Kaynak Kodlardaki Kötü Kokuların Otomatik Tespiti için Eclipse Eklenti Önerisi

Melih Altıntaş[1,2] ve Ebru Akçapınar Sezer[1]

[1] Bilgisayar Mühendisliği Bölümü, Hacettepe Üniversitesi, Ankara, Türkiye
[2] Yazılım Mühendisliği Müdürlüğü, UGES, Aselsan, Ankara, Türkiye
mealtintas@aselsan.com.tr, ebru@hacettepe.edu.tr

Özet Kaynak kodlarda yer alan kötü kokular; geliştirilen uygulamanın doğru çalışmasına engel teşkil etmeyen ancak kod kalitesini azaltarak bakım ve anlaşılabilirliğini zorlaştıran ve bu nedenle yeniden düzenlenmesi gereken kod parçalarıdır. Bunlar bir sınıfın genelinde ya da sınıfın belli bir metodunda gözlenebilir. Kötü kokuların gözle tespit edilmesi, projelerin büyüklüğü arttıkça, zaman ve iş gücü maliyetleri açısından ihmal edilme mümkünlüğü artan bir aşamadır. Bunlar tasarım aşamasındaki hatalardan kaynaklanabileceği gibi, tasarımın koda çevrimi aşamasında geliştiricinin tercihlerinden de kaynaklanabilir. Bu bildiride Java kaynak kodlarında yer alan kötü kokuların otomatik olarak tespit edilmesi, yazılımcı ve bakımcılara raporlanmasını sağlayan bir Eclipse eklentisi sunulmaktadır. Böylece yazılımcı ve bakımcılar sürekli olarak yazılımın kalitesini daha somut verilerle değerlendirebilecek, hataya neden olabilecek modülleri önceden farkedip yeniden düzenleyebileceklerdir. Bunun sonucunda ortaya daha kaliteli, bakımı ve test edilebilirliği daha kolay yazılımlar ortaya çıkacaktır. Geliştirilen eklenti, hata kestiriminde kullanılan veri kümeleri üzerinde denenerek, yazılım hatası ile kötü kokular arasındaki birlikte yer alma ilişkisi istatistiksel olarak sunulmuştur. Elde edilen sonuçlara göre kötü kokuların varlığı yazılım hatalarından bağımsızdır, ancak mevcut hatalar kötü kokularla istatistiksel olarak ilişkilidir.

Anahtar Kelimeler: Kötü koku · Kötü koku tespit eklentisi · Java · Yazılım metrikleri · Yazılım kalitesi

Eclipse Plugin for Automatic Detection of Code Smells in Source Codes

Abstract. Code smells in source codes are code fragments; that do not prevent the functionality of the developed application, but which reduce code quality, make code maintenance and understandability difficult and require refactoring. Those types of smells could be found in a class as a whole or in a specific method of a class. Detecting those code smells by manual reviewing is a process that could increase the probability of

CEUR-WS.org/Vol-2201/UYMS_YTM_2018_paper_6.pdf

unintentional omission in terms of the requirement of time, budget, and manpower as the project grows. Code smells can be caused by errors in the design phase as well as by the developer's preferences in the design to code conversion phase. In this article, we will introduce an Eclipse plugin that enables automatic detection of code smells in Java source code and presents the detected code smells to developers and maintainers. In this way, the software developers and maintainers can continuously evaluate the quality of the software with realistic values, recognize and refactor the modules that could cause a bug. This provides better quality, easier maintainability and effective testability in software. The developed plugin is tested on the data sets used in fault estimation, and statistical correlation between software fault and code smells is presented. According to results, existence of code smells is unrelated with the software faults. However, existing faults are statistically related with code smells.

Keywords: Code smells · Code smell detection tool · Java · Software metrics · Software quality

1 Giriş

Dünyada üretilen ya da geliştirilen hiçbir ürün kusursuz olarak nitelenemez; çünkü kusursuzluğun tanımı her gelişme ile birlikte yeniden yapılır. Ancak kusur tanımı yapabilmek daha kolaydır. Bu bağlamda, gerek kaynak kodun kendisi gerekse kaynak kodu üretirken kullanılan kütüphaneler elle geliştirilmiş olduğundan kusurların ve hataların oluşumuna açıktır. Bu çalışma ile amaçlanan, kaynak kodlar için kusur olarak tanımlanmış kötü koku olarak adlandırılan kodlama biçimlerini tespit etmektir.
Tespiti yapılan kaynak kodlarda yer alan kötü kokular; geliştirilen uygulamanın doğru çalışmasına engel teşkil etmez, bu nedenle beyaz kutu (white box) ve siyah kutu (black box) testleri yapılarak tespit edilemezler. Fonksiyonel hatalar olmasalar da bunlar kodun kalitesi üzerinde çok etkilidir; çünkü kötü kokular kodun ne zaman hangi bölümünün yeniden düzenlenmesi (refactoring) gerektiğinin cevabıdır. Eğer iyi yönetilmezlerse kodun okunabilirliğini, yönetimini ve bakımını zorlaştırırlar. Özellikle büyük projelerde zaman geçtikçe birikerek hataya neden olabilirler. Bu nedenle kaliteli yazılımda kötü kokular olmamalı ve süreklileştirilen gözden geçirmeler ile kaynak kod bilinen kötü kokulardan temizlenmelidir.
Bu çalışmada Java diliyle geliştirilmiş projelerdeki kötü kokuların bulunması amaçlanmış, yazılımcı ve bakım sorumlularına kullanıma hazır bir yardımcı araç olarak Eclipse platformu üzerinden sunulmuştur. Java programlama dili üzerinde çalışma yapılmasının nedeni, en yaygın kullanılan programlama dillerinden biri olmasıdır.[31] Kötü koku tespitinin geliştirme ortamına entegre bir hizmet olarak kurgulanmasının nedeni ise kaynak kod kalitesinin anlık değerlendirilebilme gerekliliğidir. Böylece geliştirme sırasında anlık gözden geçirme ve yeniden düzenleme yapmak mümkün olacaktır. Java kullanan yazılımcıların %33'lük bir kısmının geliştirme ortamı olarak Eclipse kullanması bizim eklentimizi Eclipse

üzerinden sunmamızın temel nedenidir.[32]. Eklentimiz ile birlikte büyük projelerdeki bakım ve test gibi maliyetli işlemlerin süresini kısaltarak iş gücü, zaman ve maliyet kazancı yaratmayı hedefliyoruz.

2 Alanyazın Özeti

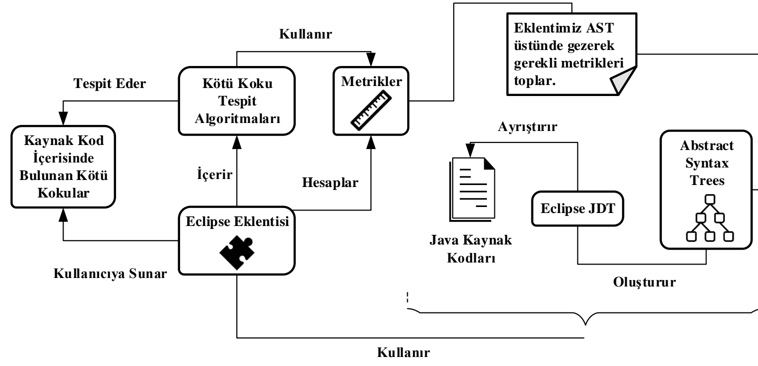
Kötü koku (code smells) kavramı, ilk olarak Martin Fowler'ın kitabında [1] Kent Beck tarafından sistem kodunda sorun yaratabilecek belirtiler olarak tanımlanmıştır. Kitapta **22** farklı kötü koku tanımlanmıştır. Kitap resmî olmayan tanımlar yapmış, tespit ve düzeltme için otomatik bir yol önermemiştir. Daha sonra Mäntylä [2] ve Wake [3] bunların sınıflandırılması konusunda çalışmalar yapmışlardır. Kötü kokular sınıflandırılmış ancak bunların tespiti hâlâ gözle kontrole bırakılmıştır.
Marinescu tespit sırasında çok zaman harcanması, tespitinin tekrarlanamaması ve ölçeklenememesi nedeniyle kötü kokuların tespiti için *IPLASMA* aracını geliştirerek metrik tabanlı bir yaklaşım önermiştir.[4]. Daha sonra bu çalışmadaki metrik tabanlı yaklaşım geliştirilerek *"Object-Oriented Metrics in Practice"* adıyla kitap hâline getirilmiş ve bu alandaki temel taş olarak yerini almıştır. [5] Kitapta 3 ayrı sınıflama yapılarak toplam **11** adet kötü koku tanımlanmıştır. Bu çalışmayla birlikte kaynak kod içerisinde yer alan kötü kokuları, otomatik olarak tespit edebilecek metrik tabanlı kurallar da açık biçimde ortaya çıkmıştır.
Metrik tabanlı kuralların ortaya çıkmasından sonra *IPLASMA*, *JDeodorant*, *PMD*, *StenchBlossom* gibi birçok otomatik tespit aracı geliştirilmiştir.
IPLASMA [6] Literatürde tanımlanan ve bizim bu çalışmada üstünde durduğumuz **10** adet kötü kokunun tamamını bulmaktadır, ancak uygulama bağımsız şekilde ele alınmıştır. Başka bir ifade ile, herhangi bir geliştirme platformu ile entegrasyonu yoktur ve bu nedenle kod geliştirilirken eş zamanlı olarak kötü kokuların tespiti mümkün değildir.
JDeodorant [7] [8] [9] Java dilinde yazılmış kaynak kod üzerinden toplam **5** çeşit (*Feature Envy, Type Checking, Long Method, God Class, Duplicated Code*) kötü koku tespiti yapabilen bir Eclipse eklentisidir. Geliştirme ortamına entegrasyonu olması açısından başarılı bir çalışma olsa bile bulduğu kötü koku sayısının az olması dezavantajıdır.
StenchBlossom [10] Java dilinde yazılmış kaynak kod üzerinden **8** adet (*Data Clumps, Feature Envy, InstanceOf, Long Method, Large Class, Message Chain, Switch Statement, Typecast*) kötü koku tespiti yapabilen bir Eclipse eklentisidir. Geliştirme ortamına entegrasyonu olması açısından başarılı bir çalışma olsa bile bulduğu kötü koku sayısının az olması dezavantajıdır.
PMD [11] Java dilinde yazılmış kaynak kod üzerinden statik analiz yaparak birçok olası hata bulmaktadır. Birçok geliştirme ortamına entegrasyonu vardır. Ancak kötü koku tespitine çok fazla yer vermemiştir, *Large Class, Long Method, Long Parameter List* olmak üzere **3** adet kötü koku tespit etmektedir.

Bizim çalışmamız, aynı amaç doğrultusunda, kaynak kod içerisinde yer alan kötü kokuların tespiti için geliştirilmiştir. Ancak geliştirdiğimiz eklenti *IPLASMA*

aracı hariç diğer araçlardan daha fazla kötü koku tespiti yapabilmektedir. Eklentimizin *IPLASMA*'ya göre avantajı ise kötü koku tespitini bağımsız bir uygulama olarak yapmaması, geliştirme ortamı üzerinden sunmasıdır. Bu sayede yazılımcılar ürünü geliştirirken sürekli olarak kalite yönetimi yapabileceklerdir.

3 Kötü Kokuların Tespitinin Gerçekleştirimi

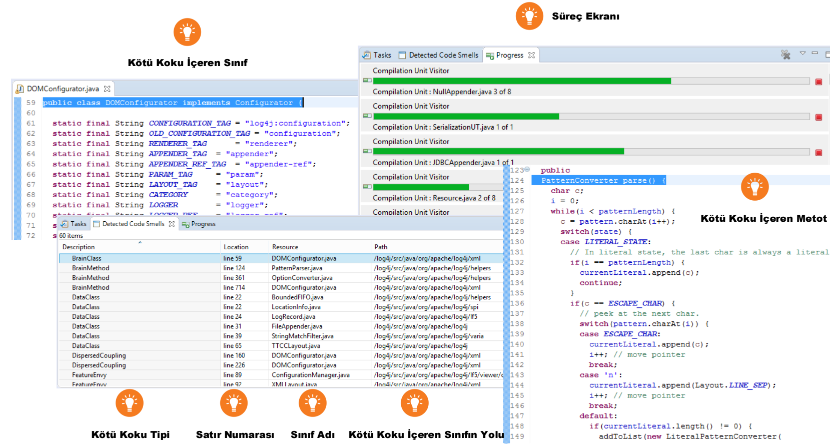
Eklentimiz Lanza ve Marinescu'nun kitabında [5] tanımlanmış, *God Class, Feature Envy, Data Class, Brain Method, Brain Class, Intensive Coupling, Dispersed Coupling, Shotgun Surgery, Refused Parent Bequest, Tradition Breaker* olmak üzere toplam **10** adet kötü kokuyu yine aynı kitapta tanımı verilmiş **23** adet metrik kullanarak otomatik olarak tespit edebilme yeteneğine sahiptir.



Şekil 1: Kötü kokuların tespitinin gerçekleştirimi

Tespit gerçekleştirimimiz Şekil 1'de özetlenmiştir. Gerçekleştirim sırasında metrikleri toplamak için herhangi bir üçüncü parti kütüphane kullanılmamış, tüm metrikler Eclipse platformunun sunduğu JDT (Java Development Tools) kütüphanesiyle bu çalışmaya özel kodlar üzerinden toplanmış, doğrulukları birim testleri ile kontrol edilmiştir.
Eclipse JDT, çalışma ortamındaki projelerin kaynak kodlarından *Soyut Sözdizim Ağaçları (Abstract syntax trees)* oluşturabilme yeteneğine sahiptir ve oluşan bu ağaçlar yine Eclipse tarafından sunulan *ASTVisitor* sınıfı sayesinde gezilebilmektedir. Eklentide her bir metrik değerinin hesaplanması için, ASTVisitor sınıfından hesaplama yapacağımız sınıflar türetilerek *visit* ve *endVisit* metotları yeniden gerçeklendi. Hesaplanan bu metrikler, kötü kokuların tespiti için kural tabanlı algoritmalara girdi olarak kullanıldı.
Tespit edilen her bir kötü koku için Eclipse platformu üzerinde bunları gösteren işaretçi (marker) oluşturuldu. Oluşturulan her bir işaretçi, yine bizim geliştirdiğimiz yeni bir ekran (view) üzerinden yazılımcı ve bakımcılara sunuldu. Bu

ekran üzerinden yazılımcılar kötü koku barındıran sınıf ve metotları görerek, ilgili kötü kokuyu içeren sınıf veya metoda ulaşıp gerekli düzenlemeleri yapabilir hâle geldiler.
Gerçekleştirilen eklentinin kaynak kodları Github hesabımız üzerinden erişime ve kullanıma açılmıştır. [33] Daha önce yapılmış çalışmalar bizim hesapladığımız kadar metrik hesaplamadığı için ve bizim kadar hesaplama yapan çalışmalar da, kötü kokuları sadece tespit etmekle kalıp metrik değerlerini kullanıcılara sunmadığı için; hesaplanan metriklerin doğruluğu başka bir çalışmayla karşılaştırılamamış ve bu nedenle çok yoğun birim testlerle birlikte her biri birçok senaryo ile doğrulanmıştır. Kötü kokular da bu metriklerin bir formulasyonu olduğundan, otomatik olarak doğrulukları ispatlanmıştır. Kaynak kodlarımızı Github [33] üzerinden paylaşarak, kötü koku tespitlerini otomatik olarak yapabilen bir eklentiyle birlikte, doğrulukları ayrıntılı irdelenmiş metrik hesaplayıcı sınıflarımızda erişime açılmıştır. Sonuç olarak ortaya çıkan eklentimizin görüntüsü Şekil 2'de verilmiştir.



Şekil 2: Eclipse eklenti görüntüsü

3.1 Hesaplanan Metrikler

Kötü kokuların tespitinin gerçekleştirimi için hesaplanması gereken tüm metrikler Lanza ve Marinescu'nun kitabında [5] tanımlanmış, Tablo 1'de kısa tanımları verilmiştir. Tanımı verilen tüm metrikler bizim kendi yazdığımız Java sınıfları üzerinden hesaplanmış, herhangi bir üçüncü parti kütüphane kullanılmamıştır.

Tablo 1: Kötü kokuların tespiti için hesaplanan metrikler

Metrik Adı	Metrik Tanımı
Average Method Weight (AMW)	Ölçüm yapılacak sınıfın içerdiği metotların karmaşıklıkları toplamının, sınıfın içerdiği toplam metot sayısına oranıdır.[5]

Access To Foreign Data (ATFD)	Ölçüm yapılacak metot veya sınıfın, diğer sınıflardan (sınıfın ata sınıfları hariç) erişimci metotlar (getter-/setter) üzerinden veya direkt olarak eriştiği nitelik (attribute) sayısıdır.[5]
Base Class Overriding Ratio (BOvR)	Ölçüm yapılacak sınıfta geçersiz kılınan (overridden) metotların sayısının, sınıf içerisindeki tüm metot sayısına oranıdır.[5]
Base Class Usage Ratio (BUR)	Ölçüm yapılacak sınıfta kalıtımdan gelen nitelik ve metotların kullanım oranıdır.[5]
Changing Classes (CC)	Ölçüm yapılacak metodu çağıran metotların ait oldukları farklı sınıfların sayısıdır. (Ölçüm yapılan metodun kendi sınıfı dahil değildir.) [5] [16]
Coupling Intensity (CINT)	Ölçüm yapılacak metodun diğer sınıflardan çağırdığı farklı metot sayıdır.[5]
Coupling Dispersion (CDISP)	Ölçüm yapılacak metodun diğer sınıflardan çağırdığı metotların ait oldukları sınıfların sayısının, CINT metrik değerine bölünmesiyle bulunur. [5]
Changing Methods (CM)	Ölçüm yapılacak metotu diğer sınıflardan çağıran farklı metot sayısıdır. [5] [16]
McCabe's Cyclomatic Number (CYCLO)	Ölçüm yapılacak metodun karmaşıklığıdır. [15] [5]
Foreign Data Providers (FDP)	Ölçüm yapılacak metodun eriştiği niteliklerin (direkt veya erişimci metotlar üzerinden) tanımlandığı farklı sınıf sayısıdır.[5]
Locality of Attribute Accesses (LAA)	Ölçüm yapılacak metodun diğer sınıflardan eriştiği niteliklerin sayısının, eriştiği tüm niteliklerin sayısına oranıdır. (Kendi nitelikleri + erişilen diğer sınıf nitelikleri)[5]
Lines of Code (LOC)	Ölçüm yapılacak metodun satır sayısıdır.[5] [12] (Satır sayısına, yorum satırları ve boşluklar dahildir.)
Maximum Nesting Level (MAXNESTING)	Ölçüm yapılacak metodun içe doğru dallanma sayısıdır.[5]
Number of Added Services (NAS)	Ölçüm yapılan sınıfta, ata sınıftan kalıtımla alınmamış ve geçersiz (override) kılınmamış public metot sayısıdır.[5]
Number of Accessor Methods (NOAM)	Ölçüm yapılacak sınıftaki erişimci metotların (getter / setter) sayısıdır.[5]
Number of Accessed Variables (NOAV)	Ölçüm yapılacak metodun eriştiği değişken / nitelik sayısıdır. (parametreler, yerel değişkenler, nesne nitelikleri, diğer sınıf nitelikleri)[5]
Number of Methods (NOM)	Ölçüm yapılacak sınıftaki toplam metot sayısıdır.[5]

Number of Public Attributes (NOPA)	Ölçüm yapılacak sınıftaki public nitelik sayısıdır.[5]
Number of Protected Members (NProtM)	Ölçüm yapılacak sınıftaki protected metot ve nitelik sayısıdır.[5]
Percentage of Newly Added Services (PNAS)	Ölçüm yapılan sınıfta, ata sınıftan kalıtımla alınmamış ve geçersiz (override) kılınmamış public metot sayısının, toplam public metot sayısına oranıdır.
Tight Class Cohesion (TCC)	Ölçüm yapılacak sınıftaki, en az bir niteliğe ortak olarak erişen metot çiftlerinin sayısının, sınıfta yer alan olası tüm metot çiftlerinin sayısına oranıdır. [5] [14] (Olası tüm metot çiftleri : (metot sayısı * (metot sayısı -1)) / 2)
Weighted Method Count (WMC)	Ölçüm yapılacak sınıftaki tüm metotların karmaşıklıkları toplamıdır. Bir metodun karmaşıklık hesabı CYCLO metriği ile bulunur. [5] [13] [15]
Weight Of a Class (WOC)	Ölçüm yapılacak sınıfta tanımlanmış fonksiyonel (yapıcı ve erişimci metotlar hariç) public metot sayısının, sınıftaki tüm public metot sayısına oranıdır. [5]

3.2 Tespit Edilen Kötü Kokular

God Class Sistemdeki yapılacak işlerin bir sınıf üzerinde merkezîleştirilmesi problemidir. Sistemdeki birçok sorumluluk bu sınıfa yüklenmiştir.

$$\text{God Class}\begin{cases}\text{Sınıf diğer sınıfların niteliklerine erişmeli,} & ATFD > 3\\ \text{Sınıfın fonksiyonel karmaşıklığı fazla olmalı,} & WMC \geq 47\\ \text{Sınıf düşük uyumluluğa sahip olmalı} & TCC < \frac{1}{3}\end{cases}$$

Feature Envy Bir metodun, diğer bir sınıfın verilerine kendi sınıfındaki verilerden daha çok ihtiyaç duyması problemidir.

$$\text{Feature Envy}\begin{cases}\text{Metot diğer sınıfların niteliklerine erişmeli,} & ATFD > 3\\ \text{Metot diğer sınıfların niteliklerine, kendi sınıfındaki niteliklerden daha fazla erişmeli,} & LAA < \frac{1}{3}\\ \text{Metodun diğer sınıflardan eriştiği nitelikler az sayıda sınıfa dağılmış olmalı} & FDP \leq 3\end{cases}$$

Data Class Bu sınıflar kendi verileri üzerinde işlem yapmayan, sadece veri tutan ve diğer sınıfların bu verilere eriştiği sınıflardır.

$$\text{Data Class}\begin{cases}\text{Sınıfın arayüzü fonksiyonellikten ziyade, daha çok veri sunmalı,} & WOC < \frac{1}{3}\\ \text{Sınıfın, dışarıya açık çok sayıda niteliği var ve karmaşıklığı yüksek olmamalı} & (1)\end{cases}$$

(1) =>

$$[(NOPA + NOAM > 3) \wedge (WMC < 31)]$$
$$\vee$$
$$[(NOPA + NOAM > 7) \wedge (WMC < 47)]$$

Brain Method Bir metodun bulunduğu sınıfa ait işlerin çoğunu üstlenmesinden kaynaklanır. Metodu yönetmek ve anlamak zorlaşır.

Brain Method:

Metot oldukça büyük olmalı,	$LOC > 65$
Metot birçok koşullu dallanmaya sahip olmalı,	$CYCLO \geq 4$
Metot birçok iç içe seviye içermeli,	$MAXNESTING \geq 3$
Metot birçok değişken kullanmalı	$NOAV > 7$

Brain Class *God class* kötü kokusuna çok benzer. Çünkü sistemde yapılacak işler bu sınıflar üzerinde merkezîleştirilmiştir. Ancak aralarında temel farklar vardır. *God class* sadece büyük karmaşık sınıflar değildir. Diğer sınıfların verilerine direkt olarak ulaşırken, kapsülleme kurallarını bozan sınıflardır. *Brain class* kötü kokusundan etkilenmiş sınıflar da büyük sınıflardır ama daha uyumlu (cohesive) sınıflardır ve en az 1 *Brain Method* kötü kokusuna sahip metot içermek zorundadırlar.

Brain Class:

Sınıf birden fazla *Brain Method* kötü kokusuna sahip metot içermeli ve büyük olmalı ya da sınıf 1 adet *Brain Method* içermeli ancak çok büyük olmalı,	(1)
Sınıfın karmaşıklığı yüksek, uyumluluğu ise düşük olmalı	(2)

(1) =>

$$[(\text{Birden fazla } \textit{Brain Method} \text{ var}) \wedge (LOC \geq 195)]$$
$$\vee$$
$$[(\text{Sadece 1 } \textit{Brain Method}\text{'a sahip}) \wedge (LOC \geq 390) \wedge (WMC \geq 94)]$$

$$(2) => (WMC \geq 47) \wedge (TCC < \frac{1}{2})$$

Intensive Coupling Bir metodun, birkaç sınıfta toplanmış birçok metodu çağırması problemidir.

Intensive Coupling:

Metot birden fazla iç içe koşul içermeli,	$MAXNESTING > 1$
Metot birkaç sınıfta toplanmış birçok metodu çağırmalı	(1)

$$(1) => (CINT > 7 \wedge CDISP < \frac{1}{2}) \vee (CINT > 3 \wedge CDISP < \frac{1}{4})$$

Dispersed Coupling Bir metodun, birçok farklı sınıfta yer alan birçok metodu çağırması problemidir.

$$\text{Dispersed Coupling}\begin{cases}\text{Metot birden fazla iç içe koşul içermeli,} & MAXNESTING > 1\\ \text{Metot birçok farklı sınıfta yer alan birçok metodu çağırmalı} & (1)\end{cases}$$

$$(1) => (CINT > 7) \wedge (CDISP \geq \frac{1}{2})$$

Shotgun Surgery Sistemdeki bir metodun, farklı sınıflarda yer alan birçok metot tarafından çağırılması problemidir.

$$\text{Shotgun Surgery}\begin{cases}\text{Metot çok sayıda başka metot tarafından çağırılmalı,} & CM > 7\\ \text{Metodu çağıran metotlar birçok sınıfa dağılmalı} & CC > 10\end{cases}$$

Refused Parent Bequest Çocuk sınıfların, ata sınıftan kalıtımla gelen üyeleri (metotları ve nitelikleri) kullanmaması veya az kullanması problemidir.

$$\text{Refused Parent Bequest}\begin{cases}\text{Çocuk sınıf mirası reddetmeli,} & (1)\\ \text{Çocuk sınıf çok küçük ve basit olmamalı} & (2)\end{cases}$$

$$(1) => [(NProtM > 3) \wedge (BUR < \frac{1}{3})] \vee (BOvR < \frac{1}{3})$$

$$(2) => [(AMW > 2) \vee (WMC > 14)] \wedge (NOM > 7)$$

Tradition Breaker Çocuk sınıfların, ata sınıflarından daha çok hizmet sunması mantıklı bir durumdur. Ancak çocuk sınıflar, ata sınıfın geleneğini sürdürmelidir. Bu kötü koku çocuk sınıfların ata sınıflarıyla ilişkisiz birçok yeni hizmet sunması problemidir.

$$\text{Tradition Breaker}\begin{cases}\text{Çocuk sınıfın arayüzündeki artış çok fazla olmalı,} & (1)\\ \text{Çocuk sınıfın karmaşıklığı ve boyutu çok büyük olmalı,} & (2)\\ \text{Ata sınıf çok küçük ve işlevsiz olmamalı} & (3)\end{cases}$$

$$(1) => (NAS \geq 7) \wedge (PNAS \geq \frac{2}{3})$$

$$(2) => [(AMW > 2) \vee (WMC \geq 47)] \wedge (NOM \geq 10)$$

$$(3) => (AMW > 2) \wedge (NOM > 5) \wedge (WMC \geq 24)$$

4 Kötü Koku Analizleri

Geliştirdiğimiz eklenti *Apache Log4j (1.2)* [17], *Apache Ivy (2.0)* [18], *PBeans (1.0)* [19], *Apache Velocity (1.4)* [20], *Apache Tomcat (6.0.38)* [21], *Apache POI (2.0rc1)* [22],*Apache Synapse (1.0)* [23] olmak üzere 7 adet açık kaynak proje üzerinde denenmiştir. Bu projelerin üzerinde çalışılmasının temel nedeni, bu projelerin hata veri kümelerinin (fault-dataset) herkese açık olarak sunulmasıdır. Normalde bu veri kümeleri hata tahmini (fault-estimation) için kullanılmaktadır. Ancak kötü kokularla hata oluşumunun ilişkisinin gösterilmesi açısından bizim içinde güzel bir girdi olmuşlardır. İlişki gösterilirken veri kümelerindeki hata bilgisi aynen korunmuştur. Kullandığımız veri kümeleri Marian Jureckzo tarafından tera-promise üzerinden sunulmuştur.[24] [25] [26] [27] [28] [29] [30]. Hata veri kümeleri bulunan bu projelerde eklentimiz denenerek, kötü kokuların hata oluşumundaki etkisi gözlemlenmiştir.

Tablo 2: Yazılımda yer alan kötü kokuların ve hatalarının istatistiksel dağılımı

	log4j		pbeans		ivy		synapse		velocity		poi		tomcat	
Toplam Sınıf Sayısı	223		30		422		161		221		370		1086	
Hatalı Sınıf Sayısı / Hata İçermeyen Sınıf Sayısı	189	34	20	10	40	382	16	145	147	74	37	333	77	1009
Kötü Koku İçeren Sınıf Sayısı / Kötü Koku İçermeyen Sınıf Sayısı	43	180	6	24	106	316	63	98	36	185	36	334	228	858
Toplam Hata Sayısı	498		36		56		21		210		39		114	
Toplam Kötü Koku Sayısı	62		8		239		131		72		68		543	
Hata İçeren Sınıflardaki Maksimum Hata Sayısı	10		4		3		4		7		2		6	
Kötü Koku İçeren Sınıflardaki Maksimum Kötü Koku Sayısı	6		2		23		8		12		9		14	
Hata İçeren Sınıflardaki Ortalama Hata Sayısı	2.63		1.80		1.40		1.31		1.43		1.05		1.48	
Kötü Koku İçeren Sınıflardaki Ortalama Kötü Koku Sayısı	1.44		1.33		2.25		2.07		2.00		1.89		2.38	
	!H & K	H & K	!H & K	H & K	!H & K	H & K	!H & K	H & K	!H & K	H & K	!H & K	H & K	!H & K	H & K
Data Class	0	8	1	2	39	1	15	2	3	5	5	4	43	0
God Class	0	3	0	0	6	5	2	1	0	0	0	1	7	10
Brain Class	0	1	0	0	1	3	1	0	1	0	2	1	6	6
Refused Parent Bequest	0	16	0	0	9	1	6	5	0	2	6	6	31	4
Tradition Breaker	0	0	0	0	0	0	0	0	0	0	0	0	4	0
Feature Envy	3	5	0	1	32	15	24	3	0	2	8	1	37	7
Brain Method	0	3	0	2	16	13	14	6	20	15	7	1	59	38
Shotgun Surgery	0	17	0	0	23	8	17	9	1	13	11	3	177	14
Intensive Coupling	0	4	0	2	20	13	14	9	3	3	9	2	37	24
Dispersed Coupling	0	2	0	0	21	13	3	0	1	3	1	0	20	19

!H & K : Hata raporlarında hata içermeyen sınıflarda bulunan kötü kokular.

H & K : Hata raporlarında hata içeren sınıflarda bulunan kötü kokular.

Projelerdeki ortak bulgu; hata sayısı fazla olan projelerin kötü koku sayısının da fazla olmasına rağmen, hata ile kötü koku arasında varoluşsal bir ilişkinin olmadığıdır. Örneğin log4j için açık biçimde hata ve kötü koku arasında varoluşsal bir ilişki mevcut iken, synapse için tam tersi bir durum görülmektedir. Kötü kokuların, projenin tüm sınıflarına yayılmak yerine, belli sınıflarında yoğunlaştığı da Tablo 2 içinde yer alan kötü koku içeren/kötü koku içermeyen sınıflar ve ortalama kötü koku sayısı değerlerinden anlaşılmaktadır. Genel olarak hatalı sınıfların hata sayısı, kötü koku içeren sınıfların kötü koku sayısından düşüktür. Bu durum hayatın içinde de olağandır çünkü hatanın fark edilmesi için testler ve yazılımın kullanılması gibi doğal süreçler çalışır ve hataların azaltılması çabalanır. Ancak kötü kokular için böylesi bir düzenleyici ve önleyici faaliyetin olmaması nedeniyle kaynak kodun doğal gelişimi içinde mevcudiyetlerini koru-

maya devam ederler. İstatistiksel dağılımda Tomcat projesinde diğer projelere göre oldukça yoğun bir kötü koku mevcudiyeti görülmektedir. Bu projeyi genel değerlendirmenin dışında tutarsak, Tradition Breaker kötü kokusunun en az rastlanan kötü koku olduğunu ve Brain Class, God Class kötü kokularının diğer en az karşılaşılan kötü koku türü olduğu söylenebilir.

5 Sonuçlar

Yapılan çalışma sonucunda, Java projelerindeki kaynak kodlar içerisinde yer alan kötü kokuları otomatik olarak bulan bir *Eclipse* eklentisi geliştirdik. Bu eklentiyle birlikte yazılımın kalitesi gerçekçi değerlerle ölçülebilecek, üretkenlik iyileştirilebilecek, hatalı modüller erkenden tespit edilip düzeltilebilecek, maliyet azaltılıp, bakım ve test edilebilirlik arttırılabilecektir.
Kötü koku tespit aracımızın kullanımını örneklemek ve bunlar ile ilgili durumu yansıtabilmek için; Tera Promise kapsamında Svn ve Sourceforge üzerinden kaynak kodlarına erişilebilen projeler kullanılmış ve projelerin mevcut hata bilgileri ile bu çalışmada tespit edilen kötü kokular bir arada verilerek, yazılım geliştiricilerin sıklıkla hangi kötü koku türlerini yapmaya açık olduğu tespit edilmeye çalışılmıştır. Kötü kokuların yazılım üzerindeki etkilerini ölçmek için, kötü koku içeren ve bilinen kötü kokuları içermeyen kaynak kodların bakım sürelerinin ve bakım sonrası hata istatistiklerinin karşılaştırılması gerekmektedir. Özellikle birden çok sınıfın etkilendiği bakım faaliyetleri önemli olacağı düşünülmektedir. Bu nedenle, şu an kendi içinde metrik toplama ve kötü koku tespit etme eklentisi için, gelecek çalışmalarda üç yönelim olacaktır: (I) Tespit edilen kötü kokulara ait literatürde sunulan düzeltme önerilerinin otomatik olarak kod içine yansıtılması (II) Kötü kokuların kodun bakım pratiklerine süre ve bakım sonrası hata oluşumu üzerine etkisinin deneylerle irdelenmesi.(III) Gerçeklenen eklentinin çıktılarının literatürde kabul görmüş *IPLASMA* programı ile karşılaştırılması.

Kaynaklar

1. Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, (1999).
2. M. Mäntylä, Bad smells in software - a taxonomy and an empirical study. Ph.D. dissertation, Helsinki University of Technology, 2003.
3. W. C. Wake, Refactoring Workbook. Boston, MA, USA: Addison Wesley Longman Publishing Co., Inc., 2003.
4. R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in Proceedings of the 20th International Conference on Software Maintenance. IEEE Computer Society Press, 2004, pp. 350– 359.
5. M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Springer-Verlag, 2006
6. C. Marinescu, R. Marinescu, P. Mihancea, D. Ratiu, and R. Wettel. iplasma: An integrated platform for quality assessment of objectoriented design. In Proceedings of 21st International Conference on Software Maintenance (ICSM 2005), Tools Section, 2005
7. M. Fokaefs, N. Tsantalis and A. Chatzigeorgiou, JDeodorant: Identification and Removal of Feature Envy Bad Smells, IEEE International Conference on Software Maintenance, 2007 October, pp. 519-520.

8. T. Chaikalis, N. Tsantalis and A. Chatzigeorgiou, JDeodorant: Identification and Removal of TypeChecking Bad Smells, 12th European Conference on Software Maintenance and Reengineering, 2008, pp. 329-331.
9. JDeodorant http://marketplace.eclipse.org/content/jdeodorant [Accessed May 2018].
10. Emerson Murphy-Hill and Andrew P. Black, An interactive ambient visualization for code smells, Proceedings of SOFTVIS '10, USA, October 2010
11. https://pmd.github.io/ [Accessed May 2018]
12. Mark Lorenz and Jeff Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994.
13. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 20(6):476–493, June 1994
14. J.M. Bieman and B.K. Kang. Cohesion and reuse in an objectoriented system. In Proceedings ACM Symposium on Software Reusability, April 1995.
15. T.J. McCabe. A measure of complexity. IEEE Transactions on Software Engineering, 2(4):308–320, December 1976.
16. Radu Marinescu. Measurement and Quality in Object-Oriented Design. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002.
17. Apache Log4j 1.2 Source Code http://svn.apache.org/repos/asf/logging/log4j/tags/v1_2_1/ [Accessed May 2018]
18. Apache Ivy 2.0 Source Code http://svn.apache.org/repos/asf/ant/ivy/core/tags/2.0.0/ [Accessed May 2018]
19. PBeans 1.0 Source Code https://sourceforge.net/projects/pbeans/files/pbeans/1.0/ [Accessed May 2018]
20. Apache Velocity 1.4 Source Code https://svn.apache.org/repos/asf/velocity/tools/tags/1.4/ [Accessed May 2018]
21. Apache Tomcat 6.0.38 Source Code http://svn.apache.org/repos/asf/tomcat/archive/tc6.0.x/tags/TOMCAT_6_0_38/ [Accessed May 2018]
22. Apache Poi 2.0.rc1 Source Code http://svn.apache.org/repos/asf/poi/tags/REL_2_0_RC1/ [Accessed May 2018]
23. Apache Synapse 1.0 Source Code http://svn.apache.org/repos/asf/synapse/tags/1.0/ [Accessed May 2018]
24. Marian Jureckzo. (2010). tomcat [Data set]. Zenodo. https://doi.org/10.5281/zenodo.322454 [Accessed May 2018]
25. Marian Jureckzo. (2010). velocity [Data set]. Zenodo. https://doi.org/10.5281/zenodo.322455 [Accessed May 2018]
26. Marian Jureckzo. (2010). synapse [Data set]. Zenodo.https://doi.org/10.5281/zenodo.322449 [Accessed May 2018]
27. Marian Jureckzo. (2010). poi [Data set]. Zenodo. https://doi.org/10.5281/zenodo.322443 [Accessed May 2018]
28. Marian Jureckzo. (2010). pbeans [Data set]. Zenodo. https://doi.org/10.5281/zenodo.322440 [Accessed May 2018]
29. Marian Jureckzo. (2010). log4j [Data set]. Zenodo. https://doi.org/10.5281/zenodo.268451 [Accessed May 2018]
30. Marian Jureckzo. (2010). ivy [Data set]. Zenodo. https://doi.org/10.5281/zenodo.322436 [Accessed May 2018]
31. https://www.tiobe.com/tiobe-index/ [Accessed May 2018]
32. https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/ [Accessed May 2018]
33. https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector