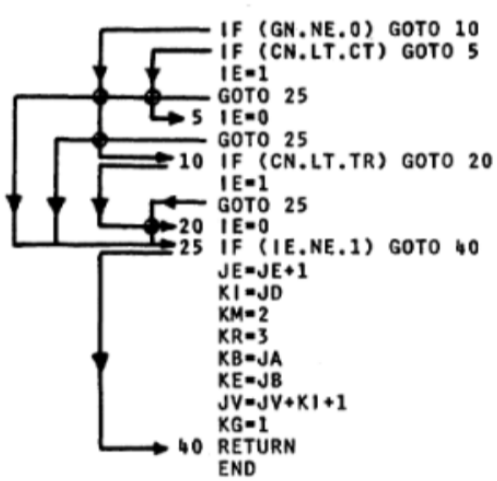
#### 4.4.2.2 Halstead Metrıcs Kullanmanın Dezavantajları

Halstead metrics kullanmanın en büyük dezavantajı kodun tamamına bağlı olmasıdır.

### 4.4.3 Knot (Düğüm) Karmaşıklık Hesabı

Yazılım karmaşıklığını ölçmek için kullanılan ilk yöntem 1968 yılında Dijkstra tarafından bulunmuştur. Bu çalışmada program kalitesi ,içerisinde bulunan "goto" komutlarının azalan bir fonksiyonu olarak tanımlanmıştır ve karmaşıklığın basit bir ölçüsü olarak "goto" komutu belirlenmiştir.

Fortran programlamada programın akışını göstermek amacıyla, bir metin satırından program metninin başka bir satırına olan bir atlamanın nerede gerçekleştiğini göstermek amacıyla oklu çizgiler çizilir. Çizilen bu çizgilerle "GOTO" komutları görselleştirilmiş olur. Üst üste gelen noktalara da "Knot" yani düğüm adı verilmiştir. Bu düğüm sayılarının toplam sayılarından karmaşıklığın bir ölçüsü elde edilmiştir [43].




**Şekil 4.5** Knot karmaşıklık hesabı örnek kod bloğu

Bir karmaşıklık ölçüsü olarak düğüm sayımının kullanımının FORTRAN programlarında daha basit olduğu gözlemlenmiştir çünkü FORTRAN dili, satır başına bir ifade içeren bir dildir. Şekilde görülen Fortran kodunda goto komutlarıyla gerçekleşen sıçramalar oklu çizgiler ile gösterilmiş ve okların kesiştiği yerler de yani düğümlerin oluştuğu yerler nokta ile belirgin hale getirilmiştir. Bu noktaları saydığımızda aşağıdaki kod bloğunun karmaşıklığını bu yaklaşım için 4 olarak buluruz [43].

### 4.4.4 Oviedo Veri Akışı Karmaşıklık Hesabı

Oviedo ‘ nun bir "program kalitesi modeli" olarak geliştirdiği Oviedo karmaşıklık ölçümü yöntemi , bir programın veri akışı özelliklerinden yararlanır. Bu yaklaşıma göre bir program , bir dizi özelliğe sahip sıralı ifadelerin ayrık bloklarından oluşur. Ve bu bloklar arasında veri akışı vardır. Sırayla yürütülür.

Bu model, kontrol akışı karmaşıklığını ve veri akışı karmaşıklığını birlikte ele alır. Nihai karmaşıklık bu ikisinin toplamıdır. Burada kontrol akışı karmaşıklığı CF , kontrol akış çizgisindeki toplam kenar sayısını gösterirken , veri akışı karmaşıklığı DF , her bir blok içerisindeki toplam veri akışını temsil eder [43]. Bir G akış grafiği için Oviedo karmaşıklığı aşağıdaki eşitlikte verilmiştir.

$$OV(G) = CF + DF \tag{4.11}$$

CF (Control Flow Complexity)= Akış grafiği için kontrol akışı karmaşıklığı ;

CF =|E|

E , G de ki kenar sayısıdır.

DF (Data Flow Complexity) = Veri akışını ifade eder.

DF’ un tanımı Oviedo ya göre bir değişken tanımı bir program deyiminde veya bir atama deyiminde yer alır ve bir bloktaki yerel olarak açıklanmış bir değişken referansı, o değişkenin tanımından önce blokta bulunmayan bir değişkene referanstır.

Oviedo hesaplaması kolay bir karmaşıklık yöntemi iken büyük bloklardan oluşan büyük programlarda , hesaplaması oldukça zaman aldığından pek tercih edilmez [43].

**Tablo 4.3** Karmaşıklık modellerinin sınıflandırılması

| **Kontrol Akışı** | **Veri Akışı** |
|---|---|
| Çevrimsel karmaşıklık (McCabe) | Kod Bloklarının Arasındaki Veri Akışı (Oviedo) |
| Kontrol yollarının kesişmesi sonucunda oluşan düğümler (Knot) | Veri Nesnesi Akışı (Halstead) |

Bahsi geçen karmaşıklık modellerini kontrol akışı ve veri akışı olarak gruplandırdığımızda Tablo 4.3 gibi bir sonuç elde ederiz.

### 4.4.5 Bilişsel (Cognitive) Karmaşıklık Hesabı

Yingxu Wang tarafından geliştirilen bu karmaşıklık hesabı "Wang karmaşıklık hesabı" olarak da bilinir.Bilişsel karmaşıklık daha önce öne sürülen karmaşıklık hesaplarından farklı olarak yazılımın dahili yapısının yanı sıra işlediği I/O ' ları da hesaba katar. Wang ortaya çıkardığı bu yeni karmaşıklık hesabını , bir insan zekası eseri olarak yazılımın bilişsel ve psikolojik karmaşıklığının bir ölçüsü olarak tanımlamıştır ve dahili işleme, girdi ve çıktı olmak üzere 3 temel faktöre bağlamıştır.

Çalışmada bir programı anlamak için, yazılımın mimarisine ve temel kontrol yapılarına (Basic Control Structures – BCS'ler) odaklanılmıştır. BCS'ler, mantıksal yazılım mimarileri oluşturmak için kullanılan bir dizi temel akış kontrol mekanizmalarıdır. Bu karmaşıklık hesabı için 10 adet BCS belirlenmiş ve bu BCS ler 5 ayrı dalda kategori edilmiştir.Bu kategoriler sırasıyla ardışık, dallanma, tekrarlama, gömülü sistem ve eş zamanlılıktır. Daha sonrasında her bir BCS'nin bilişsel ağırlığı bulunmuştur. Yazılımın bilişsel ağırlığı, bir dizi BCS tarafından modellenen belirli bir yazılım parçasını anlamak için gereken zorluk derecesi veya göreceli zaman ve çabadır. Bu bilişsel ağırlıkları bulmak için 2 yola gidilmiştir. İlk yaklaşım BCS lerin doğrusal bir düzeyde oldukları ve bilişsel ağırlığı olan tüm BCS lerin toplanması yönünde olurken diğeri bazı BCS elemanlarının daha gömülü olduğu yönündedir ve iç BCS elemanlarının ağırlıklarının dıştakilerle çarpılması sonucu bulunur [44].

Wang , daha önce ortaya atılan geleneksel karmaşıklık hesaplamalarının , yazılımın mimarisini ve dinamik davranışlarını anlamada yoksun, biçimsel yaklaşımlar olduğunu savunmuş ve gerçek zamanlı süreç cebri olarak adlandırdığı RTPA modellerini geliştirmiştir. RTPA ,yazılım sistemlerinin mimarilerini ve davranışlarını resmi ve kesin olarak tanımlamak ve belirlemek için tasarlanmıştır. Dilden bağımsızdır ve bu özelliği sayesinde belirli bir spesifikasyonun herhangi bir uygulaması aynı sonucu verir. Wang, belirlediği 10 adet BCS'yi Java ile kodlamış ve bilişsel psikolojik deneyle (cognitive psychological experiment) ağırlık hesaplamıştır. En sonunda Tablo da "Ayarlanmış Ağırlıklar" kolonunda yer alan değerleri elde etmiştir. Bilişsel ağırlıklar matematiksel çözümlemeler ile elde edilmişken ayarlanmış ağırlıklar bilişsel psikolojik deneyle elde edilmiştir. Burdaki amaç bilişsel ağırlıkları bulmak için bir model ortaya çıkarmaktır ancak bulunan farklı sonuçlar bunun zorluğunu ispatlar niteliktedir [45].

**Wang Karmaşıklık Hesabı İle Karmaşıklığın Bulunması:** Karmaşıklığın hesaplanmasında aşağıdaki formül kullanılır.

$$S_f = \sum_{p=1}^{n_p} S_f(p) = \sum_{p=1}^{n_p} \sum_{c=1}^{n_c} S_f(p,c)[CWU] \tag{4.12}$$

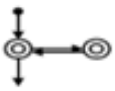
Örnek Kod Parçası ;

```
int main() {
long a,b;
    printf("\n Input the first number A: ");
    scanf("%I", &a);
    printf("\n Input the second number B:");
    scanf("%I", &b);

if (a>b){
    return a;
}
return 0;
}
```

Sequence ağırlığı 1 , branch ağırlığı 2 olmak üzere karmaşıklık hesabı aşağıdaki gibi bulunur.

$$S_f = Ni + NoWc = (2+1).3 = 9[CWU] \tag{4.13}$$

Yukarıdaki sonuç ile hem iç mimari karmaşıklık hem de I / O devir hızı dikkate alındığında, bu algoritmanın karmaşıklığının 9 CWU'ya eşdeğer olduğu görülür

| Kategori | BCS | Simgesel Gösterimi | Bilişsel Ağırlıklar | Ayarlanmış Ağırlıklar |
|---|---|---|---|---|
| Ardışık (Sequence) | Sequence (SEQ) |  | 1 | 1 |
| Dallanma (Branch) | If-then-else |  | 2 | 3 |
| | Case (Switch) |  | 3 | 4 |
| Tekrarlama (Iteration) | For-do |  | 3 | 7 |
| | Repeat-Until |  | 3 | 7 |
| | While-do |  | 3 | 8 |
| Gömülü Sistem (Embedded) | Function Call |  | 2 | 7 |
| | Recursion (REC) |  | 3 | 11 |
| Eşzamanlılık (Concurrency) | Parallel (PAR) |  | 4 | 15 |
| | Interrupt (INT) |  | 4 | 22 |

**Şekil 4.6** Kategorilerle ağırlık ve simgesel gösterim

# 5
# SAKLI YORDAMLAR VE ÖZELLİKLERİ

## 5.1 Saklı Yordamlar

Stored procesdures yani saklı yordamlar, veritabanının veri sözlüğünde depolanan ve uzak bir programdan, başka bir saklı yordamdan veya komut satırından çağrılan bir veya daha fazla veritabanı deyiminden oluşan bir gruptur. Stored prosedür genellikle SPROCS veya SP olarak adlandırılır. SP özellikleri ve komut sözdizimi, veritabanı motoruna özeldir. Geleneksel olarak Oracle, dili olarak PL / SQL kullanır; SQL Server T / SQL kullanır [46].

### 5.1.1 Saklı Yordam Çeşitleri

**1.System Stored Procedure (Sistem Saklı Yordamları):**Ana veri tabanında tutulan , sql server tarafından tanımlanmış ,genellikle ”sp_ “ ön eki ile başlayan ve sistemsel işler için kullanılan saklı yordamlardır.

**2.Extended Stored Procedure (Genişletilmiş Saklı Yordamlar):**Genellikle genişletilmiş anlamına gelen “xp_” ön ekiyle başlayan , sql server dışında dinamik olarak çalıştırılan “dll” şeklinde olan saklı yordamlardır.

**3.Local Stored Procedure (Kullanıcı Tanımlı /Yerel Saklı Yordamlar):**Tez çalışmasında incelenen saklı yordam çeşidi bu sınıfa aittir. Bu sp türü ,sql server üzerinde kullanıcıların tanımladığı sp lerdir.Ekle , silme , güncelleme gibi işlemleri yapabilir.

**4.CLR Stored Procedure(CLR Saklı Yordamları):**Sql server üzerinde CLR ortamında herhangi bir dil kullanılarak yaratılan saklı yordamlardır [46].

### 5.1.2 Saklı Yordam Kullanmanın Avantajları

**1.Network Verimliliği:**SP ‘ler birçok komut içerebilir ve istenen sonucu elde etmek için büyük miktarlarda bilgiyi işleyebilir. Tüm programlama mantığını sunucuda

tutabilir böylece bir istemci programı tarafından işlenmek üzere ağ üzerinden sorgu sonuçlarını almak zorunda kalmaz. Sonuç olarak network trafiğini azaltarak performansı artırır.

**2.Sürdürülebilirlik:**Karmaşık iş kuralları ve programlama mantığı, depolanan prosedürlerde merkezileştirildiğinde, değişiklikler çok daha kolay yapılabilir hale getirir. Her uygulamadaki alanları araştırmak ve değişiklikler yapmak yerine, yalnızca saklı yordamda değişiklik yapmak yeterli olur. Kısacası tek bir noktadan bakım yapılabilmeyi sağlar.

Ayrıca kaydedilip derlendikten sonra, tüm kullanan programlar değişiklikten yararlanır. Birçok uygulamanın aynı sp'yi çalıştırmasıyla tutarlılık daha iyi kontrol edilir ve işleyiş planının tekrar kullanılmasına olanak verir. Yine bu, veritabanınızın kalitesini artırmaya yardımcı olabilir.

**3.Güvenlik:** Veritabanınızın güvenliğini, uygulamaların yalnızca saklı yordam çağrıları aracılığıyla verilere erişip bunları değiştirebilmesiyle sağlayabiliriz böylece anlık sorgulara veya tablolara doğrudan erişime izin verilmez. SP ler üzerinde gerekli güvenlik ayarlamaları yapılarak sp lerin kim ya da kimler tarafından çağrılabileceği yönetilebilir. Saklı yordamlar kodun güvenliğini artırır. Sql injection saldırılarına karşın sistemi korurlar. Ayrıca içlerindeki implementasyonu gizleyerek belli bir güvenlik sağlamış olurlar.

**4.Performans:**Store procedure ler ilk çalıştıklarında bir defa mahsus derlenirler , kullanım sırasında tekrar derlenmezler bu da performans açısından pozitif katkı sağlar.Normalde bir sql komutu çağrıldığında ayrıştırma , derleme ve çalıştırma aşamalarından geçer ancak sp ler daha önceden derlendiğinden normal bir sql sorgusuna göre çok daha hızlı yanıt döner.

### 5.1.3 Saklı Yordam Kullanmanın Dezavantajları

**1.Bellek kullanımı:**Çok sayıda saklı yordam kullanırsak, bu saklı yordamları kullanan her bağlantının bellek kullanımı önemli ölçüde artacaktır.

**2.Hata Ayıklaması ve Bakımı Zordur:**Database bağımlılığı vardır , debug etmesi oldukça zordur bu da hata ayıklamada zorluklara neden olmaktadır. Ayrıca geliştirmesi nesneye yönelik programlamaya göre daha komplike olduğundan ve daha çok karmaşık iş mantıkları için oluşturulduklarından yazılım yaşam döngüsü boyunca bakımı oldukça meşakkatlidir.

## 5.2 Neden Saklı Yordamları Seçtik?

Bankacılık sistemleri birçok farklı açıdan kritik sistemlerdir. Basit bir yuvarlama hatası, finans şirketinin itibarı üzerinde yıkıcı bir etkiye sahip olabilir, bu nedenle geliştiriciler üzerinde hassas bir şekilde çalışmaları ve kodlarını mümkün olduğunca test etmeleri için yüksek bir baskı vardır. Bununla birlikte bankalar arası rekabet , yasal talepler , çağın ihtiyaçlarına karşılık verme gibi nedenlerden dolayı iş birimleri taleplerin hızlı bir şekilde uygulamaya alınmasını talep eder. Geliştiriciler genellikle zaman kısıtlamasından dolayı, kodlarının kalitesini ve sürdürülebilirliğini göz önünde bulundurarak çözümleri doğru şekilde tasarlamak yerine, hızlı, küçük değişikliklerle hali hazırda çalışan ve test edilmiş çözümleri yeniden kullanmayı tercih ederler. Bu genellikle belirsiz ve anlık kararlara dayalı olarak sistemin hızlı bir şekilde gelişmesine ve büyümesine neden olur ve bu da uzun vadede kod tabanı üzerindeki kontrolü kaybetmeye neden olabilir.

Bu noktada bankacılık sisteminde yukarıda saydığımız avantajlarından ve yoğun veritabanı kullanımından dolayı çok fazla kullanılan store procedure'lerin kod kalitesinin ve kontrol altında tutulmasının önemi ortaya çıkmaktadır [47].

# 6
# DEĞERLENDİRME YÖNTEMİ VE SONUÇLAR

## 6.1 Değerlendirme Yöntemi

Bankacılık gibi günlük işlem sayısı fazla olan kurumlarda Bölüm 5 'te açıklanan nedenlerden dolayı bir çok işlem veri tabanında (db) yürütülmektedir. Bu nedenle db nesnelerinin de performanslı çalışması oldukça önem arz etmektedir. Performansın yanı sıra sürekli değişen ve gelişen müşteri ihtiyaçlarına yönelik geliştirme yapılmaya açık olmalı yani sürdürülebilir olmalıdır. Ayrıca hataların ciddi sorunlara yol açabileceği bankacılık sektöründe tüm bu nesnelerin belirli bir kalite standını yakalamış ve bunu devam ettirebiliyor olması gerekir. Bu nesnelerin yaratılması sürecinden itibaren kalite, tüm yaşam döngüsü boyunca izlenmeli ve kontrol altında tutulmalıdır. İstenilen yakından takip karmaşıklığın yapılan her değişiklik sonrası ölçümlenerek belirli sınırlar içerisinde tutulmasıyla sağlanabilir. Bunun için karmaşıklığın yüksek çıktığı kod blokları yeniden tasarlanabilir ya da test senaryoları bu durumlar için arttırılabilir. Böylece alınacak risk minimuma indirilmeye çalışılır. Bu çalışmada db de çalışan saklı yordamların (sp) karmaşıklıklarını hesaplamak için farklı yöntemler üzerinde durulmuştur.

Hazır regex kütüphanesi kullanılarak yapılan ilk çalışmada Cyclomatic Complexity , ikinci çalışmada ise Halstead Metrıcs kullanılmıştır. Üçüncü ve son çalışmada ise tüm bu metrikleri kapsayan, mevcut modellere esneklik ve yeni fonksiyonlar ekleme imkanı veren yeni bir model tasarlanmaya çalışılmıştır. İlk iki çalışmada Regex kütüphanesi kullanılarak belirlenen kontrol anahtar kelimelerin ayıklanıp bunların sayılması ve literatüre uygun hesaplanması ile bir karmaşıklık belirlenirken 3. çalışma da bu belirlenen anahtar kelimelere ağırlık verebilme imkanı sağlamıştır. İlk iki çalışmada veri tabanından okuma ve yazma işlemi yapılırken, üçüncü modele ait işlemler direk veri tabanında gerçekleştirilip performans artışı sağlanmıştır.

Ayrıca ilk iki çalışmada tüm anahtar sözcüklerin ağırlıkları eşittir yapılan hesaplamalara eşit etki de bulunmaktadır, yapılan son çalışmada bu sözcüklere ağırlık verme imkanı sağlanmıştır. Son çalışmada yapılan öznitelik çıkarımı (feature

extraction) genişletilebilir , çalışmasının uzun sürdüğü bilenen saklı yordamlar ya da çok fazla dataya sahip olduğu bilenen tablolar bu anahtar kelimelere eklenebilir ve ağırlık ataması yapılabilir. Böylece saklı yordamlarımızın performanslarına dair daha gerçekçi sonuçlar elde edebiliriz.

### 6.1.1 Cyclomatic Complexity Kullanılarak Karmaşıklığın Hesaplanması

Literatür çalışmasında belirtilen Cyclomatic Complexity nin son formülü kullanılarak karmaşıklık hesaplaması yapılmıştır.

**1.Adım:** İlk aşama olarak hesaplamada kullanılacak key Word ler belirlenmiştir . Bizim hesaplamamızda kullanılacak key Word ler aşağıdaki gibidir.

string[] keyWordSets = "END LOOP", "END IF", "ELSIF" ;

Store Procedure (sp) lerin olduğu tablodan okuma yapılır , her bir sp nin içeriği önce regex kütüphanesi ile düzenlenir. Düzenleme aşamaları aşağıdaki adımları içermektedir;

Yoruma alınmış (commentlenmiş) satırlardan temizlenmesi için bir regex ifadesi yazıldı. Aşağıdaki şekilde sp nin içeriği (linetext) ifadeye verilerek , commentlenmiş satırlar “” ifadesi ile replace edildi.

```
string regexPatternCommentMulti = @"\s*\-\-.*?\n";
Regex regexPatternCommentMultiValue = new Regex(regexPatternCommentMulti, RegexOptions.Multiline);
MatchCollection matchesCommentMultiValue = regexPatternCommentMultiValue.Matches(lineText);
foreach (var item in matchesCommentMultiValue)
{
    string x = item.ToString();
    lineText = lineText.Replace(x, "");
}
```

**Şekil 6.1** Kod içerisinde yorum satırlarının arındırılması

Yorum satırlarından kaçan tarih gibi ifadeleri ayıklamak için aşağıdaki gibi bir regex ifadesi yazılmıştır.

```
Regex regexPatternCommentMulti1 = new Regex(@"\/\*[^\/]*\/", RegexOptions.Multiline);
MatchCollection regexPatternCommentMulti1Value = regexPatternCommentMulti1.Matches(lineText);
foreach (var item in regexPatternCommentMulti1Value)
{
    string x = item.ToString();
    lineText = lineText.Replace(x, "");
}
```

**Şekil 6.2** Kodun istenmeyen bileşenlerden arındırılması

Üstteki adımın benzeri şekilde sp nin create edildiği satırı kaldırmak için bir regex ifadesi yazıldı ve sp bu ifadeden arındırıldı.

```
string regexPatterCreate = @"(PROCEDURE|CREATE OR REPLACE PROCEDURE|procedure)\s*[^\(]*\([^\)]*\)\s*(IS|AS)";
Regex regxPatterCreateValue = new Regex(regexPatterCreate, RegexOptions.Multiline);
Match matchPatterCreateValue = regxPatterCreateValue.Match(lineText);
if (matchPatterCreateValue.Length > 0)
    lineText = lineText.Replace(matchPatterCreateValue.Value, "");

lineText = lineText.Replace("\r", "").Replace(";", "").Trim();
```

**Şekil 6.3** Koddaki sabit ifadelerin arındırılması

**2.Adım:** Düzenlenmiş sp bilgisinde satır bazında dönülerek ,satırdaki her bir kelimenin belirlediğimiz key Word leri içerip içermediği kontrol edilir. İçeriyorsa "idxNode" değeri 1 arttırlır , toplamEdge sayısı 3 arttırılır. Ve elde edilen parametre değerleri aşağıdaki formülde yerine konularak sonuç değeri elde edilir.

$$intSonuc = (toplamEdge + idxNode - 1 + 2) - (toplamNode + 2) + (2 * 1) \quad (6.1)$$

Karmaşıklık seviyesi artacak şekilde 1 den 4 e verilerek elde edilen sonuç aşağıdaki gibi yorumlanmıştır.

```
if (Sonuc >= 1 && Sonuc <= 10)
    dtr2["COMPLEXITYGROUP"] = 1;
else if (Sonuc >= 11 && Sonuc <= 20)
    dtr2["COMPLEXITYGROUP"] = 2;
else if (Sonuc >= 21 && Sonuc <= 50)
    dtr2["COMPLEXITYGROUP"] = 3;
else
    dtr2["COMPLEXITYGROUP"] = 4;
```

**Şekil 6.4** Koddaki karmaşıklık seviyesinin yorumlanması

### 6.1.2 Halstead Metrıcs Kullanılarak Karmaşıklığın Hesaplanması

**1.Adım:**

lk aşamada hesaplamada kullanılacak operand ve operatörler belirlenmiştir. Belirlenen listeler aşağıdaki gibidir;

string[] keyWordSetsOperators = "END LOOP", "END IF", "ELSIF", "=", "-", "+", "/", "*", "++", "RETURN", "<", ">", "VARCHAR2", "NUMBER", "IF", "SELECT", "INSERT", "DELETE", "UPDATE", "COMMIT", "DISTINCT", "WHEN", "CASE", "THEN", "FROM", "WHERE","INTO" ;

string[] keyWordSetsOperands = "0","1", "2","3","4","5","6","7","8","9" ,"p

int n1 = 0; // number of uniq operators

int n2 = 0; // number of uniq operands

int N1 = 0; // Total number of operators

int N2 = 0; // Total number of operands

Store Procedure (sp) lerin olduğu tablodan okuma yapılır , her bir sp nin içeriği önce regex kütüphanesi ile düzenlenir. Düzenleme aşamaları aşağıdaki adımları içermektedir;

Yoruma alınmış (commentlenmiş) satırlardan temizlenmesi için bir regex ifadesi yazıldı. Aşağıdaki şekilde sp nin içeriği (linetext) ifadeye verilerek , commentlenmiş satırlar "" ifadesi ile replace edildi.

```
string regexPatternCommentMulti = @"\s*\-\-.*?\n";
Regex regexPatternCommentMultiValue = new Regex(regexPatternCommentMulti, RegexOptions.Multiline);
MatchCollection matchesCommentMultiValue = regexPatternCommentMultiValue.Matches(lineText);
foreach (var item in matchesCommentMultiValue)
{
    string x = item.ToString();
    lineText = lineText.Replace(x, "");
}
```

**Şekil 6.5** Kod içerisinde yorum satırlarının arındırılması

Yorum satırlarından kaçan tarih gibi ifadeleri ayıklamak için aşağıdaki gibi bir regex ifadesi yazılmıştır.

```
Regex regexPatternCommentMulti1 = new Regex(@"\/\*[^\/]*\/", RegexOptions.Multiline);
MatchCollection regexPatternCommentMulti1Value = regexPatternCommentMulti1.Matches(lineText);
foreach (var item in regexPatternCommentMulti1Value)
{
    string x = item.ToString();
    lineText = lineText.Replace(x, "");
}
```

**Şekil 6.6** Kodun istenmeyen bileşenlerden arındırılması

Üstteki adımın benzeri şekilde sp nin create edildiği satırı kaldırmak için bir regex ifadesi yazıldı ve sp bu ifadeden arındırıldı.

**2.Adım:**

```
string regexPatterCreate = @"(PROCEDURE|CREATE OR REPLACE PROCEDURE|procedure)\s*[^\(]*\([^\)]*\)\s*(IS|AS)";
Regex regxPatterCreateValue = new Regex(regexPatterCreate, RegexOptions.Multiline);
Match matchPatterCreateValue = regxPatterCreateValue.Match(lineText);
if (matchPatterCreateValue.Length > 0)
    lineText = lineText.Replace(matchPatterCreateValue.Value, "");

lineText = lineText.Replace("\r", "").Replace(";", "").Trim();
```

**Şekil 6.7** Koddaki sabit ifadelerin arındırılması

Düzenlenmiş sp bilgisinde satır bazında dönülerek ,satırdaki her bir kelimenin belirlediğimiz operand ve operatörleri içerip içermediği kontrol edilir. İçeriyorsa n1 ve n2 değerleri 1 arttırlır , toplam sayıları da N1 ve N2 değerlerine atılır.

Bulunan değerler sonucunda Halstead metrikleri aşağıdaki gibi yazdırılır.

```
Console.WriteLine("N1",N1);
Console.WriteLine("N2",N2);
Console.WriteLine("n1",n1);
Console.WriteLine("n2",n2);
Console.WriteLine("N", N1+N2);
Console.WriteLine("V",(N1+N2)* Math.Log((n1+n2),2));//V = N * log2 n
Console.WriteLine("D", (n1 / 2) * (N2 / n2)); //D = (n1 / 2) * (N2 / n2)
Console.WriteLine("E" ,D+V); //E = D * V
Console.WriteLine("T" , (D+V)/18);//T = (E / 18)
Console.WriteLine("B" , V/3000);//B=V / 3000
```

**Şekil 6.8** Halstead metriklerinin yazdırılması

N1=95 ,n1=11 N2=51 ,n2=14 N=146 V=146/Log2(26)=103,182 D=20,035 E=123,217 T=6,845 B=0,03

### 6.1.3 Kurulan Yeni Model ile Karmaşıklığın Hesaplanması

İşlem 4 adımdan oluşmaktadır. Adımlar sırasıyla, temel anlamda aşağıdaki işlevleri yerine getirmektedir.

**1.Adım:** SP nesnesi parse edilip kelime bazında her bir texti bir tabloda tutar. Hesaplaması yapılacak olan sp nin her bir satırı satır sırasına göre “v_ObjectList” adında bir cursor a atılır.Her satırdaki kelimler boşluk ve virgül karakterine göre ayrılıp “TT_OBJECTTEXTPARSER” geçici temp tablosuna atılır.

**2.Adım:** 1.Adımda elde edilen textler ayıklanarak, karmaşıklık hesaplamada kullanılacak textler belirlenir.

TT_OBJECTTEXTPARSER temp tablosundaki verilerden anlam ifade etmeyen

| | OWNER | NAME | TEXT | LINE |
|---|---|---|---|---|
| 1 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | PROCEDURE MNK_SEL_MUSTERILISTESI1401K_SP | 1 |
| 2 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | ( | 2 |
| 3 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_Soyadi IN VARCHAR2, | 3 |
| 4 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_MusteriTipi IN NUMBER, | 4 |
| 5 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_AltLimit IN NUMBER, | 5 |
| 6 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_UstLimit IN NUMBER, | 6 |
| 7 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_DovizCinsi IN NUMBER, | 7 |
| 8 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | p_Kapali IN VARCHAR2, --E:Kapalı hesaplar da gelsin, H:Kapalı hesapla | 8 |
| 9 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | RC1 IN OUT omwb_emulation.globalpkg.RCT1 | 9 |
| 10 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | ) AS | 10 |
| 11 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | | 11 |
| 12 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | --Yazan/Tarih : Erdal Akyildiz, 27.06.2005 | 12 |
| 13 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | -- Rev : h.Hsp_Kapali_Kod "KapaliKod" eklendi-e.ö 21/11/2005 | 13 |
| 14 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | v_TCMB_MusteriNo NUMBER; | 14 |
| 15 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | v_IMKB_MusteriNo NUMBER; | 15 |
| 16 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | BEGIN | 16 |

**Şekil 6.9** Saklı yordam parse işlemi

yani objecttext i boşluk , parantez, VARCHAR2(%, NUMBER(%, CHAR(% , TIMESTAMP(%,%_SP(% karakterleri atılarak TT_OBJECTTEXTPARSERDETAIL temp detay tablosuna atılır. Ayrıca bu tabloda bir sonraki aşamada mapping yapılacak anahtar kelime ifadesinin hangi satırda başlayıp ne kadar sürdüğünün bulunması için bu aşamada her bir satır için satır numarası bilgisi tutulur.

**3.Adım:** Ayıklanmış olan veriler burada, maplenerek (örneğin; Loop ve If conditionlarının başlangıç ve bitişleri, derinlikleri vs.) parse edilen SP nesneleri ve varsa derinlikleri 2 farklı tabloda tutulur.

```
v_ParameterNameInput          VARCHAR2(16) := 'INPUTPARAMETER';
v_ParameterNameValue          VARCHAR2(16) := 'DECLAREVALUE';
v_ParameterNameCondition      VARCHAR2(16) := 'CONDITIONAL';
v_ParameterNameLoop           VARCHAR2(16) := 'LOOP';
v_ParameterNameDML            VARCHAR2(16) := 'DML MANIPULATION';
v_ParameterNameProcedure      VARCHAR2(16) := 'PROCEDURE';
v_ParameterNameUdf            VARCHAR2(16) := 'FUNCTION';
v_ParameterNameFrom           VARCHAR2(20) := 'FROM';
```

**Şekil 6.10** Kullanılan parametre isimleri

Bu adımın ilk aşamasında bir önceki adımda elde edilen veriler çekilerek bir cursora atılır. Aşağıda “LOOP” anahtar kelimesi için bu bilgilerin elde edilişi gösterilmektedir. Karmaşıklığını hesapladığımız saklı yordam içerisindeki tüm “LOOP” ifadelerinin bilgilerini elde etmiş oluruz.Aynı işlem “IF” iadeleri için de yapılmaktadır.

Elde edilen bilgilerle saklı yordam da bulunma durumuna göre , geçici tablodan bilgiler map işlemi yapılarak ana tabloya alınır.

Aşağıda conditional yani “IF” ve döngü yani "LOOP" ifadelerinin maplenmesini görüyoruz burada “objectOwner” değişkeni veri tabanı adını ifade ederken

```
CURSOR v_LoopInformation IS
   SELECT *
     FROM (SELECT a.objecttext,
                  a.objecttextline,
                  a.objectcontroltype,
                  a.objectowner,
                  a.objectname,
                  CASE
                     WHEN a.OBJECTTEXT = 'LOOP' AND LEAD(a.OBJECTTEXT, 1) OVER(ORDER BY a.OBJECTTEXTLINE) = 'LOOP;' THEN
                      LEAD(a.OBJECTTEXTLINE, 1) OVER(ORDER BY a.OBJECTTEXTLINE)
                     ELSE
                      0
                  END CHECKNEXTTEXTRESULT
             FROM TT_OBJECTTEXTPARSERDETAIL a
            WHERE a.objectcontroltype = 'LOOP'
              AND a.objectsituation = 0
              AND a.objectowner = v_ObjectOwner
              AND a.objectname = v_ObjectName) x
    WHERE x.CHECKNEXTTEXTRESULT <> 0;
```

**Şekil 6.11** Geçici tablodan LOOP bilgisinin alınması

```
CURSOR v_ConditionalInformation IS
   SELECT *
     FROM (SELECT a.objecttext,
                  a.objecttextline,
                  a.objectcontroltype,
                  a.objectowner,
                  a.objectname,
                  CASE
                     WHEN a.OBJECTTEXT = 'IF' AND LEAD(a.OBJECTTEXT, 1) OVER(ORDER BY a.OBJECTTEXTLINE) = 'IF;' THEN
                      LEAD(a.OBJECTTEXTLINE, 1) OVER(ORDER BY a.OBJECTTEXTLINE)
                     ELSE
                      0
                  END CHECKNEXTTEXTRESULT
             FROM TT_OBJECTTEXTPARSERDETAIL a
            WHERE a.objectcontroltype = 'IF'
              AND a.objectsituation = 0
              AND a.objectowner = v_ObjectOwner
              AND a.objectname = v_ObjectName) x
    WHERE x.CHECKNEXTTEXTRESULT <> 0;
```

**Şekil 6.12** Geçici tablodan IF bilgisinin alınması

(development , test gibi farklı db ortamları için) “objectName” değişkeni işlem yapılan saklı yordamın adıdır.

```
----************************** MAPPING ************ -------------
--- 6.1 CONDITION MAPPING
BEGIN
   SELECT COUNT(*)
     INTO v_CountConditional
     FROM TT_OBJECTTEXTPARSERDETAIL tt
    WHERE tt.objectcontroltype = 'IF'
      AND tt.objectsituation = 0
      AND tt.objectowner = v_ObjectOwner
      AND tt.objectname = v_ObjectName;

   v_NewParameterText := 'IF - END IF;';

   WHILE v_CountConditional > 0
   LOOP
      FOR v_TempConditionalMapping IN v_ConditionalInformation
      LOOP
         IF v_TempConditionalMapping.OBJECTTEXT = 'IF' THEN
            INSERT INTO OBJECTTEXTPARSERDETAIL
               (objectcontroltype,
                objecttext,
                objecttextstartline,
                objecttextfinishline,
                objectlineid,
                objectowner,
                objectname)
            VALUES
               (v_ParameterNameCondition,
                v_NewParameterText,
                v_TempConditionalMapping.OBJECTTEXTLINE,
                v_TempConditionalMapping.CHECKNEXTTEXTRESULT,
                OBJECTTEXTPARSERDETAIL_SEQ.Nextval,
                TRIM(v_TempConditionalMapping.objectowner),
                TRIM(v_TempConditionalMapping.objectname));
            COMMIT;

            UPDATE TT_OBJECTTEXTPARSERDETAIL tt
               SET tt.objectsituation = 1
             WHERE tt.objectcontroltype = v_TempConditionalMapping.objectcontroltype
               AND tt.objectowner = TRIM(v_TempConditionalMapping.objectowner)
               AND tt.objectname = TRIM(v_TempConditionalMapping.objectname)
               AND tt.objecttextline IN (v_TempConditionalMapping.OBJECTTEXTLINE, v_TempConditionalMapping.CHECKNEXTTEXTRESULT);
            COMMIT;
         END IF;
      END LOOP;


      SELECT COUNT(*)
        INTO v_CountConditional
        FROM TT_OBJECTTEXTPARSERDETAIL tt
       WHERE TRIM(upper(tt.objectcontroltype)) = 'IF'
         AND tt.objectsituation = 0
         AND tt.objectowner = v_ObjectOwner
         AND tt.objectname = v_ObjectName;
   END LOOP;
END;
```

**Şekil 6.13** Koşul ifadesi map işlemi

Derinlik hesabı satır bilgisi ile yapılır. Bulunan ilk IF veya LOOP ile END ifadeleri bir blok oluşturmaktadır. Ve bunların derinliği 1’dir. Bu bloğun içinde bulunan IF veya LOOP ifadelerinin derinlikleri bir artarak ilerler. İçinde olmayan dışındaki bir blok için derinlik yeniden 1 olacak şekilde devam eder. Derinlik bulma işlemi için öncelikle “OBJECTTEXTPARSERDETAIL” tablosundan bilgilerle “OBJECTPARSERDETAILDEPTH” tablosu doldurulur sonrasında derinlik bulma işlemine geçilir.

**4.Adım:** Bu adımda tüm veriler parametrik olarak tasarlanan hesaplama yapısına göre hesaplanarak puanlama yapılmıştır.

4.Adımda, tasarlanan parametrik hesaplama yapısı şu özelliklere sahiptir:

```
--- 6.2  LOOP MAPPING

BEGIN
   SELECT COUNT(*)
     INTO v_CountLoop
     FROM TT_OBJECTTEXTPARSERDETAIL tt
    WHERE tt.objectcontroltype = 'LOOP'
      AND tt.objectsituation = 0
      AND tt.objectowner = v_ObjectOwner
      AND tt.objectname = v_ObjectName;

   v_NewParameterText := 'LOOP - END LOOP;';

   WHILE v_CountLoop > 0
   LOOP
      FOR v_TempLoopMapping IN v_LoopInformation
      LOOP
         IF v_TempLoopMapping.OBJECTTEXT = 'LOOP' THEN
            INSERT INTO OBJECTTEXTPARSERDETAIL
               (objectcontroltype,
                objecttext,
                objecttextstartline,
                objecttextfinishline,
                objectlineid,
                objectowner,
                objectname)
            VALUES
               (v_ParameterNameLoop,
                v_NewParameterText,
                v_TempLoopMapping.OBJECTTEXTLINE,
                v_TempLoopMapping.CHECKNEXTTEXTRESULT,
                OBJECTTEXTPARSERDETAIL_SEQ.Nextval,
                TRIM(v_TempLoopMapping.objectowner),
                TRIM(v_TempLoopMapping.objectname));
            COMMIT;

            UPDATE TT_OBJECTTEXTPARSERDETAIL tt
               SET tt.objectsituation = 1
             WHERE tt.objectcontroltype = v_TempLoopMapping.objectcontroltype
               AND tt.objectowner = TRIM(v_TempLoopMapping.objectowner)
               AND tt.objectname = TRIM(v_TempLoopMapping.objectname)
               AND tt.objecttextline IN (v_TempLoopMapping.OBJECTTEXTLINE, v_TempLoopMapping.CHECKNEXTTEXTRESULT);
            COMMIT;
         END IF;
      END LOOP;


      SELECT COUNT(*)
        INTO v_CountLoop
        FROM TT_OBJECTTEXTPARSERDETAIL tt
       WHERE TRIM(upper(tt.objectcontroltype)) = 'LOOP'
         AND tt.objectsituation = 0
         AND tt.objectowner = v_ObjectOwner
         AND tt.objectname = v_ObjectName;
   END LOOP;
END;
```

**Şekil 6.14** Döngü ifadesi map işlemi

```
select * from OBJECTTEXTPARSERDETAIL where  OBJECTNAME='FDS_SEL_RAPORKREDIDETAY_SP'
```

| | OBJECTCONTROLTYPE | OBJECTTEXT | OBJECTTEXTSTARTLINE | OBJECTTEXTFINISHLINE | OBJECTLINEID | OBJECTOWNER | OBJECTNAME |
|---|---|---|---|---|---|---|---|
| 1 | INPUTPARAMETER | p_SubeKodu | 3 | -1 | 22810 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 2 | INPUTPARAMETER | p_PortfoyNo | 4 | -1 | 22811 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 3 | INPUTPARAMETER | p_IslemTipi | 5 | -1 | 22812 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 4 | INPUTPARAMETER | p_MusteriNo | 6 | -1 | 22813 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 5 | INPUTPARAMETER | RC1 | 7 | -1 | 22814 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 6 | DECLAREVALUE | C_PARAMETREGECERLIGUN | 17 | -1 | 22815 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 7 | DECLAREVALUE | C_PARAMETREKIMMODEL | 18 | -1 | 22816 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 8 | DECLAREVALUE | C_PARAMETRERAPORTURU | 19 | -1 | 22817 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 9 | DECLAREVALUE | C_RTFDEVAMEDEN | 20 | -1 | 22818 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 10 | DECLAREVALUE | C_KARGIRIS | 21 | -1 | 22819 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 11 | DECLAREVALUE | C_FARGIRIS | 22 | -1 | 22820 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 12 | DECLAREVALUE | C_FARONAYDA | 23 | -1 | 22821 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 13 | DECLAREVALUE | C_KARONAYDA | 24 | -1 | 22822 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 14 | DECLAREVALUE | C_RAPORONAYLI | 25 | -1 | 22823 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 15 | DECLAREVALUE | v_GecerlilikGunSayisi | 27 | -1 | 22824 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 16 | DECLAREVALUE | v_TahsisMaksGecerlilikSure | 28 | -1 | 22825 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 17 | DECLAREVALUE | v_TahsisGecerlilikSureBolge | 29 | -1 | 22826 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 18 | DECLAREVALUE | v_TahsisGecerlilikSureSube | 30 | -1 | 22827 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 19 | DECLAREVALUE | v_TahsisGecerlilikSureBirim | 31 | -1 | 22828 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 20 | DECLAREVALUE | v_SimdikiZamanGun | 32 | -1 | 22829 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |
| 21 | DECLAREVALUE | v_KimModelListe | 33 | -1 | 22830 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP |

**Şekil 6.15** Map işlemi sonrası tablo görüntüsü

```
BEGIN

   FOR v_temp IN (SELECT *
                    FROM TT_OBJECTPARSERDETAILDEPTH A
                   WHERE A.OBJECTCONTROLTYPE = v_ParameterNameCondition
                     AND A.OBJECTOWNER = v_ObjectOwner
                     AND A.OBJECTNAME = v_ObjectName
                   ORDER BY OBJECTTEXTLINE)
   LOOP
      IF TRIM(upper(v_temp.objectstatus)) = 'IF' THEN
         v_Counter := v_Counter + 1;
      ELSIF TRIM(upper(v_temp.objectstatus)) = 'END IF;' THEN
         v_Counter := v_Counter - 1;
      END IF;

      UPDATE OBJECTPARSERDETAILDEPTH odpt
         SET odpt.depthlevel = v_Counter
       WHERE odpt.objectcontroltype = v_temp.objectcontroltype
         AND odpt.depthlevel = 0
         AND odpt.objectowner = v_temp.objectowner
         AND odpt.objectname = v_temp.objectname
         AND odpt.objecttextstartline = v_temp.objecttextline
         AND odpt.objectparentid = 0;
      COMMIT;
   END LOOP;
END;
```

**Şekil 6.16** Derinlik bulma işlemi

```
select * from OBJECTPARSERDETAILDEPTH where OBJECTNAME='FDS_SEL_RAPORKREDIDETAY_SP'
```

| | OBJECTCONTROLTYPE | OBJECTTEXTSTARTLINE | OBJECTTEXTFINISHLINE | OBJECTID | DEPTHLEVEL | OBJECTOWNER | OBJECTNAME | OBJEC |
|---|---|---|---|---|---|---|---|---|
| 1 | CONDITIONAL | 50 | 71 | 22836 | 1 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 2 | CONDITIONAL | 57 | 63 | 22831 | 2 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 3 | CONDITIONAL | 65 | 67 | 22832 | 2 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 4 | CONDITIONAL | 74 | 86 | 22833 | 1 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 5 | CONDITIONAL | 89 | 629 | 22837 | 1 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 6 | CONDITIONAL | 91 | 351 | 22834 | 2 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 7 | CONDITIONAL | 355 | 627 | 22835 | 2 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 8 | LOOP | 56 | 69 | 22838 | 1 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |
| 9 | LOOP | 80 | 84 | 22839 | 1 | FINARTYTL | FDS_SEL_RAPORKREDIDETAY_SP | |

**Şekil 6.17** Derinlik bulma işlemi sonrası tablo görüntüsü

Hesaplamada kullanılacak featureların belirlendiği bir tanım tablosu bulunmaktadır. Burada , yukarıdaki 4 adımda elde edilen bilgiler üzerinden bir feature exraction çalışması yapılmış , belirlenen featureları katsayılarına göre hesaplama da kullanılmıştır.

```
select * from parserfeaturedefiniton
```

| | OBJECTFEATUREID | OBJECTFEATUREDESC | OBJECTFEATURERECORDSTATUS |
|---|---|---|---|
| 1 | 1 | TYPE | 1 |
| 2 | 2 | DEPTH | 1 |
| 3 | 3 | COUNT | 1 |
| 4 | 4 | LEVEL | 1 |

**Şekil 6.18** Çıkartılan nitelikler

Hesaplamada kullanılacak diğer bir tanım tablosu ise, parser işlemi için belirlenen içerik olarak kontrol tiplerini barındıran tablodur. Burada da keywordleri(hesaplamada kullanılacak kontrol türleri) belirlemek için bir çalışma yapılmıştır.Objectcontrolrecordstatus alanı 1 ve 0 değerlerini almaktadır. 0 ise hesaplamaya katılmamaktadır.

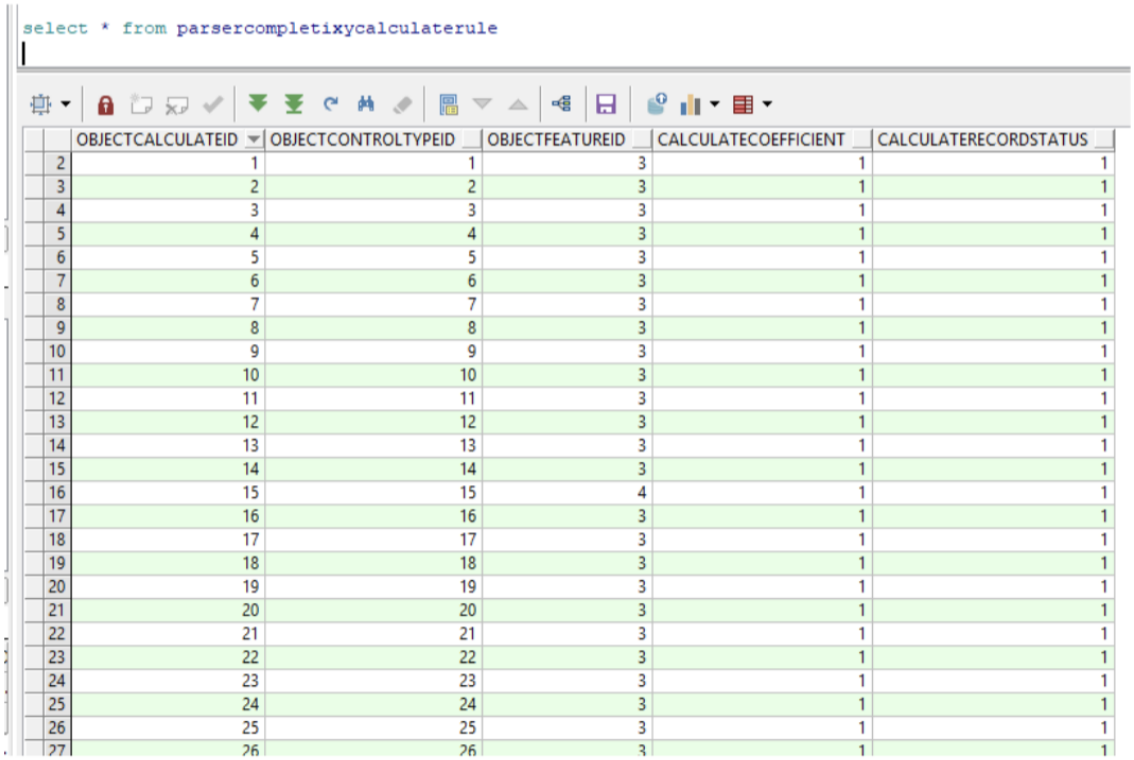
```
select * from parsercontroldefiniton
```

| | OBJECTCONTROLTYPEID | OBJECTCONTROLTYPE | OBJECTCONTROLRECORDSTATUS | OBJECTCONTROLDESC |
|---|---|---|---|---|
| 1 | 1 | CONDITIONAL | 1 | TOTAL |
| 3 | 2 | CONDITIONAL | 0 | OPERATION |
| 4 | 3 | CONDITIONAL | 0 | SELECT |
| 5 | 4 | CONDITIONAL | 0 | UPDATE |
| 6 | 5 | CONDITIONAL | 0 | PROCEDURE |
| 7 | 6 | CONDITIONAL | 0 | INTERNALPROCEDURE |
| 8 | 7 | CONDITIONAL | 0 | IF |
| 9 | 8 | CONDITIONAL | 0 | ASSIGN |
| 10 | 9 | CONDITIONAL | 0 | FETCH |
| 11 | 10 | CONDITIONAL | 0 | LOOP |
| 12 | 11 | CONDITIONAL | 0 | INSERT |
| 13 | 12 | CONDITIONAL | 0 | DELETE |
| 14 | 13 | CONDITIONAL | 0 | UDF |
| 15 | 14 | CONDITIONAL | 0 | INTERNALUDF |
| 16 | 16 | CONDITIONAL | 0 | NESTEDUDF |
| 17 | 17 | LOOP | 1 | TOTAL |
| 18 | 18 | LOOP | 0 | OPERATION |

**Şekil 6.19** Tanım tablosu

on olarak oluşturulan hesaplama kuralları tablosu; feature tanım tablosu ve kontrol tip tanım tablolarının crossunu içeren, hesaplama kurallarının olduğu tablodur. Bu

tabloda, her bir kontrol tipi için kullanılmak istenen her bir feature'ın, puanlamanın oluşmasında kullanılacak , katsayılar ve o kontrol tıpında, o feature'ın kullanılıp kullanılmama durumları tutulmaktadır.Her bir hesaplama kuralının etkisi 1 olarak belirlenmiştir.(calculateofficent)



```
select * from parsercompletixycalculaterule
```

| | OBJECTCALCULATEID | OBJECTCONTROLTYPEID | OBJECTFEATUREID | CALCULATECOEFFICIENT | CALCULATERECORDSTATUS |
|---|---|---|---|---|---|
| 2 | 1 | 1 | 3 | 1 | 1 |
| 3 | 2 | 2 | 3 | 1 | 1 |
| 4 | 3 | 3 | 3 | 1 | 1 |
| 5 | 4 | 4 | 3 | 1 | 1 |
| 6 | 5 | 5 | 3 | 1 | 1 |
| 7 | 6 | 6 | 3 | 1 | 1 |
| 8 | 7 | 7 | 3 | 1 | 1 |
| 9 | 8 | 8 | 3 | 1 | 1 |
| 10 | 9 | 9 | 3 | 1 | 1 |
| 11 | 10 | 10 | 3 | 1 | 1 |
| 12 | 11 | 11 | 3 | 1 | 1 |
| 13 | 12 | 12 | 3 | 1 | 1 |
| 14 | 13 | 13 | 3 | 1 | 1 |
| 15 | 14 | 14 | 3 | 1 | 1 |
| 16 | 15 | 15 | 4 | 1 | 1 |
| 17 | 16 | 16 | 3 | 1 | 1 |
| 18 | 17 | 17 | 3 | 1 | 1 |
| 19 | 18 | 18 | 3 | 1 | 1 |
| 20 | 19 | 19 | 3 | 1 | 1 |
| 21 | 20 | 20 | 3 | 1 | 1 |
| 22 | 21 | 21 | 3 | 1 | 1 |
| 23 | 22 | 22 | 3 | 1 | 1 |
| 24 | 23 | 23 | 3 | 1 | 1 |
| 25 | 24 | 24 | 3 | 1 | 1 |
| 26 | 25 | 25 | 3 | 1 | 1 |
| 27 | 26 | 26 | 3 | 1 | 1 |

**Şekil 6.20** Özelliklerin hesaplamalara etkileri

| | OBJECTCONTROLTYPE | OBJECTFEATUREDESC | OBJECTCONTROLDESC | CALCULATECOEFFICIENT | CALCULATERECORDSTATUS |
|---|---|---|---|---|---|
| 1 | CONDITIONAL | COUNT | TOTAL | 1 | 1 |
| 2 | CONDITIONAL | LEVEL | MAX DEPTH | 1 | 1 |
| 3 | DECLAREVALUE | COUNT | TOTAL | 1 | 1 |
| 4 | DML MANIPULATION | COUNT | DELETE | 1 | 1 |
| 5 | DML MANIPULATION | COUNT | INSERT | 1 | 1 |
| 6 | DML MANIPULATION | COUNT | UPDATE | 1 | 1 |
| 7 | FROM | COUNT | TOTAL | 1 | 1 |
| 8 | FUNCTION | COUNT | TOTAL | 1 | 1 |
| 9 | INPUTPARAMETER | COUNT | TOTAL | 1 | 1 |
| 10 | JOIN | COUNT | TOTAL | 1 | 1 |
| 11 | LOOP | LEVEL | MAX DEPTH | 1 | 1 |
| 12 | LOOP | COUNT | TOTAL | 1 | 1 |
| 13 | PARSERSP | COUNT | CODELINE | 1 | 1 |
| 14 | PROCEDURE | COUNT | TOTAL | 1 | 1 |
| 15 | SELECT | COUNT | TOTAL | 1 | 1 |
| 16 | WHERE | COUNT | TOTAL | 1 | 1 |

**Şekil 6.21** Tanım tablolarının birleşim sonucu

**Hesaplama:**Cross yapılan tablodan sırasıyla calculaterecordstatus ü 1 olan yani hesaplamaya dahil edilecek kurallar çekilerek bir loop içerisinde objecttextparserdetail tablosundan , objectcontroltype ı o kuralı sağlayan kayıtların count u alınır ve o kuralın etki katsayısı (hepsi bir olarak alınmıştır) ile çarpılarak genel toplama eklenir. Tüm

kuralların sonunda elde ettiğimiz toplam çalıştığımız saklı yordamın karmaşıklığını verir.

```
FOR v_Calculate IN (SELECT TRIM(d.objectcontroltype) AS objectcontroltype,
                           TRIM(f.objectfeaturedesc) AS objectfeaturedesc,
                           TRIM(d.objectcontroldesc) AS objectcontroldesc,
                           r.calculatecoefficient,
                           r.calculaterecordstatus
                      FROM parsercompletixycalculaterule r
                      JOIN parsercontroldefiniton d
                        ON r.objectcontroltypeid = d.objectcontroltypeid
                       AND r.calculaterecordstatus = d.objectcontrolrecordstatus
                       AND r.calculaterecordstatus = 1
                      JOIN parserfeaturedefiniton f
                        ON f.objectfeatureid = r.objectfeatureid
                       AND f.objectfeaturerecordstatus = r.calculaterecordstatus
                       AND f.objectfeaturerecordstatus = 1
                     ORDER BY OBJECTCONTROLTYPE)
LOOP
   IF v_Calculate.Objectcontroltype = 'CONDITIONAL' AND v_Calculate.Objectfeaturedesc = 'COUNT' AND v_Calculate.Objectcontroldesc = 'TOTAL' THEN
      --- total count
      SELECT COUNT(*)
        INTO v_TotalConditional
        FROM objecttextparserdetail DET
       WHERE det.objectcontroltype = v_Calculate.Objectcontroltype
         AND det.objectowner = v_ObjectOwner
         AND det.objectname = v_ObjectName;

      v_Result := v_Result + v_TotalConditional * v_Calculate.Calculatecoefficient;
```

**Şekil 6.22** Karmaşıklık hesaplama çalışması

| | ID | OBJECTOWNER | OBJECTNAME | COMPLEXITYRESULT |
|---|---|---|---|---|
| 1 | 13 | FINARTYTL | PLM_INS_EMAILSENDER_SP | 52 |
| 2 | 16 | FINARTYTL | BLD_UPD_SENETBLDONAYREF_SP | 58 |
| 3 | 17 | FINARTYTL | FDS_SEL_OZELDURUMTELCO_SP | 44 |
| 4 | 19 | FINARTYTL | ATM_UPD_SLAFLMREACTIVATION_SP | 312 |
| 5 | 20 | FINARTYTL | DTI_SEL_GIDENHAVALEMUKERRER_SP | 98 |
| 6 | 21 | FINARTYTL | BKR_SEL_MSTKAMPANYAGRUPLMT_SP | 78 |
| 7 | 22 | FINARTYTL | BKR_UPD_ONAYAGONDERILDI_SP | 497 |
| 8 | 23 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401_SP | 664 |
| 9 | 24 | FINARTYTL | MNK_SEL_KIYMETGUNBILGILERI_SP | 144 |
| 0 | 25 | FINARTYTL | KKB_SEL_DNMGECIKMESAYKMHSIZ_SP | 455 |
| 1 | 26 | FINARTYTL | MNK_SEL_MUSTERILISTESI1401K_SP | 767 |

**Şekil 6.23** Test sonuçlarında elde edilen puanlamalar

# 7
# BULGULAR VE DEĞERLENDİRMELER

## 7.1 Bulgular

Tez kapsamında toplanan , farklı işlevlere sahip saklı yordamların kurulan yeni modelle karmaşıklıkları ölçülmüş ve bu karmaşıklığa etki eden kriterler çıkarılmıştır. Bu kriterler, iki ya da daha çok değişkenin arasındaki ilişkiyi ölçmek için kullanılan regresyon analizine tabi tutulmuştur.Sonucun tamamen değişkenlere bağlı olduğu görülmüştür. Karmaşıklığı ölçülen saklı yordam sayısı 69'dur.

| *Regression Statistics* | |
|---|---|
| Multiple R | 0,999907604 |
| R Square | 0,999815216 |
| Adjusted R Square | 0,999790578 |
| Standard Error | 31,26699507 |
| Observations | 69 |

**Şekil 7.1** Regresyon analiz istatistikleri

Regresyon test sonuçlarını yorumlarken anlamlılık değeri olan p değerleri için 0,05'ten küçük olanlar anlamlı olarak kabul edilmiştir.

| | *Coefficients* | *Standard Error* | *t Stat* | *P-value* | *Lower 95%* | *Upper 95%* | *Lower 95,0%* | *Upper 95,0%* |
|---|---|---|---|---|---|---|---|---|
| KARMAŞIKLIK SONUCU | 10,84677634 | 7,580143832 | 1,430945979 | 0,157633657 | -4,315768858 | 26,00932154 | -4,315768858 | 26,00932154 |
| KOD SATIR SAYISI | 1,027174132 | 0,012606937 | 81,4768969 | 3,78605E-63 | 1,001956503 | 1,052391762 | 1,001956503 | 1,052391762 |
| LOOP SAYISI | 1,904261707 | 1,780069659 | 1,069768083 | 0,2890085 | -1,656407755 | 5,464931168 | -1,656407755 | 5,464931168 |
| WHERE SAYISI | -0,374753637 | 0,404256402 | -0,927019672 | 0,357630394 | -1,183386838 | 0,433879563 | -1,183386838 | 0,433879563 |
| SELECT SAYISI | 1,457338559 | 1,227403216 | 1,187334806 | 0,239775045 | -0,99783342 | 3,912510539 | -0,99783342 | 3,912510539 |
| JOIN SAYISI | 0,199132682 | 0,186447868 | 1,068034109 | 0,289783512 | -0,173818581 | 0,572083945 | -0,173818581 | 0,572083945 |
| IF SAYISI | 2,328734293 | 0,401912149 | 5,794137598 | 2,70425E-07 | 1,524790297 | 3,132678289 | 1,524790297 | 3,132678289 |
| FROM SAYISI | 1,833396053 | 0,904469396 | 2,027040451 | 0,047108353 | 0,024187889 | 3,642604216 | 0,024187889 | 3,642604216 |
| DML MANIPUALTION (INS,UPD,DEL) | 1,130931836 | 0,846076143 | 1,336678555 | 0,186374193 | -0,56147243 | 2,823336103 | -0,56147243 | 2,823336103 |

**Şekil 7.2** Regresyon analiz sonuçları

## 7.2 Değerlendirmeler

Regresyon analizi sonucunda elde edilen sonuçlar bu kısımda yorumlanmıştır.

### 7.2.1 Kod Satır Sayısı ile Karmaşıklık Arasındaki ilişkinin Değerlendirilmesi

Yapılan analiz sonucu incelendiğinde kod satır sayısının karmaşıklık sonucunu doğrudan etkileyen bir faktör olduğu görülmüştür.Karmaşıklığın belirli sınırlar içerisinde tutulmasını sağlamak için kod satır sayısına güvenli sayılabilen maksimum bir değer ataması yapılabilir. Eğer yazılan saklı yordam bu değeri aşarsa geliştiriciye güvenli sınırın aşıldığını , yazılan saklı yordamın gereksiz kod satırlarından arındırılması gerektiği uyarısı yapılabilir. Saklı yordamdan kod silinemiyorsa , anlamlı işlere bölünerek bölünen bu işlemler farklı bir saklı yordama alınabilir ve ana saklı yordamın içerisinden çağıralabilir.

### 7.2.2 Saklı Yordamları Oluşturan Ana Metrikler ile Karmaşıklık Arasındaki ilişkinin Değerlendirilmesi

Farklı kullanıcılar tarafından yazılan farklı işlevlere sahip incelenen 69 adet saklı yordamın anahtar kelimelerine bakıldığında ; 146 adet "loop" ifadesi , 4614 adet "where" ifadesi, 3506 adet "select" ifadesi , 3090 adet "join" ifadesi , 1006 adet "if" ifadesi , 3264 adet "from" ifadesi ve 292 adet dml manipulation ifadesi yer almaktadır.

Karmaşıklık ölçümü yapılan tüm modellerde döngü ifadeleri karmaşıklığı etkileyen önemli bir faktördür ancak toplanan veriler incelendiğinde saklı yordam yazarken geliştiricilerin döngülerden kaçındığını ve fazla kullanmadığını ortaya çıkmıştır.

Analiz sonuçları incelendiğinde karmaşıklık sonucuna kod satır sayısından sonra etki eden en önemli faktörün saklı yordam yazımında sıklıkla kullanılan ve kod içerisinde dallanmayı sağlayan "IF" ifadeleri olduğu görülmüştür. Bu nedenle modelimizde karmaşıklık hesaplarken bu ifadenin kat sayısı diğerlerinden fazla verilmelidir.

Regresyon analiz sonuçlarındaki p anlamlılık değerine bakıldığında , karmaşıklık sonucunu doğrudan etkileyen üçüncü bir faktörün kullanılan "From" ifadeleri olduğu görülmüştür. Yazılan saklı yordam içerisinde gidilen tablo sayısı arttıkça karmaşıklığın arttığı sonucu çıkarılmıştır. Ortaya çıkacak olan karmaşıklığın kontrol altına alınabilmesi için gidilebilecek tablo sayısı güvenli sınırlar altında tutulabilir , bunu aşan geliştiriciler derleme aşamasında uyarılabilir. Çözüm için yapılabiliyorsa yapılacak iş mikro işlemlere ayrılarak , bu işlemleri gerçekleştirecek mikro saklı yordamların yazılması ve ana saklı yordamdan çağrılması sağlanabilir.

# 8
# SONUÇ VE ÖNERİLER

## 8.1 Sonuç

Yazılım projelerinde kalite ölçülerek bakım-onarım, işlevsellik ve güvenilirlik gibi önemli noktalarda oluşabilecek sorunlar önlenebilir. Bu tez çalışmasında literatürde yer alan yazılım kalite modelleri ve standartları incelenmeye çalışılmış ve bir yazılım sisteminin kalite değerlendirmesi için operasyonel hale getirilemeyecek kadar soyut olduğu görülmüştür.

Yazılım projeleri, ölçümler yapmak için yazılım metrikleri kullanılarak sayısallaştırabilir. Böylelikle mevcut modellerde elde edilen veriler karşılaştırılarak yazılımın durumu ve ilerleyişi hakkında bilgiler elde edilebilir ve yeni modeller geliştirilebilir.

Yapılacak olan ölçümler alınacak kararlar açısından güvenilir olmalı ve ortak bir anlayışla yorumlanmalıdır. Çalışmamızda görüldüğü üzere metrik araçları aynı girdiye göre farklı sonuçlar üretebilir. Bu nedenle tüm sonuçları kapsayacak böylece tutarsızlıkları azaltacak daha esnek ve geliştirilmeye açık bir yapı kurulmaya çalışılmıştır.

Karmaşıklık, kötü yazılım kalitesinin ana nedeni olduğu için her zaman kontrol altında tutulması gereken bir ölçüdür. Çalışmada, başta veritabanı nesneleri olmak üzere yazılım mimarisindeki farklı katmanların karmaşıklığının hesaplanmasında kullanılabilecek yöntemler vurgulanmış ve sonuçların hata tahmini üzerindeki etkileri analiz edilmiştir.

## 8.2 Öneriler

Yazılım projelerinde ekstra karmaşıklığa neden olarak sistemlerin performansını ve verimliliğini olumsuz yönde etkileyecek işlevsel olmayan gereksinimleri elemek için gereksinim analizi iyi yapılmalı ve bu noktada kalite standarları göz önünde

bulundurulmalıdır.

Karmaşıklığı belirli sınırlar dahilinde tutmak kod kalitesini arttıracağından yazılım geliştirme yaşam döngüsü boyunca karmaşıklığın ölçülmesi ve gerekli aksiyonların alınması sağlanmalıdır.

Karmaşıklığı analiz etmek için uygun ve verimli araçlara sahip olmak karmaşıklığın daha iyi anlaşılmasını sağlayacak ve proje içerisindeki tüm paydaşların karmaşıklığı nasıl yöneteceklerine dair rehber görevi görecektir.

## 8.3 Gelecek Çalışmalar

Veri tabanında oluşturulan saklı yordamların karmaşıklığının kontrol edilebilmesi için yazılan yeni modelin araç haline dönüştürülmesi ve bu aracın geliştirme ortamında kodun commit yapılmasından sonra diğer ortamlara taşınma aşamasında kullanılması , geliştiriciyi yönlendirecek olan ve DEĞERLENDİRMELER bölümünde örneklendirilen ve arttırılması düşünülen uyarıların verilmesi planlanmaktadır.

# KAYNAKÇA


[1] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

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

[3] D. Mola, "The effectiveness of code inspection on software quality," 2007.

[4] S. Delaune, F. Jacquemard, "A theory of dictionary attacks and its complexity," in *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, IEEE, 2004, pp. 2–15.

[5] V. R. Basili, "Qualitative software complexity models: A summary," *Tutorial on models and methods for software management and engineering*, 1980.

[6] J. P. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, M. A. Adler, "Software complexity measurement," *Communications of the ACM*, vol. 29, no. 11, pp. 1044–1050, 1986.

[7] E. E. Ogheneovo *et al.*, "On the relationship between software complexity and maintenance costs," *Journal of Computer and Communications*, vol. 2, no. 14, p. 1, 2014.

[8] T. Mens, "Research trends in structural software complexity," *arXiv preprint arXiv:1608.01533*, 2016.

[9] Á. Sillye, Z. Porkoláb, "Language ındependent software complexity measurements,"

[10] M. K. Debbarma, N. Kar, A. Saha, "Static and dynamic software metrics complexity analysis in regression testing," in *2012 International Conference on Computer Communication and Informatics*, IEEE, 2012, pp. 1–6.

[11] A. Kıral, "Yazılımların bakım kolaylığı ölçümü için yazılım ölçütleri önerisi," M.S. thesis, Başkent Üniversitesi Fen Bilimleri Enstitüsü, 2019.

[12] S. Kaur, "Software quality," *International Journal of Computers & Technology*, vol. 3, no. 1, 2012.

[13] I. Singh, "Different software quality model," *International Journal on Recent and Innovation Trands in Computing and Communication*, vol. 1, no. 5, pp. 438–442, 2013.

[14] A. Rawashdeh, B. Matalkah, "A new software quality model for evaluating cots components," *Journal of Computer Science*, vol. 2, no. 4, pp. 373–381, 2006.

[15] K. Beck, *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[16] L. Crispin, "Is quality negotiable?" İn *STARWest 2001 conference*, 2001.

[17] P. Berander, L.-O. Damm, J. Eriksson, T. Gorschek, K. Henningsson, P. Jönsson, S. Kågström, D. Milicic, F. Mårtensson, K. Rönkkö, *et al.*, "Software quality attributes and trade-offs," *Blekinge Institute of Technology*, vol. 97, no. 98, p. 19, 2005.

[18] I. Abuhav, *ISO 9001: 2015-A complete guide to quality management systems*. CRC press, 2017.

[19] I. Iso, "Iec 9126-1: Software engineering-product quality-part 1: Quality model," *Geneva, Switzerland: International Organization for Standardization*, vol. 21, 2001.

[20] O. Lysne, "Software quality and quality management," in *The Huawei and Snowden Questions*, Springer, 2018, pp. 87–98.

[21] O. I. de Normalización, *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO, 2011.

[22] M. Sari, O. Kalipsiz, "Yazılım kalitesi ve insan faktörü arasındaki ilişkinin değerlendirilmesi.," in *UYMS*, 2014.

[23] S. Schweizerische, "Information technology-security techniques-ınformation security management systems-requirements," *ISO/IEC International Standards Organization*, 2013.

[24] A. Gillies, *Software quality: theory and management*. Lulu. com, 2011.

[25] S. Nystedt, C. Sandros, *et al.*, "Software complexity and project performance," *School of Economics and Commercial Law at the University of Gothenburg*, 1999.

[26] H. R. Bhatti, *Automatic measurement of source code complexity*, 2011.

[27] Ç. Çatal, B. Diri, "Yazılım metriklerini kullanarak düşük kaliteli/yüksek kaliteli modüllerin otomatik tespiti," *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu, İstanbul*, vol. 1, pp. 8–9, 2008.

[28] M. F. Oliveira, R. M. Redin, L. Carro, L. da Cunha Lamb, F. R. Wagner, "Software quality metrics and their impact on embedded software," in *2008 5Th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, IEEE, 2008, pp. 68–77.

[29] M. Khan, F. Ahmad, M. A. Khanum, *et al.*, "Lıterature revıew on software complexıty, software usabılıty and software delıverabılıty.," *International Journal of Advanced Research in Computer Science*, vol. 9, no. 2, 2018.

[30] E. Belachew, F. Gobena, S. Nigatu, "Analysis of software quality using software metrics," *International Journal of Computational Science & Application*, vol. 8, 2018.

[31] R. Lincke, J. Lundberg, W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 131–142.

[32] C. Jones, *Applied software metrics*, 1996.

[33] A. Madi, O. K. Zein, S. Kadry, "On the improvement of cyclomatic complexity metric," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 2, pp. 67–82, 2013.

[34] QSM, *QSM - function point languages table*, http://www.qsm.com/resources/function-point-languages-table/, [Online; accessed 08-Mart-2021], 2021.

[35] G. Regulwar, P. Deshmukh, R. Tugnayat, P. Jawandhiya, V. Gulhane, "Variations in v model for software development," *International Journal of Advanced Research in Computer Science*, vol. 1, no. 2, 2010.

[36] C. Ikerionwu, "Cyclomatıc complexıty as a software metrıc.," *International Journal of Academic Research*, vol. 2, no. 3, 2010.

[37] M. A. M. Azeem, "Cyclomatic complexity calculation methods and its limitations,"

[38] M. Ayyıldız, "Yazılım projeleri ölçüm sonuçları veri tabanının oluşturulması ve yeni yazılım projelerinin maliyet tahmininde kullanimi," 2007.

[39] N. Govil, "Applying halstead software science on different programming languages for analyzing software complexity," in *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, IEEE, 2020, pp. 939–943.

[40] F. Kabakci, *HALSTEAD$_K$ARMASIKLIGI*, https://fatihkabakci.com/article-HALSTEAD_KARMASIKLIGI/, [Online; accessed 08-Mart-2021], 2021.

[41] T. Hariprasad, G. Vidhyagaran, K. Seenu, C. Thirumalai, "Software complexity analysis using halstead metrics," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, IEEE, 2017, pp. 1109–1113.

[42] S. Chiemeke, A. Oladipupo, "Theoretical approaches in software complexity metrics," *AJST*, vol. 2, no. 2, 2001.

[43] A. Yıldızhan, A. Çetin, "Yazılımın bilişsel karmaşıklığını ölçme üzerine bir inceleme," *Çankaya University Journal of Science & Engineering*, pp. 361–365, 2015.

[44] Y. Wang, "On the cognitive informatics foundations of software engineering," in *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, IEEE, 2004, pp. 22–31.

[45] J. Shao, Y. Wang, "A new measure of software complexity based on cognitive weights," *Canadian Journal of Electrical and Computer Engineering*, vol. 28, no. 2, pp. 69–74, 2003.

[46] K. Wenzel, *what is a stored procedure*, https://www.essentialsql.com/what-is-a-stored-procedure//, [Online; accessed 08-Mart-2021], 2021.

[47] C. Nagy, R. Ferenc, T. Bakota, "A true story of refactoring a large oracle pl/sql banking system," in *Industrial Track of the 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*, 2011.

# TEZDEN ÜRETİLMİŞ YAYINLAR

## Konferans Bildirisi

1. Mutlu, O. , Kalıpsız, O. (2020). "New Approaches to Improving Software Code Quality" ICENTE2020, Konya,Turkey,pp.69-75.