Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi

Object Oriented Software Metrics and Software Quality

Ural, Erdemir
TÜBİTAK - UEKAE[1], Kocaeli
ural@uekae.tubitak.gov.tr

Umut, Tekin
TÜBİTAK - UEKAE[1], Kocaeli
umuttekin@uekae.tubitak.gov.tr

Feza, Buzluca
Bilgisayar Mühendisliği Bölümü
İTÜ[2], İstanbul
buzluca@itu.edu.tr

Özet

Nesneye dayalı yazılım metrikleri ve yazılımda kalite kavramları, yazılım dünyası için son yıllarda üzerinde en çok çalışılan konular haline gelmiştir. Çalışma kapsamında bu kavramlar üzerine geniş bir literatür taraması yapılarak metriklerin tanımlamaları ve kaliteyle ilişkileri açıklanmıştır. Buna paralel olarak kaliteyi arttırıcı kimi yöntemler ve prensipler anlatılarak, kaliteyi arttırma çalışmalarında kullanılabilecek bazı yardımcı araçlar tanıtılmıştır. Tartışma bölümünde ise mevcut çalışmalar değerlendirilerek, eksik noktalar ortaya koyulmuş ve ileriki geliştirmeler hakkında çeşitli çalışma önerileri verilmiştir.

Abstract

Recently, object-oriented software metrics and quality concepts became some of the most focused topics in the software world. Definition of software metrics and their relevance with software quality were described by performing a thorough literature search within the scope of these studies. In addition to this, some methods and principles which can improve the quality were explained and some auxiliary tools that can be used to improve software quality were also introduced. In the discussion section, missing issues were pointed out by evaluating available studies and some recommendations about future improvements were provided.

1. Giriş

Günümüzde bilgisayar donanımları düşük maliyet ve hata oranları ile üretilebilirken, yazılımların maliyetleri ve hata oranları oldukça yüksek seviyelere ulaşmıştır. Yazılımların boyutlarının büyümesi aynı zamanda bakım masraflarının ve geliştirme zamanının artmasına sebep olmuştur. Bugün birçok yazılım projesi başarısızlıkla sonuçlanabilmektedir. Tüm bunların somut bir örneği, Amerikan ordusunun yazılım projeleri üzerine yaptığı bir araştırmada da görülmüştür. Bu araştırmaya göre yapılan yazılım projelerinin;

i. %47'si kullanılmamakta
ii. %29'u müşteri tarafından kabul edilmemekte
iii. %19'u başladıktan sonra iptal edilmekte ya da büyük ölçüde değiştirilerek yeniden başlatılmakta
iv. %3'ü kimi değişikliklerden sonra kullanılmakta
v. %2'si ancak teslim alındığı gibi kullanılmakta olduğu görülmüştür.

Aynı zamanda 2002 yılında NIST tarafından yapılan bir araştırma sonucunda yazılım hatalarının Amerikan ekonomisine yıllık maliyeti 59 Milyar $ olarak tahmin edilmiştir ki o yıllarda satılan yazılımların toplam değeri 180 Milyar $ civarındaydı.

Bilişim sektöründeki yazılım projelerinin maliyetlerinin bu denli büyümesiyle birlikte, yazılımda kalite kavramı günümüzde üzerinde en çok çalışılan konulardan biri haline gelmiştir. Crosby, genel olarak kaliteyi "isterlere uygunluk" olarak tanımlamıştır [1]. Yazılım dünyasında kalite, kavram olarak herkesin hissedebildiği ancak neticede kimsenin tam olarak tanımlayamadığı soyut ve öznel bir olgu olarak karşımıza çıkmaktadır. Bir disiplin içerisinde kalitenin tam olarak tanımlanabilmesi için gelişmiş ölçme araçlarına ve sağlıklı karşılaştırmalar için tanımlı referans noktalarına gerek vardır. Ancak yazılım sektöründe bu iki olgunun da henüz çok gelişmemiş olduğunu görmekteyiz. Her ne kadar yazılım dünyası, yazılım kalitesinde, başlangıç evresinde olsa da, bu konudaki küçük çalışmaların bile yazılım üreticilerine kazançlarının çok büyük olduğu görülmektedir. Bu açıdan kalite kavramına endüstri tarafından verilen önem ve üzerinde yapılan akademik çalışmalar her geçen gün artmaktadır.

Yazılım kalitesini temelde iki farklı bakış açışıyla ele almak mümkündür. Bunları müşteri gözünden kalite ve yazılım üreticisi gözünden kalite olarak isimlendirebiliriz. Müşteriler genel olarak satın aldıkları yazılımların kolay kullanılabilir, hatasız, tüm isteklerini

[1] Ulusal Elektronik ve Kriptoloji Araştırma Enstitüsü
[2] İstanbul Teknik Üniversitesi

karşılayan ve yeterli performansa sahip olmasını isterler. Buna karşılık yazılım üreticileri ise geliştirme ve bakım maliyetlerinin düşük ve üretilen yazılımın parçalarının bir sonraki projelerinde de kullanılabilir olmasını isterler. Yazılım kalitesini çeşitli sınıflar halinde kategorize etmek mümkündür. ISO 9126 yazılım ürünlerinin kalitesi anlatan ve sınıflandıran uluslar arası bir standarttır [2]. ISO 9126' ya göre kalite sınıfları ve alt sınıfları şunlardır:

i. **İşlevsellik:** Yazılımın ihtiyaçları karşılama becerisi olarak tanımlanmaktadır. Uygunluk, doğruluk, birlikte çalışabilirlik ve güvenlik konuları bu kategori altında incelenmektedir.
ii. **Güvenilirlik:** Yazılımın düzgün çalışma halini muhafaza edebilme becerisi olarak tanımlanmaktadır. Olgunluk, hata toleransı ve geri kurtarma konuları bu kategori altında incelenmektedir.
iii. **Kullanılabilirlik:** Yazılımın kullanım kolaylığı sağlayan yetenekleri olarak tanımlanmaktadır. Öğrenebilme, anlaşılabilirlik, işletilebilirlik ve kullanıcı etkileşimi konuları bu kategori altında incelenmektedir.
iv. **Verimlilik:** Yazılımın ihtiyaç duyulan ölçüde yeterli performansla çalışabilme becerisi olarak tanımlanmaktadır. Zaman ve kaynak kullanımı konuları bu kategori altında incelenmektedir.
v. **Bakılabilirlik:** Yazılımın değişiklik veya düzeltme isteklerine adaptasyon yeteneği olarak tanımlanmaktadır. Değiştirilebilirlik, test edilebilirlik, analiz edilebilirlik ve bağışıklılık konuları bu kategori altında incelenmektedir.
vi. **Taşınabilirlik:** Yazılımın farklı çalışma ortamlarına uyum sağlayabilme yeteneği olarak tanımlanmaktadır. Adaptasyon yeteneği, yüklenebilirlik özellikleri**,** ortam değiştirme imkânı ve diğer yazılımlarla uyum konuları bu kategori altında incelenmektedir.

ISO 9126'da aynı zamanda bu sınıfların hepsi için sınıfa özgü mevcut standartlara, kanunlara vs.. uyumluluk konusu ortak bir özellik olarak ele alınmaktadır.

ISO 9126 standardı incelendiğinde kalite kavramının daha çok ürün bazlı olarak müşteri gözünden elle alındığı görülmektedir. Oysa yazılım geliştiren firmalar açısından bakıldığında kalite kavramına geliştirilen modüllerin yeniden kullanılabilirliği gibi diğer önemli kalite sınıflarını da eklememiz gerekecektir.

Çalışma kapsamında daha kaliteli yazılımların üretilebilmesi amacıyla genel bir literatür taraması yapılarak öncelikle yazılımda toplam kalite yönetimi ve ölçme kavramları açıklanmıştır. 3. bölümde yazılım kalitesini belirleyen iç özellikler, 4. bölümde bu özellikleri ölçmeye yarayan temel metrik kümeleri karşılaştırılarak anlatılmıştır. 5. bölümde yazılım metriklerini ölçmek için faydalanılabilecek kimi yardımcı araçlardan örnekler tanıtılmıştır. 6. bölümde tasarım prensipleri ve tasarım kalıpları ile ölçülerin ilişkisi açıklanmıştır. Son olarak metriklerin yazılım geliştirmede kullanımıyla ilgili değerlendirmeler yapılarak, yeni öneriler getirilmiştir.

2. Yazılımda Toplam Kalite Yönetimi ve Ölçme Kavramı

Toplam kalite yönetiminin [3], üretim bantlarında kullanılmaya başlamasıyla birlikte, endüstriyel ürünlerde kalitenin sistematik bir biçimde arttığı çeşitli tecrübelerle saptanmıştır. Günümüzde birçok firma toplam kalite sistematiklerini kullanmaktadır. Toplam kalite yönetimi süreçlerin etkin yönetimini amaçlamaktadır. Süreç kısaca, girdileri çıktılara dönüştüren işlemler kümesi olarak tanımlanabilir. Toplam kalite disiplinindeki temel esas, kaliteli ürünlerin ancak iyi tanımlanmış olgun süreçler sonucunda çıktığıdır. Mevcut süreçlerin sürekli geliştirilmesi ve iyileştirilmesi ile daha kaliteli ürünlerin ortaya çıkartılması hedeflenmektedir. Günümüzde toplam kalite prensiplerini temel alan SPICE, ISO ve CMMI gibi uluslar arası standartlar yazılım şirketleri tarafından süreç çalışmalarında sıklıkla kullanılmaktadır.

Sürekli gelişme ve iyileşmeye prensibinin temelinde sürecin planlaması, denetlemesi, çıktılarının ve performansının ölçülmesi en nihayetinde ise sürecin değerlendirmesi vardır. Tanımlı bir süreç boyunca birçok ara ürün veya ürün olarak nitelendirebileceğimiz çıktılar oluşturulmaktadır. Bir yazılım geliştirme sürecini ele alacak olursak bu çıktılar örneğin; müşteri isterleri, tasarım dokümanları, yazılım kodu, test sonuçları ve bunun gibi diğer çıktılar olacaktır.

Süreç esnasında toplanan ölçümler, sürecin değerlendirilebilmesi ve kontrol edilebilmesi için kullanılan en önemli girdilerdir. Ölçme ise kavram olarak varlıkların özelliklerinin sayısallaştırılması olarak tanımlanabilir. Bu noktada karşımıza bir yazılımda neleri sayılaştırabileceğimiz soruları çıkmaktadır. Bir sonraki bölümde bu kapsamda detaylı olarak yazılım ölçüleri ve sınıfları açıklanmıştır.

3. Yazılımın İç Özellikleri:

Üretilen bir yazılımın müşterilerine bakan dış özellikleri, yazılımın iç özelliklerinin bir yansımasıdır. Nesneye dayalı yazılımlarda bağımlılık, uyumluluk ve karmaşıklık yazılımın en önemli iç özellikleridir. En genel anlamda bağımlılık, yazılım parçaları arasındaki karşılıklı bağımlılığın derecesini, uyumluluk yazılım parçalarının kendi yaptıkları işlerdeki tutarlılığın derecesini, karmaşıklık ise yazılımın içyapısının kavranmasındaki zorluğun derecesini belirtir. Kaliteli yazılımların uyumluluğunun yüksek, karmaşıklığının ve

bağımlılığının düşük olduğu yaygın olarak kabul görmüş bir olgudur [4].

Nesneye dayalı yazılım geliştirme yöntemlerinin en büyük avantajı bu yöntemlerin gerçek dünyaya olan yakınlığıdır, çünkü gerçek dünya da nesnelerden oluşmaktadır. Gerçek dünyadaki nesnelerde olduğu gibi nesneye dayalı yazılımlarda da bağımlılık, uyumluluk ve karmaşıklık gibi kavramlar tanımlanabilmektedir.

Matematiksel olarak, nesneye dayalı bir yazılımda, bir nesne X = (x, p(x)) şeklinde gösterilebilir. Burada x nesnenin tanımlayıcısı (Örneğin adı), p(x), x' in sonlu sayıdaki özelliklerini belirtir. Nesnenin nitelik değişkenleri (Instance Variables) ve metotları nesnenin özelliklerini oluştururlar. Mx metotların kümesi, Ix nitelik değişkenlerin kümesi olmak üzere, nesnenin özellikleri P(x) = {Mx} U {Ix} şeklinde gösterilebilir. Bundan sonraki bölümlerde kavramların matematiksel anlatımında bu gösterim kullanılacaktır.

3.1. Bağımlılık (Coupling):

Ontolojik terminolojide, bağımlılık iki nesneden en az birinin diğerine etki etmesi olarak tanımlanır. A nesnesinin B nesnesine etki etmesi, B'nin kronolojik durumlarının sırasının A tarafından değiştirebilmesidir [5]. Bu tanıma göre Ma'in Mb ya da Ib üzerinde herhangi bir eylemi, aynı şekilde Mb'nin Ma ya da Ia üzerinde eylemi, bu iki nesne arasında bağımlılığı oluşturur [6].

Aynı sınıfın nesneleri aynı özellikleri taşıdığından, A sınıfının B sınıfına bağımlılığı aşağıdaki durumlarda oluşur:

i. A sınıfının içinde B sınıfı cinsinden bir üye (referans, işaretçi ya da nesne) vardır.
ii. A sınıfının nesneleri B sınıfının nesnelerinin metotlarını çağırıyordur.
iii. A sınıfının bir metodu parametre olarak B sınıfı tipinden veriler (referans, işaretçi ya da nesne) alıyordur ya da geri döndürüyordur.
iv. A sınıfının bir metodu B tipinden bir yerel değişkene sahiptir.
v. A sınıfı, B sınıfının bir alt sınıfıdır [7].

Bir sınıfın bağımlılığı; kendi işleri için başka sınıfları ne kadar kullandığı, başka sınıflar hakkında ne kadar bilgi içerdiği ile ilgilidir. Kaliteli yazılımlarda nesneler arası bağımlılığın mümkün olduğunca düşük olması tercih edilir.(low coupling) Sınıflar arası bağımlılık arttıkça:

i. Bir sınıftaki değişim diğer sınıfları da etkiler.(maintability)
ii. Sınıfları birbirlerinden ayrı olarak anlamak zordur.(understandibility)
iii. Sınıfları tekrar kullanmak zordur.(reusability)

3.2. Uyumluluk (Cohesion):

Bunge benzerliği σ(), iki varlık arasındaki özellik kümesinin kesişimi olarak tanımlamaktadır [5]. σ(X,Y) = p(x) ∩ p(y). Benzerliğin bu genel tanımından yola çıkarak metotlar arasındaki benzerliğin derecesi, bu iki metot tarafından kullanılan nitelik değişkenleri (instance variables) kümesinin kesişimi olarak tanımlanmaktadır [6]. Elbette metodun kullandığı nitelik değişkenleri metodun özellikleri değildir ama nesnenin metotları, kullandığı nitelik değişkenlerine (instance variables) oldukça bağlıdır.

σ (M1. M2), M1 ve M2 metotlarının benzerliği, {Ii} = Mi metodu tarafından kullanılan nitelik değişkenleri olmak üzere σ(M1,M2) = {I1} ∩ {I2} 'dir. Örneğin {I1} = { a , b , c , d , e } ve {I2} = {a,b,e} için σ (M1,M2)= {a, b, e } olur.

Metotların benzerlik derecesi, hem geleneksel yazılım mühendisliğindeki uyumluluk kavramı (ilgili şeyleri bir arada tutmak) ile hem de kapsülleme (encapsulation) (nesne sınıfındaki veriler ve metotları bir arada paketlemek) ile ilişkilidir. Metotların benzerlik derecesi, nesne sınıfının uyumluluğunun başlıca göstergesi olarak görülebilir. Uyumluluk bir sınıftaki metot ve niteliklerin birbiriyle ilgili olmasının ölçüsüdür. Bir sınıftaki metot parametrelerindeki ve nitelik tiplerindeki güçlü örtüşme iyi uyumluluğun göstergesidir. Eğer bir sınıf aynı nitelik değişkenleri kümesi üzerinde farklı işlemler yapan farklı metotlara sahipse, uyumlu bir sınıftır. Eğer bir sınıf birbiri ile ilgili olmayan işler yapıyorsa, birbiriyle ilgili olmayan nitelik değişkenleri barındırıyorsa veya çok fazla iş yapıyorsa sınıfın uyumluluğu düşüktür. Düşük uyumluluk şu sorunlara yol açar:

i. Sınıfın anlaşılması zordur.
ii. Sınıfın bakımını yapmak zordur.
iii. Sınıfı tekrar kullanmak zordur.
iv. Sınıf değişikliklerden çok etkilenir.

3.3. Karmaşıklık (Complexity):

Karmaşıklık bir sınıfların iç ve dış yapısını, sınıflar arası ilişkileri kavramadaki zorluğun derecesidir. "Bir nesnenin karmaşıklığı: bileşiminin çokluğudur" [5] Buna göre karmaşık bir nesne çok özelliğe sahip olur. Bu tanıma göre (x, p (x)) nesnesinin karmaşıklığı özellik kümesinin kardinalitesi yani |p(x)|, olur [6].

4. Yazılım Metrik Kümeleri

Yazılım metrikleri çeşitli açılardan sınıflandırılabilir. Bilgilerin toplanma zamanı açısından statik ve dinamik iki tür metrik sınıfı vardır. Statik metrikler yazılım çalıştırılmadan yazılımın yapısıyla ilgili belgelerden (Örneğin: kaynak kodu) elde edilebilirler. Dinamik metrikler ise yazılım çalışması sırasında toplanan verilere dayanır. Metrikler ölçmede kullanılan bilgilere göre de sınıflandırılabilir. Bazı metrikler metotların

sadece parametre erişimlerine bakarken, bazıları metotlardan tüm veri erişimlerini dikkate alırlar. Metrikler ürettikleri verinin tipi ve aralığına göre de sınıflandırabilir. Bazı metrikler [0,+∞), bazıları sonlu bir aralıkta gerçel ya da tam sayı değerler üretirken, bazıları sınırlı bir aralıkta gerçel sayılar üretebilir. Özellikle [0–1] arası değer üreten metrikler karşılaştırılma açısından daha kullanışlıdır.

Bu çalışmada, nesneye dayalı yazılım metrikleri incelenmiştir. Geleneksel işleve dayalı yazılım metriklerinden (Örneğin: Kaynak kodu satır sayısı, Açıklama Yüzdesi, McCabe Çevrimsel Karmaşıklığı [8] (Cyclomatic Complexity) sınıfların metotlarına uygulanarak kullanılır. Bu bölümde literatürde yaygın olarak kabul görmüş 3 nesneye dayalı yazılım metrik kümesi; Chidamber & Kemerer metrik kümesi, MOOD metrik kümesi ve QMOOD metrik kümesi anlatılmıştır.

4.1. Chidamber & Kemerer (CK) Metrik Kümesi

Chidamber ve Kemerer 6 temel metrik tanımlamışlardır. Bu metriklerin tanımları ve kısaca hangi özelliklerle ilişkili oldukları aşağıda açıklanmıştır [6].

Sınıfın Ağırlıklı Metot Sayısı - Weighted Methods per Class (WMC): Bir sınıfın tüm metotlarının karmaşıklığının toplamıdır. Tüm metotların karmaşıklığı 1 kabul edilirse, sınıfın metot sayısı olur.

Sınıfın metotlarının sayısı ve metotlarının karmaşıklığı, Sınıfın geliştirilmesine ve bakımına ne kadar zaman harcanacağı hakkında fikir verebilir. Metot sayısı çok olan taban sınıflar, çocuk düğümlerde daha çok etki bırakırlar. Çünkü tanımlanan tüm metotlar türetilen sınıflarda da yer alacaktır. Sınıf sayısı çok olan sınıfların uygulamaya özgü olma ihtimali yüksektir. Bu nedenle tekrar kullanılabilirliği düşürürler.

Kalıtım Ağacının Derinliği - Depth of Inheritance Tree (DIT): Sınıfın, kalıtım ağacının köküne uzaklığıdır. Hiçbir sınıftan türetilmemiş sınıflar için 0'dır. Eğer çoklu kalıtım varsa, en uzak köke olan uzaklık kabul edilir. Kalıtım hiyerarşisinde daha derinde olan sınıflar, daha çok metot türettiklerinden davranışlarını tahmin etmek daha zordur. Derin kalıtım ağaçları daha çok tasarım karmaşıklığı oluşturur.

Alt Sınıf Sayısı - Number of Children (NOC): Sınıftan doğrudan türetilmiş alt sınıfların sayısıdır. Eğer bir sınıf çok fazla alt sınıfa sahipse, bu durum kalıtımın yanlış kullanıldığının bir göstergesi olabilir. Bir sınıfın alt sınıf sayısı tasarımdaki potansiyel etkisi hakkında fikir verir. Çok alt sınıfı olan sınıfların metotları daha çok test etmeyi gerektirdiğinden bu metrik sınıfı test etmek için harcanacak bütçe hakkında bilgi verir.

Nesne Sınıfları Arasındaki Bağımlılık - Coupling Between Object Classes (CBO): Sınıfın bağımlı olduğu sınıf sayısıdır. Bu metrikte bağımlılık bir sınıf içinde nitelik (attribute) ya da metotlar diğer sınıfta kullanılıyorsa ve sınıflar arasında kalıtım yoksa iki sınıf arasında bağımlılık olduğu kabul edilmiştir. Sınıflar arasındaki aşırı bağımlılık modüler tasarıma zarar verir ve tekrar kullanılabilirliği azaltır. Bir sınıf ne kadar bağımsızsa başka uygulamalarda o kadar kolaylıkla yeniden kullanılabilir. Bağımlılıktaki artış, değişime duyarlılığı da arttıracağından, yazılımın bakımı daha zordur. Bağımlılık aynı zamanda tasarımın farklı parçalarının ne kadar karmaşık test edileceği hakkında fikir verir. Bağımlılık fazla ise testlerin daha özenli yapılması gerektiğinden test maliyetini arttıracaktır.

Sınıfın Tetiklediği Metot Sayısı - Response For a Class (RFC): Verilen sınıftan bir nesnenin metotları çağrıldığında, bu nesnenin tetikleyebileceği tüm metotların sayısıdır. Bu metrik sınıfın test maliyeti hakkında da fikir verir. Bir mesajın çok sayıda metodun çağrılmasını tetiklemesi, sınıfın testinin ve hata ayıklamasının zorlaşması demektir. Bir sınıftan fazla sayıda metodun çağrılması, sınıfın karmaşıklığının yüksek olduğunun işaretçisidir.

Metotların Uyumluğu - Lack of Cohesion in Methods (LCOM): Bu metriğin birden çok tanımı vardır. Chidamber ve Kemerer ilk olarak şu tanımı yapmışlardır. C1 sınıfının M1,M2, … ,Mn metotları olduğunu ve {Ii} kümesinin de Mi metodunda kullanılan nitelik değişkenleri kümesi olduğunu kabul edelim. Bu durumda LCOM bu n kümenin kesişiminden oluşan ayrık kümelerin sayısıdır (LCOM1[9]). Daha sonra bu tanım yenilenerek şu tanım getirilmiştir. P hiçbir ortak nitelik değişkeni(I) paylaşmayan metot çiftlerinin kümesi, Q en az bir ortak nitelik değişkeni paylaşan çiftlerin kümesi olsun. Bu durumda LCOM= {|P|>|Q| ise P|-|Q| ; aksi durumda 0 } olur (LCOM2[6]). Literatürde farklı LCOM metrik tanımları da mevcuttur: LCOM3[10], LCOM4[11]. Sınıfın uyumluluğunun düşük olması, sınıfın 2 veya daha fazla alt parçaya bölünmesi gerektiğini gösterir. Düşük uyumluluk karmaşıklığı artırır, bu nedenle geliştirme aşamasında hata yapılma ihtimali yükselir. Ayrıca metotlar arasındaki ilişkisizliklerin ölçüsü sınıfların tasarımındaki kusurların belirlenmesinde de yardımcı olabilir.

4.2 Brito e Abreu MOOD Metrik Kümesi

MOOD metrik kümesi [12], nesneye dayalı yöntemin temel yapısal mekanizmalarına dayanır. Bunlar kapsülleme (MHF AHF), kalıtım (MIF, AIF), çok şekillilik (PF), mesaj aktarımı (CF) olarak sıralanır.

Metot Gizleme Faktörü - Method Hiding Factor (MHF): Sistemde tanımlı tüm sınıflardaki görünür (çağrılabilir) metotların, tüm metotlara oranıdır. Bu metrikle sınıf tanımının görünürlüğü ölçülür Burada kalıtım ile gelen metotlar hesaba katılmamaktadır.

Nitelik Gizleme Faktörü - Attribute Hiding Factor (AHF): Sistemde tanımlı tüm sınıflardaki görünür (erişilebilir) niteliklerin, tüm niteliklere oranıdır. Bu metrikle sınıf tanımının görünürlüğü

ölçülür Burada kalıtım ile gelen metotlar hesaba katılmamaktadır.

Metot Türetim Faktörü - Method Inheritance Factor (MIF): Sistemde tanımlı tüm sınıflardaki kalıtım ile gelen metot sayısının, tüm metotların (kalıtımla gelenler dâhil) sayısına oranıdır.

Nitelik Türetim Faktörü - Attribute Inheritance Factor (AI): Sistemde tanımlı tüm sınıflardaki kalıtım ile gelen niteliklerin, tüm niteliklere (kalıtımla gelenler dahil) oranıdır.

Çok Şekillilik Faktörü - Polymorphism Factor (PF): Bir C sınıfı için sistemdeki farklı çok şekilli durumların, maksimum olası çok şekilli durumlara oranıdır. Bilindiği gibi türetilen nitelikler ve metotlar alt sınıflarda yeniden tanımlanarak, ana sınıftaki metotları örtebilir (override).

Bağımlılık Faktörü - Coupling Factor (CF): Sistemde sınıflar arasında var olan bağımlılık sayısının, oluşabilecek maksimum bağımlılık sayısına oranıdır. Burada kalıtımla oluşan bağımlılıklar hesaba katılmamaktadır.

4.3. Bansiya ve Davis QMOOD Metrik Kümesi

QMOOD metrikleri [13], Bansiya ve Davis tarafından yazılımın toplam kalite endeksi hesaplanması için 4 seviyeden oluşan hiyerarşik bir model içinde tanımlanmıştır. En alt seviyede sınıf, metot gibi nesneye dayalı yazılım bileşenleri vardır. 3. seviyede QMOOD metrikleri vardır. QMOOD metriklerinden, karmaşıklık, uyumluluk gibi bir üst seviyedeki yazılım özellikleri hesaplanır. Bu yazılım özelliklerinden de anlaşılabilirlik, esneklik, tekrar kullanılabilirlik gibi en üst seviyedeki kalite nitelikleri hesaplanır. Son olarak kalite niteliklerinden toplam kalite endeksi hesaplanır. QMOOD çalışmasında tanımlanan seviyeler ve bağlar Şekil 1‘ de gösterilmiştir. Bu bölümde 11 QMOOD metriğinden 8 tanesi aşağıda verilmiştir:

Ortalama Ata Sayısı- Avarage Number of Ancestors (ANA): Tüm sınıfların DIT (Kalıtım Ağacının Derinliği - Depth of Inheritance Tree) değerlerinin ortalamasıdır. Metrik değeri yazılımda soyutlamanın kullanımını gösterir.

Metotlar Arası Uyumluluk - Cohesion Among Methods (CAM): Metotların imzaları arasındaki, benzerliğin ölçüsüdür. Tam tanımı Bansiya’ nın metrik kümesi çalışmasında yer almamaktadır. Literatürde metotların imzalarına bakarak uyumluluk ölçen farklı metrik tanımları bulunmaktadır. Bunlar temelde metotların parametre kullanım matrisi üzerinde çalışırlar ve imzalardaki uyumluluğu tam uyumlu olmaya yakınlıkla kıyaslayarak ölçerler. Metotlar ve tüm parametreler numaralandırılır, i. metodun imzasında, j. parametre kullanılıyorsa, matriste Mij 1 aksi halde 0 olur. CAMC[14] parametre matrisindeki 1 sayısına bakarken, NHD[15] matris satırları arasındaki Hamming uzaklıklarına bakar. SNHD[16], ise NHD metriğinin olası en küçük ve büyük değerine göre ölçeklenmiş halidir.

Sınıf Arayüz Boyutu - Class Interface Size (CIS): Sınıfın açık (public) metotlarının sayısıdır ve yazılımın mesajlaşma özelliği hakkında fikir verir.

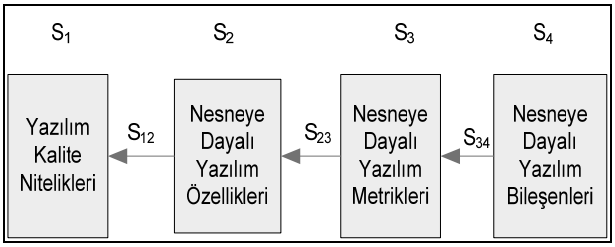
Veri Erişim Metriği - Data Access Metric (DAM): Sınıfın özel(private) ve korumalı(protected) niteliklerinin tüm niteliklere oranıdır. Bu metrik yazılımın kapsülleme özelliğini gösterir.

Doğrudan Sınıf Bağımlılığı- Direct Class Coupling (DCC): Bir sınıfı parametre olarak kabul eden sınıfların sayısı ile bu sınıfı nitelik değişkeni olarak barındıran sınıfların sayısının toplamıdır. Sınıfın bağımlılığı bu metriğe bakarak anlaşılabilir.

Kümeleme Ölçüsü - Measure Of Aggregation (MOA): Kullanıcı tarafından tanımlanmış sınıf bildirimlerinin, tanımlı temel sistem veri tiplerine (int, char, double vs…) oranıdır.

İşlevsel Soyutlama Ölçüsü - Measure of Functional Abstraction (MFA): Sistemde tanımlı tüm sınıflardaki türetilmiş metot sayısının, tüm metot sayısına (türetilmiş metotlar dâhil) oranıdır. Yazılım kalıtım özelliği bu metrikle belirlenir.

Metot Sayısı - Number of Methods (NOM): Sınıfta tanımlı metot sayısı tüm metotlar bir birim kabul edilirse WMC metriği ile aynı değeri taşır. Sınıfın metot sayısı sınıfın karmaşıklığının bir göstergesidir.



Şekil 1: Seviyeler ve Seviyeler Arası Bağlar

5. Yardımcı Araçlar

Bu bölümde özellikle nesneye dayalı yazılımların kalite çalışmaları kapsamında kullanılabilecek bazı popüler açık kaynak kodlu ve ticari araçlar tanıtılmıştır. Bu araçlar yazılım kalitesinin arttırılmasına yönelik çalışmalarda ihtiyaç duyulan kimi ölçümleri otomatik ve hızlı biçimde yaparak kalite analizi çalışmalarını oldukça kolaylaştırabilirler.

Bu kapsamda tanıtacağımız ilk program olan **FindBugs** [17], Maryland üniversitesi tarafından geliştirilmiş açık kaynaklı bir statik kod analiz aracıdır. Yazılımın mevcut Java kodu üzerinde çeşitli analizler yaparak, yaygın yazılım hatalarını ve tasarım kusurlarını otomatik olarak kısa sürede bulabilmektedir. Findbugs; java, netbeans, jboss gibi günümüzde popüler olan birçok programın geliştirme aşamalarında da etkin şekilde kullanılmaktadır. Örneğin Netbeans

uygulamasının 6.0-8m versiyonunun analizi için kullanılan program 189 adet hatayı tespit edebilmiştir.

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods (avg/max per type)	11	2,2	2,926	8
Number of Attributes (avg/max per type)	9	1,8	2,713	7
Number of Children (avg/max per type)	3	0,6	1,2	3
Number of Classes (avg/max per packageFragment)	5	5	0	5
Method Lines of Code (avg/max per method)	206	6,438	10,105	40
Number of Methods (avg/max per type)	32	6,4	6,681	19
src	32	6,4	6,681	19
(default package)	32	6,4	6,681	19
SortItem.java	19	19	0	19
SortAlgorithm.java	7	7	0	7
QSortAlgorithm.java	4	4	0	4
BidirBubbleSortAlgorithm.java	1	1	0	1
BubbleSortAlgorithm.java	1	1	0	1
Nested Block Depth (avg/max per method)		1,688	0,982	4
Depth of Inheritance Tree (avg/max per type)		2,4	1,356	5
Number of Packages	1			
McCabe Cyclomatic Complexity (avg/max per method)		2,438	2,703	12
src		2,438	2,703	12
(default package)		2,438	2,703	12
QSortAlgorithm.java		3,75	4,763	12
BidirBubbleSortAlgorithm.java		10	0	10
SortItem.java		1,947	1,669	8
BubbleSortAlgorithm.java		6	0	6
SortAlgorithm.java		1,429	0,495	2
Total Lines of Code	300			
Instability (avg/max per packageFragment)		1	0	1
Number of Parameters (avg/max per method)		0,812	0,845	3
Lack of Cohesion of Methods (avg/max per type)		0,255	0,324	0,776
Normalized Distance (avg/max per packageFragment)		0	0	0
Specialization Index (avg/max per type)		1,321	0,89	2,105
Weighted methods per Class (avg/max per type)	78	15,6	11,074	37

Şekil 2: SortingDemo Programının Metrikleri

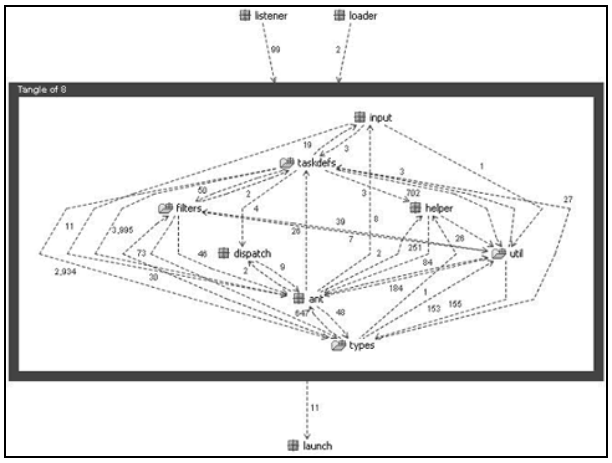
Metrics [18], Eclipse projesine bir eklenti olarak geliştirilen açık kaynak kodlu bir başka programdır. Eclipse geliştirme ortamındaki Java projelerine, tümleşik biçimde çalışabilen program, yaygın kullanılan bir çok yazılım metriğini otomatik olarak ölçerek geliştiriciye raporlamaktadır. Statik fonksiyon sayısı, türeme derinliği, bağımlılık, fonksiyon karmaşıklığı, soyutlama adedi gibi metrikler bunlardan yalnızca birkaçıdır. Metrics programı aynı zaman yazılımdaki bağımlılıkları grafiksel olarak gösterme yeteneğine de sahiptir. Bu sayede geliştiricilerinin yazılımdaki bağımlıkları görsel olarak daha iyi analiz edebilmeleri mümkün olmaktadır. Şekil 2' de JDK kütüphanesinin örnek programlarından SortingDemo[19]'nun Metrics programı yardımıyla ölçülmüş bazı metrikleri verilmiştir. Burada yazılımdaki modüller (paket/sınıf ya da metotlar) için LCOM, DIT, ANA, WMC, NOC gibi metrik değerleri görülmektedir.

PMD [20] ise başka bir statik kod analiz aracıdır. Yine Java kodları üzerinde çalışan bu program mevcut kod üzerinde otomatik analizler yaparak olası kusurları; kullanılmayan, tekrarlanmış ve gereksiz kod parçalarını hızlıca bulmaktadır.

Coverlipse [21] gerçekleme yani yazılım kodu ile gereksinimler ve test senaryolarının arasındaki örtüşme ilişkilerini inceleyen açık kaynak kodlu bir Eclipse eklentisidir. Program yazılım geliştirmenin bu üç temel aşaması arasındaki örtüşme ilişkilerini analiz ederek aradaki boşlukları ortaya çıkartmaktadır. Dolayısıyla unutulmuş bir gereksinime veya test edilmemiş bir yazılım parçasına yer vermemeye yardımcı olmaktadır.

CheckStyle programı [22] yazılımın yapısından çok formatı ile ilgilenen bir başka açık kaynak kodlu yazılım aracıdır. Mevcut Java kodlarının üzerinde format analizi yaparak geliştiricilerin kod yazım standartlarına uyumlu çalışmalarına yardımcı olmaktadır. Kurumun kendi yazım standartlarını da oluşturabileceği bu yazılımda, aynı zamanda genel kabul görmüş kimi uluslararası yazım standartları da mevcuttur.

SDMetrics [23], yukarda anlatılan programlardan farklı olarak kod üzerinde değil de UML tasarım dokümanları üzerinde çeşitli görsel ve sayısal analizler yapabilen ticari bir programdır. Program, tasarım aşamasından yani geliştirme aşaması başlamadan yazılımın tasarımını analiz ederek bağımlılık ve karmaşıklığa dair birçok uygunsuzluğu ortaya çıkartabilmektedir. Bu aşamada yapılacak erken tespitlerin maliyet açısından da kazanımları büyük olabilmektedir.



Şekil 3: Örnek Bağımlılık Grafiği

Coverity [24], Java'nın yanı C++ ve C program kodları üzerinden çalışabilen ticari ve oldukça kapsamlı bir kod analiz aracıdır. Analiz sonuçlarının yanı sıra bazı otomatik düzeltmeler de yapabilen program yazılım dünyası tarafından çok büyük maliyetli ve kapsamlı projelerde kullanılmaktadır. Bu programın diğer bir önemli özelliği de, statik analizin yanı sıra çalışma esnasında dinamik analizleri de yapabilmesidir. Şekil 3'de Coverity programı ile üretilmiş örnek bir bağımlılık grafiği verilmiştir.

6.Metriklerin Yazılım Kalitesini Arttırma Yöntemleriyle İlişkisi

Bu bölümde yazılım kalitesini iyileştirmeye yönelik yazılım dünyasınca kabul görmüş, sezgisel tecrübeler anlatılmıştır. Yazılım metriklerinin kalitesiz kod göstergelerini tespit etmede yardımcı olması beklenir. Aynı şekilde tasarım prensipleri ve tasarım kalıpları yerinde kullanıldığında yazılımın kalitesinin arttığı metriklerle de doğrulanmalıdır.

6.1. Düşük Kaliteli Kod Göstergeleri: (Bed Smells):

Kalitesiz kod göstergeleri kavramı ilk olarak, Martin Fowler tarafından var olan kodların iyileştirilmesi için ortaya koyulmuştur [25]. Yazılım yaşam döngüsü boyunca sürekli değişim halindedir. Bu değişiklikler bazen dağınık kodları toparlama bazen de uygulamanın yapısını değiştirme şeklinde olabilir. Yazılım geliştiriciler, kod üzerinde değişiklik yaparken kalitesizlik göstergelerini fark ederek gerekli düzeltmeleri yapmalıdırlar. Bu göstergelerin büyük bir çoğunluğu metrik değerlerini olumsuz yönde değiştirirken, bazıları metriklerle analiz sırasında gözden kaçabilir. Yazılım metriklerinden düşük kaliteli kod göstergelerini tespit etmekle ilgili başlangıç düzeyinde bir çalışma [26]'te verilmiştir. Aşağıda düşük kaliteli kod göstergelerinden bazıları verilmiştir [27].

Tekrar eden kod parçaları: Aynı kod parçasının bir den çok yerde gözükmesidir. Bu kod parçalarının metot haline getirilmesi tavsiye edilir. Ancak tanımlanan metriklerle bu tasarım kusurunu fark edecek bir yöntem bulunmamaktadır.

Uzun Metot: Nesneye dayalı programlarda metotların kısa olması yeğlenir. Ağırlıklı metot sayısı metriklerinde, ağırlık ölçüsü olarak metodun kod uzunluğu alınırsa, bu kusur fark edilebilir.

Büyük Sınıf: Çok fazla iş yapan, çok sayıda üye nitelik değişkeni veya kod tekrarı bulunduran sınıflar büyük sınıflardır. Bu sınıfların yönetimi zordur. Uyumluluk metrik değeri düşük, bağımlılık metrik değeri yüksek olan sınıfların büyük sınıf olma ihtimali yüksektir.

Uzun parametre listesi: Uzun parametre listelerinin anlaşılması, kullanımı zordur ve ilerde yeni verilere erişmek isteneceğinden sürekli değişiklik gerektirir. Uzun parametre listesi olan sınıfların metotlar arası uyumluluk (CAM) metrikleri genelde düşük çıkar.

Farklı Değişiklikler (Divergent Change): Farklı tipte değişiklikler için kodda tek bir sınıf üzerinde değişiklik yapılmasıdır. Bu tasarım kusurunu tanımlanan metriklerle tespit etmek oldukça zordur. Ancak bu özelliğin bulunduğu sınıfın uyumluluk metriğinin düşük olması beklenir.

Parçacık Tesiri (Shotgun Surgery): Bir değişiklik yapılması gerektiğinde kodda birçok sınıfta küçük değişiklik yapılmasının gerekmesidir. Böyle bir durumda önemli bir değişikliğin atlanma ihtimali yüksektir. Parçacık tesiri tespit edilen sınıflarda karmaşıklık ve bağımlılık metriğinin yüksek olması beklenir.

Veri Sınıfı: Sadece veriler ve bu verilerin değerlerini okumak ve yazmak için gerekli metotları barındıran ve başka hiçbir şey yapmayan sınıflardır. Tanımlanan metriklerle bu tür kusurları bulmak çok kolay gözükmemektedir. Ancak sınıfın açık ve gizli metotlarının oranı bu konuda yardımcı olabilir.

Reddedilen Miras: Türetildiği sınıfın sadece küçük bir parçasını kullanan sınıflar bu durumu oluştururlar. Bu kalıtım hiyerarşisinin yanlış olduğunun bir göstergesidir. Sınıf için Metot Türetim Faktörü ve Nitelik Türetim Faktörü metrikleri hesaplandığında bu değerlerin beklenenin altında olduğu görülecektir.

6.2. Tasarım Kalıpları (Design Patterns):

Tasarım kalıpları ilk olarak mimar Christopher Alexander tarafından kaliteli mimari yapılardaki ortak özelliklerin sorgulanması ile ortaya çıkmıştır .[28] C. Alexander yaptığı araştırmalar sonucunda beğenilen (kaliteli) yapılarda benzer problemlerin çözümünde benzer çözüm yollarına başvurulduğunu belirlemiş ve bu benzerliklere tasarım kalıpları adını vermiştir. 1980'lerde, Kent Beck bu çalışmalardan esinlenerek yazılım kalıplarını ortaya atmıştır [29]. Mimarlar gibi yazılım geliştiriciler de, tecrübeleriyle, bir çok ortak problemin çözümünde uygulanabilecek, prensipler ve deneyimler (kalıplar) oluşturmuşlardır [30], [31]. Bu konudaki ilk önemli çalışma dört yazar tarafından hazırlanan bir kitap olmuştur [32]. Bu yazarlara dörtlü çete (Gang of Four) adı takılmış ve ortaya koydukları kalıplarda GoF kalıpları olarak adlandırılmıştır. GoF kalıpları üç gruptan oluşmaktadır:

Yaratımsal Kalıplar: Bu kalıplar nesne yaratma sorumluluğunun sınıflar arasında etkin paylaştırılmasıyla ilgilidir. Bu kalıplar da kendi aralarında iki türe ayrılabilir, birinci türde nesne yaratma sürecinde kalıtım etkin olarak kullanılır, ikinci türde ise delegasyon kullanılır. Tekil Nesne (Singleton), Soyut Fabrika (Abstract Factory), İnşacı (Builder) kalıpları bu gruptandır ve nesne yaratmanın getireceği ek karmaşıklıkları ve bağımlılıkları en aza indirirken, sınıfların uyumluluğunu yüksek tutmayı amaçlamaktadırlar.

Yapısal Kalıplar: Bu gruptaki kalıplar, sınıf ve nesnelerin kompozisyonuyla ilgilidir. Genel olarak arayüzler oluşturmak için kalıtımdan faydalanırlar ve yeni işlevler kazandırmak için nesnelerin yapılarını oluşturacak uygun yöntemler önerirler. Adaptör, Köprü, Ön yüz (Facade) bu gruptaki bazı kalıplardandır. Örneğin Köprü kalıbı soyutlamayı gerçeklemeden ayrı tutar, böylece ikisi bağımsız olarak değiştirilebilir. Adaptör kalıbı ile farklı arayüze sahip sınıflara ortak bir arayüz oluşturulur. Böylece bu sınıflardan hizmet alan istemci sınıfların bağımlılığı azaltılır ve karmaşıklık etkin bir şekilde yönetilmiş olur.

Davranışsal Kalıplar: Bu kalıplar özellikle nesneler arasındaki iletişimle ilgilidir. Gözlemci (Obsever), Strateji bu gruptan sıkça başvurular kalıplardır. Bu kalıplar aracılığıyla, yazılım ihtiyaçlara göre kolayca genişletilebilir, hataların yerleri daha çabuk tespit edilebilir.

Yazılımda kalite kavramının sorgulanmasıyla ilgili ilk çalışmalar tasarım kalıplarına dayanmaktadır. Bu açıdan yüksek kaliteli nesneye dayalı yazılımların oluşturulması büyük ölçüde tasarım kalıplarının yerinde

kullanılmasına bağlıdır. Tasarım kalıplarının yazılım kalitesini yükselttiği uzun zamandır sezgisel olarak bilinmekle beraber, metriklerle sayısal olarak doğrulanmasıyla ilgili bir çalışma henüz bulunmamaktadır. Hatta var olan metrikler kullanıldığında kalıpların bazı metriklerde olumsuz etkilere yol açtığı da görülebilir. Örneğin Adaptör kalıbının kullanılması, hizmet alacak sınıfların çok sayıda ayrı sınıfa bağımlığını önlerken, adaptör sınıfından türetilmiş sınıfların kalıtımdan kaynaklanan bağımlılıkları ortaya çıkacağından sonuçta toplam bağımlılık sayısı artacaktır. Diğer basit bir örnek Gözlemci kalıbında gözükmektedir. Abone sınıfın arayüzünü gerçekleyen çok sayıda alt sınıf olacağından Alt Sınıf Sayısı (NOC) metriği daha yüksek çıkacaktır. Ancak bu iki kalıpla ileride değişebilecek, genişletilebilecek noktalar belirlenerek etrafları örülmekte ve değişikliklerin kolay yapılarak başka modüllerin bu değişikliklerden mümkün olduğunca az etkilenmesi sağlanmaktadır.

7. Değerlendirme (Tartışma)

Metrikler yazılım kalitesinin belirlenmesi ve iyileştirilmesi çalışmalarında etkin biçimde kullanılmaktadır. Bu çalışmalar sonucunda aşırı bağımlı, karmaşık ve hataya eğilimli modüllerin belirlenmesi gibi önemli bilgiler elde edilebilmektedir. Bu bilgiler yazılım kalitesinin iyileştirilmesinde, daha sonra hangi kısımların öncelikli olarak test edileceğine karar verilmesinde, bakım için gereken bütçe ve zaman analizlerinde kullanılabilmektedir. Ancak tüm bu çalışmaların başarımı büyük ölçüde doğru metriklerin tanımlanmasına, bu metriklerin doğru biçimde ölçülmesine ve sonunda doğru yorumlanmasına bağlıdır.

Literatürde, tanımlanan metriklerin doğrulanması için iki ayrı yönteme başvurulduğu görülmektedir. İlk olarak metrik tanımları özellik tabanlı metrik ölçümüne [33] göre doğrulanır. Örneğin karmaşıklık metrikleri negatif değer üretmemeli, boş küme için Null değer üretebilmeli, yeni bir ilişki eklenmesi sistemin karmaşıklığını azaltmamalı, iki ayrık modülün birleşiminin karmaşıklığı bu iki modülün karmaşıklıklarının toplamına eşit olmalıdır gibi... İkinci yöntemde ise örnek uygulamalar üzerinde, tanımlanan metriğin ürettiği değerlerin, aynı amaçla kullanılan diğer metriklerin ürettiği değerlerle korelâsyonuna bakılır. Bu iki yöntem metrik doğrulamada olmazsa olmaz gerekleri belirtmekle beraber, yeterli değildir. Metriklerin doğrulanması ve değerlendirmesi için yazılım dünyasında daha çok çalışmalara ve geri beslemelere gerek vardır. Özellikle metrik ve kalite ilişkisi, üzerinde derin araştırmalar yapılaması gereken bir konudur. Bansiya'nın ortaya koyduğu metrikler üzerinden doğrudan sihirli bir formülle yazılım kalite özelliklerini çıkarmak [13], bu konudaki ilk çalışma olması açısından önemli olmakla birlikte, akademisyenler tarafından daha uygun ve gelişmiş yöntemlerin geliştirilmesi gerektiği açıktır.

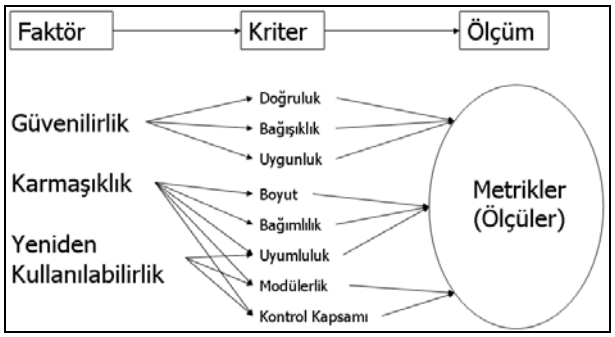
Öte yandan literatürde bulunan bazı metrik tanımlarının çok net olmadığı ve farklı yorumlara açık olduğu görülmektedir. Ürettikleri değerin nasıl yorumlanması gerektiği konusunda da anlaşmaya varılmamıştır. Aynı zamanda bu değerlerin yazılımın büyüklüğüne göre ölçeklenebilir olmaması pratik kullanımlarını zorlaştırmaktadır. Örneğin CBO ve RFC bu türden metriklerdir.

Metrikler üzerine yapılmış birçok çalışma olmasına rağmen halen yeni metrik tanımlarına ihtiyaç olduğu görülmektedir. Çünkü mevcut metriklerin ihtiyaçları karşılamada yeterli değildir. Örneğin kodun okunurluğu yazılım bakımı için oldukça önemli bir unsurdur. Çünkü kodlar yazıldıklarından çok okunmaktadırlar. Ancak bu özelliği doğrudan ölçen bir metrik tanımı bulunmamaktadır. Her ne kadar koddaki açıklama sayısı, kod yazım standartlarına uyum gibi ölçümler yapılabilse de klasik saymaya ve formata dayalı metriklerin yanında anlam analizine dayalı yaklaşımlardan da faydalanılmalıdır.

Yazılım geliştirme maliyetinin büyük kısmını oluşturan bakım maliyetleri her geçen gün artmaktadır. Değişiklik istekleri ve ortaya çıkan hatalar bakım maliyetini oluşturan en önemli etkenlerdir. Bu maliyetler ancak kaliteli yazılımların üretilmesiyle düşürülebilir. Metrikler pek hala hata çıkabilecek kısımların erken tespitinde kullanılabilmektedir. Literatürde yazılımdaki hata meyilli modüllerin metrikler yardımıyla tespit edilmesiyle ilgili çalışmalar mevcuttur [34], [35]. Çalışmalar henüz yeterli düzeyde olmasa da endüstri tarafından kullanılmaya başlanması hem mevcut çalışmaların iyileştirmesini sağlayacak hem de elde edilen deneyimlerle yeni yaklaşımların önü açılacaktır.

Diğer bir önemli konu da kalite kapsamında metriklerin tek başına değil de bir arada değerlendirmenin gerekliliğidir. Verilen kararlar bir metriği iyileştirirken diğer bir metriği daha kötü hale getirebilir. Dolayısıyla yazılım geliştiricilerinin hedefleri doğrultusunda bir denge sağlaması ve birden fazla metriği bir arada kullanması gerekir. Örneğin web teknolojileri geliştiren bir yazılım şirketi için pazara girme hızı önemliyken, gerçek zamanlı gömülü sistemler geliştiren bir yazılım şirketi için güvenilirlik beklentisi öne çıkabilir. Günümüz kalite çalışmalarının bu durumu göz önüne alacak şekilde organize edilmeli ve yeni tanımlanacak metrik kümelerinin metrikler arasındaki ilişkileri de içermesi gerekliliği görülmektedir. Bu gerçeği baz alan üst düzey bir modelin örnek bir parçası Şekil 4' de verilmiştir. Bu çalışmada faktörler, kriterler ve metrikler arasındaki ilişkiler incelenmiştir. Ancak bu ilişkilerin

derecelendirilmesi ve birbirlerine etkileri üzerine genel kabul görmüş bir çalışma yapılmamıştır.



Şekil 4: Faktör Kriter Ölçüm Modeli

Yazılım metriklerini otomatik olarak ölçen araçların çeşitlendirilmesi ve bu araçların yeteneklerinin arttırılması gerekmektedir. Araçların sadece rakam göstermemesi, ayrıca metriklerin görselleştirmesi de analizcilerin işini kolaylaştıracaktır. Metrik araçları yeni metriklerin kolayca uyarlanabilmesini desteklemeli ve geliştiricilerin kendi özel metrik görüntülerini ayarlamasına izin vermelidir. Ayrıca metrik araçlarının nesneye dayalı programlama dilinden bağımsız olması da faydalı olacaktır, çünkü metrik tanımları büyük ölçüde programlama dilinden bağımsızdır. Ancak piyasada bulunan metrik araçlarının birçoğu Java programlama diline özgüdür. C++ ve diğer programlama dillerini destekleyen araçlar çok daha azdır.

Metrik ölçümlerinin yazılım geliştirme süreçlerine daha sıkı olarak bağlanması gereklidir. Her aşamada belirlenen kurumsal amaçlar doğrultusunda hangi metriklerden faydalanılması gerektiği kurum içinde standartlaştırılmalıdır. Metriklerin değişimlerinin geliştirme süreci esnasında her aşamada izlenmesi ve değerlendirilmesi de yazılımın gidişatını saptamak açısından faydalı olacaktır. Aynı kurum içinde zamanla çeşitli projelere ilişkin metriklerden oluşan ortak bilgi bankalarının tanımlanması ileride geliştirilecek projelerin analizlerinde çeşitli faydalar sağlayacaktır.

8. Sonuç

Hızla artan yazılım maliyetlerini azaltmak, ancak geliştirilen yazılımların daha kaliteli üretilmesiyle veya mevcut yazılımların kalitelerinin arttırılmasıyla sağlanabilir. Yazılım kalitesinin istenen düzeyde arttırılabilmesi için her şeyden önce mevcut kalitenin doğru biçimde ölçebilmesi gerekmektedir. Toplam kalite çalışmalarının da felsefesinin temelinde yer alan, Tom DeMarco' nun ifade ettiği "Ölçemediğimiz bir şeyi kontrol edemeyiz ve iyileştiremeyiz" [36] düşüncesi yazılım için de geçerlidir.

Bu çalışma kapsamında yazılım kalitesi ile yazılım metriklerinin temel kavramları kısaca açıklanmış. Metrik tanımlamalarının ve mevcut araçların şu anki durumu değerlendirilerek, gelecekte varılması gereken noktalar tartışılmıştır. Çalışma, yazılımda kalite kavramı ile ilgilen endüstri için yön verici ve bilgilendirici olacak şekilde hazırlanmıştır. Yazılım kalitesini arttırılması ve metriklerin kullanması konusunda hem akademiye, hem de endüstriye büyük görevler düşmektedir.

9. Kaynaklar

[1] Crosby, P. Quality Is Free: The Art of Making Quality Certain. McGraw- Hill, New York, 1979.

[2] ISO/IEC 9126-1: Information Technology - Software Product Quality - Part 1: Quality Model. ISO/IEC JTC1/SC7/WG6 (1999)

[3] Ishikawa, Kaoru. What is Total Quality Control? The Japanese Way. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1985.

[4] G. Booch, Object Oriented Design with Applications. Redwood City, CA: Benjamin/Cummings, 1991.p547

[5] M. Bunge, Treatise on Basic Philosophy: Ontology I : The Furniture of the World. Boston: Riedel, 1977. p871

[6] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476- 493, June 1994.

[7] Buzluca F., Yazılım Modelleme ve Tasarımı Ders Notları, 2002-2007,

[8] McCabe,T.J. A complexity measure, IEEE Transactions on Software Engineering 2(4):308-320, 1976.

[9] Chidamber, S.R, and C.F. Kemerer. Towards a metric suite for object-oriented design, Proceedings : OOPSLA '91, Phoenix, AZ, July 1991, pp. 197-211.

[10] Li, W., S. Henry, D. Kafura, and R. Schulman. Measuring Object-oriented design. Journal of Object-Oriented Programming, Vol. 8, No. 4, July 1995, pp. 48-55.

[11] Hitz, M., and B. Montazeri. Chidamber and Kemerer's metric suite : A Measurement Theory Perspective, IEEE Transactions on Software Engineering, Vol. 4, April 1996, pp. 267-271.

[12] F. Brito e Abreu, G. Pereira, and P. Soursa, "Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems," Proc. Euromicro Conf. Software Maintenance and Reeng., pp. 13-22, 2000.

[13] J. Bansiya and C. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," IEEE Trans. Software Eng., vol. 28, no. 1, pp. 4-17, Jan. 2002.

[14] Bansiya, J., Etzkorn, L., Davis, C., and Li, W.. 1999. A class cohesion metric for object-oriented designs. J. Object-Oriented Program. 11, 8, 47–52

[15] Counsell, S., Mendes, E., Swift, S., and Tucker, A. 2002. Evaluation of an object-oriented cohesion metric through Hamming distances. Tech. Rep. BBKCS-02-10,

[16] Steve Counsell and Stephen Swift, The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design, ACM Transactions on Software Engineering and Methodology, Vol. 15, No. 2, April 2006.

[17] FindBugs. http://findbugs.sourceforge.net/index.html

[18] Metrics. http://metrics.sourceforge.net/

[19] The Sorting Algorithm Demo. http://java.sun.com /applets/jdk/1.4/demo/applets/SortDemo/example1.html
[20] PMD. http://pmd.sourceforge.net/
[21] Coverlipse. http://coverlipse.sourceforge.net/index.php
[22] Checksytle. http://checkstyle.sourceforge.net/
[23] SDMetrics. http://www.sdmetrics.com/
[24] Coverity. http://www.coverity.com/
[25] M. Fowler and K. Beck, "Bad Smells in Code," in Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000, pp. 75-88.
[26] Mazeiar Salehie, Shimin Li, Ladan Tahvildari, A, "Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws", Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06).
[27] M. Fowler, Refactoring: Improving the Design of Existing Code, Canada: Addison-Wesley, 2000.
[28] Alexander, C., Ishikawa, S., and Silverstein, M.. A Pattern Language — Towns-Build-ing-Construction. Oxford University Press.1997.
[29] Beck, K., and Cunningham, W. 1987. Using Pattern Languages for Object-Oriented Programs. Tektronix Technical Report No. CR-87-43.
[30] Beck, K. 1994. Patterns and Software Development. Dr. Dobbs Journal. Fob 1994.
[31] A. J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.
[32] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns. Reading, MA.:Addison-Wesley
[33] L. C. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," IEEE Transactions on Software Engineering, vol. 22 (1), 1996.
[34] R. Ferenc, I. Siket, and T. Gyimothy, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897-910, Oct. 2005.
[35] Hector M Olague, Letha H Etzkorn, Sampson Gholston, Stephen Quattlebaum, Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes Software Engineering, IEEE Transactions on, Vol. 33, No. 6. (2007), pp. 402-419.
[36] DeMarco, Tom., Controlling Software Projects: Management, Measurement and Estimation. ISBN 0-13-171711-1 Prentice Hall, 1982