KAYNAK KODLARDAKİ KÖTÜ KOKULARIN OTOMATİK TESPİTİ İÇİN ECLIPSE EKLENTİSİ

ECLIPSE PLUGIN FOR AUTOMATIC DETECTION OF CODE SMELLS IN SOURCE CODES

MELİH ALTINTAŞ

PROF. DR. EBRU AKÇAPINAR SEZER
Tez Danışmanı

Hacettepe Üniversitesi
Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin
Bilgisayar Mühendisliği Anabilim Dalı için Öngördüğü
YÜKSEK LİSANS TEZİ olarak hazırlanmıştır.

2018

MELİH ALTINTAŞ' ın hazırladığı "**Kaynak Kodlardaki Kötü Kokuların Otomatik Tespiti İçin Eclipse Eklentisi**" adlı bu çalışma aşağıdaki jüri tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Dr. Öğr. Üyesi Ayça TARHAN
Başkan .....

Prof. Dr. Ebru AKÇAPINAR SEZER
Danışman .....

Doç. Dr. Murat HACIÖMEROĞLU
Üye .....

Doç. Dr. Hacer KARACAN
Üye .....

Dr. Öğr. Üyesi Engin DEMİR
Üye .....

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarafından **YÜKSEK LİSANS TEZİ** olarak onaylanmıştır.

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU
Fen Bilimleri Enstitüsü Müdürü

YAYINLAMA VE FİKRİ MÜLKİYET HAKLARI BEYANI

Enstitü tarafından onaylanan lisansüstü tezimin / raporumun tamamını veya herhangi bir kısmını, basılı (kağıt) ve elektronik formatta arşivleme ve aşağıda verilen koşullarla kullanıma açma iznini Hacettepe Üniversitesine verdiğimi bildiririm. Bu izinle Üniversiteye verilen kullanım hakları dışındaki tüm fikri mülkiyet haklarım bende kalacak, tezimin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım hakları bana ait olacaktır.

Tezin kendi orijinal çalışmam olduğunu, başkalarının haklarını ihlal etmediğimi ve tezimin tek yetkili sahibi olduğumu beyan ve taahhüt ederim. Tezimde yer alan telif hakkı bulunan ve sahiplerinden yazılı izin alınarak kullanılması zorunlu metinlerin yazılı izin alınarak kullandığımı ve istenildiğinde suretlerini Üniversiteye teslim etmeyi taahhüt ederim.

Yükseköğretim Kurulu tarafından yayınlanan **" Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"** kapsamında tezim aşağıda belirtilen koşullar haricinde YÖK Ulusal Tez Merkezi / H. Ü. Kütüphaneleri Açık Erişim Sisteminde erişime açılır.

- o Enstitü / Fakülte yönetim kurulu kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren 2 yıl ertelenmiştir. [1]
- o Enstitü / Fakülte yönetim kurulunun gerekçeli kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren .... Ay ertelenmiştir. [2]
- o Tezimle ilgili gizlilik kararı verilmiştir. [3]

15 /10 /2018

Melih ALTINTAŞ

"Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"

(1) Madde 6. 1. Lisansüstü tezle ilgili patent başvurusu yapılması veya patent alma sürecinin devam etmesi durumunda, tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü veya fakülte yönetim kurulu iki yıl süre ile tezin erişime açılmasının ertelenmesine karar verebilir

(2) Madde 6. 2. Yeni teknik, materyal ve metotların kullanıldığı, henüz makaleye dönüşmemiş veya patent gibi yöntemlerle korunmamış ve internetten paylaşılması durumunda 3. Şahıslara veya kurumlara haksız kazanç imkanı oluşturabilecek bilgi ve bulguları içeren tezler hakkında tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü ve fakülte yönetim kurulunun gerekçeli kararı ile altı ayı aşmamak üzere tezin erişime açılması engellenebilir.

(3) Madde 7. 1. Ulusal çıkarları veya güvenliği ilgilendiren, emniyet, istihbarat, savunma ve güvenlik, sağlık vb. konulara ilişkin lisansüstü tezlerle ilgili gizlilik kararı, tezin yapıldığı kurum tarafından verilir*. Kurum ve kuruluşlarla yapılan işbirliği protokolü çerçevesinde hazırlanan lisansüstü tezlere ilişkin gizlilik kararı ise, ilgili kurum ve kuruluşun önerisi ile enstitü veya fakültenin uygun görüşü üzerine üniversite yönetim kurulu tarafından verilir. Gizlilik kararı verilen tezler Yükseköğretim Kuruluna bildirilir.
Madde 7. 2. Gizlilik kararı verilen tezler gizlilik süresince enstitü veya fakülte tarafından gizlilik kuralları çerçevesinde muhafaza edilir, gizlilik kararının kaldırılması halinde Tez Otomasyon Sistemine yüklenir.

* Tez danışmanının önerisi ve enstitü anabilim dalının uygun görüşü üzerine enstitü veya fakülte yönetim kurulu tarafından karar verilir.

ETİK

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, tez yazım kurallarına uygun olarak hazırladığım bu tez çalışmasında,

- tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlak kurallarına uygun olarak sunduğumu,
- başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- atıfta bulunduğum eserlerin tümünü kaynak olarak gösterdiğimi,
- kullanılan verilerde herhangi bir tahrifat yapmadığımı,
- ve bu tezin herhangi bir bölümünü bu üniversitede veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı

beyan ederim.

11.10.2018

MELİH ALTINTAŞ

ÖZET

KAYNAK KODLARDAKİ KÖTÜ KOKULARIN OTOMATİK TESPİTİ İÇİN ECLIPSE EKLENTİSİ

Melih ALTINTAŞ

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Danışmanı: Prof. Dr. Ebru AKÇAPINAR SEZER

Ekim 2018, 92 Sayfa

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 tezde 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

ABSTRACT

ECLIPSE PLUGIN FOR AUTOMATIC DETECTION OF CODE SMELLS IN SOURCE CODES

Melih ALTINTAŞ

Master of Science, Department of Computer Engineering

Supervisor: Prof. Dr. Ebru AKÇAPINAR SEZER

October 2018, 92 Pages

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 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

TEŞEKKÜR

Yüksek lisans çalışmamda beni cesaretlendiren; gerek derslerde gerekse de tez aşamasında büyük bir sabır ve özveriyle her türlü desteği ve yapıcı yönlendirmeyi sağlayan tez danışmanım ve saygıdeğer hocam Prof. Dr. Ebru AKÇAPINAR SEZER'e sonsuz saygı ve teşekkürlerimi sunuyorum.

Çalışmalarım boyunca sabırla beni destekleyen sevgili eşim Pınar ALTINTAŞ'a hoşgörü ve anlayışları için teşekkür ediyorum.

İÇİNDEKİLER

Sayfa

ÇİZELGELER

Sayfa

ŞEKİLLER

Sayfa

SİMGELER VE KISALTMALAR

AMW	Sınıfın Ortalama Metot Ağırlığı (Average Method Weight)
ATFD	Dış Verilere Erişim Sayısı (Access to Foreign Data)
BOvR	Ata Sınıfın Geçersizlik Oranı (Base Class Overriding Ratio)
BUR	Ata Sınıfın Kullanım Oranı (Base Class Usage Ratio)
CC	Değişen Sınıf Sayısı (Changing Classes)
CDISP	Bağımlılık Dağılımı (Coupling Dispersion)
CINT	Bağımlılık Yoğunluğu (Coupling Intensity)
CM	Değişen Metot Sayısı (Changing Methods)
CYCLO	Çevrimsel Karmaşık (McCabe's Cyclomatic Number)
FDP	Dış Veri Sağlayan Sınıf Sayısı (Foreign Data Providers)
LAA	Erişilen Niteliklerin Yerelliği (Locality of Attribute Accesses)
LOC	Kodun Satır Sayısı (Lines of Code)
MAXNESTING	Maksimum İç İçe Seviye Sayısı (Maximum Nesting Level)
NAS	Eklenen Servis Sayısı (Number of Added Services)
NOAM	Erişim Metotları Sayısı (Number of Accessor Methods)
NOAV	Erişilen Değişken Sayısı (Number of Accessed Variables)
NOM	Metot Sayısı (Number of Methods)
NOPA	Sınıfın Genel Nitelik Sayısı (Number of Public Attributes)
NProtM	Sınıfın Korunumlu Nitelik ve Metot Sayısı (Number of Protected Members)
PNAS	Yeni Eklenen Servislerin Oranı (Percentage of Newly Added Services)
TCC	Sınıf Uyumunun Sıkılığı (Tight Class Cohesion)
WMC	Sınıfın Ağırlıklı Metot Sayısı (Weighted Method Count)
WOC	Sınıfın Ağırlığı (Weight of a Class)

1. GİRİŞ

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 ve siyah kutu 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].

1.1. Amaç ve Hedefler

- Kaynak koddan yazılım metriklerinin toplanması ve kötü koku içeren kod parçacıklarının bulunması,
- Büyük projelerdeki bakım ve test gibi maliyetli işlemlerin süresini kısaltarak iş gücü, zaman ve maliyet kazancı yaratılması,
- Yazılımın kaynak kodunun kalitesinin gerçekçi değerlerle ölçülmesi,

- Yazılım kalitesinin ve üretkenliğin iyileştirilmesi,
- Büyük sistemlerdeki hataya eğilimli modüllerin ortadan kaldırılması,

tezin başlıca amaçlarıdır.

1.2. Tezin Kapsamı ve Yapısı

Tez kapsamında; hedeflerimi gerçekleştirmek için Eclipse platformu üzerinde çalışan bir eklenti geliştirilmiştir. Bu eklenti Eclipse çalışma ortamındaki tüm projeleri, projelerin içindeki tüm paketleri ve paket içerisinde tanımlanan tüm Java dosyalarını gezerek sınıf ve metotları ayrıştırmaktadır. Daha sonra bu sınıf ve metotlar üzerinden metrikler toplayarak Lanza ve Marinescu'nun *"Object-Oriented Metrics in Practice"* adlı kitabında [5] tanımlanmış 10 adet kötü kokuyu otomatik olarak bularak Eclipse platformu üzerinden kullanıcılara sunmaktadır. Böylece yazılımcı ileride hatalara neden olabilecek sınıf ve metotları, hataya sebep olmadan önce tahmin ederek düzeltme şansı yakalayabilecektir.

Tezin yapısı aşağıda maddeler halinde verilmiştir:

- Giriş bölümünde; çalışmanın amacı, hedefleri, kapsamı ve tezin yapısı hakkında bilgiler verilmiştir.
- İkinci bölümde; literatür özetlenmiştir.
- Üçüncü bölümde; kötü kokuların tespiti hakkında bilgi verilmiştir.
- Dördüncü bölümde; 7 proje üzerinde test ettiğimiz eklentimizin istatistiksel sonuçları anlatılmıştır.
- Beşinci bölümde; eklentimizin sonuçları IPLASMA aracı ile karşılaştırılmıştır.
- Altıncı bölümde; çalışmanın sonuçları ve gelecekte yapılması planlanan çalışmalar sunulmuştur.

2. LİTERATÜR ÖZETİ

Kötü koku 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. Bu çalışmalardan sonra 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.

2.1. Mäntylä'nın Kötü Koku Sınıflandırması

Mäntylä [2] Fowler'ın [1] kitabında yapılan kötü koku tanımlarının ortak özellikler içerdiğini ve bunların 5 temel grupta sınıflandırılabileceğini söylemiştir. Sınıflandırma Çizelge 2.1'de verilmiştir.

Çizelge 2.1. Mäntylä Kötü Koku Sınıflandırması

Grup Adı	Grup İçerisindeki Kötü Kokular
Çok Büyük Kod Parçaları (Bloaters)	Uzun Metot (Long Method) Büyük Sınıf (Large Class) Temel Veri Tiplerine Eğilim (Primitive Obsession) Uzun Parametre Listesi (Long Parameter List) Veri Yığını (Data Clumps)

Nesne Yönelimli Programlama Prensiplerini Bozan Kod Bölümleri (Object-Orientation Abusers)	Koşul İfadeleri (Switch Statements) Geçici Alan (Temporary Field) Miras Reddi (Refused Bequest) Değişik Arayüzlere Sahip Alternatif Sınıflar (Alternative Classes with Different Interfaces)
Değişikliği Birçok Yerin Değişikliğini Gerektiren Kod Parçaları (Change Preventers)	Iraksayan Değişiklik (Divergent Change): Parçacık Tesiri (Shotgun Surgery) Paralel Kalıtım Hiyerarşilerinin Oluşturulması (Parallel Inheritance Hierarchies)
Vazgeçilebilir Kod Parçaları (Dispensables)	Yorum Satırları (Comments) Kod Tekrarı (Duplicate Code) Tembel Sınıf (Lazy Class) Veri Sınıfı (Data Class) Ölü Kod (Dead Code) Spekülatif Genelleme (Speculative Generality)
Sınıflar Arası Aşırı Bağ Kuran Kod Parçaları (Couplers)	Özellik Kıskançlığı (Feature Envy) Uygunsuz İlişki (Inappropriate Intimacy) Mesaj Zincirleri (Message Chains) Aracı Sınıflar (Middle Man)

2.1.1. Çok Büyük Kod Parçaları

Kontrolsüz büyüyen sınıf ve metotların yönetiminin zorlaşarak düzenleme yapmanın çok zor hâle geldiği kötü koku türleridir [2].

Uzun Metot: Metot çok fazla kod satırı içerir. Genel olarak, on satırdan uzun olan herhangi bir metot içerisinde bu kötü koku belirmeye başlar. Bu metot yönetilmez bir boyuta ulaşana kadar farkedilmeden kalır. İnsan beyninin çalışma şekli yeni bir şey yaratmak yerine, varolan bir şeye ekleme yapmaya daha müsaittir. Bu nedenle yeni bir istek geldiğinde yeni bir metot yazmak yerine varolan güçlendirilir ancak bu durum, zamanla metodun çok büyümesine ve yönetilmez duruma gelmesine neden olur. Çözüm olarak bu metotlar birden fazla metoda bölünmeli, her bir işlem başka metotlara dağıtılmalıdır.

Büyük Sınıf: Sınıf çok fazla nitelik, metot ve kod satırından oluşur. Sınıflar genellikle küçük başlamalarına rağmen yeni istekler arttıkça büyümeye başlarlar. Nesne yönelimli programlamada temel olan, bir sınıfın bir sorumluluğunun olmasıdır. Bu kötü kokudan etkilen sınıflar birçok sorumluluk üstlenmiş ve çok fazla sayıda farklı işi yapmaktadırlar. Çözüm olarak her bir sınıfa bir sorumluluk yüklenmeli, sorumluluğu çok olan sınıflar parçalanarak sorumluluklar yeni sınıflara dağıtılmalıdır.

Temel Veri Tiplerine Eğilim: Basit işler için küçük objeler yerine temel veri tiplerin kullanılması, bunun sürekli olarak eklenmesi ile de karmaşık bir yapının oluşturulmasıdır. Genelde yazılımcılar yeni bir istek karşısında yeni bir sınıf yaratmak yerine, sınıfa bir nitelik veya metoda bir parametre ekleyerek durumu çözmeye çalışırlar. Ancak eklenen bu niteliklerin zamanla artmasıyla sınıf gittikçe büyür ve karmaşıklaşır. Çözüm olarak, eklenen primitive nitelikler mantıksal olarak gruplanıp başka bir sınıfta toplanmalı ve o sınıf üzerinden davranışları modellenip öyle kullanılmalıdır.

Uzun Parametre Listesi: Bir metot üç ya da dört parametreden fazlasına sahipse bu kötü kokudan etkilenmiştir. Uzun parametre listesinin genel sebebi bir metodun içinde birden çok algoritmanın yer almasıdır. Çözüm olarak; ilgili metot, bulunduğu sınıfın niteliklerini kullanacak hâle getirilmeli, sadece dışarıdan verilmesi zorunlu parametreleri parametre olarak kullanmalı veya metot parametreleri mantıksal olarak gruplanarak sınıf hâline getirilmeli ve metot bu sınıfı parametre olarak almalıdır.

Veri Yığını: Belli başlı kod parçalarının, bir metodun veya sınıfın içerisinde tekrar tekrar kullanılması problemidir. Genellikle kopyala yapıştır kodlaması sonucu sıklıkla rastlanır. Bu gibi kod parçaları kodu uzatır ve okunabilirliği düşürür. Çözüm olarak, tekrar tekrar kullanılan kod blokları ortaklanarak bir metot hâline getirilip ilgili her yerde bu metot kullanılabilir.

2.1.2. Nesne Yönelimli Programlama Prensiplerini Bozan Kod Parçaları

Bu grupta yer alan kötü kokuların ortak özellikleri nesne yönelimli programlama prensiplerini bozmalarıdır [2].

Koşul İfadeleri: Kod içerisinde bulunan karmaşık *switch* ve *if* koşullarından kaynaklanır. İyi bir nesneye yönelik programlamada *switch* ve *if* yapıları birçok dallanma içeriyorsa nadir kullanılmalıdır. Bunun sebebi yeni eklenen her yeni durum için yeni bir koşul gerekliliğidir. Yeni eklenen bir durum birçok yerdeki *switch* ve *if* koşullarını etkileyebilir. Çözüm olarak tasarım örüntüleri ve polimorfizm kullanılabilir.

Geçici Alan: Tanımlanan bu değişkenlere sadece belli bir koşul sağlandığında değer atanır, bunun dışında boş kalırlar. Kodun anlaşılabilirliğini ve okunabilirliğini çok azaltırlar.

Miras Reddi: Kalıtım kavramının yanlış kullanımından kaynaklanan bir kötü kokudur. Çocuk sınıflar, ata sınıf içerisindeki metotların ve niteliklerin çok azını kullanıyorsa bu kötü koku belirmeye başlar. Kalıtım gereği miras alan çocuk sınıflar, ata sınıflarının miraslarını kullanmalıdır, aksi takdirde yapılan bu kalıtım gereksizdir.

Değişik Arayüzlere Sahip Alternatif Sınıflar: Sistemdeki 2 sınıfın aynı işlevi gerçekleştirmesi için aynı işi yapan, fakat farklı isimde iki metoda sahip olması problemidir. Genellikle yazılımcıların o işi yapmak için önceden yazılmış metodun varlığından haberdar olmayıp, aynı işlevi gerçekleştiren başka bir metot yazmasından kaynaklanır. Çözüm olarak kopya metotlardan biri silinmelidir.

2.1.3. Değişikliği Birçok Yerin Değişikliğini Gerektiren Kod Parçaları

Bu grupta yer alan kötü kokuların ortak özelliği, kodun herhangi bir yerinde bir değişiklik yapıldığında kodun diğer bölümlerinin de çok sayıda değişiklik gerektirmesidir [2].

Parçacık Tesiri: Bir metot içerisinde yapılan bir değişikliğin sistemde yer alan birçok farklı metot içerisinde de değişikliğe neden olmasıdır. Bunun temel nedeni, birçok metodun değişiklik yapılacak metodu çağırarak işlemler yapmasıdır.

Iraksayan Değişiklik: Bu kötü koku genelde Parçacık Tesiri kötü kokusuyla karıştırılmaktadır. Aralarında temel bir fark vardır: Iraksayan Değişiklik kötü kokusunda yapılacak bir değişiklik aynı sınıf içerisinde birçok değişikliğe neden olur. Parçacık Tesiri kötü kokusunda ise yapılacak değişiklik birçok başka sınıfı etkiler. Genellikle zayıf tasarımlardan ve kopyala yapıştır programlamadan kaynaklanmaktadır.

Paralel Kalıtım Hiyerarşilerinin Oluşturulması: Bir kalıtım ağacının bir başka kalıtım ağacına bağlı olması problemidir. Bir sınıftan yeni bir sınıf türettiğimizde, bağlı olunan kalıtım hiyerarşisindeki başka bir sınıftan da aynı sınıfı türetme zorunluluğu oluşuyor ise bu kötü koku belirmeye başlar.

2.1.4. Vazgeçilebilir Kod Parçaları

Bu grupta yer alan kötü kokuların ortak özelliği kodun gereksiz parçaları olmalarıdır. Bunların kod içerisinden temizlenmesi ile kod daha temiz, daha verimli ve anlaşılması kolay hale gelir [2].

Yorum Satırları: Yorum satırları kodun okunabilirliği ve bakımı için çok önemlidir. Ancak bir metot veya sınıfın üzerinde bulunan yorum satırlarının çok fazla olması, o sınıf veya metodun çok karmaşık olduğunu gösterir. Yazılımcı kodu tekrar anlayabilmek için bu yorum satırlarına ihtiyaç duymuştur. Bir sınıf veya metot için yapılabilecek en iyi yorum, sınıf veya metodun adının anlamlı bir şekilde verilmesidir. Sınıflar ve metotlar yorum satırlarına en az gereksinim kalacak şekilde basit ve anlaşılabilir yazılmalıdır.

Kod Tekrarı: Kaynak kod içerisinde yer alan 2 kod parçasının neredeyse veya tamamen birbiriyle aynı olması problemidir. Bu kötü kokunun oluşmasının temel nedenleri; farklı yazılımcıların, yazılımın farklı bölümleri için geliştirme yapması nedeniyle birbirlerinden haberdar olmayarak, birbirleriyle aynı veya yakın kod parçalarını yazmış olmalarından veya projenin son teslim tarihine yetişmesi kaygısıyla kopyala yapıştır kodların sisteme eklenmesidir.

Tembel Sınıf: Her sınıf bir amaç doğrultusunda yazılır. Ancak birçok yeniden düzenleme aşamasından sonra gittikçe küçülerek işlevsiz hâle gelebilirler. Düzenlemelerden sonra gittikçe küçülüp işlevsiz hâle gelen bu sınıflar Tembel Sınıf kötü kokusundan etkilenmiştir. Çözüm olarak; bu sınıfların yaptığı küçük işler, ilgili diğer bir sınıfa taşınarak bu sınıflar tamamen silinmelidir.

Veri Sınıfı: Veri sınıfı içerisinde sadece nitelikler ve bu niteliklere ulaşmayı sağlayan erişimci (getter-setter) metotları barındıran sınıflardır. Genellikle diğer sınıflar için veri tutmak amacıyla kullanılırlar. Herhangi bir işlevsellik içermez, kendi verileriyle ilgili tek bir işlem bile yapmazlar. Buradaki temel problem, nesne yönelimli

programlamanın temel özellliği olan, nesnenin kendi verileri üzerinde işlemler gerçekleştirebilme özelliğini devre dışı bırakmasıdır.

Ölü Kod: Bir değişkenin, parametrenin, metodun veya sınıfın artık kullanımda olmamasıdır. Yazılım gereksinimleri değiştiğinde veya yeni bir düzenleme yapıldığında, eski kodu temizlemek gerekir. Bu yapılmadığında kullanılmayan eski isteklere ait kod parçaları hâlâ kaynak kod içerisinde yer alır. Bu durum, kodun okunabilirliğini ve bakımını zorlaştırır. Çözüm olarak, herhangi bir değişkenin, sınıfın veya metodun işlevi kalmadığında kod içerisinden temizlenmelidir.

Spekülatif Genelleme: Gelecekteki isteklerin hesaplanarak aslında asla hayata geçirilmeyecek sınıf, metot ve değişkenlerin tanımlanmasıdır. Sonuç olarak o projeye dönük okunan kodun anlaşılabilirliği ve okunabilirliği azalır.

2.1.5. Sınıflar Arası Aşırı Bağ Kuran Kod Parçaları

Bu gruptaki tüm kokular, sınıflar arasında aşırı bir bağ kurulmasına neden olurlar. Nesneye yönelik programlamada istenen sınıflar arası bağlılığın (coupling) düşük, sınıf içerisi uyumun (cohesion) yüksek olmasıdır. Bu kötü kokular bu prensibi bozarlar [2].

Özellik Kıskançlığı: Bu metotlar başka bir sınıfın nitelik ve metotlarına kendi sınıflarınınkinden daha çok ihtiyaç duymaktadır. Nesne yönelimli programlamada bir metot daha çok kullandığı nitelik ve metotlarla aynı yerde olmalıdır. Bunun temel nedeni, kodda bir değişiklik gerektiğinde değişiklik yapılacak her yerin aynı bileşende olması gerekliliğidir. Çözüm olarak; metot diğer bir sınıfın metot ve niteliklerine daha çok ihtiyaç duyuyorsa, metot ihtiyaç duyduğu nitelik ve metotları barındıran sınıfa taşınmalıdır.

Uygunsuz İlişki: Bu sınıflar başka bir sınıfın nitelik ve metotlarını çok fazla kullanırlar. Nesneye yönelik programlama prensiplerinden biri sınıflar arası bağlılığın düşük olmasıdır. Her sınıf diğer sınıflar hakkında çok az şey bilmelidir. Bu sınıflar bakım kolaylığını ve yeniden kullanılabilirliği azaltmaktadır.

Mesaj Zincirleri: Zincirleme metot çağrılarını ifade eder. İstemci bir metot çağrımı yaptığında, bu zincirleme çağrılara sebep olur. Bu zincirler, istemcinin isteğinin tüm

istek yapılan sınıflara bağlı olduğu anlamına gelir. Bu ilişkilerde meydana gelen değişiklikler istemciyi değiştirmeyi gerektirebilir.

Aracı Sınıflar: Kendisine gelen metot çağrılarını başka sınıf metotlarına delege eden aracı sınıflardır. Nesneye yönelik proglamlamada nesneden beklenen kendi nitelik ve metotlarıyla bir iş üretmesidir. Ancak bu sınıflar sadece delege işlemi yaptığından nesneye yönelik programlamanın prensiplerini bozmaktadır.

2.2. Lanza ve Marinescu'nun Kötü Koku Sınıflandırması

Lanza ve Marinescu "Object-Oriented Metrics in Practice" adlı kitaplarında [5] 3 ayrı sınıflama yaparak toplam **11** adet kötü koku tanımlamışlardır. Bu çalışmayla birlikte kaynak kod içerisinde yer alan kötü kokuları, otomatik olarak tespit edebilecek metrik tabanlı kurallar da açık biçimde ortaya çıkmıştır. Sınıflandırma Çizelge 2.2'de verilmiştir. Bu bölümde yer alan kötü kokular üzerinde Türkçe bir çalışma olmadığından, isimlerin kafa karışıklığı yaratmaması adına kötü koku isimleri Türkçeleştilrilmemiş aynen kullanılmıştır.

Çizelge 2.2. Lanza ve Marinescu'nun Kötü Koku Sınıflandırması

Grup Adı	Grup İçerindeki Kötü Kokular
Kimlik Uyumsuzlukları (Identity Disharmonies)	God Class Feature Envy Data Class Brain Method Brain Class Significant Duplication
İşbirliği Uyumsuzlukları (Collaboration Disharmonies)	Intensive Coupling Dispersed Coupling Shotgun Surgery
Sınıflama Uyumsuzlukları (Classification Disharmonies)	Refused Parent Bequest Tradition Breaker

2.2.1. Kimlik Uyumsuzlukları

Bu grupta yer alan kötü kokular aynı anda sadece bir metot veya bir sınıfı etkiler. Nesne yönelimli programlamada bir sınıfın veya metodun bir sorumluluğunun olması tavsiye edilmektedir. Sınıf ve metotlara birden fazla sorumluluk yüklenmemelidir. Aynı zamanda sınıf içi uyum yüksek olmalı, metotlar sınıfı temsil etmelidir. Bu grupta yer alan kötü kokulardan etkilenen varlıklar, genellikle birden fazla sorumluluk yüklenmiş, uyumlulukları düşük sınıf ve metotlardır.

God Class: Nesneye yönelik iyi tasarlanmış bir kod tasarımında, yapılacak işler sınıflar arasında eşit olarak dağıtılmalıdır. *God Class* kötü kokusu sistemdeki yapılacak işlerin bir sınıf üzerinde merkezileştirilmesi problemidir. Sistemdeki birçok sorumluluk bu sınıfa yüklenmiştir. Çok büyük, karmaşık, uyumluluğu (cohesion) düşük sınıflardır ve diğer sınıfların verilerine yoğun olarak erişirler. Tekrar kullanılabilirliği, kod okunabilirliğini ve anlaşılabilirliğini azaltırlar. God Class tespit yöntemi Şekil 2.1' de verilmiştir.

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

Şekil 2.1. God Class Tespit Yöntemi

Feature Envy: Nesneler, veri ve bu verileri işleyen metotları bir arada tutan yapılardır. Feature Envy kötü kokusu metotlarda görülür. Bir metodun, diğer bir sınıfın verilerine kendi sınıfındaki verilerden daha çok ihtiyaç duyması problemidir. Bu metotlar, diğer sınıfların verilerine direkt veya erişimci metotlar üzerinden çok fazla erişirler. Bu durum, metodun yanlış bir sınıfa koyulduğunun göstergesi olup, verilerini daha çok kullandığı sınıfa taşınması gerektiğinin bir göstergesidir. Feature Envy tespit yöntemi Şekil 2.2'de verilmiştir.

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

Şekil 2.2. Feature Envy Tespit Yöntemi

Data Class: Bu sınıflar kendi verileri üzerinde işlem yapmayan, sadece veri tutan ve diğer sınıfların bu verilere eriştiği sınıflardır. Sınıfta işlevselliği sağlayan metotların olmaması, veri ve metotların bir arada tutulmadığını gösterir. Bu durum da, nesne yönelimli programlamanın prensiplerine aykırıdır. Data Class tespit yöntemi Şekil 2.3'te verilmiştir.

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

$$(1) => [(NOPA + NOAM > 3) \wedge (WMC < 31)] \vee [(NOPA + NOAM > 7) \wedge (WMC < 47)]$$

Şekil 2.3. Data Class Tespit Yöntemi

Brain Method: Bir metodun sınıfa ait işlerin çoğunu yüklenmesiyle oluşur. Metot kontrolden çıkana kadar birçok işlevsellik üzerine eklenir. Bunun sonunda metodu yönetmek ve anlamak zorlaşır. God Class kötü kokusuyla benzerdir. God Class bir sistemin yapması gereken işlerin tek bir sınıfta yapılması iken, Brain Method bir sınıfın yapması gereken işi metotlara ayırmayıp tek bir metot üzerinde yoğunlaştırmasıdır. Brain Method tespit yöntemi Şekil 2.4'te verilmiştir.

$$\text{Brain Method} \begin{cases} \text{Metot oldukça büyük olmalı,} & LOC > 65 \\ \text{Metot birçok koşullu dallanmaya sahip olmalı,} & CYCLO \geq 4 \\ \text{Metot birçok iç içe seviye içermeli,} & MAXNESTING \geq 3 \\ \text{Metot birçok değişken kullanmalı} & NOAV > 7 \end{cases}$$

Şekil 2.4. Brain Method Tespit Yöntemi

Brain Class: Bu kötü koku karmaşık sınıfların en az bir metodunun Brain Method kötü kokusundan etkilenmiş olduğu sınıflardır. God Class kötü kokusuna çok benzer. Çünkü sistemde yapılacak işler bu sınıflar üzerinde merkezileştirilmiştir. Ancak aralarında temel farklar vardır. God Class sadece büyük karmaşık sınıflar değildir. Diğer sınıfların verilerine direkt olarak ulaşırken kapsülleme kurallarını bozan sınıflardır. Kısaca Brain Class kötü kokuları da büyük sınıflardır ama içlerinde daha uyumlu (cohesive) sınıflardır. Brain Class tespit yöntemi Şekil 2.5'te verilmiştir.

$$\text{Brain Class} \begin{cases} \text{Sınıf birden fazla } \textit{Brain Method} \text{ kötü kokusuna sahip metot içermeli ve büyük olmalı ya da sınıf 1 adet } \textit{Brain Method} \text{ içermeli ancak çok büyük olmalı,} & (1) \\ \text{Sınıfın karmaşıklığı yüksek, uyumluluğu ise düşük olmalı} & (2) \end{cases}$$

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

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

Şekil 2.5. Brain Class Tespit Yöntemi

2.2.2. İşbirliği Uyumsuzlukları

Bu grupta yer alan kötü kokular birbirlerine bağımlılığı olan birden çok metodu etkiler. Nesneye yönelik programlamada tavsiye edilen düşük bağımlılık ve yüksek uyumdur. Bu grupta yer alan kötü kokular, metotlar arası yüksek bağımlılık sonucu ortaya çıkar.

Intensive Coupling: Nesne yönelimli bir sistemde bir metodun, sistemdeki diğer birçok metodu çağırması sonucu bu metotlara bağımlı olması durumudur. Bu bağımlılığın yoğun olarak adlandırılmasının temel sebebi, bağlı olunan metotların az sayıda sınıfta toplanmış olmasıdır. Intensive Coupling tespit yöntemi Şekil 2.6'da verilmiştir.

$$\text{Intensive Coupling} \begin{cases} \text{Metot birden fazla iç içe koşul içermeli,} & MAXNESTING > 1 \\ \text{Metot birkaç sınıfta toplanmış birçok metodu çağırmalı} & (1) \end{cases}$$

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

Şekil 2.6. Intensive Coupling Tespit Yöntemi

Dispersed Coupling: Nesne yönelimli bir sistemde bir metodun, sistemdeki diğer birçok metodu çağırması sonucu bu metotlara bağımlı olması durumudur. Bu bağımlılığın dağılmış olarak adlandırılmasının temel sebebi, bağlı olunan metotların birçok sınıfa dağılmış olmasıdır. Dispersed Coupling tespit yöntemi Şekil 2.7'de verilmiştir.

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

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

Şekil 2.7. Dispersed Coupling Tespit Yöntemi

Shotgun Surgery: Nesne yönelimli tasarımlarda sadece diğer sınıflara bağımlılık sorun yaratmaz. Diğer sınıfların metotlarının, bir sınıfın bir metoduna bağımlı olması da sorun yaratır. Çünkü bağımlı olunan metot üzerinde yapılacak küçük bir değişiklik birçok yeri etkilemeye başlar. Shotgun Surgery kötü kokusunun tespitinde amaç, değişiklik yapıldığında sistemdeki birçok yeri etkileyen metotları bulmaktır. Shotgun Surgery tespit yöntemi Şekil 2.8'de verilmiştir.

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

Şekil 2.8. Shotgun Surgery Tespit Yöntemi

2.2.3. Sınıflama Uyumsuzlukları

Nesne yönelimli programlamanın en güçlü yanlarından biri kalıtımdır. Kalıtım sayesinde ata sınıfın özellikleri, çocuk sınıflar tarafından miras alınarak yeniden kullanılabilir ve bu sayede ata sınıf ile çocuk sınıf arasında ilişki kurulmuş olur. Bu grupta yer alan kötü kokuların ortak özelliği, kalıtımın yanlış veya eksik kullanılmasıdır.

Refused Parent Bequest: Nesne yönelimli programlamada en önemli bileşenlerden biri kalıtımdır. Kalıtımın temel amacı kodun yeniden kullanılabilirliğidir. Çocuk sınıflar, ata sınıftan gelen üyeleri (metot ve nitelikler) kullanmalıdır. Eğer çocuk sınıf, ata sınıftan gelen özellikleri kullanmıyorsa; bu durum, kalıtımda bir problem olduğuna işaret eder. Kalıtımın reddi; sistemdeki kod tekrar kullanılabilirliğini, metotlar arası uyumluluğu azaltır. Refused Parent Bequest tespit yöntemi Şekil 2.9'da verilmiştir.

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

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

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

Şekil 2.9. Refused Parent Bequest Tespit Yöntemi

Tradition Breaker: Çocuk sınıflar, ata sınıfların dışarıya sunduğu hizmetleri (arayüzü) otomatik olarak alırlar. Çocuk sınıfların daha çok hizmet sunması, diğer bir deyişle, bu arayüze yeni metotlar eklemesi beklenen bir durumdur. Ancak çocuk

sınıflar, ata sınıfın geleneğini sürdürmelidir. Yani yeni sunduğu hizmetler eski hizmetleri kullanmalı ve bunlarla ilişkili olmalıdır. Eğer çocuk sınıfın sunduğu yeni metotlar, ata sınıftan gelen metotlarla ilişkili değilse (anlamsal olarak çok farklıysa) bu durum, yine kalıtımda bir problemin işaretidir. Tradition Breaker tespit yöntemi Şekil 2.10'da verilmiştir.

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

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

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

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

Şekil 2.10. Tradition Breaker Tespit Yöntemi

2.3. Lanza ve Marinescu'nun Kötü Koku Tespiti İçin Kullandığı Metrikler

Kötü kokuların tespitinin gerçekleştirimi için hesaplanması gereken tüm metrikler Lanza ve Marinescu'nun kitabında [5] tanımlanmış, alt bölümlerde kısa tanımları verilmiştir. Tanımlarla birlikte değerleri hesaplanacak metriklerin hangi varlıklar (sınıf veya metot) için hesaplanacağı, hangi kötü koku türleri için kullanıldıkları, sahip olması gereken erişim belirteçleri (public, private, protected), statik veya soyut olup olamayacakları detaylı bir şekilde belirtilmiştir. Tanımı verilen tüm metrikler bizim kendi yazdığımız Java sınıfları üzerinden hesaplanmış, herhangi bir üçüncü parti kütüphane kullanılmamıştır.

AMW: Ölçüm yapılacak sınıfın içerdiği metotların karmaşıklıkları toplamının, sınıfın içerdiği toplam metot sayısına oranıdır. [5] Bir metodun karmaşıklık değeri CYCLO metrik değeriyle bulunur. AMW metrik detayları Çizelge 2.3'te verilmiştir.

Çizelge 2.3. AMW Metrik Detayları

<table>
<tr><td colspan="2">Kullanım Amacı</td><td colspan="6">Refused Parent Bequest ve Tradition Breaker kötü kokuları için kullanılır.</td></tr>
<tr><td colspan="2" rowspan="2">Varlık</td><td rowspan="2">Sınıf</td><td colspan="3">Tanım</td><td colspan="2">Soyut</td></tr>
<tr><td colspan="3">Kullanıcı Tanımlı</td><td colspan="2">-</td></tr>
<tr><td colspan="8">İlişkiler</td></tr>
<tr><td rowspan="2">Sahip Olunan</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Kullanıcı Tanımlı</td><td>Hepsi</td><td>+</td><td>+</td><td>+</td><td>+</td></tr>
</table>

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]. ATFD metrik detayları Çizelge 2.4'te verilmiştir.

Çizelge 2.4. ATFD Metrik Detayları

<table>
<tr><td>Kullanım Amacı</td><td colspan="8">God Class ve Feature Envy kötü kokuları için kullanılır.</td></tr>
<tr><td rowspan="2">Varlık</td><td rowspan="2">Sınıf</td><td colspan="6">Tanım</td><td>Soyut</td></tr>
<tr><td colspan="6">Kullanıcı Tanımlı</td><td>+</td></tr>
<tr><td rowspan="2">Varlık</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Kullanıcı Tanımlı</td><td>Hepsi</td><td>+</td><td>+</td><td>+</td><td>-</td></tr>
<tr><td colspan="9">İlişkiler</td></tr>
<tr><td rowspan="4">Erişim Yapılan</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Aynı Hiyerarşide Olmayan Sınıflar</td><td>Public</td><td>Sadece</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td rowspan="2">Nitelik</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Statik</td><td>Sabit</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Aynı Hiyerarşide Olmayan Sınıflar</td><td>Public</td><td>+</td><td>-</td></tr>
</table>

BOvR: Ölçüm yapılacak sınıfta geçersiz kılınan (overridden) metotların sayısının, sınıf içerisindeki tüm metot sayısına oranıdır [5]. **BOvR** metrik detayları Çizelge 2.5'te verilmiştir.

Çizelge 2.5. BOvR Metrik Detayları

<table>
<tr><td>Kullanım Amacı</td><td colspan="6">Refused Parent Bequest kötü kokusu için kullanılır.</td></tr>
<tr><td rowspan="2">Varlık</td><td colspan="2" rowspan="2">Sınıf</td><td colspan="2">Tanım</td><td colspan="2">Soyut</td></tr>
<tr><td colspan="2">Kullanıcı Tanımlı</td><td colspan="2">-</td></tr>
<tr><td colspan="7">İlişkiler</td></tr>
<tr><td rowspan="4">Geçersiz Kılınan Metotlar</td><td>Yer</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Ölçülen Sınıf</td><td>Hepsi</td><td>+</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Yer</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Ölçülen Sınıfın Ata Sınıfı</td><td>Hepsi</td><td>+</td><td>-</td><td>-</td><td>+</td></tr>
<tr><td rowspan="2">Sahip Olunan Metotlar</td><td>Yer</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Ölçülen Sınıftaki Metotlar</td><td>Hepsi</td><td>+</td><td>-</td><td>-</td><td>-</td></tr>
</table>

BUR: Ölçüm yapılacak sınıfta kalıtımdan gelen nitelik ve metotların kullanım oranıdır [5]. BUR metrik detayları Çizelge 2.6'da verilmiştir.

Çizelge 2.6. BUR Metrik Detayları

<table>
<tr><td colspan="2">Kullanım Amacı</td><td colspan="6">Refused Parent Bequest kötü kokusu için kullanılır.</td></tr>
<tr><td colspan="2" rowspan="2">Varlık</td><td colspan="2" rowspan="2">Sınıf</td><td colspan="3">Tanım</td><td>Soyut</td></tr>
<tr><td colspan="3">Kullanıcı Tanımlı</td><td>+</td></tr>
<tr><td colspan="8">İlişkiler</td></tr>
<tr><td rowspan="4">Erişim Yapılan</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Ölçülen Sınıf</td><td>Hepsi</td><td>+</td><td>+</td><td>+</td><td>-</td></tr>
<tr><td rowspan="2">Nitelik</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Statik</td><td>Sabit</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Ata Sınıfı</td><td>Protected</td><td>+</td><td>+</td></tr>
<tr><td rowspan="3">Çağrılan</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Ölçülen Sınıf veya Metot</td><td>Hepsi</td><td>+</td><td>+</td><td>+</td><td>-</td></tr>
<tr><td>Metot</td><td>Aynı Hiyerarşide Olmayan Sınıflar</td><td>Protected</td><td>Sadece</td><td>+</td><td>+</td><td>-</td></tr>
</table>

CC: Ölçüm yapılacak metodu çağıran metotların ait oldukları farklı sınıfların sayısıdır. (Ölçüm yapılan metodun kendi sınıfı dahil değildir.) [5,16] CC metrik detayları Çizelge 2.7'de verilmiştir.

Çizelge 2.7. CC Metrik Detayları

Kullanım Amacı		**Shotgun Surgery** kötü kokusu için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	+
İlişkiler							
Çağıran	Sınıf	Tanım					Soyut
		Ölçülen Metotu Çağıran Metotun Ait Olduğu Sınıf					+

CDISP: Ölçüm yapılacak metodun diğer sınıflardan çağırdığı metotların ait oldukları sınıfların sayısının, CINT metrik değerine bölünmesiyle bulunur [5]. CDISP metrik detayları Çizelge 2.8'de verilmiştir.

Çizelge 2.8. CDISP Metrik Detayları

Kullanım Amacı		**Intensive Coupling ve Dispersed Coupling** kötü kokuları için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-
İlişkiler							
Çağrılan	Sınıf	Tanım	Soyut				
		Kullanıcı Tanımlı	+				

LOC: Ölçüm yapılacak metodun satır sayısıdır [5,12]. (Satır sayısına, yorum satırları ve boşluklar dahildir.) LOC metrik detayları Çizelge 2.9'da verilmiştir.

Çizelge 2.9. LOC Metrik Detayları

Kullanım Amacı	**Brain Method ve Brain Class** kötü kokuları için kullanılır.						
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-

CINT: Ölçüm yapılacak metodun diğer sınıflardan çağırdığı farklı metot sayıdır [5]. CINT metrik detayları Çizelge 2.10'da verilmiştir.

Çizelge 2.10. CINT Metrik Detayları

Kullanım Amacı		**Intensive Coupling ve Dispersed Coupling** kötü kokuları için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-
İlişkiler							
Çağrılan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	+

MAXNESTING: Ölçüm yapılacak metodun içe doğru dallanma sayısıdır [5]. MAXNESTING metrik detayları Çizelge 2.11'de verilmiştir.

Çizelge 2.11. MAXNESTING Metrik Detayları

Kullanım Amacı		**Intensive Coupling, Dispersed Coupling** kötü kokuları için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-

CM: Ölçüm yapılacak metodu diğer sınıflardan çağıran farklı metot sayısıdır [5,16]. CM metrik detayları Çizelge 2.12'de verilmiştir.

Çizelge 2.12. CM Metrik Detayları

Kullanım Amacı		**Shotgun Surgery** kötü kokusı için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	+
İlişkiler							
Çağıran	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-

CYCLO: Ölçüm yapılacak metodun karmaşıklığıdır [5,15]. CYCLO metrik detayları Çizelge 2.13'te verilmiştir.

Çizelge 2.13. CYCLO Metrik Detayları

Kullanım Amacı		**Brain Method** kötü kokusu için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-

FDP: Ölçüm yapılacak metodun eriştiği niteliklerin (direkt veya erişimci metotlar üzerinden) tanımlandığı farklı sınıf sayısıdır [5]. FDP metrik detayları Çizelge 2.14'te verilmiştir.

Çizelge 2.14. FDP Metrik Detayları

Kullanım Amacı		**Feature Envy** kötü kokusu için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-
İlişkiler							
Erişilen	Sınıf	Tanım					Soyut
		Kullanıcı Tanımlı, Erişilen Nitelik ve Erişimci Metotların Bulunduğu Sınıf					+

LAA: Ölçüm yapılacak metodun kendi sınıfından eriştiği niteliklerin sayısının, eriştiği tüm niteliklerin sayısına oranıdır. (Kendi nitelikleri + erişilen diğer sınıf nitelikleri) [5] LAA metrik detayları Çizelge 2.15'te verilmiştir.

Çizelge 2.15. LAA Metrik Detayları

<table>
<tr><td>Kullanım Amacı</td><td colspan="7">Feature Envy kötü kokusu için kullanılır.</td></tr>
<tr><td rowspan="2">Varlık</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Kullanıcı Tanımlı</td><td>Hepsi</td><td>+</td><td>+</td><td>+</td><td>-</td></tr>
<tr><td colspan="8">İlişkiler</td></tr>
<tr><td rowspan="2">Erişim Yapılan</td><td rowspan="2">Nitelik</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Statik</td><td>Sabit</td><td colspan="2" rowspan="2"></td></tr>
<tr><td>Kullanıcı Tanımlı</td><td>Hepsi</td><td>+</td><td>+</td></tr>
</table>

NAS: Ölçüm yapılan sınıfta, ata sınıftan kalıtımla alınmamış ve geçersiz (override) kılınmamış public metot sayısıdır [5]. NAS metrik detayları Çizelge 2.16'da verilmiştir.

Çizelge 2.16. NAS Metrik Detayları

Kullanım Amacı	**Tradition Breaker** kötü kokusu için kullanılır.						
Varlık	Sınıf	Tanım				Soyut	
		Kullanıcı Tanımlı				+	
İlişkiler							
Sahip Olunan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Public	+	-	-	+

NOAM: Ölçüm yapılacak sınıftaki erişimci metotların (getter / setter) sayısıdır [5]. NOAM metrik detayları Çizelge 2.17'de verilmiştir.

Çizelge 2.17. NOAM Metrik Detayları

Kullanım Amacı	**Data Class** kötü kokusu için kullanılır.						
Varlık	Sınıf	Tanım				Soyut	
		Kullanıcı Tanımlı				-	
İlişkiler							
Sahip Olunan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Public	Sadece	-	-	-

NOAV: Ölçüm yapılacak metodun eriştiği değişken / nitelik sayısıdır. (Parametreler, yerel değişkenler, nesne nitelikleri, diğer sınıf nitelikleri) [5] NOAV metrik detayları Çizelge 2.18'de verilmiştir.

Çizelge 2.18. NOAV Metrik Detayları

Kullanım Amacı		**Brain Method** kötü kokusu için kullanılır.					
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-
İlişkiler							
Erişilen	Nitelik Değişken	Tanım	Erişim Belirteçleri		Statik		Sabit
		Kullanıcı Tanımlı	Hepsi		+		-

NOM: Ölçüm yapılacak sınıftaki toplam metot sayısıdır [5]. NOM metrik detayları Çizelge 2.19'da verilmiştir.

NOPA: Ölçüm yapılacak sınıftaki public nitelik sayısıdır [5]. NOPA metrik detayları Çizelge 2.20'de verilmiştir.

NProtM: Ölçüm yapılacak sınıftaki protected metot ve nitelik sayısıdır [5]. NProtM metrik detayları Çizelge 2.21'de verilmiştir.

Çizelge 2.19. NOM Metrik Detayları

Kullanım Amacı		**Refused Parent Bequest veTradition Breaker** kötü kokuları için kullanılır.					
Varlık		Sınıf		Tanım		Soyut	
				Kullanıcı Tanımlı		**+**	
İlişkiler							
Sahip Olunan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	+

Çizelge 2.20. NOPA Metrik Detayları

Kullanım Amacı		**Data Class** kötü kokusu için kullanılır.			
Varlık		Sınıf	Tanım		Soyut
			Kullanıcı Tanımlı		-
İlişkiler					
Sahip Olunan	Nitelik / Değişken	Tanım	Erişim Belirteçleri	Statik	Sabit
		Kullanıcı Tanımlı	Public	-	-

Çizelge 2.21. NProtM Metrik Detayları

<table>
<tr><td colspan="2">Kullanım Amacı</td><td colspan="6">Refused Parent Bequest kötü kokusu için kullanılır.</td></tr>
<tr><td colspan="2" rowspan="2">Varlık</td><td rowspan="2">Sınıf</td><td colspan="4">Tanım</td><td>Soyut</td></tr>
<tr><td colspan="4">Kullanıcı Tanımlı</td><td>+</td></tr>
<tr><td colspan="8">İlişkiler</td></tr>
<tr><td rowspan="4">Sahip Olunan</td><td rowspan="2">Nitelik Değişken</td><td colspan="3">Tanım</td><td colspan="2">Erişim Belirteçleri</td><td>Statik</td><td>Sabit</td></tr>
<tr><td colspan="3">Kullanıcı Tanımlı</td><td colspan="2">Protected</td><td>+</td><td>+</td></tr>
<tr><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Kullanıcı Tanımlı</td><td>Protected</td><td>+</td><td>-</td><td>+</td><td>+</td></tr>
</table>

PNAS: Ölçüm yapılan sınıfta, ata sınıftan kalıtımla alınmamış ve geçersiz (override) kılınmamış public metot sayısının, toplam public metot sayısına oranıdır [5]. PNAS metrik detayları Çizelge 2.22'de verilmiştir.

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) TCC metrik detayları Çizelge 2.23'te verilmiştir.

WMC: Ölçüm yapılacak sınıftaki tüm metotların karmaşıklıkları toplamıdır. Bir metodun karmaşıklık hesabı CYCLO metriği ile bulunur [5,13,15]. WMC metrik detayları Çizelge 2.24'te verilmiştir.

Çizelge 2.22. PNAS Metrik Detayları

Kullanım Amacı	**Tradition Breaker** kötü kokusu için kullanılır						
Varlık	Sınıf	Tanım					Soyut
		Kullanıcı Tanımlı					+
Varlık	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	-
İlişkiler							
Erişim Yapılan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Aynı Hiyerarşide Olmayan Sınıflar	Public	Sadece	-	-	-
	Nitelik	Tanım	Erişim Belirteçleri	Statik	Sabit		
		Aynı Hiyerarşide Olmayan Sınıflar	Public	+	-		

Çizelge 2.23. TCC Metrik Detayları

Kullanım Amacı	**God Class ve Brain Class** kötü kokuları için kullanılır.						
Varlık	Sınıf	Tanım				Soyut	
		Kullanıcı Tanımlı				-	
İlişkiler							
Erişim Yapılan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Ölçülen Sınıf	Hepsi	+	-	+	-
	Nitelik	Tanım	Erişim Belirteçleri	Statik	Sabit		
		Ölçülen Sınıf	Hepsi	+	-		

Çizelge 2.24. WMC Metrik Detayları

Kullanım Amacı	**Refused Parent Bequest, Tradition Breaker, God Class, Data Class, Brain Class** kötü kokuları için kullanılır.						
Varlık	Sınıf		Tanım			Soyut	
			Kullanıcı Tanımlı			-	
İlişkiler							
Sahip Olunan	Metot	Tanım	Erişim Belirteçleri	Get Set	Yapıcı	Statik	Soyut
		Kullanıcı Tanımlı	Hepsi	+	+	+	+

WOC: Ölçüm yapılacak sınıfta tanımlanmış fonksiyonel (yapıcı ve erişimci metotlar hariç) public metot sayısının, sınıftaki tüm public metot sayısına oranıdır [5]. WOC metrik detayları Çizelge 2.25'te verilmiştir.

Çizelge 2.25. WOC Metrik Detayları

<table>
<tr><td colspan="2">Kullanım Amacı</td><td colspan="6">Data Class kötü kokusu için kullanılır.</td></tr>
<tr><td colspan="2" rowspan="2">Varlık</td><td colspan="2" rowspan="2">Sınıf</td><td colspan="2">Tanım</td><td colspan="2">Soyut</td></tr>
<tr><td colspan="2">Kullanıcı Tanımlı</td><td colspan="2">-</td></tr>
<tr><td colspan="8">İlişkiler</td></tr>
<tr><td rowspan="6">Sahip Olunan</td><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Fonksiyonel Metot</td><td>Public</td><td>-</td><td>-</td><td>+</td><td>-</td></tr>
<tr><td rowspan="2">Metot</td><td>Tanım</td><td>Erişim Belirteçleri</td><td>Get Set</td><td>Yapıcı</td><td>Statik</td><td>Soyut</td></tr>
<tr><td>Getter Setter</td><td>Public</td><td>Sadece</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td rowspan="2">Nitelik</td><td>Tanım</td><td>Erişim Belirteçleri</td><td colspan="2">Statik</td><td colspan="2">Sabit</td></tr>
<tr><td>Sınıftaki Nitelikler</td><td>Public</td><td colspan="2">+</td><td colspan="2">-</td></tr>
</table>

2.4. Kötü Koku Tespit Araçları

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 ifadeyle, herhangi bir geliştirme platformu ile entegrasyonu yoktur ve bu nedenle kod geliştirilirken eş zamanlı olarak kötü kokuların tespiti mümkün değildir.

Avantajları [43]:

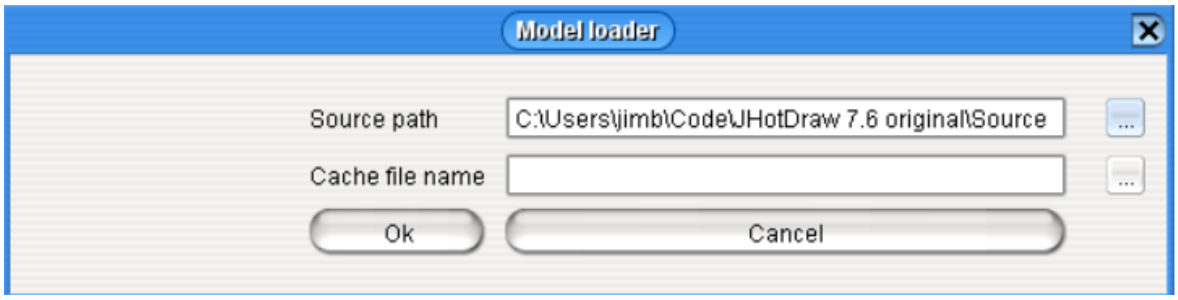
- Kaynak kodu ayrıştırma hızı çok fazladır.
- Açık kaynak kodlu bir yazılım olup herhangi bir lisans gerektirmemektedir.
- Kötü kokular dışında kaynak kodda yer alan birçok problemi bulup sunabilmektedir.

Dezavantajları [43]:

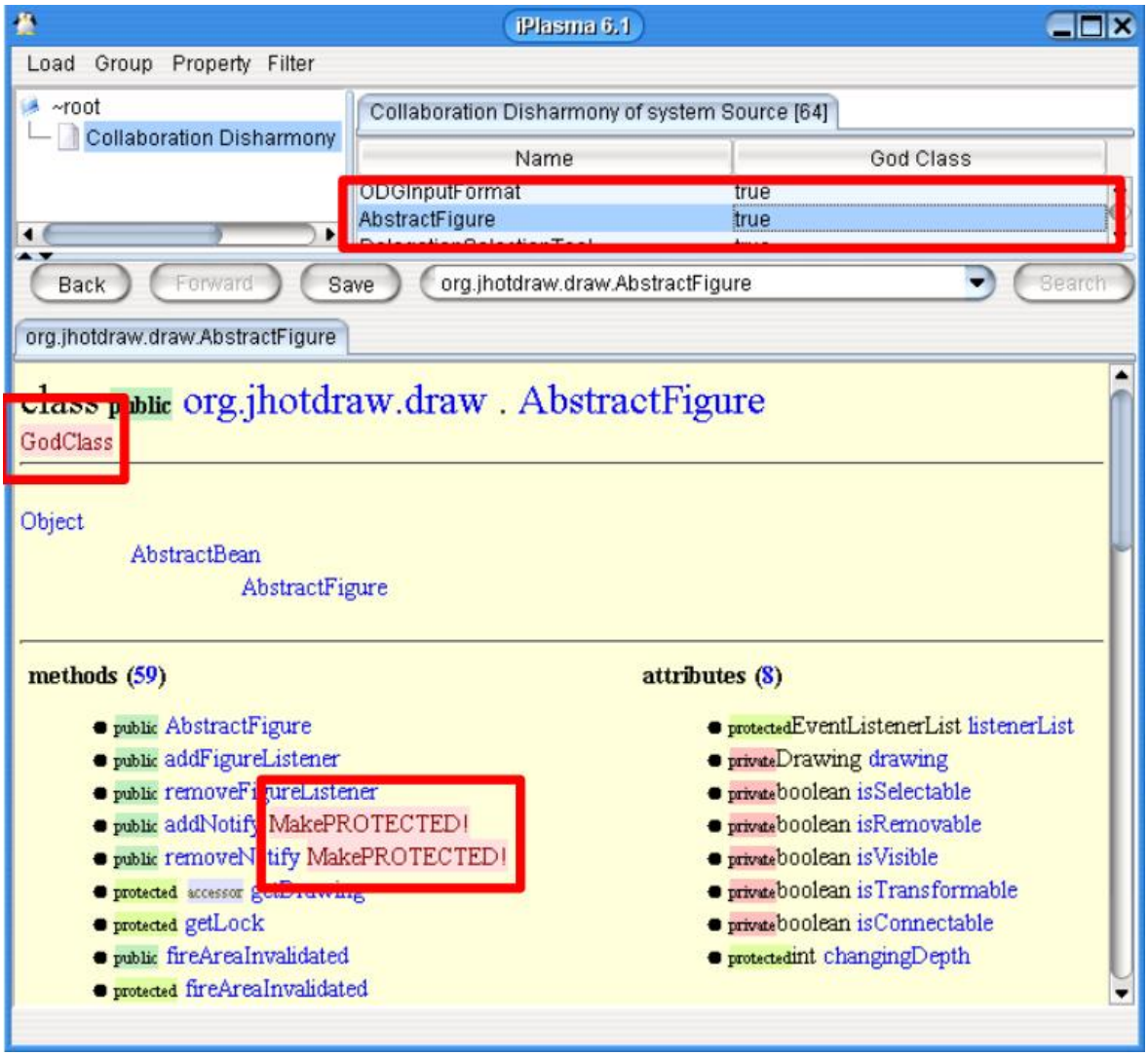
- Java 8 kodlarındaki yeni yapıları desteklememektedir.
- Kullanımı zor ve son kullanıcı dostu bir uygulama değildir.
- Programda yer alan tüm özellikler tamamen gerçeklenmemiştir. (Sadece menülerde yer alıyor ancak kullanılmıyor.)

Programın en büyük dezavantajlarından biri geliştirme ortamına entegre olmayıp, bağımsız bir uygulama olmasıdır. İndirilen program, tools dizini altındaki insider.bat dosyası çift tıklanarak çalışmaktadır. Bir bat dosyası üzerinden çalıştırılıyor olması, programın kullanıcı dostu olmadığının göstergesidir. İncelemek istediğiniz projenin bulunduğu dizini programa vererek üzerinde çalışmasını sağlayabilirsiniz. Şekil 2.11'de bir örneği gösterilmiştir. IPLASMA'nın bir diğer dezavantajı, projede bulunan kötü kokuların tamamını kullanıcıya bir ekran üzerinden sunmamasıdır. Her bir sınıf tek tek incelenmek zorundadır. Yani, tüm rapor bir anda verilememektedir. Çok hızlı sonuç vermesinin temel nedeni de budur. Program çıktısını inceleyen kişiye toplu bir rapor üretmediğinden diğer uygulamalara göre hızlıdır ancak bu durum, kötü koku tespiti yapılacak projenin incelenmesini çok zorlaştırmaktadır. Kötü kokuları gruplamak mümkündür ancak bunu yapmak bile bir projeyi incelemek açısından çok

zaman kaybettirmektedir. IPLASMA tarafından JHotDraw projesinde tespit edilen bir God Class örneği Şekil 2.12'de verilmiştir.



Şekil 2.11. IPLASMA Programına Proje Dizini Verilmesi



Şekil 2.12. IPLASMA Programı Tarafından Tespit Edilmiş Bir God Class Örneği

JDeodorant [7,8,9] Java dilinde yazılmış kaynak kod üzerinden toplam **5** adet (Feature Envy, Type Checking, Long Method, God Class, Duplicated Code) kötü koku tespiti yapabilen bir Eclipse eklentisidir. Geliştirme ortamına entegrasyonu olması açısından başarılı bir çalışmadır. Bizim eklentimize göre bulduğu kötü koku sayısı çok azdır.

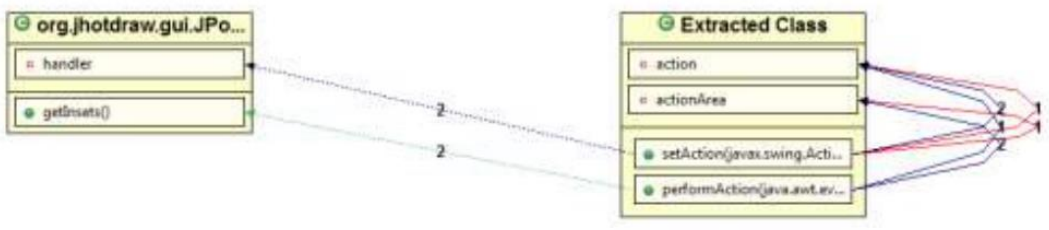
Avantajları [43]:

- Eclipse'in e4 versiyonları hariç en güncel versiyonlarını desteklemektedir.
- Açık kaynak kodlu bir yazılım olup herhangi bir lisans gerektirmemektedir.
- Eklenti üzerinden bir proje üstünde kötü koku tespiti yapıldığında, aynı proje üzerinde bir daha kötü koku tespiti yapılmak istenirse eklentinin önbellek yaklaşımı sayesinde ayrıştırma işlemi bir önceki işleme göre daha hızlı yapılmaktadır.
- Kötü kokuların tespitinin dışında, otomatik olarak düzeltme önerileri sunmaktadır.
- UML üzerinden görselleştirilmeye destek vermektedir. (Şekil 2.13)

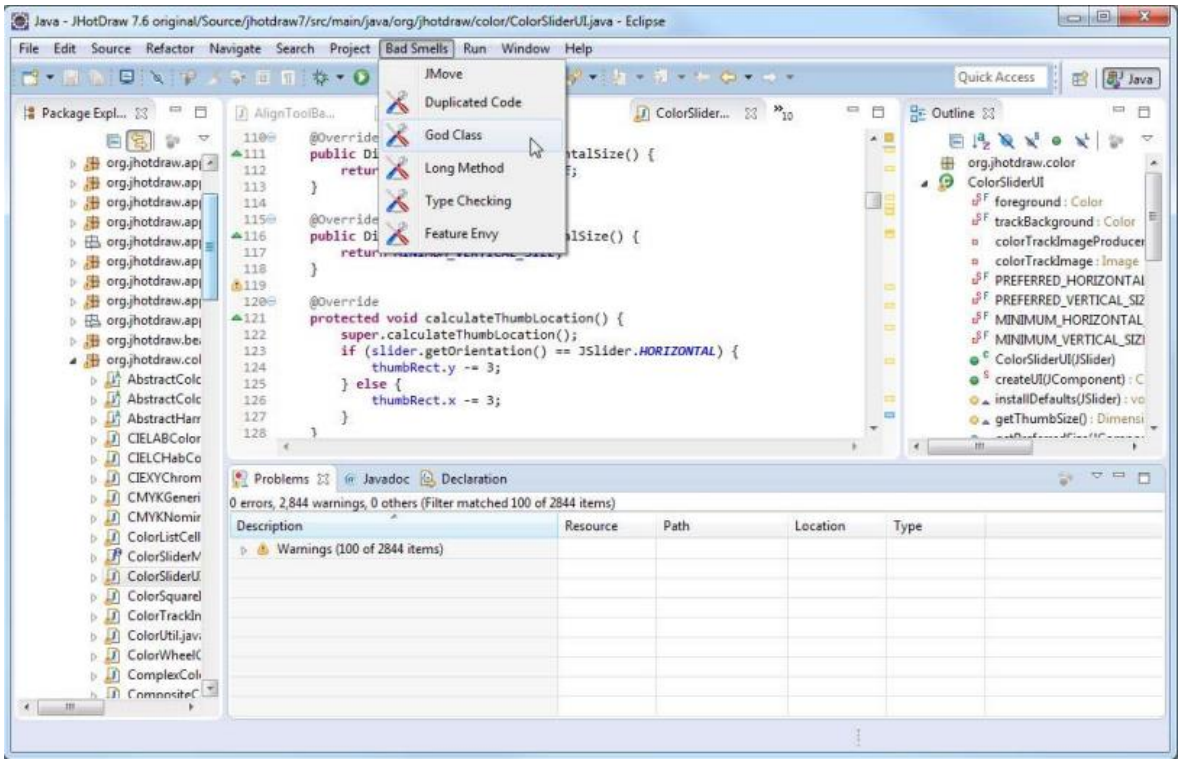
Dezavantajları [43]:

- Eklenti arka planda çalışamamaktadır. Bu nedenle aynı anda kod yazmaya devam edilememektedir.
- Çok büyük projelerde kötü koku tespiti çok uzun zamanlar alabilmektedir. (10-20 dakika)
- Java 8 ile gelen yeni özellikleri desteklememektedir.

Eklentinin kullanımına Eclipse'e eklenen Bad Smells bölümünden herhangi bir kötü koku seçilerek başlanabilir. Eklentinin kullanımı Şekil 2.14'te örneklenmiştir.



Şekil 2.13. JDeodorant UML Görselleştirme



Şekil 2.14. JDeodorant Eklentisinin Kullanılması

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 çalışmadır.

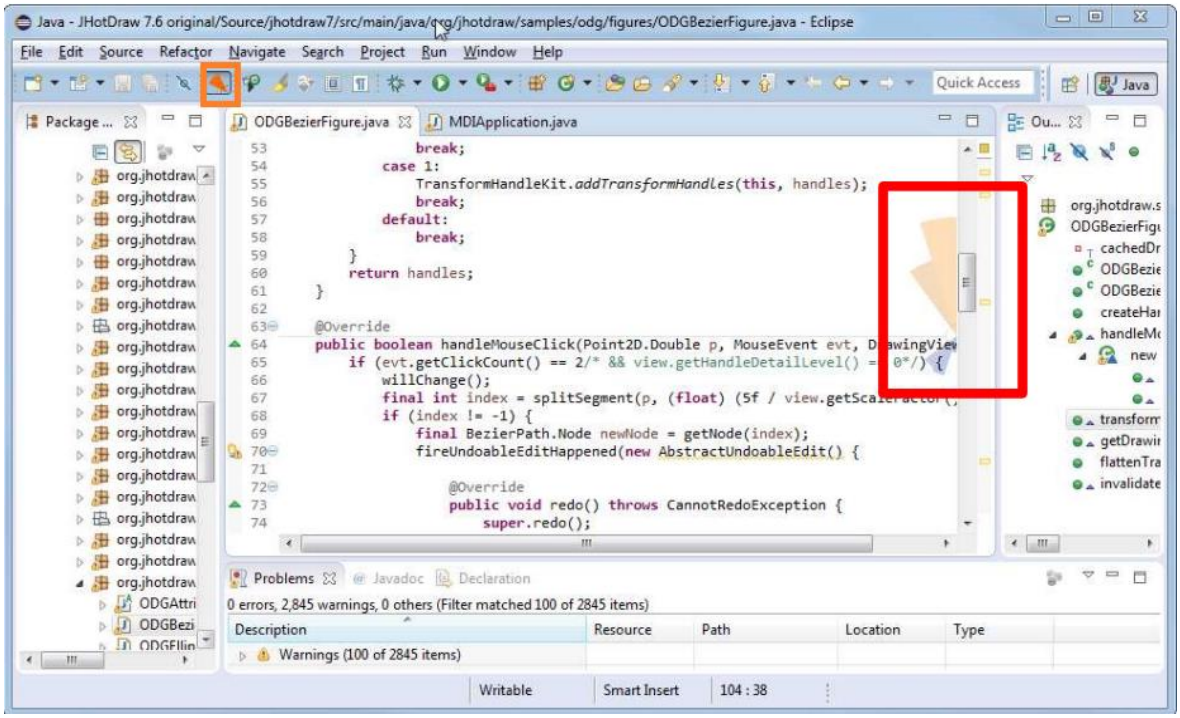
Avantajları [43]:

- Eclipse'in e4 versiyonları hariç en güncel versiyonlarını desteklemektedir.
- Açık kaynak kodlu bir yazılım olup herhangi bir lisans gerektirmemektedir.
- Eklenti arka planda çalışabilmektedir. Bu nedenle kötü koku tespiti sırasında aynı anda kod yazmaya devam edebilirsiniz.

Dezavantajları [43]:

- Java 8 ile gelen yeni özellikleri desteklememektedir.

Eklentinin kullanımına Eclipse'e eklenen araç çubuğu üzerindeki Stench Blossom Smell Indicator üzerinden başlanabilir. Eklentinin kullanımı Şekil 2.15'te örneklenmiştir.



Şekil 2.15. Stench Blossom Eklentisinin Kullanımı

Şekil 2.15'te sağ tarafta gördüğünüz kırmızı çerçevelenmiş ekran kötü kokuları göstermektedir. Daha az etkili olduğu düşünülen kötü kokular turuncu, etkili olduğu düşünülen kötü kokular ise mavi renkte belli bir dilimle gösterilir. Her bir dilimin açı derecesi ile kötü kokunun yayılma oranı gösterilmektedir. Daha büyük bir dilimle gösterilen kötü kokular, sisteme daha fazla yayılmışlardır.

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 vermemekte; *Large Class, Long Method, Long Parameter List* olmak üzere **3** adet kötü koku tespit etmektedir.

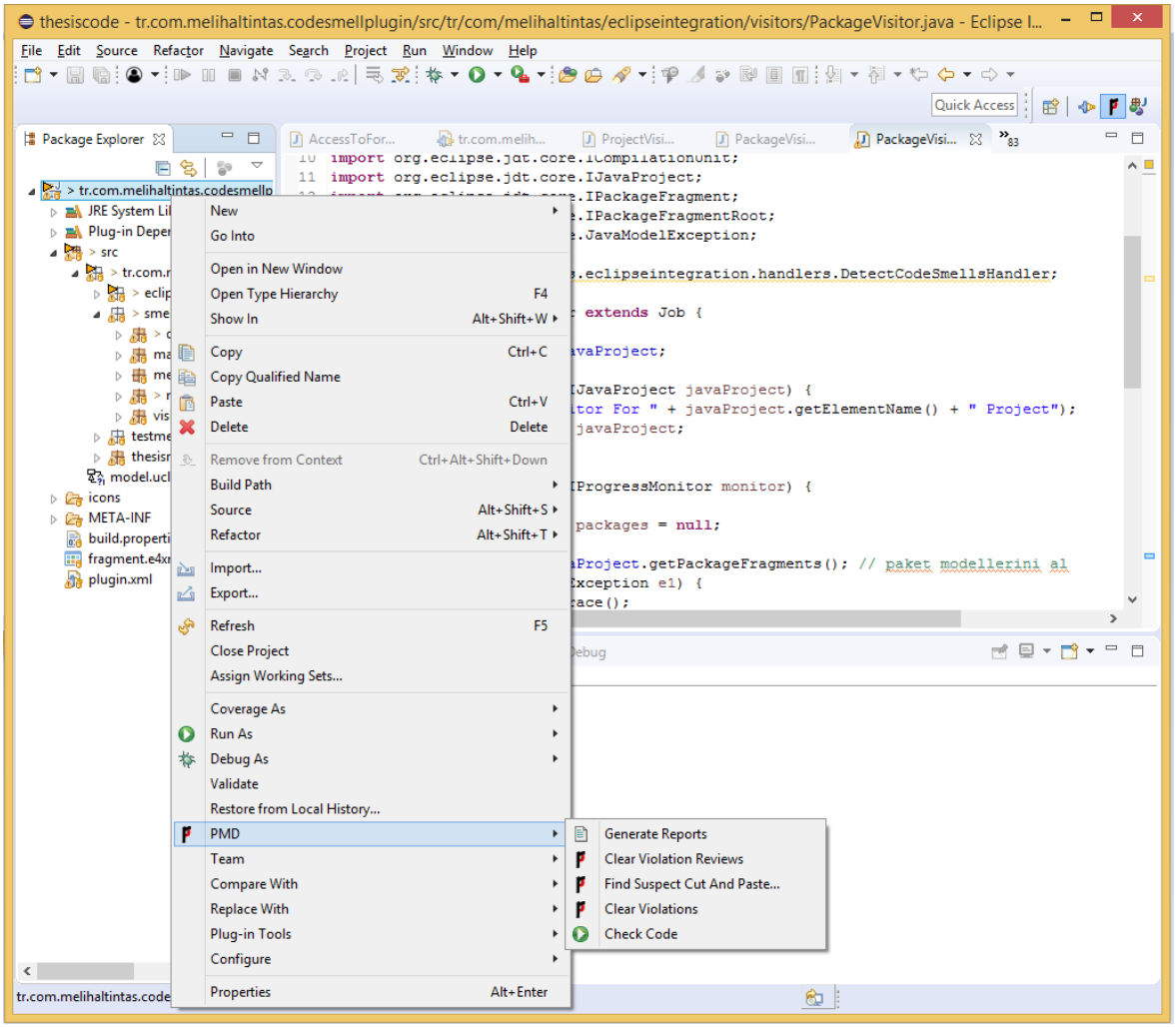
Avantajları:

- Eclipse'in en güncel versiyonlarında kullanılabilmektedir.
- Açık kaynak kodlu bir yazılım olup herhangi bir lisans gerektirmemektedir.
- Eklenti arka planda çalışabilmektedir. Bu nedenle kötü koku tespiti sırasında aynı anda kod yazmaya devam edebilirsiniz.
- Java, C, C++ gibi birçok programlama dili üzerinde çalışabilmektedir ve Eclipse, Netbeans gibi birçok geliştirme platformuna entegrasyonu vardır.

Dezavantajları:

- Statik analizde çok güçlü olsa bile, kötü koku tespitine çok yer vermemiştir.

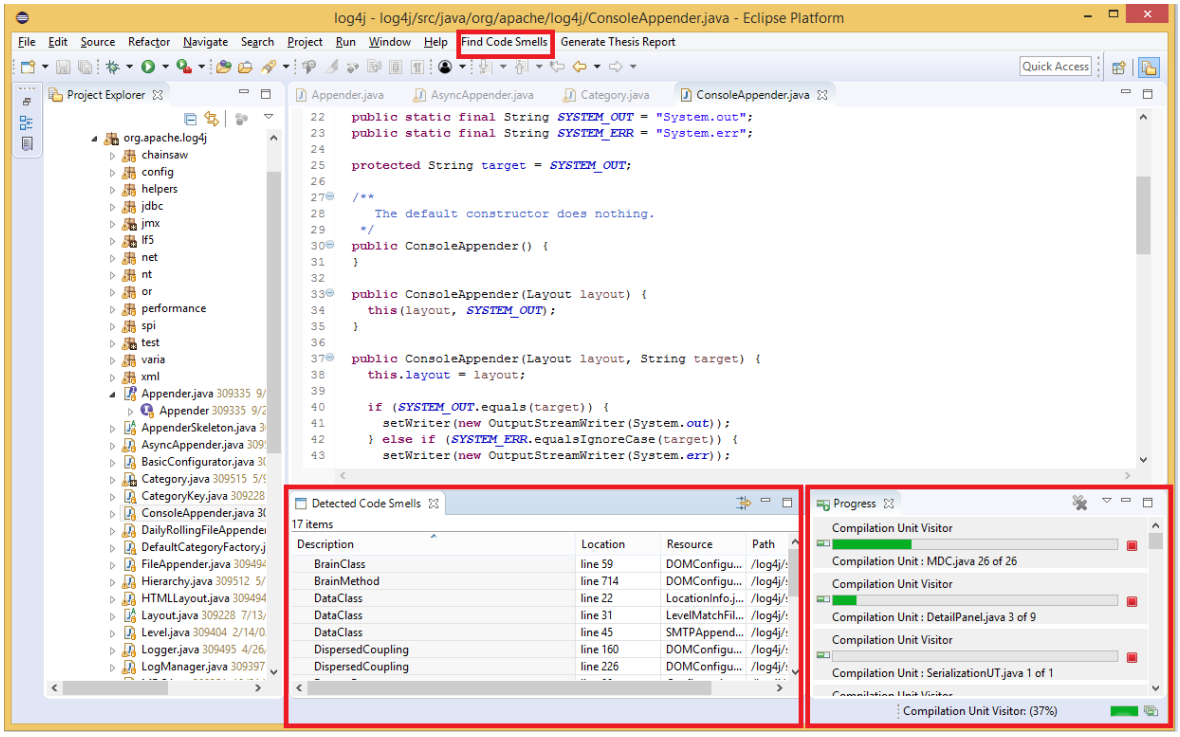
Eklentinin kullanımına Eclipse'e eklenen PMD bölümünden Generate Reports diyerek başlanabilir. Eklentinin kullanımı Şekil 2.16'da örneklenmiştir.



Şekil 2.16. PMD Eklentisinin Kullanımı

Bizim çalışmamız, aynı amaç doğrultusunda, kaynak kod içerisinde yer alan kötü kokuların tespiti için geliştirilmiştir. *God Class, Feature Envy, Brain Class, Data Class, Brain Method, Intensive Coupling , Dispersed Coupling, Shotgun Surgery, Refused Parent Bequest, Tradition Breaker* olmak üzere toplam **10** adet kötü koku tespiti yapabilmektedir.

Eklentimizin kullanımına Eclipse'e eklenen Find Code Smells bölümüne tıklayarak başlanabilir. Eklentinin kullanımı Şekil 2.17'de örneklenmiştir. Detected Code Smells adlı yeni bir ekrandan tespit edilen kötü kokular kullanıcılara sunulur. Process ekranından ise tespit işleminin hangi aşamada olduğu ve hangi sınıfların incelenmeye devam edildiği bilgisine ulaşılabilir.



Şekil 2.17. Eklentimizin Kullanım Örneği

Avantajları:

- Eclipse'in en güncel versiyonlarında kullanılabilmektedir.
- Açık kaynak kodlu bir yazılım olup herhangi bir lisans gerektirmemektedir.
- Eklenti arka planda çalışabilmektedir. Bu nedenle kötü koku tespiti sırasında aynı anda kod yazmaya devam edebilirsiniz.
- Java 8 ile gelen yeni tüm özellikleri destekleyip, yeni gelen özelliklerin içinde de (örn:lambda) kötü koku tespiti yapabilmektedir.
- IPLASMA aracı hariç diğer araçlardan daha fazla kötü koku tespiti yapabilmektedir. IPLASMA'ya göre avantajı, geliştirme ortamına entegre

olmasıdır. Bu sayede yazılımcı ve bakımcılar yazılımın kalitesini sürekli bir şekilde değerlendirebilmektedir.

- Kötü koku tespiti sırasında ayrıntılı olarak hangi adımda olunduğu, hangi sınıfların incelendiği, hangi sınıfların incelenmek üzere olduğu bilgisini kullanıcıya sunmaktadır.
- Detaylı rapor üretimi mevcuttur.

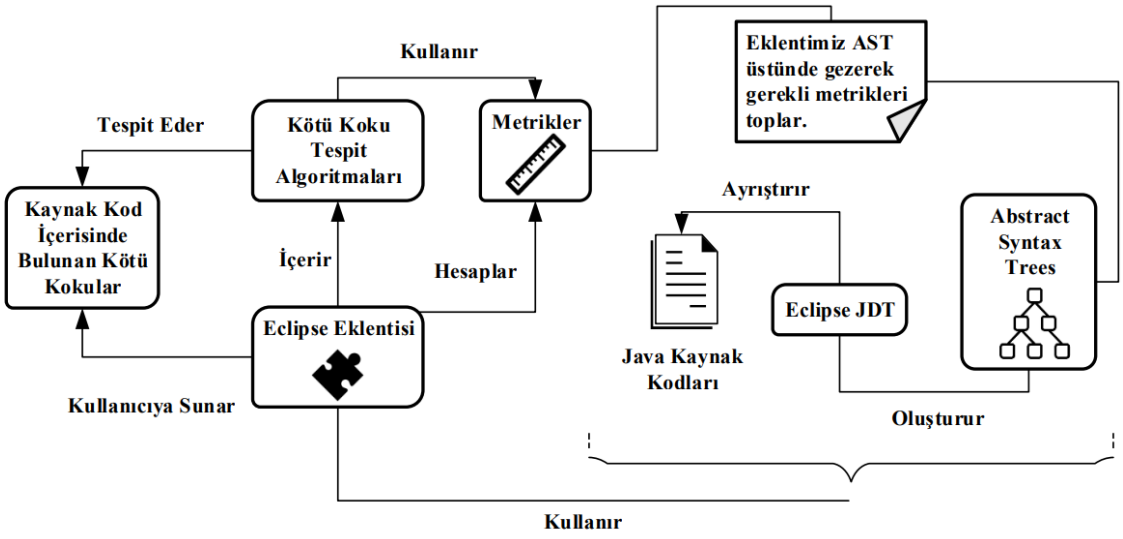
Eklentimizin diğer araçlar ile karşılaştırılması Çizelge 2.26'da verilmiştir.

Çizelge 2.26. Eklentimizin Diğer Araçlar İle Karşılaştırılması

	IPLASMA	JDeodorant	PMD	Stench Blossom	Bizim Eklentimiz
Tespit Edilen Kötü Koku Tipi	11	5	3	8	10
Geliştirme Ortamına Entegrasyon	✗	✓	✓	✓	✓
Arka Planda Çalışabilme	Eklenti Değil	✗	✓	✓	✓
Kötü Koku Otomatik Düzeltme	✗	✓	✓	✗	✗
Dil Destekleri	JAVA, C++	JAVA	JAVA, C, C++	JAVA	JAVA
Java 8 Desteği	✗	✗	✓	✗	✓
Bulunan Kötü Kokuların Detaylı Rapor Üretimi	✗	✗	✗	✗	✓

3. KÖTÜ KOKULARIN TESPİTİNİN GERÇEKLEŞTİRİMİ

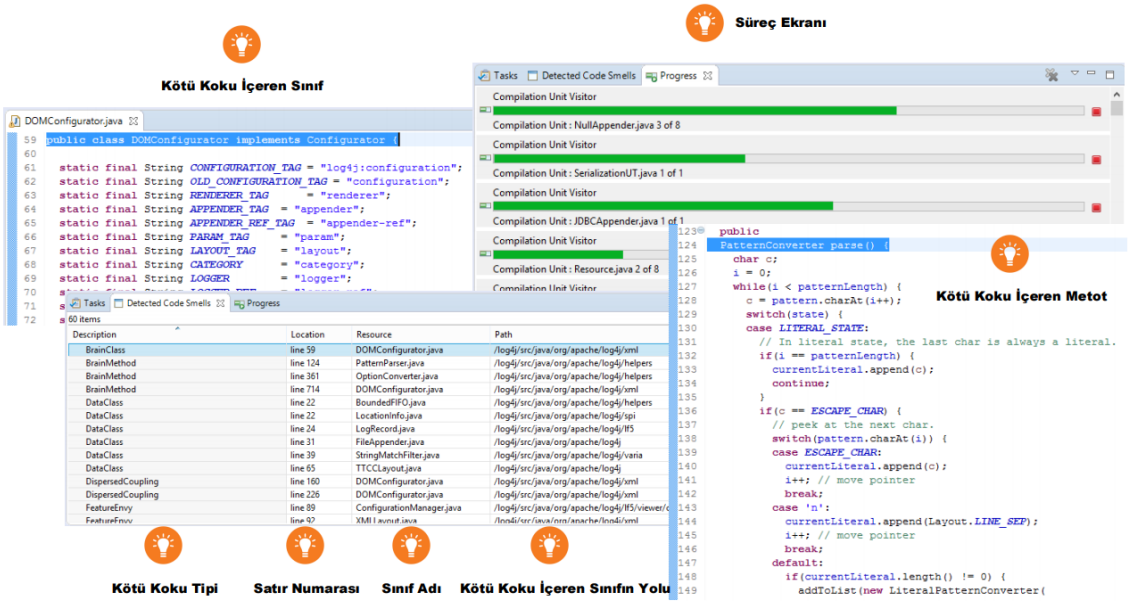
Eklentimiz Lanza ve Marinescu'nun kitabında [5] tanımlanmış, God Class, Feature Envy, Data Class, Brain Method, Brain Class, Intensive Coupling, Dispersed Coupling, Shotgun Surgery, Refused Parent Bequest, Tradition Breaker olmak üzere toplam **10** adet kötü kokuyu, yine aynı kitapta tanımı verilmiş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



Şekil 3.1. Kötü Kokuların Tespitinin Gerçekleştirimi

Tespit gerçekleştirimimiz Şekil 3.1'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 gerçekleştirilmiştir. Hesaplanan bu metrikler, kötü kokuların tespiti için kural tabanlı algoritmalara girdi olarak kullanılımıştır. Tespit edilen her bir kötü koku için Eclipse platformu üzerinde bunları gösteren işaretçi

(marker) oluşturulmuştur. Oluşturulan her bir işaretçi, yine bizim geliştirdiğimiz yeni bir ekran (view) üzerinden yazılımcı ve bakımcılara sunulmuştur. Bu ekran üzerinden yazılımcılar kötü koku barındıran sınıf ve metotları görerek, ilgili kötü kokuyu içeren sınıf veya metoda ulaşıp gerekli düzenlemeleri yapabilir hâle gelmişlerdir. Gerçekleştirilen eklentinin kaynak kodları Github [33] hesabımız üzerinden erişime ve kullanıma açılmıştır. 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 3.2'de verilmiştir.



Şekil 3.2. Geliştirilen Eklenti Görüntüsü

3.1. Metrik Hesaplanması ve Kötü Koku Tespiti için Geliştirdiğimiz Tasarım

Geliştirilen Eclipse eklentisi *Eclipse for RCP and RAP Developers (Version: Photon Milestone 5 (4.8.0M5)* üzerinde geliştirilmiştir. Tüm metrik değerleri ve kötü kokular, kendi yazdığımız Java sınıfları üzerinden tespit edilmiş, herhangi bir üçüncü parti kütüphane kullanılmamıştır. Yapılacak tüm statik analizler için Eclipse Java Geliştirme Araçları (Java Development Tools - JDT) kullanılmıştır. JDT, Java kaynak kodlarına erişmek ve bunları yönetmek için, *uygulama programlama arayüzleri (API)* sağlayan bir geliştirme çatısıdır. JDT Java kaynak kodlarına, ***Java modelleri*** veya

soyut sözdizimi ağaçları *(Abstract Syntax Trees - AST)* üzerinden erişim sağlar [42].

Java Model: Her Java projesi bir model üzerinden temsil edilir. Bu model, Java projesinin hafif ve hatasız bir temsilidir. Soyut söz dizim ağaçları kadar çok bilgi içermez, ancak oluşturulması daha hızlıdır [42].

Java modeli **org.eclipse.jdt.core** eklentisinde tanımlanmıştır. Java modeli, Çizelge 3.1 aracılığıyla açıklanabilen bir ağaç yapısı olarak temsil edilir [42].

Çizelge 3.1. Java Model Yapısı

Proje Elemanı	**Java Model Elemanı**	**Açıklama**
Java Projesi	IJavaProject	Diğer tüm objeleri içeren Java projesi
src klasörü bin klasörü	IPackageFragmentRoot	Kaynak ve binary dosyalarını tutan klasörler
Paket	IPackageFragment	Paketler IPackageFragmentRoot'un altındadır.
Java Dosyası	ICompilationUnit	Kaynak dosyalar
Tip, Nitelik, Metot	IType / IField / IMethod	Tipler, nitelikler, metotlar

Soyut Sözdizimi Ağaçları (Abstract Syntax Trees - AST): Java kaynak kodunun ayrıntılı bir ağaç temsilidir. Kaynak kodu değiştirmek, oluşturmak, okumak ve silmek için bir arayüz tanımlar [42].

AST için ana paket ***org.eclipse.jdt.core.dom*** paketidir ve ***org.eclipse.jdt.core*** eklentisinde bulunur [42].

Her bir Java elemanı, ASTNode sınıfının bir alt sınıfı olarak temsil edilir. Her soyut sözdizimi ağaç düğümü, temsil ettiği nesne hakkında özel bilgi sağlar [42].

Örneğin:

- MethodDeclaration – metotlar için
- VariableDeclarationFragment – değişkenler için
- SimpleName – herhangi bir Java anahtar sözcüğü olmayan katar (string) değeri için, Boolean değerler için (true veya false)

Sistemde yer alan tüm ASTNode bileşenleri ziyaretçi tasarım örüntüsüne (visitor design pattern) uygun şekilde geliştirilmiştir. Eclipse soyut sözdizimi ağaçlarında yer alan bu düğümleri gezmek ve üzerilerinden bilgi toplamak için ASTVisitor adlı sınıfları sunmaktadır. ASTVisitor sınıfında düğümleri gezmek için 2 temel metot bulunmaktadır:

- **public boolean visit(T node)** – Soyut sözdizimi ağacında yer alan bir düğüm üzerinde belli bir işlemi gerçekleştirmek için düğümü ziyaret eder. Metodun dönüş değeri *true* ise parametre olarak verilen düğümün çocuk düğümleri de bir sonraki işlemde ziyaret edilecektir, ancak dönüş değeri *false* ise çocuk düğümler ziyaret edilmeyecektir. Bu metodun varsayılan gerçekleştirimi hiçbir şey yapmadan *true* dönmektir. ASTVisitor sınıfından türeyen sınıflarda bu metot yeniden gerçeklenerek ziyaret sırasında hangi işlemlerin yapılacağı belirlenir.
- **public void endVisit(T node) -** Verilen düğümün tüm çocuklarının ziyaret edilmesinden sonra veya visit metodunun *false* dönmesi hâlinde bu metot çağrılır. Bu metodun varsayılan gerçekleştirimi hiçbir şey yapmamaktadır. ASTVisitor sınıfından türeyen sınıflarda bu metot yeniden gerçeklenerek ziyaret bitimi sırasında hangi işlemlerin yapılacağı belirlenir.

Java kaynak kodlarından soyut sözdizimi ağaçları oluşturulduğunda, bu ağaçlarda yer alan her bir düğümün Java sınıfı olarak karşılığı ASTNode'dur. Her bir düğüm sınıfı ASTVisitor sınıfından türetilmiş sınıflar tarafından ziyaret edilebilir. Bunu ASTNode sınıflarında bulunan accept (ASTNode.accept(ASTVisitor)) metodu sağlar. Gezilmesi istenen düğümün accept metoduna parametre olarak ASTVisitor sınıfından türetilmiş herhangi bir sınıfın verilmesiyle düğüm üzerinde gezilebilinir.

Eclipse'in sağladığı bu özel çatı sayesinde, her bir Java kaynak dosyasından soyut sözdizimi ağaçları oluşturup her bir düğümü ziyaret edebilmek mümkün hâle gelmiştir. Her bir düğüm ziyaret edilerek istenen metrikler hesaplanabilecektir.

Bu nedenle eklentimiz için öncelikle Java kaynak dosyalarından soyut sözdizimi ağaçlarını oluşturacak sınıfları daha sonra da bu ağacı ziyaret ederek metrik değerlerini hesaplayacak sınıfları yazdık.

İlk olarak, eklentimiz üzerinden kötü koku tespiti yapılmak istendiğinde Eclipse çalışma ortamı (workspace) bulunmuştur. Daha sonra çalışma ortamındaki köke ulaşılmıştır. Ardından çalışma ortamında yer alan tüm projeler ziyaret edilmeye başlanmıştır. Bu işlemleri yapan *DetectCodeSmellHandler* sınıf içeriği Şekil 3.3'te verilmiştir.

```
@Execute
public void execute(Shell shell) {

    BugInformation.getInstance();

    IWorkspace workspace = ResourcesPlugin.getWorkspace(); // calisma ortamini bul
    IWorkspaceRoot root = workspace.getRoot();  // koke ulas

    try { // daha onceden isaretlenen kotu kokulari temizle (yeniden calisdiginda bastan yapmak icin)

        root.deleteMarkers("tr.com.melihaltintas.codesmellmarker", true, IProject.DEPTH_INFINITE);
        PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage()
                .showView("tr.com.melihaltintas.eclipseintegration.parts.CodeSmellView");

    } catch (PartInitException e) {
        e.printStackTrace();
    } catch (CoreException e) {
        e.printStackTrace();
    }

    IProject[] projects = root.getProjects();  // tum projeleri al
    ProjectVisitor projectVisitor = new ProjectVisitor(projects);  // tum projeleri gezmek icin visitor yarat

    projectVisitor.setUser(true);

    projectVisitor.schedule(); // tum projeleri gezmeye basla

}
```

Şekil 3.3. Eclipse Eklentisi - Tüm Projelerin Gezilmesi

Projeler gezilirken tüm projelerin Java projesi olduğundan emin olunmuş, projelerin içerisindeki tüm paketler ziyaret edilmeye başlanmıştır. Bunu gerçekleyen *ProjectVisitor* sınıf içeriği Şekil 3.4'te verilmiştir.

```
@Override
protected IStatus run(IProgressMonitor monitor) {

    SubMonitor subMonitor = SubMonitor.convert(monitor, projects.length);
    for (int i = 0; i < projects.length; i++) { // butun projeleri gez
        try {
            if (projects[i].isNatureEnabled("org.eclipse.jdt.core.javanature")) { // sadece java projeleri

                IJavaProject javaProject = JavaCore.create(projects[i]); // projeden model olustur
                subMonitor.setTaskName(
                        "Project : " + javaProject.getElementName() + " " + (i + 1) + " of " + projects.length);
                subMonitor.split(i + 1);
                visitPackages(javaProject); // projenin icerisindeki tüm paketleri gez
            }
        } catch (CoreException e) {
            e.printStackTrace();
        }
    }

    return Status.OK_STATUS;
}

private void visitPackages(IJavaProject javaProject) {
    PackageVisitor packageVisitor = new PackageVisitor(javaProject);
    packageVisitor.schedule();
}
```

Şekil 3.4. Eclipse Eklentisi - Tüm Paketlerin Gezilmesi

Daha sonra gezilen her bir paket içerisinde yer alan Java dosyaları ziyaret edilmeye başlanmıştır. Bunu gerçekleyen *PackageVisitor* sınıf içeriği Şekil 3.5'te verilmiştir.

```
protected IStatus run(IProgressMonitor monitor) {

    IPackageFragment[] packages = null;
    try {
        packages = javaProject.getPackageFragments(); // paket modellerini al
    } catch (JavaModelException e1) {
        e1.printStackTrace();
    }
    SubMonitor subMonitor = SubMonitor.convert(monitor, packages.length);
    for (int i = 0; i < packages.length; i++) {
        subMonitor.setTaskName(
                "Package : " + packages[i].getElementName() + " " +(i + 1) + " of " + packages.length);
        subMonitor.split(i + 1);
        try {
            if (packages[i].getKind() == IPackageFragmentRoot.K_SOURCE) {
                visitCompilationUnits(packages[i]); // paket icerisindeki tum java dosyalarini gez

            }
        } catch (JavaModelException e) {

            e.printStackTrace();
        }
    }
    return Status.OK_STATUS;
}

private void visitCompilationUnits(IPackageFragment currentPackage) {
    try {
        ICompilationUnit[] units = currentPackage.getCompilationUnits();
        CompilationUnitVisitor compilationUnitVisitor = new CompilationUnitVisitor(units);
        compilationUnitVisitor.schedule(); // tum java dosyalarini gez
    } catch (JavaModelException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Şekil 3.5. Eclipse Eklentisi - Tüm Kaynak Dosyaların Gezilmesi

Daha sonra gezilen her bir Java dosyası için soyut sözdizim ağaçları oluşturulup bizim geliştirdiğimiz tespit sınıflarına verilerek kötü kokular tespit edilmeye başlanmıştır. Bunu gerçekleyen *CompilationUnitVisitor* sınıf içeriği Şekil 3.6'da verilmiştir.

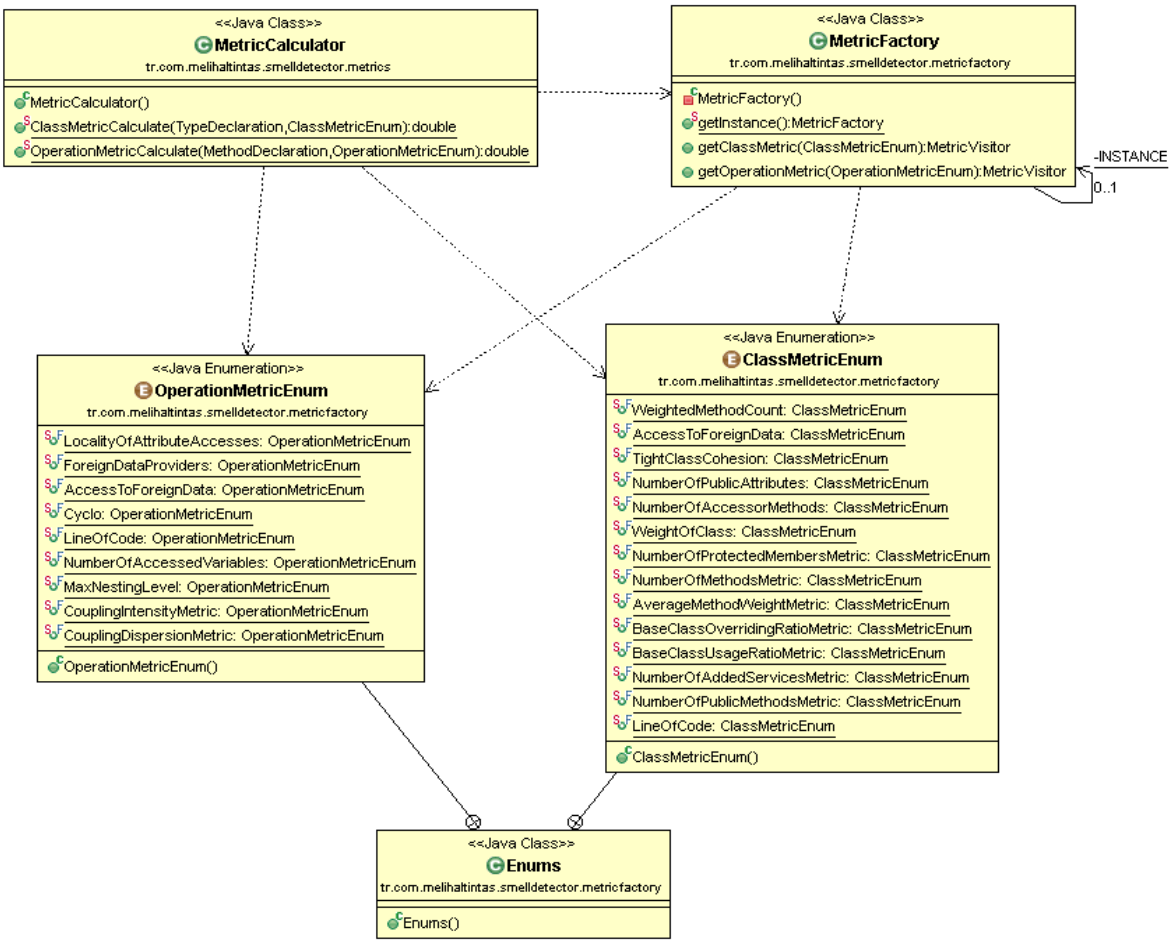
```
protected IStatus run(IProgressMonitor monitor) {
    try {
        SubMonitor subMonitor = SubMonitor.convert(monitor, units.length);
        for (int i = 0; i < units.length; i++) { // her java dosyasini gez
            subMonitor.setTaskName(
                    "Compilation Unit : " + units[i].getElementName() + " " + (i + 1) + " of " + units.length);
            subMonitor.split(i + 1);

            CompilationUnit parse = parse(units[i]); // java dosyasindan AST olustur

            GodClassDetector godClassDetector = new GodClassDetector();
            DataClassDetector dataClassDetector = new DataClassDetector();
            FeatureEnvyDetector featureEnvyDetector = new FeatureEnvyDetector();
            BrainClassDetector brainClassDetector = new BrainClassDetector(parse);
            IntensiveCouplingDetector intensiveCouplingDetector = new IntensiveCouplingDetector();
            DispersedCouplingDetector dispersedCouplingDetector = new DispersedCouplingDetector();
            ShotgunSurgeryDetector shotgunSurgeryDetector = new ShotgunSurgeryDetector();
            RefusedParentBequestDetector refusedParentBequestDetector = new RefusedParentBequestDetector();
            TraditionBreaker traditionBreaker = new TraditionBreaker();

            parse.accept(godClassDetector);
            parse.accept(dataClassDetector);
            parse.accept(featureEnvyDetector);
            parse.accept(brainClassDetector);
            parse.accept(intensiveCouplingDetector);
            parse.accept(dispersedCouplingDetector);
            parse.accept(shotgunSurgeryDetector);
            parse.accept(refusedParentBequestDetector);
            parse.accept(traditionBreaker);

        }
    } catch (Exception ex) {
        System.err.println(ex);
    }

    return Status.OK_STATUS;

}

private CompilationUnit parse(ICompilationUnit unit) {
    ASTParser parser = ASTParser.newParser(AST.JLS9);
    parser.setKind(ASTParser.K_COMPILATION_UNIT);
    parser.setSource(unit); // kaynak dosyayi ver
    parser.setEnvironment(null, null, null, true);
    parser.setResolveBindings(true);

    return (CompilationUnit) parser.createAST(null); // AST olustur
}
```

Şekil 3.6. Eclipse Eklentisi - Soyut Sözdizim Ağaçlarının Oluşturulması

Yazdığımız tüm kötü koku tespit sınıfları, tespit edilmesi planlanan kötü kokuyla ilgili metriklerin eşik değerlerinin sağlanması durumunda tespit yapan sınıflardır. Bu

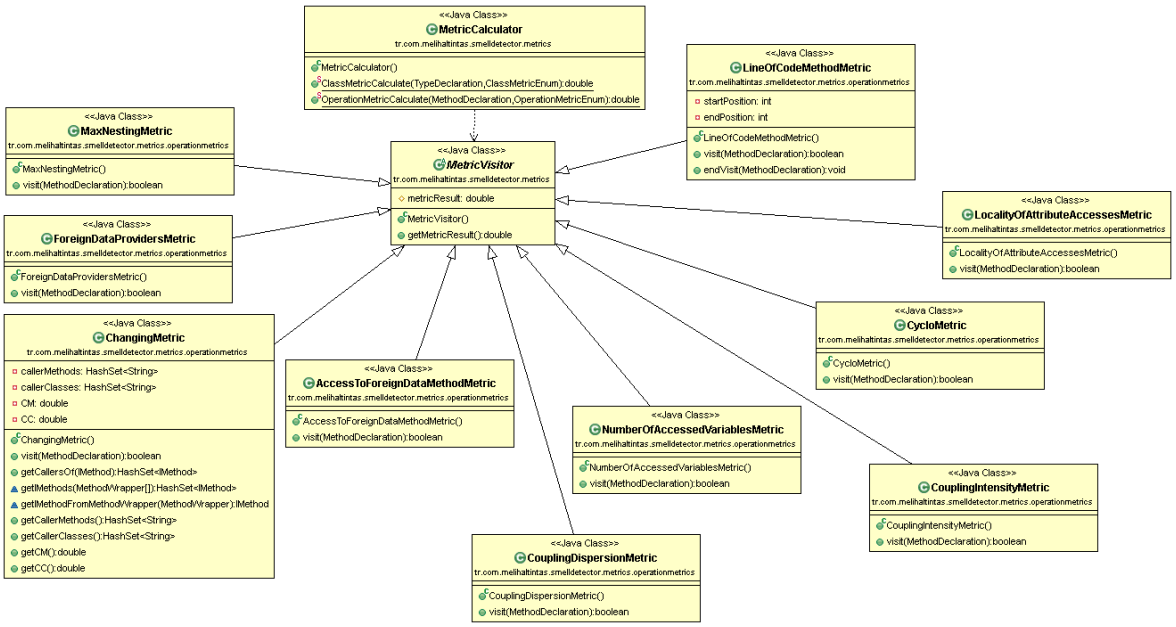
sınıfların tespit yapabilmesi için; soyut sözdizimi ağaçlarındaki düğümler gezilerek, sınıf ve metotlar üzerinden metriklerin toplanması gerekmektedir.

Literatürde sınıf üzerinden hesaplanacak metrikler ve metot üzerinden hesaplanacak metrikler farklılık göstermektedir. Bu nedenle kod geliştirimi öncesi yaptığımız tasarımda sınıf ve metot metriklerini birbirinden ayırmak için 2 farklı enum kullanıldı. Sınıf metrikleri *ClassMetricEnum'u* içerisinde, metot metrikleri ise *OperationMetricEnum*'u içerisinde tanımlandı. İlgili metrik değerlerini hesaplamayı sağlayacak sınıfların üretimi için fabrika (factory) tasarım örüntüsünü kullanıldı. Eklentimizin kaynak kodlarında yer alan *MetricFactory* sınıfı, ilgili metriği hesaplamayı sağlayacak sınıfları üreten, sistemde sadece bir adet bulunan (singleton) bir fabrika (factory) sınıfıdır. Üretim aşamasında sınıf metriği hesaplanmasını sağlayacak bir sınıfı üretmek için, *ClassMetricEnum* içerisinde tanımlanan değerlerden birini *MetricFactory* sınıfı içerisinde yer alan *getClassMetric* metoduna parametre olarak vermek yeterlidir. Metot metriği hesaplanmasını sağlayacak bir sınıfı üretmek için ise, *OperationMetricEnum* içerisinde tanımlanan değerlerden birini *MetricFactory* sınıfı içerisinde yer alan *getOperationMetric* metoduna parametre olarak vermek yeterlidir. *MetricFactory* sınıfı, *MetricCalculator* sınıfı üzerinden yönetilerek hizmet vermektedir. *MetricCalculator* içerisindeki *ClassMetricCalculate* metoduna, bir sınıf ve o sınıf üzerinde hangi sınıf metriğinin hesaplanacağı verilerek sınıf için metrik değeri hesaplanabilir. Aynı işlem metotlar içinse *OperationMetricCalculate* metoduna, bir metot ve o metot üzerinde hangi metot metriğin hesaplanacağı verilerek metrik değeri hesaplanabilir. Detaylı UML diyagramı Şekil 3.7'de verilmiştir.



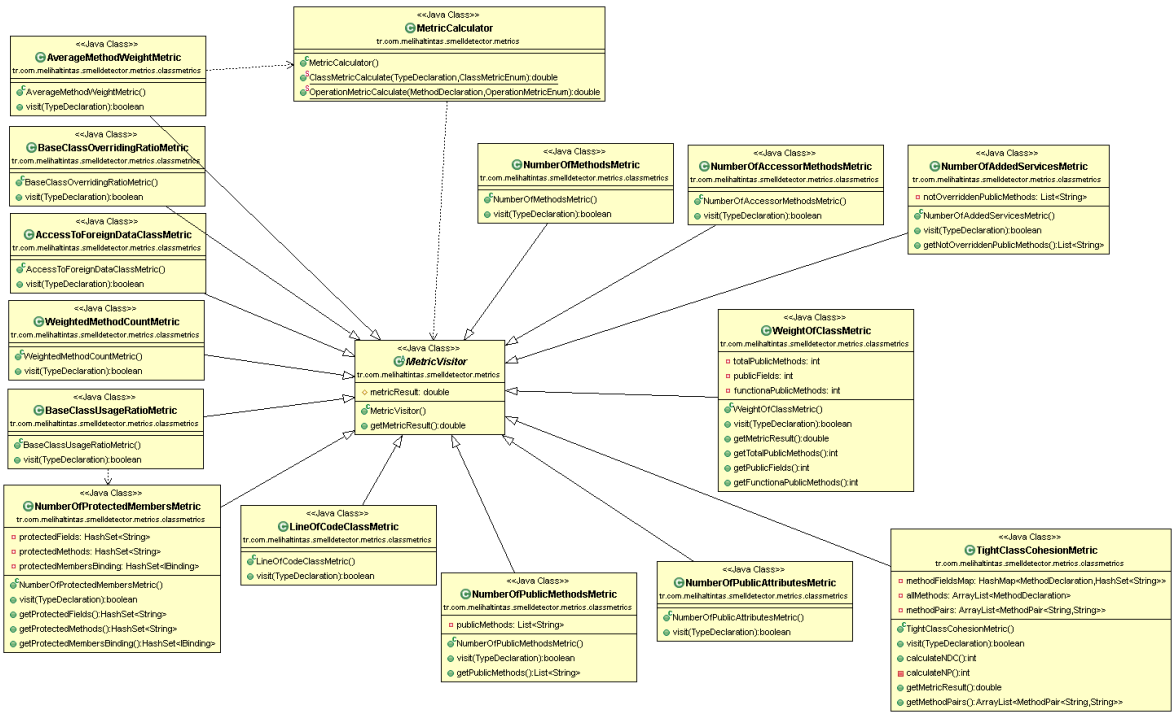
Şekil 3.7. Metrik Hesaplayıcı Sınıfların Üretimi

Eclipse'in soyut sözdizimi ağaçlarını gezmek için, *ASTVisitor* sınıflarını sunduğunu belirtmiştik. Biz eklentimiz için bu sınıftan *MetricVisitor* adlı bir sınıf türettik. Bu sınıfı tüm metrik hesaplayıcı sınıfların ata sınıfı olarak belirledik. İçerisinde hesaplanacak metrik değeri tutan bir nitelik ve bu niteliğe erişimi sağlayan *getMetricResult* adlı bir metot tanımladık. Bu sayede *MetricVisitor* sınıfından türetilen her sınıf, bu nitelik ve metodu kalıtımla almış oldu. *MetricVisitor* sınıfından türemiş, metrik değeri hesaplamaya yarayan sınıfların visit metotları içerisinde metrik hesaplama algoritmalarımız tanımlayarak metrik değerinin hesaplanması sağladık. Çıkan sonucu *metricResult* değişkenine atayarak istenildiğinde bu değere ulaşılmasını sağladık. Eklentimizde yer alan metot metrik değerlerini hesaplayan sınıf hiyerarşisi Şekil 3.8'de verilmiştir.



Şekil 3.8. Metot Metriklerini Hesaplayan Sınıfların Hiyerarşisi

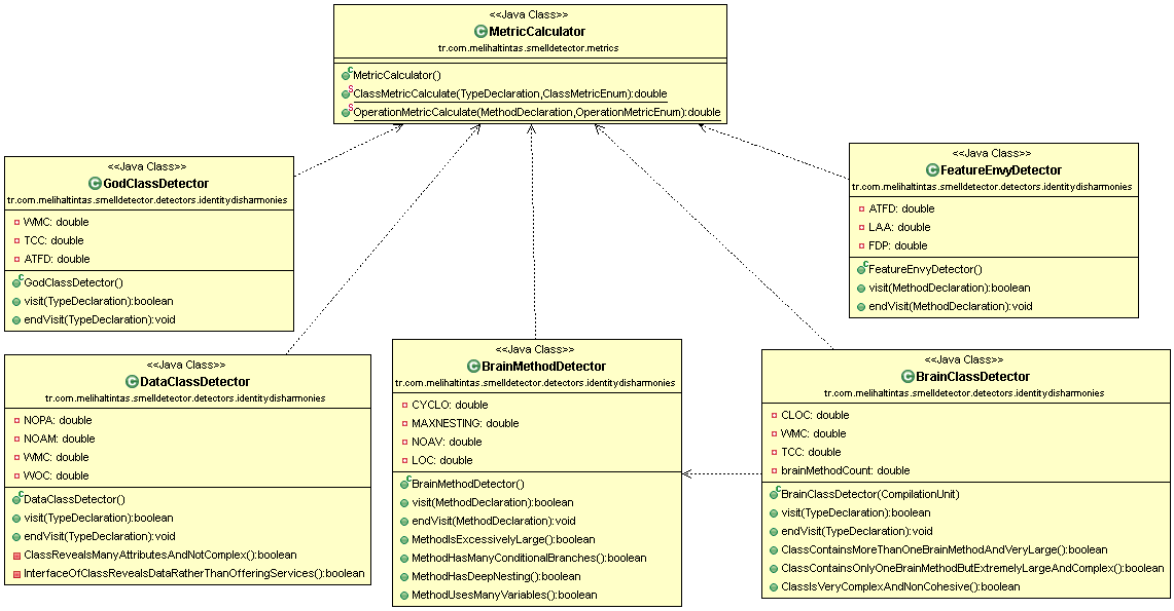
Eklentimizde yer alan sınıf metrik değerlerini hesaplayan sınıf hiyerarşisi Şekil 3.9'da verilmiştir.



Şekil 3.9 Sınıf Metriklerini Hesaplayan Sınıfların Hiyerarşisi

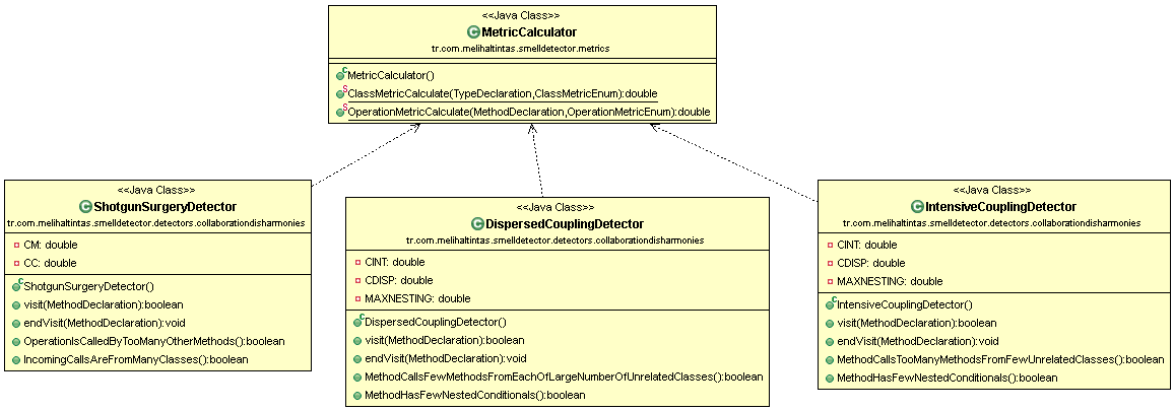
Tüm metriklerin hesaplanacağı sınıflar yazıldıktan sonra kötü kokuların tespiti için sınıfların gerçekleştirimini yaptık.

Kimlik uyumsuzlukları grubunda yer alan kötü koku tespit sınıfları Şekil 3.10'da verilmiştir.



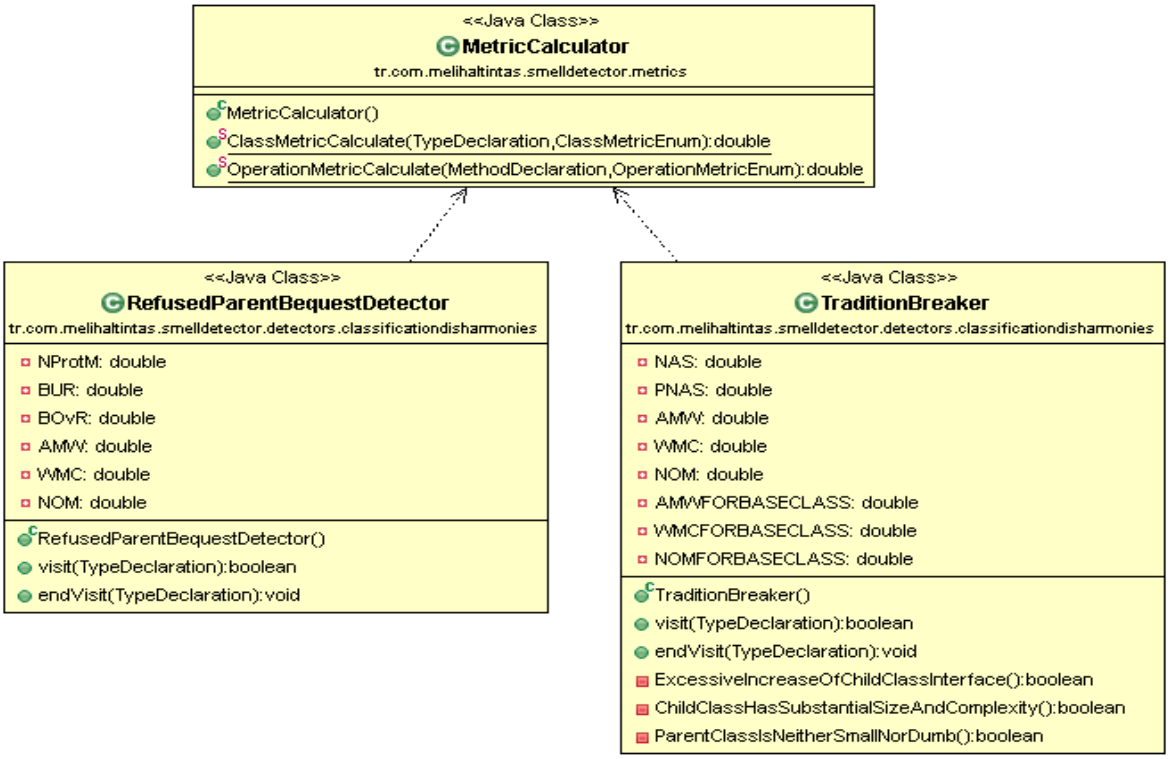
Şekil 3.10. Kimlik Uyumsuzlukları Kötü Koku Grubu Sınıf Diyagramı

İşbirliği uyumsuzlukları grubunda yer alan kötü koku tespit sınıfları Şekil 3.11'de verilmiştir.



Şekil 3.11. İşbirliği Uyumsuzlukları Kötü Koku Grubu Sınıf Diyagramı

Sınıflama uyumsuzlukları grubunda yer alan kötü koku tespit sınıfları Şekil 3.12'de verilmiştir.



Şekil 3.12. Sınıflama Uyumsuzlukları Kötü Koku Grubu Sınıf Diyagramı

Genel mimarimizin daha iyi anlaşılması için örnek olarak God Class kötü kokusunun nasıl tespit edildiği bu bölümde anlatılacaktır. God Class kötü kokusunun tespitini yapan sınıf *GodClassDetector* sınıfıdır. İçeriği Şekil 3.13'te verilmiştir.

```
public class GodClassDetector extends ASTVisitor {

    private double WMC;
    private double TCC;
    private double ATFD;

    public boolean visit(TypeDeclaration typeDeclaration) {

        WMC = MetricCalculator.ClassMetricCalculate(typeDeclaration, ClassMetricEnum.WeightedMethodCount);

        TCC = MetricCalculator.ClassMetricCalculate(typeDeclaration, ClassMetricEnum.TightClassCohesion);

        ATFD = MetricCalculator.ClassMetricCalculate(typeDeclaration, ClassMetricEnum.AccessToForeignData);

        return true;
    }

    @Override
    public void endVisit(TypeDeclaration typeDeclaration) {


        if (ATFD > MetricThresholds.FEW_THRESHOLD && WMC >= MetricThresholds.WMC_VERY_HIGH_THRESHOLD
                && TCC < MetricThresholds.ONE_THIRD_THRESHOLD) {

            MarkerCreator.addMarker(typeDeclaration, MarkerType.GodClass);

        }
        super.endVisit(typeDeclaration);
    }

}
```

Şekil 3.13. God Class Kötü Koku Tespiti Gerçekleştirimi

Genel mimaride *MetricCalculator* sınıfının, sınıf veya metotlar için hesaplanması gereken metrik değerlerini hesapladığını anlatmıştık. God Class kötü kokusu, sınıflar üzerinde gözlemlenen bir kötü koku türüdür. Bu yüzden sınıf metrikleri hesaplanacağından *MetricCalculator* sınıfının *ClassMetricCalculate* metodundan yararlanılmıştır.

Literatüre göre bir sınıfın God Class kötü kokusundan etkilenmiş olması için; *WMC* metrik değerinin **47**'den büyük ve eşit, *TCC* metrik değerinin **1/3**'ten küçük, *ATFD* değerinin ise **3**'ten büyük olması gerekmektedir. *Visit* metodu içerisinde bu değerler hesaplanmış, *endVisit* metodunda ise metrik sonuçları ile metrik eşik değerleri karşılaştırılmıştır. Karşılaştırma sonucunda, kriterler sağlanıyorsa ilgili sınıf God Class olarak işaretlenerek, Eclipse üzerinde tespit edilen kötü kokuların gösterildiği tabloya eklenmiştir.

Metriklerin hesaplanmasının anlatımı *WMC* metriği üzerinden detaylandırılarak anlatılacaktır. *WMC* metriğinin hesaplanması için, *MetricCalculator* sınıfına o an incelen sınıf ve *ClassMetricEnum* değerlerinden biri olan *WeightedMethodCount* enumu verilmiştir. *MetricCalculator* sınıfının içeriği Şekil 3.14'te verilmiştir.

```
public class MetricCalculator {

    public static double ClassMetricCalculate(TypeDeclaration type, ClassMetricEnum metric) {
        MetricVisitor metricVisitor = MetricFactory.getInstance().getClassMetric(metric);
        type.accept(metricVisitor);
        return metricVisitor.getMetricResult();
    }

    public static double OperationMetricCalculate(MethodDeclaration method, OperationMetricEnum metric) {
        MetricVisitor metricVisitor = MetricFactory.getInstance().getOperationMetric(metric);
        method.accept(metricVisitor);
        return metricVisitor.getMetricResult();
    }

}
```

Şekil 3.14. MetricCalculator Sınıf İçeriği

Her bir metrik hesaplayan sınıf, *MetricVisitor* sınıfından türediği için soyut sözdizimi ağaçlarında gezebilme yeteneğine sahiptir. *WeightedMethodCountMetric* sınıfı da bu sınıflardan biridir. Eclipse üzerinde Java kaynak kodlarından oluşturulan soyut sözdizimi ağaçlarında yer alan her sınıf (*TypeDeclaration*) ve her metot (*MetotDeclaration*) *accept* metotları üzerinden parametre olarak verilen *ASTVisitor* sınıflarıyla ziyaret edilebilmektedir. Yani ziyaret edilmek istenen sınıf veya metotların *accept* metoduna, *MetricFactory* üzerinden üretilen metrik hesaplayıcı sınıflar parametre verilerek gezilebilir. Gezilen sınıf veya metot üzerinden metrik değeri hesaplanarak *MetricCalculator* sınıfı içerisinde yer alan *ClassMetricCalculate* veya *OperationMetricCalculate* metotları üzerinden sunulur.

MetricCalculator sınıfı *MetricFactory* sınıfını kullanarak, ziyaret edilecek sınıfının *WMC* metrik değerini hesaplamayı sağlayan *WeightedMethodCountMetric* sınıfını yaratarak ilgili düğümü ziyaret etmek üzere *accept* metoduyla düğüme vermiştir.

MetricFactory sınıfı içerisinde yer alan *getClassMetric* metodunu kullanarak *WeightedMethodCountMetric* sınıfını yaratmıştır. *GetClassMetric* metodunun içeriği Şekil 3.15'te verilmiştir.

```
public MetricVisitor getClassMetric(ClassMetricEnum classMetric) {
    switch (classMetric) {
    case WeightedMethodCount:
        return new WeightedMethodCountMetric();
    case AccessToForeignData:
        return new AccessToForeignDataClassMetric();
    case TightClassCohesion:
        return new TightClassCohesionMetric();
    case NumberOfPublicAttributes:
        return new NumberOfPublicAttributesMetric();
    case NumberOfAccessorMethods:
        return new NumberOfAccessorMethodsMetric();
    case WeightOfClass:
        return new WeightOfClassMetric();
    case NumberOfProtectedMembersMetric:
        return new NumberOfProtectedMembersMetric();
    case NumberOfMethodsMetric:
        return new NumberOfMethodsMetric();
    case AverageMethodWeightMetric:
        return new AverageMethodWeightMetric();
    case BaseClassOverridingRatioMetric:
        return new BaseClassOverridingRatioMetric();
    case BaseClassUsageRatioMetric:
        return new BaseClassUsageRatioMetric();
    case NumberOfAddedServicesMetric:
        return new NumberOfAddedServicesMetric();
    case NumberOfPublicMethodsMetric:
        return new NumberOfPublicMethodsMetric();
    case LineOfCode:
        return new LineOfCodeClassMetric();
    default:
        return null;
    }

}
```

Şekil 3.15. GetClassMetric Metodu İçeriği

GetClassMetric metoduna parametre olarak *WeightedMethodCount* enum değeri verildiğinde, *WMC* metric değerinin hesaplanması için *WeightedMethodCountMetric* sınıfı yaratılmaktadır. *WeightedMethodCountMetric* sınıfının içeriği Şekil 3.16'da verilmiştir.

```
//WMC - Weighted Method Count
//The sum of the statical complexity of all methods of a class. The CYCLO metric
//is used to quantify the method's complexity
//Used for Refused Parent Bequest, Tradition Breaker, God Class, DataClass, Brain Class

public class WeightedMethodCountMetric extends MetricVisitor{

    public boolean visit(TypeDeclaration typeDeclaration){

        MethodDeclaration[] methods = typeDeclaration.getMethods();

        for (MethodDeclaration method : methods) {

            CycloVisitor cycloVisitor = new CycloVisitor(method);

            try{
                method.accept(cycloVisitor);
                double cyclo = cycloVisitor.getMetricResult();
                metricResult += cyclo;
            }catch(Exception e){
                e.printStackTrace();
            }

        }
        return true;
    }

}
```

Şekil 3.16. WeightedMethodCountMetric Sınıf İçeriği

Bir metodun karmaşıklığı *CYCLO* metriği ile bulunur. *WMC* metriği de tüm sınıfın karmaşıklığıdır. Yani ilgili sınıfta yer alan tüm metotların karmaşıklıklarının toplamıdır. *CYCLO* metrik değeri, *CycloVisitor* sınıfının *WMC* metrik değeri hesaplanmak istenen sınıfın içerisindeki tüm metotları ziyaret etmesiyle hesaplanır. Boş bir metot için *CYCLO* değeri 1'dir. Metodun içerisinde geçen her bir for, while, if, case gibi ifadeler için bu değer 1 arttırılır. *CycloVisitor* sınıfının bir bölümü Şekil 3.17'de verilmiştir.

GodClassDetector sınıfı için *WMC* metriğinin nasıl hesaplandığını anlattık. *TCC* ve *ATFD* metrikleri için de benzer işlemler yapıldı. 3 metriğin değerlerinin hesaplanması sonucunda, sınıfın God Class kötü kokusu içerdiği veya içermediği kararı verilebilir hâle geldi. Diğer metriklerin hesabı ve kötü koku tespit sınıfları için eklentimizin kodlarına Github hesabımız üzerinden erişebilirsiniz [33].

```
public class CycloVisitor extends MetricVisitor {

    private MethodDeclaration currentMethod;

    public CycloVisitor(MethodDeclaration currentMethod) {
        if (!((currentMethod.getModifiers() & Modifier.ABSTRACT) > 0)) {
            metricResult = 1; // base cyclo for method = 1
        } else {
            metricResult = 0;
        }

        this.currentMethod = currentMethod;

    }

    public boolean visit(CatchClause catchClause) {
        metricResult++; // catch clause cyclo ++
        return true;
    }

    public boolean visit(ConditionalExpression conditionalExpression) { // Expression ? Expression : Expression
        metricResult++; // conditionalExpression cyclo ++
        booleanOperatorCountInExpression(conditionalExpression.getExpression());
        return true;
    }

    public boolean visit(DoStatement doStatement) {
        metricResult++; // doStatement cyclo ++
        booleanOperatorCountInExpression(doStatement.getExpression());
        return true;
    }

    public boolean visit(ForStatement forStatement) {
        metricResult++; // forStatement cyclo ++
        booleanOperatorCountInExpression(forStatement.getExpression());
        return true;
    }

    public boolean visit(IfStatement ifStatement) {
        metricResult++; // ifStatement cyclo ++
        booleanOperatorCountInExpression(ifStatement.getExpression());
        return true;
    }

    public boolean visit(SwitchCase switchCase) {
        if (!switchCase.isDefault())
            metricResult++;
        booleanOperatorCountInExpression(switchCase.getExpression());
        return true;
    }
```

Şekil 3.17. CycloVisitor Sınıfının Bir Bölümü

4. KÖTÜ KOKU ANALİZLERİ

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 için de 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.

Çizelge 4.1. Projelerdeki Kötü Kokuların ve Hatalarının İstatistiksel Dağılımı

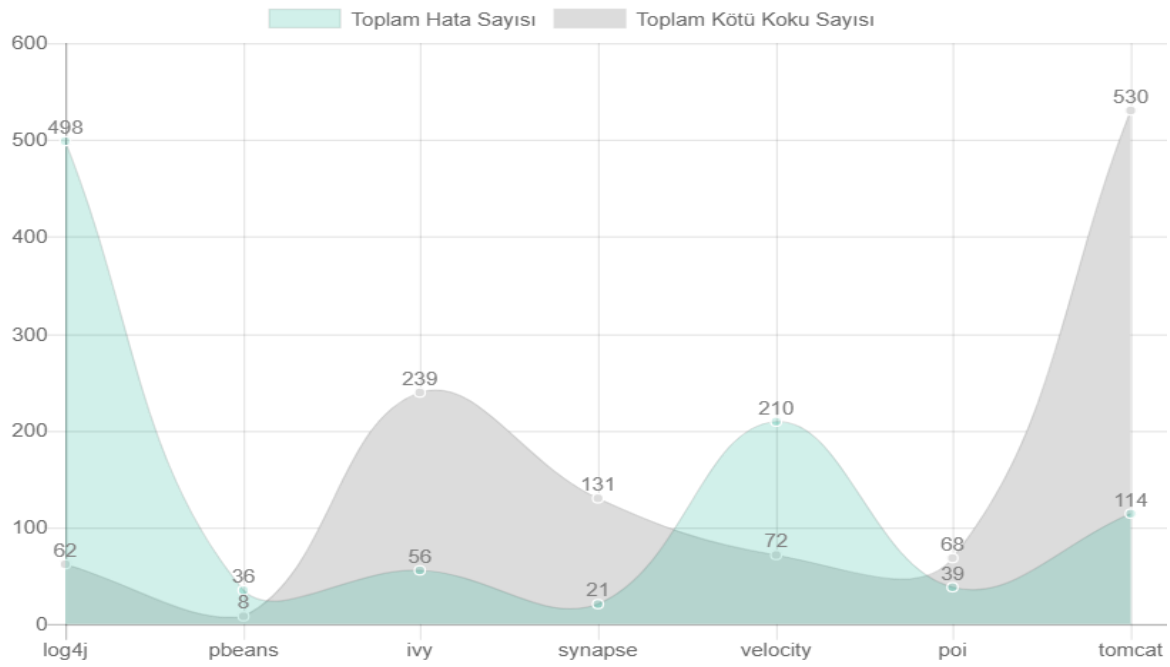
	log4j		pbeans		ivy		synapse		velocity		poi		tomcat	
Toplam Sınıf Sayısı	223		30		422		161		221		370		1086	
Hatalı Sınıf Sayısı / Hata İçermeyen Sınıf Sayısı	189	34	20	10	40	382	16	145	147	74	37	333	77	1009
Kötü Koku İçeren Sınıf Sayısı / Kötü Koku İçermeyen Sınıf Sayısı	43	180	6	24	106	316	63	98	36	185	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 ile kötü koku arasında varoluşsal bir ilişkinin olmadığıdır. Örneğin; Log4J [17] için açık biçimde hata ve kötü koku arasında varoluşsal bir ilişki mevcut iken, Synapse [23] için tam tersi bir durum görülmektedir. Kötü kokuların, projenin tüm sınıflarına yayılmak yerine, belli sınıflarında yoğunlaştığı da Çizelge 4.1 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 korumaya devam ederler. İstatistiksel dağılımda Tomcat [21] 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.



Şekil 4.1. Projelerdeki Hata - Kötü Koku Sayı Eğrisi

Projelerde yer alan kötü koku sayısının fazlalığıyla hata sayısı arasında da doğrusal bir ilişki olmadığı Şekil 4.1 üzerinden anlaşılmaktadır.

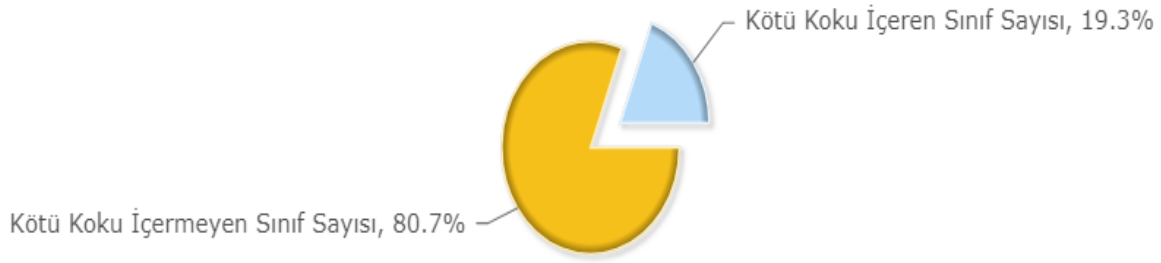
Çizelge 4.1 üzerinden projeleri ayrı ayrı incelemek gerekirse Log4J için toplam hata sayısı **498,** toplam kötü koku sayısı ise **62**'dir. Log4J toplam **223** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **189,** bilinen bir hatası olmayan sınıfların sayısının **34** olduğu gözlemlenmektedir.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%84.8**'lik bir bölümünde hataya rastlandığı, **%15.2**'lik kısmının ise hatasız olduğu gözlemlenmektedir. Log4J için sınıfların hata durumu Şekil 4.2'de verilmiştir.

Şekil 4.2. Log4J Sınıfların Hata Durumu

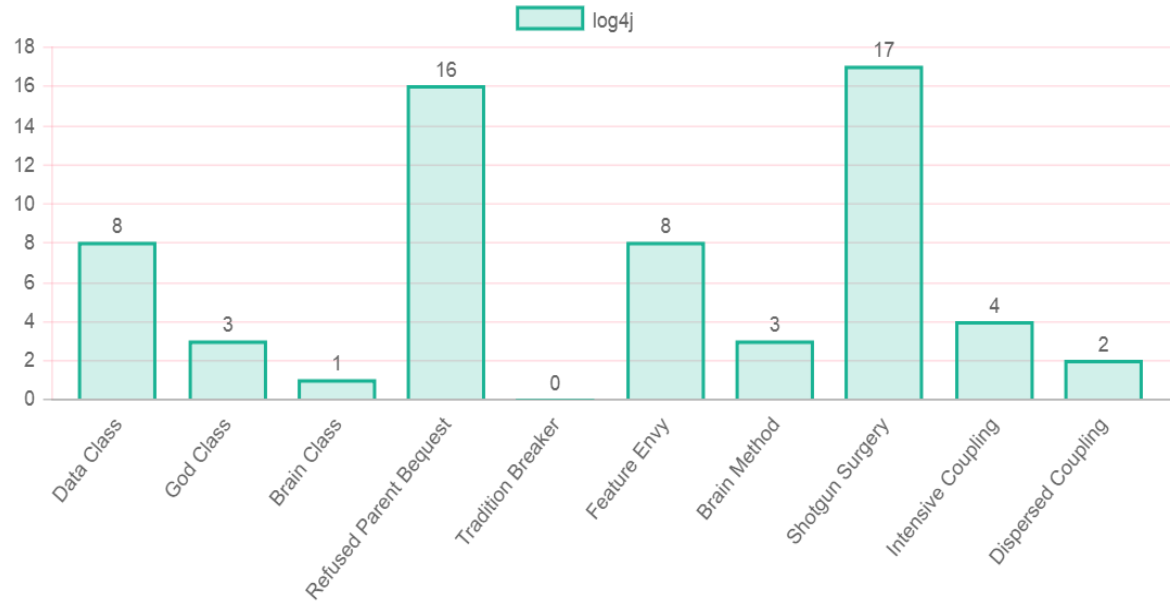
Eclipse eklentimiz üzerinden Log4J için kötü koku tespiti yaptığımızda **43** adet kötü koku içeren sınıf, **180** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%19.3**'lük bir bölümünde kötü kokuya rastlandığı, **%80.7**'lik kısmının ise kötü koku içermediği gözlemlenmektedir. Log4J için sınıfların kötü koku durumu Şekil 4.3'te verilmiştir.



Şekil 4.3. Log4J Sınıfların Kötü Koku Durumu

Log4J'nin kaynak kodlarında en çok rastlanan kötü kokular Shotgun Surgery (**17**) ve Refused Parent Bequest (**16**) olmakla birlikte, en az rastlanan kötü kokular Tradition Breaker (**0**) ve Brain Class (**1**) olarak gözlemlenmektedir. Log4J için detaylı kötü koku dağılımı Şekil 4.4'te verilmektedir.



Şekil 4.4. Log4J Kötü Koku Dağılımı

PBeans için toplam hata sayısı **36,** toplam kötü koku sayısı ise **8**'dir. PBeans toplam **30** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **20,** bilinen bir hatası olmayan sınıfların sayısının **10** olduğu gözlemlenmektedir.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%66.7**'lik bir bölümünde hataya rastlandığı, **%33.3**'lük kısmının ise hatasız olduğu gözlemlenmektedir. PBeans için sınıfların hata durumu Şekil 4.5'te verilmiştir.

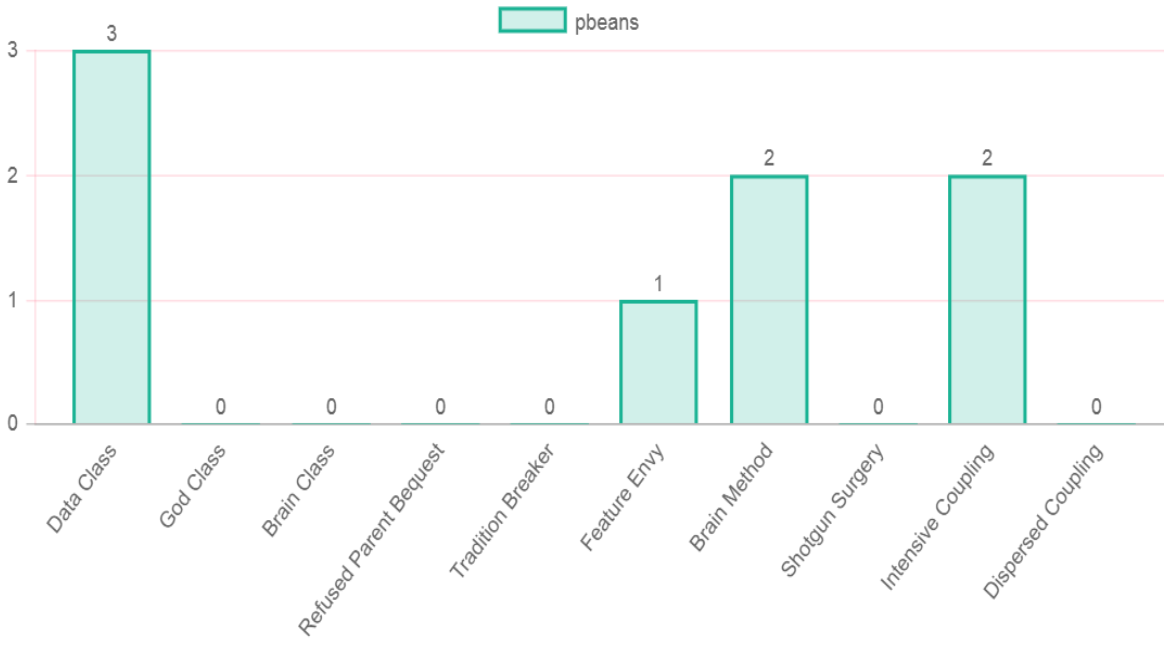
Şekil 4.5. PBeans Sınıfların Hata Durumu

Eclipse eklentimiz üzerinden PBeans için kötü koku tespiti yaptığımızda **6** adet kötü koku içeren sınıf, **24** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%20**'lik bir bölümünde kötü kokuya rastlandığı, **%80**'lik kısmının ise kötü koku içermediği gözlemlenmektedir. PBeans için sınıfların kötü koku durumu Şekil 4.6'da verilmiştir.

Şekil 4.6. PBeans Sınıfların Kötü Koku Durumu

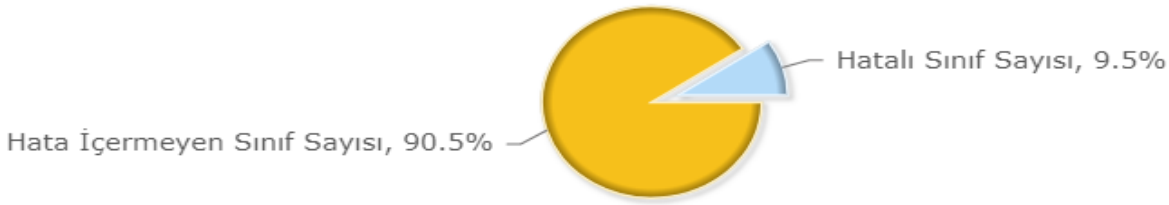
PBeans'nin kaynak kodlarında en çok rastlanan kötü koku Data Class (**3**) olmakla birlikte, kaynak kod içerisinde Tradition Breaker, God Class, Brain Class, Refused Parent Bequest, Shotgun Surgery ve Dispersed Coupling kötü kokularına hiç rastlanmamıştır. PBeans için detaylı kötü koku dağılımı Şekil 4.7'de verilmektedir.



Şekil 4.7. PBeans Kötü Koku Dağılımı

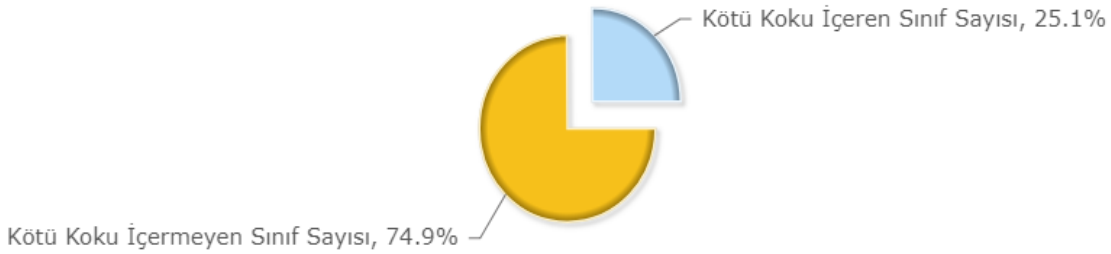
Ivy için toplam hata sayısı **56,** toplam kötü koku sayısı ise **239**'dur. Ivy toplam **422** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **40,** bilinen bir hatası olmayan sınıfların sayısının **382** olduğu belirtilmiştir.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%9.5**'lik bir bölümünde hataya rastlandığı, **%90.5**'lik kısmının ise hatasız olduğu gözlemlenmektedir. Ivy için sınıfların hata durumu Şekil 4.8'de verilmiştir.



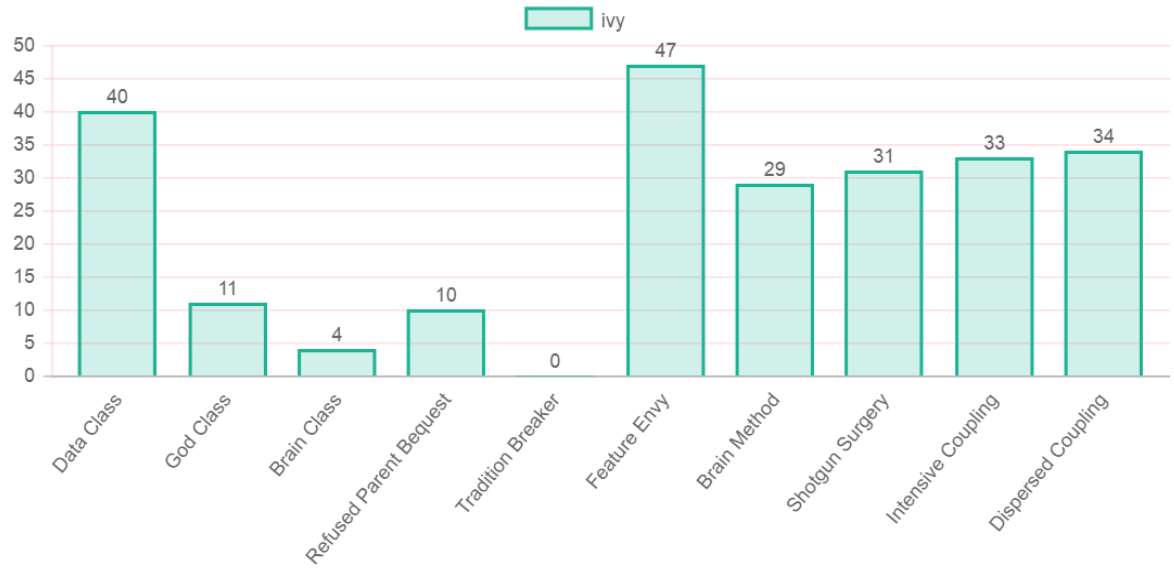
Şekil 4.8. Ivy Sınıfların Hata Durumu

Eclipse eklentimiz üzerinden Ivy için kötü koku tespiti yaptığımızda **106** adet kötü koku içeren sınıf, **316** adet kötü koku içermeyen sınıf ile karşılaşılmıştır. Yüzdelik olarak ifade etmek gerekirse sınıfların **%25.1**'lik bir bölümünde kötü kokuya rastlandığı, **%74.9**'luk kısmının ise kötü koku içermediği gözlemlenmektedir. Ivy için sınıfların kötü koku durumu Şekil 4.9'da verilmiştir.



Şekil 4.9. Ivy Sınıfların Kötü Koku Durumu

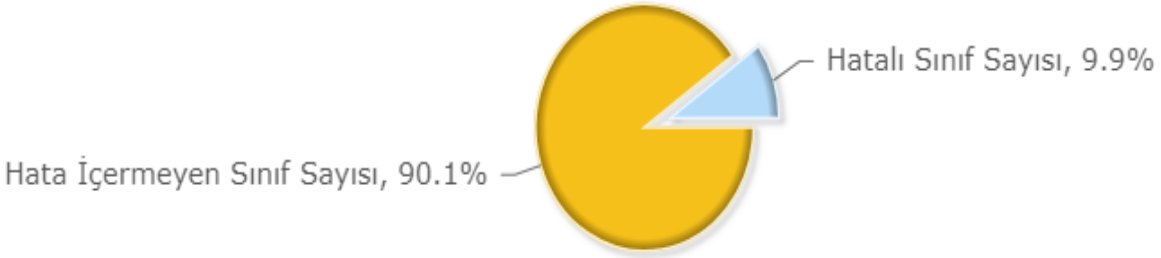
Ivy'nin kaynak kodlarında en çok rastlanan kötü koku Feature Envy (**47**) olmakla birlikte, en az rastlanan kötü kokular Tradition Breaker (0) ve Brain Class (**4**) olarak gözlemlenmektedir. Ivy için detaylı kötü koku dağılımı Şekil 4.10'da verilmektedir.



Şekil 4.10. Ivy Kötü Koku Dağılımı

Synapse için toplam hata sayısı **21,** toplam kötü koku sayısı ise **131**'dir. Synapse toplam **161** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **16,** bilinen bir hatası olmayan sınıfların sayısının **145** olduğu gözlemlenmiştir.

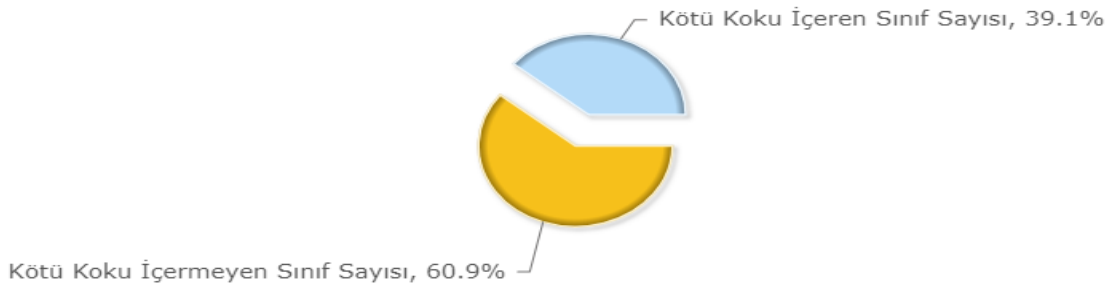
Yüzdelik olarak ifade etmek gerekirse sınıfların **%9.9**'luk bir bölümünde hataya rastlandığı, **%90.1**'lik kısmının ise hatasız olduğu gözlemlenmektedir. Synapse için sınıfların hata durumu Şekil 4.11'de verilmiştir.



Şekil 4.11. Synapse Sınıfların Hata Durumu

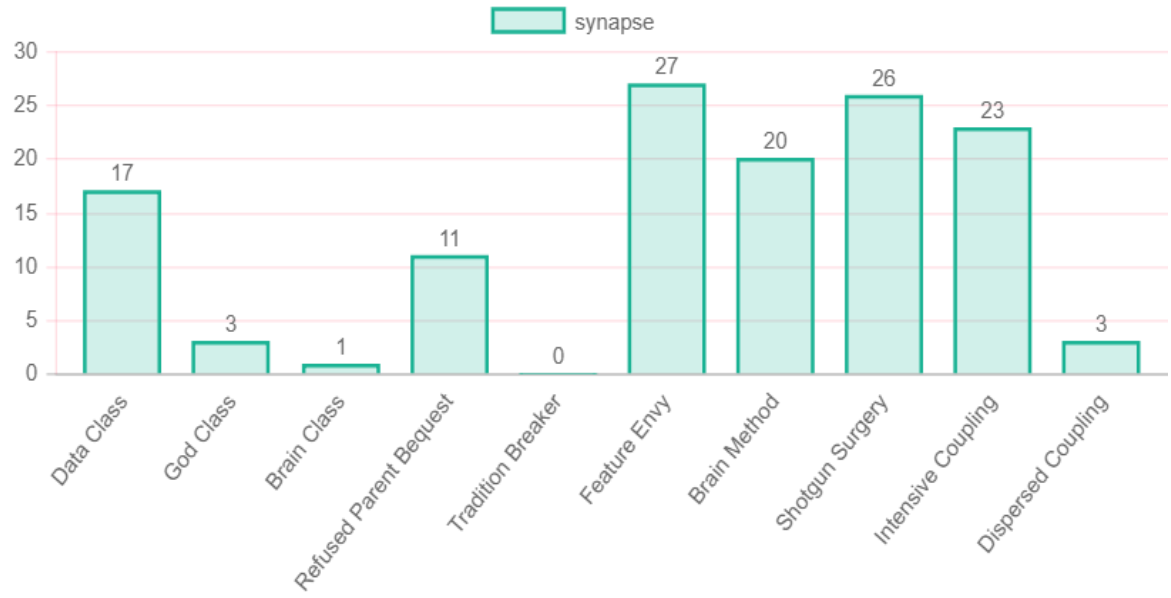
Eclipse eklentimiz üzerinden Synapse için kötü koku tespiti yaptığımızda **63** adet kötü koku içeren sınıf, **98** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%39.1**'lik bir bölümünde kötü kokuya rastlandığı, **%60.9**'luk kısmının ise kötü koku içermediği gözlemlenmektedir. Synapse için sınıfların kötü koku durumu Şekil 4.12'de verilmiştir.



Şekil 4.12. Synapse Sınıfların Kötü Koku Durumu

Synapse'nin kaynak kodlarında en çok rastlanan kötü kokular Feature Envy (**27**) ve Shotgun Surgery (**26**) olmakla birlikte, en az rastlanan kötü kokular Tradition Breaker (**0**) ve Brain Class (**1**) olarak gözlemlenmektedir. Synapse için detaylı kötü koku dağılımı Şekil 4.13'te verilmektedir.



Şekil 4.13. Synapse Kötü Koku Dağılımı

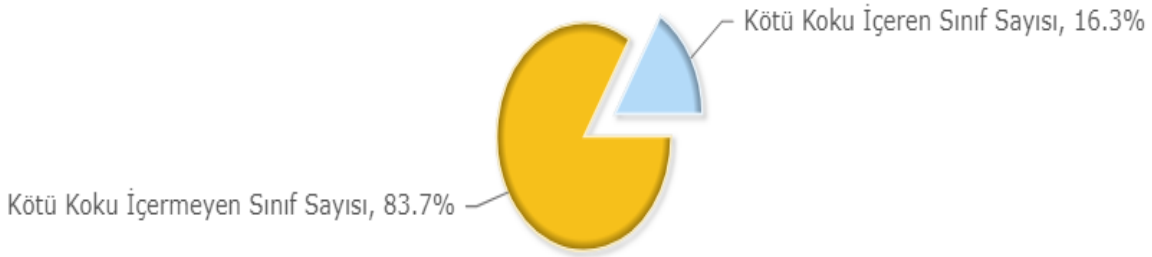
Velocity için toplam hata sayısı **210,** toplam kötü koku sayısı ise **72**'dir. Velocity toplam **221** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **147,** bilinen bir hatası olmayan sınıfların sayısının **74** olduğu belirtilmiştir.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%66.5**'lik bir bölümünde hataya rastlandığı, **%33.5**'lik kısmının ise hatasız olduğu gözlemlenmektedir. Velocity için sınıfların hata durumu Şekil 4.14'te verilmiştir.

Şekil 4.14. Velocity Sınıfların Hata Durumu

Eclipse eklentimiz üzerinden Velocity için kötü koku tespiti yaptığımızda **36** adet kötü koku içeren sınıf, **185** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

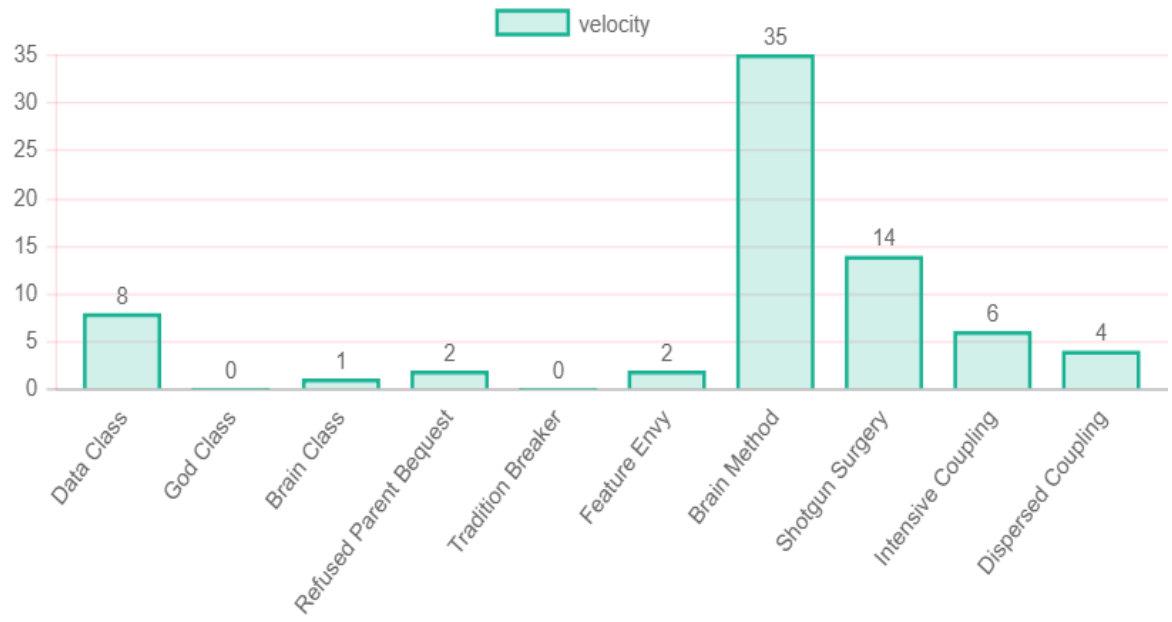
Yüzdelik olarak ifade etmek gerekirse sınıfların **%16.3**'lük bir bölümünde kötü kokuya rastlandığı, **%83.7**'lik kısmının ise kötü koku içermediği gözlemlenmektedir. Velocity için sınıfların kötü koku durumu Şekil 4.15'te verilmiştir.



Şekil 4.15.Velocity Sınıfların Kötü Koku Durumu

Velocity'nin kaynak kodlarında en çok rastlanan kötü koku Brain Method (**35**) olmakla birlikte, Tradition Breaker (**0**) ve God Class (**0**) kötü kokularına hiç

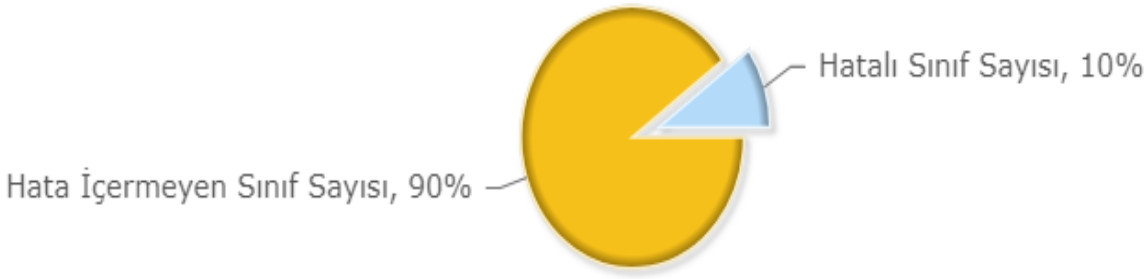
rastlanmadığı olarak gözlemlenmektedir. Velocity için detaylı kötü koku dağılımı Şekil 4.16'da verilmektedir.



Şekil 4.16. Velocity Kötü Koku Dağılımı

Poi için toplam hata sayısı **39,** toplam kötü koku sayısı ise **68**'dir. Poi toplam **370** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **37,** bilinen bir hatası olmayan sınıfların sayısının **333** olduğu gözlemlenmiştir.

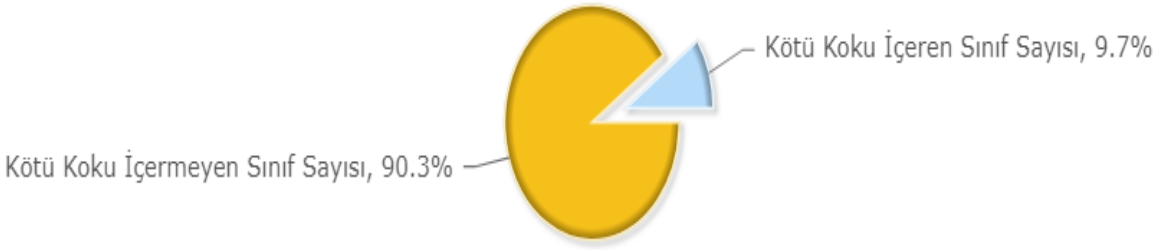
Yüzdelik olarak ifade etmek gerekirse sınıfların **%10**'luk bir bölümünde hataya rastlandığı, **%90**'lık kısmının ise hatasız olduğu gözlemlenmektedir. Poi için sınıfların hata durumu Şekil 4.17'de verilmiştir.



Şekil 4.17. Poi Sınıfların Hata Durumu

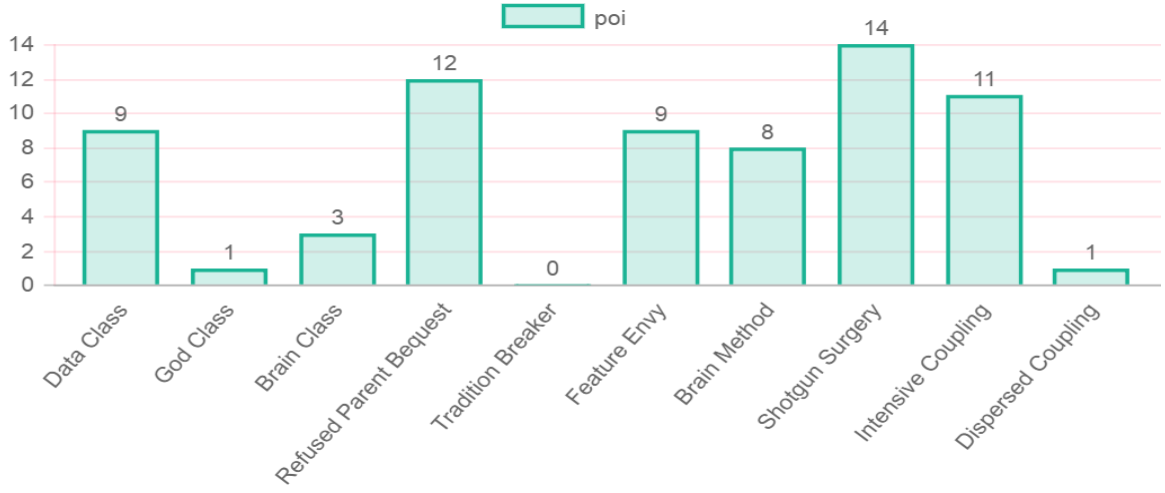
Eclipse eklentimiz üzerinden Poi için kötü koku tespiti yaptığımızda **36** adet kötü koku içeren sınıf, **334** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%9.7**'lik bir bölümünde kötü kokuya rastlandığı, **%90.3**'lük kısmının ise kötü koku içermediği gözlemlenmektedir. Poi için sınıfların kötü koku durumu Şekil 4.18'de verilmiştir.



Şekil 4.18. Poi Sınıfların Kötü Koku Durumu

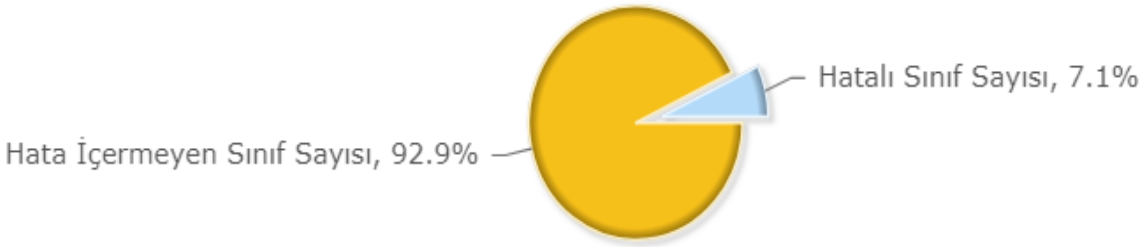
Poi'nin kaynak kodlarında en çok rastlanan kötü kokular Shotgun Surgery (**14**) ve Refused Parent Bequest (**12**) olmakla birlikte, Tradition Breaker (**0**) ve God Class (**0**) kötü kokularına hiç rastlanılmazken, en az rastlanan kötü koku türü ise Dispersed Coupling (**1**) olarak gözlemlenmektedir. Poi için detaylı kötü koku dağılımı Şekil 4.19'da verilmektedir.



Şekil 4.19. Poi Kötü Koku Dağılımı

Tomcat için toplam hata sayısı **114,** toplam kötü koku sayısı ise **543**'tür. Tomcat toplam **1086** sınıftan oluşmaktadır. Hata raporlarında hata içeren sınıf sayısının **77,** bilinen bir hatası olmayan sınıfların sayısının **1009** olduğu belirtilmiştir.

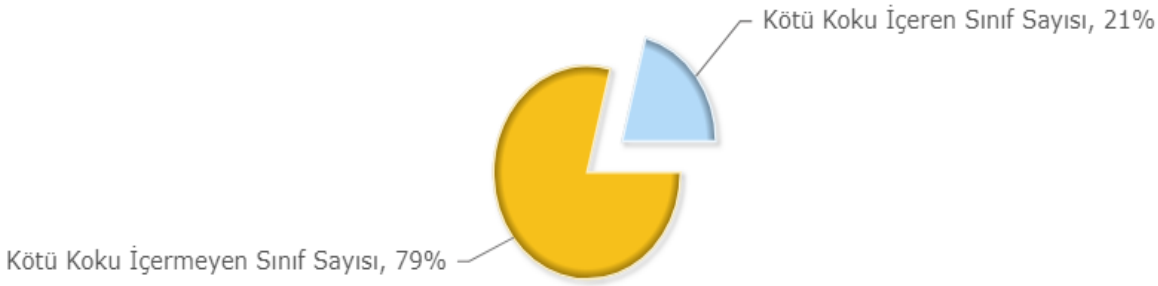
Yüzdelik olarak ifade etmek gerekirse sınıfların **%7.1**'lik bir bölümünde hataya rastlandığı, **%92.9**'luk kısmının ise hatasız olduğu gözlemlenmektedir. Tomcat için sınıfların hata durumu Şekil 4.20'de verilmiştir.



Şekil 4.20. Tomcat Sınıfların Hata Durumu

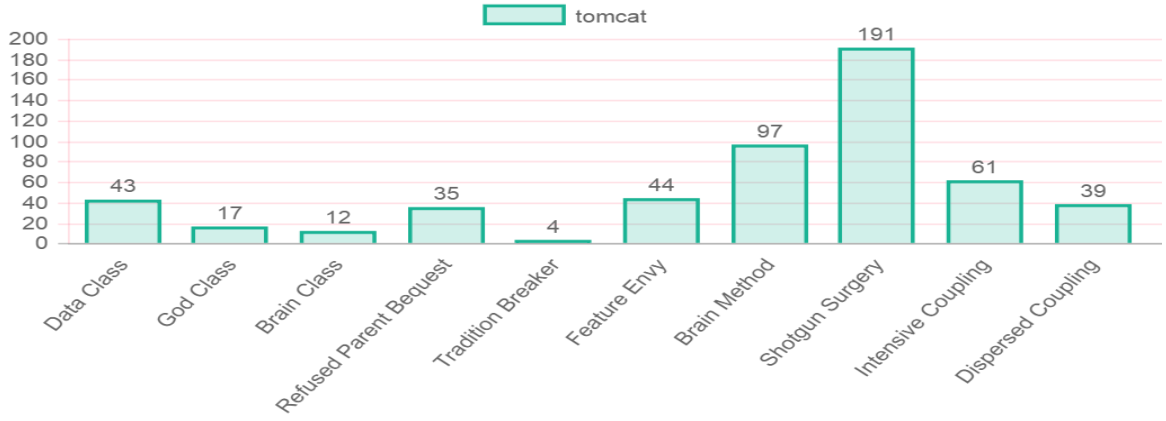
Eclipse eklentimiz üzerinden Tomcat için kötü koku tespiti yaptığımızda **228** adet kötü koku içeren sınıf, **858** adet kötü koku içermeyen sınıf ile karşılaşılmıştır.

Yüzdelik olarak ifade etmek gerekirse sınıfların **%21**'lik bir bölümünde kötü kokuya rastlandığı, **%79**'luk kısmının ise kötü koku içermediği gözlemlenmektedir. Tomcat için sınıfların kötü koku durumu Şekil 4.21'de verilmiştir.



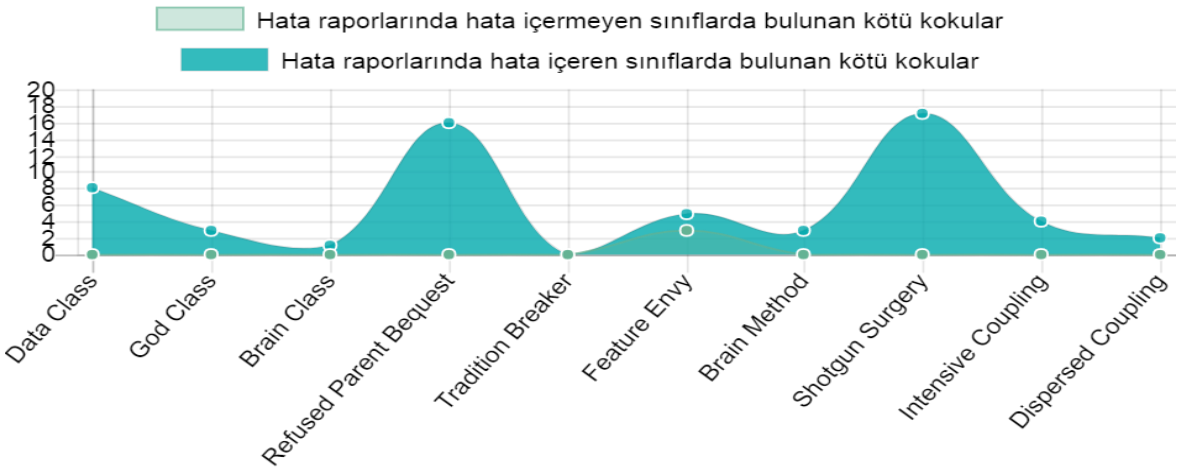
Şekil 4.21. Tomcat Sınıfların Kötü Koku Durumu

Tomcat'in kaynak kodlarında en çok rastlanan kötü koku Shotgun Surgery (**191**) olmakla birlikte, en az rastlanan kötü koku Tradition Breaker (**4**) olarak gözlemlenmektedir. Tomcat için detaylı kötü koku dağılımı Şekil 4.22'de verilmektedir.



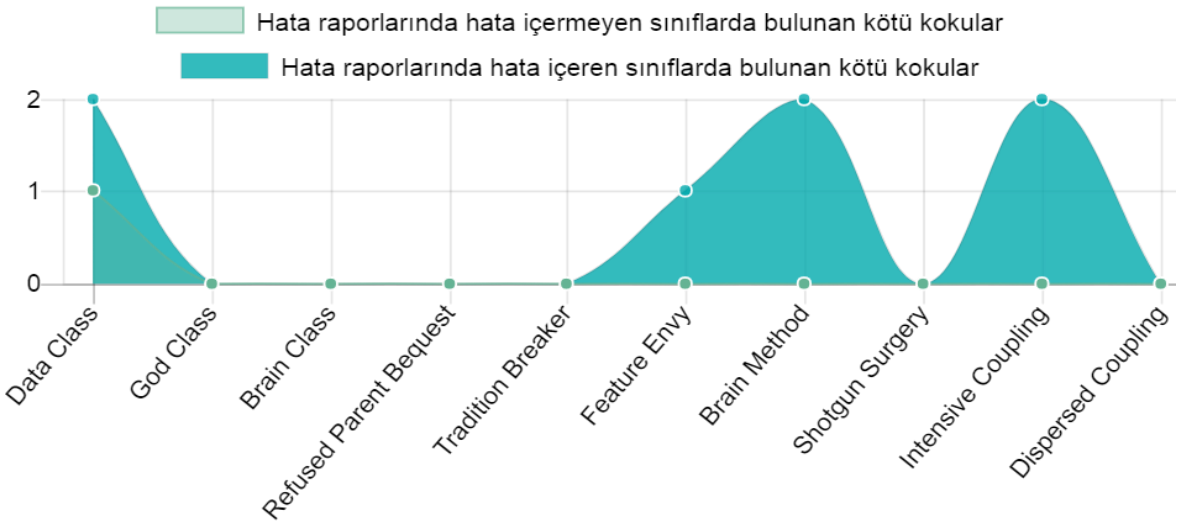
Şekil 4.22. Tomcat Kötü Koku Dağılımı

Bizim bu tezdeki en büyük amaçlarımızdan biri kötü kokuların hata oluşumuna etkisini inceleyebilmektir. Bu nedenle versiyon bazlı hata raporları bulunan 7 adet açık kaynak proje seçilmiştir. İlk olarak teste Log4J ile başlanmıştır. Hata raporlarında hata içermeyen sınıflarda toplam **3** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **59** adet kötü koku bulunmuştur [34]. Log4J için kötü koku hata ilişkisi grafiği Şekil 4.23'te verilmektedir.



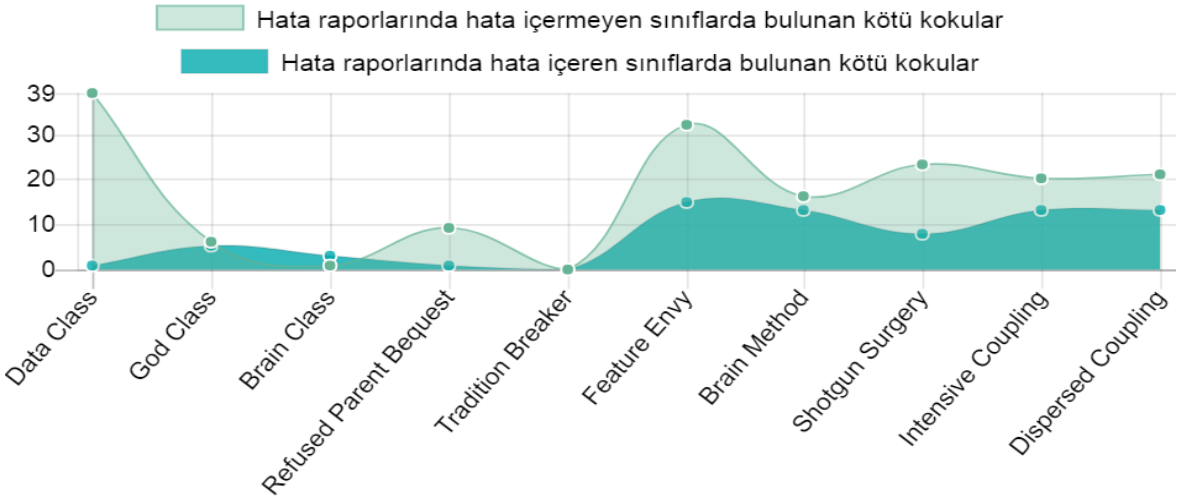
Şekil 4.23. Log4J Kötü Koku Hata İlişkisi

PBeans için ise hata raporlarında hata içermeyen sınıflarda toplam **1** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **7** adet kötü koku bulunmuştur [35]. PBeans için kötü koku hata ilişkisi grafiği Şekil 4.24'te verilmektedir.



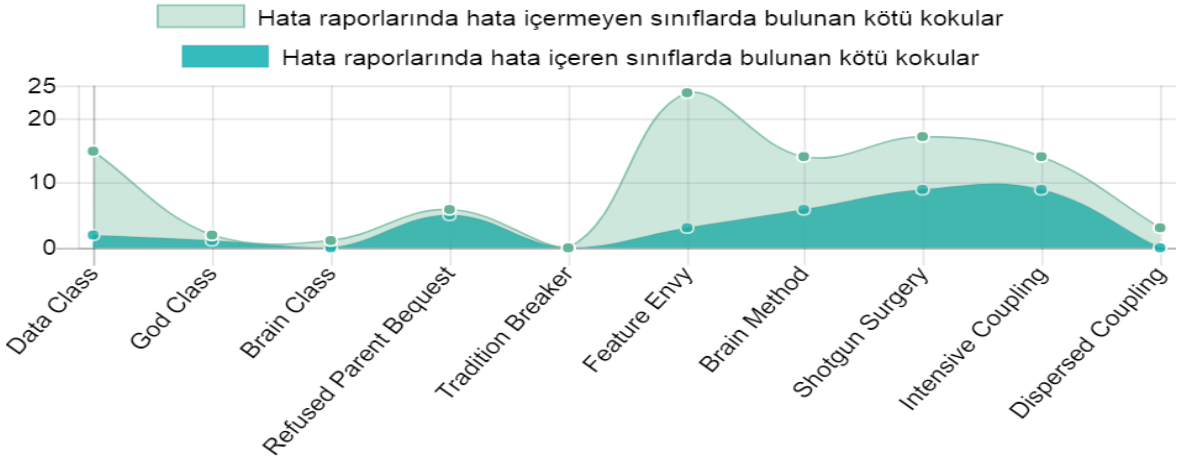
Şekil 4.24. PBeans Kötü Koku Hata İlişkisi

Ivy için ise hata raporlarında hata içermeyen sınıflarda toplam **167** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **72** adet kötü koku bulunmuştur [36]. Ivy için kötü koku hata ilişkisi grafiği Şekil 4.25'te verilmektedir.



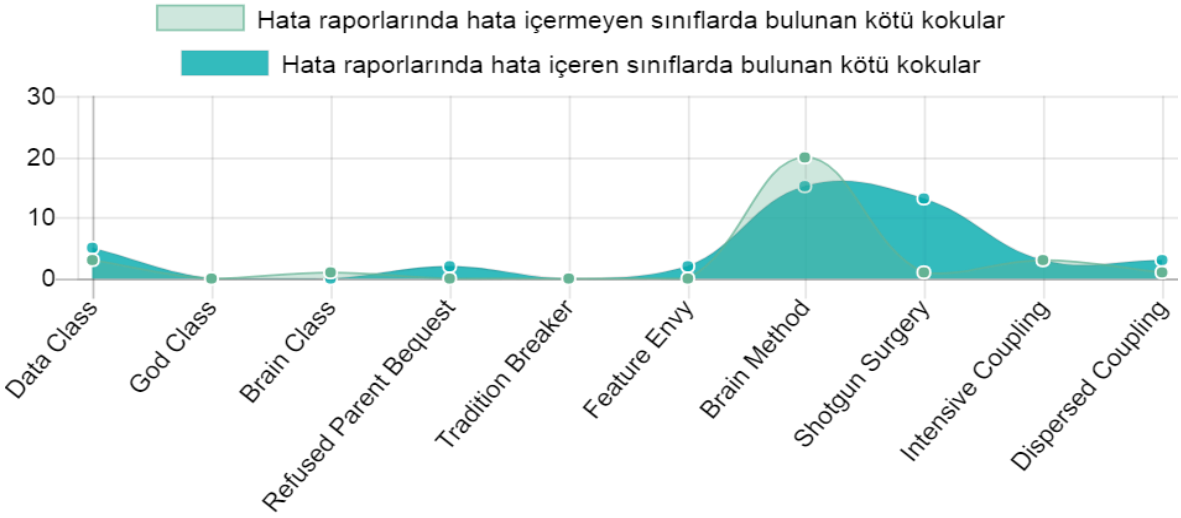
Şekil 4.25. Ivy Kötü Koku Hata İlşkisi

Synapse için ise hata raporlarında hata içermeyen sınıflarda toplam **96** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **35** adet kötü koku bulunmuştur [37]. Synapse için kötü koku hata ilişkisi grafiği Şekil 4.26'da verilmektedir.



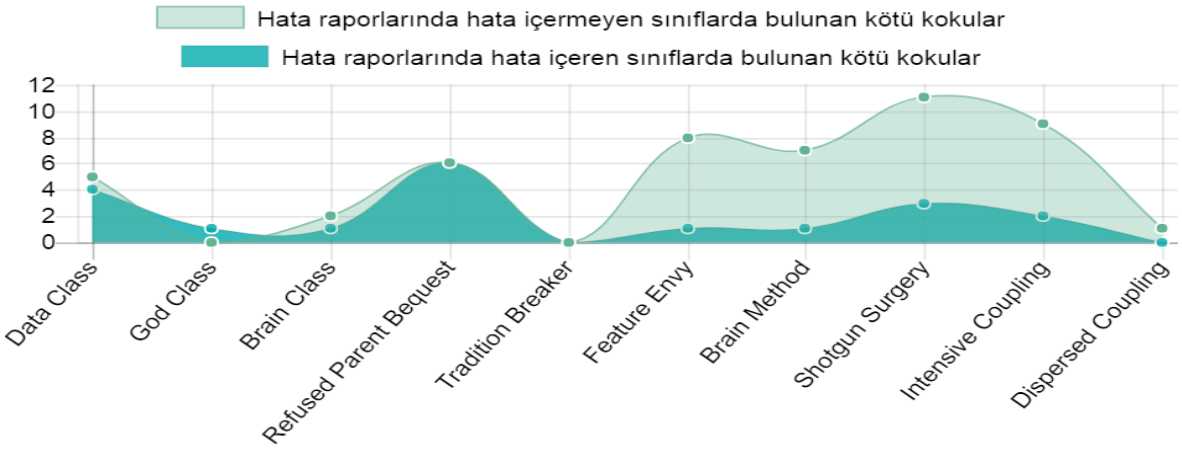
Şekil 4.26. Synapse Kötü Koku Hata İlişkisi

Velocity için ise hata raporlarında hata içermeyen sınıflarda toplam **29** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **43** adet kötü koku bulunmuştur. [38] Velocity için kötü koku hata ilişkisi grafiği Şekil 4.27'de verilmektedir.



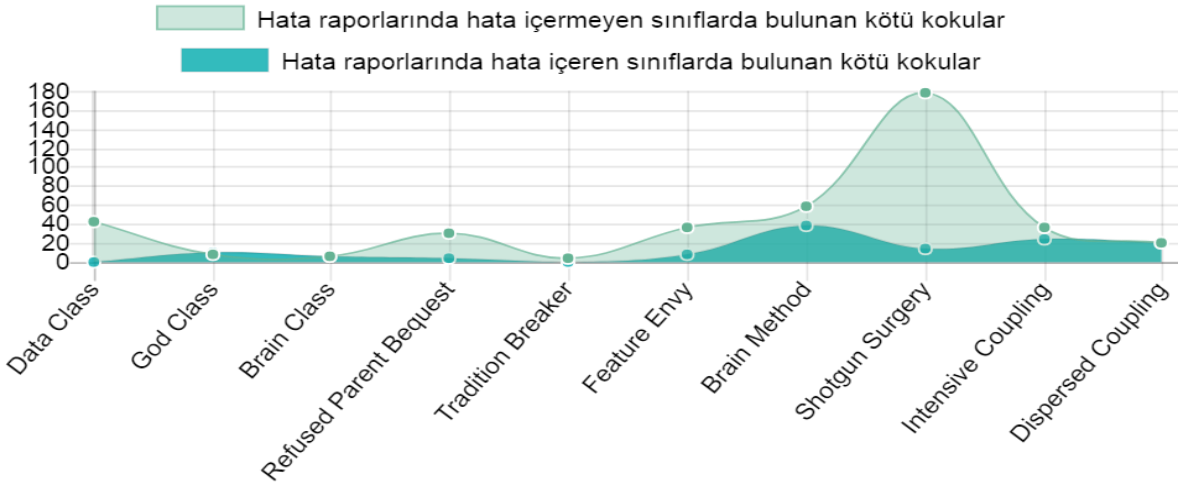
Şekil 4.27. Velocity Kötü Koku Hata İlişkisi

Poi için ise hata raporlarında hata içermeyen sınıflarda toplam **49** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **19** adet kötü koku bulunmuştur [39]. Poi için kötü koku hata ilişkisi grafiği Şekil 4.28'de verilmektedir.



Şekil 4.28. Poi Kötü Koku Hata İlişkisi

Tomcat için ise hata raporlarında hata içermeyen sınıflarda toplam **421** kötü koku, hata raporlarında hata bulunan sınıflarda ise toplam **122** adet kötü koku bulunmuştur [40]. Tomcat için kötü koku hata ilişkisi grafiği Şekil 4.29'da verilmektedir.



Şekil 4.29. Tomcat Kötü Koku Hata İlişkisi

7 proje için kötü kokuların hatalı ve hatasız sınıflarda görülme istatistikleri Çizelge 4.2'de verilmiştir.

Çizelge 4.2. Kötü Kokuların Hatalı-Hatasız Sınıflarda Görülme İstatistikleri

Kötü Koku	Hatasız Sınıfta Gözlenme	Hatalı Sınıfta Gözlenme
Data Class	106	22
God Class	15	20
Brain Class	11	11
Refused Parent Bequest	52	34
Tradition Breaker	4	0
Feature Envy	104	34
Brain Method	116	78
Shotgun Surgery	229	64
Intensive Coupling	86	57
Dispersed Coupling	46	37

Çizelgeden çıkarılabilecek sonuç; eğer kötü kokular ile hata arasında ileride yapılacak çalışmalarda bir ilişki kesinleşirse, Data Class, Shotgun Surgery ve Feature Envy'nin hataya en az etki edecek kötü kokular olacağı gözlenmektedir.

5. EKLENTİ SONUÇLARININ IPLASMA ILE KARŞILAŞTIRILMASI

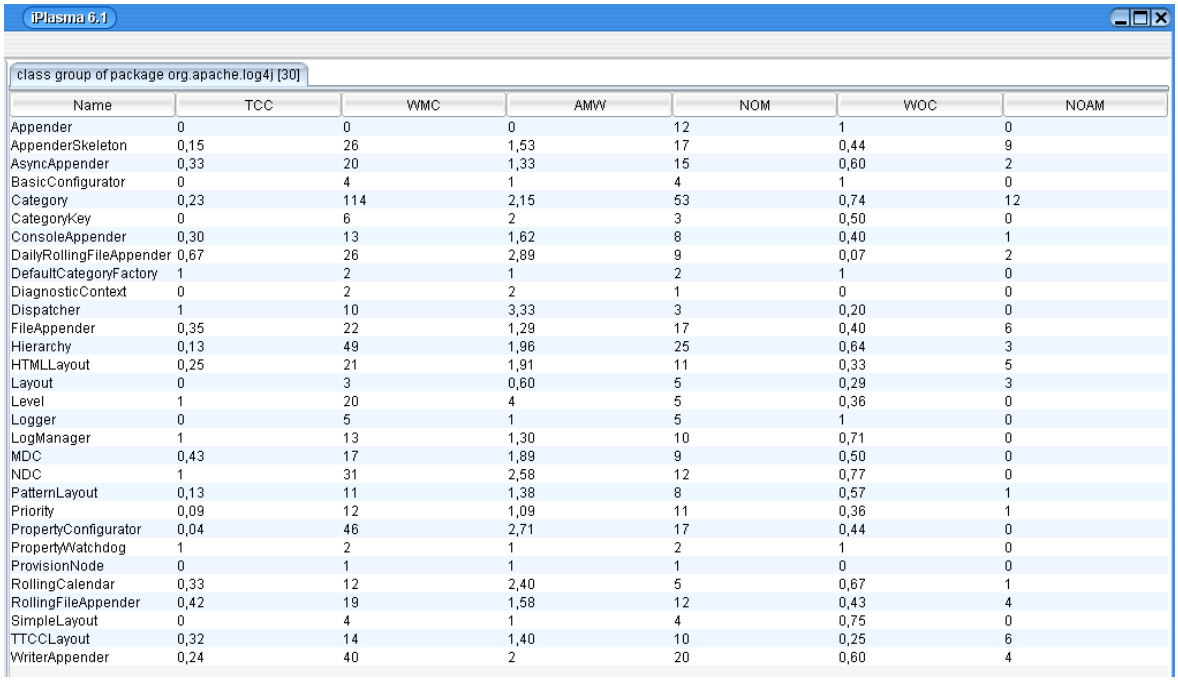
Eklentimizin hesapladığı tüm metriklerin doğruluğu birim testlerle doğrulanmış ve kötü koku tespiti de bu metriklerin bir formülasyonu olduğundan sonuçlarımızın doğruluğu ispatlanmıştır. Ancak bu ispatın yanında, eklentimizin bulduğu kötü kokuların doğruluğunun ispatı için, bu alanda geliştirilmiş diğer programlarla da sonuçlarının karşılaştırılması gerekmektedir. Kötü koku tespiti yapabilen araçlardan bizim kadar kötü koku tespiti yapabilen en iyi çalışma IPLASMA aracıdır. Bu nedenle sonuçlarımızın IPLASMA aracı ile karşılaştırılmasına karar verilmiştir.

Bizim eklentimiz, bir proje üzerindeki tespit edilen tüm kötü kokuların ve metriklerin sınıf bazlı detaylı sonuçlarını verebilmektedir, ancak IPLASMA metrik ve kötü koku raporlarını bir çıktı üzerinden verememektedir. IPLASMA üzerinde rapor üretebilme yeteneği olmadığından kötü koku gruplarını, kısaca tüm sınıfları birer birer inceleme zorunluluğu vardır. Tüm sınıfların birer birer incelenecek olması çok zaman alacak bir iş olduğundan, analiz edilen 7 projeden 186 sınıf sayısıyla en az sınıf sayısına sahip olan Log4J projesi üzerinden iki programın çıktılarının karşılaştırılmasına karar verilmiştir. [41]

5.1. Metrik Değerlerinin Karşılaştırılması

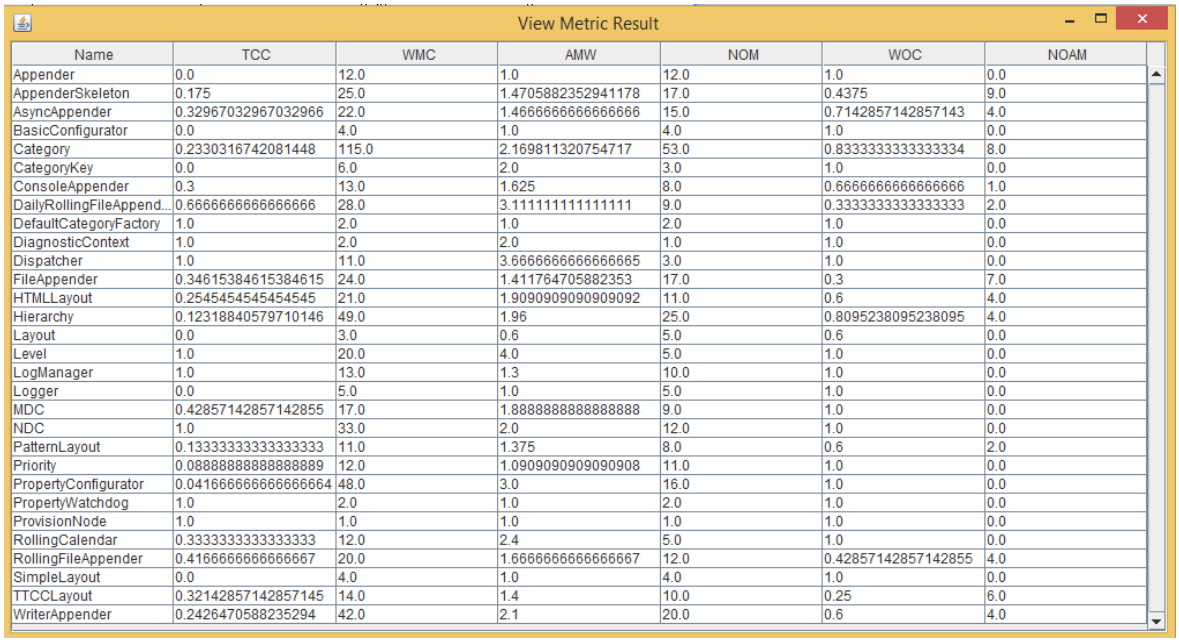
IPLASMA bir paket içerinde yer alan sınıflar için sınıf metriklerini raporlayabilmektedir, ancak bu destek metotlar için yoktur. *IPLASMA* üzerinden Log4J projesinin **org.apache.log4j** paketinde yer alan 30 adet sınıf için *TCC, WMC, AMW, NOM, WOC ve NOAM* sınıf metriklerinin değeri Şekil 5.1'de verilmiştir.

Aynı pakette yer alan sınıflar için aynı metrikler bizim eklentimiz üzerinden de hesaplanmış sonuçlar Şekil 5.2'de verilmiştir.



Name	TCC	WMC	AMW	NOM	WOC	NOAM
Appender	0	0	0	12	1	0
AppenderSkeleton	0,15	26	1,53	17	0,44	9
AsyncAppender	0,33	20	1,33	15	0,60	2
BasicConfigurator	0	4	1	4	1	0
Category	0,23	114	2,15	53	0,74	12
CategoryKey	0	6	2	3	0,50	0
ConsoleAppender	0,30	13	1,62	8	0,40	1
DailyRollingFileAppender	0,67	26	2,89	9	0,07	2
DefaultCategoryFactory	1	2	1	2	1	0
DiagnosticContext	0	2	2	1	0	0
Dispatcher	1	10	3,33	3	0,20	0
FileAppender	0,35	22	1,29	17	0,40	6
Hierarchy	0,13	49	1,96	25	0,64	3
HTMLLayout	0,25	21	1,91	11	0,33	5
Layout	0	3	0,60	5	0,29	3
Level	1	20	4	5	0,36	0
Logger	0	5	1	5	1	0
LogManager	1	13	1,30	10	0,71	0
MDC	0,43	17	1,89	9	0,50	0
NDC	1	31	2,58	12	0,77	0
PatternLayout	0,13	11	1,38	8	0,57	1
Priority	0,09	12	1,09	11	0,36	1
PropertyConfigurator	0,04	46	2,71	17	0,44	0
PropertyWatchdog	1	2	1	2	1	0
ProvisionNode	0	1	1	1	0	0
RollingCalendar	0,33	12	2,40	5	0,67	1
RollingFileAppender	0,42	19	1,58	12	0,43	4
SimpleLayout	0	4	1	4	0,75	0
TTCCLayout	0,32	14	1,40	10	0,25	6
WriterAppender	0,24	40	2	20	0,60	4

Şekil 5.1. IPLASMA'nın Log4J İçin Sınıf Metrik Sonuçları



Name	TCC	WMC	AMW	NOM	WOC	NOAM
Appender	0.0	12.0	1.0	12.0	1.0	0.0
AppenderSkeleton	0.175	25.0	1.4705882352941178	17.0	0.4375	9.0
AsyncAppender	0.32967032967032966	22.0	1.4666666666666666	15.0	0.7142857142857143	4.0
BasicConfigurator	0.0	4.0	1.0	4.0	1.0	0.0
Category	0.2330316742081448	115.0	2.169811320754717	53.0	0.8333333333333334	8.0
CategoryKey	0.0	6.0	2.0	3.0	1.0	0.0
ConsoleAppender	0.3	13.0	1.625	8.0	0.6666666666666666	1.0
DailyRollingFileAppend...	0.6666666666666666	28.0	3.111111111111111	9.0	0.3333333333333333	2.0
DefaultCategoryFactory	1.0	2.0	1.0	2.0	1.0	0.0
DiagnosticContext	1.0	2.0	2.0	1.0	1.0	0.0
Dispatcher	1.0	11.0	3.6666666666666665	3.0	1.0	0.0
FileAppender	0.34615384615384615	24.0	1.411764705882353	17.0	0.3	7.0
HTMLLayout	0.2545454545454545	21.0	1.9090909090909092	11.0	0.6	4.0
Hierarchy	0.12318840579710146	49.0	1.96	25.0	0.8095238095238095	4.0
Layout	0.0	3.0	0.6	5.0	0.6	0.0
Level	1.0	20.0	4.0	5.0	1.0	0.0
LogManager	1.0	13.0	1.3	10.0	1.0	0.0
Logger	0.0	5.0	1.0	5.0	1.0	0.0
MDC	0.42857142857142855	17.0	1.8888888888888888	9.0	1.0	0.0
NDC	1.0	33.0	2.0	12.0	1.0	0.0
PatternLayout	0.13333333333333333	11.0	1.375	8.0	0.6	2.0
Priority	0.08888888888888889	12.0	1.0909090909090908	11.0	1.0	0.0
PropertyConfigurator	0.041666666666666664	48.0	3.0	16.0	1.0	0.0
PropertyWatchdog	1.0	2.0	1.0	2.0	1.0	0.0
ProvisionNode	1.0	1.0	1.0	1.0	1.0	0.0
RollingCalendar	0.3333333333333333	12.0	2.4	5.0	1.0	0.0
RollingFileAppender	0.4166666666666667	20.0	1.6666666666666667	12.0	0.42857142857142855	4.0
SimpleLayout	0.0	4.0	1.0	4.0	1.0	0.0
TTCCLayout	0.32142857142857145	14.0	1.4	10.0	0.25	6.0
WriterAppender	0.2426470588235294	42.0	2.1	20.0	0.6	4.0

Şekil 5.2. Eklentimizin Log4J İçin Sınıf Metrik Sonuçları

TCC metriği için genel sonuçlar çok yakındır. IPLASMA aracı sayının noktadan sonraki bölümünü 2 sayı olacak şekilde yuvarlamayı tercih etmiştir.

WMC metriği için bazı farklılıklar gözlemlenmektedir. WMC metriği metotların CYCLO değerlerinin toplamı hesaplanarak bulunmaktadır. Biz eklentimizde CYCLO değerini içinde hiçbir şey bulundurmayan bir metot için 1'den başlatarak metot içinde gördüğümüz her bir *if, while, for, catch, do, switch, case, koşullu ifade (a ? b : c), &&,* || ifadeleri için bir artırıyoruz. IPLASMA ise *koşullu ifade, &&* ve || işlemleri için artırım yapmamaktadır. Bu nedenle bazı sınıflar için bizim bulduğumuz değerler daha fazladır.

NOM metrik değeri ProperyConfigurator sınıfı hariç birebir aynıdır. Bu sınıftaki farklılığın nedeni araştırıldığında, IPLASMA'nın hesaplama hatası yaptığı gözlemlenmiştir. ProperyConfigurator.java dosyasının içerisinde ayrı olarak PropertyWatchdog adında bağımsız bir sınıf daha yazılmış, bu sınıfın içerisinde yer alan doOnChange metodu da ProperyConfigurator'a ait bir metot gibi NOM metrik değerine eklenmiştir. IPLASMA'ya yapılan bu hata elektronik posta yoluyla bildirilmiştir. ProperyConfigurator.java dosyasının içeriği çok büyük olduğundan resim olarak eklenmemiştir. İçeriğe Github üzerinden erişebilirsiniz. [44]

AMW metrik değeri WMC metrik değerinin NOM değerine bölünmesiyle bulunan bir metriktir. WMC metriği üzerindeki farklılık nedeniyle bu değer üzerinde de farklılıklar oluşmaktadır.

NOAM metrik değerlerinde birçok farklılık bulunmaktadır. NOAM metrik değeri bir sınıf içerisindeki erişimci (getter-setter) metotlarının sayısıdır. Bir metodun erişimci bir metot olarak değerlendirilmesinin şartı, sınıf içerisinde tanımlanan bir niteliğe ulaşım sağlamasıdır. IPLASMA sistemde yer alan get ve set ön ekiyle başlayan her metot için erişimci metot nitelendirmesi yapmaktadır. Bu durum yanlış bir yaklaşımdır. Bizim eklentimiz ise sadece bir niteliğin değerini değiştirmeye veya o niteliğe ulaşmayı sağlayan metotlara erişimci metot demektedir. Örneğin; IPLASMA, Layout sınıfının NOAM değerini 3 olarak bulmuştur. Ancak get ile başlayan bu metotlar sınıf içerisinde yer alan hiçbir niteliğe ulaşım sağlamamaktadır. Bu nedenle bizim eklentimiz için erişimci metot olarak nitelendirilemezler. Layout sınıfının içeriği Şekil 5.3'te verilmiştir.

```
public abstract class Layout implements OptionHandler {


  public final static String LINE_SEP = System.getProperty("line.separator");
  public final static int LINE_SEP_LEN  = LINE_SEP.length();

  abstract
  public
  String format(LoggingEvent event);

  public
  String getContentType() {
    return "text/plain";
  }

  public
  String getHeader() {
    return null;
  }


  public
  String getFooter() {
    return null;
  }

  abstract
  public
  boolean ignoresThrowable();
```

Şekil 5.3. Log4J Layout Sınıf İçeriği

WOC metiriği için de birçok farklılık vardır. WOC metriği sınıftaki fonksiyonel metotların sayısının, tüm metotların sayısına bölünmesiyle bulunur. Fonksiyonel metotlar yapıcı ve erişimci metotlar dışındaki metotlardır. IPLASMA ile erişimci metot kavramını farklı kullandığımızdan bu değerinde farklı çıkması normaldir.

5.2. Kötü Koku Sonuçlarının Karşılaştırılması

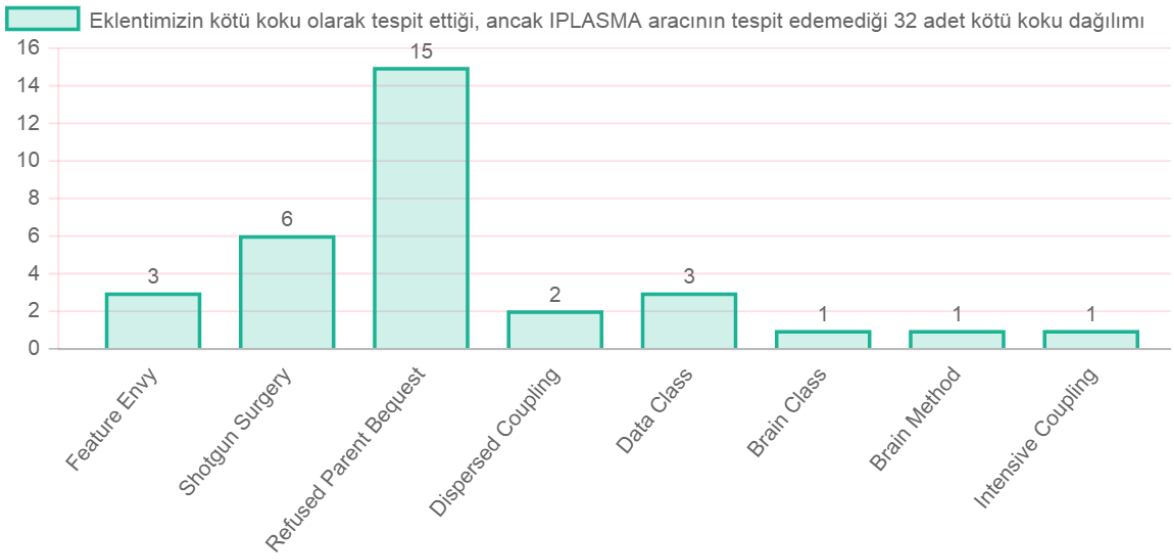
Log4J için eklentimiz ile IPLASMA aracı karşılaştırıldığında iki aracın da ortak olarak tespit ettiği 30 adet kötü koku gözlenmiştir. Bizim eklentimizin kötü koku olarak tespit ettiği ancak IPLASMA aracının tespit edemediği 32 adet kötü koku gözlenmiştir. IPLASMA aracının tespit edebildiği, ancak bizim eklentimizin tespit edemediği 8 adet kötü koku gözlenmiştir. Sonuçlar Şekil 5.4'te verilmiştir.

Şekil 5.4. Eklentimizin Sonuçlarının IPLASMA ile Karşılaştırılması

Ortak olarak tespit edilen 30 kötü kokunun dağılımı Şekil 5.5'te verilmiştir.

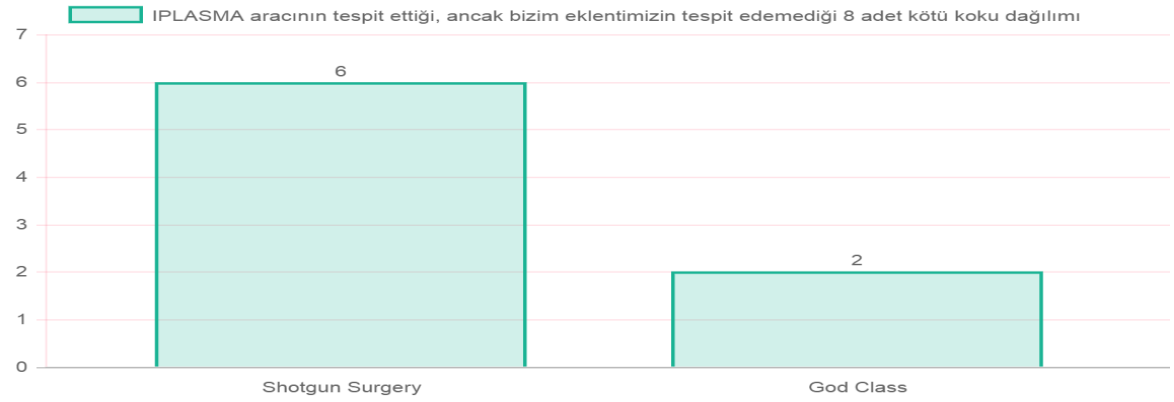
Şekil 5.5. IPLASMA İle Ortak Olarak Tespit Edilen Kötü Kokuların Dağılımı

Bizim eklentimizin tespit ettiği, ancak IPLASMA aracının tespit edemediği 32 adet kötü koku dağılımı Şekil 5.6'da verilmiştir.



Şekil 5.6. Eklentimizin Tespit Ettiği - IPLASMA Aracının Tespit Edemediği Kötü Kokuların Dağılımı

IPLASMA aracının tespit ettiği, ancak bizim eklentimizin tespit edemediği 8 adet kötü koku dağılımı Şekil 5.7'de verilmiştir.

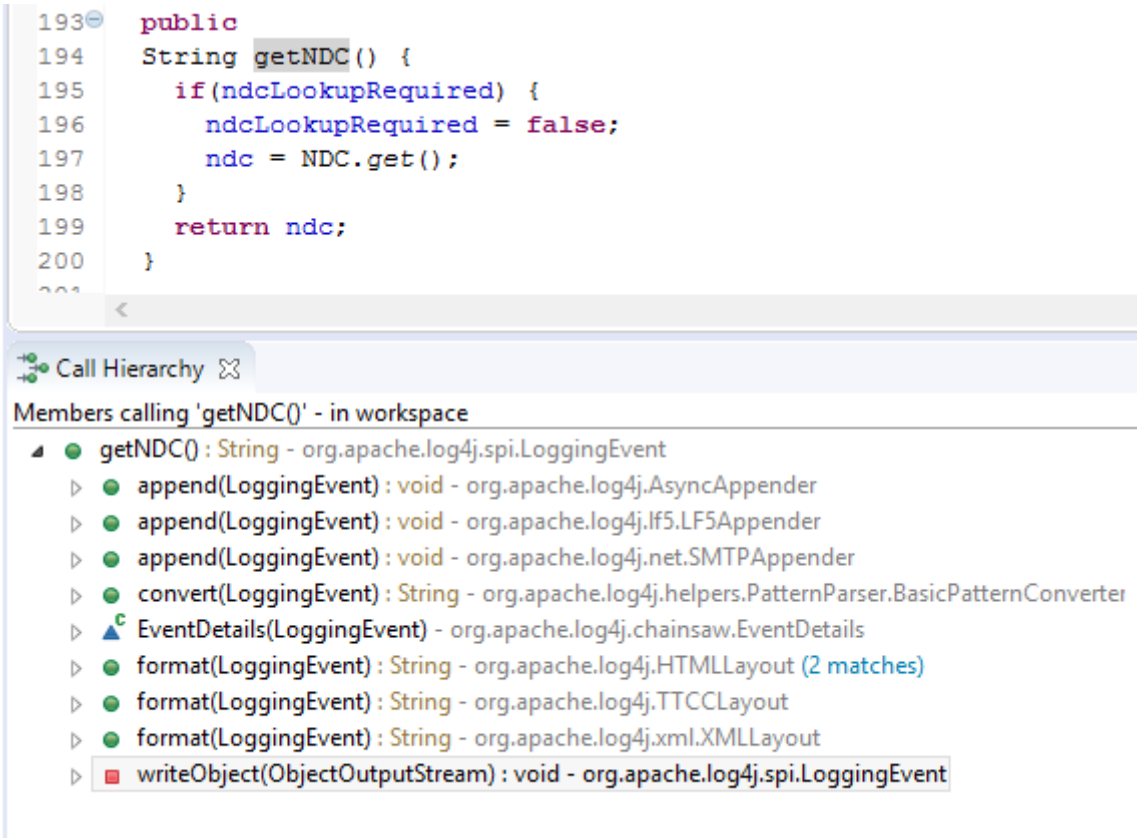


Şekil 5.7. IPLASMA Aracının Tespit Ettiği - Eklentimizin Tespit Edemediği Kötü Kokuların Dağılımı

Bu karşılaştırılmalar sonucu incelenmesi gereken en önemli nokta IPLASMA tarafından bulunup, bizim eklentimiz tarafından bulunamayan kötü kokuların neden bulunamadığıdır. 8 adet kötü kokunun her biri ayrı ayrı incelenerek 6 Shotgun Surgery için IPLASMA programının vermemesi gereken kötü koku alarmlarını

verdiği, 2 God Class'ın ise ATFD metriği için farklı eşik değerleri kullanmamızdan kaynaklı bir fark olduğu tespit edilmiştir.

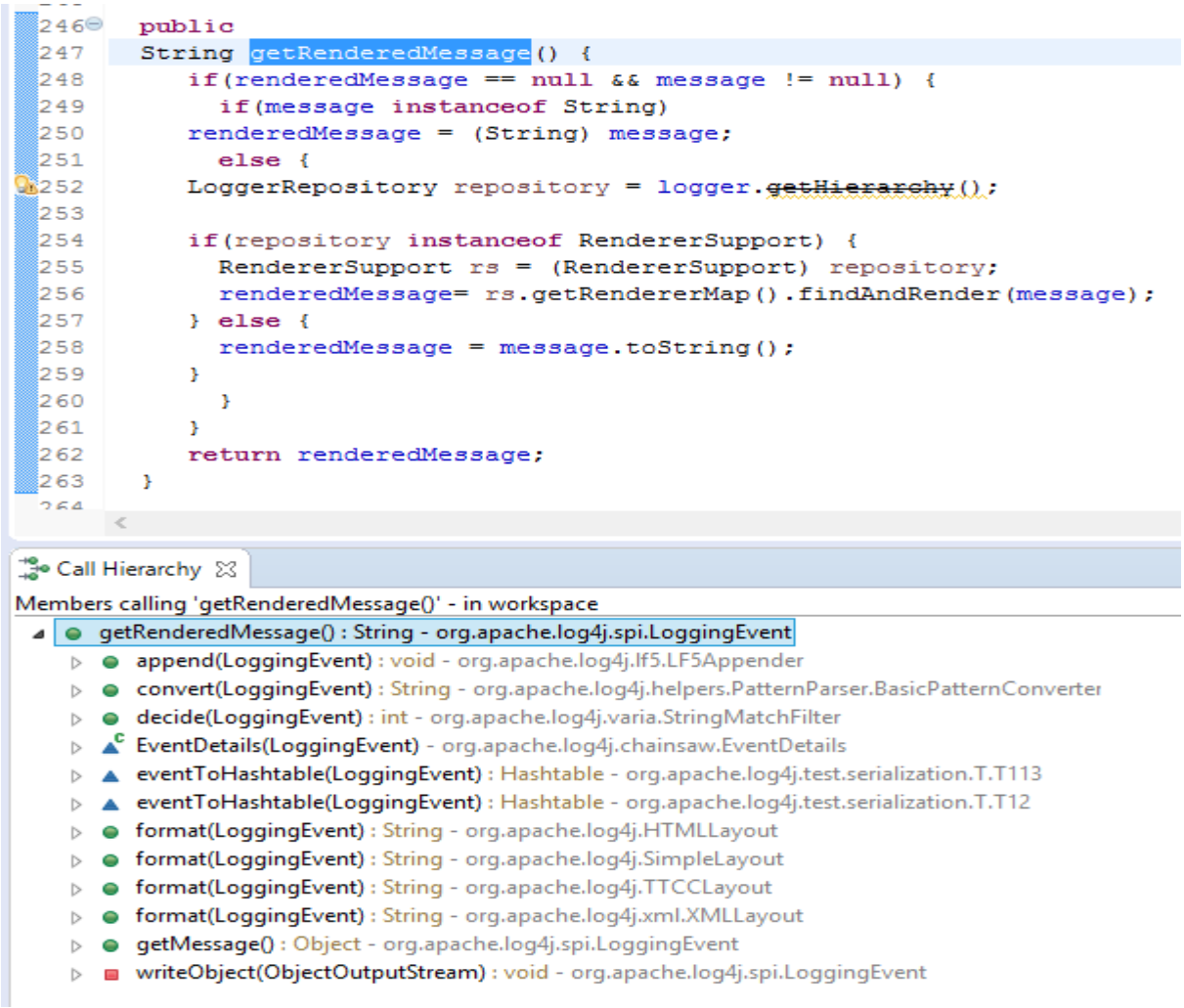
IPLASMA **org.apache.log4j.spi.LoggingEvent** sınıfı içinde yer alan **getNDC** metodu içinde **Shotgun Surgery** kötü kokusu tespit etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşide yer almayan sınıflar) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10**'dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan çağıran metot sayısı **8** olmakla birlikte, yalnızca **8** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için **yanlış** kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.8'de verilmiştir.



Şekil 5.8. GetNDC Metodu için Çağrım Hiyerarşisi

IPLASMA **org.apache.log4j.spi.LoggingEvent** sınıfı içinde yer alan **getRenderedMessage** metodu içinde **Shotgun Surgery** kötü kokusu tespit

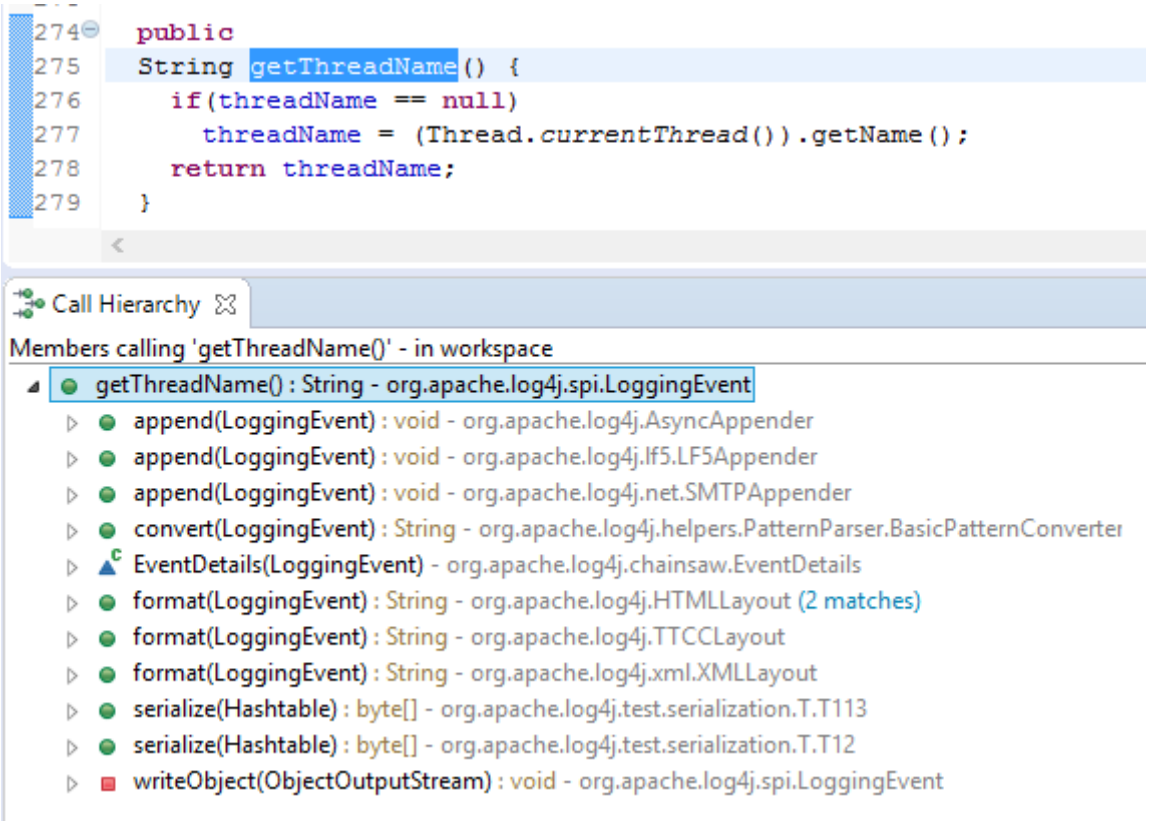
etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşide yer almayan) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10**'dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan çağıran metot sayısı **10** olmakla birlikte, yalnızca **10** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için **yanlış** kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.9'da verilmiştir.



Şekil 5.9. GetRenderedMessage Metodu için Çağrım Hiyerarşisi

IPLASMA **org.apache.log4j.spi.LoggingEvent** sınıfı içinde yer alan **getThreadName** metodu içinde **Shotgun Surgery** kötü kokusu tespit etmiştir.

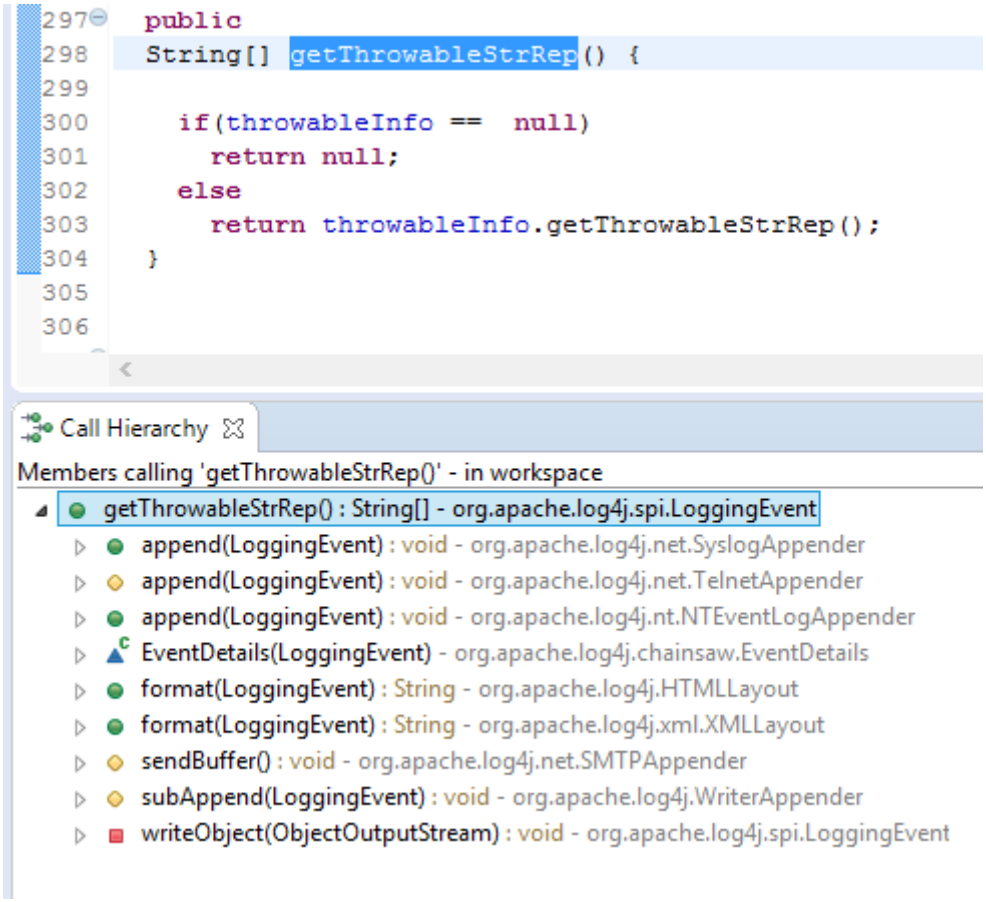
Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşide yer almayan) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10** dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan çağıran metot sayısı **10** olmakla birlikte, yalnızca **10** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için **yanlış** kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.10'da verilmiştir.



Şekil 5.10. GetThreadName Metodu için Çağrım Hiyerarşisi

IPLASMA **org.apache.log4j.spi.LoggingEvent** sınıfı içinde yer alan **getThrowableStrRep** metodu içinde **Shotgun Surgery** kötü kokusu tespit etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşjide yer almayan) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10**'dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan

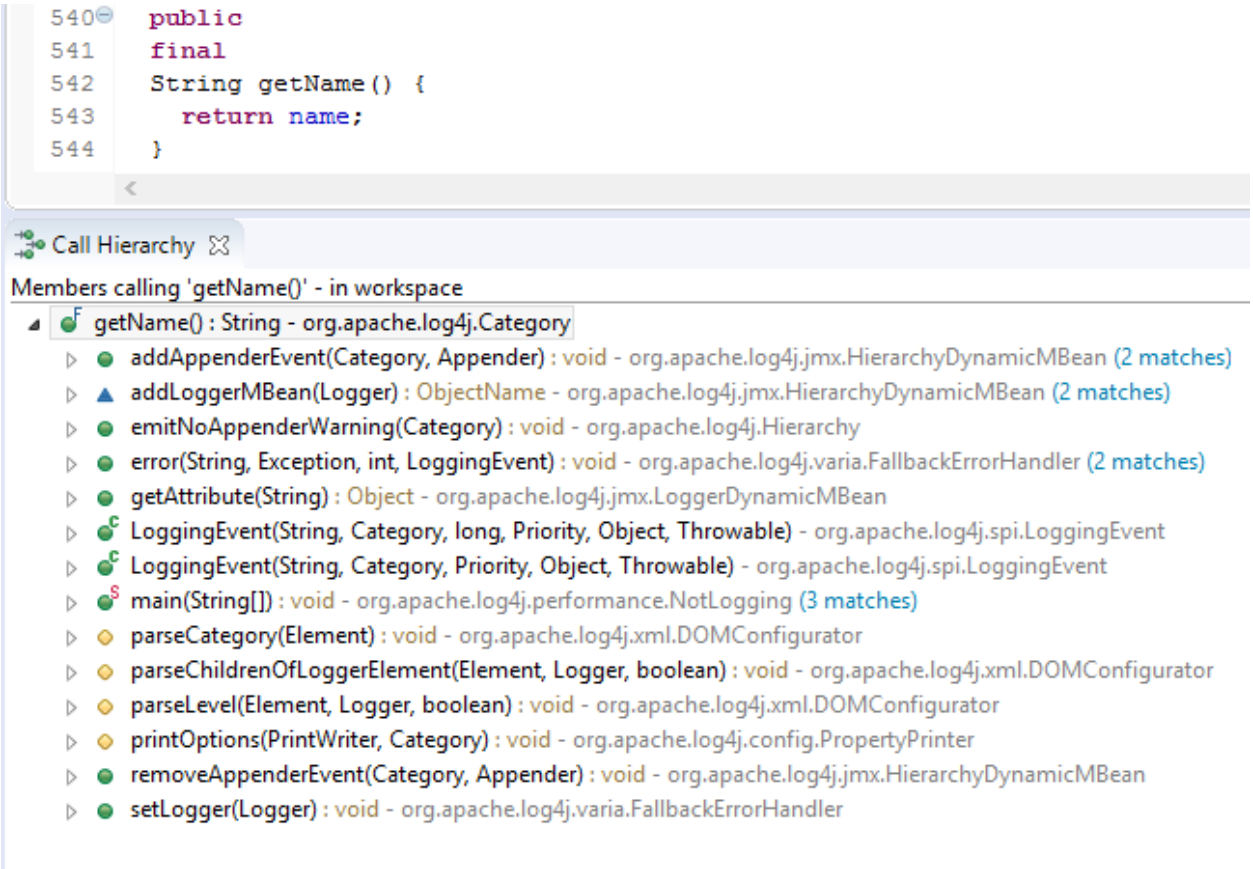
çağıran metot sayısı **8** olmakla birlikte, yalnızca **8** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için **yanlış** kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.11'de verilmiştir.



Şekil 5.11. GetThrowableStrRep Metodu için Çağrım Hiyerarşisi

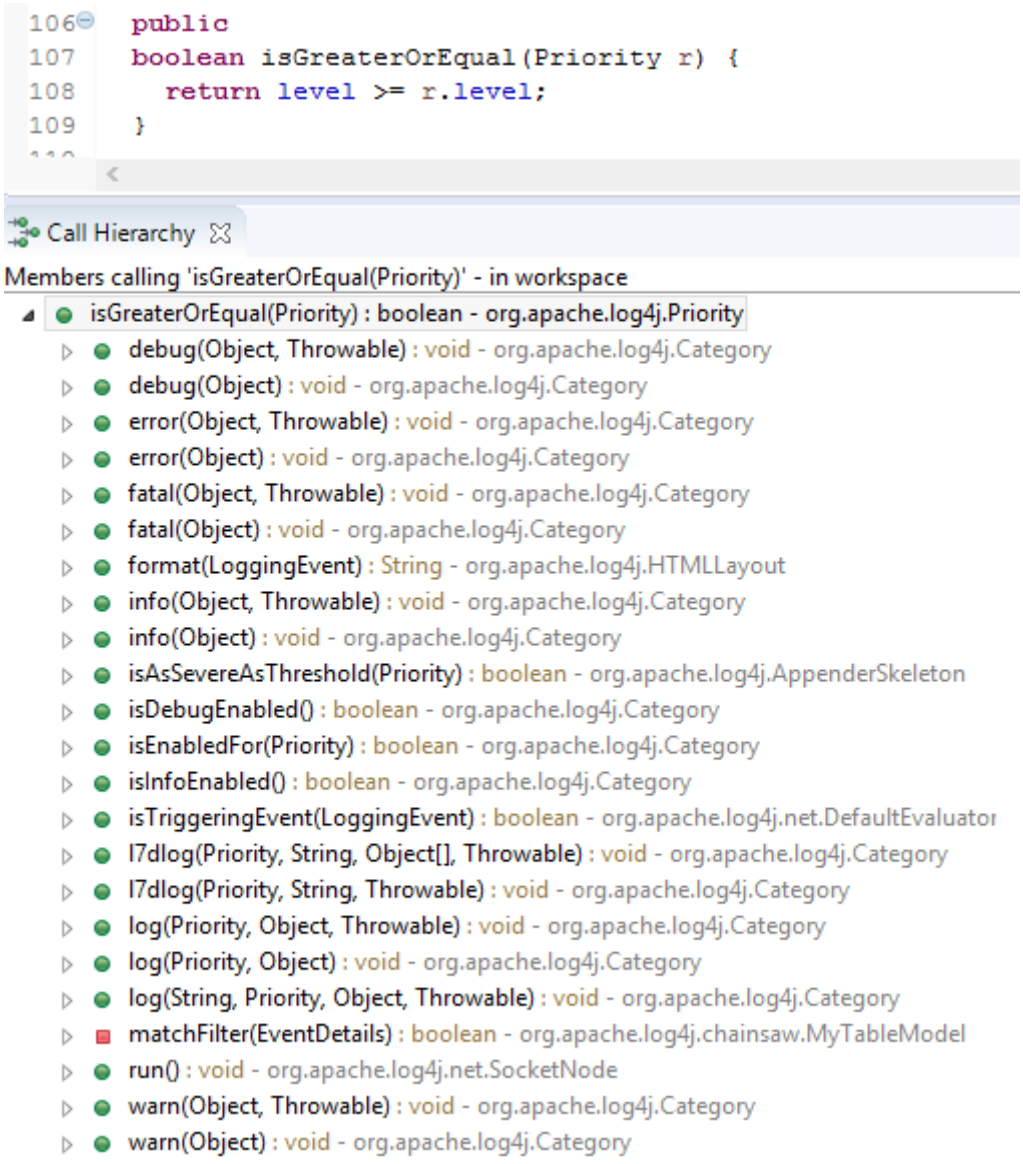
IPLASMA **org.apache.log4j.Category** sınıfı içinde yer alan **getName** metodu içinde **Shotgun Surgery** kötü kokusu tespit etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşide yer almayan) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10**'dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan çağıran metot sayısı **14** olmakla birlikte, yalnızca **9** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için

yanlış kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.12'de verilmiştir.



Şekil 5.12. GetName Metodu için Çağrım Hiyerarşisi

IPLASMA **org.apache.log4j.Priority** sınıfı içinde yer alan **isGreaterOrEqual** metodu içinde **Shotgun Surgery** kötü kokusu tespit etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Çünkü literatüre göre bir metodun Shotgun Surgery kötü kokusundan etkilenmiş olması için o metodu diğer sınıflardan (aynı hiyerarşide yer almayan) **7**'den fazla metot çağırmalı ve çağrım yapan metotlar **10**'dan fazla sınıfa dağılmış olmalıdır. Bu metodu diğer sınıflardan çağıran metot sayısı **23** olmakla birlikte, yalnızca **6** farklı sınıfa dağılmıştır. Dağılım sayısı **10**'dan fazla olmadığından metot **Shotgun Surgery** kötü kokusunu içermemelidir. IPLASMA bu metot için **yanlış** kötü koku alarmı vermiştir. Metodun çağrım hiyerarşisi Şekil 5.13'te verilmiştir.



Şekil 5.13. IsGreaterOrEqual Metodu için Çağrım Hiyerarşisi

IPLASMA **org.apache.log4j.PropertyConfigurator** sınıfını **God Class** kötü kokusundan etkilenmiş olarak tespit etmiştir. Bizim eklentimiz ise bu kötü kokuyu tespit etmemiştir. Eklentimizin bu kötü kokuyu bulamamasının nedenleri incelendiğinde temel nedenin ATFD metriğindeki eşik değerinden olduğu anlaşılmıştır. Çünkü literatüre göre bir sınıfın God Class kötü kokusundan etkilenmiş olması için WMC metrik değerinin **47**'den büyük ve eşit, TCC metrik değerinin **1/3**'ten küçük, ATFD değerinin ise 2-5 arası belirlenen bir değerden büyük olması gerekmektedir. Biz eklentimiz için 2-5 arası eşik değerini 3 olarak belirledik, ancak IPLASMA en alt değeri kullanarak ATFD metriği için eşik değerini 2 olarak belirlediği

için bu sınıf üzerindeki sonuçlar farklılık göstermiştir. Eklentimizi geliştirmemizin temel amacı; sistemde büyük sorunlar yaratabilecek sınıf ve metotları bulmak olduğundan, belirlenen en alt değeri seçmek yerine, aralığın tam ortasını eşik değeri seçmek daha mantıklı bulunmuştur. İncelenen bu sınıfın metrik değerleri; WMC için **48**, TCC için **0.042** ve ATFD için **3** olarak tespit edilmiştir. ATFD metrik değeri 3'ten büyük olmadığından biz bu sınıfı God Class olarak değerlendirmemekteyiz.

IPLASMA **org.apache.log4j.xml.DomConfigurator** sınıfını **God Class** kötü kokusundan etkilenmiş olarak tespit etmiştir. İncelenen bu sınıfın metrik değerleri; WMC için **99**, TCC için **0.16** ve ATFD için **3** olarak tespit edilmiştir. Yukarıda belirtilen nedenlerle ATFD metrik değeri 3'ten büyük olmadığından IPLASMA ile sonuçlarımız farklılık göstermiştir.

6. SONUÇLAR

Yapılan çalışma sonucunda, Java projelerindeki kaynak kodlar içerisinde yer alan kötü kokuları otomatik olarak bulan bir Eclipse eklentisi geliştirilmiştir. 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 iki 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.

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, pp. 350– 359, **2004.**

[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,October, pp. 519-520, **2007.**

[8] T. Chaikalis, N. Tsantalis and A. Chatzigeorgiou, JDeodorant: Identification and Removal of TypeChecking Bad Smells, 12th European Conference on Software Maintenance and Reengineering, pp. 329-331, **2008.**

[9] M. Fokaefs, N. Tsantalis and A. Chatzigeorgiou, JDeodorant, http://marketplace.eclipse.org/content/jdeodorant (Eylül, **2018**).

[10] Emerson Murphy-Hill and Andrew P. Black, An interactive ambient visualization for code smells, Proceedings of SOFTVIS ’10, USA, October **2010.**

[11] PMD, PMD Source Code Analyzer, https://pmd.github.io/ (Eylül, **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 Timisoara, **2002**.

[17] Apache, Log4j 1.2 Kaynak Kod, http://svn.apache.org/repos/asf/logging/log4j/tags/v1_2_1/ (Eylül, **2018**).

[18] Apache, Ivy 2.0 Kaynak Kod, http://svn.apache.org/repos/asf/ant/ivy/core/tags/2.0.0/ (Eylül, **2018**).

[19] SourceForge, PBeans 1.0 Kaynak Kod, https://sourceforge.net/projects/pbeans/files/ pbeans/1.0/ (Eylül, **2018**).

[20] Apache, Velocity 1.4 Kaynak Kod, https://svn.apache.org/repos/asf/velocity/tools/tags/1.4/ (Eylül, **2018**).

[21] Apache, Tomcat 6.0.38 Kaynak Kod, http://svn.apache.org/repos/asf/tomcat/archive/tc6.0.x/tags/TOMCAT_6_0_38/ (Eylül, **2018**).

[22] Apache, Poi 2.0.rc1 Kaynak Kod, http://svn.apache.org/repos/asf/poi/tags/REL_2_0_RC1/ (Eylül, **2018**).

[23] Apache, Synapse 1.0 Kaynak Kod, http://svn.apache.org/repos/asf/synapse/tags/1.0/ (Eylül, **2018**).

[24] Marian Jureckzo, (2010). tomcat [veri seti]. Zenodo. https://doi.org/10.5281/zenodo.322454 (Eylül, **2018**).

[25] Marian Jureckzo, velocity [veri seti] Zenodo. https://doi.org/10.5281/zenodo.322455 (Eylül, **2018**).

[26] Marian Jureckzo, Synapse Veri Seti, Zenodo, **2010**, https://doi.org/10.5281/zenodo.322449 (Eylül, **2018**).

[27] Marian Jureckzo, (2010). Poi Veri Seti, Zenodo, **2010**, https://doi.org/10.5281/zenodo.322443 (Eylül, **2018**).

[28] Marian Jureckzo, (2010). Pbeans Veri Seti, Zenodo, **2010**, https://doi.org/10.5281/zenodo.322440 (Eylül, **2018**).

[29] Marian Jureckzo, (2010). Log4j Veri Seti, Zenodo, **2010**, https://doi.org/10.5281/zenodo.268451 (Eylül, **2018**).

[30] Marian Jureckzo, Ivy Veri Seti, Zenodo, **2010**, https://doi.org/10.5281/zenodo.322436 (Eylül, **2018**).

[31] TIOBE, TIOBE Index for September, https://www.tiobe.com/tiobe-index/ (Eylül, **2018**).

[32] Simon Maple, Why Do You Use The Java Tools You Use, https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/ (Eylül, **2018**).

[33] Melih Altıntaş, Automatic Java Code Smell Detector, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector (Eylül, **2018**).

[34] Melih Altıntaş, Log4J Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/log4j_1.2/report.xlsx (Eylül, **2018**).

[35] Melih Altıntaş, PBeans Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/pbeans_1.0/report.xlsx (Eylül, **2018**).

[36] Melih Altıntaş, Ivy Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/ivy_2.0/report.xlsx (Eylül, **2018**).

[37] Melih Altıntaş, Synapse Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/synapse_1.0/report.xlsx (Eylül, **2018**).

[38] Melih Altıntaş, Velocity Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/velocity_1.4/report.xlsx (Eylül, **2018**).

[39] Melih Altıntaş, Poi Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/poi_2_0rc1/report.xlsx (Eylül, **2018**).

[40] Melih Altıntaş, Tomcat Kötü Koku Hata İlişkisi Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/tomcat_6.0.38/report.xlsx (Eylül, **2018**).

[41] Melih Altıntaş, Log4J İçin IPLASMA ile Eklentimizin Kötü Koku Sonuçlarının Karşılaştırılma Raporu, https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/IPLASMACOMPARE.xlsx (Eylül, **2018**).

[42] Vogella, Eclipse Plugin Geliştirme Eğitim Kitabı, http://www.vogella.com/tutorials/EclipseJDT/article.html (Eylül, **2018**).

[43] Jim Bethancourt, Refactor to the Limit, https://www.slideshare.net/jimbethancourt/refactor-to-the-limit (Eylül, **2018**).

[44] Melih Altıntaş, Log4J 1.2 PropertyConfigurator.Java Dosyası İçeriği https://github.com/MelihAltintas/AutomaticJavaCodeSmellDetector/blob/master/PropertyConfigurator.java (Eylül, **2018**).

ÖZGEÇMİŞ

Kimlik Bilgileri

Adı Soyadı	:	Melih ALTINTAŞ
Doğum Yeri	:	Ankara
Medeni Hali	:	Evli
E-Posta	:	mealtintas@aselsan.com.tr
Adresi	:	2861. Cad Bakyuvam Sitesi A Blok 24 Numara Çayyolu / Ankara

Eğitim

Lise	:	Ümitköy Anadolu Lisesi 2010
Lisans	:	Hacettepe Üniversitesi Bilgisayar Mühendisliği Yıl:2015 Ortalama: 3.80 Mühendislik Fakültesi Birinciliği Bilgisayar Mühendisliği Bölüm Birinciliği
Yüksek Lisans	:	Hacettepe Üniversitesi Bilgisayar Mühendisliği Ortalama: 3.29

Yabancı Dil ve Düzeyi

İngilizce	:	Okuma Dinleme Yazma: İyi Konuşma: Orta

İş Deneyimi

Aselsan Uges Yazılım Mühendisliği Mdl. (2015-Hâlen)

Deneyim Alanları

Ulaşım ve Güvenlik Sistemleri

Tezden Üretilmiş Projeler ve Bütçesi

Tezden Üretilmiş Yayınlar

Melih Altıntaş ve Ebru Akçapınar Sezer, "*Kaynak Kodlardaki Kötü Kokuların Otomatik Tespiti için Eclipse Eklenti Önerisi*", 12. Ulusal Yazılım Mühendisliği Sempozyumu, Sabancı Üniversitesi, 2018

Tezden Üretilmiş Tebliği ve/veya Poster Sunumu ile Katıldığı Toplantılar

12. Ulusal Yazılım Mühendisliği Sempozyumu (UYMS 2018)

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
YÜKSEK LİSANS/DOKTORA TEZ ÇALIŞMASI ORJİNALLİK RAPORU

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLER ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI BAŞKANLIĞI'NA

Tarih: 15/10/2018

Tez Başlığı / Konusu: KAYNAK KODLARDAKİ KÖTÜ KOKULARIN OTOMATİK TESPİTİ İÇİN ECLIPSE EKLENTİSİ

Yukarıda başlığı/konusu gösterilen tez çalışmamın a) Kapak sayfası, b) Giriş, c) Ana bölümler d) Sonuç kısımlarından oluşan toplam 107 sayfalık kısmına ilişkin, 15/10/2018 tarihinde ~~şahsım~~/tez danışmanım tarafından *Turnitin* adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı % 7'dir.

Uygulanan filtrelemeler:

1- Kaynakça hariç
2- Alıntılar hariç/~~dâhil~~
3- 5 kelimeden daha az örtüşme içeren metin kısımları hariç

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü Tez Çalışması Orjinallik Raporu Alınması ve Kullanılması Uygulama Esasları'nı inceledim ve bu Uygulama Esasları'nda belirtilen azami benzerlik oranlarına göre tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Gereğini saygılarımla arz ederim.

Tarih ve İmza

Adı Soyadı: MELİH ALTINTAŞ
Öğrenci No: N15228373
Anabilim Dalı: BİLGİSAYAR MÜHENDİSLİĞİ
Programı: BİLGİSAYAR MÜHENDİSLİĞİ
Statüsü: ☒ Y.Lisans ☐ Doktora ☐ Bütünleşik Dr.

15.10.2018

DANIŞMAN ONAYI

UYGUNDUR.

Prof. Dr. Ebru AKÇAPINAR SEZER

(Unvan, Ad Soyad, İmza)