BİTİRME PROJESİ

Pragmatik Yaklaşımla Kod Kalitesi İyileştirme Üzerine Literatür Taraması

Bilal Kaya - 2204040136

Proje Danışmanı – Dürdane Yılmaz

İçindekiler

Anahtar Terimler ve Kısaltmalar ii

Giriş 1

1. Pragmatik Programcı İlkeleri ve Teknik Borç 2

DRY (Don't Repeat Yourself): 2

Ortogonalite (Orthogonality): 3

YAGNI (You Aren't Gonna Need It): 3

Tracer Bullets: 3

2. Kod Kokuları (Code Smells) 4

3. Yeniden Yapılandırma (Refactoring) Teknikleri 5

4. Statik Kod Analizi ve Araçları 7

4.1. PMD 7

4.2. Checkstyle 8

4.4. SonarQube 9

5. Kod Kokusu Tespit ve Otomatik Refactoring Araçları 10

5.1. JDeodorant 10

5.2. Diğer Refactoring Öneri Sistemleri 11

5.3. Otomatik Düzeltme ve Program Onarma Yaklaşımları 12

6. Performans Analizi ve Profiling Araçları 13

6.1. Java Flight Recorder (JFR) ve JDK Mission Control 13

6.2. Java VisualVM 14

6.3. JProfiler ve YourKit 15

6.4. Diğer Araçlar ve Teknikler 16

7. Yazılım Kalite Metrikleri ve Ölçütleri 17

7.1. Chidamber & Kemerer (CK) Metrikleri 17

- WMC (Weighted Methods per Class) 17
- DIT (Depth of Inheritance Tree) 17
- NOC (Number of Children) 17

7.2. Çevrimsel Karmaşıklık (Cyclomatic Complexity) 18

7.3. Halstead Metrikleri 19

7.4. Maintainability Index (Bakım Endeksi) 21

Sonuç 23

Kaynakça 25

Anahtar Terimler ve Kısaltmalar

- **Kod Kalitesi (Code Quality):** Bir yazılımın okunabilir, sürdürülebilir, anlaşılır ve genişletilebilir olup olmadığını belirten genel bir ölçüttür. Yüksek kod kalitesi, hatasız geliştirme ve kolay bakım açısından kritiktir.
- **Teknik Borç (Technical Debt):** Geliştirme sırasında alınan kısa vadeli kararların, ileride daha fazla bakım ve yeniden yapılandırma gerektirmesiyle ortaya çıkan yazılımsal yük. Genellikle hızlı geliştirme uğruna yapılan tavizler sonucu oluşur.
- **Kod Kokusu (Code Smell)**: Hatalı olmayan ancak kodun gelecekte bakımını zorlaştırabilecek yapılar. Örneğin, çok uzun metodlar veya tekrar eden kod parçaları birer "kokudur".
- **Refactoring (Yeniden Yapılandırma):** Bir yazılımın dış davranışını değiştirmeden, iç yapısını daha okunabilir, sürdürülebilir veya verimli hâle getirme sürecidir.
- **Pragmatic Programming (Pragmatik Programcılık):** "The Pragmatic Programmer" kitabının sunduğu, yazılım geliştirme sürecine esneklik, sorumluluk alma, sürekli öğrenme ve faydaya odaklanma yaklaşımıdır. DRY, Orthogonality, YAGNI gibi prensipleri içerir.
- **DRY (Don't Repeat Yourself):** Aynı bilgi ya da işlevin yazılım içinde birden fazla yerde tekrar edilmemesi gerektiğini vurgulayan ilkedir.
- **YAGNI (You Aren't Gonna Need It):** Şu anda ihtiyaç duyulmayan işlevlerin kodlanmaması gerektiğini savunur.
- **Orthogonality (Ortogonalite):** Yazılım bileşenlerinin birbirinden bağımsız çalışabilirliğini ifade eder. Bir bileşende yapılan değişiklik diğerlerini etkilememelidir.
- **Tracer Bullets (İzleyici Mermiler):** Sistemin nihai haline benzer şekilde işleyen küçük, çalışır alt parçalarla erken test ve geri bildirim alma yaklaşımıdır.
- **Statik Kod Analizi:** Yazılımın çalıştırılmadan önce kaynak kod üzerinden hatalar, stil bozuklukları ve potansiyel risklerin analiz edilmesidir. Otomasyon için kritik öneme sahiptir.
- **Profiling (Çalışma Zamanı Profillemesi)**: Yazılımın gerçek çalışması sırasında hangi fonksiyonların ne kadar süre çalıştığını, belleği nasıl kullandığını analiz etme işlemidir.
- **Java Flight Recorder (JFR)**: JVM üzerinde çalışan Java uygulamalarının çalışma zamanındaki davranışlarını, özellikle sıcak metotları, CPU kullanımı ve tahsisatları izlemek için kullanılan düşük yük profilleme aracıdır.
- **Java Microbenchmark Harness (JMH):** Java kodlarının mikro düzeyde karşılaştırmalı performans testlerini yapmaya yarayan resmi benchmark çerçevesidir.
- **Checkstyle:** Java kodlarının stil kurallarına uygunluğunu denetleyen statik analiz aracıdır. Kurumsal yazılım standartlarına uyum açısından önemlidir.
- **PMD:** Java ve diğer dillerde kod kokularını, potansiyel bug'ları ve kötü kodlama alışkanlıklarını tespit eden statik analiz aracıdır.
- **SpotBugs (eski adıyla FindBugs):** Java bytecode'u üzerinde çalışan statik analiz aracı. Güvenlik açıkları ve hata eğilimli bölgeleri analiz eder.
- **SonarQube:** Statik analiz sonuçlarını merkezi bir platformda birleştiren, kod kalitesini takip eden ve teknik borç seviyesini yöneten kapsamlı bir araçtır.

- **JDeodorant:** Eclipse IDE üzerinde çalışan bir refactoring yardımcısıdır. Kod kokularını tespit eder ve iyileştirme önerileri sunar.
- **Cyclomatic Complexity (Çevrimsel Karmaşıklık):** Bir fonksiyonun sahip olduğu bağımsız yürütme yollarının sayısını ifade eder. Karmaşıklık arttıkça test etme ve hata ayıklama zorluğu da artar.
- **Halstead Metrikleri:** Bir programın sözcük ve sembollerine dayalı olarak zorluk, hacim ve tahmini geliştirme çabasını ölçen metriklerdir.
- **Maintainability Index (Bakım Endeksi):** Halstead, cyclomatic complexity ve LOC gibi metriklerin birleşimiyle oluşturulan, kodun bakım kolaylığını skorlama yöntemidir.
- **CK Metrikleri (Chidamber and Kemerer Metrics):** Nesne yönelimli yazılımlar için geliştirilen altı temel metrikten oluşur. Bunlar:
    - **WMC (Weighted Methods per Class):** Bir sınıftaki metotların karmaşıklık ağırlıklı toplamıdır.
    - **DIT (Depth of Inheritance Tree):** Sınıfın kalıtım zincirindeki derinliğidir.
    - **NOC (Number of Children):** Bir sınıftan türeyen alt sınıf sayısıdır.
    - **CBO (Coupling Between Object Classes):** Sınıfın diğer sınıflarla olan bağlantı sayısıdır.
    - **RFC (Response For a Class):** Bir sınıfa gelen mesajın tetikleyebileceği toplam metot sayısıdır.
    - **LCOM (Lack of Cohesion in Methods):** Sınıf içindeki metotların birbiriyle ilişkili olup olmadığını ölçer.
- **APM (Application Performance Monitoring):** Uygulamaların çalışma süresi, gecikme süreleri, kaynak kullanımı gibi performans göstergelerini üretim ortamında izlemeye yarayan araçlar grubudur.

Giriş

Andy Hunt ve Dave Thomas'ın *The Pragmatic Programmer* kitabında yer alan "Don't leave 'broken windows' (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered" ("Bozuk 'pencereleri' (kötü tasarımlar, yanlış kararlar veya kötü kod) onarılmadan bırakmayın. Her birini keşfedildiği anda düzeltin") sözü, yazılımdaki küçük problemleri görmezden gelmeme prensibini vurgular [1]. Bu yaklaşım, yazılım projelerinde **kırık pencere teorisi** olarak bilinmekte ve ufak kusurların bile hızla onarılmasını öğütlemektedir. Nitekim, küçük hataların birikmesi "*software rot*" (yazılım çürümesi) denen olguyu hızlandırabilir. Pragmatik programcılık felsefesi, kod kalitesini sürekli yüksek tutarak yazılım çürümesini önlemeyi ve uzun vadede sürdürülebilir bir geliştirme hızını hedefler.

Bu literatür taraması, pragmatik programcı ilkelerinden başlayarak teknik borç kavramına, kod kokularına, refactoring yaklaşımlarına ve Java ekosistemindeki kalite araçlarına kadar geniş bir alanı incelemektedir. Yazılım kalitesini etkileyen **kod kokuları** ve **teknik borç** gibi kavramlar tanımlanacak; **refactoring** teknikleri ve ilkeleri ele alınacaktır. Ayrıca Java dünyasında yaygın olarak kullanılan **statik kod analizi araçları** (PMD, Checkstyle, SpotBugs, SonarQube vb.), otomatik koku tespiti ve giderme araçları (ör. JDeodorant) ile **profiling** yöntemleri (JFR, JMH, VisualVM) ve önemli **yazılım kalite metrikleri** (CK metrikleri, Halstead, çevrimsel karmaşıklık, maintainability index) ayrıntılı olarak incelenecektir. Tüm bu konular, kod kalitesini iyileştirme hedefiyle bağlantılı biçimde ele alınıp, ilgili akademik literatür ve kaynaklardan alıntılar ile desteklenecektir.

1. Pragmatik Programcı İlkeleri ve Teknik Borç

Pragmatik programcılık yaklaşımı, yazılım geliştiricilere esnek, sorumluluk sahibi ve sürekli iyileştirme odaklı bir kültür aşılar. Bu yaklaşımın temelinde, kod kalitesini korumak ve problemleri büyümeden çözmek vardır. Örneğin, The Pragmatic Programmer eserinde vurgulanan ***kırık pencere*** metaforu, yazılımdaki küçük problemlerin onarılmazsa zamanla birikerek sistemi çürütebileceğini belirtir [1]. Bu bakış açısı, Ward Cunningham tarafından ortaya atılan **teknik borç** (*technical debt*) kavramıyla da yakından ilişkilidir. Cunningham, 1992'de yazdığı bir notta "*Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with refactoring*" ("İlk seferde yazılan kodu yayınlamak borca girmeye benzer. Küçük bir borç, refactoring ile hızla geri ödendiği sürece geliştirmeyi hızlandırır") diyerek kısa vadeli kazanımlar için kaliteyi düşürmenin bir tür borç oluşturduğunu ifade etmiştir [2]. Ancak borcun ödenmemesi halinde zamanla faiz yükü bineceğini vurgular: "*Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored implementation*" ("Refaktöre edilmemiş bir uygulamanın borç yükü, tüm mühendislik organizasyonlarını durma noktasına getirebilir") [2]. Bu metafor, kalitesiz kodla hızlı yol almanın yazılıma ileride **birikmiş bakım maliyeti** olarak geri döneceğini anlatır.

Pragmatik programcılar, teknik borcu en aza indirmek için **sürekli iyileştirme** ve **erken müdahale** prensiplerini benimser. "*Yapılmamış her düzeltme, teknik borç defterine eklenen bir kalemdir*" deyişiyle ifade edilebilecek bir anlayış, kodda tespit edilen her sorunun – hataya yol açmasa bile – ileride daha büyük sorunlara neden olabileceği gerekçesiyle hemen ele alınmasını öğütler. Bu bölümde, pragmatik programcılığın öne çıkan ilkeleri olan **DRY**, **Ortogonalite**, **YAGNI** ve **Tracer Bullets** prensipleri kısaca açıklanmaktadır.

DRY (Don't Repeat Yourself):

Pragmatik Programcı'nın en bilinen prensiplerinden biri olan DRY, bir bilgi veya mantığın sistem içinde tek bir yerde tutulması gerektiğini söyler. Kitapta bu ilke "EVERY PIECE OF KNOWLEDGE MUST HAVE A SINGLE, UNAMBIGUOUS, AUTHORITATIVE REPRESENTATION WITHIN A SYSTEM" ("Bir sistem içerisinde her bir bilgi parçasının tek, açık ve otorite kabul edilen bir temsili olmalıdır") şeklinde tanımlanır [1]. Aynı kod veya bilginin tekrarı, ileride güncelleme yapıldığında tutarsızlıklara yol açabileceğinden kaçınılmalıdır. DRY prensibi, kodda gereksiz yinelemeleri önleyerek hem bakım yükünü azaltır hem de değişiklikleri tek noktadan yönetmeyi sağlar.

Ortogonalite (Orthogonality):

Bu ilke, yazılım bileşenlerinin birbiriyle minimum bağımlılığa sahip olmasını, yani birbirinden **bağımsız (decoupled)** çalışabilmesini ifade eder. *The Pragmatic Programmer* yaklaşımı ortogonaliteyi "In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others." ("Bilgisayar bilimlerinde bu terim, bağımsızlık veya ayrık olma anlamına gelir. Eğer birbiriyle ilişkili iki ya da daha fazla şeyde, birinde yapılan değişiklik diğerlerini etkilemiyorsa bunlar ortogonaldir.") diye açıklar [1]. Ortogonal bir tasarımda, modüller veya sınıflar birbirinden yalıtıldığı için birinde yapılan değişiklik diğerinin davranışını bozmaz. Bu sayede bir parça değiştirilirken diğer kısımlara dokunulmaz ve hata olasılığı düşer. Ortogonalite prensibi, sistemde *bağlantısızlık* sağlayarak paralel geliştirmeyi kolaylaştırır ve **bakım maliyetini** azaltır.

YAGNI (You Aren't Gonna Need It):

Extreme Programming (XP) felsefesinden gelen bu prensip, gelecekte gerekebileceği düşünülen bir özelliği önceden kodlamamak gerektiğini savunur. Yani "*şu an ihtiyacın olmayan bir şeyi yapma*" kuralıdır. YAGNI, gereksiz karmaşıklık yaratmamak için geliştirmede sadece o an gerçekten ihtiyaç duyulan işlevlerin uygulanmasını öğütler. Bu prensip, aşırı mühendislik ve **gereğinden fazla tasarım** yapma eğilimine karşı bir uyarıdır. İngilizce bir özdeyişle, *"Always implement things when you actually need them, not when you just foresee that you need them"* ("Bir işlevi gerçekten ihtiyacınız olduğunda hayata geçirin, sadece ileride lazım olabileceğini öngördüğünüz için değil") şeklinde özetlenebilir. YAGNI sayesinde geliştiriciler, kullanmayabilecekleri özellikler için vakit harcamaktan kaçınır ve mevcut gereksinimlere odaklanarak daha yalın, bakımı kolay bir kod tabanı oluşturur.

Tracer Bullets:

Pragmatik Programcı yaklaşımında önerilen bu teknik, belirsiz gereksinimler altında proje geliştirmeye yönelik bir stratejidir. Tracer bullet, sistemin uçtan uca "ince bir dilimi"nin hızlıca kodlanıp çalışır hale getirilmesini temsil eder. Amaç, tam özellikli bir çözüm yazmadan önce mimari yaklaşımın işe yarayıp yaramadığını erken aşamada görmek ve geri bildirim toplamaktır. Bu kavram, gerçek mermi atışları arasına yerleştirilen izli mermilerin hedefi vurup vurmadığını gözlemlemeye benzetilir. Nitekim, bir kaynak *"Tracer bullets are written as a dry run to test whether new architectural designs and technologies are feasible for a given project"* ("Tracer bullet'lar, yeni mimari tasarımların ve teknolojilerin bir proje için uygulanabilir olup olmadığını sınamak amacıyla yapılan bir provadır") diye açıklar [3]. Bu yaklaşımda geliştiriciler, tüm katmanları kapsayan basit bir uçtan uca iskelet uygulamayı hızla hayata geçirip, gerçek koşullarda test eder. Eğer bu *tracer bullet* hedefe isabet ediyorsa yani çözüm doğru yöne gidiyorsa, kalıcı koda dönüştürülebilir; aksi halde hızlıca farklı bir yöne evrilebilir. Tracer bullet tekniği, özellikle belirsiz gereksinimlerde deneme-yanılma yoluyla doğru çözümü keşfetmeyi ve **erken geri bildirim** almayı sağlar. Prototip geliştirmeye benzer ancak tracer bullet kodu genellikle atılmayıp sistemin parçası olabilir ve kısa süreli iterasyonlarla geliştirilir. Bu sayede ekipler, uzun analizler yerine küçük parçalar halinde denemeler yaparak **hızlı yineleme** ile en uygun çözümü ortaya çıkarabilir.

Yukarıdaki ilkeler, pragmatik programcılığın "önce işi çalışır hale getir, ardından sürekli iyileştir" felsefesini somutlaştırır. Küçük hataları büyümeden onarmak, gereksiz işleri yapmaktan kaçınmak ve modüler bir tasarım benimsemek, teknik borcun kontrol altında tutulmasına ve yazılım kalitesinin korunmasına yardımcı olur. Özellikle *kırık pencereyi hemen onarma* ve *teknik borcu biriktirmeme* yaklaşımı, uzun vadede geliştirmenin hızını artırır. Robert C. Martin'in ünlü ifadesiyle, "The only way to go fast, is to go well" ("Hızlı gitmenin tek yolu, iyi gitmektir") — yani temiz ve kaliteli kod, uzun vadede en yüksek geliştirme hızını sağlar [4]. Dolayısıyla pragmatik programcılık, **kısa vadede disiplin, uzun vadede hız** mottosuyla hareket eden bir kalite kültürüdür.

2. Kod Kokuları (Code Smells)

Kod kokusu, kaynak kodda bariz bir hataya yol açmayan ancak ileride bakım ve geliştirmeyi zorlaştırabilecek belirti veya antipaternlere verilen isimdir. Terim ilk olarak Martin Fowler ve Kent Beck tarafından 1999'da *Refactoring: Improving the Design of Existing Code* kitabında tanımlanmıştır. Fowler ve Beck, yazılımda 22 çeşit "bad smell in code" (kötü kod kokusu) belirlemişlerdir [5]. Fowler'ın ifadeleriyle, kötü kokular koda dair "refactoring ile çözülebilecek bir sıkıntı olduğuna dair işaretler" verir ("Bad Smells can give **indications that there is trouble that can be solved by a refactoring**") [6]. Bir başka deyişle, kod kokuları, yazılımın derinlerinde olası bir tasarım veya kalite probleminin habercisidir. **"If it stinks, change it"** ("Eğer kötü kokuyorsa, değiştir") şeklindeki yaygın deyiş de, kodda kötü kokular hissedildiğinde müdahale edilmesi gerektiğini vurgular.

Kod kokularının tipik örnekleri arasında şunlar sayılabilir: Tek bir sınıfın aşırı büyük ve çok sayıda sorumluluğu üstlendiği **God Class** (Tanrı Sınıf) kokusu; bir metodun haddinden fazla uzun ve karmaşık olduğu **Long Method** (Uzun Metot) kokusu; bir sınıfın, asıl ait olmadığı başka bir sınıfın verilerine erişip işlem yaptığı **Feature Envy** kokusu; veya aynı kod parçasının birden çok yerde tekrarlandığı **Duplicate Code** kokusu. Bu kokular doğrudan bir çalışma zamanı hatasına sebep olmasalar da, kodun anlaşılabilirliğini, modülerliğini ve değişime karşı dayanıklılığını azaltır. Örneğin, God Class durumunda tek bir dev sınıf, değişiklik yapmayı güçleştirir ve hata yapma olasılığını artırır; Feature Envy kokusu ise sınıflar arası yüksek bağımlılık ve kapsülasyon ihlali anlamına gelir, bu da ileride yapılacak değişikliklerde hata riskini yükseltir.

Yazılım mühendisliği literatüründe kod kokularının yazılım kalitesine etkisi üzerine birçok çalışma yapılmıştır. Genel kanı, kokuların temizlenmeden bırakılmasının zamanla yazılım çürümesine yol açtığıdır. Nitekim ampirik araştırmalar, *God Class*, *Feature Envy*, *Long Method* gibi kokular barındıran bileşenlerin değişime ve hataya daha yatkın olduğunu göstermiştir. Örneğin, on yıllık bir derleme çalışmasında, bu tür kod kokularına sahip sınıfların hatalanma ve değiştirilme oranlarının belirgin biçimde daha yüksek olduğu not edilmiştir [7]. Diğer yandan, literatürde her kod kokusunun mutlaka acil düzeltme gerektirmeyebileceği yönünde tartışmalar da mevcuttur. Zhang ve arkadaşlarının bir çalışması, bazı kod kokularının varsayıldığı kadar ciddi probleme yol açmayabileceğini ve her kokunun otomatik olarak giderilmesinin her zaman fayda sağlamayabileceğini belirtmiştir [6]. Bu bulgu, "Bad Smells may not indicate problems that significantly affect software" şeklinde rapor edilmiştir ("Kötü kokular, yazılımı önemli ölçüde etkileyen problemlere işaret

etmeyebilir") [6]. Bununla birlikte, çoğu araştırmacı ve uygulayıcı, kod kokularının uzun vadede birikerek bakım maliyetlerini artırdığı ve bu nedenle imkan oldukça giderilmeleri gerektiği konusunda hemfikirdir.

Kod kokularını tespit etmek ve yönetmek için çeşitli yöntemler geliştirilmiştir. **Manuel inceleme (code review)** yoluyla deneyimli geliştiriciler kokuları sezgisel olarak bulabilir. Ancak büyük kod tabanlarında bu yaklaşım zahmetlidir. Bu nedenle, **statik kod analizi** araçları ve metrik tabanlı yaklaşımlar kokuları otomatik tespit etmede kullanılır. Fowler'ın tanımladığı kokuların birçoğu, metrik eşik değerleriyle saptanabilir: Örneğin, çok yüksek sayıda metodu olan veya çok fazla diğer sınıflara bağımlılığı (yüksek CBO değeri) olan bir sınıf, God Class kokusu kriterlerini taşıyabilir; çok uzun bir metot, satır sayısı veya karmaşıklık metriği (örn. McCabe çevrimsel karmaşıklık) eşiğini aşıyorsa Long Method kokusu olarak işaretlenebilir. Bu tür heuristic kurallar kullanan araçlar (PMD, SonarQube vb.) belirli kod kokusu tiplerini raporlayabilir. Bunun yanı sıra, akademik dünyada kokuları tespit etmeye yönelik özel algoritmalar ve araçlar geliştirilmiştir. Bir sonraki bölümde ele alınacak **JDeodorant** aracı bunların önde gelen bir örneğidir; belirli kokuları (Feature Envy, God Class gibi) tespit edip uygun refactoring önerileri sunabilmektedir.

Özetle, kod kokuları yazılım sisteminin "koku sinyalleri" olup kalite problemlerine işaret eder. Kod kokularını erken tespit edip gidermek, teknik borcun büyümesini önler ve yazılımın zaman içindeki evrimini sağlıklı kılar. Pragmatik programcılık bakışıyla, kokular göz ardı edilmemeli; tam tersine, küçük bir tasarım kusuru bile olsa "kırık pencere" mantığıyla hemen onarılmalıdır. Aksi halde bu küçük kusurlar birleşerek yazılımı kırılgan ve değişime dirençli hale getirebilir. Nitekim Douglas Rocha'nın belirttiği gibi, ufak tefek sorunları görmezden gelmek yazılımın çürümesini hızlandırır (ignoring minor code issues accelerates software rot) dolayısıyla temiz ve hoş kokan bir kod üssü için gerektiğinde cesurca refactoring yapmak şarttır.

3. Yeniden Yapılandırma (Refactoring) Teknikleri

Yeniden yapılandırma ya da özgün deyimiyle **refactoring**, var olan bir kodun dış davranışını değiştirmeden iç yapısını iyileştirme sürecidir. Martin Fowler, refactoring kavramını "Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify **without changing its observable behavior**" ("Refactoring: bir yazılımın iç yapısında, dışardan gözlemlenebilen davranışı değiştirmeden, onu daha kolay anlaşılır ve daha ucuza değiştirilebilir kılacak şekilde yapılan bir değişiklik") şeklinde tanımlar [8]. Bu tanım, refactoring'in amacının kodun işlevselliğine dokunmadan kalitesini artırmak olduğunu net bir biçimde ortaya koyar. Refactoring, yeni özellik eklemek veya hata düzeltmek gibi dışa dönük bir sonuç üretmez; bunun yerine mevcut kodun **temizlenmesi, düzenlenmesi ve iyileştirilmesi** üzerine odaklanır.

Refactoring kavramı akademik olarak ilk kez William Opdyke'ın 1992 tarihli doktora tezinde dile getirilmiş, ancak yazılım topluluğunda yaygınlaşması büyük ölçüde Fowler'ın 1999'da yayımladığı kitabıyla olmuştur. Fowler ve Kent Beck'in derlediği *Refactoring* kitabı, sık karşılaşılan kod kokularına yönelik sistematik yeniden yapılandırma tekniklerinin katalogunu sunar [5]. Bu katalogda **Extract Method (Metot Ayıkla)**, **Inline Method (Metot**

Göm), **Extract Class (Sınıf Ayır)**, **Move Method/Field (Metot/Alan Taşı)**, **Rename Method (Metot Yeniden Adlandır)**, **Replace Temp with Query (Geçici Değişkeni Sorguyla Değiştir)** gibi pek çok temel refactoring yöntemi detaylı olarak anlatılmıştır. Her bir refactoring adımı için uygulanabilir koşullar, izlenmesi gereken işlemler ve sonrasında kod kalitesine olan faydalar belirtilir. Örneğin, *Uzun Metod* kokusu tespit edildiğinde **Extract Method** tekniğiyle metot içinde mantıksal olarak ayrılabilecek parçalar ayrı metotlara bölünür; böylece kod daha okunaklı ve yeniden kullanılabilir hale gelir. Benzer şekilde, bir *God Class* tespit edildiğinde **Extract Class** veya **Move Method** gibi işlemlerle sorumluluklar birkaç sınıfa dağıtılır.

Refactoring uygularken dikkat edilmesi gereken en önemli nokta, davranışın korunmasıdır. Kodun dışarıya verdiği çıktı veya gerçekleştirdiği iş aynı kalmalı, yalnızca iç yapısı değişmelidir. Bu sebeple, refactoring sürecinde **otomatik testler** kritik rol oynar. Fowler, refactoring yapmadan önce kapsamlı bir test setine sahip olmanın şart olduğunu vurgular: “Before you start refactoring, make sure you have a solid suite of tests. These tests must be self-checking.” (“Refactoring’e başlamadan önce sağlam bir test paketin olduğundan emin ol. Bu testler kendi kendini kontrol eden (otomatik doğrulayan) testler olmalıdır.”) [9]. Refactoring esnasında sık sık testleri çalıştırarak hiçbir davranışın bozulmadığını teyit etmek gerekir. Bu sayede geliştiriciler, yaptıkları iç değişikliklerin güvenli olduğunu bilerek ilerlerler. Martin Fowler, kitabının girişinde, testlerin refactoring sırasındaki güvenlik ağı olduğunu ve değişikliklerin olası yan etkilerini anında yakalamayı sağladığını belirtir [9]. Gerçekten de, birim testleriyle desteklenmeyen kodda refactoring yapmak ciddi risk içerir; istenmeyen regresyon hataları doğurabilir. Bu nedenle iyi bir test kapsamı, refactoring’i cesaretle ve etkin şekilde yapabilmenin anahtarıdır.

Refactoring stratejileri iki şekilde ortaya çıkabilir: **Planlı (proaktif)** refactoring ve **fırsatçı (opportunistic)** refactoring. Planlı refactoring, belirli aralıklarla veya geliştirmenin ayrı bir safhasında kod iyileştirme işine odaklanılmasıdır. Örneğin bir sürüm tamamlandıktan sonra, yeni özellik eklemeden önce bir sprint’i tamamen kod temizliğine ayırmak gibi. Fırsatçı refactoring ise, geliştirici kodla uğraşırken karşısına çıkan iyileştirme fırsatlarını anında değerlendirmesidir – yani “işini yaparken boynuna takılan kokuları temizlemek”. Fowler, refactoring’in günlük geliştirme pratiğinin doğal bir parçası olması gerektiğini ve kodu değiştirmek için illaki ayrı bir görev beklenmemesi gerektiğini savunur [10]. Kodla her temas edildiğinde, **boyscout rule** olarak bilinen “kamp alanını terk ederken bulduğundan daha temiz bırak” kuralı uygulanmalıdır. Bu, yazılımcının dokunduğu kod bölümünü mümkün olduğunca ufak da olsa iyileştirerek bırakması demektir. Böylece zaman içinde küçük refactoring adımlarıyla kod tabanı evrilir ve iyileşir.

Refactoring teknikleri uygulanırken ekibin ortak standartlara ve en iyi pratiklere dikkat etmesi gerekir. Örneğin, bir değişiklik yaparken yalnızca kokuyu gidermekle kalmayıp kodun genel tasarım ilkelerine uygun hale getirilmesi önemlidir (SOLID prensipleri, düşük bağımlılık, yüksek tutarlılık vb.). Ayrıca, refactoring sonrası mutlaka tüm testlerin geçtiği doğrulanmalı ve mümkünse akran kod incelemesiyle değişiklikler gözden geçirilmelidir. Büyük ölçekli refactoring işlemlerinde (örneğin mimari bir dönüşüm veya çok sayıda modüle dokunan bir yeniden yapılandırma) adımların küçük parçalara bölünmesi ve kademeli entegrasyon tavsiye edilir. Aksi takdirde, çok büyük bir değişikliği tek seferde yapmak proje riskini artırabilir.

Sonuç olarak refactoring, temiz ve sürdürülebilir kod yapısına ulaşmanın en önemli araçlarındandır. Teknik borcun ödenmesi, kod kokularının giderilmesi ve tasarımın zamanla evrilerek iyileşmesi için refactoring vazgeçilmez bir süreçtir. Pragmatik programcı yaklaşımı da refactoring'i sürekli ve doğal bir faaliyet olarak görür. **"Kırık pencereyi onarmak"** için çoğu zaman yapılması gereken, uygun refactoring tekniğini uygulayarak kodu bir adım iyileştirmektir. Bunu yaparken otomasyon (IDE refactoring araçları) ve testlerin güvenliğine başvurmak geliştiricilerin elini güçlendirir. Unutulmamalıdır ki, temiz kod ve iyi tasarım için harcanan çaba, uzun vadede katlanarak geri ödenir – bakım kolaylaşır, hata oranı düşer, yeni özellik ekleme hızı artar. Refactoring, bu anlamda yazılımın sağlıklı büyümesinin temel taşlarındandır.

4. Statik Kod Analizi ve Araçları

Büyük ve karmaşık kod tabanlarında elle kod incelemesi yapmak zor olduğu için, **statik kod analizi araçları** yazılım kalitesini otomatik olarak değerlendirmede kritik bir rol oynar. Statik analiz araçları, kodu derleme veya çalıştırma gerektirmeden belirli kurallara ve metriklere göre tarayarak olası hataları, kötü kokuları ve standart dışı yapıları tespit eder. Java ekosisteminde yıllar içinde geniş bir yelpazede statik analiz aracı geliştirilmiştir. Bu bölümde özellikle popüler ve yaygın kullanılan bazı araçlar tanıtılmaktadır: **PMD**, **Checkstyle**, **FindBugs/SpotBugs** ve **SonarQube**.

4.1. PMD

PMD, Java (ve diğer diller) için geliştirilmiş açık kaynaklı bir statik kod analiz aracıdır. Kod üzerinde önceden tanımlanmış kural setlerini çalıştırarak olası sorunları raporlar. Resmi tanımına göre *"PMD is an extensible multilanguage static code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth."* ("PMD, genişletilebilir yapıda, birden çok dili destekleyen bir statik kod analizörüdür. Kullanılmayan değişkenler, boş yakalama blokları, gereksiz nesne oluşturma gibi yaygın programlama hatalarını tespit eder.") [11]. Örneğin PMD, bir değişken tanımlandığı halde hiç kullanılmamışsa veya bir try-catch bloğunun içi boş bırakılmışsa uyarı verir. Ayrıca kopya kod tespiti, kompleks ifadeler, anlaşılması güç yapılar gibi birçok

kod kokusunu da kural tanımları aracılığıyla saptayabilir. PMD'nin kural seti özelleştirilebilir; projeye özgü kurallar eklenebilir ya da istenmeyen kurallar kapatılabilir. Hafif yapısı ve kolay entegrasyonu sayesinde PMD, birçok Java projesinde *build* sürecine entegre edilerek her derlemede kodu taramak için kullanılmaktadır. Bu sayede geliştiriciler, derleme aşamasında

olası kalite problemlerinden haberdar olup kodu düzeltme imkanı bulur. PMD, özellikle **kod stili ihlalleri**, **basit logik hatalar** ve **performans anti-paternlari** yakalamada faydalı bir ilk savunma hattıdır.

4.2. Checkstyle

Checkstyle, Java kodunun belirli bir kodlama standardına uyup uymadığını denetleyen popüler bir statik analiz aracıdır. Temel amacı, kodun stil ve format konularında tutarlılığını sağlamaktır. Checkstyle'ın kendi belgelerinde belirtildiği gibi *"Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code."* ("Checkstyle, programcıların belirlenmiş bir kodlama standardına uygun Java kodu yazmalarına yardımcı olan bir geliştirme aracıdır. Java kodunu otomatik olarak bu standarta göre kontrol eder.") [12]. Genellikle Google Java Style Guide veya Sun/Oracle kodlama standartları gibi yaygın stil rehberlerine göre yapılandırılır. Checkstyle, kodu satır satır inceleyerek isimlendirme kuralları (sınıf, metot, değişken adları), girinti boşlukları, satır uzunluğu, import sırası, kod blok yerleşimleri, Javadoc yorumlarının varlığı gibi pek çok yönergeyi denetler. Örneğin, metod isimleri küçük harfle başlamalı ve camelCase olmalı gibi bir kural tanımlıysa, buna uymayan tüm metodları raporlar. Yine, bir sınıf çok fazla public metot içeriyorsa (tasarım açısından fazla sorumluluk belirtisi olabilir) uyarı verebilir. Checkstyle, hataları önem derecelerine göre (uyarı, hata vs.) sınıflandırarak bir rapor sunar. Kod inceleme süreçlerinde stil tartışmalarını en aza indirmek ve otomatikleştirmek için idealdir; böylece ekip üyeleri aynı standartta kod yazarak daha okunaklı ve bakım dostu bir kod tabanı oluşturabilir. Checkstyle da PMD gibi build araçlarına (Maven, Gradle) kolayca entegre edilebilir ve sürekli entegrasyon ortamlarında her derlemede kalite kontrolü yapar. Unutulmamalıdır ki, Checkstyle daha çok **stil ve format** odaklıdır; işlevsel hataları bulma konusunda sınırlıdır. Ancak kod düzeninin tutarlılığı, genel kalite kültürünün önemli bir parçası olduğundan, Checkstyle'ın disipline edici etkisi yazılım yaşam döngüsünde değerli katkı sunar.

4.3. FindBugs / SpotBugs

FindBugs, Java bytecode'u üzerinde çalışan ve olası bug örüntülerini yakalamaya odaklanan bir statik analiz aracıdır. Maryland Üniversitesi'nde geliştirilmiş ve uzun yıllar boyunca Java topluluğunda yaygın kabul görmüştür. 2017'den itibaren orijinal FindBugs projesi duraksadığı için topluluk tarafından **SpotBugs** adıyla çatallanarak sürdürülmüştür. SpotBugs, FindBugs'ın devamı niteliğinde aynı analiz yaklaşımını güncel Java sürümleri için devam ettirir. FindBugs için proje web sayfasında şu ifade yer alır: *"FindBugs is a program which uses static analysis to look for bugs in Java code."* ("FindBugs, Java kodunda bug aramak için statik analiz kullanan bir programdır.") [13]. FindBugs/SpotBugs, PMD ve Checkstyle'dan farklı olarak daha çok olası program hatalarına odaklanır.

Örneğin, *null pointer* kullanımı riski, dizilerde indeks taşması, eşitlik kontrolünde == operatörünün yanlış kullanımı, sonsuz döngü potansiyeli, kullanılmadan atanan değişkenler, hatalı bit işleç kullanımı gibi birçok spesifik **bug paterni** için dedektörlere sahiptir. Toplamda yüzlerce tanımlı bug deseni bulunur ve SpotBugs bu kalıpları bytecode analizine dayanarak

tespit eder. Bytecode seviyesinde çalışması, dilin semantiğini (ör. Java Memory Model) daha iyi anlamasını ve derleyicinin ürettiği kodu analiz ederek bazı dil tuzaklarını yakalamasını sağlar. FindBugs'ın etkin kullanımıyla büyük ölçekli projelerde ciddi hatalar keşfedilebildiğine dair raporlar mevcuttur. Örneğin Google, kendi kod tabanında FindBugs kullanımını teşvik etmiş ve "FindBugs Fixit" adıyla bir etkinlik düzenleyerek geliştiricilerin yüzlerce potansiyel hatayı düzeltmesini sağlamıştır [14]. Bu girişimde, statik analiz aracının uyarıları sayesinde önceden farkedilmemiş hataların ortaya çıkarıldığı bildirilmiştir [14]. SpotBugs, FindBugs'ın mirasını sürdürürken yeni Java özelliklerini (modüler sistem, lambda ifadeleri vb.) destekleyecek şekilde güncellenmiştir ve eklenti mimarisi sayesinde ek dedektörler geliştirilebilmektedir. Sonuç olarak, SpotBugs bir projede **otomatik hata avcısı** rolünü oynar – derlemenin yakalayamadığı veya runtime'da henüz gerçekleşmemiş hataların izlerini kod üzerinde keşfederek yazılımcıya erken uyarılar verir.

4.4. SonarQube

SonarQube, açık kaynaklı ve güçlü bir **sürekli kod kalitesi denetim** platformudur. Diğer tekil araçlardan farklı olarak, birden fazla analiz motorunu ve kural setini entegre eden, sonuçları merkezi bir sunucuda toplayıp görselleştiren kapsamlı bir çözümdür. SonarQube, kodu birden fazla boyutta değerlendirir: **maintainability (sürdürülebilirlik)**, **reliability (güvenilirlik)**, **security (güvenlik)**, **coverage (test kapsaması)** gibi kategorilerde metrikler ve göstergeler sunar. Özellikle maintainability alanında, kod kokuları ve teknik borç ölçümleriyle öne çıkar. SonarQube analizleri sonucunda her proje için toplam **Code Smell** sayısı ve toplam **Technical Debt** süresi raporlanır. Teknik borç genellikle dakika veya gün cinsinden ifade edilir ve SonarQube bunu "kod kokularını gidermek için gereken tahmini süre" olarak tanımlar [15]. Örneğin SonarQube dokümantasyonunda *"Technical debt (sqale_index): A measure of effort to fix all code smells. The measure is stored in minutes in the database."* şeklinde bir tanım vardır ("Teknik borç: tüm kod kokularını düzeltmek için gereken eforun ölçüsüdür. Bu ölçüm veritabanında dakika cinsinden saklanır.") [15]. Bu, kod kalitesindeki eksiklerin ne kadarlık bir iş yükü doğurduğunu somutlaştıran faydalı bir göstergedir. Ayrıca SonarQube, **Maintainability Rating** adı altında bir harf notu (A, B, C, D, E) verir ve bunu teknik borcun kod tabanını geliştirmek için harcanan süreye oranına göre hesaplar (örn. teknik borç oranı %5'ten az ise A, %20'ye kadarsa C gibi) [15].

SonarQube sadece metrik sunmakla kalmaz, aynı zamanda **kalite eşikleri (Quality Gates)** tanımlamaya imkan verir. Yazılım ekipleri, kabul edilebilir en yüksek kod kokusu sayısı, en düşük kapsama oranı gibi eşikleri belirleyip SonarQube üzerinde bir kalite kapısı oluşturabilirler. CI (Continuous Integration) süreçleriyle entegre edildiğinde, SonarQube analizi bu eşiklerin aşılması durumunda uyarı verebilir veya derlemeyi başarısız kılabilir. Bu, kalitenin sürekliliğini sağlamak için oldukça etkili bir mekanizmadır. Örneğin, bir projede kalite kapısı "Yeni kodda teknik borç oranı %0.1'i geçmeyecek" şeklinde belirlenmişse, geliştiriciler her yeni geliştirmede temiz kod yazmaya teşvik edilir çünkü aksi halde SonarQube alarm verecektir.

SonarQube, altında yatan kurallar motoru olarak PMD, Checkstyle, SpotBugs benzeri analizleri kullanır (veya kendi eşdeğerlerini). Yani kodu tarayıp bulguları toplar ve merkezi bir veritabanında saklar. Geliştiriciler, SonarQube arayüzünden kod kokusu detaylarını, ilgili

kod satırlarını ve düzeltme önerilerini görebilirler. SonarQube ayrıca tarihsel trendler sunarak zaman içindeki kalite değişimini izlemeye olanak tanır (örneğin son 3 ayda teknik borç eğrisi nasıl ilerlemiş gibi). Java projelerinde SonarQube kullanımı oldukça yaygındır, özellikle kurumsal ortamlarda **Sürekli Entegrasyon** devrelerine entegre edilir. SonarQube'un bir avantajı da **SonarLint** gibi IDE eklentileriyle entegrasyondur – böylece geliştirici kod yazarken anlık geri bildirim alabilir, kod kokularını daha commit etmeden IDE üzerinde görebilir.

Özetle, SonarQube bir projenin kod kalitesini çok boyutlu bir şekilde izlemek ve yönetmek için kapsamlı bir platform sunar. Kod kokularını sayısallaştırıp teknik borca çevirerek iş birimi ile geliştiriciler arasında ortak bir dil oluşturur (örneğin "Bu özelliği hemen teslim edebiliriz ama 2 gün teknik borç birikiyor" gibi). Bu yönüyle, pragmatik programcılığın **sürekli kalite farkındalığı** ilkesini araçsal olarak destekleyen en önemli çözümleyicilerden biridir.

5. Kod Kokusu Tespit ve Otomatik Refactoring Araçları

Kod kokularının tespiti ve giderilmesi konusunda, genel statik analiz araçlarının ötesinde, özel amaçlı bazı akademik ve endüstriyel araçlar da geliştirilmiştir. Bu araçlar, belirli kokuları daha sofistike analizlerle bulmaya ve mümkünse otomatik veya yarı-otomatik refactoring uygulamaya odaklanır. Bu bölümde özellikle **JDeodorant** aracı ile diğer bazı araştırma prototiplerinden bahsedilecektir.

5.1. JDeodorant

JDeodorant, yazılım mühendisliği araştırma topluluğunda tanınmış bir araçtır ve temel amacı, belirli kötü kokuları otomatik analizle tespit edip bunları gidermek için refactoring önerileri sunmaktır. Nikolaos Tsantalis ve ekibi tarafından akademik olarak geliştirilmiş olan JDeodorant, özellikle *Feature Envy*, *God Class*, *Long Method* gibi kokular üzerinde etkilidir. Eclipse IDE üzerinde bir eklenti olarak çalışır ve kodu analiz ederek bulduğu kokuları geliştiriciye listeler; ardından her bir koku için uygulanabilecek refactoring işlemlerini (ör. Move Method, Extract Class, Extract Method gibi) önerir. Uygulayıcı isterse bu önerileri bir kaç tıklamayla kod üzerinde gerçekleştirebilir. JDeodorant'ın on yılı aşkın geliştirme serüveni üzerine yapılan bir derleme makalesinde, aracın başarıları ve kısıtları değerlendirilmiştir [7]. Bu çalışma, JDeodorant'ın Feature Envy ve Large Class (God Class) gibi kokuları belirlemede yüksek başarı sağladığını, ayrıca benzer amaçlı başka araç ve tekniklerin de JDeodorant sonuçları ile kıyaslanarak geliştirildiğini belirtmiştir [7]. Örneğin, JDeodorant'ın tespit ettiği kokular ile manüel kod incelemelerinin büyük ölçüde örtüştüğü, aracın bir anlamda deneyimli bir geliştiricinin sezgisel bulgularını otomatikleştirdiği rapor edilmiştir.

JDeodorant'ın ardındaki teknoloji, kodun **nesne yönelimli metriklerini** ve kullanım örüntülerini analiz etmeye dayanır. Örneğin *Feature Envy* kokusunu tespit etmek için, bir metodun kendi sınıfı dışındaki başka bir sınıfın alan ve metodlarını ne sıklıkla kullandığına bakar; belirli bir eşik üzerinde ise o metodun aslında diğer sınıfa ait olması gerektiğini çıkarımlar (bu, Move Method refactoring'ine bir adaydır). *God Class* tespiti için, sınıfın büyüklüğü, karmaşıklığı, bağımlı olduğu ve kendisine bağımlı olan sınıfların sayısı gibi metrikleri değerlendirir; aşırı yüksekse sınıfın parçalanması önerilir. JDeodorant, geliştiriciye sunduğu refactoring önerilerinde değişikliğin etki analizini de gösterir – örneğin bir metodu taşımanın hangi diğer sınıfları etkileyeceği, taşıma sonrası erişim düzeyi değişimleri vs. gibi bilgiler sağlanır. Geliştirici onay verdiğinde, JDeodorant ilgili kod değişikliklerini Eclipse ortamında otomatik olarak yapar (metodu yeni sınıfa taşır, eski yerde delegasyon bırakır veya kullanımları günceller vb.). Bu yarı otomatik onarım, kokuların giderilmesini hızlandırır ve insan hatasını minimize eder.

JDeodorant'ın akademik literatüre katkılarından biri de kod kokularının etkisine dair empirik çalışmalara imkan tanımasıdır. Araç kullanılarak geniş kod tabanlarında kokular taranmış, giderilmiş ve sonrasında değişim hızı veya hata oranı gibi metrikler ölçülmüştür. Tsantalis ve arkadaşlarının 2018'deki derlemesi, JDeodorant'ın gelişimi sayesinde kokular üzerine yapılan araştırmalarda önemli içgörüler kazanıldığını not eder [7]. Örneğin, JDeodorant kullanarak belirli bir projede Feature Envy kokularını giderdiklerinde, gereksinim izlenebilirliğinde (requirements traceability) iyileşme gözlemlendiğini rapor etmişlerdir – çünkü fonksiyonlar doğru yerlere taşınınca, bir özelliğin kod izleri daha tutarlı hale gelmiştir [7].

Kısıtlarına gelirsek: JDeodorant her kokuyu mükemmel tespit edemez; bazen yanlış pozitif (aslında problem olmayan bir durumu kokulu sanma) veya yanlış negatif (bazı kokuları kaçırma) durumları olabilir. Ayrıca otomatik refactoring önerileri her zaman uygulanabilir olmayabilir – özellikle geliştirme ortamındaki diğer kısıtlar (örn. özel mimari kurallar, eşzamanlı geliştirme, vs.) nedeniyle. Bu yüzden araç önerileri genellikle geliştirici incelemesinden sonra uygulanır. Yine de, JDeodorant gibi bir aracın varlığı, geliştiricilere kokular konusunda *otomatik bir ikinci görüş* sunmakta ve refactoring işlemlerini kolaylaştırmaktadır.

5.2. Diğer Refactoring Öneri Sistemleri

JDeodorant'ın yanı sıra, akademik alanda ve endüstride çeşitli refactoring öneri sistemleri geliştirilmiştir. Örneğin **JetBrains IntelliJ IDEA** IDE'si, kod analizine dayalı bazı akıllı öneriler sunabilmektedir (ör. "bu kod parçası tekrarlanıyor, bir metot çıkarılsın mı?" gibi). Microsoft Research tarafından geçmişte geliştirilmiş **"Refactoring Miner"** gibi araçlar, versiyon kontrol geçmişini analiz ederek sık yapılan değişikliklerden refactoring desenleri çıkarmaya çalışmıştır. Benzer şekilde **CodeCritics** gibi araştırma prototipleri, yazılımcıların kodu nasıl kokulara karşı refactor ettiğini öğrenip proaktif uyarılar verme amacı taşımıştır.

Son dönemde, **yapay zeka ve makine öğrenmesi** tabanlı yaklaşımlar da refactoring önerileri için kullanılmaktadır. Örneğin bazı çalışmalar, büyük kod depolarından elde edilen verilere dayanarak "hangi kod parçasının hangi refactoring ile iyileştirildiği" konusunda modeller eğitmiştir. Bu modeller, yeni bir projede benzer bir kokulu durum bulduğunda öneride bulunabilir. Bunun ilk örneklerinden biri, **Han et al. (2012)** tarafından geliştirilen bir sistemde, kod kokusu içeren sınıfların geçmişteki benzerleri hangi refactoring'lerden geçmişse onu önermesiydi. Günümüzde GitHub gibi devasa kaynaklardan yararlanarak bu tür tavsiye sistemleri daha da gelişmektedir.

"Araç" sınıfına girmese de, **revizyon kontrol analizi** de refactoring'i tespit ve tavsiye için kullanılır. Yaroslav Golubev ve ekibinin 2021'de yaptığı büyük ölçekli anket çalışması, geliştiricilerin refactoring araçlarına bakışını incelemiş ve IntelliJ IDEA kullanıcılarının otomatik refactoring özelliklerini sıklıkla kullandığını ancak tamamen otomatik düzeltme öneren harici araçlara temkinli yaklaştığını ortaya koymuştur [16]. Bunun sebepleri arasında güven eksikliği (araç önerisinin yan etkisi olabilir endişesi) ve kodun kontrolünü elde tutma isteği sayılabilir.

Özetle, JDeodorant öncü bir araç olup onun izinden giden çeşitli refactoring tavsiye sistemleri geliştirilmiştir. Akademik araştırmalar, hiçbir tekniğin veya aracın *mükemmel kapsama* sunmadığını, dolayısıyla farklı araçların kombinasyonu ve son tahlilde **insan yargısı** ile en iyi sonucun alındığını göstermektedir [16]. Bu nedenle uygulamada, JDeodorant gibi araçlar geliştiriciye yardımcı asistan rolünde kullanılmalı; son karar ve doğrulama geliştiricide olmalıdır. Yine de, bu araçların varlığı kod kalitesini iyileştirme yolunda pragmatik bir destek sağlar – geliştiriciye zaman kazandırır ve dikkatinden kaçabilecek kokuları yakalayarak sürekli temiz bir kod üssü oluşturmasına katkı sunar.

5.3. Otomatik Düzeltme ve Program Onarma Yaklaşımları

Kod kokusu ve refactoring özelinde olmasa da, genel olarak yazılım hatalarını otomatik olarak düzeltmeye çalışan araştırma alanı **program repair (program onarma)** olarak bilinir. Bu alanda, birim testleri başarısız olan kodu analiz edip, küçük değişikliklerle testi geçer hale getirmeyi amaçlayan teknikler geliştirilmiştir. Genellikle sezgisel arama (genetic programming vb.) ile çalışan bu sistemler, kodda hataya sebep olabilecek bölgeyi tespit edip oraya yönelik yaygın onarım stratejilerini dener (ör. null kontrolü ekleme, sınır değerlerini düzeltme gibi). Program onarma teknikleri doğrudan geliştirme pratiğinde yaygın kullanılmasa da (henüz araştırma prototipi aşamasındalar), yazılım kalitesini otomatik iyileştirme vizyonunun bir parçasıdır. Bu tekniklerin kod kokularına uygulanması da teorik olarak mümkündür: Örneğin bir araç, çok uzun bir metodu tespit edip onu ikiye bölecek bir onarım önerebilir. Ancak kokular işlevsel hatalar kadar somut ve tek ölçüte indirgenebilir olmadığından, otomatik kokugiderme daha karmaşık bir problemdir.

Endüstride ise **IDE destekli refactoring** zaten bir tür yarı otomatik program onarma sayılabilir, geliştirici niyetini belirtir (örneğin "bu metodu başka bir sınıfa taşı" komutu verir) ve IDE bunun kodu bozmayacak şekilde gerçekleşmesini sağlar. Bu, bug onarmadan ziyade kalite onarmadır. Yine GitHub gibi platformlarda, **Copilot** gibi yapay zeka tabanlı araçlar artık geliştiricilere hatalı veya kokulu kodu düzeltme konusunda öneriler sunabilmektedir. Bu araçlar, sorunu doğal dilde tarif edip çözüm kodunu üretebiliyorlar. Bu gelişmeler ışığında, gelecekte daha akıllı otomatik refactoring yardımcıları görmemiz olasıdır.

Genel olarak, **otomatik düzeltme** vizyonu ne tamamen ütopik ne de tam anlamıyla gerçek olmuş bir durumdur. Şu an için en iyi yaklaşım, *araçlar ve insan bilgeliğini bir araya getirerek* kaliteli kod elde etmektir. Kod kokusu tespiti ve otomatik refactoring araçları, geliştiricinin işini kolaylaştıran ve hızlandıran birer pragmatik çözümdür fakat nihai kalite için geliştiricinin dikkatli incelemesi ve yazılım tasarım prensiplerine hakimiyeti vazgeçilmezdir.

6. Performans Analizi ve Profiling Araçları

Kod kalitesi iyileştirmelerinde temel amaç daha sürdürülebilir ve hatasız bir kod üssü oluşturmak olsa da, performans karakteristikleri de göz ardı edilmemelidir. Çoğu durumda refactoring ve temiz kod uygulamaları performansı olumlu yönde etkiler veya en azından nötrdür (daha az kompleks kod genelde daha verimlidir). Ancak bazı durumlarda, örneğin soyutlamaların artması veya ek kontrollerin gelmesi, çok hassas performans kısıtları altında ufak maliyetler doğurabilir. Bu nedenle, **profiling** adı verilen performans analiz yöntemleriyle, kod iyileştirmelerinin hız ve kaynak kullanımı üzerindeki etkisini ölçmek iyi bir pratiktir. Java ekosisteminde performans analizi için çeşitli araçlar ve yöntemler mevcuttur: **Java Flight Recorder (JFR)** ve bağlı aracı **JDK Mission Control**, **Java VisualVM**, bağımsız ticari araçlar (**JProfiler**, **YourKit** gibi) ve mikro-benchmark yazma kütüphaneleri (örn. **JMH** – Java Microbenchmark Harness). Bu bölümde özellikle açık kaynak veya JDK ile gelen araçlara odaklanılacaktır.

6.1. Java Flight Recorder (JFR) ve JDK Mission Control

Java Flight Recorder (JFR), Oracle JDK ile entegre gelen, Java uygulamalarının çalışma zamanında düşük overhead ile profilini çıkarabilen bir araçtır. JFR, tıpkı bir uçak kara kutusu gibi uygulamanın çalışma sırasında çeşitli olayları kaydeder ve gerektiğinde bu kayıtlar analiz edilerek performans sorunları tespit edilir. JFR'nin en güçlü yanlarından biri, prodüksiyon ortamlarında bile çok az performans cezasıyla sürekli çalıştırılabilecek hafiflikte olmasıdır. Oracle'ın JFR dokümanında belirtildiği üzere *"Flight Recorder provides a low overhead way of collecting detailed profiling information about a running Java application"* (Flight Recorder, çalışan bir Java uygulaması hakkında ayrıntılı profil bilgilerini düşük ek yükle toplamanın bir yolunu sağlar) şeklinde tanımlanabilir. Nitekim JFR, JDK içinde -*XX:StartFlightRecording* parametreleriyle etkinleştirilip uygulamanın CPU kullanımından, bellek ayak izine; garbage collection duraklamalarından, IO bekleme sürelerine kadar pek çok metrik ve olayı kayda alabilir.

JFR çıktıları genellikle binary bir .jfr dosyası olarak saklanır. Bu kayıtların analizi için Oracle tarafından sunulan **JDK Mission Control (JMC)** aracı kullanılır. JMC, grafik arayüzlü bir uygulama olup JFR kayıtlarını zaman ekseninde görselleştirir, iş parçacığı aktivitelerini, *hotspot* dediğimiz en çok zaman harcanan kod bölgelerini, kilitlenme (lock contention) durumlarını ve daha fazlasını kullanıcıya sunar. Örneğin JFR kaydı inceleyerek bir uygulamanın CPU zamanının %40'ını belirli bir metodda harcadığını ve bunun çoğunun bekleme (sleep) olduğunu fark edebilirsiniz; ya da bellek tahsis oranlarının anormal yüksek olduğu bir kod parçasını tespit edebilirsiniz. Bu tür bilgiler, performans darboğazlarını gidermede kritiktir.

JFR/JMC ikilisi özellikle **sunucu tarafı** Java uygulamalarının (ör. web uygulamaları, servisler) üretimde karşılaştığı anlık sorunları teşhis etmede çok işe yarar. Örneğin, belirli bir günde sunucu yanıt süreleri düşmüşse, o ana ait JFR kaydı incelenerek belki de o sırada anormal bir GC olayı ya da veritabanı çağrısı beklemesi gerçekleştiği anlaşılabilir. JFR'nin pragmatik programcı bakışı açısından önemi, geliştiriciye **gerçek veriye dayalı** optimize etme imkanı sunmasıdır. Böylece gereksiz "premature optimization" (zamansız optimizasyon) uğruna kod karmaşıklığını artırmak yerine, gerçek darboğazları bulup onlara odaklanabiliriz. Bu bağlamda Donald Knuth'un ünlü "Premature optimization is the root of all evil" ("Erken optimizasyon, tüm kötülüklerin anasıdır") sözü hatırlanabilir [17]. JFR gibi bir araç, erken optimizasyon yapmadan önce performansı ölçerek hareket etmeyi mümkün kılar.

Özetle, Java Flight Recorder ve Mission Control, Java uygulamalarının ayrıntılı profilini çıkarmak için güçlü bir kombinasyondur. Refactoring veya kod değişiklikleri sonrasında sistemin performans davranışını gözlemlemek için de idealdir. Örneğin, bir *Long Method* parçalandıktan sonra aynı işlev belki biraz daha fazla metod çağrısı gerektirecek; JFR ile ölçüldüğünde bu ek çağrıların ihmal edilebilir düzeyde olduğu doğrulanabilir. Veya tam tersi, bir değişiklik beklenmedik şekilde CPU tüketimini artırmışsa JFR bunu gösterebilir ve sorunu geri almak gerekebilir. **Düşük overhead'li profil kaydı**, pragmatik yaklaşımla sürekli kalite iyileştirmenin performans kanadını destekler niteliktedir.

6.2. Java VisualVM

VisualVM, Java ile birlikte gelen (JDK 6 ve 7'de resmi, sonrasında ayrı proje olarak) grafiksel bir profil aracıdır. VisualVM, Java uygulamalarına bağlanarak onların anlık bellek ve CPU kullanımını, *thread dump* ve *heap dump* gibi iç yapısını görmeye imkan tanır. VisualVM'nin sunduğu özelliklerden bazıları: **CPU Profiler** (hangi metodların ne kadar CPU süresi harcadığını örnekleme tekniğiyle ölçer), **Memory Profiler** (hangi tipten kaç nesne oluşturulduğunu ve hafızada ne kadar yer kapladıklarını izler), *Garbage Collector monitoring* (GC kaç kez çalışmış, toplam ne kadar süre harcanmış vs.), ve thread izleme (şu anda hangi thread ne yapıyor, beklemede mi çalışıyor mu gibi).

VisualVM'nin avantajı, kullanımı kolay ve görsel olmasıdır. Örneğin bir Java uygulamasını VisualVM ile profile modunda başlatıp birkaç dakika test senaryolarını çalıştırdığınızda, araç size en çok zaman alan 10 metodu sıralayabilir. Ya da bellek profili alıp hangi nesne türlerinin yığın bellekten çoğunu tükettiğini gösterir. Eğer bir bellek sızıntısı (memory leak) şüphesi varsa, heap dump alıp inceleyerek hangi nesnelerin beklenmedik şekilde tutulduğunu saptayabilirsiniz. VisualVM, JFR kadar düşük seviyeli detaylara inmez ve sürekli kayıt yerine anlık örnekleme yapar, dolayısıyla overhead'i biraz daha hissedilir ama kısa süreli analizler için idealdir.

Özellikle masaüstü uygulamaları veya test ortamları için VisualVM hızlı teşhisler sağlar. Diyelim ki bir refactoring sonrası performansta bir düşüş fark ettiniz; VisualVM ile bakıp belki de farkında olmadan çok sayıda kısa ömürlü nesne yaratmaya başladığınızı (ve GC yükünü artırdığınızı) görebilirsiniz. Mesela, bir önceki tasarım bir sonuç listesini caching yaparken, yeni tasarım her seferinde yeniden oluşturmaya başlamış olabilir – VisualVM bellek profilinde bu artışı gösterir. Yine, VisualVM ile thread durumlarını izleyerek olası *deadlock* veya bekleme problemlerini de tespit edebilirsiniz.

VisualVM'nin güncel sürümleri artık ayrı bir açık kaynak proje (visualvm.github.io) olarak ilerlemektedir ve eklentilerle zenginleştirilebilmektedir. Alternatif olarak IntelliJ IDEA ve Eclipse gibi IDE'lerde de yerleşik basit profilleyiciler bulunur, ancak VisualVM bağımsız ve hafif oluşuyla hala tercih edilir.

6.3. JProfiler ve YourKit

JProfiler ve **YourKit**, Java için yaygın kullanılan ticari profil araçlarıdır. Bu araçlar, VisualVM'nin sahip olduğu hemen hemen tüm özellikleri (CPU, bellek, thread analizi vs.) daha gelişmiş arayüzler ve ek fonksiyonlarla sunarlar. Örneğin JProfiler, veritabanı sorguları ve TCP soket trafiği gibi uygulama içinde sıkça kullanılan kütüphane çağrılarını da izleyip raporlayabilir. YourKit, *allocation profiling* denilen ve hangi kod satırının ne kadar nesne oluşturduğunu kesin olarak hesaplayan bir özellik içerir.

Ticari profiller genellikle kurumsal projelerde performans darboğazlarını çözmek için kullanılır. Bu araçlar sayesinde geliştiriciler, uygulamanın en küçük ayrıntısına kadar inebilir ve spesifik sorunları izole edebilir. Örneğin, bir web uygulamasında toplam yanıt süresinin yüksek olduğu bir uç noktada (endpoint) problemi bulmak için JProfiler ile profil aldığınızda, belki de sorunun bizim kodumuzdan değil, çağırdığımız harici bir web servisine beklemeden kaynaklandığını görebilirsiniz. Yani I/O bekleme süresi ortaya çıkabilir. Bu tür bütüncül analizler için araçlar entegre görünümler sunar.

Ancak şunu da not etmeli: Bu güçlü profillerin sistem üzerindeki yükü de vardır, bu yüzden genelde test veya pre-prod ortamında kullanılması tercih edilir, prod ortamda kısa süreli veya kontrollü kullanılabilir. JFR gibi üretimde sürekli açık tutulacak kadar hafif değillerdir.

Ticari araçlar aynı zamanda kullanım kolaylığı ve desteğiyle de öne çıkıyor. Örneğin, bir profil oturumunu kaydedip ekip arkadaşınıza gönderebilir ve aynı arayüzde onun da incelemesini sağlayabilirsiniz. Raporlama ve karşılaştırma (profil A vs profil B) gibi özellikleri ile refactoring öncesi ve sonrası performansı karşılaştırmak da mümkündür.

6.4. Diğer Araçlar ve Teknikler

Yukarıda sayılanların dışında da çeşitli özel performans analiz araçları bulunmaktadır. Örneğin, **Java Microbenchmark Harness (JMH)**, mikro düzeyde performans testleri yazmaya yarayan bir çerçevedir. JMH ile belirli bir kod parçasını (örneğin bir algoritma implementasyonu) çeşitli senaryolarda defalarca çalıştırıp nanosaniseviyesinde ölçümler yapabilir, farklı versiyonların hızını istatistiksel olarak karşılaştırabilirsiniz. JMH, JVM'in JIT optimizasyonlarını ve ısınma sürelerini dikkate alarak doğru sonuçlar üretmek üzere tasarlanmıştır. Refactoring sonrası özellikle küçük bir fonksiyonun performansını merak ediyorsanız, JMH ile iki versiyonu kıyaslayabilirsiniz.

Heap dump analyzer araçları (Eclipse MAT – Memory Analyzer Tool gibi) bellek kullanımını derinlemesine analiz etmeye yarar. Bir heap dump alıp, içinde en büyük nesne grafiklerini ve referans zincirlerini inceleyerek bellek sızıntısı kökenini bulmak mümkündür.

Thread dump araçları (jstack gibi komut satırı veya FastThread.io gibi web araçları) anlık olarak thread'lerin durumunu kaydedip analiz eder. Özellikle sunucularda ani tıkanmalar olduğunda thread dump analizleri sorunun kitlenme mi yoğunluk mu olduğunu anlamayı sağlar.

Garbage Collection log analiz araçları (gceasy.io vb.), GC'nin detaylı loglarını alıp anlaşılır raporlara dönüştürür. Bu sayede bellek yönetimi kaynaklı performans sorunları teşhis edilebilir.

Tüm bu araç ve tekniklerin ortak noktası, **ölçmeden iyileştirme yapmama** prensibine hizmet etmeleridir. Pragmatik bir yaklaşımla, önce mevcut performansı sayısal olarak anlamak, sonra gerekliyse optimize etmek en doğrusudur. Kod kalitesi iyileştirmeleri sonrasında da sistemin kabul edilebilir performans bandında kaldığı bu araçlarla doğrulanmalıdır. Çoğu durumda temiz ve iyi tasarlanmış kodun performansı da tatmin edici olur; ancak beklenmedik durumlar için profil araçları geliştiricinin elinin altında olmalıdır. Bu sayede "temiz kod vs hızlı kod" ikilemi yaşamadan, ikisini birlikte elde etmek mümkün olur. Zaten bir yazılım ne kadar kaliteli olursa, performans sorunları da o denli kolay izole edilip çözülebilir, çünkü kod anlaşılırlığı ve modülerliği yüksek olduğundan, dar boğazı bulmak ve düzeltmek basitleşir.

7. Yazılım Kalite Metrikleri ve Ölçütleri

Yazılım sistemlerinin kalitesini nicel olarak değerlendirebilmek için yıllar içinde çeşitli **yazılım metrikleri** geliştirilmiştir. Kod kalitesi denince akla gelen ölçütlerden bazıları nesneye yönelik tasarım metrikleri, kod karmaşıklık metrikleri ve bakım endeksleridir. Bu bölümde öne çıkan bazı metrikler ele alınmıştır: **Chidamber & Kemerer (CK) metrikleri**, **Çevrimsel Karmaşıklık (Cyclomatic Complexity)**, **Halstead Metrikleri** ve **Maintainability Index (Bakım Endeksi)**. Bu metrikler, kod kalitesini tek boyutlu ölçülere indirgeme iddiasında değildir; ancak belirli perspektiflerden (ör. karmaşıklık, bağlılık, büyüklük) kod hakkında bilgi verir ve kokuların/teknik borcun kantitatif takibi için faydalıdır.

7.1. Chidamber & Kemerer (CK) Metrikleri

1994 yılında Shyam Chidamber ve Chris Kemerer tarafından önerilen **CK metrik seti**, nesne yönelimli (object-oriented) tasarımların yapısal özelliklerini ölçmeye yönelik en bilinen metrikler bütünüdür [18]. Toplam altı adet metriği içerir:

- **WMC (Weighted Methods per Class)** – Bir sınıftaki metotların sayısı (ve karmaşıklık ağırlıkları). Basit haliyle bir sınıfın sahip olduğu metotların adedini ifade eder. Çok sayıda metodu olan sınıflar daha fazla sorumluluk alıyordur, bu bir **God Class** belirtisi olabilir. WMC arttıkça sınıfın anlaşılması ve bakımı zorlaşır.

- **DIT (Depth of Inheritance Tree)** – Sınıfın kalıtım ağacındaki derinliği. Bir sınıf kaç seviye üst sınıfa sahip? DIT değeri büyüdükçe, o sınıfın kalıtım yoluyla üstlerinden çok şey devraldığı anlamına gelir. Aşırı derin kalıtım hiyerarşileri, anlaşılması güç tasarımlar doğurabilir.

- **NOC (Number of Children)** – Bir sınıftan türeyen doğrudan alt sınıf sayısı. Bir sınıfın çocuk sayısı fazlaysa, o sınıfın üst düzey bir soyutlama olduğunu ve değiştirildiğinde bir çok alt sınıfı etkileyebileceğini gösterir. Yüksek NOC, potansiyel etki alanının genişliğine işaret eder.

- **CBO (Coupling Between Object Classes)** – Bir sınıfın, başka sınıflarla olan bağlantılarının (coupling) sayısı. Bir sınıf, diğer kaç farklı sınıfı kullanıyor veya ondan kullanılıyor? CBO ne kadar düşükse o sınıf o kadar bağımsız demektir. Yüksek CBO, modüller arası sıkı bağlılık demektir ki bu da değişiklik yapmayı zorlaştırır (birini değiştirince diğerlerini de değiştirmek gerekebilir). **Orthogonality** prensibine aykırı durumlar genelde yüksek CBO ile kendini gösterir.

- **RFC (Response for a Class)** – Bir sınıfın verebileceği toplam farklı mesaj (metot çağrısı) sayısı. Teknik olarak, o sınıftaki tüm metotlar çağrıldığında tetiklenebilecek diğer metotların birleşimi. Bu metrik, bir sınıfın dış dünyaya karşı ne kadar geniş bir etkileşim yüzeyi olduğunu gösterir. Yüksek RFC, sınıfın karmaşık davranışlara sahip olduğunu ve anlaşılmasının zor olabileceğini ima eder.

- **LCOM (Lack of Cohesion in Methods)** – Bir sınıf içindeki metotların ne derece ortak özellikler kullandığını ölçen bir tutarlılık metriğidir. Bir sınıfın metotları aynı alanların (member variables) üzerinde çalışıyorsa uyum (cohesion) yüksektir; tersi durumda LCOM yüksek olur (yani uyumsuzluk fazla). Yüksek LCOM, bir sınıfın farklı farklı sorumlulukları rastgele bir arada tuttuğunu, dolayısıyla **Single Responsibility Principle** ilkesinin ihlal edildiğini gösterir. Bu da genellikle *God Class* kokusuyla ilişkilidir.

CK metrikleri, nesne yönelimli tasarımın farklı boyutlarını sayısal hale getirdiği için akademik araştırmalarda ve pratik araçlarda yaygın kullanılmıştır. Örneğin, bir çalışmada CK metrikleri ile yazılımın bakım maliyeti arasındaki ilişkiye bakılmış ve özellikle CBO ile WMC'nin yüksek olduğu sınıfların hata oranlarının daha yüksek olduğu raporlanmıştır. CK metrikleri hesaplamak için araçlar bulunmaktadır (ör. ckjm aracı [19]). SonarQube ve benzeri platformlar da bu metriklerin bir kısmını otomatik hesaplayıp sunabilir. Bu metrikler, tek başlarına "iyi" veya "kötü" şeklinde yorumlanamaz; ancak eşik değerlerinin aşılması bir alarmdır. Örneğin CBO'su 20'den büyük bir sınıf normalin dışında yüksek bağlı kabul edilebilir, ve tasarımı gözden geçirilebilir. Aynı şekilde DIT çok yüksekse (7-8 seviye gibi) belki kalıtım yapısı aşırı karmaşıktır. CK metrikleri, kod kokularını dolaylı olarak işaret edebildiği için teknik borç yönetiminde de kullanılır – mesela bir sınıfın WMC değeri refactoring sonrası düşmüşse, teknik borç azalmış demektir.

7.2. Çevrimsel Karmaşıklık (Cyclomatic Complexity)

Çevrimsel karmaşıklık (cyclomatic complexity), Thomas J. McCabe tarafından 1976'da önerilen ve bir programın mantıksal karmaşıklığını ölçen klasik bir metriktir [20]. Basitçe, bir yazılım parçasındaki bağımsız yürütme yollarının sayısını ifade eder. Matematiksel olarak bir kontrol akışı grafının döngüsel karmaşıklığı, M = E - N + 2P formülüyle hesaplanır (burada E grafikteki kenar sayısı, N düğüm sayısı, P ise bağlı bileşen sayısıdır). Ancak pratikte, çevrimsel karmaşıklığı hesaplamak için genellikle koşul ifadeleri sayılır: Her if, while, for, case vb. dallanma bir karar noktası ekler ve karmaşıklık +1 artar. Basit bir fonksiyonun temel karmaşıklığı 1'dir (düz bir yol). Her ilave bağımsız dal, ilave bir yol demektir.

Bu metrik, bir fonksiyonun veya metodun ne kadar kompleks olduğunu, dolayısıyla test edilmesi gereken durum sayısını da gösterir. Yaygın bir ifade ile, *"Cyclomatic complexity is defined as the number of linearly independent paths through a program's source code"* ("Çevrimsel karmaşıklık, bir programın kaynak kodu içinden geçen doğrusal olarak bağımsız yolların sayısıdır") şeklinde tanımlanır [21]. Örneğin hiç koşul içermeyen düz bir metot CC=1 değerine sahiptir. İçinde bir if varsa, şart sağlandığında ve sağlanmadığında olmak üzere 2 farklı akış olabilir, CC=2 olur. Birden çok iç içe if-else veya karmaşık switch yapıları CC'yi hızla artırır.

McCabe karmaşıklığı, özellikle bir modülün *test edilebilirliği* ile ilişkilendirilir. CC yüksekse, o modülü kapsamlı test etmek için o kadar çok olası durum (path) vardır. Örneğin CC değeri 10'u aşan metotlar genelde karmaşık kabul edilir ve daha basit parçalara bölünmesi (refactoring ile) önerilir. Pek çok kalite standardında (MISRA C gibi endüstriyel standartlar dahil) bir fonksiyonun CC değerinin belirli bir eşik üstüne çıkmaması istenir (sıklıkla eşik 10 veya 15 gibi belirlenir). Bunu sağlamak için koşulları basitleştirmek, bir kısmını alt metotlara çıkarmak, erken dönüşler kullanmak gibi teknikler uygulanır.

Cyclomatic complexity metriklerini çoğu modern IDE veya araç otomatik hesaplayabilir. Örneğin SonarQube, her fonksiyonun CC değerini gösterir ve "karmaşık fonksiyon" kod kokusu olarak belli bir eşik üstünü işaretler. Bu sayede geliştiriciler hangi parçaların çok dallandığını ve potansiyel olarak yeniden düzenlenmeye ihtiyaç duyduğunu görebilir.

Kısacası, çevrimsel karmaşıklık metrik olarak **kod karmaşıklığının sayısal bir temsilidir**. Kod kalitesi perspektifinde, düşük tutulması arzu edilir çünkü insanın kodu okuma ve anlama yükü de kabaca bu karmaşıklıkla orantılıdır. Clean Code yaklaşımı da fonksiyonların kısa ve tek görevli olmasını öğütler ki bu, CC düşük olmasıyla paraleldir. Elbette her zaman çok düşük CC mümkün olmayabilir, özellikle karmaşık iş kuralları içeren yerlerde; ama en azından bu metrik, "şu modül çok karışık" hissiyatını objektif bir veriye dökerek iyileştirme çabasını tetikleyebilir.

7.3. Halstead Metrikleri

Halstead metrikleri, Maurice Halstead'in 1977'de ortaya attığı *"yazılım bilimsel metrikleri"*dir [22]. Halstead, bir programı metinsel bir dizi olarak ele alıp operatörler ve operantlar üzerinden ölçüler tanımlamıştır. Temel kavramlar:

- **n1** = programdaki farklı operatör sayısı (örneğin +, -, if, while, fonksiyon çağrısı gibi işlemler).
- **n2** = programdaki farklı operand sayısı (değerler, değişken isimleri, sabitler vs.).
- **N1** = toplam operatör sayısı (tekrarlarla birlikte, programdaki tüm operatör sembollerinin sayısı).
- **N2** = toplam operand sayısı (tekrarlarla birlikte, programdaki tüm operand sembollerinin sayısı).

Bu temel sayımlar üzerinden Halstead çeşitli türetilmiş metrikler tanımlar:

- **Program Vocabulary (Kelime dağarcığı)** = n1 + n2. Yani programda kullanılan farklı sembollerin toplam sayısı.
- **Program Length (Uzunluk)** = N1 + N2. Programın toplam sembolik uzunluğu.
- **Calculated Program Length (Tahmini uzunluk)** = n1 * log2(n1) + n2 * log2(n2). (Halstead, bir programın "beklenen" uzunluğunu bu formülle hesaplar).
- **Volume (Hacim)** = N * log2(n1+n2) (bazı kaynaklarda Volume = N * log2(Vocabulary)). Bu metrik, programın bilgi içeriğini bit cinsinden ifade etmeye çalışır. Hacim ne kadar yüksekse, programın içerdiği bilginin (veya zihinsel yükün) o kadar fazla olduğunu söyleyebiliriz.
- **Difficulty (Zorluk)** = (n1/2) * (N2/n2). Bu, programın yazılması veya anlaşılmasının zorluğunu göstermeyi amaçlar. Operatör çeşitliliğinin yarısıyla, operant kullanımlarının çeşitliliği oranını çarparak hesaplanır.
- **Effort (Efor)** = Volume * Difficulty. Bu da programı yazmak için gereken zihinsel efor birimi olarak tanımlanır.

Halstead bu metriklerden yola çıkarak hata tahmini vs. de yapmaya çalışmıştır (örneğin Effort'un belirli bir katsayesi insani efor saatine karşılık gelir gibi varsayımlarla). Günümüzde Halstead metrikleri birebir projeksiyonlar için kullanılmasa da, kodun **metinsel karmaşıklığını** nicelleştiren ilginç araçlar olarak değerlendirilir. Özellikle Volume ve Difficulty, bir kod parçasının ne kadar "yoğun" olduğunu gösterir. Örneğin aynı işlevi yapan iki koddan biri çok daha uzun ve farklı semboller içeriyorsa, Halstead hacmi daha yüksek çıkacaktır.

Halstead metrikleri bazı araçlarda ve raporlarda sunulur. Örneğin SonarQube, *Complexity* olarak McCabe karmaşıklığını verirken, *Cognitive Complexity* gibi bir kavramı da raporlar; bu kavram Halstead'e doğrudan dayanmasa da kodun anlaşılma zorluğunu modele eden bir yaklaşımdır. Halstead'in Difficulty metriği de benzer bir amaca sahiptir.

Pratikte, Halstead metriklerinin en değerli tarafı, bir modülün içerdiği bilgi miktarını Volume ile ifade edebilmesidir. Bir modül ne kadar büyük bir vocabulary ve uzunluk içeriyorsa, o kadar çok kavram ve detay barındırıyor demektir. Bu da haliyle bakımını zorlaştırır. Bu nedenle Halstead hacmi çok büyük olan modüller tespit edilip belki ikiye bölünerek (refactoring ile) hacimleri küçültülebilir.

Eleştirel bir bakışla, Halstead formülleri bazı teorik varsayımlara dayanır ve her durumda pratik gerçekliği yansıtmayabilir. Örneğin bir algoritma çok karmaşık matematiksel işlemler içeriyorsa n1 ve n2 değerleri artabilir; ama belki de mantık sadedir. Yine de, Halstead metrikleri yazılım metriği literatürünün klasiklerindendir ve kod kalitesine dair çok yönlü bir perspektif sunar: **Uzunluk, hacim, zorluk, efor** gibi boyutlarda ölçüm yapar.

7.4. Maintainability Index (Bakım Endeksi)

Maintainability Index (MI), bir yazılım parçasının bakım yapılabilirliğini tek bir sayıyla ifade etmeyi amaçlayan bir endekstir. 1990'ların başında Paul Oman ve Jack Hagemeister tarafından ortaya atılmıştır [23]. MI, birden fazla metriği bir araya getirerek 0 ile ~100 arasında bir skor üretir; yüksek değer daha kolay bakım demektir. Klasik formül, Halstead Hacmi (V), McCabe karmaşıklığı (CC) ve yorum satırı yüzdesi (LOC-related) gibi bileşenleri içerir. Orijinal formülün bir versiyonu şöyledir:

MI = 171 - 5.2 * log2(V) - 0.23 * CC - 16.2 * log2(LOC) + 50 * sin(sqrt(2.46 * perComment))

Bu formül farklı araçlarda biraz değişmiş halleriyle uygulanmıştır. Basitleştirilmiş hali genellikle:

MI = 171 - 5.2 * log(V) - 0.23 * CC - 16.2 * log(LOC) (yorumlar dahil değil)

veya bazı araçlarda 100 üzerinden normalize edilerek verilir. Microsoft Visual Studio gibi araçlar, MI değerini kod analizinde göstermişlerdir (2010'lar civarında). Genellikle MI değeri 85'in üzeriyse "mükemmel", 65-85 arası "iyi", 65 altı "zayıf" gibi kategoriler belirtilebilir [24].

Maintainability Index, bir bakıma yukarıda tartışılan metriklerin bir sentezini sunar. Mesela LOC (satır sayısı) arttıkça, CC arttıkça, Halstead hacmi arttıkça MI düşer (yani bakım zorlaşır). Kodun anlaşılabilirliğine etki eden bu temel unsurlar tek skora yedirilmiştir. Yorum satırları bir miktar pozitif katkı yapar (yorum oranı yüksekse MI bir parça artar, çünkü anlaşılmayı kolaylaştırdığı varsayılır).

MI kavramı, yöneticilere veya metriklere aşina olmayan kişilere kalite durumu sunmak için cazip olsa da, çok tartışmalı yanları da vardır. Tek sayıya indirgeme her zaman risklidir. Bazı eleştirmenler, MI değerinin yanıltıcı olabileceğini, zira formül parametrelerinin rastgele seçilmiş olduğunu iddia etmişlerdir [24]. Örneğin R. Niedermayr bir blog yazısında MI'ın pratik faydasını sorgulayarak bazı durumlarda yüksek MI değerine rağmen kodun kötü olabildiğini belirtmiştir [24]. Bununla birlikte, MI halen pek çok araçta raporlanan bir metriktir ve özellikle **tarihsel takip** için kullanışlı olabilir. Yani bir projede refactoring çabaları sonrası MI değerinin yükselmesi genelde olumlu bir göstergedir (farklı metriklerin de iyileştiğini gösterir zaten).

SonarQube gibi platformlar MI kullanmak yerine kendi türev maintainability hesaplamalarını yapar. Örneğin SonarQube, teknik borç dakikasına ve toplam kod geliştirme süresine dayalı bir **Maintainability Rating** harf notu hesaplar [15]. Bu, MI ile benzer amaçlıdır ama farklı hesaplanır. Sonuçta hepsi, kodun bakım kolaylığına dair bir gösterge sunmayı hedefler.

Özetle, Maintainability Index kod kalitesini tek bir skorla özetleme girişimidir. İçinde Halstead hacmi, cyclomatic complexity ve LOC barındırdığı için, kodun hem boyutunu hem karmaşıklığını birlikte değerlendirir. Eğer bir modülde MI çok düşükse, bu tipik olarak o modülün aşırı büyük, kompleks ve zor anlaşılır olduğunun alarmıdır. Refactoring yaparak bu modül bölünebilir veya basitleştirilebilir; başarılı olunursa MI skorunun yükseldiği görülür. Bu nedenle, MI veya benzeri endeksler teknik borç takibinde kullanılır: Örneğin zaman içinde MI düşüyorsa, projenin bakımı zorlaşıyor demektir, belki borç birikiyor anlamına gelir. Tam tersi, sürekli iyileştirmelerle MI değerini yüksek tutmak, projenin sağlıklı evrimini destekler.

Ancak, MI tek başına her şeyi söylemez; mutlaka detay metriklerle ve uzman değerlendirmesiyle birlikte ele alınmalıdır. Nitekim, “Maintainability Index’i körü körüne takip etmek yerine, alt metriklere odaklanmak ve kodun somut yapısına bakmak gerekir” şeklinde öneriler de literatürde yer almaktadır [24]. Bu uyarıyla birlikte, MI yine de literatür taramamızın kapsaması gereken önemli bir kavramdır, çünkü kod kalitesinin sayısallaştırılmasında tarihsel bir yere sahiptir ve günümüz araçlarının atası sayılır.

Sonuç

Bu literatür taramasında **kod kalitesi** kavramına pragmatik bir gözle bakılarak, yazılım geliştirmede kaliteyi etkileyen unsurlar, problemler ve çözümler çok yönlü olarak incelenmiştir. **Pragmatik Programcı** ilkelerinin (DRY, Ortogonallik, YAGNI, Tracer Bullets vb.) özünde, yazılımcılara temiz, esnek ve gereksiz yüklerden arınmış bir kod üssü oluşturmayı öğütlediğini gördük. Bu prensiplerin uygulanmaması sonucu biriken **teknik borç**, kısa vadeli kazanımlar uğruna uzun vadeli bedeller getirmektedir – Ward Cunningham'ın borç metaforu ve "faiz" analojisi bunu güçlü bir şekilde anlatmaktadır [2]. Teknik borcun somut belirtileri çoğunlukla **kod kokuları** şeklinde karşımıza çıkar. Fowler ve Beck'in tanımladığı kod kokuları, yazılımda derin tasarım sorunlarının yüzeydeki işaretleridir ve "kötü kokuyorsa değiştirmek gerekir" anlayışı yaygındır. Kod kokularını gidermenin yolu ise planlı veya fırsatçı şekilde **refactoring** uygulamaktır. Martin Fowler'ın sistematik refactoring kataloğu [5][8] ve testlerin koruyucu desteğiyle [9], mevcut kod yapısını iyileştirmek mümkün ve gereklidir.

Java ekosisteminde kod kalitesini yönetmek için pek çok araç bulunduğunu ele aldık. **Statik analiz araçları** (PMD, Checkstyle, SpotBugs gibi) hataları, stil ihlallerini ve kokuları otomatik saptamada büyük kolaylık sağlar. Örneğin PMD "unused variable" gibi hataları yakalarken, SpotBugs null pointer risklerini uyarabilir – bunlar erken dönemde kaliteyi yükselten önlemlerdir. **SonarQube** gibi platformlar ise bu analizleri entegre edip teknik borcu dakika cinsinden ölçerek yönetime raporlayabilecek düzeye taşır. Ayrıca, araştırma kökenli **JDeodorant** aracı ve benzerleri, kod kokularını tespit edip otomatik düzeltme önerileri sunarak refactoring sürecini yarı otomatik hale getirmektedir. Geliştiricilerin bu araçları pragmatik bir şekilde kullanarak – yani araçların güçlü yanlarını insan aklıyla birleştirerek – en iyi sonuca ulaşabileceği vurgulanmıştır [16].

Kod kalitesini iyileştirme uğraşında, **performans** boyutunun da izlenmesi gerektiği unutulmamıştır. "Önce çalışır yap, sonra optimize et" prensibi geçerli olmakla birlikte, yaptığımız değişikliklerin istenmeyen performans sonuçları doğurmadığını kontrol etmek gerekir. Bu amaçla incelediğimiz **profiling araçları** (JFR/JMC, VisualVM, JProfiler vb.), hem potansiyel dar boğazları ortaya çıkarmak hem de yapılan refactoring'lerin performansa etkisini ölçmek için kullanışlıdır. JFR gibi araçlar sayesinde üretimde dahi düşük maliyetle veri toplayıp, gerektiğinde analiz yapabiliriz – böylece "prematüre optimizasyon" hatasına düşmeden, gerçek verilere dayalı iyileştirmeler yapabiliriz. Yine de, temiz kod ile yüksek performans çoğu zaman çelişkili olmadığından, asıl odak noktası kodun doğru ve temiz olması olmalıdır; performans ise ölçümlere dayalı olarak gerektiğinde ele alınmalıdır.

Son olarak, yazılım kalitesini sayısal olarak ifade etmeye yarayan **metrikler** değerlendirildi. CK metrikleri (WMC, CBO, LCOM vb.) bize nesne yönelimli tasarımın yapısal kalitesi hakkında ipuçları verirken, McCabe karmaşıklık ölçütü fonksiyonların ne denli karmaşık olduğunu gösterir. **Halstead**'in yazılım biliminden türettiği hacim ve zorluk ölçütleri, kodun içerdiği bilgi miktarını ve zihinsel efor gereksinimini nicelleştirir. **Maintainability Index** ise bu çeşitli boyutları tek bir endekste harmanlayarak bakım kolaylığına dair bir genel bakış sunar. Bu metriklerin hiçbiri kaliteyi tam kapsayıcı şekilde ölçemese de, birlikte ele alındığında bir projenin güçlü ve zayıf yönlerini objektif olarak

ortaya koyabilirler. Özellikle zaman içindeki metrik trendleri, teknik borcun birikip birikmediğini veya refactoring çabalarının meyve verip vermediğini takip etmeyi mümkün kılar. Örneğin, düzenli refactoring yapılan bir projede ortalama cyclomatic complexity düşer, CBO azalır, LCOM iyileşir, maintainability index yükselir. Bu değişimler, kod kalitesinin sürdürülebilirliğinin göstergeleridir.

Sonuç olarak, kaliteli kod yazımı ve mevcut kodun kaliteli tutulması, yazılım mühendisliğinde süreklilik arz eden bir çabadır. Pragmatik programcı yaklaşımı bize bu konuda hem zihniyet hem araç anlamında rehberlik etmektedir. "Broken window" metaforundan hareketle, küçük problemlerin bile ihmal edilmeden düzeltilmesi, **temiz ve sağlıklı bir kod üssü** için şarttır. Bunu başarmak için yazılım ekipleri uygun ilkeleri (DRY, YAGNI, vs.) süreçlerine dahil etmeli; statik analiz ve test otomasyonu gibi araçlarla kaliteyi sürekli kontrol etmeli; teknik borcu görünür kılarak yönetmeli; belirgin kokuları refactoring ile gidermeli; önemli metrikleri izleyerek alarm veren bölgeleri tespit etmelidir. Bütün bu aktivitelerin amacı, yazılımı hem bugünün ihtiyaçlarına cevap veren hem de gelecekte değişime direnç göstermeyen canlı bir organizma gibi yönetebilmektir.

Unutulmamalıdır ki, **kod kalitesi doğrudan geliştirme hızıyla ilişkilidir**: Kısa vadede hızlı iş çıkarma adına kaliteyi feda etmek, uzun vadede gelişmeyi yavaşlatır çünkü biriken borçlar insanın ayağına dolanır. Aksine, temiz ve kaliteli kod ile ilerlemek belki başlangıçta biraz disiplin gerektirir ama uzun vadede ivmeyi artırır. Robert Martin'in dediği gibi "The only way to go fast, is to go well." [4] Bu literatür taramasında incelenen yöntem ve araçlar da, "iyi gitmek" için elimizde bulunan yardımcılar olarak değerlendirilmeli ve yazılım süreçlerimize entegre edilmelidir.

Kaynakça

[1] Andrew Hunt, David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999. (Pragmatik programcılık yaklaşımı, Broken Window metaforu ve DRY, Orthogonality gibi ilkelerin tanımı)

[2] Jean-Louis Letouzey, Declan Whelan. "Introduction to the Technical Debt Concept." *Agile Alliance White Paper*, 2016. (Ward Cunningham'ın teknik borç benzetmesini aktaran ve teknik borç kavramını açıklayan beyaz kitap)

[3] Jaime González García. "Tracer bullets." *Barbarian Meets Coding Blog*, 12 Jul 2023. (Tracer Bullet tekniğini Pragmatic Programmer bağlamında açıklayan blog yazısı)

[4] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017. (Temiz kodun hız ile ilişkisini ele alan, "The only way to go fast is to go well" ifadesinin kaynağı)

[5] Martin Fowler, Kent Beck, et al. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. (Kod kokuları listesi ve klasik refactoring kataloğu; bad smell kavramının tanımı)

[6] Min Zhang, Tracy Hall, et al. "Do Bad Smells Indicate 'Trouble' in Code?" *DEFECTS '08: Workshop on Defects in Large Software Systems*, ACM, 2008. (Kod kokularının yazılım hatalarıyla ilişkisini inceleyen çalışma; Fowler'ın bad smell tanımını içermekte)

[7] Nikolaos Tsantalis, Theodoros Chaikalis, Alexander Chatzigeorgiou. "Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells." *25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018. (JDeodorant aracının geliştirilmesinden çıkarılan dersler, kod kokusu tespiti ve giderme üzerine derleme makalesi)

[8] Martin Fowler. "Definition of Refactoring." *martinfowler.com Bliki*, Sep 1, 2004. (Refactoring kavramının Fowler tarafından tanımlandığı çevrimiçi not; iç yapıyı değiştirmeden dış davranışı koruma vurgusu)

[9] Martin Fowler. *Refactoring: Improving the Design of Existing Code (2nd Edition)*. Addison-Wesley, 2018. (Refactoring kitabının güncellenmiş baskısı; refactoring öncesi testlerin gerekliliği ve güvenli geri bildirim döngüsü vurgusu)

[10] Martin Fowler. "OpportunisticRefactoring." *martinfowler.com Bliki*, Jul 1, 2018. (Refactoring'i fırsatçı şekilde, günlük geliştirme esnasında yapmanın önemini anlatan çevrimiçi makale)

[11] PMD Official Documentation. "Introduction and Features of PMD." pmd.github.io, 2023. (PMD aracının tanıtımı, desteklediği diller ve yakaladığı yaygın programlama hataları – örneğin kullanılmayan değişkenler, boş catch blokları)

[12] Checkstyle Official Website. "Checkstyle – A development tool to help programmers write Java code that adheres to a coding standard." checkstyle.sourceforge.io,

v10.x Docs, 2023. (Checkstyle aracının amacı ve çalışma prensibini tanımlayan resmi açıklama)

[13] University of Maryland, FindBugs Project. “FindBugs – Find Bugs in Java Programs (Official website).” findbugs.sourceforge.net, 2015. (FindBugs aracının tanıtımı, statik analizle Java kodunda bug yakalama hedefine dair bilgi)

[14] Nathaniel Ayewah, William Pugh, et al. “The Google FindBugs Fixit.” *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA ’10)*, ACM, 2010. (Google’ın geniş kod tabanında FindBugs uyarılarını ele alma deneyimi ve yüzlerce hatayı düzeltme hikayesi)

[15] SonarSource Documentation. “Metric Definitions – Maintainability.” *SonarQube 10.1 Documentation*, 2023. (SonarQube’da maintainability ile ilgili metriklerin (code smell, technical debt, maintainability rating) resmi tanımları ve hesaplama yöntemleri)

[16] Rahmon A. Badru, A. O. Ogunlade, I. O. Adewumi. “Overview of Bad Code Smells in Software Development and Researches.” arXiv preprint arXiv:2508.01944, Aug 2025. (Kod kokularının tespiti için SonarQube, PMD, JDeodorant gibi araçların kullanımını ve çoklu araç yaklaşımının önemini ele alan güncel derleme çalışması)

[17] Donald E. Knuth. “Structured Programming with Go To Statements.” *Computing Surveys*, vol. 6, no. 4, 1974. (Knuth’un “premature optimization is the root of all evil” sözünün geçtiği klasik makale)

[18] Shyam R. Chidamber, Chris F. Kemerer. “A Metrics Suite for Object Oriented Design.” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, 1994. (CK metriklerinin orijinal makalesi; WMC, DIT, CBO, RFC, NOC, LCOM tanımları ve nesne yönelimli tasarım ölçümü)

[19] Diomidis Spinellis. *ckjm – Chidamber & Kemerer Java Metrics*. Version 1.9, 2020. (CK metric hesaplayıcı aracı ckjm’in dokümantasyonu, kullanım örnekleri; CK metriklerinin pratik hesaplanması)

[20] Thomas J. McCabe. “A Complexity Measure.” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976. (McCabe’in cyclomatic complexity metrikini tanımladığı orijinal makale; bağımsız program yolları ve yazılım karmaşıklığı ilişkisi)

[21] Sebastian Feldmann. “Hello, my name is ‘if’ – On Complexity and Conditionals.” *International PHP Conference Blog*, May 2019. (Cyclomatic complexity’nin “linearly independent paths through a program” şeklindeki tanımını da içeren açıklama; programlama blogu yazısı)

[22] Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977. (Halstead’in yazılım metrikleri kitabı; operatör/operand sayıları, hacim, zorluk ve efor metrikleri)

[23] Paul Oman, Jack Hagemeister. “Metrics for Assessing Maintainability of Software.” *Proceedings of International Conference on Software Maintenance (ICSM)*, IEEE, 1992. (Maintainability Index’in tanıtıldığı çalışma; Halstead, McCabe ve LOC bileşenleriyle bakım endeksi formülasyonu)

[24] Rainer Niedermayr. “Why we don’t use the Software Maintainability Index.” *Teamscale Engineering Blog*, Feb 2016. (Maintainability Index’in kısıtlarını ve pratikte neden yanıltıcı olabileceğini tartışan makale; MI formülü ve eleştirisi)