**ANKARA ÜNİVERSİTESİ**
**FEN BİLİMLERİ ENSTİTÜSÜ**

**YÜKSEK LİSANS TEZİ**

# SOLID PRENSİPLERİ İLE BAKIM İÇİN YAZILIMI YENİDEN YAPILANDIRMA YÖNTEMİ

**Osman TURAN**

**BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**

**ANKARA**
**2019**

**Her hakkı saklıdır**

**TEZ ONAYI**

Osman TURAN tarafından hazırlanan **"Solid Prensipleri ile Bakım için Yazılımı Yeniden Yapılandırma Yöntemi"** adlı tez çalışması 02/08/2019 tarihinde aşağıdaki jüri tarafından oy birliği ile Ankara Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalı'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

**Danışman:** Dr. Öğr Üyesi Ömer Özgür TANRIÖVER
Ankara Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı

**Jüri Üyeleri:**

**Başkan:** Doç. Dr. Ayça Tarhan
Hacattepe Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı

**Üye** : Doç. Dr. Mehmet Serdar Güzel
Ankara Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı

**Üye** : Dr. Öğr Üyesi Ömer Özgür Tanrıöver
Ankara Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı

**Yukarıdaki sonucu onaylarım.**

**Prof. Dr. Özlem YILDIRIM**
**Enstitü Müdürü**

## ETİK

Ankara Üniversitesi Fen Bilimleri Enstitüsü tez yazım kurallarına uygun olarak hazırladığım bu tez içindeki bütün bilgilerin doğru ve tam olduğunu, bilgilerin üretilmesi aşamasında bilimsel etiğe uygun davrandığımı, yararlandığım bütün kaynakları atıf yaparak belirttiğimi beyan ederim.

02.08.2019

Osman TURAN

# ÖZET

Yüksek Lisans Tezi

SOLID PRENSİPLERİ İLE BAKIM İÇİN YAZILIMI YENİDEN YAPILANDIRMA YÖNTEMİ

Osman TURAN

Ankara Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Danışman: Dr. Öğr Üyesi Ömer Özgür TANRIÖVER

SOLID prensipleri uyumluluk, bağıntılık ve kenetlilik arasındaki dengeyi sağlayarak yazılım sistemlerindeki karmaşıklığı azaltma yoluyla modülerliği artırır. Bu kapsamda prensiplerle ilgili iki çalışma yapılmıştır. İlk çalışmada, ISO 9126(25010) bakım yapılabilirliğin her alt özelliği SOLID tasarım ilkeleri ile ilişkilendirilmiş ve yeniden yapılandırma işlemleri gerçekleştirilmiştir. Her bir aşamada kod değişiklikleri VS kod metrik aracıyla ölçülmüştür. İlk çalışmaya ilave olarak tek sorumluluk ilkesine yönelik çalışma ile yeniden yapılandırma göstergeleri ele alınırken metot ve sınıf isimlendirmelerinde kullanılan sözcükleri de anlamsal olarak birbirleri ile olan ilişkileri yönünden değerlendirmiştir. Anlamsal ilişki değerlendirmesinde WordNet anlamsal veri tabanı kullanılmıştır. Yazılım içerisinde yer alan olası yeniden yapılandırma yapılabilecek kod bölümleri öne sürdüğümüz tez ile otomatik olarak listelenerek belirlenen puanlama sistemine göre sıralanmış ve yazılım yeniden yapılandırma için yazılım geliştiricilerine sunulmuştur. Öne sürülen teorinin doğrulanabilmesi için kurumsal bir yapıda kullanılan iki büyük proje üzerinde vaka çalışması yapılmıştır. Yazılım yeniden yapılandırma çalışmalarında yazılım geliştiricilere önemli fırsatlar sunduğu, tek sorumluluk ilkesinin uygulanmasında faydalı olduğu görülmüştür.

**Ağustos 2019, 81 sayfa**
**Anahtar Kelimeler:** Nesne Yönelimli Programlama Prensipleri, SOLID, ISO/IEC 9126, Kod Metrikleri

## ABSTRACT

Master Thesis

METHOD OF RESTRUCTURING SOFTWARE FOR MAINTENANCE WITH SOLID PRINCIPLES

Osman TURAN

Ankara University
Graduate School of Natural andApplied Sciences
Department of Computer Engineering

Supervisor: Assistant Professor Dr. Ömer Özgür TANRIÖVER

SOLID principles increase modularity by reducing the complexity of software systems by providing a balance between compatibility, relevance, and connectivity. In this context, two studies were carried out on the principles. In the first study, each sub-feature of ISO 9126 (25010) maintenance ability was associated with SOLID design principles and refactoring was performed. The code changes at each stage were measured with the VS code metric tool. In addition to the first study, while considering the single responsibility principle and the restructuring indicators, the words used in the method and class naming were evaluated in terms of semantic relations with each other. In the semantic relationship evaluation, WordNet semantic database was used. The possible code sections that can be restructured within the software are listed automatically according to the proposed scoring system and presented to software developers for software restructuring. Case studies were conducted on two major projects used in an institutional structure to validate the proposed theory. It has been found that it offers significant opportunities to software developers in software restructuring efforts and is beneficial in the implementation of the sole responsibility principle.



**August 2019, 81 pages**
**Key Words:** Object Oriented Design Principles, SOLID, ISO/IEC 9126, code metrics

## TEŞEKKÜR

Çalışmamın her aşamasında yardımlarını ve fikirlerini paylaşmayı esirgemeyen ve bu projenin sonuca kavuşturulmasında büyük katkısı olan tez danışmanım Dr. Öğr Üyesi Ömer Özgür TANRIÖVER'e (Ankara Üniversitesi Bilgisayar Mühendisliği Anabilim Dalı) ve uzun süren çalışmamın her aşamasında birçok fedakârlık göstererek bana destek veren eşime en içten duygularımla teşekkür ederim.

Osman TURAN
Ankara, Ağustos 2019

# İÇİNDEKİLER

**TEZ ONAYI**
**ETİK** ... ii
**ÖZET** ... iii
**ABSTRACT** ... iv
**TEŞEKKÜR** ... v
**KISALTMALAR DİZİNİ** ... viii
**ŞEKİLLER DİZİNİ** ... ix
**ÇİZELGELER DİZİNİ** ... x
**1. GİRİŞ** ... 1
**2. KURAMSAL TEMELLER** ... 4
**2.1 Solid Tasarım İlkelerinin Tanımı** ... 5
**2.2 Kullanılan Kod Metrikleri** ... 16
**2.3 ISO/IEC 9126 Yazılım Mühendisliği Ürün Kalitesi** ... 18
**2.4 VS Kod Metriklerinin ISO / IEC 9126 Yazılım Ürün Kalitesi ile Eşleştirilmesi** ... 19
**2.5 WordNet Anlamsal Kütüphanesi** ... 22
**2.6 Yapısal Bağıntı (Cohesion)** ... 25
**2.7 Metot Uzunluğu (LOC)** ... 26
**2.8 Metot Parametre Sayısı** ... 26
**2.9 Sınıftaki Metot Çağırımlarının Sayısı** ... 27
**3. SOLID TASARIM İLKELERİNİN UYGULANMASI** ... 28
**3.1 Çalışma Kapsamı** ... 28
**3.2 SOLID Tasarım İlkelerinin Kullanılmasının Uygulama Üzerindeki Etkisi** ... 30
**3.2.1 İlgili çalışmalarla karşılaştırma** ... 40
**3.2.2 Değerlendirme** ... 41
**4. TEK SORUMLULUK İLKESİNE UYGUN OLARAK YENİDEN YAPILANDIRMA İÇİN BİR YAKLAŞIM** ... 43
**4.1 Çalışma Kapsamı** ... 43
**4.2 Yöntem** ... 44
**4.2.1 İlgili çalışmalar** ... 45
**4.2.2 Wordnet'in önerilen yaklaşımda kullanılması** ... 47

**4.2.3 Yapısal bağıntının çalışma içerisinde kullanılması ..... 50**
**4.2.4 Metod uzunluğunun çalışma içerisinde kullanılması ..... 51**
**4.2.5 Parametre sayısının çalışma içerisinde kullanılması ..... 51**
**4.2.6 Metot çağırımlarının kullanılması ..... 52**
**4.2.7 Uygulama tarafından yapılan önerileri skorlama ..... 52**
**4.2.8 Yaklaşımın değerlendirilmesi ..... 54**
**4.2.9 Vaka çalışması 1 ..... 54**
**4.2.10 Vaka çalışması 2 ..... 58**
**5. SONUÇ ..... 60**
**5.1 Bulgular ve Tartışma ..... 60**
**5.1.1 SOLID-metrik çalışması ..... 60**
**5.1.2 SRP çalışması ..... 60**
**5.1.3 Geçerliliğe yönelik tehditler ..... 62**
**5.2 Değerlendirme ..... 63**
**5.3 Öneriler ..... 63**
**KAYNAKLAR ..... 65**
**ÖZGEÇMİŞ ..... 70**

## KISALTMALAR DİZİNİ

| | |
|---|---|
| API | Application Programming Interface |
| DIP | Bağımlılık Değiştirme Prensibi (Dependency Inversion Principle) |
| GUI | Graphical User Interface |
| HTTP | Hyper Text Transfer Protocol |
| ISP | Arayüz Ayrıştırma İlkesi (Interface Segregation Principle) |
| LSP | Liskov İkame Prensibi (Liskov Substitution Principle) |
| OCP | Açık Kapalılık İlkesi (Open Closed Principle) |
| OOPS | Nesne Yönelimli Programlama Sistemi |
| PC | Personal Computer |
| SRP | Tek Sorumluluk İlkesi (Single Responsibility Principle) |

# ŞEKİLLER DİZİNİ

Şekil 2.1 SRP kod örneği ..... 5
Şekil 2.2 SRP için yapılandırılan kod ..... 6
Şekil 2.3 OCP dikdörtgen sınıfı ..... 7
Şekil 2.4 OCP alan hesaplama sınıfı ..... 7
Şekil 2.5 OCP yapılandırma sonrası kod ..... 8
Şekil 2.6 LSP ilk kod ..... 9
Şekil 2.7 LSP dosya okuma kod parçası ..... 9
Şekil 2.8 LSP yapılandırma sonrası kod ..... 10
Şekil 2.9 ISP ilk kod parçası ..... 11
Şekil 2.10 ISP görev ayrıştırma kod parçası ..... 11
Şekil 2.11 ISP yapılandırma sonrası kod parçası ..... 12
Şekil 2.12 DIP ilk kod parçası ..... 13
Şekil 2.13 DIP veri aktarma kod parçası ..... 14
Şekil 2.14 DIP yapılandırma sonrası kod parçası ..... 15
Şekil 2.15 ISO/IEC 9126 kalite modeli yapısı (www.iso.org) ..... 19
Şekil 2.16 Dış ve iç olmak üzere kalite modeli(www.iso.org) ..... 20
Şekil 2.17 WordNet örnek hiyerarşik gösterim ..... 24
Şekil 2.18 Örnek sınıf tanımlaması ..... 27
Şekil 3.1 İlk versiyon ..... 31
Şekil 3.2 SRP ile yeniden yapılandırma sonrası ..... 32
Şekil 3.3 OCP uygulanmadan önceki sınıf ..... 32
Şekil 3.4 OCP uygulandıktan sonraki diyagram ..... 33
Şekil 3.5 LSP uygulanmadan önceki sınıf ..... 34
Şekil 3.6 LSP uygulandıktan sonraki diyagram ..... 35
Şekil 3.7 ISP uygulandıktan sonraki diyagram ..... 36
Şekil 3.8 Sınıfların ilk versiyonu ..... 38
Şekil 3.9 DIP uygulandıktan sonraki versiyon ..... 39
Şekil 3.10 Projenin sınıf diyagramı ..... 40
Şekil 4.1 Tanımlayıcı isimlendirme içeren örnek bir sınıf ..... 48

# ÇİZELGELER DİZİNİ

Çizelge 2.1 Kod metrikleri ile system karakteristiklerinin eşleştirilmesi ..... 22
Çizelge 3.1 VS kod metrik sonuçları ..... 31
Çizelge 3.2 SRP sonrası kod metrik sonuçları ..... 32
Çizelge 3.3 OCP sonrası kod metrik sonuçları..... 33
Çizelge 3.4 LSP sonrası kod metrik sonuçları ..... 34
Çizelge 3.5 LSP sonrası kod metrik sonuçları ..... 36
Çizelge 3.6 VS kod metrik sonuçları ..... 38
Çizelge 4.1 Örnek Benzerlik Tablosu (*: metot değerinin aritmetik ortalaması) ..... 49

# 1. GİRİŞ

Yazılımla ilgili yayınlama sonrası bakım işleri BT departmanlarında önemli bir yere sahiptir. Zaman içinde değişmesi gerekmeyen bir yazılım sistemi düşünülemez. Kullanıcı gereksinimlerindeki değişiklikler, altyapıda meydana gelen değişiklikler nedeniyle sistemin çalışma şartlarındaki değişiklikler, öngörülemeyen hataların oluşması gibi nedenlerle yazılım bakımı gereklidir.

Literatüre göre, yazılımlarda yaklaşık olarak bakım için harcanan maliyet yüzde 40-80 arasındadır. Bu da demek oluyor ki ortalama yüzde 60'ı yazılım maliyetidir (Glass, 2001). Bu nedenle, en önemli yaşam döngüsü aşaması kabul edilebilir. Yazılımın bakım işlemlerine uygunluğu maliyetin düşürülmesinde önemli bir etkendir. Buna ek olarak bakım işlemleri aşamasında kodların uzatılması veya gereksiz yönlendirmelerin yapılması uygulamanın performansını düşürücü bir etkendir. Dolayısıyla performans ve bakım arasında bir ilişki vardır. Bazen bakım yapılan uygulamaların performansı istenmeyen şekilde düşebilir. Veya yazılım bakım işlemlerinde sadece yazılım için yapılması gereken temel düzenlemeler yapılmaz aynı zamanda kohezyon değeri de artırılıp, kod karmaşıklığı da azaltılmaya çalışılır. Burada bakım işlemlerini yapan yazılımcının yaptığı işlemler önem kazanmaktadır. Yani sadece temel nesne yönelimli programlama kavramlarını kullanmak uygulamalarımızda kolay bakım yapılabilir kod yazıldığını göstermez. Dolayısıyla, uygulamaları ve hizmetleri tasarlayan, inşa eden veya işleten herhangi bir yazılım mimarı, bilgi teknolojisi (BT) geliştiricisi veya BT uzmanı, nesne yönelimli programlama sisteminin (OOPS) nasıl uygulanacağını bilmeli ve bunları doğru şekilde kullanmalıdır. Aksi halde bakım ve bakım sonrası süreçler bir sorun haline gelebilmektedir.

Bakım işlemlerini yapan yazılımcı yeniden düzenleme işlemleri için kendisine yardımcı 3. parti diyebileceğimiz uygulamalar kullanabilir ve bu uygulamalar yazılımcının işini de bir hayli kolaylaştırır. Fakat yine de geliştiricinin kaynak kod içerisinde yeniden yapılandırma yaparken kaynak kod içerisinde taşınması, çıkarılması veya değiştirilmesi gereken yerlere kendisinin karar vermesi gerekmektedir. Otomatik yeniden düzenleme araçları genellikle, programcıların yeniden düzenleme konusunda yeterli bilgiye sahip

olduklarını varsaymaktadır (Fowler,1999). Geliştiriciler genellikle bilgi eksikliği nedeniyle veya incelenecek kod çok karmaşık olduğu ve yeterli zaman olmadığı için (Pinto vd, 2013) yeniden yapılandırma işlemlerinin tam olarak yapamamaktadırlar (Szoke, 2017).

Ancak nesneye yönelik programlama tasarım ilkeleri bu sorunların üstesinden gelebilir. Bu tasarım ilkeleri SOLID olarak da adlandırılan beş nesne yönelimli ilkedir. SOLID beş nesne yönelimli tasarım ilkelerinin (Tek sorumluluk Prensibi, Açık-Kapalı Prensibi, Liskov İkame Prensibi, Arayüz Ayrıştırma Prensibi ve Bağımlılık İlerleme Prensibi) baş harflerinin R.C. Martin (Martin, 2000) tarafından kısaltmasıdır. Bu ilkelerin birlikte uygulanması sonucunda maliyet olarak çok zaman alan yazılım bakımları daha kolay hale gelmektedir (Metz, 2009).

Bu çalışmada SOLID tasarım prensiplerinin kod metriklerine etkisi incelenmiş ve aynı zamanda bu tasarım ilkelerinden birincisi olan tek sorumluluk ilkesi detaylı olarak ele alınmıştır. Çalışmada uygulanmaya çalışılan durum çalışmasında kullanılan yazılımlar, koddaki yazılım metrikleri değişiklikleri Microsoft Visual Studio (VS) kod metrikleri aracı ile ölçülmüştür. Kod metrik aracı, kod satırı, karmaşıklık, miras alınan sınıf hiyerarşisi derinliği, bakım yapılabilirlik endeksi gibi metrikler içerir (Microsoft docs).

Bakım yapılabilirlik endeksi, kaynak kodun bakım işleminin maliyetini belirlemek için bir fikir verebilir. Ancak bu göstergeyi istenen ölçüde kullanmak kolay değildir. Çünkü bakım endeksinin hesaplanan değeri, bakımın nasıl yapılacağı ile ilgili detaylı bir fikir vermez. Bakım yapılabilirlik endeksi değerini geliştirmek için nasıl bir eylemde bulunulabileceği konusunda da ipucu vermez. Bu endeks sadece yazılım sistemlerinin sürdürülebilirliğini ortaya çıkarmak için önerilmiştir. Bu çalışmada, Visual Studio (VS) kod metrik aracı yardımıyla ISO/IEC bakım yapılabilirliğe ait her bir alt özellik değerlendirilmiştir. Değerlendirme, metriklerin her bir sürdürülebilirlik özelliği için VS kod metrik sonuçları ile ilişkilendirilmesiyle yapılmıştır. ISO/IEC bakım yapılabilirlik değerleri ve kod metrik aracı ile bir analiz yapılabilmesi için öncelikle, ISO/IEC 9126 standardının bakım yapılabilirlik bölümünün her bir alt özelliği, karakteristiklerin ölçümü için beş VS kod metriğiyle eşleştirilmiştir.

Prensiplerle ilgili olan metrik çalışmasında öne sürülen teoriyi desteklemek amacıyla yapılan durum çalışmasında, HRS olarak adlandırdığımız bir insan kaynakları yönetim sistemi projesi kullanılmıştır. HRS projesi SOLID ilkelerinin Visual Studio kod ölçümleri üzerindeki etkisinin bir değerlendirmesini içermektedir. Proje, değerlendirme yapılabilmesi için ilk önce SOLID prensipleri olmadan ve daha sonra SOLID prensipleri ile geliştirilmiştir. Varsayılan tasarımda HRS'nin kod ölçümleri VS aracılığı ile alınmıştır. SOLID ilkeleri uygulandıktan sonra tekrar kod ölçümleri alınmıştır. Sonuçlar ise SOLID uygulamasından elde edilen iyileştirmeler ve faydaların bağlamı ile karşılaştırılmıştır. Aynı zamanda, sonuçlar, genel kaliteyi belirleyen kullanılabilirlik, verimlilik, taşınabilirlik, işlevsellik ve güvenilirlik gibi faktörlerin bağlı olduğu ana etkenin sürdürülebilirlik olduğunu öne süren ISO/IEC 9126 (ISO 9126, 2001) kapsamında da değerlendirilmiştir. Bu değerlendirmenin ana odak noktası, sürdürülebilirlik ve analiz edilebilirlik, test edilebilirlik, değişkenlik ve istikrar gibi alt özelliklerdir.

Tek sorumluluk ilkesi ile ilgili çalışmada, yine bu ilkeye göre yazılımların nasıl yeniden yapılandırılabileceği ile ilgili bir yöntem sunulmuştur. Buna ek olarak öne sürülen yönteme uygun olarak yazılım geliştiricilerine yeniden yapılandırma ile ilgili öneriler sunan bir uygulama geliştirilmiş ve bu uygulamanın ve öne sürülen teorinin doğruluğu 2 farklı vaka çalışması ile de doğrulanmaya çalışılmıştır. İkinci bölümde sadece tek prensip üzerinde bir çalışma yapılmasının nedeni çalışma kapsamını daha dar tutarak ele alınan prensip üzerinde detaylı bir çalışma imkânının elde edilmesidir. Tek sorumluluk ilkesinin seçilmiş olmasının sebebi ise ilk prensip olması ve yeniden yapılandırma ve bakım işlemlerini kolaylaştıracak şekilde önemli olmasıdır.

Tezin geri kalanı şu şekilde organize edilmiştir; 2. bölümde tez çalışması içerisinde geçen kavramların bahsedildiği kuramsal temeller bölümü bulunmaktadır. Tezde çalışma yapılan iki bölümden ilki olan SOLID prensiplerinin değerlendirilmesi ile ilgili kısım 3. bölümde ve tek sorumluluk ilkesinin etkisi ile ilgili kısım ise 4. bölümde bulunmaktadır. Son 5. bölüm de ise çalışmaların değerlendirildiği, bulguların tartışıldığı ve önerilere yer verildiği kısımlara yer verilmiştir.

## 2. KURAMSAL TEMELLER

Çalışmanın nasıl gerçekleştirildiğinin anlatımına başlanmadan önce çalışma içerisinde geçen bazı kavramlardan bahsedilmesinin gerekli olduğu düşünülmektedir.

Çalışmamız iki bölüm halinde gerçekleştirilmiş ve yapılan araştırmalar ile deneysel gözlemler bu bağlamda gerçekleştirilmiştir. Çalışmamızın ilk bölümünü genel olarak nesne yönelimli programlama tasarım prensiplerinden olan SOLID ile bu prensiplerin ISO/IEC bilgi teknolojileri kalite süreçleri arasındaki ilişkinin gösterilmesi ve tasarım prensiplerinin bir yazılım süreci üzerine olan etkileri oluşturmaktadır.

Çalışmamızın ikinci bölümünde ise SOLID tasarım prensiplerinden olan tek sorumluluk ilkesi (single responsibility principle) ele alınarak yazılımın tek sorumluluk ilkesi uyarınca yeniden yapılandırma işlemlerinin nasıl yapılabileceği üzerinde durulmuş ve konuyla ilgili bir durum çalışması yapılmıştır. Ayrıca yine bu bölümde yazılım içerisinde kullanılan sınıf ve metot isimlendirmeleri arasındaki anlamsal ilişki WordNet Kütüphanesi kullanılarak değerlendirilmiştir. Çalışma içerisindeki önermeleri destekleyebilmek amacıyla da çalışma içerisinde savunulan ilkelere bağlı olarak bir uygulama geliştirilmiştir. Uygulama yazılımlar üzerinde çalışarak tek sorumluluk ilkesine bağlı olarak yeniden yapılandırma ile ilgili yazılım geliştiricilerine öneriler sunmaktadır.

Bu bağlamda SOLID tasarım ilkeleri nedir, yazılımın hangi aşamalarında kullanılır, VS kod metrikleri nelerdir, ISO/IEC 9126 standardı içeriği nedir, WordNet kütüphanesi nedir gibi sorulara yanıt verilmesi öngörülmüştür.

Tüm çalışma kapsamına bakıldığında en önemli olayın yazılan kod içerisinde Solid tasarım ilkelerinin ve WordNet kütüphanesinin kullanılması işlemi olduğu görülmektedir. Bu ilkelerin kullanılması yazılan kodun kalitesini artırıcı bir rol almaktadır. Bu nedenle ilk ele alınması gereken konunun da Solid tasarım ilkeleri ve daha sonrasında WordNet Kütüphanesi olduğu değerlendirilmektedir.

## 2.1 Solid Tasarım İlkelerinin Tanımı

Nesne yönelimli bilgisayar programlamasında, SOLID, yazılım tasarımlarını daha anlaşılır, esnek ve sürdürülebilir hale getirmeyi amaçlayan beş tasarım ilkesinin baş harfleriyle oluşturulmuş kısaltmasıdır. Her ne kadar nesne yönelimli bir tasarıma başvursalar da, SOLID ilkeleri çevik geliştirme veya uyarlanabilir yazılım geliştirme gibi metodolojiler için temel bir felsefe oluşturabilir. (Metz, 2009). SOLID ilkeleri teorisi, Robert Martin tarafından 2000 yılında yayınlanan Tasarım İlkeleri ve Tasarım Desenleri adlı makalesinde tanıtılmıştır.

**Tek Sorumluluk İlkesi** - S, bir sınıfın değişmesi için sadece bir neden olması gerektiği anlamına gelir (Martin, 2000). Bir sınıfı değiştirmek için birden fazla sebep varsa, o zaman sınıfın birden fazla sorumluluk aldığı değerlendirilebilir. Sınıf içerisindeki metotlar başka sınıflar içerisindeki metotlarla olmaması gereken derecede ilişki içerisinde bulunuyor olabilir. O zaman bu sınıfın yüksek derecede bağlı olduğu düşünülebilir. Bu duruma göre, bağlılığın hareketsiz tasarımlara yol açtığı ve herhangi bir gereksinim veya değişiklik yaparken istenmeyen bir yolla sonuçlanabileceği söylenebilir (Singh vd, 2015). Bu prensip uygulamada kullanılan sınıfların daha tasarım aşamasında kimliklendirilmesi ve her türlü değişim için hazır hale getirilmesini amaçlar. Görevlerin iyi bir şekilde ayrılması programında nasıl iyi bir şekilde çalışacağını gösterir. Bunu bir örnekle gösterecek olursak (Şekil 2.1);

```
1.  public class UserService
2.  {  public void Register(string email, string password)
3.     {
4.       // formatın kontrol edildiği kod parçası
5.     }
6.     public virtual bool ValidateEmail(string email)
7.     {
8.       return email.Contains("@");
9.     }
10.    public bool SendEmail(MailMessage message)
11.    {
12.      _smtpClient.Send(message);
13.    }
```

Şekil 2.1 SRP kod örneği

Şekil 2.1'de yazılan kod iyi gibi görünse de SRP prensibini izlememektedir. "SendEmail" ve "ValidateEmail" metotlarının "UserService" sınıfı içerisinde yerleri yoktur. Çünkü "UserService" sınıfı sadece kullanıcı işleminin yapıldığı bir sınıftır. E-posta adresinin kontrol edilmesi ve gönderilmesi başka bir sınıf içerisinde yapılmalıdır. Bu duruma göre kodu yeniden yapılandırırsak şu kodu elde etmiş oluruz (Şekil 2.2).

```
1.  public class UserService
2.  {
3.    EmailService _emailService;
4.    DbContext _dbContext;
5.    public UserService(EmailService myEmailService, DbContext myDbContext)
6.    {
7.      _emailService = myEmailService;
8.      _dbContext = myDbContext;
9.    }
10.   public void Register(string email, string password)
11.   {
12.     // formatın kontrol edildiği kod parçası
13.   }
14.
15. public class EmailService
16.   {
17.     SmtpClient _smtpClient;
18.   public EmailService(SmtpClient aSmtpClient)
19.   {
20.     _smtpClient = aSmtpClient;
21.   }
22.   public bool virtual ValidateEmail(string email)
23.   {
24.     return email.Contains("@");
25.   }
26.   public bool SendEmail(MailMessage message)
27.   {
28.     _smtpClient.Send(message);
29.   }
30. }
```

Şekil 2.2 SRP için yapılandırılan kod

**Açık Kapalılık Prensibi** - O, sınıflar; prosedürler veya fonksiyonlar gibi yazılım parçalarını gerektirir. Bu yazılım parçaları eklenti için açıkken, modifikasyon için kapalı olmalıdır (Martin, 2000). Bir yazılım bölümünün bağlamı, kaynak kodunu değiştirmeden davranışını genişletebilir veya bir sınıf, sınıf davranışı değiştirilmeden kolayca genişletilebilir. Gereksinimler değiştiğinde, zaten çalışan eski kod değiştirilerek değil, yeni kod eklenerek bu tür modüllerin davranışı genişletilebilir. Bu prensip ile ilgili anlaşılabilir bir örnek geometrik şekil ile ilgili yükseklik ve genişliği olan sınıftır. Örneğin elimizde yükseklik ve genişliği olan bir dikdörtgen sınıfı olsun (Şekil 2.3).

```
1.  public class Rectangle{
2.      public double Height {get;set;}
3.      public double Wight {get;set; }
4.  }
```

Şekil 2.3 OCP dikdörtgen sınıfı

Uygulamamız dikdörtgenin alanını hesaplasın. Bunun için öncelikle alan hesaplayan bir sınıfımız olmalıdır (Şekil 2.4).

```
1.  public class AreaCalculator {
2.      public double TotalArea(Rectangle[] arrRectangles)
3.      {
4.          double area;
5.          foreach(var objRectangle in arrRectangles)
6.          {
7.              area += objRectangle.Height * objRectangle.Width;
8.          }
9.          return area;
10.     }
11. }
```

Şekil 2.4 OCP alan hesaplama sınıfı

Şekildeki kod, alanı hesaplar gözükse de genişleme için müsait değildir. Burada biz dairenin, üçgenin alanlarını da hesaplamak istersek sınıfın içerisine devamlı "if" blokları yazarak genişletmek zorunda kalırız. Fakat bu prensibin temel amacı eklenti için açık ve modifikasyon için kapalı sınıflar sağlanmasıdır. Bunun için aşağıdaki gibi kod yazabiliriz (Şekil 2.5).

Bu şekilde uygulamamız hem SRP hem de OCP sağlamış olur. Bu şekilde alan hesaplanması için başka bir şekil gelirse, bu şekil için sınıfın alan hesaplama sınıfında değişikliğe gidilmeden "Shape" soyut sınıfından türetilmesi gerekmektedir.

```
public abstract class Shape
{
  public abstract double Area();
}

public class Rectangle: Shape
{
  public double Height {get;set;}
  public double Width {get;set;}
  public override double Area()
  {
    return Height * Width;
  }
}
public class Circle: Shape
{
  public double Radius {get;set;}
  public override double Area()
  {
    return Radius * Radius * Math.PI;
  }
}

public class AreaCalculator
{
  public double TotalArea(Shape[] arrShapes)
  {
    double area=0;
    foreach(var objShape in arrShapes)
    {
      area += objShape.Area();
    }
    return area;
  }
}
```

Şekil 2.5 OCP yapılandırma sonrası kod

**Liskov Değiştirme Prensibi** - L, bir tip eğer temel olan bir temel tipten türetilmişse, türetilmiş tipin, temel tip ikame desteğine sahip olmasını gerektirir. (Martin, 2000) Her bir alt sınıf/türetilmiş sınıf, temel/ebeveyn sınıfının yerine kullanılabilir olmalıdır. Bir modül bir sınıf kullanıyorsa, o zaman bu sınıfa yapılan referans, modülün işlevselliğini etkilemeden temel sınıftan türetilmiş bir sınıfla değiştirilebilir. Türetilmiş sınıfların yazılımı geliştirilirken, türetilmiş sınıflar, temel sınıfların işlevselliğini değiştirmeden, bu sınıfların işlevselliğini genişletebilir. Bu prensibe aykırı olan bir kod ve yeniden yapılandırılması gösterilecek olursa (C Sharp Corner, 2019) bu durum aşağıda görülebilir. Başta alınan verileri SQL dosyalarına yazan bir kod parçamız olsun (Şekil 2.6).

```
1.  public class SqlFile
2.  {
3.    public string LoadText()
4.    {
5.      /* SQL dosyasından veri okuyacak kod parçası */
6.    }
7.    public string SaveText()
8.    {
9.      /* SQL dosyasına veri yazacak kod parçası */
10.   }
11. }
12. public class SqlFileManager
13. {
14.   public string GetTextFromFiles()
15.   {
16. /* Dosyadan veri alan kod parçası */
17.  }
18.   public void SaveTextIntoFiles()
19.   {
20. /* Dosyaya veri yazan kod parçası */
21.   }
```

Şekil 2.6 LSP ilk kod

Daha sonrasında programda sadece okunabilir dosyalarında olabileceği şeklinde değişiklik yapılmak istense bu "SQLFile" sınıfından miras alacak şekilde "ReadOnlySqlFile" sınıfının oluşturulması ile mümkün olabilir. Fakat burada "SaveText" metodunun bir şekilde sadece okunabilir dosyalar için önlenmesi gerekmektedir (Şekil 2.7).

```
1.  public class SqlFile
2.  {
3.    public string LoadText()
4.    {
5.    /* Sql den veri okuyacak kod parçası */
6.    }
7.    public void SaveText()
8.    {
9.      /* SQL e veri yazacak kod parçası */
10.   }
11. }
12. public class ReadOnlySqlFile: SqlFile
13. {
14.   public string LoadText()
15.   {
16.     /* SQL den veri okuyacak kod parçası */
17.   }
18.   public void SaveText()
19.   {
20.     . /* Kaydetme işleminin yapıldığı kod parçası */
21.   }
22. }
```

Şekil 2.7 LSP dosya okuma kod parçası

Hatalardan kaçınılması için "SqlFileManager" sınıfının yeniden yapılandırılması gerekmektedir. Çünkü bu sınıf içerisinde dosya kaydı yapan kod parçalarının sadece okuma işi yapan durumlarda kullanılmaması gerekir. Mevcut durum LSP prensibine aykırı bir durumdur. Bu prensibe uygun olarak yeniden tasarım yapılacak olursa araya bir arayüz (interface) konularak "SqlFileManager" diğer kod parçalarından bağımsız hale getirilebilir (Şekil 2.8).

```
public interface IReadableSqlFile
{
  string LoadText();
}
public interface IWritableSqlFile
{
  void SaveText();
}

public class ReadOnlySqlFile: IReadableSqlFile
{
  public string LoadText()
  {
    /* SQL dosyasından veri okuyacak kod parçası */
  }
}

public class SqlFile: IWritableSqlFile,IReadableSqlFile
{
  public string LoadText()
  {
    /* SQL dosyasından veri okuyacak kod parçası  */
  }
  public void SaveText()
  {
    /* SQL dosyasına veri yazacak kod parçası  */
  }
}

public class SqlFileManager
{
  public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
  {
/* Dosyadan veri alan kod parçası */
  }
  public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
  {
  /* Dosyaya veri yazan kod parçası */
  }
}
```

Şekil 2.8 LSP yapılandırma sonrası kod

**Arabirim Ayrıştırma İlkesi** – I, İstemcilerin kullanmadıkları arayüzlere bağlı kalmamasını gerektirir (Martin, 2000). Bir yazılım parçası, kullanmadığı, ancak diğer yazılım parçalarının kullandığı arayüzleri içeren bir sınıfa bağlı olduğunda, o yazılım

parçası diğer yazılım parçalarının sınıfa uyguladığı değişikliklerden etkilenecektir. Örneğin bir firmada takım lideri ve programcılar olarak pozisyonlar olsun. Takım lideri işleri bölerek programcılara atasın (Şekil 2.9).

```
1.  public Interface ILead
2.  {
3.    void CreateSubTask();
4.    void AssginTask();
5.    void WorkOnTask();
6.  }
7.  public class TeamLead : ILead
8.  {
9.    public void AssignTask()
10.   {
11.     //Görevi atayacak kod.
12.   }
13.   public void CreateSubTask()
14.   {
15.     //Görevi oluşturacak kod
16.   }
17.   public void WorkOnTask()
18.   {
19.     //Atanan görevi yerine getirecek kod.
20.   }
21. }
```

Şekil 2.9 ISP ilk kod parçası

Daha sonrasında ise yönetici olarak bir rol daha gelsin ve yöneticide işleri takım liderlerine paylaştırsın (Şekil 2.10).

```
1.  public class Manager: ILead
2.  {
3.    public void AssignTask()
4.    {
5.      //Görevi atayacak kod.
6.    }
7.    public void CreateSubTask()
8.    {
9.      //Görevi oluşturacak kod.
10.   }
11.   public void WorkOnTask()
12.   {
13.     throw new Exception("Yönetici görev üzerinde çalışmaz");
14.   }
15. }
```

Şekil 2.10 ISP görev ayrıştırma kod parçası

Yönetici bir iş üzerinde çalışamayacağı ve yöneticiye iş ataması yapılamıyacağı için "WorkOnTask" metodunun yönetici sınıfı içerisinde olmaması gerekir. Fakat "ILead"

sınıfından miras alındığı için ilgili metot yönetici sınıfında öylesine duracak ve hiçbir iş yapmayacaktır. Bu yanlış bir durumdur ve bu prensibe de aykırıdır.

```
1.  public interface IProgrammer
2.  {
3.    void WorkOnTask();
4.  }
```

```
1.  public interface ILead
2.  {
3.    void AssignTask();
4.    void CreateSubTask();
5.  }
```

```
1.  public class Programmer: IProgrammer
2.  {
3.    public void WorkOnTask()
4.    {
5.      //Atanan görevi yerine getirecek kod.
6.    }
7.  }
8.  public class Manager: ILead
9.  {
10.   public void AssignTask()
11.   {
12.     //Görevi atayacak kod
13.   }
14.   public void CreateSubTask()
15.   {
16.   //Görevi oluşturacak kod.
17.   }
18. }
```

```
1.  public class TeamLead: IProgrammer, ILead
2.  {
3.    public void AssignTask()
4.    {
5.      //Görevi atayacak kod
6.    }
7.    public void CreateSubTask()
8.    {
9.      //Görevi oluşturacak kod.
10.   }
11.   public void WorkOnTask()
12.   {
13.     //Görevi yerine getirecek kod.
14.   }
15. }
```

Şekil 2.11 ISP yapılandırma sonrası kod parçası

Son durumda görevler ve amaçlar çoklu arabirimler ile paylaştırılmış olmaktadır (Şekil 2.11).

**Bağımlılık Değiştirme Prensibi** - D, yüksek seviye modüllerin düşük seviyeli modüllere bağlı olmamasını gerektirir. Hem yüksek seviyeli hemde düşük seviyeli modüller soyutlamalara bağlı olmalıdır. Soyutlamalar ayrıntılara dayanmamalıdır. Ancak ayrıntıların tam tersi, soyutlamalara dayanmalıdır (Martin, 2000). Yazılım parçaları, somutlaşmalara değil soyutlamalara dayanmalıdır. Bu ilke yüksek seviyeli modülün düşük seviyeli modüle bağlı olmaması gerektiğini, ancak soyutlamalara dayanması gerektiğini belirtir. Örneğin uygulamamızda dosya içerisine bir hata loglama modülümüz olsun. Şekildeki kod parçası dosya içerisine hata loglama işlemlerini gerçekleştirmektedir (Şekil 2.12).

```
1.  public class FileLogger
2.  {
3.    public void LogMessage(string aStackTrace)
4.    {
5.      //log u dosyaya yazan kod.
6.    }
7.  }
8.  public class ExceptionLogger
9.  {
10.   public void LogIntoFile(Exception aException)
11.   {
12. //log u dosyaya yazan kod
13.   }
14.   private GetUserReadableMessage(Exception ex)
15.   {
16. /* Kullanıcıya anlamlı mesaj üreten kod parçası */
17.   }
18. }
```

Şekil 2.12 DIP ilk kod parçası

Bir kod parçası da dosyalardan veri tabanına verilerin aktarma işlemini yapıyor olsun (Şekil 2.13).

Burada uygulamaya yeni bir işlem kayıt sistemi daha eklenmek istenirse "ExceptionLogger" sınıfını yeni metot ekleyerek değiştirmemiz gerekmektedir. Her bir eklemede de "ExceptionLogger" sınıfı şişman, büyük bir sınıf olacaktır. Buradaki tasarım sorunu "ExceptionLogger" sınıfının "FileLogger" ve "DbLogger" sınıfları ile direkt olarak bağlantılı olmasıdır. Burada bu sıkı bağlılığın soyutlama yardımı ile koparılması gerekmektedir (Şekil 2.14).

```
1.  public class DataExporter
2.  {
3.    public void ExportDataFromFile()
4.    {
5.      //dosyadan veritabanına çevirecek kod.
6.  }
7.  }
8.   public class DbLogger
9.   {
10.    public void LogMessage(string aMessage)
11.    {
12.      //mesajları veritabanına yazan kod.
13.    }
14.  }
15.  public class FileLogger
16.  {
17.    public void LogMessage(string aStackTrace)
18.    {
19.      //log u dosyaya yazan kod.
20.    }
21.  }
22.  public class ExceptionLogger
23.  {
24.    public void LogIntoFile(Exception aException)
25.    {
26.  /* Dosyaya log alan kod parçası */
27.    }
28.    public void LogIntoDataBase(Exception aException)
29.    {
30.  /* Veritabanına log alan kod parçası */
31.    }
32.    private string GetUserReadableMessage(Exception ex)
33.    {
34.  /* Anlamlı kullanıcı mesajı alan kod parçası */
35.    }
36.  }
37.  public class DataExporter
38.  {
39.    public void ExportDataFromFile()
40.    {
41.      /* Dosyadan veri gönderilen kod parçası */
42.    }
43.  }
```

Şekil 2.13 DIP veri aktarma kod parçası

```
1.  public interface ILogger
2.  {
3.    void LogMessage(string aString);
4.  }
5.  public class DbLogger: ILogger
6.  {
7.    public void LogMessage(string aMessage)
8.    {
9.      //Mesajı veritabanına yazan kod.
10.   }
11. }
12. public class FileLogger: ILogger
13. {
14.   public void LogMessage(string aStackTrace)
15.   {
16.     //loğu dosyaya yazan kod
17.   }
18. }
```

```
1.  public class ExceptionLogger
2.  {
3.    private ILogger _logger;
4.    public ExceptionLogger(ILogger aLogger)
5.    {
6.      this._logger = aLogger;
7.    }
8.    public void LogException(Exception aException)
9.    {
10. /* Exception log alan kod parçası */
11.   }
12.   private string GetUserReadableMessage(Exception aException)
13.   {
14. //kullanıcı okunaklı çıktı üreten kod
15.   }
16. }
17. public class DataExporter
18. {
19.   public void ExportDataFromFile()
20.   {
21.           //veriyi dosyadan veritabanına atan kod.
22.   }
23. }
```

```
1.  public class EventLogger: ILogger
2.  {
3.    public void LogMessage(string aMessage)
4.    {
5.      //mesajı olay görüntüleyiciye yazan kod.
6.    }
7.  }
```

```
1.  public class DataExporter
2.  {
3.    public void ExportDataFromFile()
4.    {
5.        //veriyi dosyadan veritabanına atan kod.
6.    }
7.  }
```

Şekil 2.14 DIP yapılandırma sonrası kod parçası

## 2.2 Kullanılan Kod Metrikleri

Kod metrikleri kod kalite ölçümlerinde kullanılan göstergelerdir. Kaliteli kod yazımı içerisinde kodun karmaşıklığını anlamak ve bakım yapılabilirliğini sürdürebilmek için kod ölçümleri önemli bir önlem olarak düşünülebilir ve geliştiricilere fikir verebilir. Kod karmaşıklığı, uygulamanın eksikliği ve sağlamlığı ile ilgilidir. Karmaşık kodun test edilmesi ve bakımı zordur. Bir geliştirici bir kod yazdığında, geliştiricinin, kodun iyi yazıldığından, anlaşılabilir ve sürdürülebilir olduğundan emin olmak için metriklerin sınır değerlerine uyması gerekir. Bu metrikler, ortaya çıkabilecek hataların ne kadarının yazılımın karmaşıklığından kaynaklanabileceği veya gelecekte sorun oluşturma ihtimalinin yüksek veya düşük olduğunun tahmin edilebilinmesine imkân sağlar. Geliştirici, hangi sınıfların, hangi yöntemlerin, hangi modüllerin elden geçirilmesi veya yenilenmesi gerektiğini anlayabilir. Çalışmanın ilk kısmında kod ölçümleri yapabilmek için MS Visual Studio kod ölçüm aracı kullanılmıştır. Microsoft Visual Studio, kullanıcıların kodlarını daha iyi anlamalarına yardımcı olmak için beş kod ölçümü kullanır (Microsoft, 2019), (Microsoft Docs, 2019). Bakım yapılabilirlik endeksi, döngüsel karmaşıklık, kalıtım derinliği, sınıf bağlaşması ve kod satırı sayısıdır.

Bakım yapılabilirlik endeksi (MI), yazılımın sürdürülebilirliğini değerlendirmeyi amaçlayan bir ölçümdür. Bakım yapılabilirlik endeksi 1992 yılında uluslararası yazılım bakımı konferansında tanıtılmıştır (Oman vd, 1992). MI değişik parametrelerle gelişerek ve birtakım büyük ölçekli yazılım sistemlerine başarıyla uygulanmıştır. MI temelde üç kod metriğine dayanır: Halstead Hacmi, Döngüsel Karmaşıklık ve Kod Satırı Sayısı. Bakım yapılabilirlik endeksinin formüle edilmiş hali şu şekildedir: (Microsoft, 2019)

Bakım Yapılabilirlik Endeksi (MI) = MAX (0, (171 - 5.2 * ln (Halstead Volume)

- 0.23 * Cyclomatic Complexity

- 16.2 * ln (Lines of Code)) * 100 / 171)

MI, diğer metrikleri birleştiren ve yalnızca bakım yapılabilirliğini gösteren bir grup metrikten oluşan bir ölçümdür. MI, ağırlıklı Halstead hacmi (HV), McCabe'nin siklomatik karmaşıklığı (CC) ve kod satırları sayısından (LOC) oluşur. MI, 0 ile 100 arasında bir endeks değeri hesaplar; bu değer, kodun bakım seviyesini gösterir. Yüksek bir değer daha iyi bakım yapılabilirlik demektir. Formülden görülebileceği gibi, siklomatik karmaşıklığın artması veya kod satırı sayısı, bakım yapılabilirlik endeksinin değerini düşürür. Van der Meulen ve M.A Revilla (Menten vd, 2007) 'in işaret ettiği gibi, LOC ve HV, LOC ve CC arasında çok güçlü bağlantılar vardır.

Siklomatik Karmaşıklık (CC), kodun yapısal karmaşıklığını ölçer. Programın akışındaki farklı kod akışlarının sayısı hesaplanarak oluşturulur. Çeşitli girişlere bağlı olarak, kodunuzda kaç farklı kontrol akışının çalışabileceğine bağlıdır. Bir program karmaşık ve iç içe geçmiş bir kontrol akışına sahipse, kod ve uygulama kapsamına tam olarak ulaşabilmek ve anlayabilmek için daha fazla test gerektirir ve bu programın bakım yapılabilirliği düşük olacaktır. Siklomatik karmaşıklık arttığında yazılımın yanlış olabilecek davranışlar sergileyebileceğinden söz edilebilir.

Kalıtım Derinliği (DIT), sınıf hiyerarşisinin kökenine uzanan sınıf tanımlarının sayısını gösterir. Hiyerarşi ne kadar derin olursa, belirli yöntemlerin ve alanların tanımlandığı veya yeniden tanımlandığı yeri anlamak o kadar zor olabilir. Bir kalıtım hiyerarşisi daha fazla tür içeriyorsa, kodun bakım yapılabilirliği düşük olacak dolayısıyla uygulamanın bakımı daha zor olacaktır. Diğer taraftan, yüksek bir kalıtım derinliği de, kodun yüksek oranda kullanıldığını gösterebilir. Bu halde kod kalıtım derinliğinin iyi bir yazılım için ne kadar olması gerektiği tam olarak cevaplanması zor bir sorudur. Fakat MS Visual Studio, bir devralma hiyerarşisinin dört seviyeden daha derin olduğunda uyarılar oluşturan bir kod analizi kuralı içerir.

Sınıf bağlılığı, benzersiz sınıfların parametreler, değişkenler, dönüş tipleri, yöntem çağrıları, temel sınıflar, arayüz uygulamaları ve nitelikleri gibi özelliklerle eşleşmesini ölçer. Yüksek derecede bağıntılık veya uyum içinde olma ve gevşek bir şekilde bağımlılık, her zaman iyi yazılım tasarımlarından biridir. Bağımlılığı yüksek olan bir yazılımda yazılım parçalarının yeniden kullanımı ve yazılım bakımı zordur. Eğer başka

bir sınıfa referans göstermeyen bir sınıfa sahipsek, o zaman bu sınıfın bağımlılığı sıfırdır diyebiliriz. Bunun tersi olarakta eğer bir sınıfta birçok sınıfa referans varsa o zaman yüksek bağımlılıktan söz edilebilir.

Kod satırı sayısı, koddaki yaklaşık satır sayısını gösterir. Sayı, temelde orta seviyedeki yazılım dili koduna dayanır Bu nedenle yazılımcının yazdığı kaynak kod dosyasındaki satırların tam sayısı değildir. Çok yüksek bir yazılım kod sayısı, bir tür veya yöntemin çok fazla çalıştığını ve ayrılması gerektiğini gösterebilir. Ayrıca türün veya yöntemin bakım yapılabilirliğinin zor olabileceğini de gösterebilir.

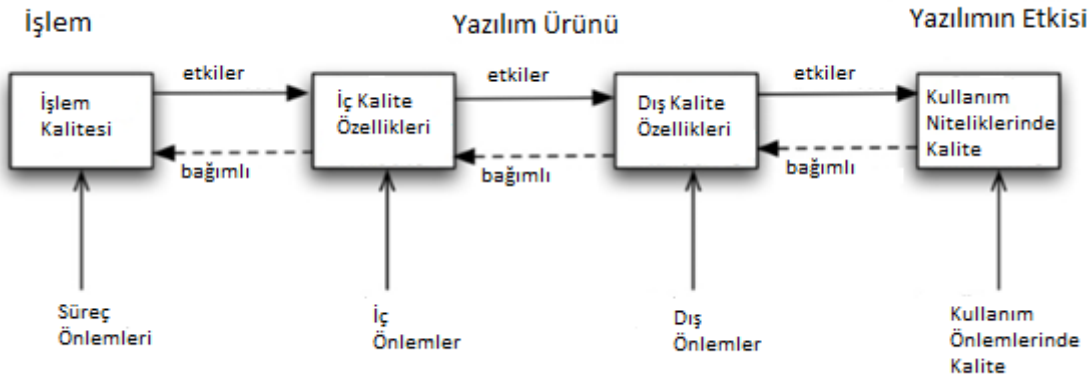
## 2.3 ISO/IEC 9126 Yazılım Mühendisliği Ürün Kalitesi

Yazılım ürün kalitesinin kapsamlı bir şekilde belirlenmesi ve değerlendirilmesi, yeterli kalitenin sağlanmasında kilit bir faktördür. Bu, yazılım ürününün kullanım amacını göz önünde bulundurarak uygun kalite özellikleri tanımlayarak başarılabilir. Doğrulanmış veya yaygın olarak kabul edilen ölçütleri kullanarak mümkün olan her yazılım ürün kalitesi özelliğinin belirlenip değerlendirilmesi önemlidir.

ISO (Uluslararası Standardizasyon Örgütü) ve IEC (Uluslararası Elektroteknik Komisyonu) dünya çapında standardizasyon için özel bir sistem oluşturur.

ISO/IEC 9126, yazılım mühendisliği ürünü kalitesi genel başlığı altında aşağıdaki bölümlerden oluşur. (Şekil 2.15)

1. Bölüm: Kalite modeli
2. Bölüm: Dış metrikler
3. Bölüm: İç metrikler
4. Bölüm: Kullanılan metriklerdeki kalite




Şekil 2.15 ISO/IEC 9126 kalite modeli yapısı (www.iso.org)

ISO/IEC 9126 yazılım ürünü değerlendirmesi bir yazılım ürünü değerlendirme süreci modelini tanımlar.

Kalite özellikleri ve ilgili ölçümler yalnızca bir yazılım ürününü değerlendirmek için değil, aynı zamanda kalite gereksinimlerini ve diğer kullanımları tanımlamak için de faydalı olabileceğinden, ISO/IEC 9126 iki ilgili çok parçalı standart ile değiştirilmiştir: ISO/IEC 9126 (Yazılım ürün kalitesi) ve ISO/IEC 14598 (Yazılım ürün değerlendirmesi). ISO/IEC 9126'nın içerisinde tanımlanan yazılım ürün kalitesi özellikleri, hem işlevsel hem de işlevsel olmayan müşteri ve kullanıcı gereksinimlerini belirtmek için kullanılabilmektedir.

## 2.4 VS Kod Metriklerinin ISO / IEC 9126 Yazılım Ürün Kalitesi ile Eşleştirilmesi

ISO / IEC 9126 ile 6 karakteristik ve 27 alt karakteristikten oluşan yazılım kalite modeli tanımlanmıştır. (Şekil 2.16). ISO/IEC 9126 ayrıca, alt özelliklerinin her birini ölçmek için bir veya daha fazla metrik tanımlar (Jung vd, 2004). Örneğin, bir yazılım ürününün sürdürülebilirliğinin kalite seviyesi, alt özelliklerinin ölçülen değerleri ile gösterilebilir. ISO/IEC 9126 standardı kalite modeli, iç ölçümler, dış ölçümler ve kullanım ölçümleri olarak dört bölüme ayrılmıştır. İlk üç bölüm, yazılım ürünlerinin kalitesini tanımlamak ve ölçmekle ilgilenmektedir. Dördüncü bölüm yazılım ürününü kullanıcı açısından değerlendirir. İç kalitenin, kullanımdaki dış kaliteyi etkilediğine inanılmaktadır.




Şekil 2.16 Dış ve iç olmak üzere kalite modeli(www.iso.org)

İç kalite dört özelliğe (işlevsellik, verimlilik, bakım kolaylığı, taşınabilirlik) ve bunların alt özelliklerine göre değerlendirilir. Bunlar bir dizi metrik kullanılarak değerlendirilmektedir. Örneğin, sürdürülebilirlik için kalite seviyesi, dört alt özelliğin ölçülen değerlerini dikkate alır. Kalite özellikleri soyut kavramlardır ve bu nedenle doğrudan ölçülebilir ve gözlemlenebilir değildir. Her biri bir dizi alt özellik ile karakterize edilir.

Çalışmamızda, ISO/IEC'nin bakım yapılabilirlik özelliğine odaklanılmıştır. Bakım yapılabilirlik özelliği aşağıdaki alt karakteristikliklere sahiptir:

- Analiz edilebilirlik: Yazılım ürününün, yazılımdaki eksiklikler veya arıza nedenleri veya tanımlanacak parçaların teşhis edilme derecesidir.
- Değiştirilebilirlik: Yazılımda yapılması gerekli olan bir değişikliğin yapılabilmesi ve ürüne uygulanabilmesi derecesini ifade eder.
- Kararlılık: Yazılım ürününün, herhangi bir yazılım değişikliğinde oluşabilecek beklenmeyen etkilerden kaçınabileceği derecedir.
- Test edilebilirlik: Yazılım ürününün, değiştirilmiş yazılımın doğrulanmasına izin verdiği derecedir.

Ancak, ISO/IEC'nin yeni sürümünde alt özelliklere modülerlik ve yeniden kullanılabilirlik eklenmiştir. (ISO:25010, 2017).

- Modülerlik: Bir sistem veya bilgisayar programının, bir bileşende yapılan değişikliğin diğer varlıklar üzerinde en az etkiye sahip olacak şekilde ayrı bileşenlerden oluştuğu derecedir.
- Yeniden Kullanılabilirlik: Yazılımın bir bölümünün birden fazla yazılım sisteminde kullanılabileceği derecedir.

ISO/IEC 9126, ölçüm girişlerinin hepsinin bir arada mı, yoksa ayrı olarak mi kullanılması gerektiği konusunda net değildir. ISO/IEC 9126; önlemlerin nasıl alınacağı, ağırlıkların nasıl hesaplanacağı veya bunların nasıl basitçe sıralanacağını göstermek için hiçbir rehberlik, kurallar dizisi veya başka bir yöntem sunmaz (Kilidar vd, 2005).

Çalışmamızın temel amacı, MS VS standart ortamıyla birleştirilmiş yazılımın bakım yapılabilirliğini değerlendirmek olduğundan, öncelikle ISO/IEC 9126 standardının bakım yapılabilirlik alt bölümünün her bir alt özelliği, karakteristiklerin ölçümü için beş VS kod metriğine eşleştirilmiştir. Bir sistemin değiştirilebilirlik özelliği, kaynak kodun karmaşıklığı gibi özelliklerle bağlantılıdır. Kaynak kodu karmaşıklığı, siklomatik karmaşıklık açısından ölçülür. Bir sistemin analiz edilebilirlik özelliği, kod satırları sayısından (LOC) ve karmaşıklık özelliklerinden etkilenir. Bir sistemin test edilebilirlik özelliği, karmaşıklık ve LOC özelliklerinden etkilenir. Kararlılık, bağımlılıktan etkilenir. Daha büyük bir sistem, genel olarak bakım yapılabilirlik için daha büyük bir çaba gerektirir. Daha yüksek boyut daha düşük analiz edilebilirliğe neden olur ve bu tip yazılım sistemlerini anlamak zordur. Kaynak koddaki içsel bozukluğun derecesi, kaynak kodun karmaşıklığı ile ifade edilebilir. Büyük kod birimleri karmaşıktır. Ayrıca, karmaşık yazılım parçalarının analiz edilebilmesi ve test edilebilmesi de zordur. Kaynak kodda çoğaltma varsa yani kod parçaları tekrar tekrar kullanılıyor ise o zaman yazılımın bakım yapılabilirlik seviyesi düşüktür ve bakımı zordur. Aşırı çoğaltma, bir sistemin olması gerekenden daha büyük olması demektir. Ek olarak bu durum, analiz edilebilirliği ve değiştirilebilirliği etkiler. VS kod metrikleri ve ISO bakım yapılabilirlik sistem özelliklerinin eşlenmesi Çizelge 2.1'de gösterilmiştir (Asadi vd, 2016).

Çizelge 2.1 Kod metrikleri ile system karakteristiklerinin eşleştirilmesi

| Bakım Yapılabilirlik Özellikleri | Kod Metrik Değerleri |
|---|---|
| Analiz Edilebilirlik | 1. Kod Satır Sayısı (LOC)<br>2. Siklomatik Karmaşıklık (CC)<br>3. Metot Sayısı&Sınıftaki Metot Ağırlığı (WMC) |
| Değiştirilebilirlik | 1. LOC<br>2. CC<br>3. Miras Alınabilirlik Derinliği (DIT) |
| Kararlılık | 1. Kenetlilik |
| Test Edilebilirlik | 1. LOC<br>2. CC |
| Modülerlik | 1. Kenetlilik<br>2. DIT |
| Yeniden Kullanılabilirlik | 1. Kenetlilik<br>2. WMC |

## 2.5 WordNet Anlamsal Kütüphanesi

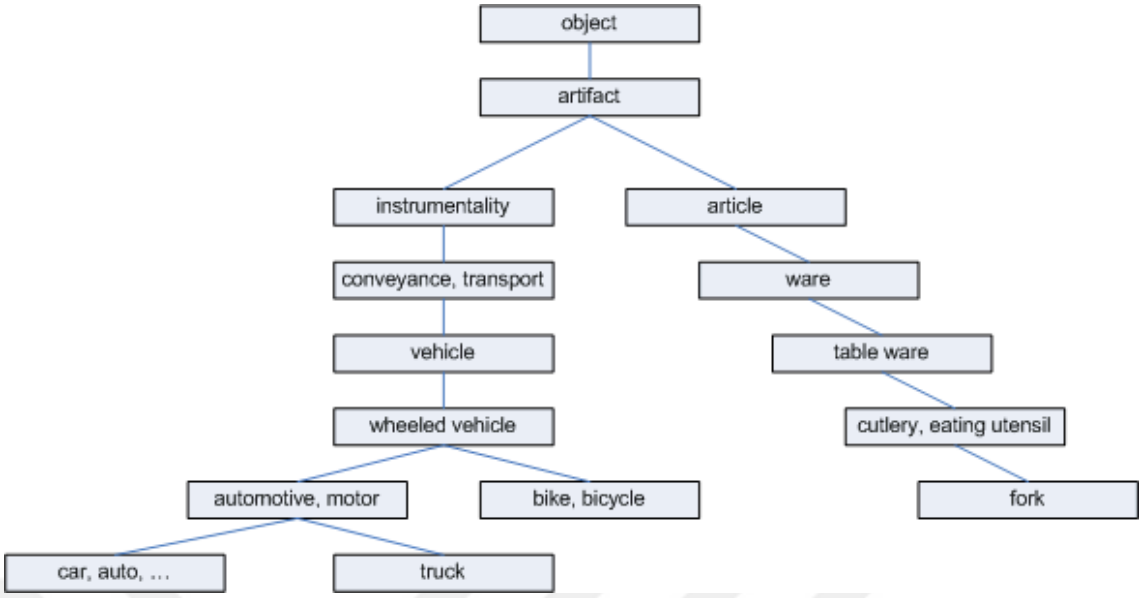
İsimlendirme yalnızca kaynak programların ve diğer tür yazılım dokümanlarının anlaşılabilirliği için önemli değildir (Laitinen, 1995), aynı zamanda geliştiricinin problemi veya çözümü için belirli bir kavramsal modeli temsil eder. Geliştiriciler, büyük ölçekli ve birbirlerinden ayrılmış projeler üzerinde çalışıyor olabilir. Bu durumda yazılım parçalarının tekrar tekrar yazılması yerine yeniden kullanılabilir yazılım bileşenlerinin kullanılması iyi düşünülmüş yazılım tasarımları için olmazsa olmazlardandır. Yazılım içerisindeki sınıf ve metotların gösteriminin tutarlılığının ötesinde, yazılım birimlerindeki adlandırma kolayca anlaşılmalı ve her eleman bileşeninin işlevini kolayca anlaşılabilir şekilde yazılım geliştiricilerine iletmelidir (www.microsoft.com). Bu nedenle, iyi tasarlanmış ve o şekilde kodlanmış olan yazılım bileşenlerinin farklı öğelerinin adlarının birbirleriyle anlamsal bir ilişki içerisinde olması gerekmektedir. Yapılan bu çalışmada, önerilen yöntemlerin deneysel olarak kanıtlanması aşamasında da yazılım bileşenlerindeki isimlendirmelerin birbirleri ile uyumlu bir şekilde yapıldığı kabul edilmiştir. Önerilen yöntemlerden bir tanesi isimlendirmelerin doğru bir anlamsal ilişki içerisinde olup olmadığının kontrol edilmesi olduğundan, pratik yaklaşım olarak, ilgili yazılım bileşenlerindeki isimler arasındaki

anlamsal mesafe kullanılmıştır. Bu nedenle, isimler arasındaki anlamsal ilişkiyi değerlendirebilmek için WordNet sözcüksel veritabanı kullanılmıştır.

WordNet, büyük bir İngilizce sözcüksel veritabanıdır. İsimler, fiiller, sıfatlar ve zarflar, her biri ayrı bir kavramı ifade eden bilişsel eş anlamlı kümeler (synsets) halinde gruplandırılmıştır. Eş anlamlı kelimeler dizisi, kavramsal-anlamsal ve sözlüksel ilişkiler ile birbirine bağlanır. (https://wordnet.princeton.edu/). İki kelime veya cümle WordNet'te karşılaştırılabilir. Kelimeler dizisi arasında en sık kodlanan ilişki süper alt ilişkidir. Bu alt ilişki {Mobilya, mobilya parçası} gibi daha genel kelime dizilerini, {yatak} ve {ranza} gibi daha özellikli olanlara bağlar. Bu nedenle, WordNet, mobilya kategorisinde yataklar ve ranza içeren yataklar içerdiğini; tersine olarak da, yatak ve ranza gibi kavramların mobilya kategorisi içerisinde bulunduğunu bilir. Tüm isim hiyerarşileri sonuçta varlık (entity) kök düğümüne gider. Alt sözcükler ilişkisi geçişlidir. Yani bir koltuk bir sandalye ise ve bir sandalye bir mobilya ise, bir koltuk bir mobilya türüdür.

İki kelime seti arasındaki anlamsal benzerliği ölçmenin basit bir yolu, kelimelerin en üstteki kök kelimeye kadar olan taksonomiyi bir grafik olarak ele almak ve iki kelime arasındaki mesafeyi WordNet'te ölçmektir. P. Resnik'e göre, bir düğümden diğerine giden yol ne kadar kısaysa o kadar benzerdir. Yol uzunluğunun, bağlantılar/kenarlar yerine düğümler/köşeler halinde ölçüldüğü unutulmamalıdır. Eş anlamlı iki kelime seti arasındaki yolun uzunluğu 1'dir.

Aşağıdaki şekil (Şekil 2.17) WordNet'te benzer kelime taksonomileri arasında kullanılan benzer yol uzunluğu ölçümünü göstermektedir.




Şekil 2.17 WordNet örnek hiyerarşik gösterim

Şekil 2.17'de, "car" ve "auto" arasındaki uzaklık 1, "car" ve "truck" arasındaki uzaklık 3, "car" ve "bicycle" arasındaki uzaklık 4, "car" ve "fork" arasındaki uzaklık ise 12' dir.

Yukarıdaki örneklerde olduğu gibi, grafiklerdeki yol uzunluğu bize iki kelime seti arasındaki ilişki mesafesini hesaplamak için basit ama kullanışlı bir yol sağlar. WordNet'te çoklu mirasa izin verilmektedir. Yani bazı kelime setleri birden fazla taksonomiye ait olabilir. Dolayısıyla, iki set arasında birden fazla yol varsa, en kısa yol seçilir.

Bu duruma göre, kod birimindeki iki isim arasındaki benzerlik şöyle ifade edilebilir:

$$Benzerlik(s,t)=1/mesafe(s,t)$$

Düğüm sayımı kullanarak s'den t'ye olan mesafe en kısa yol uzunluğudur. Kavramsal benzerlik için kullandığımız ölçüt, isimler arasındaki anlamsal mesafedir. Formülde görülebileceği gibi, benzerlik puanımız 0 ile 1 arasında olabilir. Eş anlamlı kelimeler için puan 1'dir ve kelimeler birbirinden uzaklaştığında bu değer sıfıra yaklaşır.

## 2.6 Yapısal Bağıntı (Cohesion)

Uyum yazılımla ilgili önemli bir kalite özelliğidir ve iyi yapılandırılmış yazılımın özelliklerinden biri de yüksek uyumdur. Yüksek uyumu olan yazılımlar genellikle güvenilir, tekrar kullanılabilir, sağlam ve anlaşılabilirdir. Öte yandan, düşük uyumu olan yazılımlar anlamak, test etmek, sürdürülebilirliğini sağlamak ve yazılım bileşenlerini yeniden kullanmak gibi işlevlerin zor olduğu istenmeyen özellikler gösterebilir. Yazılım içerisinde, uyum türleri arasında, yapısal veya işlevsel bütünlüğün yazılım için güçlü bir ilişki sağladığı düşünülmektedir, çünkü tüm unsurlar SRP'ye uygun olarak tek bir göreve katkıda bulunur (Izadkhah vd, 2017). Yazılım metotları içerisindeki uyum eksikliği gibi fonksiyonel uyum ölçümlerinin SRP'nin bir göstergesi olarak görülebileceği düşünülmüştür. Düşük kenetlilik ve yüksek bağıntıya (low Coupling and high cohesion) sahip tasarımlar aynı zamanda daha sürdürülebilir kod bileşenlerine de yol açmaktadır (Litz vd, 1995)

Sınıf bütünlüğünü değerlendirme ölçütleri, bir sınıfın anlaşılabilirlik seviyesini belirlemek için bir sınıfın bütünlüğünü değerlendirir; yüksek uyumu olan bir sınıf, düşük uyumu olan bir sınıftan daha anlaşılırdır. Bu nedenle, uyumu ölçmek için, bir araştırma makalesinde (Izadkhah vd, 2017) yapısal uyumla ilgili sağlanan ve farklı olarak kaynak kodundan çıkarılan yapısal bilgiler üzerinde ölçülen (örneğin, yöntem ve yöntem çağrıları gibi öznitelik referansları) farklı uyum ölçümleri araştırılmıştır. Sınıf uyumu için farklı ölçümlerin bulunmaktadır ve birini diğeriyle karşılaştırmak için normalizasyon işlemi gereklidir. Bu sorun nedeniyle, sınıf uyum ölçütlerini değerlendirmek için belirlenmiş olan standart bir ölçüt yoktur.

Bağıntıyı gösterebilmek için LCOM3 ve LCOM4 değerleri kullanılmaktadır. Bunlardan özellikle LCOM4 göstergesi bir sınıf içerisindeki birbirine bağımlı komponentleri göstermektedir. (Hitz & Montazeri, 1995). Birbirine bağımlı komponentler, ilişkili metotlarla oluşturduğu bağıntıdır. Buna göre metot içerisinde bu şekilde sadece bir tane komponent olmalıdır. Aksi halde bu o sınıfın veya metodun bölünebileceği anlamına gelir.

LCOM3, nesne yönelimli görsel programlama dillerinde özellikle, değişkenlerin değerlerine tutarlı bir şekilde erişmek için kullanılan özellik erişimcilerinin ve işlem çağrılarının etkisini dikkate almadığından LCOM4 ün ölçüm değeri daha tutarlıdır. LCOM3 daha çok C++ gibi diğer nesne yönelimli dillere daha uygun olabilirler. Yüksek LCOM4 ölçümü bağıntılı olmayan sınıfın göstergesidir. LCOM4 için en iyi ölçüm değeri 1 dir. LCOM için en büyük gösterge ortak kullanılan değişkenlerdir. LCOM4 ise aynı zamanda ortak kullanılan değişken olmayıp başka bir metot yolu ile ortak değişkene ulaşan metotları da bağıntılı olarak kabul etmektedir.

## 2.7 Metot Uzunluğu (LOC)

Kod satırları sayısı (Line of Code - LOC), programın kaynak kodunun satır sayısını sayarak bir programın boyutunu ölçmek için kullanılan bir yazılım ölçüsüdür. LOC tipik olarak, bir program geliştirmek için gerekli olan çabayı tahmin etmenin yanı sıra, yazılım üretildikten sonra programlama verimliliğini veya sürdürülebilirliğini tahmin etmek için kullanılır. Yapılan çalışmalar performansın LOC ile yüksek oranda ilişkili olduğunu doğrulasa da, işlevsellik LOC ile daha az ilişkilidir. Bu nedenle, büyük sınıfları LOC'a göre yeniden düzenlemek için belirlenmiş katı bir kural yoktur. Yine de, bir yöntemin satır sayısının 30-40'tan büyük olma eğiliminde olduğu zaman, yönteminin bu konuda çok fazla sorumluluğu olabileceği dolayısıyla SRP'yi ihlal edebileceği kabul edilmiştir (Martin Fowler, https://refactoring.com/). Bu şekilde uzun yazılmış olan bir metot kendisi içerisinde veya diğer metotlarla tamamen uyumlu ve net olabileceği gibi, birbiri ardında meydana gelen ilgisiz işlevselliği de gösterebilir.

## 2.8 Metot Parametre Sayısı

R.C. Martin yazdığı "Clean Code" kitabında, bir metot için ideal parametre sayısının sıfır olduğunu iddia etmiştir. Sonra bir tane gelir, ardından iki olarak düşünülebilir. Parametre sayısının üçten fazla olması çok özel bir gerekçelendirmeyi gerektirir ve bir yazılım prensibi olarak kullanılmamalıdır. Fakat genel olarak, yazılım geliştiricileri ve bu konuyla ilgili yazılmış olan akademik makaleler, 3 veya 4 parametre sayısı olacak bir sınırın iyi bir kural olduğuna karar verme eğilimindedir (Martin, 2008) (Martin, 2003).

## 2.9 Sınıftaki Metot Çağırımlarının Sayısı

Tek Sorumluluk İlkesine (SRP) göre bir metot sadece tek bir işlemi yapmaktan sorumludur. Eğer metot içerisindeki kod blokları birden fazla metot çağırımı yapıyorsa veya sınıf mirasından oluşturulan sınıf objeleri kullanıyor ise bu durum SRP probleminin olduğuna dair bir gösterge olabilir. Örnek olarak aşağıda gösterildiği şekilde (Şekil 2.18) bir sınıfa ait tüm metotlar başka bir sınıftan çağırılabilir. Eğer çağırıcı metot en az iki farklı sınıf objesi referans ediyor veya metot çağırıyor ise bu durum SRP ihlali için bir gösterge olabilir.

```
public class Used_Class
{
   public static void Method1()
   {
      // code here
   }
   public void Method2()
   {
      // code here
   }}
```

```
class User_Class
{
   public static void Main(string[] args)
   {
      Usee_Class.Method1();
      var instance = new User_Class ();
      instance.Method2();
   }
}
```

Şekil 2.18 Örnek sınıf tanımlaması

Eğer sınıf "static" olarak tanımlanmamışsa, bu sınıfı kullanacak kod parçacıkları bu sınıftan alınan miras yöntemi ile bu sınıfın bir objesini oluşturarak bu sınıf içeriğini kullanabilirler.

# 3. SOLID TASARIM İLKELERİNİN UYGULANMASI

## 3.1 Çalışma Kapsamı

Çalışmanın ilk aşamasında SOLID tasarım ilkelerinin yazılıma etkisinin detaylı olarak incelenebilmesi için yeniden yapılandırma yapılacak iki kurumsal yazılım ele alınmıştır. Bu yazılımların tamamında olmamak üzere belirli parçalarında tasarım ilkelerinin her biri için geliştirmeler ve güncellemeler yapılmış ve yeniden yapılandırma yapılan her bir metot ve sınıfın önceki ve sonraki halleri Visual Studio kod metrik aracı yardımı ile analiz edilerek karşılaştırılmıştır.

SOLID tasarım ilkelerinin her biri ayrı ayrı nesneye dayalı tasarım ilkeleri olarak kalitenin yazılıma etkisi, nesne yönelimli programlamada kurallar ve teknikler, bakım maliyetine katkı gibi başlıklarla incelenmesine rağmen, tüm SOLID prensiplerini içeren ve bu prensiplerin yazılım üzerine etkisi ile kod metrikleri arasında bağlantı kuran çok fazla çalışma yapılmamıştır. Al-Ahmad makalesinde (Al Ahmad, 2006), kavramsal modelleme için bir çerçeve oluşturmaya çalışarak kalıtımın bu çerçeve modeline etkisi üzerine odaklanır. Liskov İkame Prensibinin, ayrı tip, arayüzler ve sınıf hiyerarşilerinin kavramsal modelleme üzerindeki etkisi hakkında değerlendirme yapar. Liskov İkame Prensibi üzerine (Bruntink vd., 2006), (Lievens vd, 2013), (Arevalo vd, 2010) gibi bazı makalelerde çalışmalar vardır. Zotos (Zotos, 2006), matematik ve bilgisayar gibi iki farklı fakat birbiriyle yakın konular arasında yazılım problemini çözmek için nesneye yönelik tasarım ilkelerini (OODP) ortaya koymaktadır. Bizim çalışmamızda ele aldığımız tasarım ilkeleri onun tarafından da kullanılmıştır. Bu ilkeler, tasarım aşamasında tasarımın doğru yönünü gösteren kritik hatalardan kaçınmaya yardımcı olur. Bir programlama dilinin arkasındaki ilkeleri ve yöntemleri anlayarak kaliteli kod yazılabilir.

Deligiannis, Shepperd, Roumeliotis ve Stamelos, nesne yönelimli programlama için bakım yapılabilirlik hakkında bir çalışma yaparak (Deligiannis vd, 2001) iki hedefe ulaşmak istediler. İlk amaç, bir tasarım yönteminin nesne yönelimli tasarımların bakım yapılabilirliği üzerindeki etkisini incelemektir. İkincisi, nesne yönelimli tasarım sezgisi

ve program ölçütleri arasındaki ilişkiyi incelemektir. İyi düşünülmüş bir tasarım, önceki tekrarlama çevrimlerinin sonuçlarını tekrar uygulamadan yeni işlevselliklerin yeni sınıflar ve yeni yöntemler gibi kolayca bütünleştirilmesine izin verir. Diğer makalede (Bavota vd, 2011) Bavota, Oliveto ve De Lucia, sınıfı bölme yolu ile yeniden düzenleme olanaklarının belirlenmesi konusunda çalışmaktadır. Bunu yaparken yapısal ve anlamsal uyum gibi metrik ölçümleri kullanmışlardır. Metotlar arasındaki yapısal ve anlamsal ilişkilerden faydalanan, sınıfı düzenlemek için grafik teorisini temel alan bir yeniden düzenleme yöntemi önermektedirler. Çalışmalarında, yazılım geliştirirken sistem sürekli değişiyorsa ve bazı değişiklikler sürekli yapılıyorsa, kaynak kodun daha karmaşık olabileceğini ve orijinal tasarımdan uzaklaşarak tutarsızlık gösterebileceğini özetliyorlar. Bir programcının yazılımın teslim tarihi olan tarihe uyması ve bazı değişiklikler yapması veya yeni işlevler eklemesi gerekiyorsa, yazılım üzerinde yapılacak olan değişiklik, eklemelerde genellikle orijinal tasarım veya nesne yönelimli programlama (OOP) kurallarına uygun olarak hareket edilmez. Sınıflara, kod parçalarına eklenen sorumluluk arttığında yani aynı anda birden fazla fonksiyonu var ise sınıf veya fonksiyon çok karmaşık hale gelir ve kalitesi kötüleşir. Diğer bir çalışma ise (Chatzigeorgion vd, 2008), öğrencilerin tasarım modellerini anlama, uygulama ve nesneye yönelik tasarım ilkeleri kullanma yetenekleri üzerine deneysel bir araştırma yapmaya çalışmaktadır.

Makale (Zafeiris vd, 2016) yazılımlarda yeniden düzenleme yapılabilmesi için bir algoritma çalışması yaparak bağımlılığı ters çevirme prensibini değerlendirir. Bunu yaparken de sınıfların alt sınıflara düzenli bir şekilde ayrılabilmesi ve anlaşılabilir olabilmeleri için Liskov İkame Prensibi ile tasarım gereksinimlerini kullanır. Hasta bilgilerinin mobil ortamda izleyebilmek için bir yazılım çerçevesi geliştirilmesi (Al-Bashayreh vd, 2013) 'de tartışılmaktadır. Genel olarakta yeniden kullanılabilirlik ve genişletilebilirlik üzerine odaklanılmıştır. Makale (Dallal, 2012), büyük sınıfların alt sınıflar olarak yeniden sınıflandırma işlemini tahmin etmeyi amaçlayan modeller yapmak için nesne yönelimli kalite ölçümlerini kullanır. Çalışma kodun sürdürülebilirliği, anlaşılabilirliğinin yeniden düzenleme ile yapılabileceğini öne sürmektedir. Çalışmada bu nedenle, tasarım ilkeleri kullanılmıştır. Makale (Fokaefs vd, 2012), nesne yönelimli sistemlerde özüt sınıfın yeniden düzenlemesini tanımlamaya ve

uygulamaya çalışmaktadır. Yalnızca bir kavramı uygulaması gereken ve yalnızca içerdiği kavram geliştiğinde değişmesi gereken bir sınıftan bahsedilmektedir.

## 3.2 SOLID Tasarım İlkelerinin Kullanılmasının Uygulama Üzerindeki Etkisi

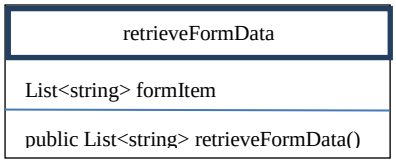
Durum çalışması için tasarım ilkeleri bir proje üzerinde uygulanmıştır. Uygulama bir insan kaynakları yönetimi programıdır. N-katmanlı mimari üzerinde çalışmaktadır. Proje içerisinde, çalışan verilerinin yönetimi, personel takibi, muhasebe ve bordro sistemi, raporlama vb. modüller bulunmaktadır. Projede yapılan değişiklikler yazılım mimarisi olarak iş katmanında ve son kullanıcı arayüz katmanında yapılmıştır. Projenin sınıf diyagramını Microsoft Visual Studio'da aldığımızda, yazılım içerisinde 48 sınıf olduğunu görüyoruz. Şekil 3.10'da gösterilmiştir.

Çalışmanın ilk aşamasında, Visual Studio (VS) kod metrik aracı yardımı ile herhangi bir değişiklik yapmadan önce tüm projenin varsayılan kod metrik değerleri alınmıştır. Projeye ait alınan bu metric değerleri Tablo 2'de gösterilmiştir. Tablo 2'de gösterilen, “Personel” tüm projeyi ifade etmektedir. “General”, “Rapor” ve “Bordro” ise yine tüm projedeki alt projeleri temsil etmektedir. “ListUpdate”, “takeFormData” ve “dataSave” yazılım içerisinde kullanılan yöntemleri belirtir. Üzerinde değişiklik yapılacak olan modüller, VS kod metrik aracı çalıştırıldığında, bakım yapılabilirlik endeks (MI) değerleri düşük olanlar arasından seçilmiştir. Değiştirme işleminin ilk aşamasında, SOLID tasarım ilkelerine göre modül içerisinde yazılmış olan sadece bir metot üzerinde değişiklik yapılmıştır. Yapılan değişiklikler SOLID tasarım ilkeleri sırası ile uygulanmıştır. Üzerinde değişiklik yapılan metot içeriği, çalışanlara ait kazanılmaya hak esas ücret hesaplaması ile ilgilidir. Metodun görevi form verilerini almak ve bu verileri tanımlanmış olan liste nesnesine atamaktır. Bu işlemler yapılırken yevmiye kayıtlarıda kontrol edilmektedir. Metot içerisinde tanımlanmış “if-blokları” bulunmaktadır. Uygulanan herbir tasarım ilkesi değişikliğinden sonra kod metrik aracı tekrar çalıştırılarak koda ait metrik değerleri yeniden hesaplanmıştır.

Çizelge 3.1 VS kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| Personnel | 72 | 1591 | 5 | 271 | 5632 |
| Personnel.General | 71 | 97 | 5 | 50 | 428 |
| Personnel.Report | 68 | 109 | 5 | 93 | 365 |
| Personnel.Payroll | 70 | 450 | 5 | 79 | 1835 |
| listUpdate | 49 | 5 | | 15 | 24 |
| takeFormData | 40 | 6 | | 21 | 46 |
| dataSave | 49 | 5 | | 15 | 24 |

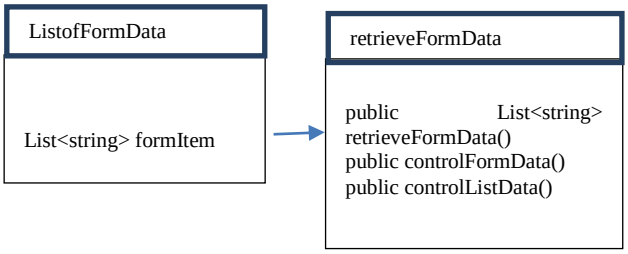
Tek Sorumluluk Prensibi (SRP): Bu prensibin bir problemi çözmek için kullandığı yöntem, etki alanında çalışan alt problemlerin bulunması ve her bir alt problemin, böyle bir problemin tek bir görevi kaldığı noktaya gelinceye kadar alt-alt problemlere bölünmesidir. Tasarım ilkelerinin tek sorumluluk ilkesi değişikliğini yapabilmek için metot içerisinde düzenlemeler yapılmıştır. Geçerli durumda, metodumuz form verilerini almakta ve bunları liste öğelerine atamaktaydı. Bu durm Şekil 3.1'de gösterilmiştir. Ek olarak, alınan verilerin doğruluğunu kontrol etme amacı ile metot içerisinde “if bloklar” vardı.




Şekil 3.1 İlk versiyon

Bu ilkeyi metot içinde uygulamak için liste öğeleri başka bir sınıf içerisinde tanımlanmıştır. Form verilerinin kontrolü veya alınan verilerin list öğelerine atanması gibi işlemler tek ve birbiri ile uyumlu olmayan bir metottan birbiri ile uyumlu ve daha küçük metotlara ayrılmıştır. Her yeni metot daha anlaşılabilir, basit olarak tasarlanmış ve her bir metodun sadece tek bir sorumluluğu vardır. SRP uygulandıktan sonraki yeni oluşan sınıflar Şekil 3.2'de gösterilmiştir. Tek sorumluluk ilkesi uygulanmasının sonunda Visual Studio kod metrik aracı yeniden çalıştırılmıştır. Buna göre bakım yapılabilirlik endeksinin yüzde 7 arttığı görülmüştür. (Tablo 3) Buna ek olarak, sınıf bağımlılık değeri azalmıştır. Öte yandan, VS kod metrik ve ISO/IEC 9126

eşleştirmemize göre de sistem özelliklerinden stabilite, modülerlik ve tekrar kullanılabilirlik artmıştır.

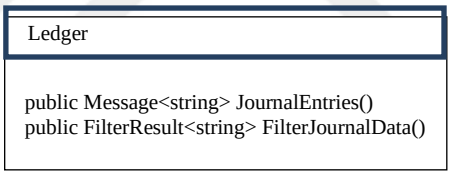



Şekil 3.2 SRP ile yeniden yapılandırma sonrası

Çizelge 3.2 SRP sonrası kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| takeFormData | 43 | 4 | | 16 | 38 |

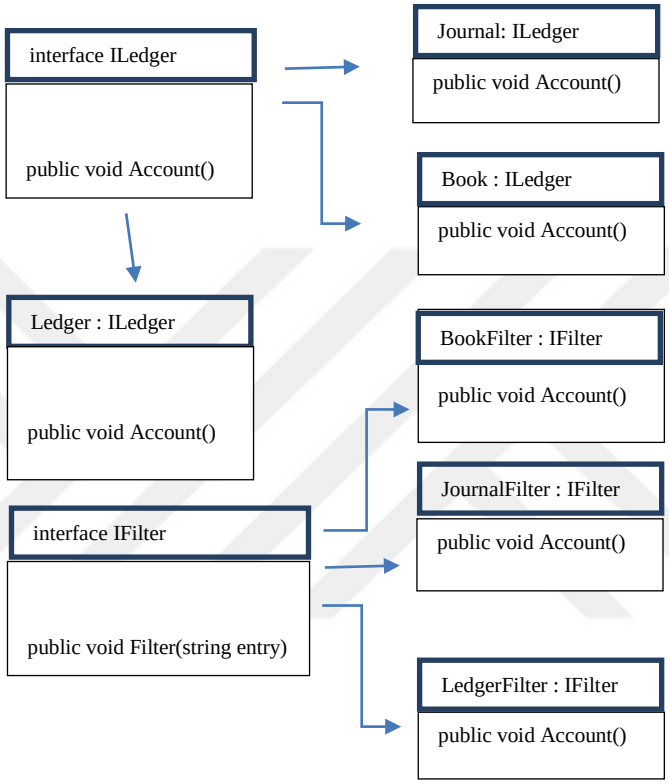
Açık Kapalılık Prensibi: Bu ilkeyi uygulamadan önce metot içeriğinde muhasebe girişlerini tespit etme ve muhasebe verisinin istenen kurallara göre filtreleme işlemleriyle ilgili kontroller bulunmaktadır.




Şekil 3.3 OCP uygulanmadan önceki sınıf

Bu prensibi uygulamak için tüm kontroller ve filtreleme işlemleri yeniden düzenlenmiştir. Bunu yapmak için, filtreleme işlemlerini yapan kod parçaları ve kontrol işlemlerini yapan kod parçaları başka bir sınıfa taşınmıştır. Değişiklikler uygulandıktan sonra, filtreleme işlemleri veya yeni kriterleri kontrol etmek için yeni oluşturulan sınıf değiştirilmek zorunda değildir. Çünkü istenen işlemlerin davranışı oluşturulan yeni sınıflara mahsustur. Ayrıca, yazılım parçalarının davranışı yeni kriterleri destekleyecek şekilde genişletilebilir. Çünkü tek yapılması gereken, yeni bir sınıfa geçilmesidir. Bu nedenle sınıf genişletilme için açıktır. Bu ilkenin uygulanması genellikle bileşik sistemlere yol açar ve genel olarak yeniden kullanım için daha fazla fırsat sağlar. Açık kapalı prensibiyle ilgili değişikliklerin gerçekleştirilmesinin sonunda, Visual Studio kod metrik aracı tekrar çalıştırılmıştır. Bakım yapılabilirlik indeksi (MI) yaklaşık yüzde 4,5

oranında arttığı görülmüştür. (Tablo 4) Siklomatik karmaşıklık değişmemiş ve sınıf bağımlılığı % 6.25 oranında azalmıştır. MI'ya ek olarak ISO 9126 eşleştirmemize göre stabilite, modülerlik, yeniden kullanılabilirlik artmıştır. Ayrıca, yeni özellikler ekleme veya mevcut olanları değiştirme kolaylığı nedeniyle analiz edilebilirlik ve değişkenlik özellikleri olumlu yönde etkilenmiştir.

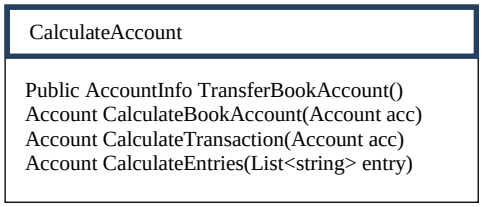



Şekil 3.4 OCP uygulandıktan sonraki diyagram

Çizelge 3.3 OCP sonrası kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| takeFormData | 45 | 4 | | 15 | 32 |

Liskov Değiştirme Prensibi: Temel sınıflara yapılan atıflar, türetilmiş sınıfların nesnelerini bilmeden kullanabilmelidir. Bir yazılımın bir temel sınıfı ve birkaç alt sınıfı varsa, kodun geri kalanı her zaman alt sınıfları değil, temel sınıfı kullanmalıdır. Bu ilke Açık Kapalılık Prensibinin bir uzantısı olarak düşünülebilir.




Şekil 3.5 LSP uygulanmadan önceki sınıf

Bu prensibin uygulanmasında başlangıçta, muhasebe ve parasal işlemler için hesap defteri ile ilgili metotlar içeren bir sınıf alınmıştır. Ele alınan sınıf hesaplarla ilgili olmasına ragmen hesaplama yöntemi farklı hesap türlerine göre farklılıklar gösterebilir. Ek olarak, üzerinde çalıştığımız "hesaplaAccount" sınıfından türetilmiş başka bir "getAccount" alt sınıfı bulunmaktadır. "getAccount" içerisindeki hesaplamalar hesap bilgileri türüne göre yapılmaktadır. Aynı sınıf içerisindeki "BookAccount", "Transaction" ve "Entries" metotları için hesaplama yöntemi ise "if-bloklarıyla" birbirlerinden ayrılan hesap bilgilerine göre farklılık göstermektedir. Bu prensibi uygulamak için, sınıf içerisindeki muhasebe hesaplama işleminin yapıldığı "calculateAccount" sınıfı ilişkili sınıftan türetilerek ve muhasebe hesap türlerine göre yeniden şekillendirilmiştir.

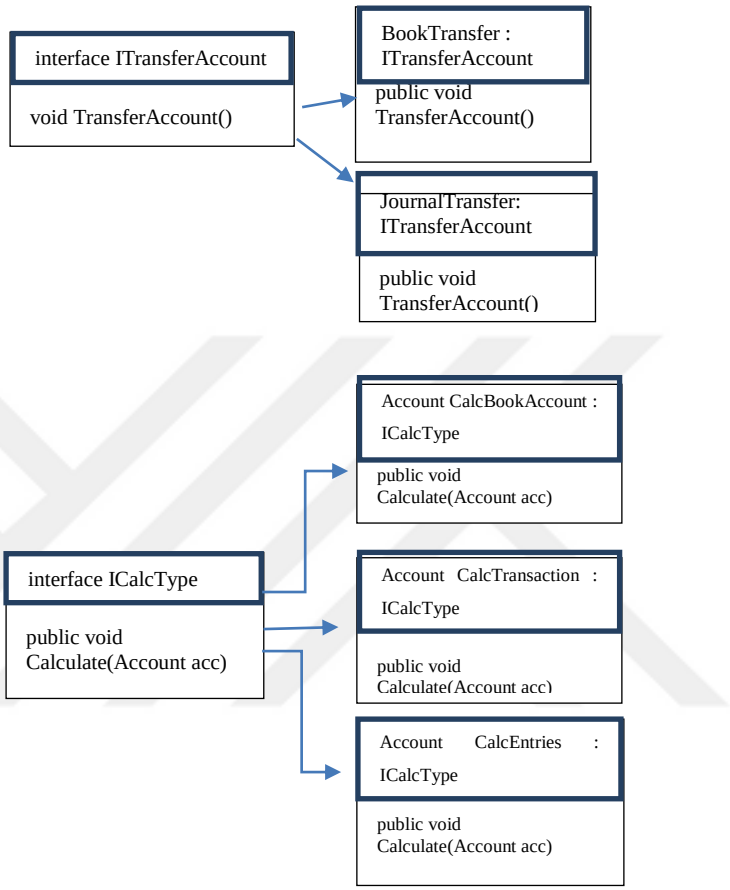
Liskov İkame Prensibi için değişiklik yaptıktan sonra, projenin bakım yapılabilirlik endeksi yaklaşık yüzde 1,3 oranında artmıştır. Fakat bununla birlikte, siklomatik karmaşıklık yaklaşık yüzde 0.15 oranında artmıştır. (Tablo 5) Bunu ISO/IEC 9126'ya göre değerlendirirsek, temel tiplerin yeniden kullanılabilir ve türetilen tiplerin değiştirilebileceği görülebilmektedir.

Çizelge 3.4 LSP sonrası kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| Personnel | 73 | 1593 | 5 | 269 | 5624 |
| Personnel. Payroll | 73 | 452 | 5 | 79 | 1827 |

Arabirim Ayrıştırma İlkesi: İstemci yazılımı kod nesnesi, kullanmadığı metotlara bağlı kalmamalıdır. Her yazılım nesnesi yalnızca ihtiyaç duyduğu objeyi yazılım içerisinde uygulamalı ve ihtiyaç duymadıklarını yazılım parçası olarak uygulamasına gerek olmamalıdır. Arabirim ayrıştırma ilkesi, yazılım geliştiricisi tarafından kullanılacak olan

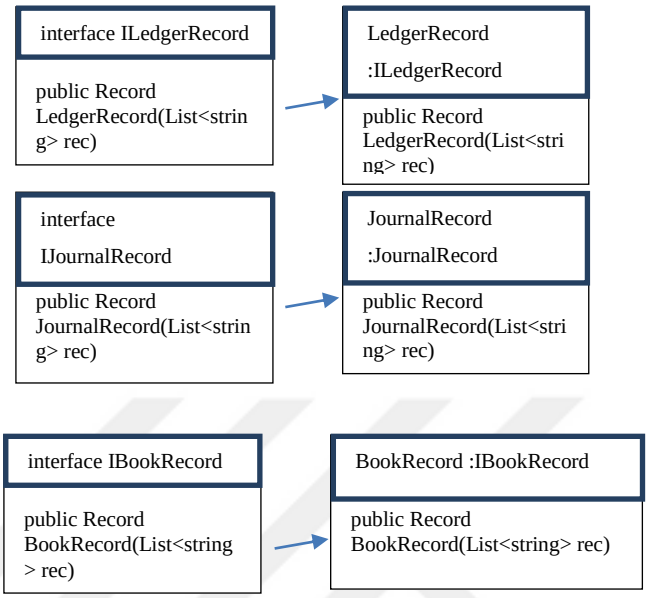
yazılım parçalarını olası en küçük haline getirerek uygulama için yazılacak yazılım nesnelerinin sadece gerek duyulanların kullanılarak gerek duyulmayanların kaldırıldığı bir yapı kurmaktadır.




Şekil 3.6 LSP uygulandıktan sonraki diyagram

Bu prensibi projemizde uygulamak için, muhasebe kayıtlarını tutan ana arayüzler daha küçük fakat gereksiz nesneler içermeyen arayüzlere bölünmüştür. Başlangıçta, "bookRecord", "ledgerRecord" ve "journalRecord" metotlarını içeren "IAccountRecord" arabirimimiz bulunmaktadır. Ancak her metot içerik olarak diğerlerinden farklılık göstermektedir. Bu prensibi uygulamak için "IAccountRecord", "IBookRecord", "ILedgerRecord" ve "IJournalRecord" arayüzleri tanımlanmış ve her sınıf ilgili arayüzden türetilmiştir. Arabirim ayrıştırma ilkesi uygulamasının sonunda, Visual Studio kod metrik aracı yeniden çalıştırılmıştır. İlkenin bakım yapılabilirlik endeksini artırması beklenirken bu değerde herhangi bir değişiklik olmamıştır. Bununla birlikte, siklomatik karmaşıklık yaklaşık yüzde 0,4 oranında artmıştır. Fakat sınıf

bağımlılığı azalmıştır. (Tablo 6) Bu ilkenin amacı, uygulama içerisindeki bağımlılığı azaltarak yazılım bakımının daha kolay olması için yardımcı olmaktır. Sonuçta esnekliği ve yeniden kullanma olasılığını artmıştır.




Şekil 3.7 ISP uygulandıktan sonraki diyagram

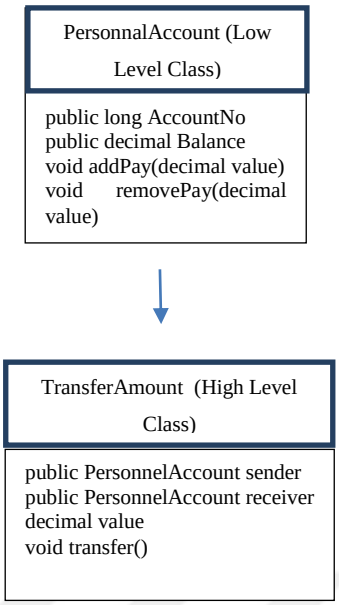
Çizelge 3.5 LSP sonrası kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| Personnel | 73 | 1595 | 5 | 267 | 5624 |

Bağımlılık Değişikliği (Tersine Çevirme) Prensibi (DIP): Esas olarak kod modülleri arasındaki bağımlılığı azaltmakla ilgilidir. Düşük seviyeli nesnelerin, düşük seviyeli nesnelerin sağladığı çok özel uygulamaları kullanabilmeleri için ihtiyaç duyulan yüksek seviyeli nesneler olmadan kullanılabilmeleridir. Bu projenin uygulanmasında proje içerisinde yer alan raporlama ve bildirim için kullanılan sınıflar ile arayüzler kullanılmıştır. Raporlar farklı dosya formatlarında istenebilmektedir. Yapılan bildirimler ise sms veya e-posta olarak iletilmektedir. Bu prensibi uygulamak için rapor oluşturma görevi ve yazdırma bölümü farklı arayüzlere ayrılmıştır. Bildirim yapılma bölümünde bir soyutlama yapılarak bildirim yapılması ile ilgili metotlar bu soyutlamayı kullanır. Sonuç olarak, yüksek ve düşük seviyeli sınıfların her birinin soyutlamalara dayanmasına izin verilir. Bağımlılık değiştirme prensibi uygulamasının sonunda, Visual Studio kod metrik aracı tekrar çalıştırılmıştır. Bakım yapılabilirlik endeksinin

beklendiği gibi arttığı görülmüştür. Arayüz ayrımı sonucunda tanımlamalar içeren yüksek seviyeli modüllere ve detay içeren düşük seviyeli modüllere ait yeniden kullanılabilirlik ve bakım yapılabilirliğin pozitif olarak etkilendiği görülmüştür.

Bağımlılık değişikliği ilkesinin proje üzerinde uygulanabilmesi için çalışan gelirleri ve gelirlerin bilançoya transferi ile ilgili bir sınıf ele alınarak değiştirilmeye çalışılmıştır. Sınıfın değiştirilmeden önceki ilk hali Şekil 3.8'de gösterilmektedir. İlk versiyonda yüksek seviye "TransferAmount" sınıfı düşük seviye "PersonnelAccount" sınıfına bağlıdır. Bu durum, bağımlılığı arttırır. Sınıf içerisindeki "Sender" ve "Receiver" yöntemleri, "TransferAmount" sınıfındaki "PersonnalAccount" türüne başvurmaktadır. Eğer başka bir hesap türü, "PersonnalAccount" içerisinde yer almıyorsa, o zaman bu hesap türlerini kullanmak mümkün değildir. Eğer diğer sınıfa parasal ödemelerle ilgili bir davranış eklemeyi hedefliyor isek, yeni sınıf "PersonnelAccount" kullanılarak türetilmeli veya devralınmalıdır. Ancak, bu durumda yeni sınıf parasal ödemelerin kaldırılmasını uygulayamaz. Bu durum ise Liskov İkame İlkesinin ihlal edilmesi anlamına gelir. Öte yandan, eğer "TransferAmount" sınıfını değiştirmek istiyorsak, bu durum ise Açık-Kapalılık Prensibi ihlal eder. Eğer "PersonnelAccount" sınıfında bir değişiklik yaparsak bu değişiklik "TransferAmount" sınıfını etkiler. Yazılım büyüdüğünde yazılım daha karmaşık hale gelerek değiştirilebilirliği zor bir hale gelebilir. İşlevsellik değiştirilirken veya yeni eklemeler yapılırken zaman en büyük bir maliyet olarak karşımıza çıkabilir. Bu nedenlerden dolayı, yazılımlara bağımlılık değişikliği prensibi uygulanır. Proje üzerinde DIP uygulandıktan sonra, ikinci versiyon Şekil 3.9'da gösterilmektedir.

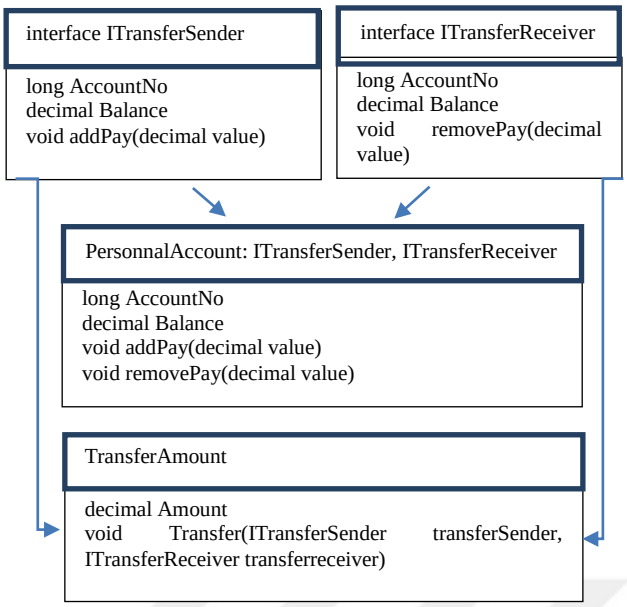



Şekil 3.8 Sınıfların ilk versiyonu

DIP proje üzerine uygulandığı zaman arayüz veya soyut sınıfları kullanarak yapılan değişikliklerle sınıfların bağımlılıklarında azalma olduğu görülecektir. Değişiklik ile alt seviye sınıflar, arayüzleri uygular veya soyut sınıflardan miras alır. Böylece, yeni sınıfların projenin diğer kısımlarına herhangi bir olumsuz etkisi olmadan kullanılabilmesi sağlanmış olur. Yazılımın esnekliği artar. Bu prensibin uygulanabilmei için ekstra çaba gerekebilir ve kod görünümü karmaşık olabilir. Ancak bakım yapılabilirliği ve yeniden kullanılabilirliği artarak sınıfların bağımlılığı azalır.

Çizelge 3.6 VS kod metrik sonuçları

| | MI | CC | DIT | Kenetlilik | LOC |
|---|---|---|---|---|---|
| Personnel | 79 | 1561 | 5 | 259 | 5629 |
| Personnel. General | 73 | 98 | 5 | 49 | 430 |
| Personnel. Report | 68 | 109 | 5 | 92 | 367 |
| Personnel. Payroll | 73 | 453 | 5 | 76 | 1833 |
| listUpdate | 51 | 5 | | 14 | 23 |
| takeFormData | 43 | 4 | | 16 | 33 |
| dataSave | 52 | 5 | | 14 | 19 |




Şekil 3.9 DIP uygulandıktan sonraki versiyon

Bu çalışmadaki elde edilen tüm sonuçlar kodlama tekniğine bağlı olarak değişebilir. Ancak, tüm ilkelerin uygulanması sonucunda bakım yapılabilirlik endeksi değeri artacaktır. Tüm ilkelerin uygulanmasından sonra oluşan kod metrik değerleri Çizelge 3.6'da gösterilmektedir.

Şekil 3.10 Projenin sınıf diyagramı

### 3.2.1 İlgili çalışmalarla karşılaştırma

Literatürde tek sorumluluk ilkesi hakkında fazla çalışma yoktur. Ancak yeniden düzenleme, çok büyük sınıfların bölünmesi, görevlerin ayrılması ile ilgili anahtar kelime ile arama yaptığımızda, çalışmaların ve makalelerin olduğunu görüyoruz. Genel olarak kod kalitesinin yeniden gözden geçirilmesi, artırılması ve bakım maliyetinin etkisine ilişkin birden fazla proje düşünülerek araştırmalar yapılmıştır. (Hegedusa vd, 2017) Hegedus ve diğer çalışma arkadaşları, yazılımın bakım yapılabilirliğinin deneysel değerlendirilmesi konusunda bir çalışma yapmışlardır. Yeniden yapılanma kavramı, geliştirme sürecinin temel bir parçasıdır. Fowler (Fowler, 1999), yeniden düzenlemeyi

tanımlamak için ilk tekniğin kod karmaşası üzerinde çalışmak olduğunu öne sürmüştür. Bu çalışma bakım ve kaynak kodu ölçümlerindeki farklılıkları yeniden yapılandırma ve yeniden yapılandırma olmaksızın kaynak kodu olarak karşılaştırmaktadır. Çalışmanın sonucunda, yeniden yapılanma yapılan kaynak kod elemanlarının, yeniden yapılanmalardan etkilenmeyen öğelere kıyasla önemli derecede daha yüksek bir bakım yapılabilirlik değerine sahip olduğu görülmüştür. Dahası, yeniden yapılandırılmış elemanlar yapılanma öncesinde önemli ölçüde daha yüksek boyutta karmaşıklık ve bağımlılığa sahipti. Ayrıca incelenen kod metrikleri yeniden yapılandırılmış elemanlarda daha belirgin bir şekilde değişmiştir. Araştırmamızda, kaynak kodun tasarım ilkelerine bağlı olarak yeniden ele alınması durumunda, kodun yüksek uyum, düşük bağımlılık, yüksek bakım yapılabilirlik endeksi değerlerine ulaşabileceğini göstermektedir. Başka bir çalışma (Ampatzoglou, 2019), tüm yazılım için bakım yapılabilirliği artırmak için, yazılımın küçük bir bölümünü dâhil etmek yerine yazılım bütününe uygulanacak olan bir yeniden düzenleme işleminin yapılması gerektiğini belirtir. (Herbold, 2009), kod metrikleri, kodlama karmaşıklığı ve geliştiricilerin yürüttüğü yeniden düzenleme faaliyetleri arasındaki ilişkiler hakkında yeterli bilgi bulunmadığı belirtilmiştir. Ancak bu çalışmamız metrikler ve yeniden yapılanma aktiviteleri arasındaki ilişkiyi göstermektedir. Kod metriklerinin iyi bir tasarım ile yeniden düzenlemesine bağlı olarak ne şekilde değiştiği gösterilmiştir. (Haas vd., 2016), (Silva, 2014) gibi diğer araştırmalar en sık yapılan yeniden düzenleme yaklaşımlarının, sınıf yeniden düzenleme ve sınıfı parçalara bölerek yapılan yeniden düzenleme yöntemi olduğu üzerinde durmaktadır. Çalışmamızda belirttiğimiz SOLID tasarım prensiplerine bağlı olarak iyi bir yeniden düzenleme yapmak için, yazılım geliştiricisinin sınıfı parçalara ayırma ve sınıf kod parçacıklarını taşıma yönteminin kullanılabileceği gösterilmiştir.

### 3.2.2 Değerlendirme

Çalışmada etken olarak kullanılan tasarım prensipleri temiz kod yazma ve uygulama konusunda yazılımcılara yardımcı olmaktadır. Yazılım projeleri her türlü kod geliştirme ile tamamlanıp geliştirilen yazılımlar uygulamaya alınabilir. Fakat tasarım prensipleri veya temiz kod, yazılıma uzun dönem sağlıklı olarak uygulamanın çalışması,

genişletilebilmesi, bakım yapılabilmesi imkânı sağlar.

Çalışmanın sonucunda, tasarım ilkelerine bağlı kalınarak yeniden yapılanma yapılan kaynak kod elemanlarının, yeniden yapılanmalardan etkilenmeyen öğelere kıyasla önemli derecede daha yüksek bir bakım yapılabilirlik değerine sahip olduğu görülmüştür. Dahası, yeniden yapılandırılmış elemanlar yapılanma öncesinde önemli ölçüde daha yüksek boyutta karmaşıklık ve bağımlılığa sahipti. Ayrıca incelenen kod metrikleri yeniden yapılandırılmış elemanlarda daha belirgin bir şekilde değişmiştir. Araştırmamızda, kaynak kodun tasarım ilkelerine bağlı olarak yeniden ele alınması durumunda, kodun yüksek uyum, düşük bağımlılık, yüksek bakım yapılabilirlik endeksi değerlerine ulaşabileceğini göstermektedir.

# 4. TEK SORUMLULUK İLKESİNE UYGUN OLARAK YENİDEN YAPILANDIRMA İÇİN BİR YAKLAŞIM

## 4.1 Çalışma Kapsamı

Bu bölümde, Robert Martin tarafından sunulan beş ilkenin birinci ilkesi olan Tek Sorumluluk İlkesi (SRP) için yeniden yapılandırma önerileri sunmaya yönelik bir yaklaşım önerilmektedir. (Martin, 2008). Yaklaşımımız, SRP'nin ihlal edilmesine neden olabilecek olası göstergelerin bir listesine odaklanmaktadır. Bununla birlikte, sınıf ve metot isimlendirmesi arasındaki kavramsal ilişkileri de değerlendirilmiştir. Tezdeki varsayım, isimlendirmenin anlaşılabilirlik ve okunabilirlik için şart olduğu ve bir yazılım birimi içindeki belirli bir derecedeki kavramsal ilişkiyi yansıttığıdır.

Çalışmanın bu aşamasında 5 tasarım ilkesinden sadece bir tanesi ele alınarak bu yönde işlemler yapılmıştır. Bunun için tasarım ilkelerinden birincisi olan tek sorumluluk ilkesi uygulanması için çalışma yapılmıştır. Tek sorunluluk ilkesinin metot ve sınıflar üzerinde uygulanabilmesi için öncelikle bu ilkenin sağlanması ve ilkenin yıkılması şartları ortaya çıkarılmıştır. Daha sonrasında ise bu şartları ölçtüğü düşünülen bir yazılım geliştirilmiştir. Geliştirilen yazılım istenilen bir c#.net ile geliştirilmiş uygulama kodlarını otomatik olarak incelemekte, analiz etmekte ve belirlediğimiz tasarım ilkesini yıkan metot ve sınıfların tek sorumluluk ilkesine göre nasıl tekrar yeniden yapılandırılabileceği konusunda yazılımcılara öneriler sunmaktadır.

Geliştirilen yazılımın doğruluğunu sağlayabilmek amacıyla, öncelikle geliştirilen uygulama 2 kurumsal proje üzerinde test edilmiştir. Test sonuçları deneyimli yazılımcılarla paylaşılarak onların fikri alınmıştır. Bunun nedeni yazılımın manuel olarak doğruluğunu kanıtlamaktır. Daha sonrasında ise uygulama internet üzerinden indirilen açık kaynak kodlu olarak kullanılan ve tasarım ilkelerine uygun olarak yazıldığı ifade edilen bir uygulama çerçevesi üzerinde test edilmiştir. Aynı uygulama çerçevesi tarafımızca geliştirilen uygulamaya benzer bir işlev ile çalışan ve internet üzerinden sıklıkla kullanılan başka bir uygulama ile de analiz edilmiştir. En sonda ise bizim geliştirdiğimiz ve internet üzerinden kullanılan uygulama sonuçları

karşılaştırılarak uygulamamızın dolayısıyla tasarım ilkelerinin uygulanması ile ilgili öne sürdüğümüz tezlerin doğruluğu kanıtlanmaya çalışılmıştır.

## 4.2 Yöntem

İleri sürülen yöntemde, SRP için önemli göstergeler ve WordNet sözcüksel veritabanı vasıtasıyla kavramsal ilişkiler kullanılmaktadır. Bu şekilde yazılımdaki olası yeniden yapılandırma fırsatlarının bir listesi yazılım geliştiricisine sunulur ve bunlar bir puanlama işlevi kullanılarak sıralanır. Önerilen yöntemi doğrulamak için, geliştirilen bir ilk örnek aracıyla, beş deneyimli geliştiriciyle bir kullanıcı değerlendirme çalışması kullanılarak iki kurumsal proje üzerinde yaklaşım değerlendirilmiştir. Geliştirilen uygulama tarafından sunulan öneriler yazılım geliştiricileri tarafından değerlendirilmiş ve yüzde 70 oranında faydalı bulunmuştur. Ayrıca, geliştirilen uygulama internet üzerinden indirilen ve .Net ile yazılmış olan açık kaynak kodlu bir uygulama çerçevesi üzerinde de çalıştırılmıştır. Elde edilen sonuçlar ise .Net ve Java tabanlı IDE'lerde kod yeniden yapılandırma aracı olarakyaygın bir şekilde kullanılan SourceMonitor adındaki uygulamadan elde edilen sonuçlarla karşılaştırılmıştır. Önerdiğimiz yönteme göre geliştirilen uygulama ile diğer uygulamadan elde edilen sonuçlar birbirlerine paralellik göstermekle birlikte sınıf ve metot isimlendirmesi arasındaki anlamsal ilişkiye dayalı önerilerin dikkate alınmadığı ve dâhil edilmediği durumlarda SRP için önemli yeniden yapılandırma fırsatlarının bazılarının kaçırılabileceği gözlenmiştir.

Yazılımın kaynak kodlarının ve yazılım dokümanlarının anlaşılabilirliği çeşitli nedenlerden dolayı önemlidir. Yazılım geliştiriciler, meslektaşları tarafından yazılan belgeleri okumak zorundadır ve yazılım geliştiriciler, daha önce bilmedikleri eski kaynak kodları incelemeye ihtiyaç duyarlar (Laitinen, 1995). Yazılım geliştiricileri başkaları için kod yazar çünkü yazılım aslında kullanıldığı sürece devamlı olarak canlı tutulmalıdır. Kod yazarken isimlendirmenin neden önemli olduğunun özü budur. İyi adlandırma, özniteliklerin amacını açıklar ve geliştiricinin bakıma nerede başlayacağı konusunda bir ipucu verir. Önerilen yöntemde WordNet'i kullanarak bakımın yapılabilirliğine katkıda bulunulduğu düşünülmüştür. Örnek olay bölümünde

görülebileceği gibi, adlandırma önerileri sistem tarafından WordNet'in benzerlik değerlerine göre üretilmektedir. Bu şekilde anlamsal bir benzerlik kullanılmasa idi, yazılımda anlamsal olarak yeniden yapılandırma için kaçırılmış fırsatlar bulunacak idi.

### 4.2.1 İlgili çalışmalar

Martin'e göre (Martin, 2003) modülerlik seviyeleri tek sorumluluk prensibi (SRP) uygulanarak sağlanabilir. Başka bir makalede (Ampatzoglou vd, 2019) yazarlar yazılım modülerliğine odaklanmışlardır. SRP'nin modülerlik ve kodun yeniden kullanımı üzerinde olumlu bir etkisi olduğu sonucuna varmışlardır. Etkiyi analiz etmek için farklı seviyelerde ayrıntı düzeyi göz önünde bulundurulmuştur; yöntem seviyesi, sınıf seviyesi ve paket seviyesi. Yeniden yapılanmanın genel olarak modülerlik üzerindeki etkisi ve bağımlılık ile bağıntı arasındaki değişimi, üç seviyenin tümü için incelenmiştir. SRP'nin modülerlik ve her seviye için tekrar kodlama üzerinde olumlu bir etkisi olduğu kanıtlanmıştır.

Benzer bir çalışmada SRP büyük metotların ayrıştırılması ve bölünmesi içinde kullanılmış ve SRP uyumunun sistemlerin tasarım kalitelerini artırdığı görülmüştür. (Charalampidou vd, 2017)

Diğer taraftan bağıntıyı artırabilmek için otomatik metot bölme teknikleri üzerine çalışmalar yapılmıştır. Genel olarak metot bölme için bir kural oluşturulması veya ölçüt belirlenmesi zordur. Bu nedenle metot bölme işlemlerinde yeniden yapılandırma önerileri vermek için makine öğrenmesi temelli yöntemler değerlendirilmiştir. Bu yönde karmaşıklık, bağıntı ve bağımlılıkların belirlenmesi için yapısal özellikler ve fonksiyonel özellikler kullanılmıştır. (Xu vd., 2017) Geleneksel yazılım metriklerinin aksine, kod bloklarının fonksiyonalitesinin veri erişimi, paket erişimi, metot çağırımları gibi özellikleri temsil ettiği kabul edilmiştir. Bunun yanısıra yeniden yapılandırma kod önerilerinin verilebilmesi amacıyla çalışmalar yapılmıştır. (Silva vd, 2016), (Ouni, 2017) (Fontana vd, 2015). Fakat bu çalışmalarda tasarım prensiplerinden bahsedilmemiştir.

Yeniden düzenleme için potansiyel adayların belirlenmesinin ardından, bir çeşit önceliklendirme mekanizması veya önerileri yeniden düzenlemek için bir sıralama yöntemi gereklidir. (Haas vd, 2016) Örneğin, Silva (Silva vd, 2014), IDE tabanlı yeniden düzenleme araçları tarafından otomatik hale getirilecek yeniden düzenleme fırsatlarını belirlemek ve sıralamak için bir yaklaşım sunmuştur. Bu yaklaşım metot bölünmesi için adaylar üretir ve adayların listesini programın anlaşılmasını arttırma potansiyellerine göre sınıflandırmak için bir sıralama fonksiyonu önerir. Sıralama fonksiyonu, "bağlantıyı en aza indirgeme" ve "uyumu en üst düzeye çıkarma" tasarım kılavuzlarından ilham almıştır.

Sorunlu kod bölümlerini rapor edebilen statik kod analiz araçları vardır. (Charalampidou vd., 2017) Bununla birlikte, statik analiz araçları yalnızca potansiyel hataları, kalıpları önleme ve kodlama kuralı ihlallerini belirleme yeteneğine sahiptir, ancak SRP için yeniden düzenleme fırsatlarını çıkarmaz (Szoke, 2017). Yazarlar (Herbold vd, 2009) kod analiz araçlarının sonuçlarını kullanarak otomatik yeniden değerlendirme önerileri oluşturmaya çalışmışlardır. Makalede, statik analiz araçları ile yeniden düzenleme araçları arasında bir boşluk olduğunu belirtilir. Bu sorunu gidermek için bir yeniden düzenleme aracı olarak kod analiz araçlarının sonuçlarını kullanarak, yazılımın kalitesini iyileştirmek için otomatik bir yaklaşım sunarlar. (Charalampidou vd, 2017) belirli bir işlevsellik sağlamak için birbiriyle bütünleşmiş çalışan kaynak kod parçalarını tanımlamayı ve kod parçalanmalarını ayrı yöntemler olarak önermeyi amaçlayan bir yaklaşım ortaya koymuşlardır. Ek olarak, tek sorumluluk ilkesi kullanımı önerilmektedir. Uzun metotların yazıldığı uygulamalarda SRP'yi ihlal etme eğiliminde olmasına rağmen, literatürde tek sorumluluk ilkesine uygunluklarını kontrol ederek metotların bölünmesini, ayrıştırılmasını belirlemeyi amaçlayan hiçbir yaklaşım yoktur.

Yeniden düzenleme önerilerinin tespiti ve sıralaması da incelenmiştir. Örneğin, (silva, 2014), IDE tabanlı yeniden düzenleme araçlarıyla otomatik hale getirme fırsatlarını yeniden yapılandırma yöntemlerini tanımlamak ve derecelendirmek için bir yaklaşım sunmuşlardır. Yöntemleri, yeniden düzenleme yapılabilecek metotların çıkarılması için aday metotların üretilmesi ve metotların anlaşılmasını geliştirme potansiyellerine göre, önerilen yaklaşım tarafından oluşturulan ilk adayların listesini sınıflandırmak için bir sıralama işlevi yapmaktır. Sıralama fonksiyonu, bağlantıyı en aza indirgeme ve uyum tasarım kılavuzunu en üst düzeye çıkarma yönteminden ilham almıştır. Uzun yollarla

refaktöre en iyi adayı bulmak için yapılan yorumları içeren bir başka puanlama fonksiyonu göz önünde bulundurulur.

Diğer taraftan, WordNet, farklı çalışmalarda kavramların benzerliğini ölçmek için kullanılmıştır. Benzerlik ölçüleri, bir hiyerarşideki bilgilere dayanarak, iki kavramın ne kadar birbirine eşleştiğini belirler. Örneğin, bir otomobil bir ağaçtan daha çok bir anlamsal olarak bir tekneye daha yakın gibi düşünülebilir. Çünkü otomobil ve tekne bir hiyerarşide en üstte ortak bir ata olarak taşıt grubunda yer alabilirler. (Pedersen, 2004) Literatürde, anlamsal benzerliği belirlemek için başlıca iki yaklaşım vardır. İlk yaklaşım, metin tabanlı yaklaşım olarak adlandırdığımız anlamsal benzerlik puanını tahmin etmek için büyük bir kitaplık veya sözcük tanımlarından faydalanmak ve bu kaynaklardan istatistiksel veriler toplamaktır. İkinci yaklaşım, yapı tabanlı yaklaşım olarak adlandırdığımız WordNet gibi eş anlamlıların ilişkilerini ve hiyerarşisini kullanır. Metin tabanlı yaklaşımda, kelime ilişkileri genellikle bir kitaplıktaki eş-oluşum dağılımlarından kaynaklanmaktadır.

### 4.2.2 Wordnet'in önerilen yaklaşımda kullanılması

Bu çalışmada önerilen yöntemin gerçekleştirilmesi aşamasında, yöntem isimleri ve sınıf isimleriyle ilgili bir problem vardı. Bu sorun, adların alt çizgi ile ayrılmış kelimelerle birlikte yazılmasından ya da çoklu kelimelerin ilk harflerinin büyük harfleriyle yan yana yazılmasından kaynaklanmakta idi. Bu sorunun üstesinden gelmek için, şu şekilde bir yol izlenmiştir.

1- İsimleri karşılaştırırken, eğer sınıf isimleri veya yöntem isimleri alt çizgi içeriyorsa, isim dizgisi bu özel karakterlere göre bölünür.
2- Sınıf adları veya yöntem adları büyük harf içeriyorsa ve özel karakter içermiyorsa, bu adlar değerlendirilir ve bu birleşik kelime setleri farklı sözcükler olacak şekilde ayrıştırılır.

Örneğin, yöntem adı "onExit" ise veya başka bir yöntem adı "takeElementValue" ise, ilk kelime seti "on" ve "Exit" olarak ayrıştırılır, ikinci kelime seti "take", "Element" ve "Value" sözcükleri olarak ayrıştırılır. Daha sonrasında bu iki kelime setlerinden oluşan

kelimeler arasında bir WordNet karşılaştırması yapılır. Bütün projeyi içerecek şekilde tüm metot isimleri Wordnet değeri için sınıf isimleri ile kontrol edilir. Ayrıca aynı sınıftaki yöntem ilişkilerine de bakılmıştır. Metot ilişkileri, kullanılan ortak niteliklerle veya ortak referanslarla hesaplanmıştır. Yöntemler ortak özellikler veya ortak referanslar kullanıyorsa, SRP ihlali olasılığı daha düşük olacaktır.

Aşağıdaki tablo (Tablo 8), yöntem adları ile sınıf adları arasındaki benzerliği ve bu benzerlikten sınıfa göre toplam kavramsal uyum değerini gösteren örnek bir sınıf içermektedir (Şekil 4.1).

```
publicclassCreditController : BaseController
{
privatereadonly ICreditService _creditService;
privateobject creditId;

public CreditController(ICreditService creditService)
   {
      _creditService = creditService;
   }
public ActionResult AddToCredit(CreditManager basket, int creditId)
   {
var creditresult = _creditService.Get(creditId);
var credit = new Credit
      {
         Credit = creditresult,
         Quantity = 1,
      };
      basket.Add(credit);
return PartialView("CreditSummary", basket);
   }
public ActionResult RemoveFromCredit(CreditManager basket, int creditId)
   {
      basket.Remove(creditId);
return PartialView("CreditSummary", basket);
   }
public ActionResult Index(CreditManager basket)
   {
return View(basket);
   }
public PartialViewResult CreditSummary(CreditManager basket)
   {
return PartialView(basket);
   }
public ActionResult TakeFree()
   {
return View(new BindDetail());
   }
}
```

Şekil 4.1 Tanımlayıcı isimlendirme içeren örnek bir sınıf

Çizelge 4.1 Örnek Benzerlik Tablosu (*: metot değerinin aritmetik ortalaması)

| **Sınıf / Metot Adı** | **Tipi** | **Karşılaştırılan Kelime** | **Sınıf İçerisinde Bulunan Kelime** | **Benzerlik Oranı** |
|---|---|---|---|---|
| CreditController | Class | Credit, controller | | 0,68* |
| AddtoCredit | Method | Add, credit | Credit, credit | 1 |
| RemovefromCredit | Method | Remove, credit | Credit, credit | 1 |
| Index | Method | Index | Index, credit | 0,16 |
| CreditSummary | Method | Credit, summary | Credit, credit | 1 |
| TakeFree | Method | Take, free | Take, credit | 0,25 |

Bir yöntemin Wordnet puanı aşağıdaki formülde gösterildiği şekilde belirlenir. Ayrıca, bir sınıftaki net puanı hesaplamak için, o sınıftaki yöntemlerin WordNet değerlerinin aritmetik ortalaması kullanılır.

$$\mathrm{WP(M)} = \mathrm{WS\ (MN,\ CN)}$$

Bu formülde;

1- WP, metodun WordNet değeri.
2- WS, metot ve sınıf isimleri arasındaki WordNet benzerlik değeri,
3- MN, metot adı
4- CN, sınıf adı

olarak ifade edilmiştir.

$$\mathrm{WP(C)} = \sum_{k=0}^{n} \mathrm{WP(Mk)} \ / \ \textit{Number of Methods in C}$$

Bu formülde;

1- WP, sınıfın WordNet değeri.
2- n, sınıf içerisindeki metot sayısı.
3- WV, bit metot için WordNet benzerlik değeri.
4- K, üzerinde işlem yapılan metot
5- N, sınıf içerisindeki metot sayısı

olarak ifade edilmiştir.

### 4.2.3 Yapısal bağıntının çalışma içerisinde kullanılması

Yapısal uyum değerini bulmak için ortak özelliklerin kullanılıp kullanılmadığına veya ortak sınıf veya yöntemlerin çağrılıp çağrılmadığına bakılmıştır. Bu ölçütlere göre, önerilen yöntemin uygulanmasında deneysel olarak kullanılan proje içerisindeki bütün yöntemler değerlendirilmiştir. Önerilen yöntemde, proje içerisindeki herhangi bir yöntemde eğer ortak bir değer kullanılırsa veya ortak bir yöntem çağrılırsa, o metot için yapısal uyum değeri bir artırılır. Bu işlem tüm yöntemler için gerçekleştirilir ve ölçütleri karşılayan her ölçüt için aynı yöntemin yapısal uyum değerine bir puan eklenir. Her ortak değişken ve her ortak yöntem veya sınıf için toplam yapısal uyum değeri bir artırılır. Buradaki en önemli işlemlerden biri, kontrol edilen yöntemin diğer yöntemlerle bağlantısı olup olmadığını belirlemektir. Bu bağlantı ortak özellik ve kullanılan ortak yöntem aracılığıyla bulunmaya çalışılmıştır.

Bu çalışmada, LCOM4 içerisindeki gibi ölçüm göstergelerin hesaplanmasında kullanılan ortak değişkenlerin yanı sıra kullanılan ortak metotlarda bir bağıntı göstergesi olarak değerlendirilmiştir. Fakat metotlar arasında yeniden yapılandırma yapılacak olanların skorlanarak listelenmesi aşaması olduğu için ölçüm göstergesi olarak LCOM4 gibi herhangi bir değer aralığı kullanılmamıştır. Aynı zamanda kullanılan birden fazla ortak değişkenden sadece bir tanesine bağıntı var ise yine bu metot bağıntılı olarak kabul edilmiştir.

Sınıftaki yöntemlerin yapısal uyum değerini hesaplamak için ayrıca, her yöntem için sınıfta tanımlanan ve yöntemde kullanılan ortak özellikler göz önünde bulundurulmuş ve buna ek olarak farklı sınıf objelerine veya yöntemlerine yapılan çağrılar veya gösterilen referanslarda ele alınmıştır.

Yapısal bağıntıyı formüle edecek olursak;

$$\mathrm{SC(M)} = \sum_{k=0}^{n} CA(MkC) + \sum_{k=0}^{n} CC(MkC) + \sum_{k=0}^{n} CM(MkC)$$

Bu tanımlamada;

1- SC, bir metot için yapısal bağıntı değeri.

2- CA, bir sınıf içerisinde tanımlanan ve metot tarafından kullanılan özelliklerin sayısı.
3- CC, metot içerisinde kullanılan veya referans edilen ortak sınıf sayısı.
4- CM, metot içerisinde kullanılan veya referans edilen ortak metot sayısı.
5- n, metot sayısı.
6- k, metot numarası
7- C, üzerinde çalışılan sınıf

olarak ifade edilmiştir.

### 4.2.4 Metod uzunluğunun çalışma içerisinde kullanılması

Önerilen yönteme uygun olan yazılan uygulamada, eğer bir yöntemin kod satır sayısı (LOC) 30'dan fazla ise, o zaman uzun bir yöntem olarak kabul edilebilir (Lippert vd, 2006) ve SRP uyumu yapılacak bir metot olarak göz önünde bulundurulacak potansiyel bir yöntem olarak işaret edilmelidir.

### 4.2.5 Parametre sayısının çalışma içerisinde kullanılması

Bu çalışmada parametre sayısı "Kuramsal Temeller" kısmında anlatılan parametre sayıları normal olarak kabul edilmiştir, çünkü 4'ten fazla parametre sayısının geliştiriciler için kavramsallaştırılması zor olabilmektedir.

Parametre sayısı dördü aşarsa;

a. Yöntem çok fazla iş yapıyor ve olası yeniden yapılandırma işlemine ihtiyaç var.
b. Bir toplama veri yapısının kullanımı düşünülmelidir.

c. Sınıf tasarımı yeniden düşünülmeli; belki bazı şeylerin etrafta geçirilmesine gerek yoktur.

Önerdiğimiz yöntemde yukarıda anlatılanlara bağlı olarak, kullandığımız projelerimizdeki yöntemlerden biri tipik bir gerçek hayat örneği olarak aşağıda verilmiştir. (Şekil 4.2) Bu kod birimi "sendRecord" fonksiyonuna ait dokuz parametre vardır. Böyle bir fonksiyonun kavramsallaştırılması zordur, çünkü yazılım geliştiricisi,

eğer parametrelerin adı açıklayıcı değilse, parametrik argümanları hangi sırayla geçireceğini hatırlamalıdır. Söz konusu yöntemin parametreleri birbirleri ile alakasız ise, o zaman parametrelerin gruplandırılmaları veya anlam ilişkisi içerisinde olmaları da zor olacaktır. Bu nedenle, çok fazla parametre çok fazla sorumluluğun göstergesi olabilir.

```
function sendRecord(firstName, lastName, dateOfBirth, streetAddress, postCode, city,
country, email, website) {
    sendResponse({
        name: lastName + ', ' + firstName,
        DOB: dateOfBirth,
        address: streetAddress + '\n' + postCode + ' ' + city + '\n' + country,
        email: email,
         url: website
    });
}
```

Şekil 4.2 Örnek Sınıf Tanımlaması

### 4.2.6 Metot çağırımlarının kullanılması

"Kuramsal Temeller" kısmında anlatılan şekilde çağırımlar yapan metot ve buna uygun olarak yazılmış sınıflar var ise bunlar önerilen yöntem önermelerine göre yeniden yapılandırma listesine metotların veya sınıfları yeniden düzenlenebilmeleri için eklenmektedir.

### 4.2.7 Uygulama tarafından yapılan önerileri skorlama

SRP için ilk önce yeniden yapılandırma için hangi sınıf veya yöntemlerin dikkate alınacağını belirlemek için bir puanlama işlemi kullanılmaktadır. Yukarıda tanımlanan tüm kriterler projedeki tüm sınıflara ve yöntemlere uygulanmıştır. Kriterlerimize sahip olan her sınıfa veya yönteme bir puan verilir. Böyle bir puanlama işlemiyle maksimum puan 5 olabilir. Bir sınıfın veya yöntemin puanı 0'dan büyükse, SRP'yi ihlal ettiği söylenebilir. Yazılımları yeniden yapılandırma için puanlama ile ilgili yapılan iki çalışma, Haas (Haas vd, 2016) veya Silva (Silva vd, 2014) sadece yeniden yapılandırma işlemi için en uygun aday metodu bulmak üzerine çalışmışlardır. Bu çalışmada ise

yeniden yapılanma için göz önünde bulundurulacak en uygun sınıfları bulmakla kalmıyoruz, aynı zamanda kod birimleri ve her kriter için önerilerde bulunuyoruz. Önerilen yöntemin son aşamasında, tüm proje için kod parçalarına yeniden düzenleme yapmak için geliştiricilere öneride bulunan bir liste hazırlanmaktadır. Bu öneriler listesi, SRP'ye uygun yeniden düzenleme için olası yöntem ve sınıf adlarını içermektedir. Liste aynı zamanda olası kod yeniden yapılandırma nedenlerini, yeniden yapılandırmanın neye göre yapılacağını ve puanlarını da içermektedir. Kapsamdaki bir kod parçası önceki bölümlerde sunulan kriterleri içeriyorsa, puanlama işleminde bu kod parçasına sabit değerler verilir. Puanlama işlevinin yalnızca önceliği belirlemek için kullanıldığı belirtilmelidir. Tablo 9, bu kriterler için puanlama değerlerinin aralığını göstermektedir.

Çizelge 4.2 Elementler ve Karşılık Skorları

| **Kriter** | **Skor** |
|---|---|
| Bağıntı | 0 veya 1 |
| Kod Satır Sayısı (LOC) | 0 veya 1 |
| Parametre Sayısı (NP) | 0 veya 1 |
| Metot İçerisinden Sınıf Çağırma Sayısı (CM) | 0 ve 1 |
| İsimlendirme (WN) | 0 ve 1 arasında |

Toplam skor bu belirtilen değerlerin toplamıdır. Bir metot 0 puana sahip olabileceği gibi 5 puana da sahip olabilir.

Bir Metodun Toplam Skoru = - Bağıntı Değeri + LOC + NP + CCM – WN

Bir Sınıf İçin Toplam Skor = Arayüz Değeri + Herbir Metodun Toplam Skoru / Metot Sayısı

Öncelik skoru, LOC, NP, CM değerlerinin toplamından yapısal bağıntı ve WordNet değerinin çıkarılması ile bulunmaktadır. Yani yapısal bağıntı ve WordNet değeri toplam puanlamayı negatif olarak etkilemektedir. Eğer kod satır sayısı 30'dan fazla ise o zaman LOC değeri 1 olarak kabul edilir. Parametre sayısı 3 ten fazla ise o zaman NP değeri de çok fazla olarak kabul edilerek NP değerine 1 eklenir. Metot çağırma sayısı da belirlenen değerlerin üzerinde ise CCM değerine 1 puan eklenir.

### 4.2.8 Yaklaşımın değerlendirilmesi

Yukarıdaki kriterleri ve bunları kontrol etmek için basit yöntemler belirledikten sonra, .NET ortamında C# dilini kullanarak prototip olarak bir bilgisayar uygulaması geliştirdik. Geliştirilen bu uygulama, değerlendirme yapılacak bütün bir projeyi analiz eder ve SRP ile ilgili öneriler için dikkate alınacak aday kod birimlerini bulur. Kriterlerimizden birisi olan WordNet anlamsal kütüphanesi için, Visual Studio.Net Nuget paketindeki Syn.WordNet kütüphanesinde bulunan WordNet motoru kullanılmıştır. Syn.WordNet kütüphanesi, WordNet'in sözcüksel veritabanı dosyalarını ve sözcük setlerini (SynSets) içe aktarmak için kullanılır. Uygulama geliştirilirken kullanılan Syn.WordNet taşınabilir bir sınıf kütüphanesi olduğundan, WordNet dosyalarını yerel makineye indirdikten sonra, WordNet motoru ile kelime setlerini yükleme ve bu setleri karşılaştırarak benzerlik oranlarını kullanma mekanizması StreamReader nesnesi tarafından yapılmıştır.

Belirlediğimiz ölçütlerin tamamını geliştirdiğimiz uygulama içerisinde yazılımsal olarak geliştirdikten sonra, SRP yaklaşımını değerlendirmek için iki örnek olay incelemesi yapılmıştır. İlk çalışma, beş geliştiricinin iki kurumsal projedeki değerlendirmeleri kullanılarak yapıldı. İkinci çalışmada ise, geliştirilen araç kalite ilkelerine uygun olarak yazılmış bir uygulama çerçevesi üzerinde yürütülmüştür. Açık kaynak kodlu olarak geliştirilmiş ve .Net ve Java yazılımları üzerinde analizler yapan bir uygulama (SourceMonitor) aynı uygulama çerçevesi üzerinde çalıştırılmıştır. Bu sonuçlar ile bu çalışma kapsamında geliştirilen benzer uygulama sonuçları karşılaştırılmıştır.

### 4.2.9 Vaka çalışması 1

İlk vaka çalışmasında, yaklaşım kurumsal seviyede kullanılan iki yazılım projesine uygulanmaktadır. Bunlardan biri 48 sınıf ve 148 yöntem içeren bir insan kaynakları yönetimi projesiydi. Proje içerisinde veri yönetimi, personel takibi, muhasebe ve bordro sistemi, raporlama vb. modüller bulunuyordu. İkinci proje, organizasyon paydaşlarının ayrıntılı bilgilerini kaydetmek ve yönetmek için geliştirilmiş bir uygulamadır. Bu proje ise 78 sınıf ve 256 yöntem içermektedir. Her iki proje de katmanlı mimari yapıda ve

.Net ortamında geliştirilmiştir. ORM (Nesne İlişkisel Haritalama) aracı olarak Entity Framework ve NHibernate kullanılmıştır. Projeler, katmanlı mimariye uygun olarak UI katmanı, iş mantığı ve veri erişim katmanı içermektedir.

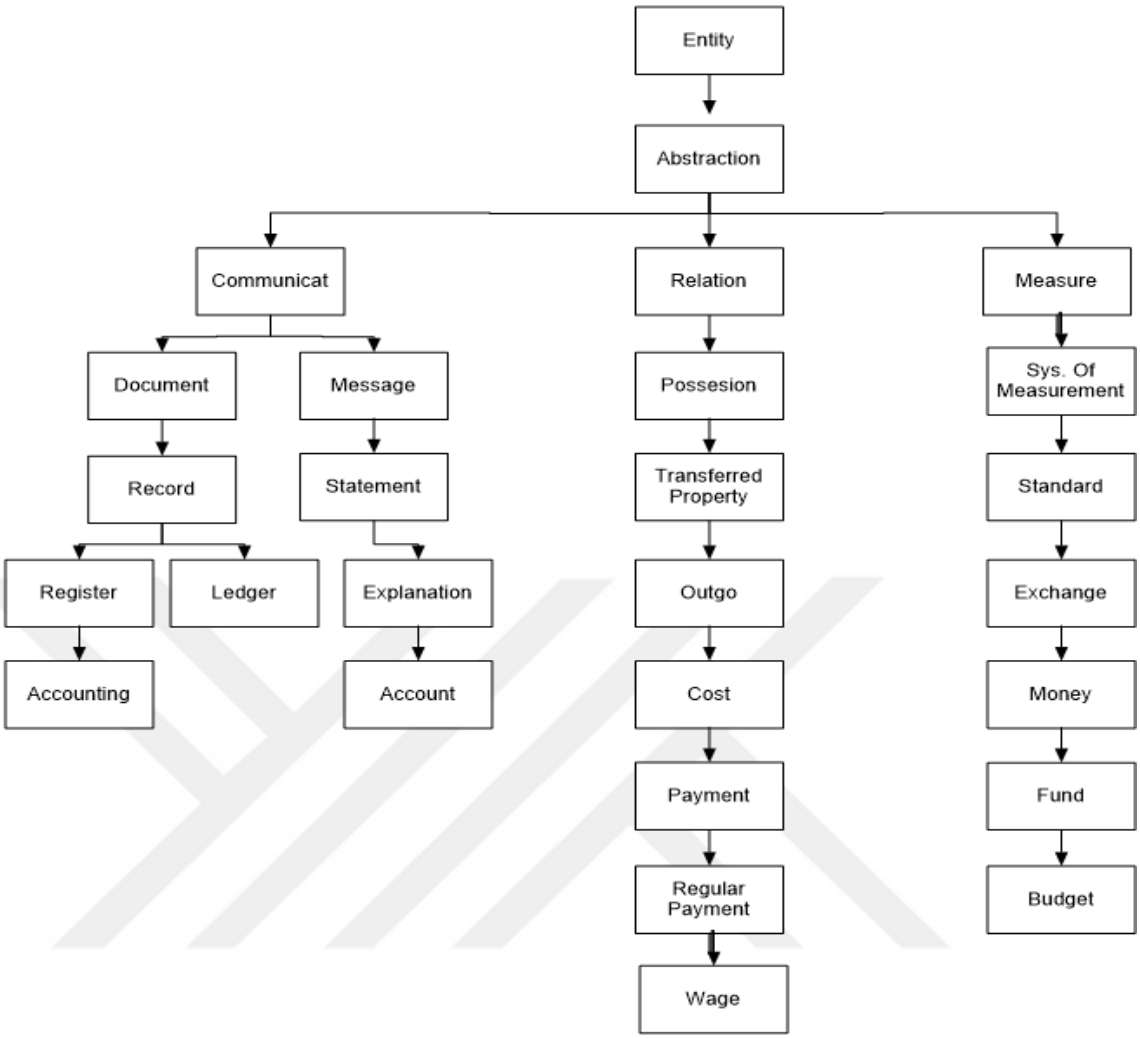
İlk olarak, yöntemden elde edilen sonuçları değerlendirmek için manuel bir yol izledik. Bunu yapmak için, iki proje için de yöntemin sonuçları beş deneyimli yazılım geliştirici tarafından kontrol edildi. Yazılımcılar öneriler, sınıflar ve yöntemler için oluşturulan puanlar hakkında değerlendirme yaptılar. Değerlendirmeden sonra geri bildirimde bulundular. Tablo 10 da gösterilmiştir.

Çizelge 4.3 Geri bildirim tablosu

| **Metot/ Sınıf Adı** | **Toplam Skor** | **Bağıntı Değeri** | **LOC** | **NP** | **CM** | **WN** |
|---|---|---|---|---|---|---|
| **Metot A** | 0 | 0,5 | 0 | 0 | 0 | 0 |
| **Metot B** | 0 | 0,7 | 0 | 0 | 0 | 0,4 |
| **Metot C** | 1,45 | 0,85 | 1 | 1 | 1 | 0,7 |
| **Sınıf ABC** | 1,45 | 0,85 | 1 | 1 | 1 | 0,7 |

Bu çalışma için, bir sınıfın örnek yöntemleri, geliştirilen uygulamanın çıktısının bir özeti tablo 6'da gösterilmektedir. A yöntemi için hesaplanan yapısal uyum 0,5 ve B yöntemi için 0,7'dir. Metot A ve B'nin 5 geliştirici tarafından oldukça uyumlu (highly cohesive) olduğu ve SRP'yi ihlal etmediği kabul edilmiştir. Öte yandan, C metodunun yeniden yapılanma için dikkate alınmasına ve bu metodun iki metot halinde ayrılmasına karar verilmiştir.

WordNet değerlendirmesine örnek olarak, insan kaynakları yönetimi projesinin taksonomik ilişkileri aşağıdaki şekilde (Şekil 4.3) verilmiştir. Daha önce belirtildiği gibi, eğer WN puanı sıfırdan büyük ise bileşenler arasında anlamsal bir ilişki vardır. Aşağıdaki tablo (Tablo 11), insan kaynakları yönetimi projesine ait bazı metot ve sınıf adları için hesaplanan bazı benzerlik değerlerini göstermektedir. “Accounting” ve “Ledger” arasındaki en kısa yol “Accounting - Register - Record - Ledger”, yol uzunluğu 4'tür ve bu nedenle benzerlik 0,25'tir. “Wage” ve “Budget” arasındaki en kısa yol “engage– contend- root- think- reason- calculate” ve yol uzunluğu 7'dir. WN değeri ise 0,14 olarak hesaplanır.




Şekil 4.3 Örnek sınıf diyagramı

Çizelge 4.4 Projedeki isimlendirmelerin WordNet benzerlik değerlerinin bir örneği

| Sınıf veya Metot Adı | Sınıf veya Metot Adı | Benzerlik Oranı |
|---|---|---|
| Accounting | Ledger | 0,25 |
| Ledger | Account | 0,14 |
| Sender | Receiver | 0,33 |
| Wage | Budget | 0,14 |
| Wage | Amount | 0,25 |
| Ledger | Journal | 0,66 |
| Account | Bookkeeping | 0,12 |
| Ledger | Book | 1 |
| Journal | Book | 0,8 |
| Loan | Money | 0,14 |
| List | Save | 0,25 |

Yazılımcılara değerlendirmeleri amacıyla beş adet sınıf verilmiş ve bu sınıflar sıralama işleviyle belirlenmiştir. Geliştirilen aracın çıktısı, kriterlerimize göre, sınıf adları, metot adları ve SRP'yi ihlal etmenin olası nedenlerini içermektedir. Geliştiricilere aşağıdaki sorular sorulmuş ve değerlendirmelerini bu sorulara göre yapmaları istenmiştir;

1- Yeniden yapılandırma için sunulan hangi kriter kullanılabilir? (Sadece öneri numarası)
2- Size göre araç tarafından yazılım yeniden yapılandırma için sunulan öneriler kullanışlı mıdır? (Evet veya Hayır)

1. soru için; geliştiricilere beş farklı öneri kategorisi sunulmuştur. Yukarıda açıklandığı gibi, bu kategoriler kod satırı (LOC), başka bir sınıfın yöntemine yapılan çağrıların sayısı, parametre sayısı ve WordNet'tir. Geliştiriciler, LOC'nin %60'ını, çağrı yönteminin %60'ını, parametre sayısının %70'ini ve WordNet ile ilgili önerilerin %40'ının faydalı göstergeler sağladığını açıkladı. Geliştiriciler tarafından verilen cevaplar Tablo 12'de gösterilmektedir.

Çizelge 4.5 Yazılım geliştiricileri değerlendirmelerinin sonuçları

| | Değerlendirilen Toplam Sınıf | Öneri Sayısı | Dev 1 | Dev 2 | Dev 3 | Dev 4 | Dev 5 |
|---|---|---|---|---|---|---|---|
| IM | 5 | 1 | 1 | 1 | 1 | 1 | 1 |
| LOC | 5 | 5 | 3 | 4 | 5 | 4 | 4 |
| CM | 5 | 7 | 5 | 5 | 4 | 6 | 3 |
| CP | 5 | 7 | 6 | 7 | 5 | 5 | 4 |
| WN | 5 | 12 | 5 | 6 | 7 | 5 | 5 |

(IM: Interface Metotları, LOC: Kod Satır Sayısı, CM: Çağırılan Metot Sayısı, CP: Parametre Sayısı, WN: WordNet)

Çizelge 4.5'de, “Geliştirici Tarafından Değerlendirilen Toplam Sınıf” (Total Class Evaluated) sütunu, geliştiricilere değerlendirme için verilen toplam sınıf sayısını tanımlar. “Öneri sayısı” (Number of Suggestions) sütunu, geliştiricilere sunulan sınıflar için aracımız tarafından sağlanan öneri sayısını ifade eder. Diğer sütunlarda, geliştiricilerin önerilerinin her birinin değerlendirme sonuçları belirtilmiştir. Örneğin, üretilen önerilerden biri parametre 7 olarak tanımlanmış ve olası bir SRP ihlali olarak uyarılmıştır.

Unutulmamalıdır ki, metodun kod satırı sınırı 30 olarak belirlenmiştir. İki geliştirici, 30 LOC'nin uzun bir yöntem olarak kabul edilmediğini iddia etti. Ayrıca, uzun yöntemlerden birini daha küçük olanlara ayırmak için de geliştiriciler arasında bir anlaşmazlık vardı. Geliştiricilerden biri yöntemin ikiye bölünmesi önerisini doğru bulurken, diğerleri gerekmeyebileceğini düşünüyordu. 2. soru için; tüm geliştiriciler bu yaklaşımı SRP fırsatları için yararlı bulduğunu söyledi.

### 4.2.10 Vaka çalışması 2

İki kurumsal projeye ek olarak, önerilen yaklaşım Visual Studio .net için geliştirilmiş olan bir uygulama çerçevesine (Uygulama Çerçevesi, 2019) uygulanmıştır. Başvuru çerçevesi açık kaynak olduğundan yapılan analiz sonuçlarına ilişkin uygulama çerçevesi yazılımı üzerinde de tüm ilgili detaylara ulaşılabilinmiştir. Uygulama çerçevesi işlem odaklı iş uygulaması olması nedeniyle, loglama, belleğe alma, doğrulama, işlem günlükleri, istisna yönetimi ve yetkilendirme gibi yardımcı sınıflar kullanılmıştır.

Bu kısımdaki çalışma kapsamında temel amaçlarımızdan biri, SOLID ilkelerinden biri olan tek sorumluluk ilkesinin nasıl uygulanabileceğini göstermektir. Bu nedenle, ilk vaka çalışmasında 2 kurumsal projede önerdiğimiz yöntemi denenmiş ve bu projelerden elde ettiğimiz sonuçları deneyimli yazılım geliştiriciler ile değerlendirmelerini istenmiştir.

Bu ikinci vaka çalışmasında, SOLID ilkelerine uygun olarak yazılmış olduğu iddia edilen bir projede önerdiğimiz yönteme göre geliştirdiğimiz bilgisayar uygulaması test edilmiştir. Böyle bir teste ihtiyaç duyulmuştur çünkü ilk vaka çalışmasında ele alınan kurumsal projelerin SOLID ilkelerine uygun olarak geliştirme yapıldığı düşünülmemiştir. Bu nedenle bu bölümde, nesne odaklı programlama tasarım ilkelerine uygun olarak yazılmış ve kurumsal düzeyde kullanılan açık kaynaklı bir uygulama çerçevesi kullanılmıştır. Buradaki amacımız çerçevenin SOLID uygunluğunu test etmek değil, bizim tarafımızdan önerilen metodolojinin çalışıp çalışmadığını test etmektir.

Çizelge 4.6 Metrik değerleri ile birlikte örnek öneri tablosu (uygulama çıktısı)

| **Sınıf Adı** | **Metot Adı** | **WN** | **LOC** | **NP** | **Öneri** |
|---|---|---|---|---|---|
| CacheExtension | Get<T> | 0 | 11 | 4 | The class name and method name are incompatible with each other.<br>Too many parameters. |
| ActiveDirectoryTool | DeleteUser | 0 | 25 | 5 | The class name and method name are incompatible with each other.<br>Too many parameters. |
| LanguageWordManager | GetByLanguage | 1 | 6 | 1 | No Suggestion |
| ManagerBase | ExecuteTransactionalOperation | 0,3333 | 17 | 1 | No Suggesiton |
| UserIndexViewModel | Users | 0 | 33 | 0 | The class name and method name are incompatible with each other.<br>Method size is too long. It can be divided into more methods. |
| OrderMap | OrderMap | 1 | 107 | 1 | Method size is too long. It can be divided into more methods. |

Uygulama çerçevesine daha detaylı olarak bakılacak olursa, 170 sınıf ve 630 metot içermektedir. Bu uygulama çerçevesi üzerinde geliştirilen araçla yapılan değerlendirmede (Çizelge 4.6), sözcüksel mesafe (WordNet) ile ilgili 37 gösterge ve uyum ile ilgili 53 gösterge ürettiği görülmüştür. Ek olarak, kod satırı hakkında 14 uyarı ve parametre sayısı hakkında 3 uyarı üretilmiştir. Elde edilen sonuçlar, sınıf adı, yöntem adı, wordnet değeri, LOC değeri, NOP ve bunlara ilişkin sağlanan önerileri içermektedir. Örneğin, Çizelge 4.6'da gösterildiği üzere "ManagerBase" sınıfındaki "ExecuteTransactionalOperation" yöntemi WordNet değeri 0,33'dür.

Bu vaka çalışmasından elde edilen sonuçlar uygulama çerçevesi içerisinden sınıflar veya metotlar seçilerek manuel olarak karşılaştırma yapılmıştır. Elde edilen sonuçlar göstermektedir ki önerilen yönteme göre geliştirilen uygulamanın doğru bir şekilde yazılım geliştiricilerine yapacakları yeniden yapılan işlemleri için çıktı üretmiş ve geliştiricilere geri bildirimde bulunmuştur.

# 5. SONUÇ

## 5.1 Bulgular ve Tartışma

### 5.1.1 SOLID-metrik çalışması

SOLID tasarım prensiplerinin amaçları yazılım içerisindeki bağımlılıkları azaltmak ve bakım yapılabilirliği artırmaktır. Çalışmanın bu kısmında yapılan bulgulardan birisi tasarım prensiplerinin uygulanmasının yazılıma zaman ve işgücü olarak ekstra maliyet verdiği görülsede yazılımın yayınlanması ve yazılım sonrasında destek, bakım yapılabilirlik aşamalarında bu maliyetin çok düşük olduğu görülmüştür. Öyleki, sonraki düşüklük ilk baştaki artan maliyetleri tolare edecek düzeydedir.

Uygulanan yöntem ile prensiplerin uygulanmasının yazılımın karmaşıklığını azalttığı ve bakım yapılabilirliğini artırdığı Visual Studio kod metrik ölçüm aracı ile ölçülmüştür. Bu ölçüm aracı ile ortaya çıkan bulgular ise kodun daha esnek bir yapıya kavuştuğu, kenetliliğin azalarak bağıntılığın arttığıdır. Aynı zamanda Visual studio içerisinde kullanılan kod metrik aracının ölçüm yapılan kodla ilgili bakım yapılabilirlik indeksi, kod karmaşıklığı, miras alma derinliği gibi özelliklerle ilgili yazılımcıya fikir edinebilmesi amacıyla ölçüm değerleri verdiği ancak bakım veya yeniden yapılandırma aşamasında kodla ilgili ne yapabileceği ile ilgili öneriler vermediği görülmüştür. Bu durum ise yazılımın sadece bakım yapılabilirlik oranlarını vermekte ama bu oranların iyileştirilebilmesi için yazılımcıya ne yapması gerektiği söylenmemekte, tamamen yazılımcının tecrübesine bırakılmaktadır.

### 5.1.2 SRP çalışması

Çalışmanın bu kısmında, yazılımın SOLID ve SOLID ilkelerinden biri olan tek sorumluluk ilkesine göre nasıl yeniden düzenlenebileceğiüzerinde durulmuştur. Literatürde yeniden düzenleme ile ilgili pek çok makale olmasına rağmen, yeniden düzenleme yöntemi hakkında nesne yönelimli tasarım ilkelerinden birine göre çok fazla çalışma yoktur. Sürdürülebilirliği artırmak ve kod karmaşıklığını azaltmak için tek

sorumluluk ilkesi'ne (SRP) göre otomatik olarak yeniden düzenleme önerileri verme yaklaşımı sunuldu. Bu yaklaşım, olası SRP ihlal kod bloklarını ve bir sınıfın ilişkisini yöntem ve sınıf adlandırma ile belirlemektedir. Her olası yöntem, SRP ihlal kriterlerini dikkate alan bir puanlama işlevi kullanılarak sıralanır. Bu işlev öncelikleri belirlemek için kullanılır.

Çalışmalarımızı değerlendirmek için bu çalışma kapsamında geliştirilen bir araç kullanılmıştır. Yaklaşımın sunulan öneriler ve yapılan çalışmalar göstermektedir ki, sunulan metodoloji, yazılımı yeniden yapılandırma için rastgele seçilen yöntemler üzerinde çalışmaya göre daha iyidir.. Geliştiriciler, araç tarafından yapılan önerileri takip edebilir. Bu, yaklaşımın önerilerinin genellikle en azından araştırma projeleri için faydalı olduğu anlamına gelir. Neredeyse tüm katılımcılar, bunları SRP'ye yeniden düzenleme yöntemi için kullanabileceğini iddia etti.

SRP kullanımıyla yazılım modülerliği, kod kullanımının artırılabileceği ve kenetlilik - bağıntı ilişkisinin en iyi seviyeye getirilebileceği vaka çalışmaları ile sunulmuştur. Ayrıca, çalışmada literatürde yeniden yapılandırma yöntemiyle ilgili yazılan makalelerin sonuçlarına paralel olarak SRP uygulandığında kodun karmaşıklığının en aza indirildiği ve tasarım kalitesinin arttığı görülmüştür.

Yeniden yapılandırmada, yönteminin yeniden yapılandırılmasının problemlerinden biri, yöntemin nasıl bölüneceğidir. Bu çalışma, SRP'nin uygun bir tasarımda kullanılmasının bu sorulara cevap verebildiğini göstermektedir. Çalışmamızın bir diğer sonucu da, yazılım geliştiricilere, SRP uyumluluğunun ileri sürdüğümüz yöntemle nasıl yapılabileceği hakkında verilen önerilerin kabul edilebilir düzeyde olmasıdır. Ayrıca anlamsal sözlük olarak kullanılabilecek WordNet kütüphanesi yöntemde en iyi tasarım için kullanılabilir ve sınıf isimlendirmesi çalışmamızın sonucudur.

Yaklaşımımızın sınıfların gereksiz arabirim yöntemlerini kullanma, çağrılma yöntemlerinin sayısı için işe yaradığı sonucuna varıyoruz. Ek olarak, SRP kodlarını yeniden düzenlemeye yönelik öneriler almak için kullanılabilir. Gelecekte, bu hipotezi doğrulamak için başka bir büyük vaka çalışması yapmak istiyoruz. Kritere farklı

göstergeler eklemeyi ve tüm nesne yönelimli tasarım ilkelerine genellemeyi planlıyoruz. WordNet ile çalışmayı sadece öneri vermekle kalmayıp, aynı zamanda uygun isimler belirlemeyi de planlıyoruz.

### 5.1.3 Geçerliliğe yönelik tehditler

Vaka çalışması araştırmasının uygulanması, geçerlilik için bazı olası tehditler ortaya çıkarmaktadır. Çalışmamızın sonuçlarıyla ilgili aşağıda açıklanan tehditlerin bertaraf edilmesi için çalışılmıştır.

İç geçerliliği sağlamak için alınan önlemlerden biri, değerlendirme çalışmasının birden fazla projeyle yürütülmesi ve proje kodundan sayısal değerlerin objektif olarak ölçülebilmesi için insan müdahalesi olmadan otomatik olarak elde edilmesidir. Çalışma yapılırken ve sonuçları rapor ederken mantıklı bir kanıt zinciri oluşturulmuştur. Elde edilen çıktılar görüşmeler yaparak geliştiriciler tarafından doğrulanmıştır. Son olarak, araştırmacının bulguların yorumlanmasındaki etkisi de iç geçerlilik için bir tehdit olarak değerlendirilebilir. Bunu azaltmak için, çalışmanın sonuçlarının yorumlanması mümkün olduğunca somut sayısal bulgular ile desteklenmiştir.

Yapısal geçerlilik, araştırma çalışmasındaki yapının iyi yapılandırılmış veya nesnel olarak tanımlanmış olup olmadığını dikkate alır. Bu sorunlardan kaçınmak için göstergeler birden fazla kaynaktan toplanmış ve ayrıca araştırma ve uygulama uzmanlar tarafından değerlendirilmiştir. Değerlendirme için hem görüşmeler hem de sonuçların gözlemleri kullanılmıştır. Ayrıca, göstergelere göre puanlamanın geçerliliği için, farklı göstergelerin puanları 0 ile 1 arasındaki değerlere göre sınıflandırılmış ve eşit olarak değerlendirilmiştir. Bu, en makul yaklaşım olarak düşünülmekte ve farklı göstergelerin katkı oranlarının önceki emprik bulgularının olmamasından kaynaklanmaktadır.

Dış geçerlilik, bulguların genelleştirilebilirliği endişeleriyle ilgilidir. Dış geçerliliği olan konulardan biri, değerlendirme çalışması diğer platformlarda değil Visual Studio .Net geliştirme ortamında yürütülmüş ve somut sonuçlar bu uygulama üzerinden alınmıştır. Ancak, gösterge belirleme süreci herhangi bir geliştirme ortamından bağımsız olarak tanımlanmıştır. Bu nedenle, sonuçların diğer ortamlar için genişletilebileceğine inanılmaktadır. Öte yandan, çalışmanın bir dereceye kadar tekrarlanabilirliğini

göstermek için değerlendirmeler yapılmıştır. Farklı projeler için deuygulama ile tutarlı sonuçlar elde edilebilir. Son olarak, otomatik bir prototip aracının geliştirilmesi, kullanılması ve paylaşılmasının benzer değerlendirmelerin diğer araştırmacılar tarafından yapılmasını sağlayacağına inanılmaktadır.

## 5.2 Değerlendirme

SOLID ilkesi bağımlılıkları azaltmayı ve bakım yapılabilirliği arttırmayı amaçlamaktadır. Yazılım geliştirme sırasında, her ilke için ekstra bir zaman ve ekstra çaba harcanması gerekir. Böylece kodun karmaşıklığı azaltılabilir ve kodun sürdürülebilirliği yani bakım yapılabilirliği artırılabilir. Bununla birlikte, esnek bir tasarım ve düşük bağımlılık ve daha yüksek bakım kolaylığı sağlarlar. Kod daha sağlam, daha kararlı ve daha iyi anlaşılabilir hale gelir. Kod metric değerleri kodun bakım yapılabilirliği ve karmaşıklığı hakkında bir fikir verebilir. Geliştirici, bakım görevine başlamadan veya yeniden bakım yapmadan önce kod ölçüm değerleri yardımıyla kod hakkında bir değerlendirme yapabilir.

Bu çalışma, SOLID tasarım ilkelerinin kodun sürdürülebilirliğini arttırdığını, genel olarak kodun karmaşıklığını ve bağımlılığı azalttığını, koda esneklik sağladığını göstermektedir. Tasarım ilkeleri, düşük bağımlılık ve daha güçlü uyum ile yazılımın geliştirilmesine yardımcı olur. Bununla birlikte, eğer bu prensipler belirli kurallar ile ve ölçümler ele alınmadan uygulanırsa, potansiyel olarak istenmeyen bazı sonuçlar ortaya çıkabilir. Bunlar miras alınan yazılım kodu derinliğinin istenmeyen şekilde artması, nispeten küçük ama kontrol dışında kalacak derecede sınıfların veya soyut sınıflar ile arayüzlerin istenmeyen şekilde çoğalmasıdır. Sonuç olarak, kod metrikleri hangi sınıfın ve hangi metodun üzerinde çalışılması gerektiği hakkında fikir verebilir.

## 5.3 Öneriler

Yazılım sistemleri sadece geliştirilip kendi haline bırakılan bir obje değil aynı zamanda devamlı olarak geliştirilmesi gereken yaşayan organizmalardır. Yazılımlarda bağımlılığın azaltılması bu canlı sistemlerin ileriye dönük olarak yeniden

yapılandırılmaları ve sürekliliklerinin sağlanması açısından önemlidir.

Tasarım ilkelerinin kullanılmaları da devamlı geliştirilmeleri gereken bu canlı sistemlerin uzun süre ayakta kamalarına olanak sağlayacak geliştirmelerin yazılımcı tarafından kolay bir şekilde yapılabilmesine imkân sağlayacaktır. Ayrıca, bu tasarım ilkelerinin de yardımıyla kod daha iyi yapılandırılabilir. Çalışmada öne sürülen teorinin gerçekleştrilmesi yazılan uygulama ile yapılırken, isimlendirilmelerin WordNet anlamsal kütüphanesi yardımıyla benzerliklerinin bulunmasında isimlendirmelerin yanlış, standart olmayan, kısaltmalar veya yarı İngilizce yarı Türkçe sözcükler içeren bir şekilde yapılmış olması karşılaşılan sorunlardan birisi idi. Bu yüzden isimlendirmelerin yazılım geliştirme ortamı içerisine gömülü uygulamalar vasıtası ile yazılımcıyı yönlendirmesi başka çalışmaların konusu olabilir,

Bu çalışmanın ikinci bölümünde sadece tek sorumluluk ilkesine göre detaylı çalışmalar yapılarak bu prensibe uygun olarak yazılım geliştirilebilmesi için bir metodoloji sunulmuştur. Daha ileriki çalışmalarda bütün prensipler için metdolojiler belirlenerek, geliştirilen öneri sunan yazılım içerisine bu metodolojiler otomatize edilebilir.

İleriye dönük olarak yapılabilecek çalışma, daha büyük bir projede tasarım prensiplerinin tamamının uygulanarak daha fazla yeniden yapılandırmanın yapılması olabilir. Sadece yeniden yapılandırma olarak düşünmeyip, yazılımın ilk baştan oluşturulmasında geliştiriciyi yönlendirecek, bilgiler verebilecek bir uygulama çerçevesi oluşturabilmek veya bunu yapabilecek paketleri geliştirme ortamına entegre etmek düzenli ve performanslı bir yazılım geliştirilmesinde büyük kolaylıklar sağlayacaktır. Bunun yanısıra yine bütün prensipleri içerecek ve yazılım geliştiricisine öneriler sunabilecek ölçüm programları ile tasarım prensiplerinin etkisini araştırmakta yeniden yapılandırma işlemlerinde fayda sağlayacağı düşünülmektedir.

## KAYNAKLAR


Al-Ahmad Walid, "A framework for conceptual modeling in OOP", Journal of the Franklin Institute, 2006

Al-Bashayreh Mahmood Ghaleb, Nor Laily Hashim, Ola Taiseer Khorma, "Context- Aware Mobile Patient Monitoring Framework Development", 2013 International Conference on Electronic Engineering and Computer Science, 2013

Ampatzoglou A, A. A. Tsintzira, E. M. Arvanitou, A. Chatzigeorgiou, I. Stamelos, A. Moga, R. Heb, O. Matei, N. Tsiridis, and D. Kehagias, "Benefits of Applying the Single Responsibility Principle to Repay Technical Debt: An Industrial Case Study", In Proceedings of 23rd Conference on the Evaluation and Assessment in Software Engineering (EASE' 19), Copenhagen, Denmark, 15-17 April 2019

Arevalo Gabriale, Stephane Ducasse, Silvia Gordillo, Oscar Nierstrasz , "Generating a catalog of unanticipated schemas in class hierarchies using Fomal Concept Analysis", Information and Software Technology, 2010

Asadi Morteza, Hassan Rashidi, "A Model for Object Oriented Software Maintainability Measurement", I.J. Intelligent Systems and Application (MECS), 2016

Bavota Gabriele, Andrea De Lucia, Rocco Oliveto , "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures", The Journal of Systems and Software, 2011

Bruntink Magiel, Arie van Deursen, "An empirical study into class testability", The Journal of System and Software, 2006

C Sharp Corner, https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/, 2019

Charalampidou Sofia , Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Antonios Gkortzis, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance", IEEE Transactions on Software Engineering 43(99) · October 2017

Charalampidou Sofia, Apostolos Ampatzoglou , Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou, "Identifying Extract Method Refactoring Opportunities Based on Functional Relevance" In IEEE Transactions On Software Engineering, vol. 43, No. 10, October 2017) (Danilo silva, Ricardo terra, Marco Tulio Valente, "Recommending Automated Extract Method Refactorings", 2014, ACM, Pages 146-156

Chatzigeorgiou Alexander, Nikolaos Tsantalis, Ignatios Deligiannis , “An empirical study on students ability to comprehend design patterns”, Computers & Education, 2008

Chen, J.-C., Huang, S.-J., “An empirical analysis of the impact of software development problem factors on software maintainability” System Software. 82, 981–992, 2009

DallalJehad Al, “Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics”, information and Software Technology, 2012

Deligiannis Ignatios, Martin Shepperd, Manos Roumeliotis, Ioannis Stamelos, “An empirical investigation of an object-oriented design heuristic for maintainability”, The journal of system and software, 2001

Fokaefs Marios, Nikolaos Tsantalis, Eleni Stroulia, Alexander Chatzigeorgiou, “Identification and application of Extract Class refactorings in object-oriented systems”, Journal of Systems and Software, 2012

Fontana, Francesca Arcelli, Mangiacavalli, Marco, Pochiero, Domenico, and Zanoni, Marco, “On experimenting refactoring tools to remove code smells” In Scientific Workshop Proceedings of the XP2015, XP ’15 workshops, ACM, pages 7:1–7:8, New York, NY, USA, 2015

Fowler M., “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 1999.

Galorath, D.D., “Software total ownership costs: development is only job one” Softw. Tech. News, 23–32, 2008

Glass Robert L. "Frequently Forgotten Fundamental Facts about Software Engineering", An Article in IEEE Software May/June 2001

Haas Roman, Benjamin Hummel, “Deriving Extract Method Refactoring Suggestions for Long Methods” In International Conference on Software Quality, SWQD 2016: Software Quality. The Future of Systems- and Software Development pp 144-155

Hegedűsa Péter, István Kádárb, Rudolf Ferenc, Tibor Gyimóthyb, “Empirical evaluation of software maintainability based on a manually validated refactoring dataset”, Information and Software Technology, 2017

Herbold Steffen, Jens Grabowski, Helmut Neukirchen, “Automated Refactoring Suggestions Using the Results of Code Analysis Tools” In First International Conference on Advances in System Testing and Validation Lifecycle 2009, DOI: 10.1109/VALID.2009.12

Hitz M., Montazeri B.: Measuring Coupling and Cohesion In Object-Oriented Systems. Proc. Int. Symposium on Applied Corporate Computing, Oct. 25-27, Monterrey, Mexico, 75-76, 197, 78-84., 1995

https://docs.microsoft.com/tr-tr/visualstudio/code-quality/code-metrics-values, 01.08.2017

ISO 9126-1 Software Engineering - Product Quality - Part 1: Quality Model, 2001.

ISO/IEC 25010:2011, http://www.iso.org/iso/catalogue_ detail.htm?csnumber=35733, 01.08.2017

Izadkhah Habib, Maryam Hooshyar, "Class Cohesion Metrics for Software Engineering: A Critical Review" In Computer Science Journal of Moldova, vol.25, no.1(73), 2017

Jung Ho-Won, Seung-Gweon Kim, Chang-Shin Chung, "Measuring Software Product Quality: A Survey of ISO/IEC 9126", IEEE Software, vol. 21, pp. 88-92, 2004

Kari Laitinen, "Natural naming in software development and maintenance", VTT Electronics / Embedded Software, Technical Research Centre of Finland, 1995

Kilidar Hiyam Al-, Karl Cox, Barbara Kitchenham, "The Use and Usefulness of the ISO/IEC 9126 Quality Standard", International Symposium on Empirical Software Engineering, 2005

Lievens David, William Harrison, "Abstraction over implementation structure with symmetrically encapsulated multimethods", Science of Computer Programming, 2013

Lippert Martin, Stephen Roock, "Refactoring in Large Software Projects: Performing Complex Restructurings Succesfully", Wiley, 2006

Martin Litz, Behzad Montazeri, "Measuring Coupling and Cohesion in Object Oriented Systems", In Proceedings of International Symposium on Applied Corporate Computing 1995, pages 25-27, Insitut für Angewandte Informatik und Sistemanalyse, University of Vienna

Martin Robert C., "The Clean Code", Clean Code: A Handbook of Agile Software Craftsmanship, ISBN: 9780132350884, 2008

Martin R. C. "Agile software development: principles, patterns and practices", Prentice Hall, 2003

Martin R. C., "Design Principles and Design Patterns", [Online]. Available: http://www.objectmentor.com, 2000

Metz Sandi (Duke University) , “SOLID Object-Oriented Design”, Talk given at the 2009 Gotham Ruby Conference in May, 2009. Online at http://www.youtube.com/watch?v=v-2yFMzxqwU

Meulen Meine J.P. van der, Miguel A. Revilla, “Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs”, 18th IEEE International Symposium on Software Reliability Engineering, 2007

Oman Paul and Jack Hagemeister. “Metrics for assessing a software system’s maintainability”. Proceedings International Conference on Software Mainatenance (ICSM), 1992, pp 337-344.

Ouni Ali, Marouane Kessentini, Mel ´O Cinneide, Houari Sahraoui, Kalyanmoy Deb, and Katsuro Inoue, “MORE: A Multi-objective Refactoring Recommendation Approach to Introducing Design Patterns and Fixing Code Smells” In Journal Of Software: Evolution And Process, volume 29, pagination -1843, 2017

Pedersen Ted, “WordNet::Similarity - Measuring the Relatedness of Concepts” In HLT-NAACL 2004, pages 38-41

Pinto, Gustavo H. and Kamei, Fernando. “What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow” In Proc. of the 6th Workshop on Refactoring Tools, pages 33–36. ACM, 2013

Silva Danilo, Nikolaos Tsantalis, Marco Tulio Valente, “Why We Refactor? Confessions of GitHub Contributors” In FSE’16, ACM, Pages 858-870, 2016

Silva Danilo, Ricardo terra, Marco Tulio Valente, “Recommending Automated Extract Method Refactorings”, 2014, ACM, Pages 146-156

Singh Harmeet, Syed Imtiyaz Hassan , “Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment”, International Journal of Scientific & Engineering Research, April-2015

Szoke Gabor, “Automating the Refactoring Process” In Acta Cybernetica 23, 2017, 715-735

Uygulama Çerçevesi, https://archive.codeplex.com/?p=devframework

www.microsoft.com, 01.08.2017

Xu Sihan, Aishwarya Sivaraman, Siau-Cheng Khoo, Jing Xu, “GEMS: An Extract Method Refactoring Recommender” In IEEE 28th International Symposium on Software Reliability Engineering 2017, DOI: 10.1109/ISSRE.2017.35

Zafeiris Vassilis E., Sotiris H. Poulias, N.A. Diamantidis, E.A. Giakoumakis, “Automated refactoring of super-class method invocations to theTemplate Method design pattern”, Information and Software Technology, 2016

Zotos Kostas, “Object-oriented design principles in mathematics”, Applied Mathematics and Computation, 2006

## ÖZGEÇMİŞ

Adı Soyadı : Osman TURAN

Doğum Yeri : Nevşehir

Doğum Tarihi : 1978

Medeni Hali : Evli

Yabancı Dili : İngilizce

**Eğitim Durumu (Kurum ve Yıl)**

Lise : Nevşehir Lisesi - 1995

Lisans : Dokuz Eylül Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği - 2003

Yüksek Lisans: Ankara Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Bölümü (Eylül 2013 – Ağustos 2019)

**Çalıştığı Kurum/Kurumlar ve Yıl**

Maliye Bakanlığı / 2003-2008

Gümrük ve Ticaret Bakanlığı / 2008-2013

Kamu Gözetimi Kurumu / 2013 -

**Yayınlar**

Hakemli Dergiler

Turan O, Tanrıöver Ö, SOLID İlkelerinin Microsoft VS Kod Metriğine Etkisinin Deneysel Olarak Değerlendirilmesi, Academic Journal of Information Technology - DOI:10.5824/1309-1581.2018.4.001