ONTOLOJİ TABANLI KAYNAK KOD SORGULAMA İÇİN BİR ARACIN TASARIMI VE GERÇEKLEŞTİRİMİ

DESIGN AND IMPLEMENTATION OF A TOOL FOR ONTOLOGY BASED SOURCE CODE QUERYING

ÖNDER KESKİN

Hacettepe Üniversitesi

Lisansüstü Eğitim – Öğretim ve Sınav Yönetmeliğinin

Bilgisayar Mühendisliği Anabilim Dalı İçin Öngördüğü

YÜKSEK LİSANS TEZİ

olarak hazırlanmıştır.

2010

Fen Bilimleri Enstitüsü Müdürlüğü'ne,

Bu çalışma jürimiz tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI 'nda YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Başkan	:….. Prof. Dr. Candan Gökçeoğlu
Üye (Danışman)	:….. Dr. Ebru Sezer
Üye	:….. Prof. Dr. Hayri Sever
Üye	:….. Yrd. Doç. Dr. Kayhan İMRE
Üye	:….. Dr. Ahmet Burak Can

ONAY

Bu tez ...../...../..... tarihinde Enstitü Yönetim Kurulunca kabul edilmiştir.

Prof.Dr. Adil DENİZLİ
Fen Bilimleri Enstitüsü Müdürü

ONTOLOJİ TABANLI KAYNAK KOD SORGULAMA İÇİN BİR ARACIN TASARIMI VE GERÇEKLEŞTİRİMİ

ÖNDER KESKİN

ÖZ

Yazılım mühendisliğinde, yazılım üzerinde kaynak kodun bakımı ve analizi gibi temel işlemlerin gerçekleştirilebilmesi için, kodun anlaşılabilmesi son derece önemli bir konudur.

Kaynak kod sorgulama araçları, kod öğeleri arasındaki ilişkilerden faydalanarak sağladıkları sorgulama imkanı ile kod hakkında ileri düzeyde bilgi edinilmesine, dolayısı ile kodun hızlı ve etkin bir şekilde incelenmesi ve anlaşılabilmesine olanak sağlarlar.

Bu tez kapsamında, Eclipse yazılım geliştirme ortamına eklenti (plug in) olarak ontoloji tabanlı kaynak kod sorgulama aracı geliştirilmiştir. Araç gerçekleştiriminde bilgi tabanı olarak OWL-DL (Web Ontology Language – Description Logics) ile gösterilmiş ontoloji, sorgulama dili olarak SPARQL (SPARQL Protocol and RDF Query Language), anlamsal çıkarsayıcı olarak ta çıkarım motoru kullanılmıştır.

Araç geliştirme süresince; ilk olarak, Java ile hazırlanmış kaynak kodları için bir ontoloji oluşturulmuş, sonra ilgilenilen Java projesi için otomatik olarak ontoloji olgularını oluşturabilen bir ayrıştırıcı geliştirilmiştir. Son olarak, kullanıcının projeyi sorgulayabilmesi için geliştirme ortamına bir sorgu görünümü ve bu sorguların işletilmesi sonucunda elde edilen sonuçların gösterildiği bir sonuç görünümü tasarlanmıştır. Geliştirilen araç ile etkin bir biçimde kod sorgulamanın yapılabildiği ve istenilen düzeyde sonuçların elde edilebildiği gözlenmiştir.

Anahtar Kelimeler: Kaynak kod sorgulama, ontoloji, OWL

Danışman: Dr. Ebru Sezer, Hacettepe Üniversitesi, Bilgisayar Mühendisliği Bölümü

DESIGN AND IMPLEMENTATION OF A TOOL FOR ONTOLOGY BASED SOURCE CODE QUERYING

ÖNDER KESKİN

ABSTRACT

In software engineering, code comprehension is highly important for achieving primary operations such as maintenance and analyze of source code on software.

Source code querying tools enables us to get information at the advanced level, and comprehension of source code in a quick and efficient way by providing querying facility, taking advantage of relations between code elements.

In this thesis, an ontology based source code querying tool is developed for Eclipse development enviroment as a plugin. In the implementation of the tool; an ontology represented by OWL-DL (Web Ontology Language – Description Logics) is used as knowledge base, SPARQL (SPARQL Protocol and RDF Query Language) is used as the query language, and finally an Inference Engine is used as the semantic reasoner.

During the development stage of the tool; first, an ontology is designed for the source code written using Java, a parser is developed that can otomatically build ontology instance for the Java project interested. Finally, a query view and a result view, that the results of the processed query are listed, are designed for querying the project. It is observed that, code querying can be achieved in an efficient way and desired results are obtained by using developed tool.

Keywords: Source code querying, ontology, OWL

Advisor: Dr. Ebru Sezer, Hacettepe University, Department of Computer Science and Engineering

TEŞEKKÜR

Tez konusunun belirlenmesini sağlayan, tez çalışmasının hazırlanmasına ve tez metninin yazılmasına yardımcı olan Sayın Dr. Ebru Sezer'e,

Tez metnini inceleyerek biçim ve içerik bakımından son halini almasına yardımcı olan Sayın Prof. Dr. Candan Gökçeoğlu'na,

Tez metnini inceleyerek biçim ve içerik bakımından son halini almasına yardımcı olan Sayın Prof. Dr. Hayri Sever'e,

Tez metnini inceleyerek biçim ve içerik bakımından son halini almasına yardımcı olan Sayın Yrd. Doç. Dr. Kayhan İmre'ye,

Tez metnini inceleyerek biçim ve içerik bakımından son halini almasına yardımcı olan Sayın Dr. Ahmet Burak Can'a,

Tez çalışmalarında verdiği desteklerden dolayı Ali Seydi Keçeli'ye,

Çalışma boyunca desteklerini eksik etmeyen ve fikirlerini benimle paylaşan, Hacettepe Üniversitesi Bilgisayar Mühendisliği'ndeki değerli çalışma arkadaşlarıma,

Yüksek Lisans eğitimim boyunca burs vererek maddi destek sağlayan TÜBİTAK'a

ve

Hayatım boyunca, bana her konuda destek olan canımdan çok sevdiğim anne ve babama, canı gönülden teşekkür ederim.

İÇİNDEKİLER DİZİNİ

ÖZ i
ABSTRACT ii
TEŞEKKÜR iii
İÇİNDEKİLER DİZİNİ iv
ŞEKİLLER DİZİNİ vii
ÇİZELGELER DİZİNİ ix
SİMGELER ve KISALTMALAR DİZİNİ x
1. GİRİŞ 1
1.1. Problem Tanımı 1
1.2. Amaç 2
1.3. Çözümden Beklenen Yararlar 3
1.4. İçerik 3
2. GENEL BİLGİLER 4
2.1. Soyut Sözdizim Ağacı (Abstract Syntax Tree - AST) 4
2.2. Anlamsal Web (Semantic Web) 5
2.3. XML (Extensible Markup Language) 6
2.4. Ontoloji (Ontology) 7
2.5. RDF (Resource Description Framwork) 8
2.6. RDFS (RDF Schema) 8
2.7. RDFS'de Çıkarsama (RDF Schema) 9
2.8. OWL (Web Ontology Language) 10
2.9. OWL Terminolojisi 11
2.10. SPARQL (SPARQL Query Language for RDF) 13
2.11. Betimleme Mantığı (Description Logics) 15
3. KAYNAK KOD SORGULAMA İLE İLGİLİ ÖNCEKİ ÇALIŞMALAR 17

3.1. Sözcük (Lexical) Tabanlı Sorgulama 17
3.1.1. Grep Tabanlı Araçlar 17
3.1.2. Arama Motoru Yaklaşımı 17
3.1.2.1. Google Kod Arama Sayfası 18
3.1.2.2 Jsearch 19
3.1.3. Sözcük Tabalı Sorgulama Yaklaşımının Değerlendirilmesi 20
3.2. Sözdizimsel (Syntactic) Sorgulama 21
3.2.1. İlişkisel Yaklaşımlar 22
3.2.1.1. Omega 22
3.2.1.2. CIA 22
3.2.2. Çizge Tabanlı Yaklaşımlar 23
3.2.2.1. GUPRO 23
3.3. Mantıksal Programlama Dili Yaklaşımı 25
3.3.1 Jquery 25
3.3.2 CodeQuest 28
3.4. Ontoloji ve Betimleme Mantığı (Description Logics) Yaklaşımı 30
3.4.1. Sound 30
4. ONTOLOJİ TABANLI JAVA KAYNAK KODU SORGULAMA ARACININ (CODONTO) GERÇEKLEŞTİRİMİ 32
4.1. CODONTO Çözüm Modeli 32
4.2. Kod Bilgi Tabanının Oluşturulmasında Kullanılan Java Kaynak Kod Ontolojisinin Tasarlanması 33
4.2.1. Protege Ontoloji Editorü ve Pellet Çıkarsama Motoru 33
4.2.2. Java Kaynak Kod Ontolojisi 34
4.2.2.1. Kavramlar (Concepts) 35
4.2.2.2. İlişkiler (Relations) 44
4.3. Java Projesi Ontoloji Olgusunun Otomatik Olarak Oluşturulmasını Sağlayan Ayrıştırıcı Tasarımı 51
4.4. Araç Kullanım Arayüzü ve Örnek Sorgulamalar 60

4.4.1. Ayrıştırma İşleminin Başlatılması.....60

4.4.2. Sorgulama ve Sonuç Görünümleri.....61

4.4.3. Sorgu Oluşturma İşlemini Kolaylaştırmak Amacıyla Geliştirilen Java SWT Sorgulama Arayüzü.....64

4.4.4. Aracın İleri Düzey Sorgulamalarda Kullanımı Örneği: Sınıf Seviyesinde Kullanım-Bağımlılık (Use - Dependency) Çözümlemesi.....65

4.4.4.1. Değişken Kullanımı (Variable Use).....65

4.4.4.2. Metod Kullanımı (Method Use).....70

4.4.4.3. Sınıf Alanı Kullanımı (Field Use).....73

4.4.4.4. Kalıtım Yolu ile Kullanım (Inheritance Use).....75

5. SONUÇ VE ÖNERİLER.....77

KAYNAKLAR.....79

ŞEKİLLER DİZİNİ

Şekil 2.1. Eclipse'te soyut sözdizim ağacı örnek kesiti. 4
Şekil 2.2. Güncel anlamsal web yığıtı [15]. 6
Şekil 2.3. Basit bir ontoloji kesiti. 7
Şekil 3.1. Google kod arama web sayfası [40]. 18
Şekil 3.2. Google kod arama web sayfasında düzenli ifade kullanım kılavuzu [40]. ..18
Şekil 3.3. CIA kavramsal modeli [39]. 22
Şekil 3.4. GUPRO kavramsal modeli [50]. 24
Şekil 3.5. Jquery, sorgu örneği [54]. 27
Şekil 3.6. Sound genel çalışma modeli [39]. 31
Şekil 4.1. Sistemin Çözüm Modeli. 33
Şekil 4.2. Protege 3.4'te sorgulama görünümü, örnek sorgu ve sonuçlar. 34
Şekil 4.3. Java Kaynak Kod Ontolojisi Kavramlar Görünümü. 36
Şekil 4.4. (a) Değişken, Blok, Sınıf Alanı Erişimi gibi alt düzey Java kavramlarının ve aralarındaki sıradüzensel ilişkilerin gösterildiği sınıf diagramı. 42
Şekil 4.4. (b) Paket, Erişim Belirteci, Tür gibi üst düzey Java kavramlarının ve aralarındaki sıradüzensel ilişkilerin gösterildiği sınıf diagramı. 43
Şekil 4.5. Java kaynak kod ontolojisi nesne türü ilişkileri görünümü. 44
Şekil 4.6. (a) Değişken, Blok, Sınıf Alanı Erişimi gibi alt düzey Java kavramları ve aralarındaki nesne türü ilişkilerin gösterildiği sınıf diagramı 47
Şekil 4.6. (b) Paket, Erişim Belirteci, Tür gibi üst düzey Java kavramları ve aralarındaki nesne türü ilişkilerin gösterildiği sınıf diagramı 48
Şekil 4.7. Java kaynak kod ontolojisi veri türü ilişkileri görünümü. 49
Şekil 4.8. Eclipse Java Modeli [13]. 51
Şekil 4.9. *CompilationUnit* türündeki AST düğümü yapısal özellikleri. 53
Şekil 4.10. *TypeDeclaration* türündeki AST düğümü yapısal özellikleri. 55
Şekil 4.11. *MethodDeclaration* türündeki AST düğümü yapısal özellikleri. 56
Şekil 4.12. Bir sınıf alanının yazılması durumu. 57
Şekil 4.13. Bir sınıf alanının yazılması durumunun AST üzerindeki görünümü. 58
Şekil 4.14. Blok yapısı içerisindeki bir metod çağrımı (MethodInvocation). 59
Şekil 4.15. Java projesi için ayrıştırma işleminin başlatılması. 61
Şekil 4.16. Sorgu görünümü. 62
Şekil 4.17. Sonuç Görünümü. 62
Şekil 4.18. Seçilen sonucun editör üzerinde gösterilmesi. 64
Şekil 4.19. Java SWT arayüz uygulaması görünümü. 64

Şekil 4.20. Bir üst OWL ilişkisi olarak tanımlanmış, geçişken (transitive) *defines* ilişkisi. 66
Şekil 4.21. Class A ve Method B olguları arasındaki ilişki. 67
Şekil 4.22. Method B ve Block C olguları arasındaki ilişki. 67
Şekil 4.23. Block C ve LocalVariable D olguları arasındaki ilişki. 68
Şekil 4.24. Class A ve LocalVariable D olguları arasındaki ara ilişkilerin tümü. 68
Şekil 4.25. Class A ve LocalVariable D olguları arasındaki çıkarsanan ilişki. 68
Şekil 4.26. Sınıflar arasındaki değişken kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi. 69
Şekil 4.27. 'Color' Java sınıfına yerel değişken kullanımı ile bağımlı olan tüm sınıfların tesbiti için Eclipse sorgu görünümünde oluşan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi. 70
Şekil 4.28. Sınıflar arasındaki metod kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi. 72
Şekil 4.29. *MethodRef* Java sınıfına metod kullanımı ile bağımlı olan diğer sınıfların tesbiti için Eclipse sorgu görünümünde oluşturulan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi. 73
Şekil 4.30. Sınıflar arasındaki sınıf alanı kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi. 75
Şekil 4.31. Sınıflar arasındaki kalıtım yolu ile bağımlılığın tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi. 75
Şekil 4.32. *ASTAttribute* Java sınıfına kalıtım yolu ile bağımlı olan diğer sınıfların tesbiti için Eclipse sorgu görünümünde oluşturulan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi. 76

ÇİZELGELER DİZİNİ

Çizelge 3.1. Jsearch ile desteklenen bazı sorgu örnekleri [43].....20
Çizelge 4.1. ASTView eklenti projesi ayrıştırma aşamaları için işletim süreleri..... 61

SİMGELER ve KISALTMALAR DİZİNİ

Abox	Assertional Box
AST	Abstract Syntax Tree
CIA	C Information Abstraction
DAML	DARPA Agent Markup Language
GreQL	Graph Repository Query Language
HTML	Hyper Text Markup Language
JLS	Java Language Specification
nRQL	new Racer Query Language
OIL	Ontology Inference Layer
OWL	Web Ontology Language
OWL-DL	Web Ontology Language – Description Logics
RACER	Renamed Abox and Concept Expression Reasoner
RDF	Resource Description Framework
RDF-S	Resource Description Framework – Schema
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SWT	Standart Widget Toolkit
Tbox	Terminological Box
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WWW	World Wide Web
XML	Extensible Markup Language

1. GİRİŞ

1.1. Problem Tanımı

Kaynak kod, yazılımın anlaşılmasında önemli rol oynayan, birçok durumda bakım işlemlerinde programcıların ulaşabileceği tek uygulama yapısıdır.

Yazılım hakkında çok basit bazı bilgilerin elde edilmesinden, karmaşık çıkarımların otomatik olarak oluşturulmasına kadar bir çok etkinliğin temelinde kaynak kodun sorgulanabilmesi ihtiyacı vardır [1].

Günümüz yazılım endüstrisinde, geliştirme aşamalarında çok çeşitli yazılım teknolojileri birbirleri ile çalışabilir şekilde entegre edilebilmektedirler. Bunlardan bazıları; kayıt tutma (loglama), güvenlik ve veritabanı ara katman teknolojileri, geliştiricilere kolaylık sağlayan hazır alt yapı ve servis kütüphaneleridir. Bu durum, ortaya çıkan yazılım sistemlerinin karmaşıklık seviyesinin artmasına neden olur.

Brachman ve diğerlerinin yaptığı araştırmada [2], geniş kapsamlı bir yazılımın bakımı ile ilgilenen kişiler ve bu kişilerin yaptığı farklı kategorilerdeki işlemler gözlemlenmiş ve tüm bu bakım işlemleri için harcanan zamanın %60'ının, yazılım üzerinde gerçekleştirilen sorgular ve kod kesimleri arasında gezinme şeklinde geçtiği sonucuna ulaşılmıştır.

Bu bağlamdaki çalışmalar göstermiştir ki; yazılımın büyüklüğü ve karmaşıklığı arttığında, yazılım üzerindeki denetim ve bakım işlemleri çok daha zor hale gelmekte ve sonuç olarak, yazılımı anlamaya yönelik geliştirilen araçların önemi daha da artmaktadır.

Sim ve diğerleri'nin çalışmasında [3], yazılım geliştiricilerin kod sorgulama alışkanlıklarını konu alan bir araştırma yapılmış, sorgulamanın en çok hata belirleme, kodun yeniden kullanımı (herhangi bir işlevselliği gerçekleştiren kod kesiminin aranması), programı anlama (ilgilenilen metod veya değişken bildirimlerinin ve bunların kod üzerindeki kullanımlarının incelenmesi), etki analizi (yapılacak bir değişikliğin kodun diğer kesimlerini nasıl etkileyeceğinin incelenmesi), kod temizleme (örneğin main metod tarafından dolaylı yada dolaysız

olarak hiç çağrılmayan metodların belirlenmesi) gibi etkinliklerde ihtiyaç duyulduğu sonucuna varılmıştır.

Kaynak kod sorgulama ve arama, birçok yazılım mühendisliği aracında etkin biçimlerde kullanılıyor olmasına rağmen; amaç, hedef ve stratejiler bağlamında halen yenilikçi yorumlara açık bir konu olarak görülmektedir [4].

Yazılım geliştirme araçları (örneğin Eclipse [5], Visual Studio [6] vb. gibi), geliştiricilerin başlıca ihtiyaçlarını karşılamak üzere kod üzerinde inceleme olanakları sunarlar. Ancak bu ortamların sunduğu inceleme yetenekleri sabit olup, geliştiricilerin yeni ya da farklı bakış açıları ile kod üzerinde inceleme/sorgulama biçimleri tanımlayıp, kullanabilmelerini sağlayacak biçimde esnek değillerdir. Örneğin Eclipse, Java programcılarına herhangi bir sınıf alanına (class field) hangi kod kesimlerinde okuma ve yazma amaçlı erişimler yapıldığı, proje içerisindeki herhangi bir arayüzü (interface) hangi sınıf ya da sınıfların gerçekleştirdiği gibi bilgilere erişim imkânı tanır. Ancak, görüleceği üzere bu tür sorgu kalıpları sabittir ve dolayısı ile sınırlı türe sahiptirler.

Sonuç olarak, yazılım geliştirme ortamları için esnek sorgulama ihtiyacı önemli bir araştırma konusu haline gelmiş, bu konuda birçok çalışma yapıldığı gözlenmiştir. Yapılan çalışmaların genel özelliklerine bakıldığında, iki önemli nokta üzerinde yoğunlaşıldığı görülür. Bu noktaların ilki, kaynak kod bilgilerinin saklanma biçimi ve kullanılan bilgi tabanının yapısı, ikincisi ise bu bilgi tabanı üzerinden istenilen sorgulamaların yapılabilmesine imkân verecek olan sorgulama mekanizmasıdır.

1.2. Amaç

Tez kapsamında, yukarıda özetlenen konunun önemi dikkate alınarak, Java programlama dili kullanılarak geliştirilmiş yazılımlar için esnek ve ileri düzey sorgulamalara olanak sağlayan bir yöntemin geliştirilmesi amaçlanmıştır. Bu yöntemin uygulandığı CODONTO adındaki Java kodu sorgulama aracı, Eclipse geliştirme ortamına eklenti olarak geliştirilmiştir. Bu çalışmada geliştirilen aracın benzerlerinden farkları, bilgi tabanı olarak OWL-DL ile gösterilmiş bir ontoloji, sorgulama dili olarak ta SPARQL kullanılmış olmasıdır. Ontoloji bilgi tabanı kullanımı ile kod bilgilerinin sorgulanabilir bir model üzerine oturtulması amaçlanmıştır. Bilgisayar bilimlerinde ontoloji, herhangi bir alan kapsamındaki

kavramlar ve bu kavramlar arasındaki ilişkileri temsil eden bir veri modeli olarak tanımlanabilir.

1.3. Çözümden Beklenen Yararlar

Geliştirilen yönteminin uygulanması sonucunda beklenen yararlar şu şekilde listelenebilir:

- Nesneye yönelik bir programlama dili olan Java için temel kaynak kod kavramlarının (paket, sınıf, metod gibi) ve birbirleri arasındaki ilişkilerin etkin biçimde ifade edilebilmesi,
- Java kaynak kod bilgilerinin sorgulanabilir bir model üzerine oturtularak, kod kesimi arama ve ileri düzey kod analizleri gibi amaçlar için sorgulamaların yapılabilmesi,
- Özyinelemeli (recursive) sorguların, geçişken özellik kullanımı ile diğer kod sorgulama yöntemlerine göre daha basit bir şekilde ifade edilebilmesi.

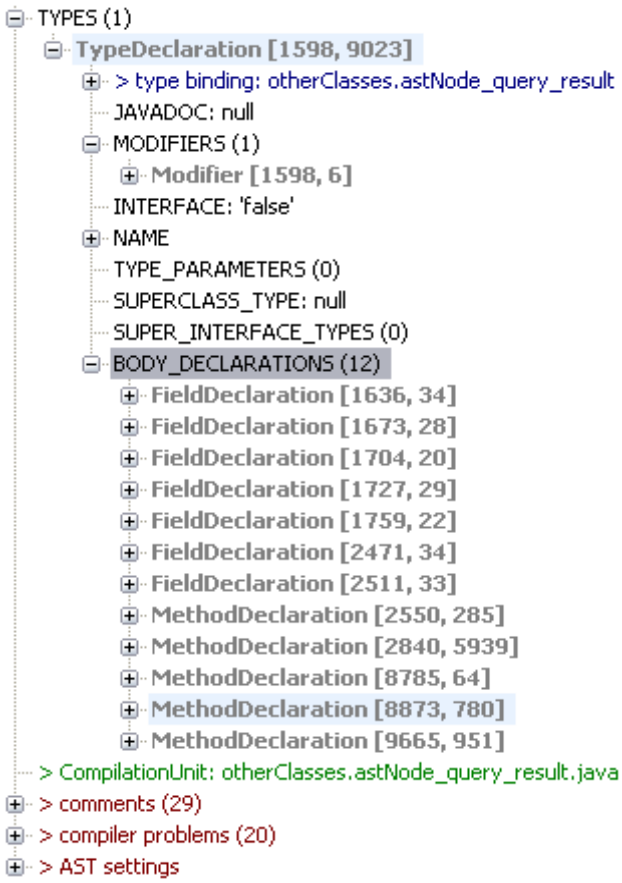
1.4. İçerik

Tezin 2. bölümünde, tez çalışması kapsamında kullanılan ontoloji, RDF [7], RDF-S, OWL [8], OWL-DL, SPARQL [9], Pellet [10] gibi teknolojilerden bahsedilmiştir. Bunların yanı sıra, AST (Abstract Syntax Tree), anlamsal web gibi konular hakkında da kısa bilgi verilmiştir. 3. bölümde, kaynak kod sorgulama konusunda literatürde yapılan çalışmalar için bir sınıflandırma yapılmış, her bir sınıf hakkında bilgi verilerek örnek çalışmalar sunulmuştur. Tezin 4. bölümünde, CODONTO aracının çözüm modeli sunulmuştur. Bu kesimde, kod bilgi tabanının oluşturulmasında kullanılan Java programlama dili ontolojisi, otomatik olarak java proje olgularını oluşturan kod ayrıştırıcısı ve Eclipse geliştirme ortamı için geliştirlen eklenti hakkında bilgi verilmiş ve örnekler sunulmuştur. Ayrıca aynı bölümde, SPARQL sorgularının oluşturulması için geliştirilen SWT (Standart Widget Toolkit) [11] arayüz uygulaması hakkında bilgi verilmiş ve örnekler sunulmuştur. Son olarak 5. bölümde ise, genel olarak tez çalışması kapsamında yapılan denemeler irdelenmiş ve gelecekte bu çalışmalara kazandırılması faydalı olacak özellikler üzerinde durulmuştur.

2. GENEL BİLGİLER

2.1. Soyut Sözdizim Ağacı (Abstract Syntax Tree - AST)

Soyut sözdizim ağacı, Eclipse gibi geliştirme ortamlarında, yeniden düzenleme (refactoring), hızlı onarım (quick fix), hızlı yardım (quick assist) gibi yardımcı araçlar için temel teşkil eden bir altyapı ve kaynak kodun ağaç gösterimidir. Ağaç yapısı, düz metin-tabanlı yapıya göre, kodu çözümleme ve değiştirme bakımından daha güvenilir ve uygundur. Eclipse ortamında java kaynak kodu soyut sözdizim ağaçlarının nasıl göründüğü, ASTView [12] eklentisi ile incelenebilir. Şekil 2.1'de ASTView eklentisi tarafından oluşturulmuş, bir Java kaynak kütüğüne ait soyut sözdizim ağacı kesiti gösterilmiştir.



Şekil 2.1. Eclipse'te soyut sözdizim ağacı örnek kesiti.

Her bir Java kaynak kütüğü, AST düğümlerinden oluşan bir ağaç olarak temsil edilebilir. Ağaç üzerindeki tüm düğümler, *ASTNode* sınıfının alt sınıflarıdır. Java programlama dilindeki her kod biriminin ağaç üzerinde bir karşılığı vardır ve her birim bir *ASTNode* alt sınıfı ile temsil edilir. Örneğin, metod bildirimleri

MethodDeclaration alt sınıfı ile, değişken bildirimleri ise *VariableDeclarationFragment* alt sınıfı ile temsil edilirler. En sık kullanılan düğümlerden bir tanesi *SimpleName* türündeki düğümdür. Örneğin, 'i = j + 7;' gibi bir atama işleminde, i ve j birer *SimpleName* türünde düğüm ile temsil edilirler.

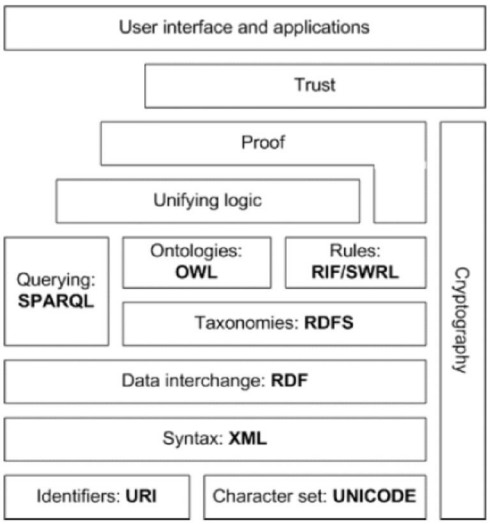
Soyut sözdizim ağacı, kaynak kodun en alt düzeyde detaylandırılmış mantıksal modelidir ve kod ile ilgili tüm işlemlerde doğrudan kullanılabilir. Kaynak koda ait soyut sözdizim ağacı üzerinden her türlü analizi yapılıp, istenilen tüm bilgi ve çıkarımlar elde edilebilir. Bunun için Eclipse'in sunduğu kütüphaneler (org.eclipse.jdt.core [13] gibi) kullanılıp, amaca uygun yazılım geliştirilmelidir.

Tez kapsamında, analiz edilecek olan Java projesini oluşturan herbir kaynak kod kütüğe ait soyut sözdizim ağacı, geliştirdiğimiz ayrıştırıcı tarafından analiz edilip, kod hakkındaki amaca uygun tüm kavram ve ilişkiler elde edilerek ontoloji olguları oluşturulur. Bu şekilde kod bilgileri daha kolay sorgulanabilir bir model üzerine oturtulmuş olur.

Proje içerisindeki herhangi bir sınıfa ait 'A' ismindeki soyut (abstract) bir metodu geçersiz kılan tüm metodlara ulaşabilmek için soyut sözdizim ağacı kullanılarak geliştirilen yazılım hem sadece belli bir amaca hizmet edecek, hem de programlama açısından karmaşık işlemleri gerektirecektir. Bunun yerine proje kapsamındaki tüm metod olgularının, erişim belirteçleriyle (access modifier) ve ait oldukları sınıflarla ilişkilendirildiği bir ontoloji olgusu üzerinden gerçekleştirilecek bir sorgu ile istenilen sonuç elde edilebilir [1].

2.2. Anlamsal Web (Semantic Web)

Anlamsal web bilgilerin, bilgisayar ve insanların işbirliği içerisinde çalışabilmesine imkân verecek şekilde, iyi tanımlanmış anlamlarda verildiği bir WWW (World Wide Web) eklentisidir [14].



Şekil 2.2. Güncel anlamsal web yığıtı [15].

Şu anki WWW, çok büyük hacimdeki bilgi yığınından oluşan bir yapıdır. HTML gösterim dili, tasarımı gereği web sayfalarının insanlar tarafından anlaşılması temelinde geliştirilmiştir. Büyük miktarlardaki bilginin, üretilme hızı ve ona olan gereksinim düşünüldüğünde, ne anlama geldiğinin anlaşılması önemlidir ve buradaki anlamanın insanlar tarafından yapılması giderek zorlaşmaktadır. İleri sürülen anlamsal web fikri ile, bilgisayarların yüksek işlem gücünden yararlanarak bilgilerin yapılandırılacağı, verinin anlamının kendi içinde barındırılması suretiyle verinin anlam kazanacağı ve veriler arasındaki ilişkilerin de anlamlandırılması ile şu anki web ile ilgili sorunların aşılacağı düşünülmektedir. Anlamsal web'in teknolojik bileşenleri XML, web servisleri ve ontolojilerdir [16].

2.3. XML (Extensible Markup Language)

XML, anlamsal web'in en önemli yapı taşlarından biridir. W3C [17] (World Wide Web Consortium) organizasyonu tarafından tasarlanan ve herhangi bir kurumun tekelinde bulunmayan XML'in ana kullanım nedeni, organizasyon içinde ve dışında veri değişiminin sağlanmasıdır. Bu bakış açısından XML, birlikte çalışabilirlik sağlayan önemli bir araçtır. XML dört temel konuda başarı ile kullanılmaktadır [18]:

- XML uygulama bağımsız veri ve belge yaratmaktadır.
- Üst veri (meta data) ortamı için standart bir gösterim sunmaktadır.

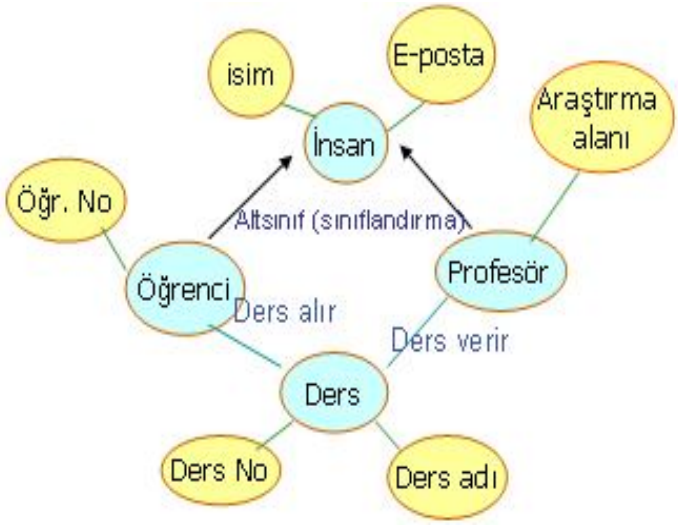
- Veri ve belge için ortak yapısal standartlar sunmaktadır.
- XML sınanmış bir teknolojidir.

XML hem bir dil hem de bir teknoloji olarak, bir verinin biçimlendirilmesi, tanımlanması ve verilerin yapılandırılmasında kullanılmaktadır. Dolayısı ile veriler standart bir şekilde tanımladığından, web ortamında veya herhangi iki program arasında veri alış verişi kolaylaşmaktadır. Bu özellikleri nedeniyle XML, anlamsal web'in geliştirilmesinde önemli bir konuma sahiptir [16].

2.4. Ontoloji (Ontology)

Ontoloji, herhangi bir alana ait kavramsal modelin, açık ve biçimsel tanımıdır. Burada belirtilen, kavramsallık biçimseldir, dolayısı ile bilgisayar tarafından anlaşılabilir [19]. Ontolojiler, herhangi bir uygulama alanı için tasarlanabilir. Örnek olarak; bilgi yönetimi, doğal dil işleme, e-ticaret, zeki bilgi bütünleştirme, bilgi elde edinimi, veri tabanlarının bütünleştirilmesi, biyo-bilişim ve eğitim verilebilir [20].

Ontolojiler kavramlar (sınıflar), ilişkiler (nitelikler), olgular ve aksiyomlardan oluşur. Şekil 2.3'te basit bir ontoloji kesiti gösterilmiştir.



Şekil 2.3. Basit bir ontoloji kesiti.

Şekil 2.3 için, *Öğrenci* bir kavram, *Öğr. No* ve *Ders alır* birer ilişki, *Matematik* adındaki bir ders olgu, "Öğrenci sınıfı, İnsan sınıfının bir alt sınıfıdır" ifadesi ise bir aksiyom örneği olarak gösterilebilir.

2.5. RDF (Resource Description Framwork)

RDF, World Wide Web üzerinde bulunan kaynaklar hakkındaki bilgileri ifade edebilmek için geliştirilmiş bir veri modelidir. Bu model, varlık-ilişki modelinde olduğu gibi bir kavramsal modelleme yaklaşımı olup, RDF terminolojisinde üçlüler (triples) olarak adlandırılan, 'özne-yüklem-nesne' üçlemeleri ile ifade edilir.

Özne, yüklem ve nesne elemanlarından herbiri bir URI'ye (Uniform Resource Identifier) sahiptir. URI, kaynak için biricik olan değerdir. Bu değer bir internet adresi olabileceği gibi bir kimlik numarası da olabilir. Özne, üzerinde konuşulan herhangi bir varlıktır (yazar, kitap, yer, kişi gibi). Yüklemler, özne ve nesne arasındaki ilişkiyi tanımlar. Nesne, öznelerin aldığı değerdir. Basit veri türünde olabileceği gibi başka URI'lerde nesne olarak kullanılabilir.

RDF ifadelerinin XML'de yazımı, gösterimi, ve uygulamalar arası taşınması için RDF/XML sözdizimi standardı kullanılır [21].

2.6. RDFS (RDF Schema)

RDFS, RDF veri modelini genişleten bir tür sistemi olup, RDF ile ifade edilen modelin sözlüğünün oluşturulmasında kullanılır. Bu sözlükte, ilgili alanda kullanılacak olan varlıklar, varlıklar arasındaki ilişkiler, nitelikler, nitelikler arasındaki ilişkiler ve niteliklerin alabilecekleri değerler tanımlanır. Aşağıda bir RDFS örneği Türkçe'ye çevrilerek verilmiştir [22].

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xml:base="http://www.cicekler.fake/cicekler#">
<rdf:Description rdf:ID="Çiçek">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>
<rdf:Description rdf:ID="Papatya">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
        <rdfs:subClassOf rdf:resource="#Çiçek"/>
```

</rdf:Description>
</rdf:RDF>

Örnekte “Papatya, bir Çiçek’tir” ifadesi modellenmiştir. Bu ifadede *Çiçek* ve *Papatya* varlık adları olup, bu varlılar arasında kalıtım (is-a) ilişkisi vardır.

2.7. RDFS’de Çıkarsama (RDF Schema)

Çıkarsama, belirli bir metodolojiyi kullanarak verilen öncüllerden hareket ederek bir sonuca ulaşma eylemidir. RDFS’de çıkarsama üç yapıdan oluşur; beyan edilen üçlüler, çıkarsanan üçlüler ve çıkarsama kuralı. RDFS çıkarsama kurallarından en önemlileri şunlardır [23]:

Tür Yayılım Kuralı:

IF ?A rdfs:subClassOf?B . AND ?x rdf:type?A . THEN ?x rdf:type?B .	Beyan Edilen Üçlüler: A, B’nin alt sınıfıdır. X, A sınıfının bir üyesidir. Çıkarsanan Üçlü: X, B sınıfının da bir üyesidir.

İlişki Yayılım Kuralı:

IF ?P rdfs:subPropertyOf?R . AND ?A P ?B . THEN ?A R ?B .	Beyan Edilen Üçlüler: P, R’nin alt ilişkisi’dir. A ile B arasında, P ilişkisi vardır. Çıkarsanan Üçlü: A ile B arasında, R ilişkisi de vardır.

Tanım Kümesi Veri Türleme Kuralı (Tanım Kümesinden Olgu Sınıfının Tesbiti):

IF P rdfs:domain D. and x P y. THEN x rdf:type D.	Beyan Edilen Üçlüler: P ilişkisinin tanım kümesi D'dir. x ile y arasında, P ilişkisi vardır. Çıkarsanan Üçlü: X, D sınıfının bir üyesidir.

Erim Kümesi Veri Türleme Kuralı (Erim Kümesinden Olgu Sınıfının Tesbiti):

IF P rdfs:range R. and x P y. THEN y rdf:type R.	Beyan Edilen Üçlüler: P ilişkisinin erim kümesi R'dir. x ile y arasında, P ilişkisi vardır. Çıkarsanan Üçlü: y, R sınıfının bir üyesidir.

2.8. OWL (Web Ontology Language)

RDFS, RDF sözlükleri oluşturmak için basit yetenekler sunmaktadır. Gelişmiş ontolojiler oluşturabilmek için bu yetenekleri genişleten üst seviye dillere ihtiyaç duyulmuştur. DAML, OIL ve OWL gelişmiş ontoloji dilleri olup, çokluk kısıtlamaları, geçişken ve simetrik özellikler, ayrık sınıflar gibi tanımlamalara imkan verirler [21].

OWL, W3C tarafından tanımlanmış bir gösterim dilleri ailesi olup, ifadesel zenginliklerine göre OWL Lite, OWL DL ve OWL Full olmak üzere üç türü vardır. [24]

1. **OWL-Lite:** OWL'nin ifadesel olarak en dar kapsamlı türüdür. Sadece basit sınıf (kavram) sıradüzenleri ve basit kısıtların ihtiyaç duyulduğu durumlarda kullanılabilir.

2. **OWL-DL:** Tez kapsamında kullanılan OWL DL, tanımlama mantığı (description logic) olarak bilinen birinci dereceden mantık (first order logic)

kuralları üzerine geliştirilmiştir. Bütün çıkarsamaların hesaplanabilir ve sonlu bir zamanda bitebilir olduğu en üst seviye anlamsallığa sahiptir [25]. Bu özelliği, otomatik çıkarsamaya imkan verir. İfadesel olarak, OWL-Lite'a göre daha güçlüdür.

3. **OWL-FULL:** OWL'nin ifadesel olarak en güçlü türüdür. Bunun yanında, otomatik çıkarsamanın gerektirdiği birçok kısıt OWL-FULL ontolojilerinde uygulanmaz. Örneğin, bir sınıf başka bir sınfın örneği olarak temsil edilebilir. Bu gibi durumlar, OWL-FULL kullanılarak hazırlanan ontolojilerde çıkarsama özelliğini ortadan kaldırır.

2.9. OWL Terminolojisi

Tüm OWL türleri için ortak bir terminoloji belirlenmiştir [26].

- **Sınıf (Class):** OWL'de sınıf, olgular kümesine karşılık gelir. Bir sınıf, başka bir sınıfın niteliklerini kalıtarak, onun alt sınıfı olabilir. Bu özelliğe "yönetici sınıfı çalışanlar sınıfının alt sınıfıdır" örneği verilebilir. Tüm OWL sınıfları, *owl:Thing* sınıfından türerler.

- **Olgu (Instance):** Herbir sınıf örneğine bir başka deyişle sınıfın yaratılmış nesnesine karşılık gelir. Örneğin, *Çalışanlar* bir sınıf ise, *A* kişisi bu sınıfın bir olgusudur.

- **Nitelikler (Property):** Nitelikler, OWL sınıflarının özelliklerinin ifade edilmesini sağlayan tek yönlü ikili ilişkilerdir. İki türü vardır:

    1. **Veri Türü Niteliği (DataType Property):** OWL sınıf olguları ile RDF verileri (dizgi, tam sayı ...) arasındaki ilişkilerdir. Örnek olarak, *İnsan* sınıfı için *yaşındadır* veri türü ilişkisi tanımlanmış ise, "Ali, 18 yaşındadır" ifadesi verilebilir.

    2. **Nesne Niteliği (Object Property*):*** İki OWL sınıfının olguları arasındaki ilişkileridir. Örnek olarak, tanım ve erim kümesi *İnsan* olan, *babasıdır* nesne ilişkisi tanımlanmış ise, "Ali, Ahmet'in babasıdır" ifadesi verilebilir.

- **İşleçler (Operators):** Sınıflar üzerinde, birleşim (union), kesişim (intersection), tümleme (complement) gibi küme işlemlerini gerçekleştirmek için tanımlanmışlardır. Bunların yanında sayı kısıtlaması (cardinality constraint) ve ayrıklık (disjoint) gibi özellikler de kullanılabilir. İşleçler var olan sınıflardan, yeni sınıflar türetmek için kullanılabilirler.

<owl:Class> <owl:complementOf> <owl:Class rdf:about="#Et"/> </owl:complementOf> </owl:Class>	Şeklinde belirtilen sınıf tanımı, 'et' olmayan tüm olguları kapsar.

<owl:Restriction> <owl:onProperty rdf:resource="#ebeveyneSahip" /> <owl:maxCardinality rdf:datatype="&xsd;nonNegativeInteger"> 2 </owl:maxCardinality> </owl:Restriction>	Şeklinde belirtilen sınıf tanımı, en fazla 2 ebeveyne sahip olan tüm olguları kapsar.

- **Nitelik Özellikleri (Attributes of Properties):**

owl:inverseOf nitelik özelliği:

IF P owl:inverseOf Q . and x P y . THEN y Q x .	<u>Beyan Edilen Üçlüler:</u> P ilişkisi, Q ilişkisinin tersidir. x ile y arasında (x'ten y'ye doğru), P ilişkisi vardır. <u>Çıkarsanan Üçlü:</u> y ile x arasında (y'den x'e doğru), Q ilişkisi vardır.

owl:transitiveProperty nitelik özelliği:

IF	Beyan Edilen Üçlüler:
P rdf:type owl:TransitiveProperty. and X P Y . and Y P Z . THEN X P Z .	P ilişkisi, geçişken (transitive) bir ilişkidir. x ile y arasında P ilişkisi vardır. y ile z arasında P ilişkisi vardır. Çıkarsanan Üçlü: x ile z arasında P ilişkisi vardır.

owl:functionalProperty nitelik özelliği:

IF	Beyan Edilen Üçlüler:
P rdf:type owl:FunctionalProperty. and X P A . and X P B . THEN A owl:sameAs B .	P ilişkisi, fonksiyonel (functional) bir ilişkidir. X ile A arasında P ilişkisi vardır. X ile B arasında P ilişkisi vardır. Çıkarsanan Üçlü: A ile B aynıdır.

2.10. SPARQL (SPARQL Query Language for RDF)

SPARQL anlamsal web için bir sorgulama dili ve veri erişim protokolüdür. W3C tarafından RDF veri modeli için tanımlanmıştır. RDF sorgulama dilleri üzerindeki çalışmalar son birkaç yıldır devam etmektedir. Bu süreç içerisinde RDQL [27], Squish [28], Versa [29] gibi farklı yaklaşımların kullanıldığı diller geliştirilmiştir. RDQL ve Squish'e benzer olarak SQL sözdizimini örnek alan bir dil olan SPARQL kendisine geniş bir kullanım alanı bulmuştur. RDF ve OWL sorgulama araçlarının büyük çoğunluğu SPARQL desteği sunmaktadır.

```
# ön ek bildirimleri
    PREFIX foo: <http://example.com/resources/> ...
# sonuç yancümlesi
```

```
    SELECT ...
# sorgu kalıbı
    WHERE {    ...   }
# sorgu değiştiricileri
    ORDER BY ...
```

Yukarıda belirtilen SPARQL sorgu biçimi bileşenleri şu şekilde açıklanabilir [30]:

- URI'leri, yani biricik kaynak tanıtıcılarını kısaltmada kullanılan *ön ek bildirimleri*,
- sorgudan elde edilmek istenen bilgilerin belirtildiği *sonuç yancümlesi*,
- veri kümesi üzerinde yapılacak sorgunun belirtildiği *sorgu kalıbı*,
- sıralama, bölme gibi sonuç kümesi üzerinde düzenleme yapmaya yarayan *sorgu değiştiricileri*.

Elementlere ait isim, sembol, renk, atom numarası ve kütle bilgilerinin yer aldığı periyodik tablo veri seti [31] üzerinden, elementlere ait isim, atom numarası ve renk bilgilerini, atom numarasına göre sıralanmış biçimde çekebilmek için tanımlanmış örnek sorgu aşağıda gösterilmiştir [32].

```
PREFIX table:
<http://www.daml.org/2003/01/periodictable/PeriodicTable#>
SELECT ?name ?number ?color
WHERE
{
    ?element table:name ?name.
    ?element table:symbol ?symbol.
    ?element table:atomicNumber ?number.
    OPTIONAL { ?element table:color ?color. }
}ORDER BY ?number
```

SPARQL herhangi bir sorguyu, yapısal yada yarı-yapısal RDF çizge-kalıpları üzerinden eşleştirme yolu ile sonuçlandırır.

Periyodik tablo örneği için, bazı elementlerin renk bilgilerinin olmadığı bilinmektedir. Buna rağmen, sorgu içerisinde renk bilgileri olmayan elementlerin

de sonuç kümesi içinde yer alması *OPTIONAL* anahtar sözcüğü ile sağlanmaktadır. Bu belirtim yapılmasaydı, renk bilgisi olmayan elementler sonuç listesinde yer almayacaktı. Bu özellik, SQL'deki 'left outer join' yapısı ile benzerdir.

Periyodik tablo örneği için, rengi olmayan elementlerin listelenmesi istendiğinde, yani tümleyen alma (negation) işlemi için *OPTIONAL – BOUND* yapısı kullanılır. Bu özellik, SQL'deki *NOT EXISTS* yapısı ile benzerdir. Aşağıda, tümleyen alma örnek sorgusu gösterilmiştir [32].

```
 PREFIX table:
<http://www.daml.org/2003/01/periodictable/PeriodicTable#>
SELECT ?name ?number ?color
WHERE
{
    ?element table:name ?name.
    ?element table:symbol ?symbol.
    ?element table:atomicNumber ?number.
    OPTIONAL { ?element table:color ?color. }
        FILTER ( !BOUND (?color ) )
}ORDER BY ?number
```

Bunların yanında SPARQL, sonuç kümeleri üzerinde mantıksal *VEYA* işlemini gerçekleştiren *UNION* mantıksal işlecini de desteklemektedir.

2.11. Betimleme Mantığı (Description Logics)

Betimleme mantığı, çeşitli uygulama alanları için kavram tanımlarını yapısal ve anlamsal olarak iyi tanımlanmış biçimde ifade edebilmek için kullanılan bilgi temsil dilleri ailesidir. Önermeler mantığını, ifadesel olarak güçlü işleçleri ile genişletir. Birinci dereceden yüklem mantığını ise daha verimli şekilde karar verilebilir (decidable) problemlere sahip olması özelliği ile genişletir. Şu anki adını 1980'lerde almış olup, daha önce (tarihi sıraya göre) 'terminoloji diller' ve 'kavram dilleri' olarak adlandırılmıştır. İlk betimleme mantığı tabanlı sistemler KL-ONE [33] (1985), LOOM [34] (1987), CLASSIC [35] (1991), son dönemdekiler ise RACER [36] (2001), ve KAON2 [37] (2005)'dur. Uygulama alanları yapay zeka, ontolojiler (anlamsal web) ve tıbbi bilgilerin kodlanmasında yoğun olarak kullanılan biyoinformatiktir.

Betimleme mantığı kavramları, rolleri, olguları ve bunlar arasındaki ilişkileri modeller. İfade yapıları arasındaki ayrım Tbox (Terminological Box) ve Abox (Assertional Box) kümeleri ile sağlanır. Tbox, kavram (concept) ve rol (Role) sıradüzenlerini tanımlayan ifadeleri kapsar. Örneğin, “Her işçi bir insandır” ifadesi, *İşçi* ve *İnsan* kavramları arasında bir sıradüzen tanımladığı için Tbox kümesine aittir. Abox ise olgular hakkında yapılan beyanları (assertions) kapsar. Örneğin, “Ali bir işçidir” ifadesi, *Ali* olgusunun hangi kavram kümesine ait olduğunu belirttiği için Abox kümesine aittir.

Betimleme mantığı, kavramların tanımlanmasını sağlayan mantıksal yapıcıları (logical constructors) destekler. Bunlardan, kavramların kesişimi (intersection) ve birleşimi (union), tümleyen alma (negation), evrensel (universal restriction) ve varoluş (existential restriction) yapıcılarını, birinci dereceden mantıktan alır. Ters alma (inverse of) ve geçişkenlik (transitivity) gibi yapıcılar ise birinci dereceden mantıkta bulunmaz.

ALC, SHIQ, SHOIN, SHIF gibi farklı yetenekte betimleme mantığı standartları tanımlanmıştır. Bunlardan SHOIN, OWL-DL’nin oluşturulmasında, SHIF ise OWL-Lite’in olusturulmasında temel alınmıştır. [38]

Fact++, KAON2, Pellet, RACER betimleme mantığı çıkarsayıcıları olup, tez kapsamında Pellet üzerine geliştirilmiş olan SPARQL-DL REASONER çıkarsayıcısı kullanılmıştır.

3. KAYNAK KOD SORGULAMA İLE İLGİLİ ÖNCEKİ ÇALIŞMALAR

3.1. Sözcük (Lexical) Tabanlı Sorgulama

Anahtar sözcüklerle kod metni ya da metin örüntüsü arama imkanı hemen hemen tüm yazılım geliştirme araçlarında sağlanmıştır. Bu gruptaki en belirgin yaklaşımlar şunlardır:

3.1.1. *Grep* Tabanlı Araçlar

Grep, karakter dizisi bazında, düzgün ifade eşleştirme (regular expression matching) yapmayı sağlayan bir araçtır. Kodun anlaşılabilmesine yönelik geliştirilen yöntemlerden ilk ve hala kullanılanlardan biridir. Bu türdeki araçların güçlü özellikleri aşağıda sıralanmaktadır:

- Programlama dillerinden bağımsız olmalarından dolayı, taşınabilir ve esnektirler. Grep unix tabanlıdır, ancak diğer işletim sistemleri için de gerçekleştirimleri mevcuttur.
- Sorgulama öncesi herhangi bir dizinlemeye gerek duymazlar.
- Atılan sorgu, herhangi bir ön işleme tabi tutulmaz.

Zayıf sayılabilecek özellikleri ise şunlardır:

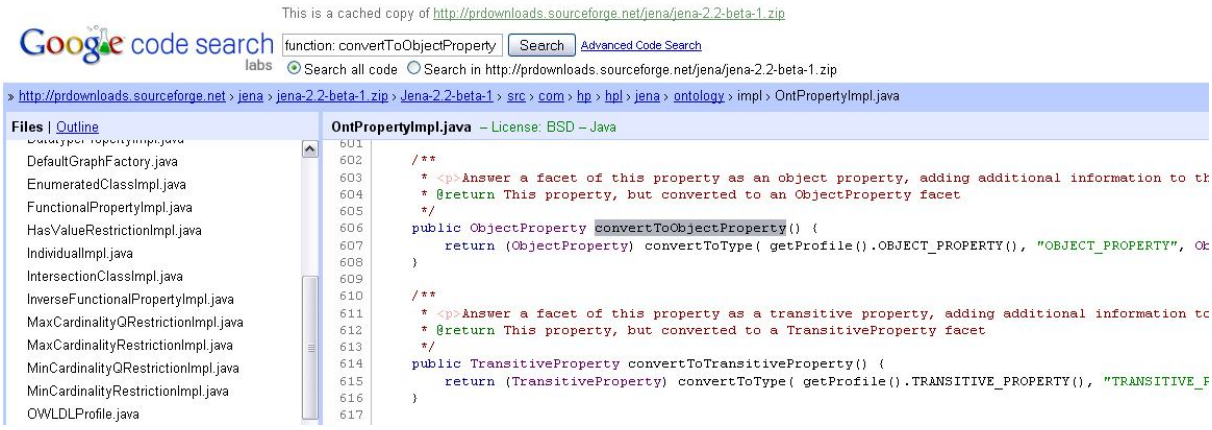
- Tüm girdiye düz metin olarak davranırlar ve bu nedenle de anlamsal sorgu için uygun değildirler. Arama alanına kısıt konulamaz. (belirteç, method, sınıf v.b. gibi)
- Sonuçlar anlamsal bilgiden yoksun olduğu için, dönen bir çok sonucun hangisinin en uygun olduğunun tesbit edilmesi zordur. [39]

3.1.2. Arama Motoru Yaklaşımı

Son yıllarda, internet üzerinde Google Code Search [40], Krugle [41], Koders [42] gibi kaynak kod arama motorları oluşturulmaya başlanmıştır. Bu araçlar, internet üzerinden açık kaynak kodları çekip getiren örümcekler (crawlers) ile desteklenir. Getirilen kod dizinlenir ve bir kod deposunda saklanır. Kullanıcılar, düzgün ifadeleri (regular expressions) kullanarak, belirledikleri dil ve proje bazında sorgulama

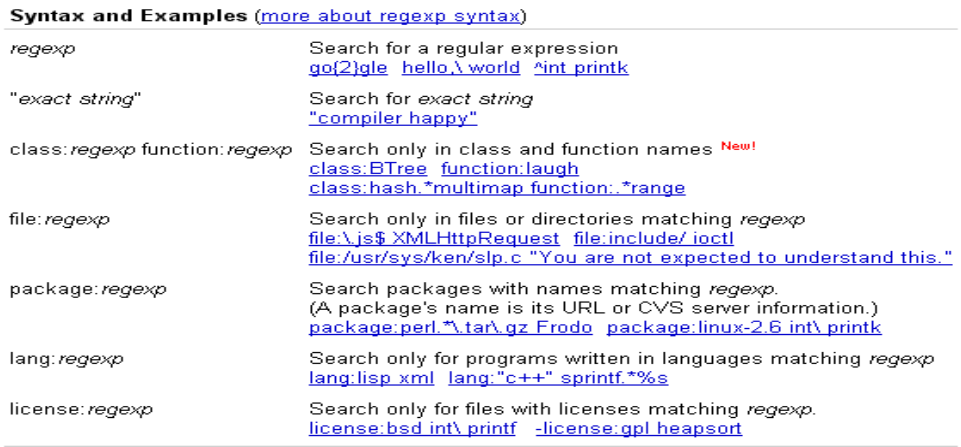
yapabilirler. Belirtilen sorguya en yakın ya da en ilgili (relevant) sonuçlar listelenir. Kaynak kod arama motorları, sözcük tabanlı yaklaşıma sahiptirler. Bir başka deyişle; yazılımın yapısı ve sözdizimine önem vermezler. Buna karşın, yüksek hacimdeki kod verilerini ele alıp ileri düzey bilgi erişim modelleri ve sıralama algoritmaları kullanarak var olan diğer tekniklere üstünlük sağlarlar. Arama motoru yaklaşımı kullanılarak geliştirilmiş bazı araçlar şunlardır:

3.1.2.1. Google Kod Arama Sayfası



Şekil 3.1. Google kod arama web sayfası [40].

Şekil 3.1'de, Google kod arama sayfasında yapılan örnek bir sorgulama gösterilmiştir. *convertToObjectProperty* adındaki metod sorgulanmış ve getirilen sonuçlar arasından istenilen seçildiğinde şekildeki gibi kod üzerinde gösterilmiştir.



Şekil 3.2. Google kod arama web sayfasında düzenli ifade kullanım kılavuzu [40].

Google kod arama sayfası, düzgün ifadeler ile sorgulamaya imkan verir ve bu doğrultuda kullanıcılara Şekil 3.2'deki gibi bir düzgün ifade kullanım kılavuzu sunar.

Google kod arama sayfası ve benzer şekilde çalışan internet üzerindeki diğer kod arama sayfaları, projelerdeki kod elemenleri ile ilgili anlamsal özellikleri ve bu elementlerin, birbirileri arasındaki ilişkileri dikkate almazlar. Örnek olarak, bu sayfalarda, herhangi bir türdeki parametreye sahip, belirtilen isimdeki metoda sahip olan sınıf sorgulanamaz. Çünkü sunulan sorgulama, saklama ve erişim mekanizması buna imkan vermemektedir.

3.1.2.2 Jsearch

Jsearch [43] kaynak kod aramada, bilgi erişim dizgesi kullanımının bazı ek özelliklerle geliştirildiği bir kaynak kod sorgulama aracıdır. Lucene [44] üzerinde geliştirilmiş olup, büyük miktardaki kaynak kod kütüğünün işlenip sorgulanabilmesine imkan verir. Aracın kullanıcı tarafı bir Eclipse eklentisi, sunucu tarafı ise kod deposunda dizinler oluşturan ve bu dizinler üzerinden aramaya imkan veren bir bilgi erişim sistemidir. Çalışma şekli aşağıda listelenmiştir:

- Kaynak kodlar, kod deposuna geliştiriciler tarafından yüklenir.
- Kaynak kodların ön işleme sokularak, kod elementleri arasındaki temel ilişkiler tesbit edilir.
- Ön işlemde tesbit edilen kod elementleri ve aralarındaki ilişkiler kullanılarak, dizinleme yapılır. Dizinlemede kullanılan elemanlardan bazıları: sınıf ismi, kalıtılan sınıf ismi, metod isimleri ve dönüş türleridir.

Gerçekleştirilen ön işlem ve dizinleme aşamalarında, kod elementleri ile ilgili bazı yapısal özelliklerin çıkartılması, kod üzerinde esnek sayılabilecek sorguların oluşturulabilmesine imkan tanır. Örnek olarak, "*Paint* adında bir metod içeren ve sınıf adı olarak *Color*'ı içermeyen kod kütükleri" gibi bir sorgu oluşturulabilir. Gelişmiş sayılabilecek bu sorgulama yeteneği, Jsearch ve diğer bilgi erişim dizgesi tabanlı kod sorgulama araçlarının, web tabanlı Google kod arama sayfası gibi sadece düzenli ifadeler ile çok kısıtlı sorgulamaya imkan veren araçlara sağlandığı bir üstünlüktür.

Çizelge 3.1. Jsearch ile desteklenen bazı sorgu örnekleri [43].

Sorgu İfadesi	**Tanımı Karşılayan Java Sınıfı**
extends:Jdialog code:Jtable	"JDialog" sınıfını kalıtan ve kod içerisinde "Jtable" sınıfını kullanan
code:Document +import:com.w3c.*	Kod içerisinde "Document" bulunan ve "com.w3c" import belirtimine sahip
parameter:JTable	"Jtable" adındaki parametreye sahip metodu olan
parameter:Jgraph code:cell	"Jgraph" parametresine sahip metoda sahip ve kod içerisinde "cell" adını kullanan
method:paint -class:Color	"Paint" adında bir metod içeren fakat "Color" adında sınıfı olmayan
method:paint +parameter:Graphics	"paint" adında bir metod içeren ve parametrelerinden bir tanesinin adı "Graphics" olan

Çizelde 3.1'de, Jquery ile desteklenen bazı sorgu örnekleri görülmektedir. Araç, her ne kadar kod elementlerinin arasındaki temel kod ilişkilerinin tesbit edilmesi ve sorgulanabilmesine imkan verse de, çözümlemenin metin tabanlı olarak yapılması, ileri düzey ilişkilerin tesbit edilmesini engeller. Örneğin, herhangi bir sınıf veri üyesine erişim yapan (yazma ya da okuma) tüm metodların sorgulanabilmesi, bu kategorideki araçlarda mümkün değildir. Çünkü, belirtilen şekildeki ilişkiler ancak kaynak kodun metod hatta blok seviyesinde çözümlenmesini gerektirir ki; bu da ancak kodun soyut söz dizim ağacı gibi herhangi bir ara model kullanılarak tesbit edilebilir.

3.1.3. Sözcük Tabalı Sorgulama Yaklaşımının Değerlendirilmesi

Sözcük tabanlı sorgulama araçlarındaki ortak özellik, kod kütüklerinin metin olarak işlenmesi ve sorgulanmasıdır.

Grep benzeri araçlar, programlama dillerinden bağımsızdırlar ve kod yapısı hakkında herhangi bir bilgiye sahip değildirler. Dolayısı ile anlamsal özellikleri yoktur.

Google kod arama sayfası benzeri web tabanlı araçlar, gelişmiş bilgi erişim modelleri üzerine kurulmuş olmaları ve ileri seviyedeki sıralama algoritmaları ile

performans açısından diğer yaklaşımlara üstünlük sağlarlar. Ancak, düzgün ifadeler ile kısıtlı sorgulamaya imkan vermeleri, kod yapısını ve kod elementleri arasındaki ilişkileri ihmal etmeleri bu araçların anlamsallıktan uzak olmalarını sağlar.

Jsearch benzeri araçlar, herhangi bir bilgi erişim modeli üzerine kurulmuş olup, yapısal kod özelliklerini de işleyebilmektedirler. Bu özellik kullanıcılara, kod elementleri arasındaki ilişkiler bazında sorgulama olanağı sağlar. Ancak ilişkisel özellikler kod metinleri üzerinden çekildiği için, gelişmiş sorgulama yapılamaz.

Sonuç olarak sözcük tabanlı kod arama araçları sadece erişim amaçlı olup, ileri düzey kod analizlerinde kullanımları uygun gözükmemektedir.

3.2. Sözdizimsel (Syntactic) Sorgulama

Karmaşık kaynak kod sorguları, sözcük tabanlı sorgulama yaklaşımı ile ifade edilemez. Karmaşık sorgulamalara;

- kod elementlerinin birbirilerine olan bağımlılıklarını incelemek amacıyla yapılan sorgulamalar
- kod hakkındaki ileri düzey tasarım bilgilerinin elde edilmesine yönelik yapılan sorgulamalar

genel örnekler olarak gösterilebilir. Bu türdeki sorgulamaların yapılabilmesi için, kaynak kodun sözdizimsel olarak modellenmesi ve bu model üzerinde çalışabilecek sorgulama dilinin belirlenmesi gerekir.

Bu türdeki kod sorgulama sistemleri aşağıda belirtilen ortak yapısal özelliklere sahiplerdir: [39]

- kod bilgilerinin saklandığı bir depo (repository)
- depoyu, kod bilgileri ile dolduran (populate), dil ayrıştırıcıları ya da durağan/dinamik bir çözümleyici
- kullanıcıların, sorgularını hazırlayıp çalıştıracağı ve sonuçları görebileceği bir arayüz

- ilgili sonuçlara erişilmesini sağlayan sorgu işleyicisi

Belirtilen özelliklere sahip sistemler, kullandıkları kaynak modeli ve sundukları sorgulama dili ile birbirilerinden ayrılırlar.

3.2.1. İlişkisel Yaklaşımlar

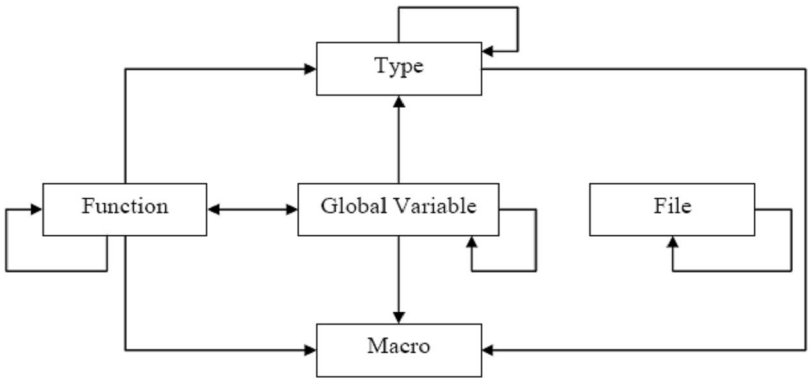
Kaynak kodun temsilinde ilişkisel modelleri kullanırlar. Örnek çalışmalar şunlardır:

3.2.1.1. Omega

İlgili en eski çalışmalardan olan Linton'un OMEGA [45] sisteminde, program bilgileri INGRES [46] adındaki veritabanı sisteminde tutulmaktadır. Sistem, kaynak kod hakkındaki elli sekiz farklı ilişkiyi ve bu ilişkilerle ifade edilebilecek ayrıntılı bilgileri saklayabilmektedir. Sorgulamalar ise SQL benzeri bir dil olan QUEL [47] sorgulama dili kullanılarak gerçekleştirilebilmektedir. QUEL'de özyinelemeli sorgulama yapılamaz. Bu özellik, çağrı çizgeleri (call graphs) gibi önemli kod çözümlemeleri için önemli bir eksikliktir.

3.2.1.2. CIA

Bu kategorideki sonraki çalışma olan CIA [48] ise, OMEGA'yı performans yönünden geliştirmiştir. CIA'da veritabanında kullanılan ilişki sayısı azaltılmış, böylece sorgu sonuçlarının üretilmesindeki performans arttırılmıştır. CIA, C dilinde yazılmış programlar için tasarlanan kavramsal modeli kullanarak, kod bilgilerini çıkartır ve bu bilgileri veritabanında saklar. Sorgulama, QUEL ile yapılmaktadır.



Şekil 3.3. CIA kavramsal modeli [39].

Şekil 3.3'te CIA sistemi için tasarlanmış kod kavramsal modeli görülmektedir. Modeli oluşturan beş programlama dili nesnesinden biri olan *Function* için şu özellikler tanımlanmıştır; tanımlandığı kütük ismi, dönüş değerinin türü, fonksiyonun ismi, *static* özellikte olup olmadığı bilgisi, başlangıç ve bitiş satır numaraları.

Bu iki çalışma, günümüz kaynak kod sorgulama araçlarından tüm beklentileri karşılamaktan uzak olsalar da, yazılım endüstrisinde program bilgilerinin veritabanında saklanması akımına öncülük etmişlerdir.

Kaynak kodun ilişkisel modeller kullanılarak temsil edilmesi ve bu modellerin SQL türevi dillerle sorgulanması yönteminin eksikleri şunlardır:

- İlişkisel yöntemlerde, karmaşık kaynak kod yapısınının modellenmesi zordur. Bunun sebebi ilişkisel yöntemlerde, kavram ve ilişkilerin sıradüzensel biçimde ifade edilememesidir. Örneğin, bir A türünün diğer bir B türünün alt türü olduğunu belirten bir özellik, bu yöntemlerde tanımlanamaz.

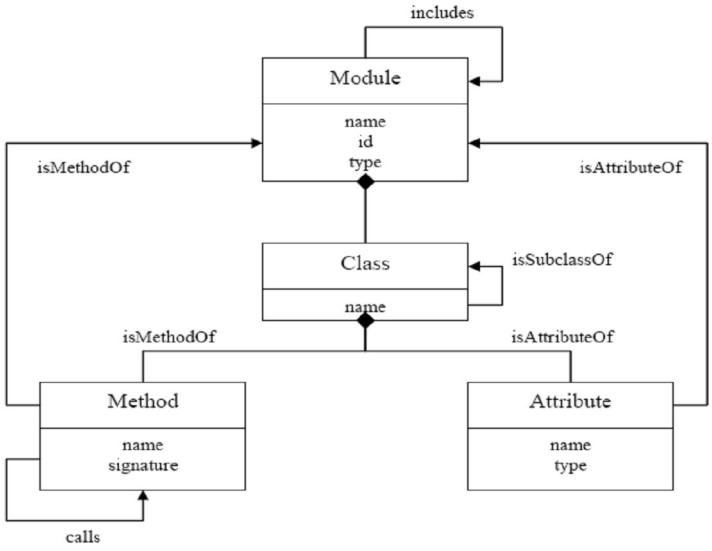
- Kapalı geçişkenliğin (transitive closure) ifade edilmesinde, SQL türevi dillerin yetenekleri kısıtlıdır.

3.2.2. Çizge Tabanlı Yaklaşımlar

Çizge tabanlı yaklaşımlar, kaynak kodu çizge yapıları şeklinde temsil ederler ve sorgulama için GreQL (Graph Repository Query Language) [49] kullanırlar. Örnek çalışmalar:

3.2.2.1. GUPRO

Kaynak kod, çizge yapısında olacak şekilde ayrıştırılmıştır.



Şekil 3.4. GUPRO kavramsal modeli [50].

Şekil 3.4'te görüldüğü gibi, kaynak kodun dört temel nesnesi olan modül, sınıf, metod ve nitelik model içerisinde tanımlanmıştır. Nesneler arasındaki ilişkiler ve nesne kısıtları da model üzerinde yer almaktadır. Gupro'da, çizge yapısındaki kod modelini sorgulamak için GreQL [51] kullanılır. GreQL, çizge yapısı üzerindeki varlıkları ve bu varlıklar arasındaki ilişkileri analiz edebilmek için güçlü bir yaklaşım sunan, bildirimsel (declarative) bir sorgulama dilidir. Aşağıda, GreQL ile tanımlanmış, sınıf ve bu sınıfın en üst sınıfının isimlerini sorgulayan bir sorgu örneği görülmektedir.

```
FROM c, super : V{Class}
WITH c --> {isSubclassOf}* super
AND
outDegree {isSubclassOf} (super = 0)
REPORT c.name, super.name
END
```

Çizge tabanlı sorgulama dilleri bazı özellikleri ile ilişkisel sorgulama dillerine üstünlük sağlarlar. Bu üstünlükler kısaca şu biçimde listelenebilir:

- GreQL, kapalı geçişkenlik özelliğinin etkin ve kolay biçimde kullanılmasına imkan verir.

- GreQL, doğrudan olarak çizge üzerinde gezinme yöntemini kullanırken, SQL türevi diller, farklı tabloları bir çok kez join eder, yani bağlamak durumundadır.

Çizge tabanlı sorgulama dillerinin eksikliği; ilişkisel sorgulama dillerinde oldugu gibi, kavram ve ilişkilerin sıradüzensel özelliklerinin tesbit edilememesi ve yorumlanamamasıdır.

3.3. Mantıksal Programlama Dili Yaklaşımı

Mantıksal programlama dillerinin kullanımı, karmaşık kaynak kod sorgularını ifadesel olarak kolaylaştırmıştır. Bu yaklaşımın sağladığı en önemli özelliklerden biri, kapalı geçişkenlik (transitive closure) özelliği ile özyinelemeli sorgulamaya imkan vermesidir.

Bu alanda yapılan ilk çalışmalar XL C++ Browser [52] ve ASTLog [53] olarak belirtilebilir. XL C++ Browser, kaynak kod sorgulamada mantıksal programlama dili kullanımının gerçekleştirildiği ilk çalışmalardan biri olmuştur. Uygulama bir Prolog sisteminin üzerine kurulmuş, kaynak kod hakkındaki tüm olgular (facts) ana bellekte tutulmuştur. ASTLog, tanımlanan sorguların C++ soyut sözdizim ağaçları üzerinde çalıştığı, diğer bir çalışmadır. Hızlı sonuç üretme özelliğinden dolayı, Microsoft tarafından bir çok projede kullanılmıştır. Yine aynı alana giren çalışmalar tez ile ilgileri ve yönlendirici olmaları nedeniyle ayrı başlıklar halinde incelenmiştir.

3.3.1 Jquery

Jquery [54], Eclipse geliştirme ortamına eklenti olarak geliştirilmiş bir kod sorgulama ve gezinme aracıdır. Kod bilgilerinin oluşturulmasında ve sorgulanmasında Prolog benzeri bir mantıksal programlama dili olan TyRuBa [55] kullanılmıştır. Çalışmada Prolog yerine, türevi olan Tyruba'nın kullanılmasının nedeni, Prolog'da yazılan sorguların sonlanamaması durumuna, sorguların işlenmesi sırasında *tabled resolution* adındaki farklı bir yöntem kullanarak çözüm bulunmuş olmasıdır. Araç, gelişmiş bir kullanıcı arayüzüne sahiptir ve doğrudan sorgulamaya da imkan vermektedir.

Jquery'de, kod bilgilerinin saklanmasında ve sorguların ifade edilmesinde iki tür yüklem (predicate) kullanılır.

1. **Çekirdek Olgu Yüklemleri (Core Fact predicates):** Bu türdeki yüklemler, doğrudan olguların saklandığı bilgi tabanında (fact base) çalışırlar. Üç türe sahiptir:

    - **Unary predicates:** Tekli yüklemler.

        method(?X): "x bir metottur"

        field(?X) : "x bir sınıf alanıdır" gibi.

    - **Binary predicates:** İkili yüklemler.

        implements(?C,?I): "C sınıfı, I arayüzünü gerçekleştirir"

        modifier(?E,?S): "E kod elementi, S erişim belirtecine sahiptir" gibi.

    - **Ternary predicates:** Üçlü yüklemler.

        writes(?B,?F,?L): "B bloğu, F sınıf alanını, L konumunda yazar" gibi.

2. **Türemiş Yüklemler (Derived predicates):** Bu türdeki yüklemler, temel yüklemlerden, önceden tanımlanmış kurallar kullanılarak türetilmesi ile oluşturulur. Bu yüklemler, doğrudan bilgi tabanında çalışmazlar. Bunun yerine, meydana geldikleri temel yüklemlerin veriler üzerinde çalışması ve elde edilen sonuçların, tanımlanan kurallar yardımı ile işletilmesi ile istenilen sonuçlar elde edilir. Aşağıda örnek bir kural görülmektedir.

    staticMethod(?M) :- method(?M),modifier(?M,static)

    Kuralın anlamı şu şekildedir: "M bir metot" ve "M durağan (static)" ise, "M durağan bir metottur". Tanımlanan bu yüklem, static metodları sorgulamak için kullanılacaktır.

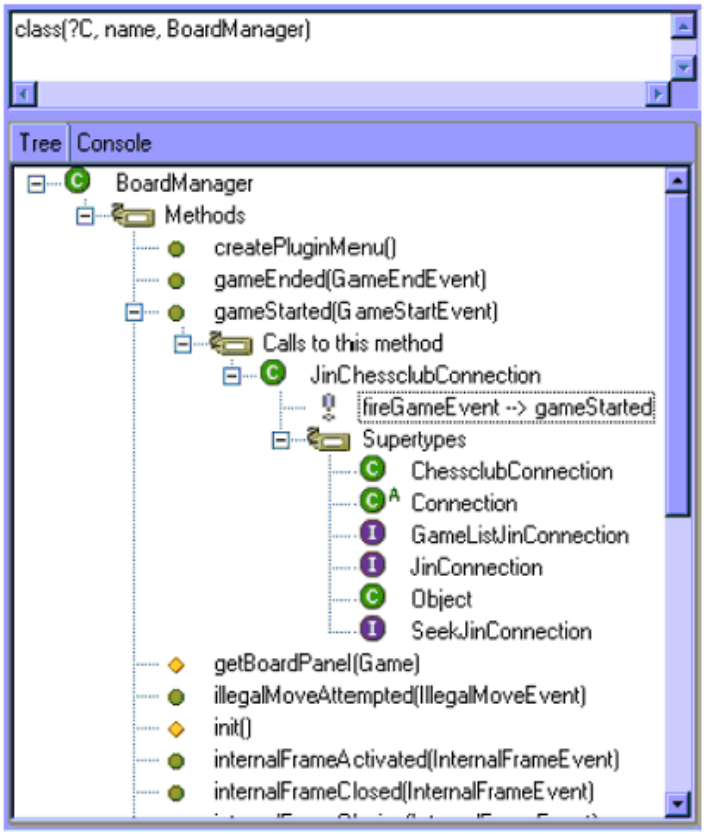
Jquery'de, bilgi tabanı gerçek zamanlı olarak güncellenebilmektedir. Yani, üzerinde sorgulama yapılacak olan veriler (kod bilgi tabanı), geliştirme ortamında, kod üzerinde gerçekleşen değişikliklere paralel olarak güncellenir.

Eclipse geliştirme ortamı, gerçekleşen her kod değişimi sırasında, projenin yeniden derlenmesini sağlayan otomotik derleme imkanına sahiptir. Eclipse gerçekleşen değişim sonrasında, olabildiğince az sayıdaki kaynak kütüğü yeniden derleyerek projenin tutarlı halde kalmasını sağlar. Jquery, Eclipse'in kod değişimi sırasındaki bu uyarı mekanizmasından yararlanarak, gerçekleşen kod değişimi doğrultusunda bilgi tabanının yeniden güncellenmesini, dolayısı ile de yapılacak herhangi bir sorguda güncel kod bilgilerine ulaşabilmeyi sağlarlar. Değişimden etkilenen tüm java kütüklerine ait olgular bilgi tabanından silinir, kütükler tekrar ayrıştırılır ve güncel kod bilgileri yeniden bilgi tabanına eklenir.

Gerçek zamanlı olarak kod bilgi tabanının güncellenmesi özelliği, tez kapsamında ele alınmamıştır. Belirtilen özellik ayrı bir çalışma konusu durumundadır.

Aşağıda, *BoardManager* adındaki sınıfa erişmek için oluşturulan bir sorgu örneği, Şekil 3.5'te ise bu sorgunun araç üzerindeki sonucu görülmektedir.

class(?C, name, BoardManager)



Şekil 3.5. Jquery, sorgu örneği [54].

3.3.2 CodeQuest

CodeQuest'in [56] gerçekleştiriminde, kod bilgilerini saklamak için bir ilişkisel veritabanı sistemi, sorgulama için ise Prolog'un bir alt türevi olan Datalog kullanılmıştır. CodeQuest, ölçeklenebilirlik ve esnek sorgulama açısından önemli bir çalışmadır.

Datalog, Prolog'un alt türevi olmasına karşın, sorguların işlenmesinde kullanılan strateji farklıdır. Bunun sonucu olarak, Prolog'ta sorguların sonlanamaması durumunu ortadan kaldırmak için kullanılan mantıksal ek notlara (annotations) ihtiyaç duyulmaz. Ayrıca Datalog, Prolog'taki liste veriyapısına sahip değildir. Jquery'den farklı olarak, mantıksal programlama dilini sadece sorgulamalarda kullanır.

Sorguların ifade edilmesinde Jquery'de olduğu gibi yüklemler (predicates) kullanılır. İki farklı tür yüklem yapısı vardır:

1. **Temel Yüklemler (Primitive Predicates):** Bu türdeki yüklemler, doğrudan veri tabanında saklanan veriler üzerinde çalışırlar. İki türe sahiptir:

    - **Kod elemlarını tanımayan yüklemler:**

        interfaces(X,Name): "X, 'Name' adındaki bir arayüzdür"

        modifiers(M,Name): "X, 'Name' adındaki bir erişim belirtecidir. (public, private, static ...)" gibi.

    - **Kod Elemanları Arasındaki İlişkileri Tanımlayan Yüklemler*:***

        extends(T1,T2): "T1 sınıfı ya da arayüzü, T2 sınıfı ya da arayüzünü kalıtır"

        writes(B,F): "B bloğu, F sınıf alanına yazar"

        reads(B,F): "B bloğu, F sınıf alanını okur" gibi.

2. **Türemiş Yüklemler (Derived Predicates):** Bu türdeki yüklemler, temel yüklemlerin, kurallar kullanılarak türetilmesi ile oluşturulur. Bu yüklemler, doğrudan veritabanında saklanmış olan veriler üzerinde çalışmazlar. Bunun

yerine, meydana geldikleri temel yüklemlerin veriler üzerinde çalışması ve elde edilen sonuçların, tanımlanan kurallar yardımı ile işletilmesi ile istenilen sonuçlar elde edilir. Örnek olarak:

hasSubtype(T1,T2): "T1, T2'nin birincil (direct) alt türüdür".

hasSubType türetilmiş yükleminin kuralı şu şekilde tanımlanmıştır:

hasSubtype(T, S) :− extends(S,T) ; implements(S,T).

Kuralın anlamı şu şekildedir: "S, T'yi extend eder" ya da "S, T'yi implement eder" ise, "S, T'nin alt türüdür". Tanımlanan bu yüklem, herhangi bir türün alt türlerini sorgulamak için kullanılacaktır.

Diğer bir türemiş yüklem örneği:

accesses(B,F): "B bloğu, F sınıf alanına erişir".

accesses türetilmiş yükleminin kuralı şu şekilde tanımlanmıştır:

access(B, F) :− read(B,F) ; write(B,F).

Kuralın anlamı şu şekildedir: "B bloğu, F sınıf veri üyesine okuma yapar" ya da "B bloğu, F sınıf veri üyesine yazma yapar" ise, "B bloğu, F sınıf veri üyesine erişir". Tanımlanan bu yüklem, herhangi bir blok yapısı için erişilen tüm sınıf veri üyelerini sorgulamak için kullanılacaktır.

Aşağıda, özyinelemeyi – kapalı geçişkenlik (transitive closure) gerektiren bir türetilmiş yüklem tanımlanmıştır:

hasSubtypePlus(T1,T2): "T2, T1'in doğrudan yada dolaylı olarak alt türüdür".

hasSubtypePlus türetilmiş yükleminin kuralı şu şekilde tanımlanmıştır:

hasSubtypePlus(T, S) :− hasSubtype(T, S) ;

hasSubtype(T,MID), hasSubtypePlus(MID, S).

Kuralın anlamı şu şekildedir: “S, T’nin alt türüdür” ya da “MID, T’nin alt türüdür” ve “S, MID’nin doğrudan ya da dolaylı olarak alt türüdür” ise “S, T’nin doğrudan ya da dolaylı olarak alt türüdür”. Tanımlanan bu yüklem, herhangi bir tür ve bu türden dolaylı ya da doğrudan türemiş olan diğer türleri sorgulamak için kullanılacaktır.

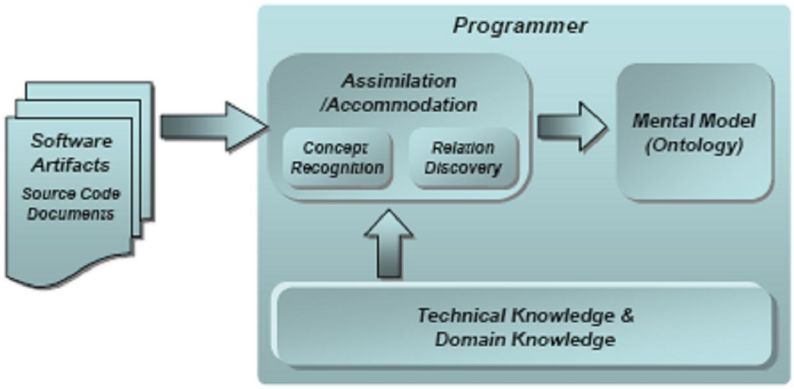
3.4. Ontoloji ve Betimleme Mantığı (Description Logics) Yaklaşımı

Yazılım mühendisliğinde ontoloji kullanımının giderek yaygınlaştığı gözlenmektedir. Bu yaklaşımın temelinde, ontolojinin çıkarım ve etkin sorgulama gibi imkanları sunan bir bilgi gösterim şekli olması yatmaktadır.Literatürdeki çalışmalara bakıldığında, ontolojinin daha çok yazılım süreçlerinin modellenmesinde ve gereksinim analizi aşamalarında alan bilgisini modellemek için kullanılan bir araç olarak kullanıldığı görülür. Bunların yanında, Sugumaran ve diğerleri [57], yazılım bileşeni erişimine anlamsal boyut kazandırmak için yaptıkları çalışmada ontolojiyi, alan terimlerini içeren bir sözlük olarak ve bileşen erişimini gerçekleştirmek için atılan sorguların otomatik olarak düzenlenmesi için kullanmışlardır. Burada belirtilen çalışmalardaki amaçlar, tez kapsamı dışındadırlar.

Literatürde, kaynak kod sorgulamada ontoloji ve betimleme mantığı kullanımını örnekleyen en belirgin çalışmanın Yonggang ZHANG’in doktora çalışması sırasında gerçekleştirilen Sound (An ontology-based Program Comprehension Tool) [39] olduğu görülmüştür.

3.4.1. Sound

Sound, yazılım gereksinim belgeleri ve kaynak kod gibi yazılım öğelerinin DL (Description Logics – Betimleme Mantığı) ontolojisi kullanılarak temsil edilerek, kod analizi, kod ve belgeler arasında izlenebilirlik sağlama gibi üst düzey özelliklere sahip gelişmiş bir yazılım anlama aracıdır. Kaynak kodun, Racer [58] adındaki bilgi temsil sistemi ve bu sistemin sağladığı “ALCQHIR+“ adındaki betimleme dili kullanılarak gösterimi ve yine Racer’in sağladığı nRQL [59] anlamsal sorgulama dili ile kod bilgilerinin sorgulanabilmesi özellikleri ile, tez kapsamında yapılan çalışmaya örnek teşkil etmiş bir çalışmadır.



Şekil 3.6. Sound genel çalışma modeli [39].

Aracın gerçekleştirim amacı genel olarak, yazılım geliştiricilerin, farklı öğrenme stratejileri kullanarak, sunulan bir arayüz ile kod ontolojisini kullanmaları ve elde ettikleri bilgilerin birikimsel olarak ontoloji içerisine eklenmesi olarak özetlenebilir.

Bu tez kapsamında bilgi gösterim dili olarak OWL-DL kullanılmıştır. OWL, W3C tarafından desteklenen web üzerinden bilgi paylaşımı amacıyla kullanılabilen ve anlamsal web kavramının temeli sayılabilecek bir gösterim dilidir. Kullanım amacı olarak, sürekli gelişmesi, SPARQL gibi güçlü bir sorgulama dili sunması, birçok çıkarım motoru tarafından desteklenebilmesi ve bilgiye paylaşılabilme özelliği katması gösterilebilir.

4. ONTOLOJİ TABANLI JAVA KAYNAK KODU SORGULAMA ARACININ (CODONTO) GERÇEKLEŞTİRİMİ

4.1. CODONTO Çözüm Modeli

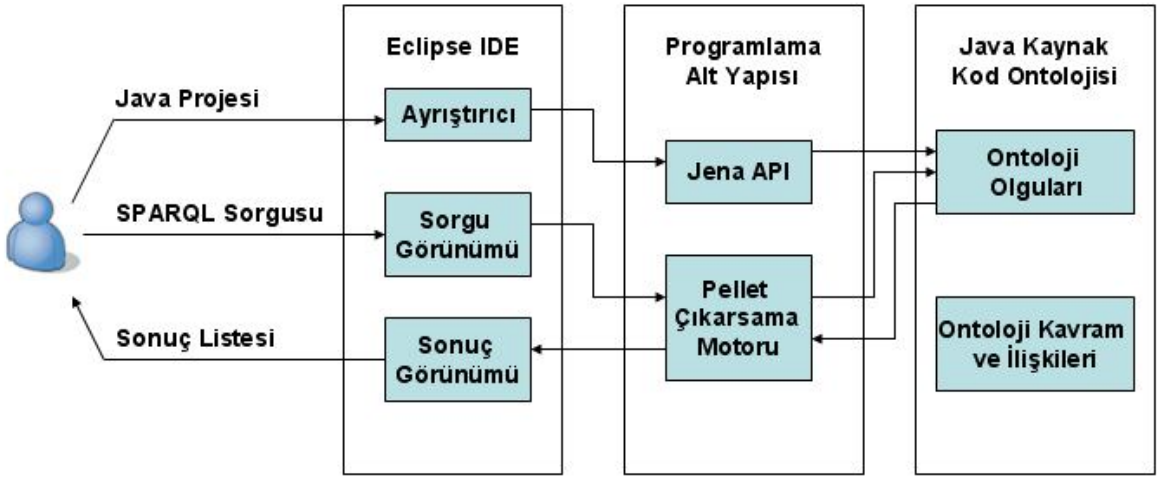
Bu bölümde, tez kapsamında geliştirilen, CODONTO olarak adlandırılan aracın geliştirilme aşamaları ayrıntılı biçimde yer almaktadır.

Araç geliştirme süresince sırasıyla;

- Java ile hazırlanmış kaynak kodlar için bir ontoloji tasarlanmış,
- ilgilenilen Java projesi için otomatik olarak ontoloji olgularını oluşturabilen bir ayrıştırıcı geliştirilmiş,
- kullanıcının, projeyi sorgulayabilmesi için geliştirme ortamına bir sorgu görünümü ve bu sorguların işletilmesi sonucunda elde edilen sonuçların gösterildiği bir sonuç görünümü tasarlanmış,
- son olarak, sorgu oluşturma işlemini kolaylaştırmak ve hızlandırmak için bir Java SWT arayüz uygulaması geliştirilmiştir.

Şekil 4.1'de geliştirilen sistemin çözüm modeli yer almaktadır. Şekil 4.1'de yer alan ok işaretlerinden anlaşılacağı üzere, kullanıcı Eclipse geliştirme ortamına yüklenmiş olan Java projesinin ayrıştırılması işlemini başlatır. Ayrıştırıcı, Jena programlama kütüphanesini [60] kullanarak kod öğelerini temsil eden ontoloji olgularını oluşturur.

Kullanıcı tarafından Eclipse ortamındaki sorgu görünümüne yazılan sorgular, araç tarafından işlenerek ve Pellet çıkarsama motoru kullanılarak yine Eclipse ortamındaki sonuç görünümünde görüntülenir.



Şekil 4.1. Sistemin Çözüm Modeli.

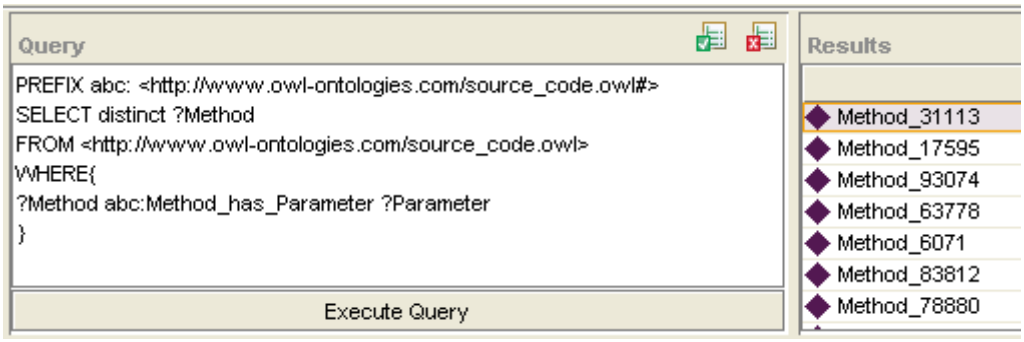
4.2. Kod Bilgi Tabanının Oluşturulmasında Kullanılan Java Kaynak Kod Ontolojisinin Tasarlanması

4.2.1. Protege Ontoloji Editorü ve Pellet Çıkarsama Motoru

Tez kapsamında, Java dil belirtimi 3 [61] (JLS 3) referans alınarak, kaynak kod kavram ve ilişkilerinin temsil edildiği üst model olarak bir OWL ontolojisi tasarlanmışıtır. Tasarım aşamasında ve java kod ayrıştırıcısının ürettiği sonuçların incelenmesinde, ontoloji editörü olarak Protege 3.4 [62] kullanılmıştır. Protege, ontolojileri kullanan bilgi tabanlı uygulamaları ve alan modellerini oluşturmak için kullanılabilen, açık kaynak kodlu bir yazılımdır. Protege,

- ontoloji sınıflarını ve bunlar arasındaki alt sınıf-üst sınıf ilişkilerini tanımlamak için kullanılan sınıf listeleyicisi ve sınıf düzenleyicisi,
- ontoloji sınıflarının olgularını oluşturmak ve bu olguların veriler ve diğer olgular arasındaki ilişkileri tanımlamak için kullanılan olgu listeleyicisi ve olgu düzenleyicisi,
- ontoloji veri türü (data type) ve nesne türü (object type) ilişkilerini (diğer bir ifade ile niteliklerini) ve bunlar arasındaki alt-üst ilişkilerini tanımlamak için kullanılan ilişki listeleyicisi ve ilişki düzenleyicisi

gibi bileşenleri ile kullanıcılara güçlü bir arayüz sunar. Bunların yanında, görüntülenen ontoloji olguları üzerinde SPARQL kullanarak sorgulama yapabilmeye imkan veren bir sorgulama görünümüne de sahiptir.



Şekil 4.2. Protege 3.4'te sorgulama görünümü, örnek sorgu ve sonuçlar.

Protege 3.4, yapılan SPARQL sorgulamalarında ve ontoloji üzerinde yapılabilecek diğer işlemlerde çıkarsamaları gerçekleştirebilmek için Pellet çıkarsayıcısını sunar. Pellet, ontoloji tutarlılık denetimi, otomatik sınıflandırma (kavramlar arasındaki sıradüzensel yapının otomatik oluşturulması) ve çıkarsanan (inferred) yani beyan edilemeyen (not asserted) olguların belirlenmesinde kullanılabilir.

Tez kapsamında geliştirilen yazılımda da sorgulama ve çıkarsama arayüzü olarak Pellet çıkarsama motoru kullanılmıştır. Protege 3.4'ün sunduğu Pellet 1.5.1 bazı önemli çıkarsama yeteneklerine sahip değildir. Örnek olarak; geçişken (transitive) bir ilişki kullanılarak yapılan sorgulama için elde edilen sonuçlarda bu özelliğin kendisini göstermediği görülür. Bu nedenle, tez kapsamında çıkarsama motoru olarak Pellet 2.1 tercih edilmiştir.

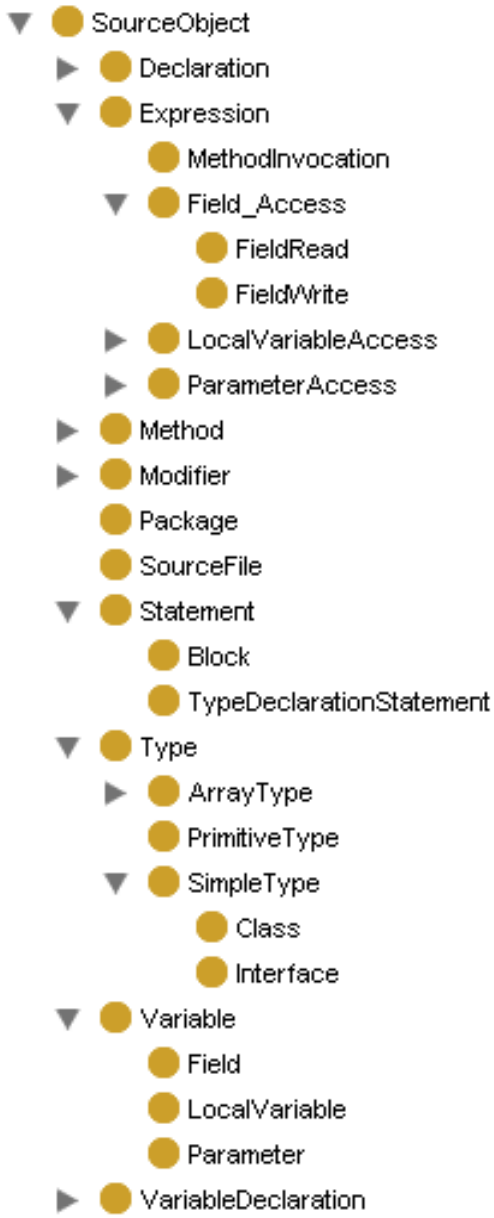
4.2.2. Java Kaynak Kod Ontolojisi

Tasarlanan Java kaynak kod ontolojisi iki kesim halinde incelenebilir. İlki sınıf, metod, değişen gibi genel nesne yönelimli programlama dilleri kavramlarının yanında Java programlama diline özel olan paket, arayüz gibi kavramların da modellendiği *kavramlar* kesimidir. İkincisi ise, tanımlanan kavramlar arasındaki ilişkilerin temsil edildiği *ilişkiler* kesimidir.

4.2.2.1. Kavramlar (Concepts)

Java kod ontolojisi, herbiri OWL sınıfı olarak tanımlanmış kavramlardan oluşmaktadır. Bazı kavramlar doğrudan bir java kod öğesine karşılık gelir (*SourceFile*, *Modifier*, *Method* gibi). Bazı kavramlar ise doğrudan kod içerisindeki herhangi bir öğeye karşılık gelmeyip, birden fazla öğenin bir araya gelerek meydana getirdiği bir oluşumu temsil eder (*MethodInvocation*, *FieldAccess* gibi). Bazı kavramlar ise sadece anlamsal gruplayıcı özelliği taşırlar (*SourceObject*, *Statement* gibi).

Ontolojiyi oluşturan kavramlar Şekil 4.3'te görüldüğü gibi anlamsal özelliklerine göre sıradüzensel bir yapıda tanımlanmıştır. Kavramlar içerisindeki *SourceObject* diğer tüm kavramları kapsayıcı durumundadır. Yani, ontoloji içerisindeki herbir olgu özel bir kavram sınıfına ait olduğu gibi, aynı zamanda *SourceObject* sınıfına da aittir.



Şekil 4.3. Java Kaynak Kod Ontolojisi Kavramlar Görünümü.

SourceObject sınıfının *Declaration*, *Expression*, *Method*, *Modifier*, *Type*, *Variable* gibi alt sınıfları tanımlanmıştır. Bu sınıflardan *Expression*, *Type*, *Modifier* gibi sınıflar da kendi alt sınıflarına sahiptir. Örneğin *Type* sınıfının alt sınıfları; *ArrayType* (Dizi Türü), *PrimitiveType* (Temel Tür), ve *SimpleType*'tır (Basit Tür). Benzer şekilde, *SimpleType* sınıfı da *Class* ve *Interface* alt sınıflarına sahiptir. Bu sıradüzensel yapı içerisinde herhangi bir Java sınıf olgusu *Class*, dolayısı ile *SimpleType*, dolayısı ile *Type*, dolayısı ile de *SourceObject* ontoloji kavramlarının bir olgusu durumundadır.

Proje kapsamındaki tüm basit türlerin listelenmesi sağlayan sorgu aşağıda gösterilmiştir.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT ?ST
WHERE {
?ST <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:SimpleType.
}
```

Basit türleri temsil eden *SimpleType* kavramı, *Class* (Java sınıfı) ve *Interface* (Java arayüzü) kavramlarının üst kavramıdır. Bu nedenle belirtilen sorgunun işletilmesi sonucunda tüm sınıf ve arayüzler listelenecektir.

Herhangi bir Java projesi ontoloji olgusunun otomatik olarak oluşturulması sırasında, herhangi bir Java sınıf veya arayüzünün *SimpleType* türünde bir olgu olduğu beyan edilmez. Beyan edilen bilgiler, sınıf veya arayüzün *Class* yada *Interface* türünde olduğu ve bu türlerin de aşağıdaki OWL kesiminde de görüldüğü gibi, *SimpleType* kavramının alt kavramı olduğu bilgileridir.

```
<owl:Class rdf:ID="Class">
   <rdfs:subClassOf rdf:resource="#SimpleType"/>
</owl:Class>
<owl:Class rdf:ID="Interface">
   <rdfs:subClassOf rdf:resource="#SimpleType"/>
</owl:Class>
```

Kavramlar ve kavramlar arasındaki ilişkiler kullanılarak, beyan edilmeyen bilgilerin çıkarsanması, çıkarsayıcı kullanımı ile mümkündür.

Ontolojiyi oluşturan kavramlardan bazıları ve önemli özellikleri:

SourceObject: Diğer tüm kavramların üst kavramıdır.

Declaration: Java kodu içerisindeki tüm bildirimleri temsil eden olguları içeren kavramdır. *TypeDeclaration*, *MethodDeclaration* gibi alt kavramlara sahiptir. Bu

türdeki kavramlar sorgu amaçlı olmayıp daha çok kodun ayrıştırılması sırasında kullanılmak üzere tanımlanmışlardır.

TypeDeclaration: Java sınıf ve arayüzleri bildirimlerini kapsayan genel bir belirtimdir. Herbir .java kütüğü içerisinde bir yada daha fazla tür bildirimi vardır.

Expression: Method çağrımı (*MethodInvocation*) ve sınıf alanı erişimi (*FieldAccess*) gibi java ifadelerini kapsayan bir üst kavramdır.

MethodInvocation: Kod içerisindeki metod çağrımlarını temsil eden olguları içeren kavramdır. *MethodInvocation* kavramı ile Method kavramı arasında, çağrımı yapılan metod olguları ile ilişkilendirme yapabilmek için *MethodInvocation_calles_Method* ilişkisi tanımlanmıştır. Bu ilişkiden ilerleyen kesimlerde bahsedilecektir. Örnek olarak,

```
buf.append(trim(child.getTreeItem().getText()));
```

şeklinde belirtilen Java kod örneği, projenin ontoloji olgusu içerisinde bir *MethodInvocation* olgusu olarak temsil edilecektir.

FieldAccess: Kod içerisindeki sınıf alanı erişimlerini temsil eden olguları içeren kavramdır. *FieldRead* ve *FieldWrite* alt kavramları tanımlanmıştır.

FieldWrite: Kod içerisindeki sınıf alanlarına yazma durumlarını temsil eden olguları içeren kavramdır.

```
fCurrentInputKind = ASTInputKindAction.USE_PARSER;
```

şeklinde belirtilen Java kod örneği, bir *FieldWrite* olgusu olarak temsil edilecektir. Örnekte belirtilen ‘fCurrentInputKind’ ait olduğu sınıfın bir alanıdır.

FieldRead: Kod içerisindeki sınıf alanlarını okuma durumlarını temsil eden olguları içeren kavramdır.

```
if (getCurrentInputKind() == ASTInputKindAction.USE_RECONCILE);
```

şeklinde belirtilen Java kod örneğindeki 'ASTInputKindAction.*USE_RECONCILE'* kesimi, bir *FieldWrite* olgusu olarak temsil edilecektir. Örnekte belirtilen '*USE_RECONCILE*' ait olduğu sınıfın *static* erişim belirtecine sahip bir alanıdır.

LocalVariableAccess: Kod içerisindeki yerel değişken erişimlerini temsil eden olguları içeren kavramdır. *LocalVariableRead* ve *LocalVariableWrite* alt kavramları tanımlanmıştır.

ParameterAccess: Kod içerisindeki metod parametre erişimlerini temsil eden olguları içeren kavramdır. *ParameterRead* ve *ParameterWrite* alt kavramları tanımlanmıştır.

Method: Basit Java türleri (sınıf ve arayüz) içerisinde tanımlanmış metodları temsil eden olguları içeren kavramdır. Parametre olguları ile ilişkilendirilme için *Method_has_Parameter*, metod gövdesi içerisindeki bloklar ile ilişkilendirilme için *Method_body_Block*, metodun dönüş türünü ifade eden tür olguları ile ilişkilendirme için *Method_returnType_Type*, metodun erişim seviyesini ifade eden erişim belirteci olguları ile ilişkilendirme için *Method_has_Modifier* ilişkileri tanımlanmıştır.

Modifier: Erişim belirteçlerini temsil eden olguları içeren kavramdır. Herbir Java erişim belirtecine karşılık gelecek şekilde, *Modifier* kavramına altı alt kavram tanımlanmıştır. Bunlar, *PrivateModifier, PublicModifier, StaticModifier, AbstractModifier, FinalModifier ve ProtectedModifier*'dır.

Package: Java projelerindeki herbir paketi temsil eden olguları içeren kavramdır.

Type: Tüm Java türlerini kapsayan üst kavramdır. *ArrayType, PrimitiveType* ve *SimpleType* alt kavramları tanımlanmıştır.

PrimitiveType: int, boolean, long gibi temel Java türlerini temsil eden olguları içerir.

SimpleType: Sınıf ve arayüz gibi basit Java türlerini temsil eden olguları içeren kavramdır. *Class* ve *Interface* alt kavramları tanımlanmıştır.

Class: Java sınıflarını temsil eden olguları içeren kavramdır. Sınıf olgularını metod olguları ile ilişkilendirme için *Class_has_Method*, erişim belirteci olguları ile

ilişkilendirme için *Class_has_Modifier*, paket olguları ile ilişkilendirme için *Class_to_Package*, kalıtılan sınıf olgusu ile ilişkilendirme için *Class_superClassType_Class*, gerçeklenen arayüz olguları ile ilişkiendirme için *Class_superInterfaceType_Interface*, sahip olunan sınıf alanı olguları ile ilişkilendirme için *Class_has_Field*, dahili sınıf (Java sınıfı içerisinde tanımlanmış üye yada yerel sınıf) olguları ile ilişkilendirme için *Class_innerClass_Class*, dahili arayüz (Java sınıfı içerisinde tanımlanmış üye yada yerel arayüz) olguları ile ilişkilendirme için *Class_innerInterface_Interface*, ilişkileri tanımlanmıştır.

Interface: Java arayüzlerini temsil eden olguları içeren kavramdır. Arayüz olgularını metod olguları ile ilişkilendirme için *Interface_has_Method*, erişim belirteci olguları ile ilişkilendirme için *Interface_has_Modifier*, paket olguları ile ilişkilendirme için *Interface_to_Package*, kalıtılan arayüz olguları ile ilişkilendirme için *Interface_superInterfaceType_Interface*, sahip olunan arayüz alanı olguları ile ilişkilendirme için *Interface_has_Field*, dahili arayüz (Java arayüzü içerisinde tanımlanmış üye arayüz) olguları ile ilişkilendirme için *Interface_innerInterface_Interface* ilişkileri tanımlanmıştır.

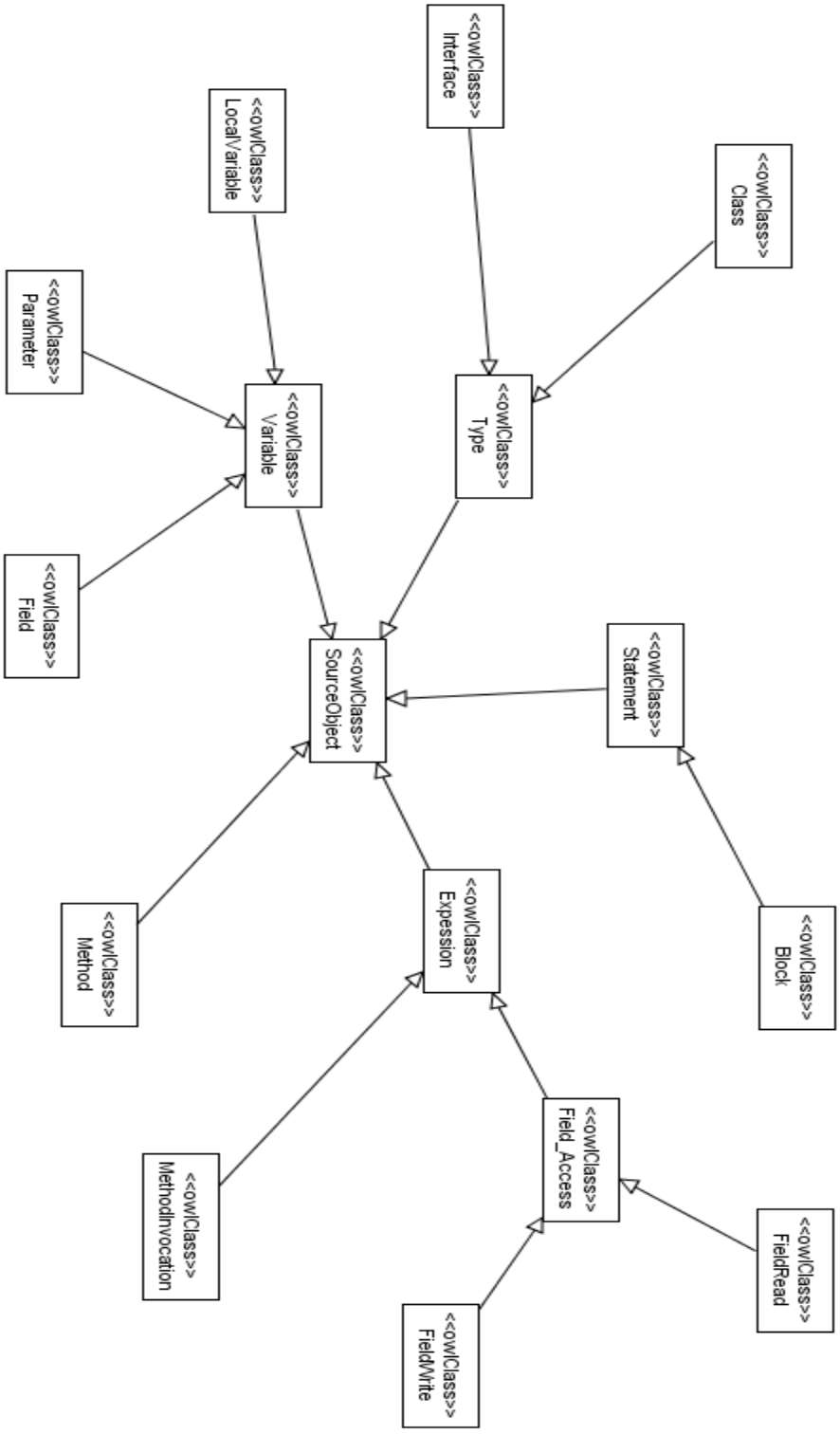
Variable: Kod içerisindeki tüm değişkenleri temsil eden olguları içeren üst kavramdır. *Field, LocalVariable* ve *Parameter* alt kavramları tanımlanmıştır.

Field: Sınıf alanlarını temsil eden olguları içeren kavramdır. Alanın türünü belirten tür olguları ile ilişkilendirme için *Field_has_Type*, erişim belirteci olguları ile ilişkilendirme için *Field_has_Modifier* ilişkileri tanımlanmıştır.

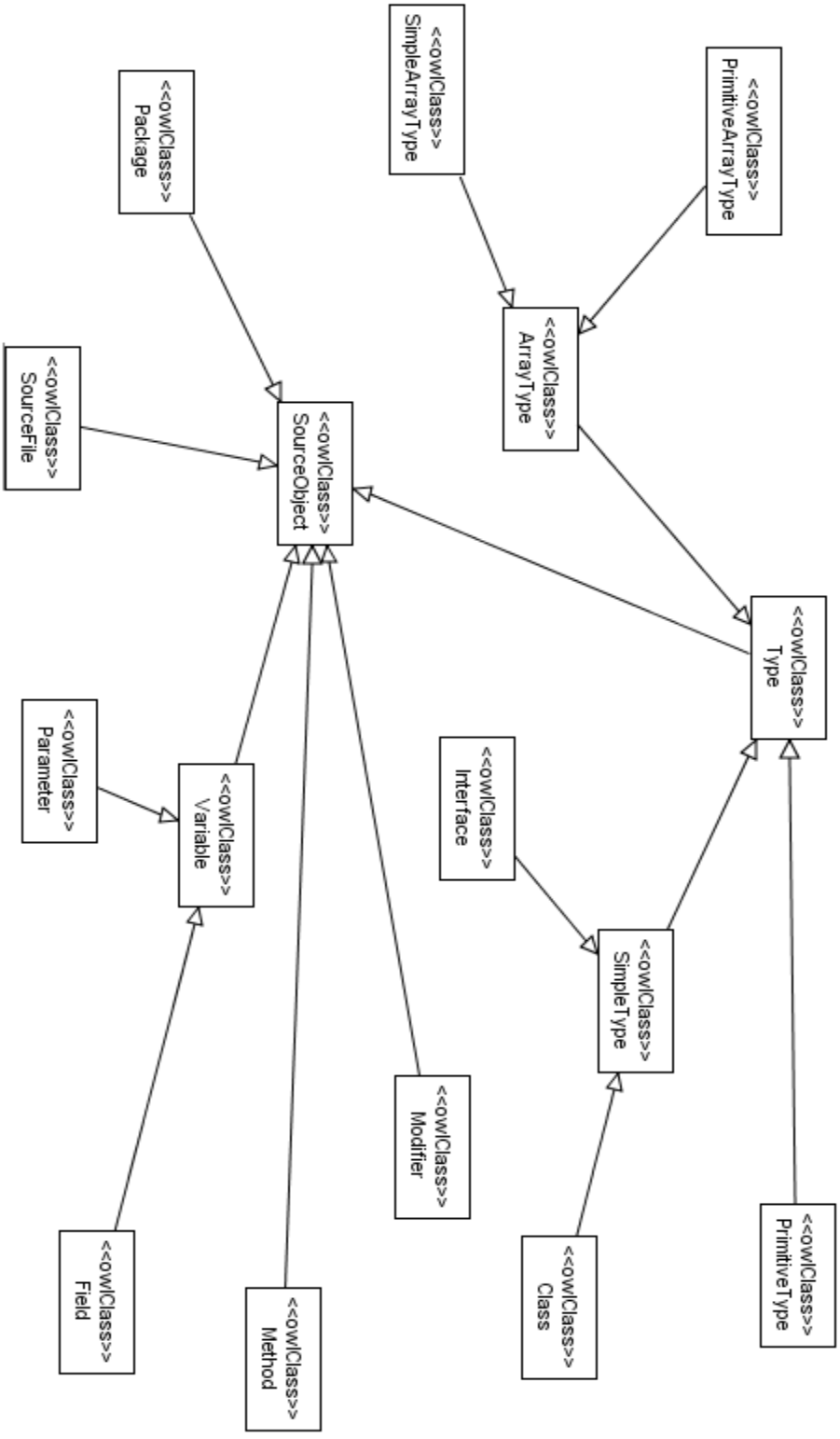
LocalVariable: Yerel değişkenleri temsil eden olguları içeren kavramdır. Yerel değişkenin türünü belirten tür olguları ile ilişkilendirme için *LocalVariable_has_Type*, erişim belirteci olguları ile ilişkilendirme için *LocalVariable_has_Modifier* ilişkileri tanımlanmıştır.

Parameter: Metod parametrelerini temsil eden olguları içeren kavramdır. Parametrenin türünü belirten tür olguları ile ilişkilendirme için *Parameter_has_Type*, erişim belirteci olguları ile ilişkilendirme için *Parameter_has_Modifier* ilişkileri tanımlanmıştır.

Block: Metodlar içerisindeki blokları temsil eden olguları içeren kavramdır. Metodların ana gövdeleri ve bunların içerdiği for, while, if, try-catch gibi kalıpları oluşturan herbir blok için ayrıştırıcı tarafından olgu oluşturulur. Blok ve bir alt seviyedeki blokları ilişkilendirme için *Block_has_Block*, blok içerisindeki *FieldRead* olguları ile ilişkilendirme için *Block_readsField_FieldRead*, *FieldWrite* olguları ile ilişkilendirme için *Block_writesField_FieldWrite*, *LocalVariable* olguları ile ilişkilendirme için *Block_declares_LocalVariable*, blok ile içerisinde tanımlanmış yerel sınıf yada arayüz belirtimi olgularını ilişkilendirme için *Block_has_LocalTypeDeclaration* ilişkileri tanımlanmıştır.



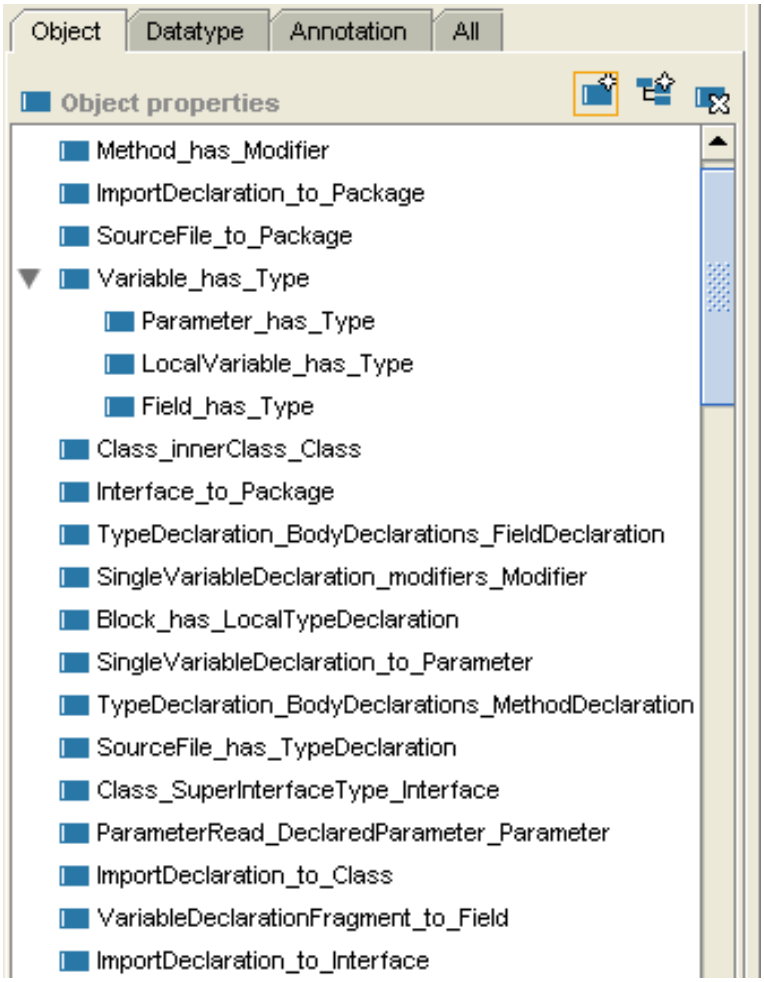
Şekil 4.4. (a) Değişken, Blok, Sınıf Alanı Erişimi gibi alt düzey Java kavramlarının ve aralarındaki sıradüzensel ilişkilerin gösterildiği sınıf diagramı.



Şekil 4.4. (b) Paket, Erişim Belirteci, Tür gibi üst düzey Java kavramlarının ve aralarındaki sıradüzensel ilişkilerin gösterildiği sınıf diagramı.

4.2.2.2. İlişkiler (Relations)

Java kod ontolojisi, herbiri OWL nesne türü ve OWL veri türü nitelikleri olarak tanımlanmış ilişkileri içermektedir.



Şekil 4.5. Java kaynak kod ontolojisi nesne türü ilişkileri görünümü.

Nesne türü ilişkiler tanım ve erim kümeleri OWL kavramları (sınıfları) olan ilişkilerdir.

Ontolojide tanımlı olan nesne türü ilişkilerden en önemlileri:

Class_has_Method: Tanım kümesi *Class* kavramı, erim kümesi *Method* kavramı olan, sınıf ve metod olguları arasındaki ilişkidir.

Class_has_Field: Tanım kümesi *Class* kavramı, erim kümesi *Field* kavramı olan, sınıf ve sınıf alanlarının olguları arasındaki ilişkidir.

Class_to_Package: Tanım kümesi *Class* kavramı, erim kümesi *Package* kavramı olan, sınıf ve ait olduğu paketi temsil eden olgular arasındaki ilişkidir.

Method_body_Block: Tanım kümesi *Method* kavramı, erim kümesi *Block* kavramı olan, metod ve blok olguları arasındaki ilişkidir. Bir metod içerisinde teorik olarak sonsuz tane iç içe blok tanımlanabilir. Metod içerisinde tanımlanan herbir bloğu temsil eden olgu ile bunları kapsayan method olgusu, ayrıştırıcı tarafından *Method_body_Block* ilişkisi ile bağlanır.

Block_has_Block: Tanım ve erim kümesi *Block* kavramı olan ve blok olgularının birbirleri ile ilişkilendirilmesinde kullanılan ilişkidir. Kod içerisindeki blokların tesbiti, ait oldukları metod ve birbirleri ile ilişkilendirilebilmeleri ileri düzey kod analizi için önemli bir özelliktir.

Class_innerClass_Class: Tanım kümesi ve erim kümesi *Class* kavramı olan, sınıflar ve sınıflar içerisinde tanımlanmış dahili üye sınıflarının olguları arasındaki ilişkidir.

Block_has_LocalTypeDeclaration: Tanım kümesi *Block*, erim kümesi *TypeDeclaration* olan, bloklar ve bu bloklar içerisinde tanımlanmış yerel sınıf yada arayüz belirtimi olguları arasındaki ilişkidir.

Block_writesParameter_ParameterWrite: Tanım kümesi *Block*, erim kümesi *ParameterWrite* olan, bloklar ve bu bloklar içerisindeki parametre yazma olguları arasındaki ilişikidir.

Block_readsLocalVariable_LocalVariableRead: Tanım kümesi *Block*, erim kümesi *LocalVariableRead* olan, bloklar ve bu bloklar içerisindeki yerel değişken okuma olguları arasındaki ilişikidir.

Method_returnType_Type: Tanım kümesi *Method*, erim kümesi *Type* olan, metotlar ve bu metotların dönüş türlerini temsil eden olgular arasındaki ilişikidir.

Method_implements_Method: Tanım ve erim kümesi *Method* olan, Java metodu ve bu metodun gerçekleştirdiği metodu temsil eden olgu arasındaki ilişikidir.

Block_methodInvocation_MethodInvocation: Tanım kümesi *Bock*, erim kümesi *MethodInvocation* olan, metod içerisindeki bir blok ve bu blok içerisindeki metod çağrımlarını temsil eden olgular arasındaki ilişikidir.

MethodInvocation_calles_Method: Tanım kümesi *MethodInvocation*, erim kümesi *Method* olan, metod çağrımı ve bu çağrımda kullanılan metodu temsil eden olgular arasındaki ilişkidir.

FieldAccess_DeclaredField_Field: Tanım kümesi *FieldAccess*, erim kümesi *Field* olan, sınıf alanı erişimi ve erişilen sınıf alanını temsil eden olgular arasındaki ilişkidir.

Method_overrides_Method: Tanım ve erim kümesi *Method* olan, Java metodu ve bu metodun geçersiz kıldığı metodu temsil eden olgular arasındaki ilişikidir.

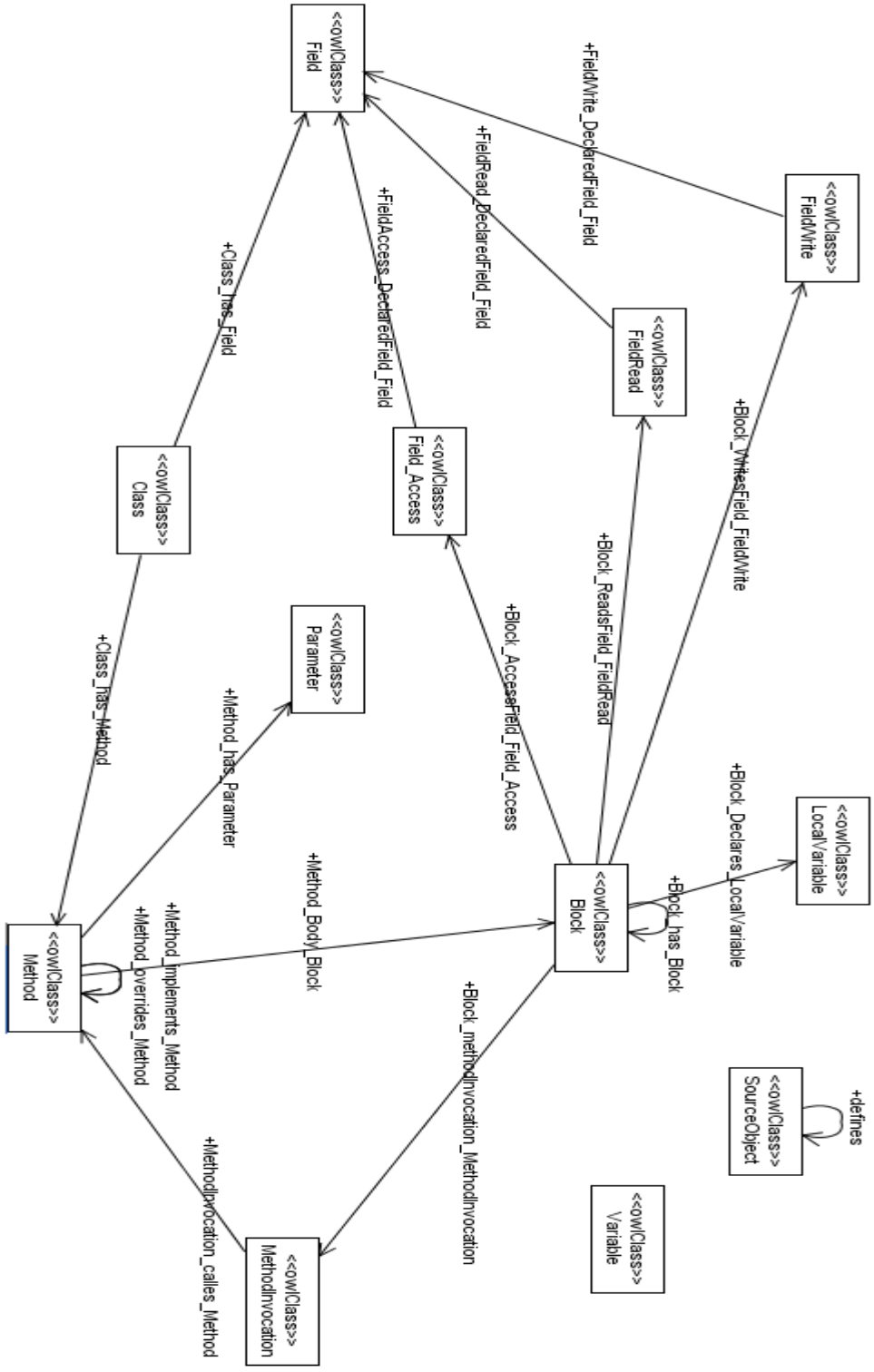
TypeDeclaration_to_Class: Tanım kümesi *TypeDeclaration* ve erim kümesi *Class* olan, tür belirtimi ve belirtimi yapılan Java sınıfını temsil eden olgular arasındaki ilişikidir.

Class_SuperClassType_Class: Tanım ve erim kümesi *Class* olan, Java sınıfı ve bu sınıfın kalıttığı sınıfı temsil eden olgular arasındaki ilişikidir. Burada kalıtılan sınıf soyut (*Abstract* erişim belirtecine sahip) bir sınıf olabilir.

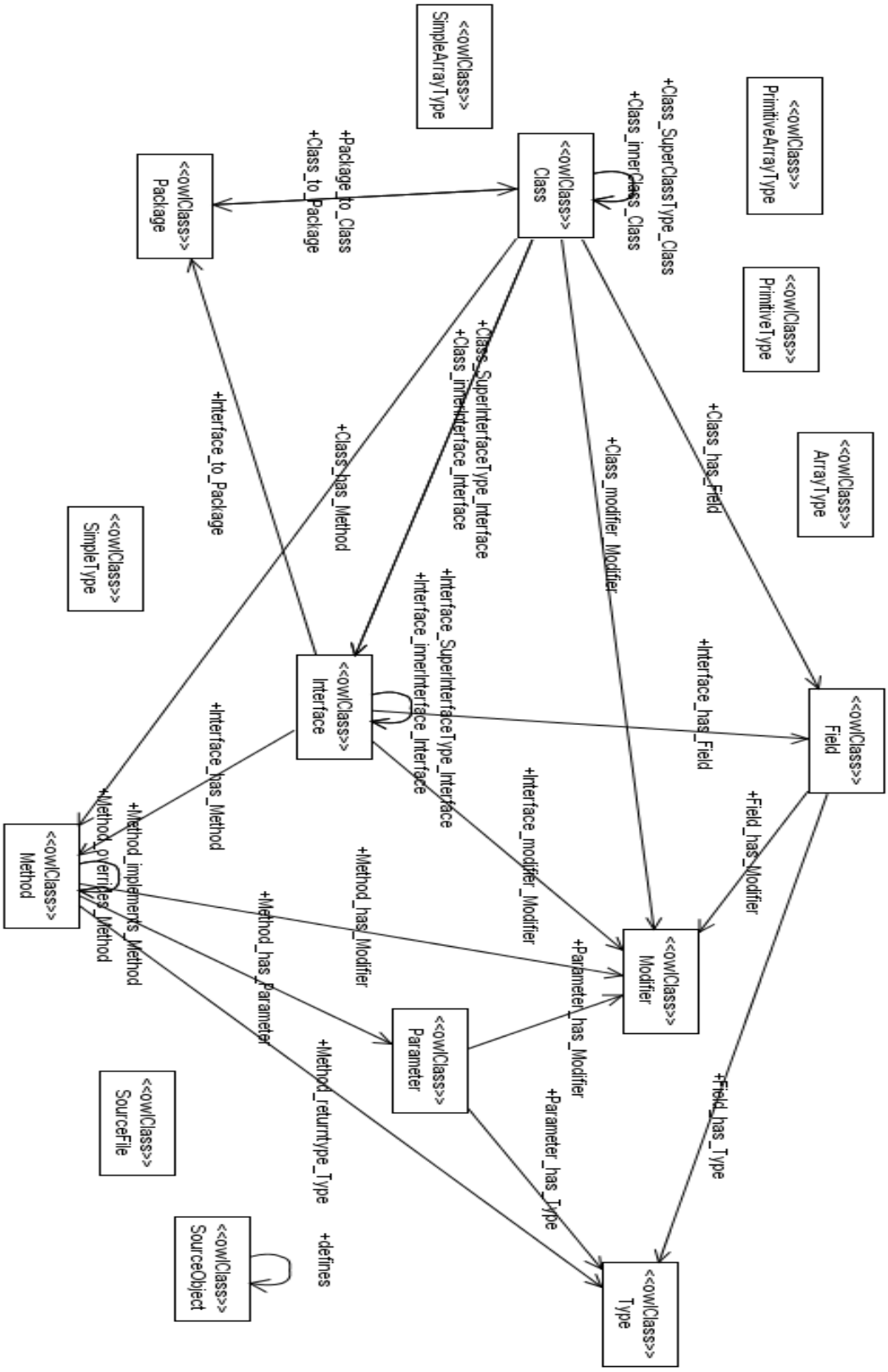
Class_innerInterface_Interface: Tanım kümesi *Class*, erim kümesi *Interface* olan, Java sınıfı ve bu sınıfın içerdiği dahili arayüzleri temsil eden olgular arasındaki ilişikidir.

SourceFile_has_ImportDeclaration: Tanım kümesi *SourceFile,* erim kümesi *ImportDeclaration* olan, kaynak kod kütüğü ve bu kütüğün başında yer alan içe aktarma bildirimlerini temsil eden olgular arasındaki ilişikidir.

ImportDeclaration_to_Class: Tanım kümesi *ImportDeclaration*, erim kümesi *Class* olan, kaynak kod kütüğü başında yer alan içe aktarma belirtimleri ile içe aktarılan sınıfları temsil eden olgular arasındaki ilişikidir.



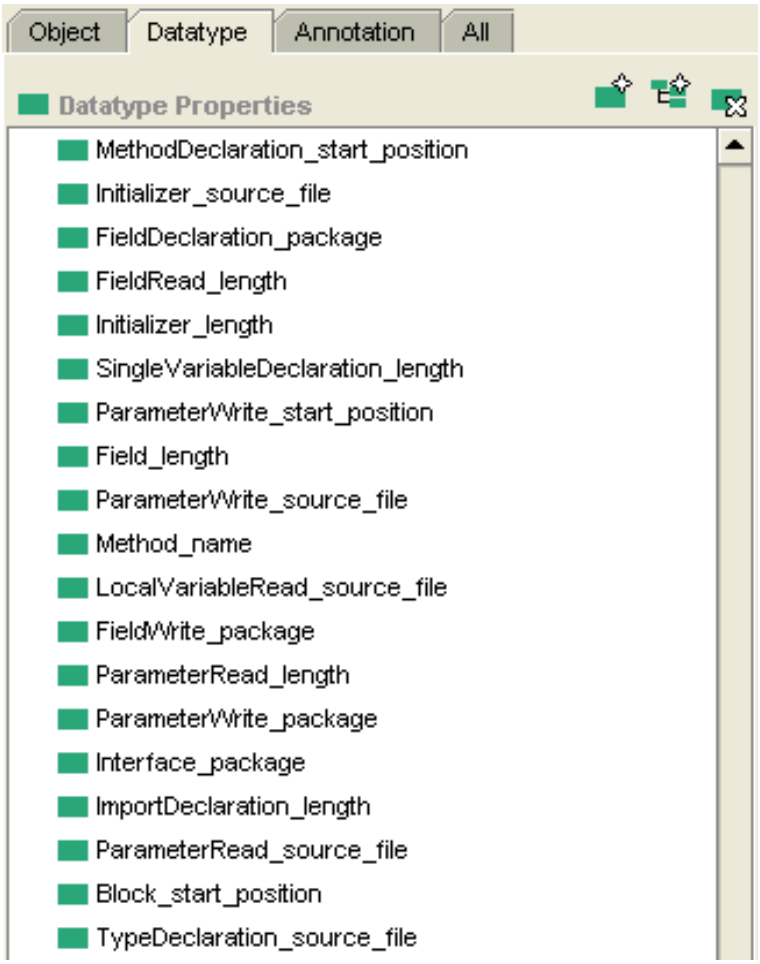
Şekil 4.6. (a) Değişken, Blok, Sınıf Alanı Erişimi gibi alt düzey Java kavramları ve aralarındaki nesne türü ilişkilerin gösterildiği sınıf diagramı



Şekil 4.6. (b) Paket, Erişim Belirteci, Tür gibi üst düzey Java kavramları ve aralarındaki nesne türü ilişkilerin gösterildiği sınıf diagramı

Veri türü ilişkileri, tanım kümesi bir OWL sınıfı, erim kümesi ise veri türünde olan ilişkilerdir.

Ontolojide tanımlı olan veri türü ilişkilerden en önemlileri:



Şekil 4.7. Java kaynak kod ontolojisi veri türü ilişkileri görünümü.

Method_is_constructor: Tanım kümesi *Method*, erim kümesi *string (http://www.w3.org/2001/XMLSchema#string)* olan, metodun yapıcı olup olmadığı bilgisini saklamak için kullanılan veri türündeki ilişkidir.

Method_name: Tanım kümesi *Method*, erim kümesi *string* olan, Java metoduna ait isim bilgisinin saklanması için kullanılan veri türündeki ilişkidir.

Method_key: Tanım kümesi *Method*, erim kümesi *string* olan, Java metoduna ait ayırt edici anahtar değer bilgisinin saklanması için kullanılan veri türündeki ilişkidir.

Method_start_position: Tanım kümesi *Method*, erim kümesi *int (http://www.w3.org/2001/XMLSchema#int)* olan, Java metoduna ait tanımlandığı kaynak kod kütüğü içerisindeki başlangıç konumu bilgisinin saklanması için kullanılan veri türündeki ilişkidir.

Method_length: Tanım kümesi *Method*, erim kümesi *int* olan, Java metoduna ait tanımlandığı kaynak kod kütüğü içerisindeki karakter bazında uzunluk bilgisinin saklanması için kullanılan veri türündeki ilişkidir.

Method_package: Tanım kümesi *Method*, erim kümesi *string* olan, Java metodunun içerisinde tanımlandığı paket ismi bilgisinin saklanması için kullanılan veri türündeki ilişkidir.

Yukarıda bahsedilen Java metodları için tanımlanmış isim, anahtar değer, başlangıç değeri, uzunluk, paket ismi gibi veri türü ilişkileri yaklaşık olarak tüm ontoloji kavramları için (*Class, Field, LocalVariable, Parameter, MethodInvocation, Interface, FieldAccess* gibi) tanımlanmıştır. Bu bilgiler, kaynak kod nesnelerinin Eclipse ortamında sorgulandıktan sonra ait oldukları kod kütükleri üzerinde gösterilmelerinde kullanılmaktadır.

İyi bir ontoloji tasarımı gereği, benzer nitelikteki bu ilişkilerin daha üst ontoloji kavramları için (*SourceObject, Type* gibi) tanımlanıp, alt ontoloji kavramları tarafından kalıtım yolu ile kullanılır duruma getirilmesi gerektiği düşünülebilir. Tez kapsamında, ontoloji tasarımındaki sıradüzensel yapının, ortak yapısal özellikteki kavramların gruplanmasından çok, ortak anlamsal ve davranışsal özellikteki kavramların gruplanması amacı güdülerek oluşturulmuş olması, belirtilen düşünceye karşı bir açıklama olarak gösterilebilir.

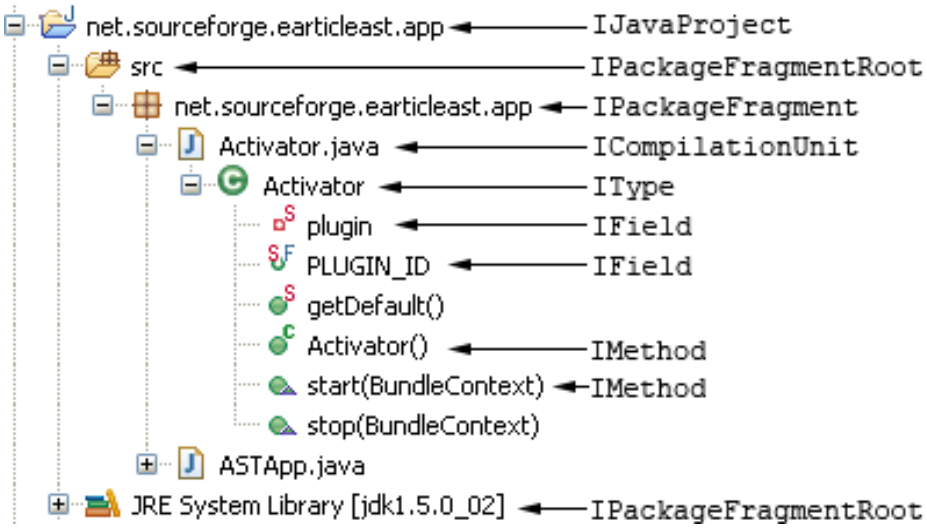
Ontoloji olgusu içerisindeki herbir olgunun adı biricik olmalıdır. Bu sorun, herbir kod elementini temsil eden olguya, temsil ettiği elementin adını vermekle çözülemez. Örneğin, proje içerisinde "getName" isminde birden fazla metod olabilir. Bu metodları temsil eden tüm olgular için "getName" ismi kullanılamaz. Bu nedenle, herbir olgu için farklı isim üretilmesini sağlayan bir yöntem geliştirilmiştir.

Oluşturulan her olgu için, olgunun ait olduğu kavram ismi ile rastgele ve daha önce başka bir olgu tarafından kullanılmamış bir sayının birleşimi, isim olarak atanır (“Method_3422” gibi).

4.3. Java Projesi Ontoloji Olgusunun Otomatik Olarak Oluşturulmasını Sağlayan Ayrıştırıcı Tasarımı

Java programlama dili için bir ontoloji tasarlandıktan sonra, üzerinde çalışılan herhangi bir Java projesi ontoloji olgusunun otomatik olarak oluşturulmasını sağlayan bir ayrıştırıcı geliştirilmiştir.

Ayrıştırıcı tasarımında Eclipse’in sunduğu *java modeli* ve *AST* yapıları kullanılmıştır.



Şekil 4.8. Eclipse Java Modeli [13].

Eclipse Java modeli, java eleman ağacının incelenebilmesi için arayüz sunmaktadır. Java eleman ağacı, herhangi bir java projesinin sadece üst düzey elemanlarını içeren bir yapıdır. Bunlar kod kütüğünü temsil eden derlem birimi (compilation unit), paket, basit türler (sınıf, arayüz gibi), method, sınıf alanı (field) gibi elemanlardır. [13]

Herhangi bir Java kodunun AST'si *ASTParser* sınıfının *createAST* metodu kullanılarak oluşturulur. *ASTParser* sınıfına kaynak kod dosyası, *ICompilationUnit* türünde parametre olarak geçilir. *ICompilationUnit*, java modelde kaynak kod dosyasında (.java) karşılık gelir. Herhangi bir kaynak kod kütüğünün AST'sinin oluşturulmasını sağlayan kod kesimi şu şekildedir:

```
ICompilationUnit tempIcu = ( IcompilationUnit ) cu_list.get( i );
ASTParser  astParser = ASTParser.newParser( AST.JLS3 );
astParser.setResolveBindings( true );
astParser.setSource( tempIcu );
        CompilationUnit  tempCu = ( CompilationUnit ) astParser.createAST ( null ) ) ;
```

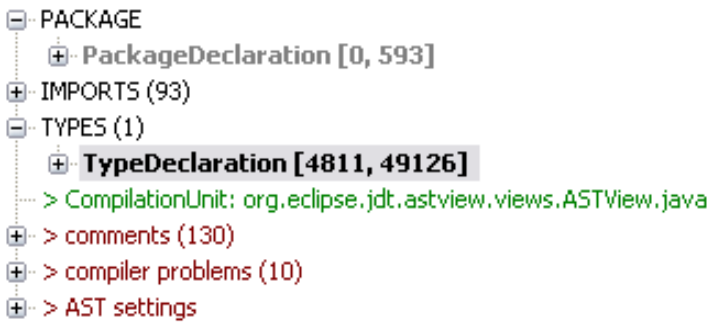
Ayrıştırıcının çalışma biçimi aşağıda ayrıntılı şekilde açıklanmıştır.

- İşleme sokulacak olan java projesi için Eclipse'in sunduğu java modeli kullanılarak proje içerindeki tüm .java kütükleri içeren paketler, sonra da bu paketler içerisindeki tüm derlem birimleri (kod kütükleri) çekilerek bir listeye atılır.

- Liste içerisinde bulunan herbir derlem birimi sıra ile işleme alınır.

- İlk olarak, sıradaki derlem birimi için *ASTParser* sınıfı tarafından oluşturulan soyut sözdizim ağacının *CompilationUnit* türündeki kök düğümü (root node) *derlem birimi yapısal özellik ayrıştırıcısı* olarak nitelendirilebilecek sınıfa parametre olarak geçilir.

- Herbir *yapısal özel özellik ayrıştırıcısı* sınıfı farklı türde AST düğümünü analiz etmeye yarar. Bunlardan *derlem birimi yapısal özellik ayrıştırıcısı* sınıfı, ilgili derlem birimini ontolojideki kavram ve ilişkiler bazında analiz ederek, ilgili OWL olgularının oluşturulmasını sağlayacak olan sınıflardan ilkidir. Bu sınıf, *CompilationUnit* türündeki AST düğümünü parametre olarak alır ve düğüme ait tüm yapısal özellikleri çeker. Bunlar, paket bildirimini içeren *package*, içe aktarma bildirimlerini içeren *imports* ve sınıf ve arayüz bildirimlerini içeren *types* özellikleridir.



Şekil 4.9. *CompilationUnit* türündeki AST düğümü yapısal özellikleri.

Aktif olan *derlem birimi yapısal özellik ayrıştırıcısı* sınıfı parametre olarak aldığı *CompilationUnit* türündeki AST düğümünün özelliklerini çektikten sonra bu özelliklerden elde ettiği değerleri kullanarak ilgili türdeki olguları oluşturur. İlk olarak, tanımlanan sınıf yada arayüzün içerisinde bulunduğu paketi temsil eden *Package* türündeki ve kod kütüğünün başında belirtilen içe aktarma belirtimlerini temsil eden *ImportDeclaration* türündeki olgular oluşturulur. Daha sonra, işleme sokulan derlem birimi içerisinde tanımlanmış tür belirtimi için *types* özelliğinin değeri kullanılarak *TypeDeclaration* türündeki OWL olgusu oluşturulacaktır. Belirtilen kod öğelerini temsil eden herbir olgu oluşturulduktan sonra, kod öğelerini niteleyen başlangıç noktası, uzunluk, isim gibi bilgiler ilgili AST düğümlerinden çekilerek, ontoloji içerisinde tanımlanmış olan length, startposition, name gibi veri türü nitelikleri ile olgular ilişkilendirilir. Bir içe aktarma belirtimi için oluşturulan olgu ve ilgili veri türü niteliklerini içeren OWL kesimi aşağıda gösterilmiştir.

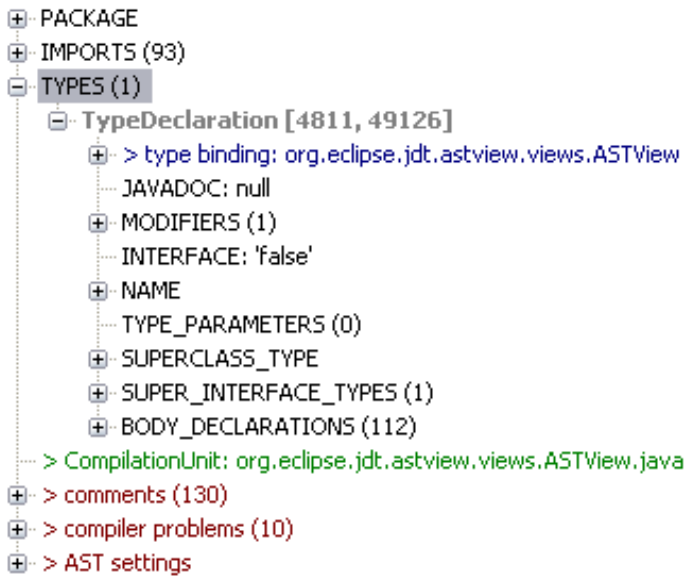
```
<ImportDeclaration rdf:ID="ImportDeclaration_60117">
    <ImportDeclaration_package rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            org.eclipse.jdt.astview.views
    </ImportDeclaration_package>
    <ImportDeclaration_name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            import org.eclipse.jdt.core.IClassFile;
    </ImportDeclaration_name>
    <ImportDeclaration_source_file
rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            ASTView.java
    </ImportDeclaration_source_file>
```

```
    <ImportDeclaration_length rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
            39
    </ImportDeclaration_length>
    <ImportDeclaration_start_position
rdf:datatype="http://www.w3.org/2001/XMLSchema#int">
            3908
    </ImportDeclaration_start_position>
</ImportDeclaration>
```

Ayrıştırma işlemine *tür bildirimi yapısal özellik ayrıştırıcısı*, *TypeDeclaration* türündeki AST düğümünü parametre alarak devam eder.

- *Tür bildirimi yapısal özellik ayrıştırıcısı*, parametre olarak aldığı *TypeDeclaration* türündeki AST düğümünün yapısal özelliklerini çeker ve herbir özellik için gerekli işlemleri gerçekleştirir. Örneğin, *name* basit türde bir özellik olup, sınıfın ismini belirtir. Sıra bu özelliğin işlenmesine geldiğinde, AST düğümünün *isInterface* özelliği kullanılarak, tanımlanan basit Java türünün sınıf mı yoksa arayüz mü olduğuna karar verilir ve *Class* yada *Interface* türünde OWL olgusu yaratılır. Sonra, bu sınıfa ait isim, uzunluk, başlangıç pozisyonu, paket ismi, ait olduğu kod kütüğü ismi gibi bilgiler çekilerek, ilgili veri türü nitelikleri ile tanımlanan OWL olgusu ilişkilendirilir bir başka ifade ile ontolojideki ilgili veri türü nitelikleri kullanılarak gerekli OWL ifadeleri oluşturulur ve ontololoji olgusuna eklenir.

- *Tür bildirimi yapısal özellik ayrıştırıcısı* sınıfının işlemesi gereken diğer özelliklerden bazıları şunlardır: *superclassType*, *superInterfaceTypes* ve *bodyDeclarations*. Bunlardan *superclassType* özelliği, işlenen sınıf başka bir sınıftan türememiş ise “null” değerine, türemiş ise *SimpleType* türündeki AST düğümü türünde bir değere sahiptir. Bu durumda, *basit tür yapısal özellik ayrıştırıcısı* sınıfı devreye girer ve türetilen sınıf için bir OWL olgusu oluşturur. Veri türü niteliklerini kullanarak bu sınıfı tanımlamak için gerekli OWL ifadelerini oluşturabilmek için düğümün yapısal özelliklerini çeker. *superInterfaceTypes* özelliği için ise yine *basit tür yapısal özellik ayrıştırıcısı* sınıfı kullanılır ve bu sefer işlenen sınıfın gerçekleştirdiği tüm arayüzler için OWL olguları oluşturulur.

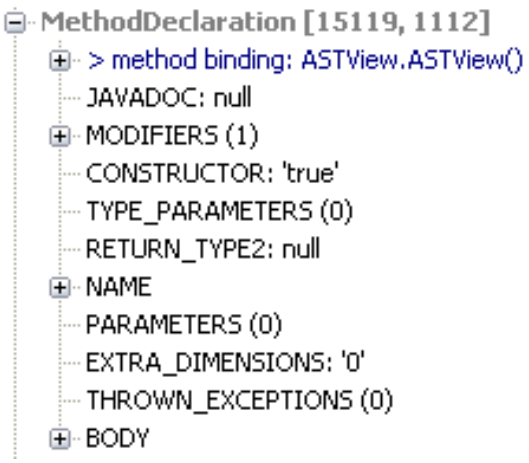


Şekil 4.10. *TypeDeclaration* türündeki AST düğümü yapısal özellikleri.

BodyDeclarations özelliği *tür bildirimi yapısal özellik ayrıştırıcısı* sınıfının işlediği en kapsamlı özelliktir. Burada, metod ve sınıf alanı bildirimleri analiz edilir. Herbir *MethodDeclaration* ve *FieldDeclaration* türündeki AST düğümü için yapılacak işlemler; *MethodDeclaration ve FieldDeclaration* türünde OWL olguları oluşturulup, işlenen AST düğümünden kod elemanına ait kod kütüğü üzerindeki uzunluk, başlangıç pozisyonu, bulunduğu kod kütüğü ve paket ismi gibi bilgilerin çekilerek ve ilgili OWL veri türü nitelileri kullanılarak, metod ve sınıf alanı bildirimi olgularını tanımlayan OWL ifadelerinin oluşturulmasıdır. Bu işlemler yapıldıktan sonra, belirtilen metod ve sınıf alanlarının sahip olduğu özelliklerin analiz edilebilmesi için (metod için; parametreler, dönüş türü, erişim belirteçleri, blok içeriği, sınıf alanı için; erişim belirteçleri, tür) *method bildirimi yapısal özellik ayrıştırıcısı* ve *sınıf alanı bildirimi yapısal özellik ayrıştırıcısı* sınıfları devreye girer ve ayrıştırmayı devam ettirir.

- Herbir metod bildirimi *metod bildirimi yapısal özellik ayrıştırıcısı* sınıfı tarafından analiz edilir. Ayrıştırıcı sınıfın işlemesi gereken özelliklerden bazıları şunlardır: *modifiers*, *constructor*, *returnType*, *parameters* ve *body*. Ayrıştırıcı sınıf *modifiers* yapısal özelliğini işlerken karşılaşılan herbir erişim

belirteci için bir OWL olgusu oluşturur (daha önce aynı olgu oluşturulmadı ise). *constructor* özelliği ile metodun yapıcı olup olmadığı anlaşılılır ve *Method_is_constructor* veri türü niteliği kullanılarak, ilgili olgu *true* yada *false* değeri ile ilişkilendirilir. *returnType* özelliği işlenirken karşılaşılan herbir java türü (temel yada basit java türleri) için bir OWL olgusu oluşturulur. *parameters* özelliği işlenirken karşılaşılan herbir parametre için *Parameter* türünde OWL olgusu oluşturulur.



Şekil 4.11. *MethodDeclaration* türündeki AST düğümü yapısal özellikleri.

Body yapısal özelliği, ileri düzey kod analizinin yapılabilmesi için önemlidir. Çünkü yazılımlarda iş mantığının kodlandığı yerler metodların gövde kesimleridir. Java ifadeleri, deyimleri, döngüler, atamalar (ilk değer atamaları dışında) ve karar yapıları gövde kesiminde yer alır.

- Herbir metod gövdesi, *metod gövdesi yapısal özellik ayrıştırıcısı* sınıfı tarafından analiz edilir. Ayrıştırıcı sınıfın görevi genel olarak, ilgili metod bildiriminin *Block* türündeki AST düğümünü parametre alarak, özyinelemeli olarak, ana blok dahil olmak üzere method içerisindeki tüm blokları (ilk değer atayıcılar, döngü ve koşul yapıları gibi) tesbit etmek ve ayrıştırmaktır. Metod içerisindeki blok yapılarının tesbit edilebilmesi için, ziyaretçi (visitor) tasarım örüntüsü kullanılarak geliştirilmiş olan *ASTVisitor* (*org.eclipse.jdt.core.dom.ASTVisitor*) çekirdek Java sınıfından türeyen, *BlockDedector* adındaki ayrıştırıcı sınıf kullanılır. Ayrıştırıcı sınıfın çalışma algoritması şu şekildedir:

1. İşlenen blok içerisindeki dahili sınıf ve arayüz belirtimleri tesbit edilerek, olgularının oluşturulup herbiri için ayrıştırmanın devam ettirilmesi için *Tür bildirimi yapısal özellik ayrıştırıcısı* sınıfına parametre olarak geçilir.

2. İşlenen bloğun bir alt seviyesindeki blokların tesbit edilmesi için *BlockDedector* ayrıştırıcı sınıfı kullanılır ve tesbit edilen herbir alt blok listeye eklenir. Listedeki herbir alt blok için;

    a) Blok içerisindeki tüm değişken (parametre, sınıf alanı ve yerel değişkenler için) okuma ve yazmaların ve metod çağrımlarının tesbit edilip, olgularının oluşturulması için ilgili ayrıştırıcı methodlar çağrılır.

    b) İşlenen alt bloğu temsil eden ontoloji olgusu ve gerekli veri türü ilişkileri kullanılarak oluşturulan OWL ifadeleri proje ontoloji olgusuna eklenir.

    c) İşlenen alt bloğun bir alt seviyesindeki bloklar için de aynı işlemlerin gerçekleştirilebilmesi için *metod gövdesi yapısal özellik ayrıştırıcısı* sınıfına parametre olarak geçilir.

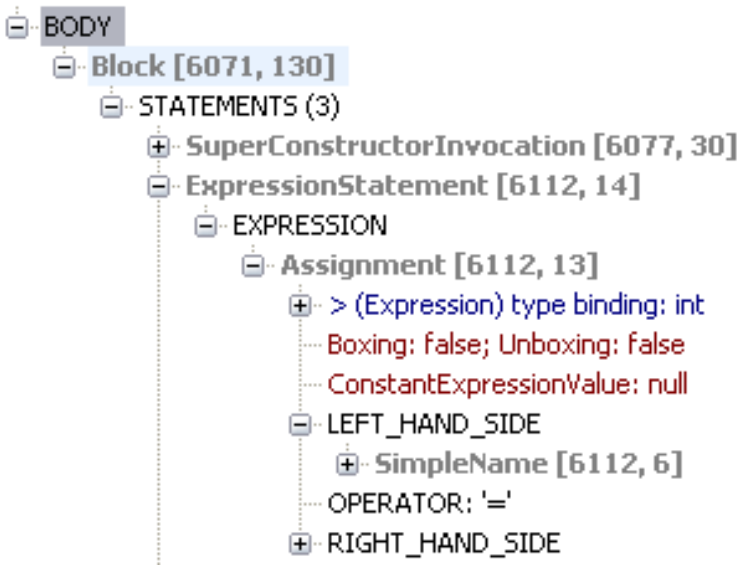
- Blok içerisindeki tüm değişken okuma ve yazmalarının tesbit edilmesi ve olgularının oluşturulması ASTVisitor çekirdek (core) Java sınıfından türeyen, *VariableReadWriteDedector* adındaki ayrıştırıcı sınıfı tarafından gerçekleştirilir.

```
private class ASTLevelToggle extends Action {
    private int fLevel;

    public ASTLevelToggle(String label, int level) {
        super(label, AS_RADIO_BUTTON);
        fLevel= level;
        if (level == getCurrentASTLevel()) {
            setChecked(true);
        }
    }
```

Şekil 4.12. Bir sınıf alanının yazılması durumu.

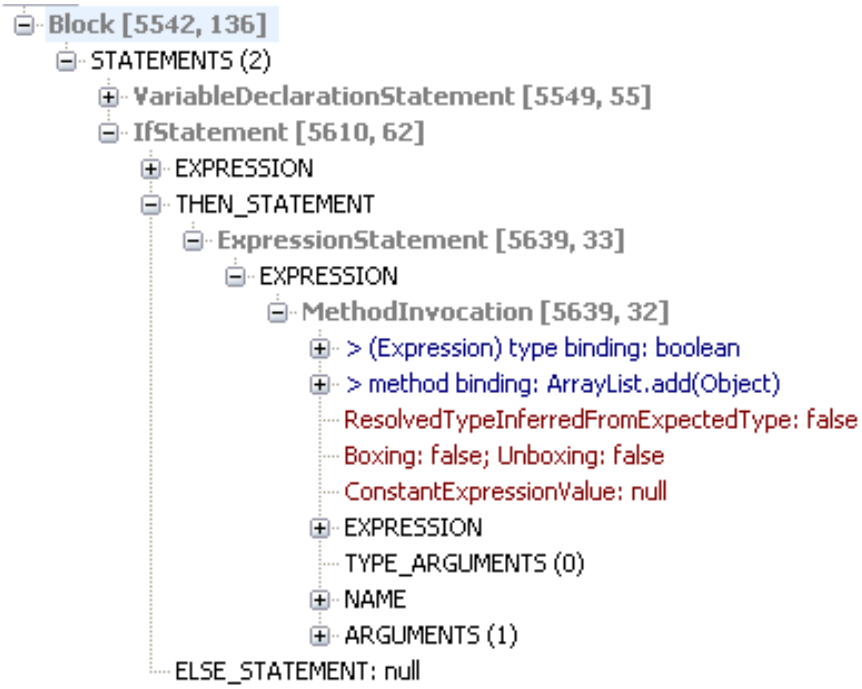
Şekil 4.13'te, Şekil 4.12'deki sınıf alanı yazılması durumunun AST üzerindeki gösterimi görülmektedir.



Şekil 4.13. Bir sınıf alanının yazılması durumunun AST üzerindeki görünümü.

Ayrıştırıcı sınıf, parametre olarak aldığı blok içerisindeki herbir parametre, yerel değişken ve sınıf alanına, içerisinde bulundukları Java ifadelerindeki konumlarını inceleyerek (atamalarda atanan durumunda olması, postfix-prefx ifadelerde yer alması gibi) okuma yada yazma yapıldığına karar verir ve incelenen değişken için okuma yada yazma olgusunu ve ilgili OWL veri türü ilişkilerini kullanarak tanımlayıcı ontoloji ifadelerini oluşturur.

- Blok içerisindeki tüm metod çağrımlarının tesbit edilmesi ve olgularının oluşturulması ASTVisitor çekirdek Java sınıfından türeyen, *MethodCallDedector* adındaki ayrıştırıcı sınıfı tarafından gerçekleştirilir.



Şekil 4.14. Blok yapısı içerisindeki bir metod çağrımı (MethodInvocation).

Ayrıştırıcı sınıf, parametre olarak aldığı blok içerisindeki method çağrımı olgusunu ve ilgili OWL veri türü ilişkilerini kullanarak tanımlayıcı ontoloji ifadelerini oluşturur.

- Anlaşılacağı üzere, ayrıştırma işlemi farklı türde AST düğümlerini işleyebilecek şekilde tasarlanmış yapısal özellik ayrıştırıcısı sınıfları tarafından sıradüzensel biçimde gerşekleştirilir. Her ayrıştırıcı sınıf, işlediği AST düğüm türü ile ilgili olan OWL sınıf ve veri türü niteliklerini kullanarak gerekli işlemleri gerçekleştirir ve java projesi OWL olgusunu günceller. Dikkat edilirse buraya kadar olan aşamalarda sadece OWL olgularının yaratılıp, ontolojideki veri türü niteliklerinin kullanılarak bu olguları tanımlayan OWL ifadelerinin oluşturulmasından bahsedilmektedir.

- Tüm derlem birimlerinin ayrıştırılıp, proje kapsamındaki tüm olgular oluşturulduktan sonra bu olgular arasındaki ilişkilerin kurulması, bunun için de analiz işleminin baştan alınması gerekir. İlk olarak tüm OWL olgularının oluşturulması, sonraki adımda ise analizin baştan alınarak, olgular arasındaki ilişkilerin kurulması şu şekilde açıklanabilir: Belirtilen iki işlemin

paralel olarak gerçekleştirilmesi, bazı problemlere yol açabilecektir. Örneğin, proje içerisindeki bir A sınıfı, bir B sınıfını kalıtıyor olabilir. Bu durumda ontoloji örneğinde, A sınıf olgusu ile B sınıf olgusu, *Class_SuperClassType_Class* nesne niteliği ile ilişkilendirilmelidir. Fakat, analiz sırasında B sınıf olgusunun henüz oluşturulmamış olması, bu ilişkinin kurulmasını engeller. Bu nedenle ilk olarak tüm olguları oluşturup, sonraki adımda olgular arasındaki ilişkileri kurmak uygun bir yöntem olarak düşünülmüştür.

- İkinci kez yapılacak olan analizde kullanılmak üzere, olguların oluşturulmasında kullanılan yapısal özellik ayrıştırıcı sınıflara benzer şekilde çalışan ve var olan olgular arasındaki ilişkileri kuracak olan sınıflar tasarlanmıştır. Bu sınıflar da sıradüzensel bir şekilde çalışırlar ve hangi durumlarda hangi tür olgular arasında hangi ilişkilerin kurulacağı bilgileri ile kodlanmışlardır.

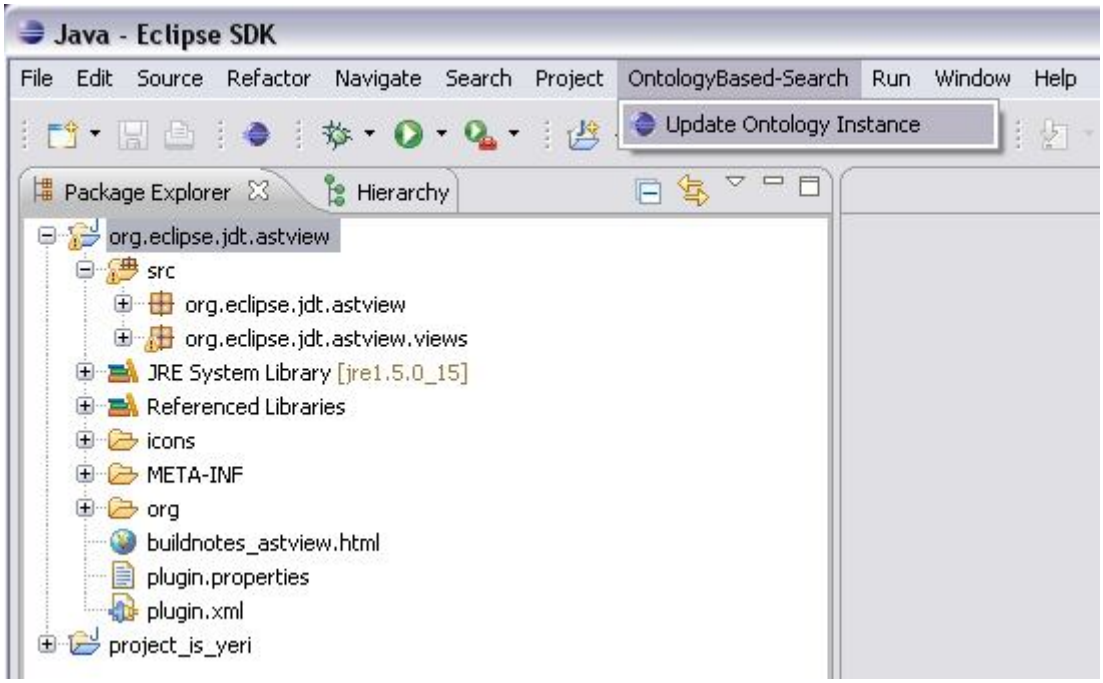
- Son olarak, kod içerisindeki kavramlar ve bu kavramlar arasındaki ilişkilerin temsil edildiği ontoloji olgusu, kalıcı olarak OWL formatında kaydedilir.

4.4. Araç Kullanım Arayüzü ve Örnek Sorgulamalar

Geliştirilen ontoloji tabanlı kod sorgulama aracını denemek amacıyla, toplam yirmi yedi .java uzantılı kod kütüğünden ve 5475 satır koddan oluşan ASTView eklenti projesi örnek olarak kullanılmıştır.

4.4.1. Ayrıştırma İşleminin Başlatılması

Üzerinde sorgulama yapılacak olan Java projesi, Eclipse geliştirme ortamında açılır ve Şekil 4.15'te görüldüğü gibi *Update Ontology Instance* menü düğmesi ile kod ayrıştırma yani ontoloji olgusunun oluşturulması işlemi başlatılır.



Şekil 4.15. Java projesi için ayrıştırma işleminin başlatılması.

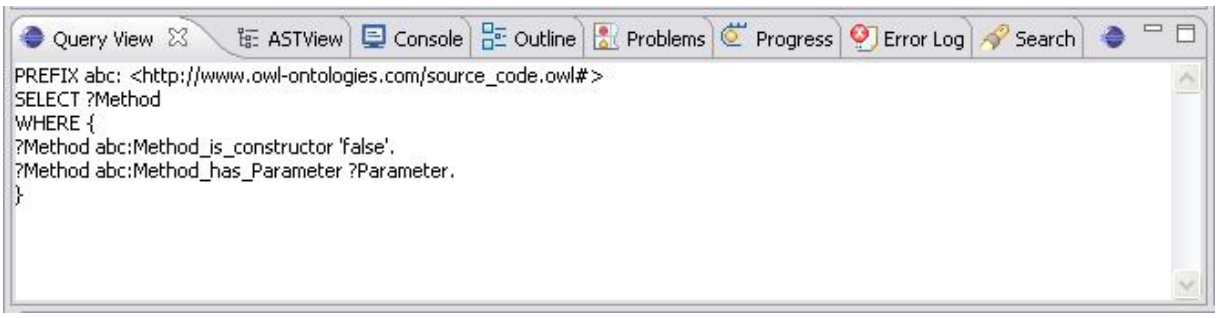
ASTView eklenti projesinin ayrıştırılması ve ontoloji olgusunun oluşturulması için harcanan işletim süreleri Çizelge 4.1'de belirtilmiştir.

Çizelge 4.1. ASTView eklenti projesi ayrıştırma aşamaları için işletim süreleri.

İşlem Aşaması	**İşletim Süresi**
Proje elementleri için ontoloji olgularının oluşturulması	~61250 ms
Oluşturulan ontoloji olguları arasındaki ilişkilerin kurulması	~ 160687 ms

4.4.2. Sorgulama ve Sonuç Görünümleri

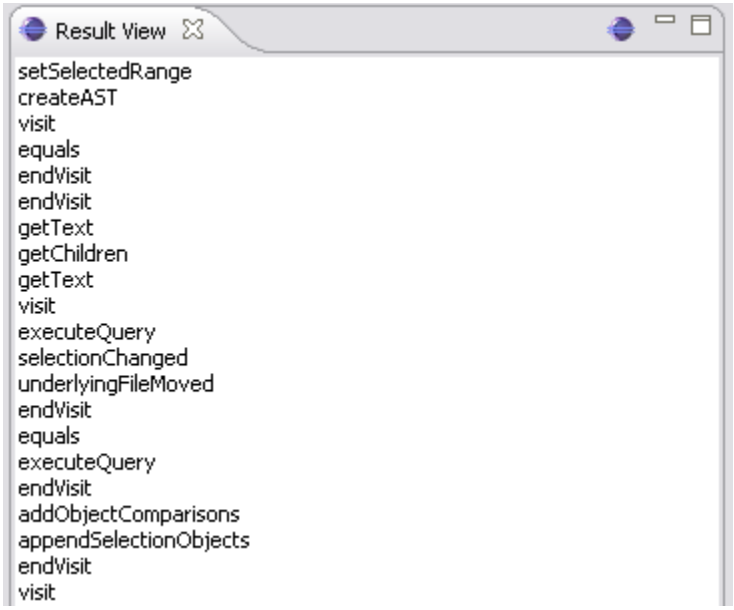
Java Projesi ontoloji olgusunun otomatik olarak üretilip, OWL formatında kaydedilebilmesinden sonra, Eclipse geliştirme ortamı ile ontoloji olgusu arasında bir sorgulama arayüzüne ihtiyaç duyulmuştur. Bu arayüz, eklenti olarak geliştirilen sorgu ve sonuç görünümleri ile sağlanmıştır.



Şekil 4.16. Sorgu görünümü.

SPARQL sorguları Şekil 4.16'daki gibi sorgu görünümüne düz metin olarak yazılmaktadır. Örnekteki sorguda, "yapıcı olmayan ve parametre alan tüm metodlar"ın listelenmesi istenmektedir.

Şekil 4.17'de, ilgili sorgunun işletilmesi sonucunda elde edilen sonuç listesi, sonuç görünümü üzerinde görülmektedir.



Şekil 4.17. Sonuç Görünümü.

Sorgu görünümü üzerine girilen herhangi bir sorgunun işlenip, listelenmesine kadar olan süreç şu adımlarla açıklanabilir:

- Sonuç görünümünün sağ üst köşesinde bulunan düğme ile eklenti tetiklenir.

- Eklenti, sorgu görünümü üzerindeki sorgu metnini karakter dizisi olarak alıp ayrıştırarak, sonuç görünümü üzerinde listelenecek olan kod elemanı türünü çeker. Şekil 4.16'daki örnek için, listelenecek kod elemanları türünün *Method* olduğu görülmektedir.

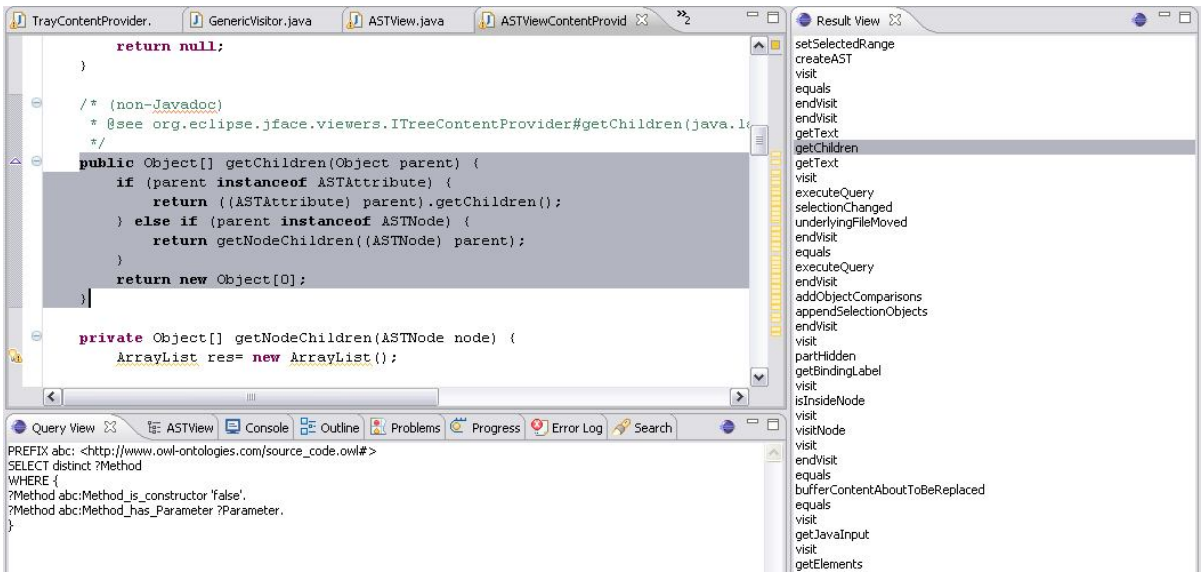
- Sorgunun işletilmesi ve sonuç listesinin elde edilmesi aşağıdaki kod kesimi ile sağlanır.

```
Query query = QueryFactory.create(queryString);
QueryExecution queryExecution = SparqlDLExecutionFactory.create( query,
model );
ResultSet rs = qe.execSelect();
```

- Sonuç kümesindeki herbir olgu için, sonuç görünümünde olguyu temsil edecek olan bir nesne oluşturulur. Bu nesne içerisinde olguya ait isim, kod kütüğü üzerindeki başlangıç konumu, uzunluğu, ait olduğu kaynak kütüğü ismi gibi bilgiler bulunmaktadır. Tüm bu bilgiler, sonuç görünümündeki herbir elamanın kod editörü üzerinde gösterilmesinde kullanılır.

- Olgular için oluşturulan nesnelerin tümü, sonuç görünümünün içerik sağlayıcı (content provider) sınıfına geçilerek, olgu isimlerinden oluşan bir listenin gösterilmesi sağlanmış olur.

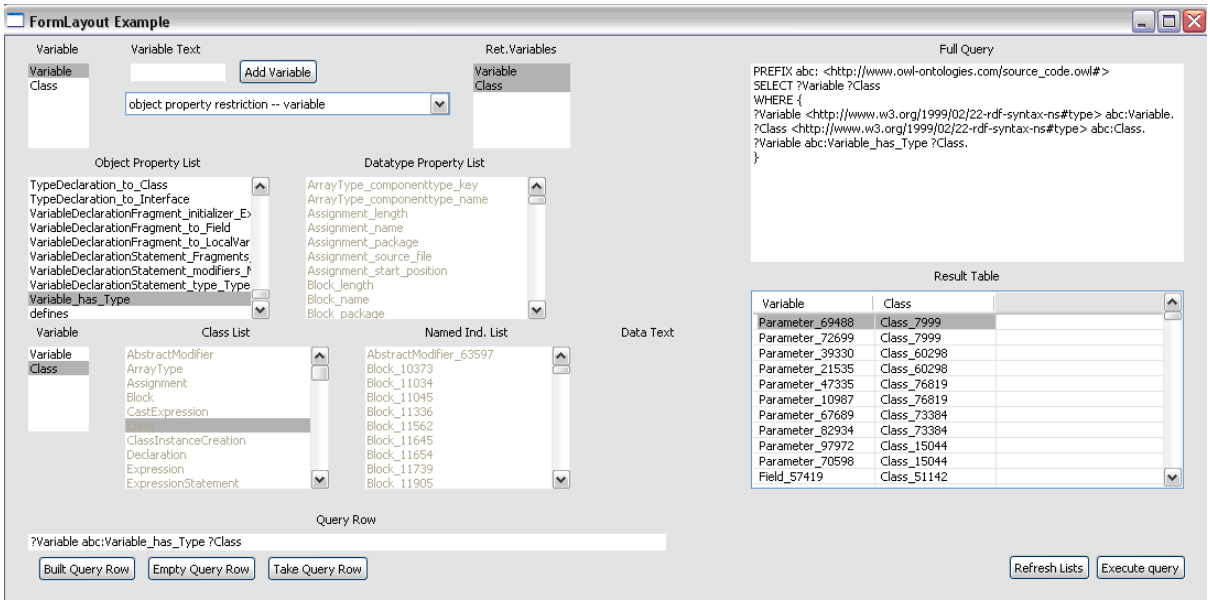
Sonuç görünümü üzerindeki listede bulunan elemanlardan birine çift tıklandığında, seçilen elemanın bulunduğu kod kütüğü içerisindeki yeri işaretlenmiş olarak editör üzerinde gösterilir.



Şekil 4.18. Seçilen sonucun editör üzerinde gösterilmesi.

4.4.3. Sorgu Oluşturma İşlemini Kolaylaştırmak Amacıyla Geliştirilen Java SWT Sorgulama Arayüzü

Tez kapsamında, daha hızlı ve kolay şekilde SPARQL sorguları oluşturabilmek için bir Java SWT arayüzü geliştirilmiştir. Belirtilen arayüzün görünümü Şekil 4.19'da gösterilmiştir.



Şekil 4.19. Java SWT arayüz uygulaması görünümü.

Arayüz, sağladığı ilişki türü, ilişki ve kavramların seçilebilmesi, istenilen sayıda ve isimde parametre tanımlanabilmesi gibi özellikleri ile kolay bir şekilde SPARQL sorgularının oluşturulmasını sağlar. Ayrıca, oluşturulan sorguların çalıştırılmasına ve sorgu sonuçlarının ızgara görünümünde listelenmesine de imkan tanır.

4.4.4. Aracın İleri Düzey Sorgulamalarda Kullanımı Örneği: Sınıf Seviyesinde Kullanım-Bağımlılık (Use - Dependency) Çözümlemesi

Kod elementleri arasındaki bağımlılıkların tesbiti, yazılımın yapısal çözümlemesinde kullanılabilecek en önemli işlemlerden bir tanesidir. Tez kapsamında geliştirilen araç için örnek uygulama olarak, sınıf seviyesinde bağımlılık ele alınmıştır.

Sınıf seviyesinde bağımlılık türleri:

4.4.4.1. Değişken Kullanımı (Variable Use)

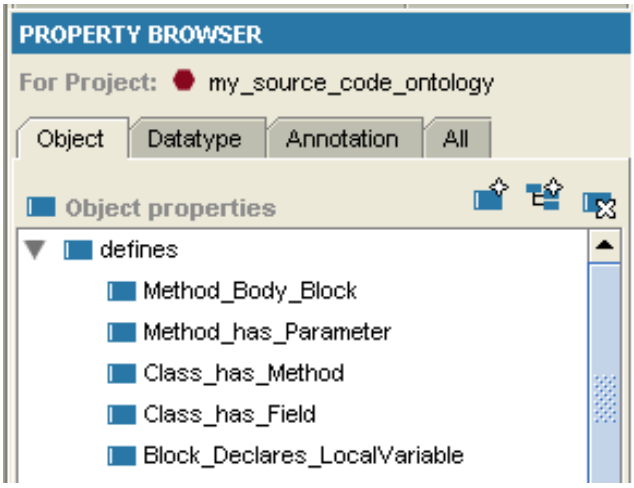
C1 sınıfı içerisinde, C2 sınıfı türünde bir değişken tanımlanması durumudur. Belirtilen durumun tesbitinde kullanılabilecek olan ve Java kod ontolojisinde ilk olarak tanımlanmış ilişkiler şunlardır:

- *Class* ve *Field* kavramları arasında tanımlanmış olan *Class_has_Field* ilişkisi
- *Method* ve *Parameter* kavramları arasında tanımlanmış olan *Method_has_Parameter* ilişkisi
- *Block* ve *LocalVariable* kavramları arasında tanımlanmış olan *Block_declares_LocalVariable* ilişkisi
- *Class* ve *Method* kavramları arasında tanımlanmış olan *Class_has_Method* ilişkisi
- *Method* ve *Block* kavramları arasında tanımlanmış olan *Method_body_Block* ilişkisi

Değişken kullanımı durumunun, proje kodu içerisinde tesbit edilebilmesi için sınıf içerisindeki parametre, yerel değişken ve sınıf alanı türündeki değişkenlere doğrudan erişilebilmesi gerekir. Yukarıda belirtilen ilişkilerden sadece

Class_has_Field ilişkisi sınıf ve sınıf alanı arasındaki ilişkiyi doğrudan ifade eder. Diğer ilişkiler yerel değişen ve parametre türündeki değişkenlere doğrudan *Class* olguları üzerinden erişime imkan vermez. Çünkü *Class* türündeki olguların *Parameter* ve *LocalVariable* türündeki olgular ile doğrudan değil, *Method* türündeki olgular ile dolaylı olarak ilişkisi vardır (*Method_has_Parameter, Block_declares_LocalVariable* gibi).

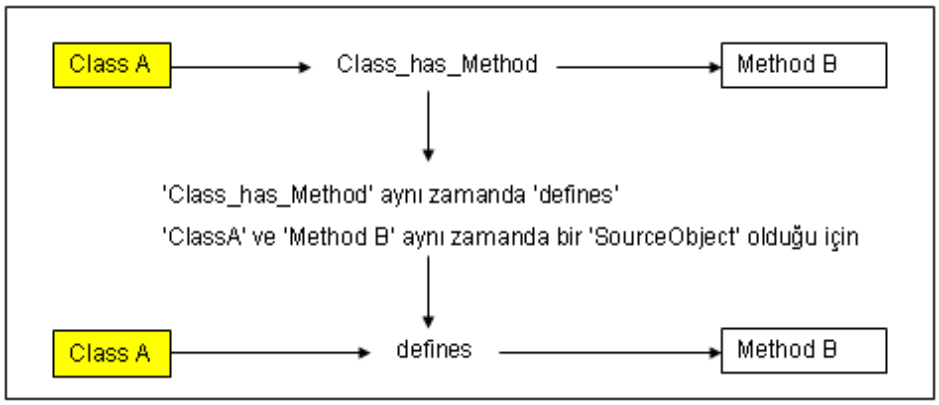
Sınıf ve değişkenler arasında doğrudan bir ilişki elde edebilmek için java kod ontolojisine birkaç ekleme yapılmıştır. İlk olarak, yukarıda belirtilen beş ilişkinin bir üst ilişkisi olacak şekilde, tanım ve erim kümesi *SourceObject* olan geçişken özelliğe sahip *defines* ilişkisi tanımlanmıştır.



Şekil 4.20. Bir üst OWL ilişkisi olarak tanımlanmış, geçişken (transitive) *defines* ilişkisi.

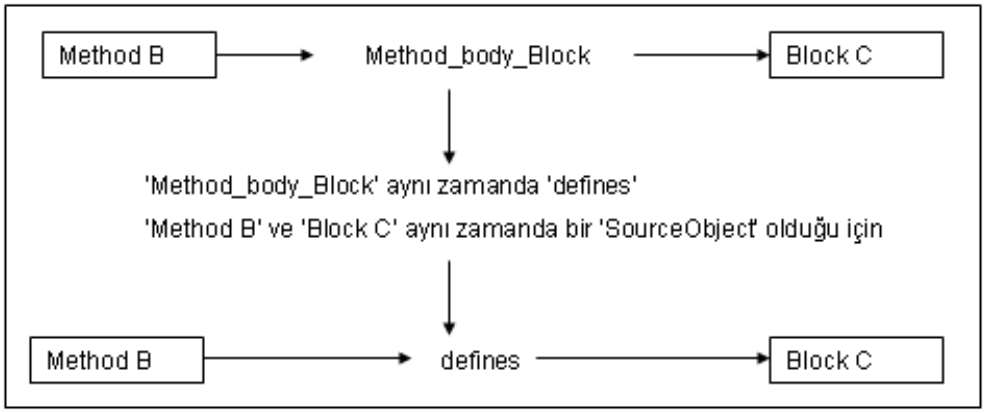
Belirtilen beş ilişkiden herhangi birini sağlayan olgu çifti, ontolojideki her olgu aynı zamanda *SourceObject* kavramının da bir olgusu olduğundan, *defines* ilişkisi de sağlayacaktır. *defines* ilişkisinin geçişken olarak tanımlanmış olması, ilişkinin sınıf olguları ve tüm değişkenler (sınıf alanı, yerel değişken ve parametreler) arasındaki dolaylı içermelerin tesbit edilebilmesine imkan sağlamıştır. Burada belirtilen özellik şekil 4.21, 4.22, 4.23 ve 4.24'te görsel olarak ifade edilmeye çalışılmıştır.

Şekil 4.21'de, Class A ve Method B olguları arasında *Class_has_Method* ilişkisinin bulunduğu görülmektedir. Yani Method B olgusunun temsil ettiği Java metodu, Class A olgusunun temsil ettiği Java sınıfı içerisinde tanımlanmıştır.



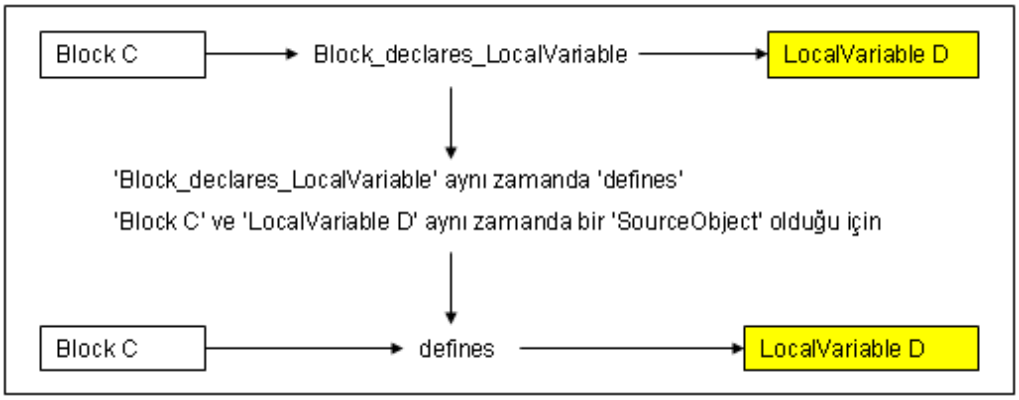
Şekil 4.21. Class A ve Method B olguları arasındaki ilişki.

Şekil 4.22'de, Method B olgusu ile temsil edilen Java metodu içerisinde, Block C olgusu ile temsil edilen bir blok yapısının (koşul, döngü gibi) tanımlandığı görülmektedir.



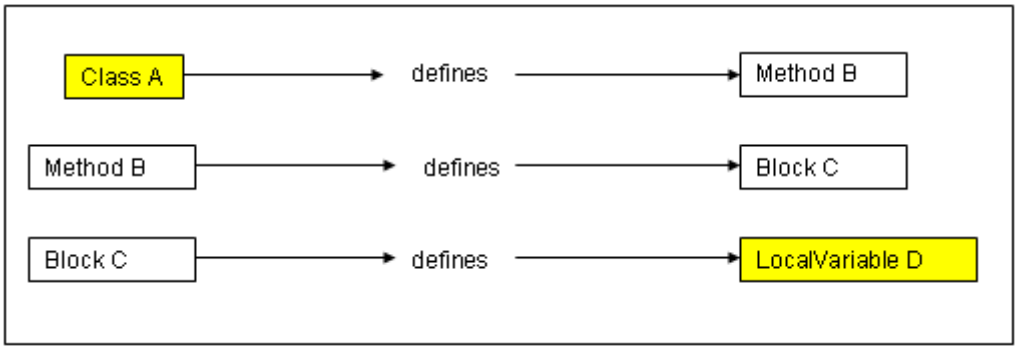
Şekil 4.22. Method B ve Block C olguları arasındaki ilişki.

Şekil 4.23'te, Block C olgusu ile temsil edilen blok yapısı içerisinde, LocalVariable D olgusu ile temsil edilen bir yerel değişkenin tanımlandığı görülmektedir.



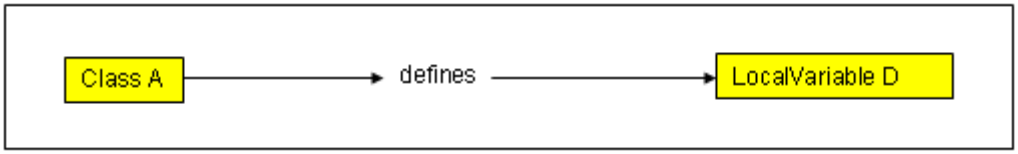
Şekil 4.23. Block C ve LocalVariable D olguları arasındaki ilişki.

Şekil 4.24'te, Şekil 4.21, 4.22 ve 4.23'te belirtilen üç durum birlikte ifade edilmiştir.



Şekil 4.24. Class A ve LocalVariable D olguları arasındaki ara ilişkilerin tümü.

Şekil 4.25'te, *defines* ilişkisinin geçişken olmasından faydalanılarak elde edilen sonuç görülmektedir. Buna göre kod bilgi tabanında, Class A olgusu ile LocalVariable D olgusu arasında doğrudan tanımlanmış yada beyan edilmiş (asserted) bir ilişki olmamasına rağmen, *defines* ilişkisinin geçişken olması ve bu iki olgu arasında dolaylı bir ilişkinin tanımlanmış olması (Method B ve Block C ile) Class A olgusu ile temsil edilen Java sınıfı ile LocalVariable D olgusu ile temsil edilen yerel değişken arasındaki ilişkinin çıkarsanmasını sağlamıştır.



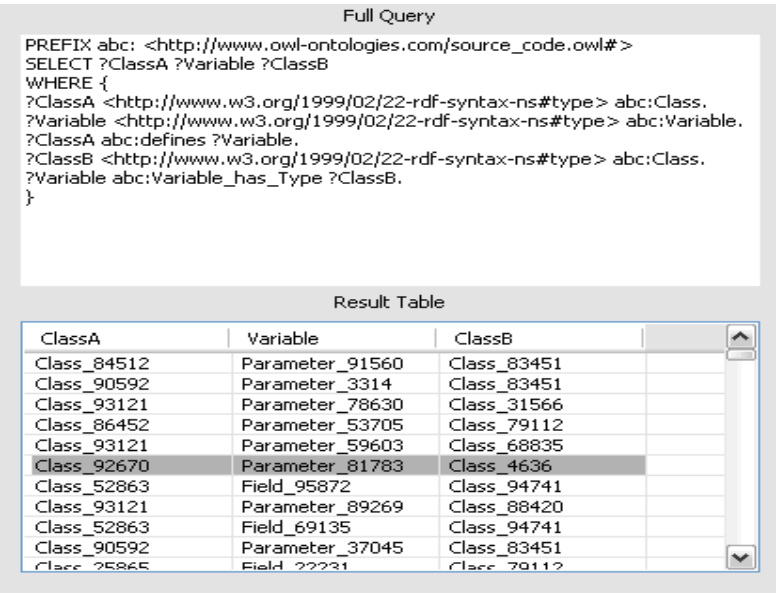
Şekil 4.25. Class A ve LocalVariable D olguları arasındaki çıkarsanan ilişki.

Verilen örnekte, herhangi bir Java sınıfı ile bu sınıf içerisinde tanımlanmış yerel değişkenlere çıkarsama yolu ile ulaşılması ele alınmıştır. Benzer şekilde, sınıflar içerisinde tanımlanmış metod parametrelerine de çıkarsama yolu ile ulaşılabilir.

ClassB: Herhangi bir Java sınıfı,

ClassA: ClassB türünde değişken içeren (Class B'ya değişken kullanımı ile bağımlı) herhangi bir Java sınıfı olmak üzere, proje kapsamındaki tüm ClassA – ClassB ikililerini listeleyecek olan sorgu aşağıda gösterilmiştir.

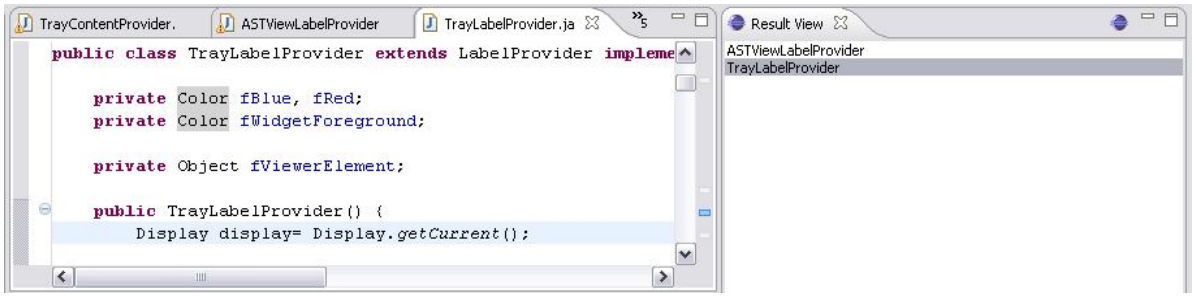
```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT ?ClassA ?Variable ?ClassB
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?Variable <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Variable.
?ClassA abc:defines ?Variable.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?Variable abc:Variable_has_Type ?ClassB.
}
```



Şekil 4.26. Sınıflar arasındaki değişken kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi.

Şekil 4.26'da gösterilen sorguyu, bağımlı olunan Java sınıfı bağlamında özelleştirecek olursak, oluşan sorgu aşağıdaki gibi olacaktır.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?Variable <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Variable.
?ClassA abc:defines ?Variable.
?Variable abc:Variable_has_Type ?ClassB.
?ClassB abc:Class_name 'Color'.
}
```



Şekil 4.27. 'Color' Java sınıfına yerel değişken kullanımı ile bağımlı olan tüm sınıfların tesbiti için Eclipse sorgu görünümünde oluşan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi.

4.4.4.2. Metod Kullanımı (Method Use)

C1 sınıfı içerisinde tanımlanmış bir metodun, C2 sınıfı içerisinde tanımlanmış bir metodu çağırması durumudur. Belirtilen durumun tesbitinde kullanılabilecek olan ve Java kod ontolojisinde tanımlı olan ilişkiler şunlardır:

- *Method* ve *Block* kavramları arasında tanımlanmış olan *Method_body_Block* ilişkisi
- *Block* ve *MethodInvocation* kavramları arasında tanımlanmış olan *Block_methodInvocation_MethodInvocation* ilişkisi

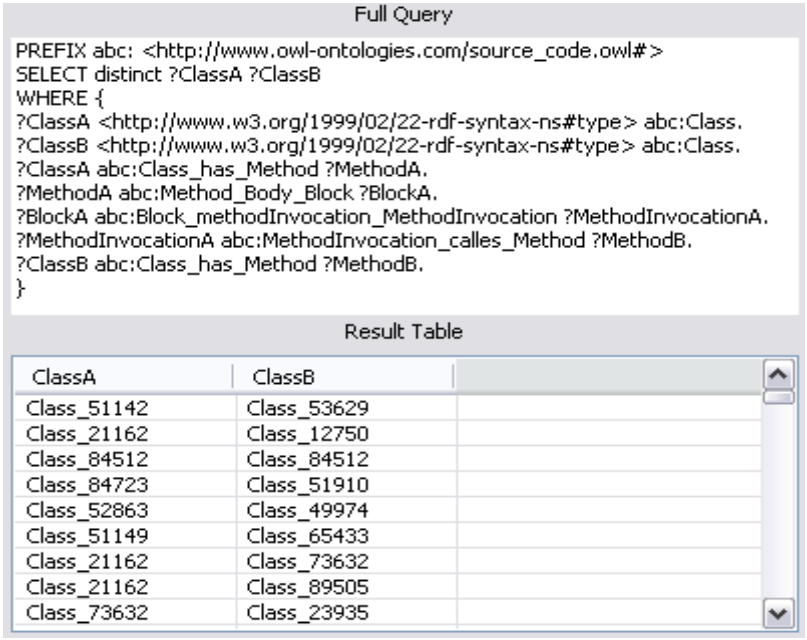
- *MethodInvocation* ve *Method* kavramları arasında tanımlanmış olan *MethodInvocation_calles_Method* ilişkisi

ClassB: Herhangi bir Java sınıfı,

ClassA: ClassB sınıfı içerisinde tanımlanmış herhangi bir methodu kullanan (Class B'ya method kullanımı ile bağımlı) herhangi bir Java sınıfı,

olmak üzere, proje kapsamındaki tüm ClassA – ClassB ikililerini listeleyecek olan sorgu aşağıda gösterilmiştir.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA ?ClassB
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassA abc:Class_has_Method ?MethodA.
?MethodA abc:Method_Body_Block ?BlockA.
?BlockA abc:Block_methodInvocation_MethodInvocation ?MethodInvocationA.
?MethodInvocationA abc:MethodInvocation_calles_Method ?MethodB.
?ClassB abc:Class_has_Method ?MethodB.
}
```



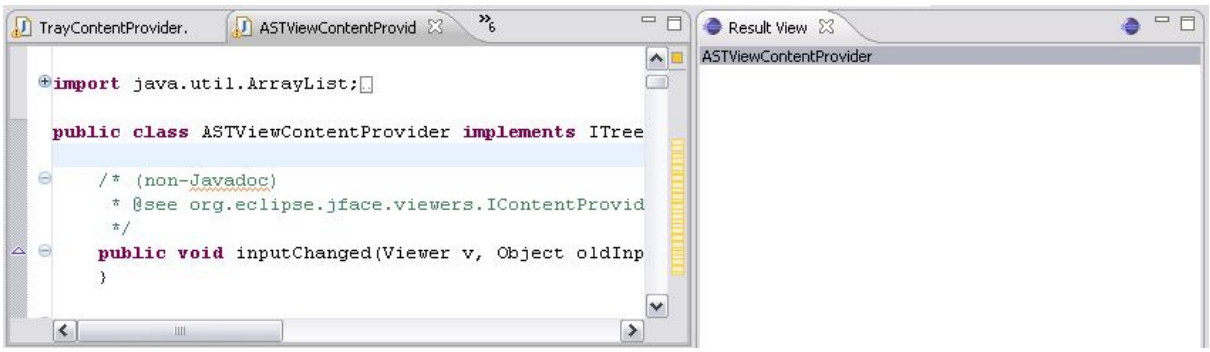
Şekil 4.28. Sınıflar arasındaki metod kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi.

Şekil 4.28'de gösterilen ve *Class_has_Method* ve *Method_Body_Block* ilişkilerini kapsayan (üst ilişki) geçişken *defines* ilişkisi kullanılarak daha kısa olarak aşağıdaki gibi yazılabilir.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA ?ClassB
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassA abc:defines ?BlockA.
?BlockA abc:Block_methodInvocation_MethodInvocation ?MethodInvocationA.
?MethodInvocationA abc:MethodInvocation_calles_Method ?MethodB.
?ClassB abc:Class_has_Method ?MethodB.
}
```

Şekil 4.28'deki sorguyu, method kullanımı ile bağımlı olunan Java sınıfı bağlamında özelleştirecek olursak, oluşan sorgu aşağıdaki gibi olacaktır:

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassA abc:defines ?BlockA.
?BlockA abc:Block_methodInvocation_MethodInvocation ?MethodInvocationA.
?MethodInvocationA abc:MethodInvocation_calles_Method ?MethodB.
?ClassB abc:Class_has_Method ?MethodB.
?ClassB abc:Class_name 'MethodRef'.
}
```



Şekil 4.29. *MethodRef* Java sınıfına metod kullanımı ile bağımlı olan diğer sınıfların tesbiti için Eclipse sorgu görünümünde oluşturulan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi.

4.4.4.3. Sınıf Alanı Kullanımı (Field Use)

C1 içerisinde tanımlanmış bir methodun, C2 içerisinde tanımlanmış bir sınıf alanına okuma ya da yazma yapması durumudur. Belirtilen durumun tesbitinde kullanılabilecek olan ve Java kod ontolojisinde ilk olarak tanımlanmış ilişkiler şunlardır:

- *Block* ve *FieldAccess* kavramları arasında tanımlanmış olan *Block_accessField_FieldAccess* ilişkisi (*FieldAccess* kavramı, bloklar içerisinde sınıf alanlarına erişim yapılan durumları temsil eden olguları içerir.)

- *FieldAccess* ve *Field* kavramları arasında tanımlanmış olan *FieldAccess_declaredField_Field* ilişkisi

ClassB: Herhangi bir Java sınıfı,

ClassA: ClassB sınıfı içerisinde tanımlanmış bir sınıf alanına erişim yapan (Class B'ye sınıf alanı kullanımı ile bağımlı) herhangi bir Java sınıfı olmak üzere, proje kapsamındaki tüm ClassA – ClassB ikililerini listeleyecek olan sorgu aşağıda gösterilmiştir.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA ?ClassB
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassA abc:Class_has_Method ?MethodA.
?ClassB abc:Class_has_Field ?FieldA.
?MethodA abc:Method_Body_Block ?BlockA.
?BlockA abc:Block_AccessField_Field_Access ?FieldAccessA.
?FieldAccessA abc:FieldAccess_DeclaredField_Field ?FieldA.
}
```

Full Query

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT distinct ?ClassA ?ClassB
WHERE {
?ClassA <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassB <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> abc:Class.
?ClassA abc:Class_has_Method ?MethodA.
?ClassB abc:Class_has_Field ?FieldA.
?MethodA abc:Method_Body_Block ?BlockA.
?BlockA abc:Block_AccessField_Field_Access ?FieldAccessA.
?FieldAccessA abc:FieldAccess_DeclaredField_Field ?FieldA.
}
```

Result Table

ClassA	ClassB
Class_40258	Class_40258
Class_73632	Class_73632
Class_89505	Class_51910
Class_51149	Class_51149
Class_23935	Class_23935
Class_81842	Class_81842
Class_22880	Class_51910
Class_46245	Class_16548
Class_46245	Class_46245
Class_53752	Class_53752

Şekil 4.30. Sınıflar arasındaki sınıf alanı kullanım bağımlılığının tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi.

4.4.4.4. Kalıtım Yolu ile Kullanım (Inheritance Use)

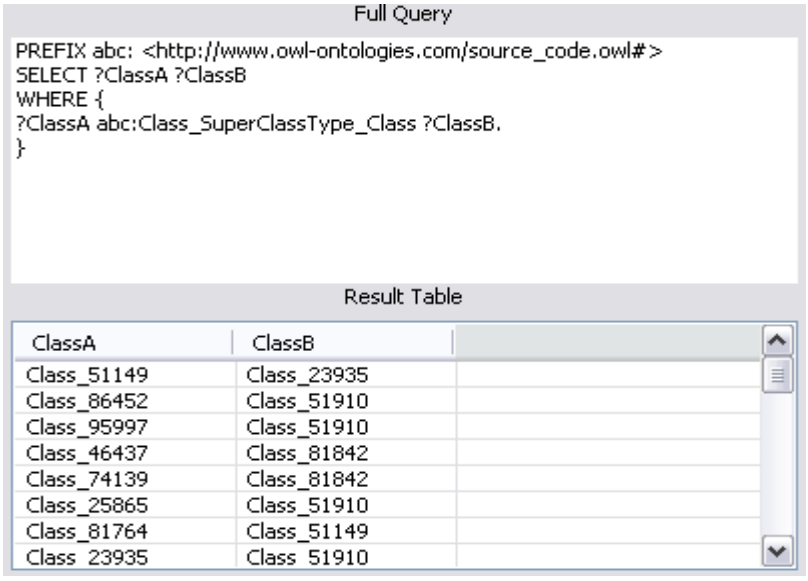
C1 sınfının C2 sınıfından türemesi durumudur. Belirtilen durumun tesbitinde kullanılabilecek olan ve Java kod ontolojisinde tanımlanmış ilişki:

- Tanım ve erim kümesi Class olan *Class_superClassType_Class* ilişkisi

ClassB: Herhangi bir Java sınıfı,

ClassA: ClassB'den türemiş olan (Class B'ye kalıtım yolu ile bağımlı) herhangi bir Java sınıfı olmak üzere, proje kapsamındaki tüm ClassA – ClassB ikililerini listeleyecek olan sorgu aşağıda gösterilmiştir.

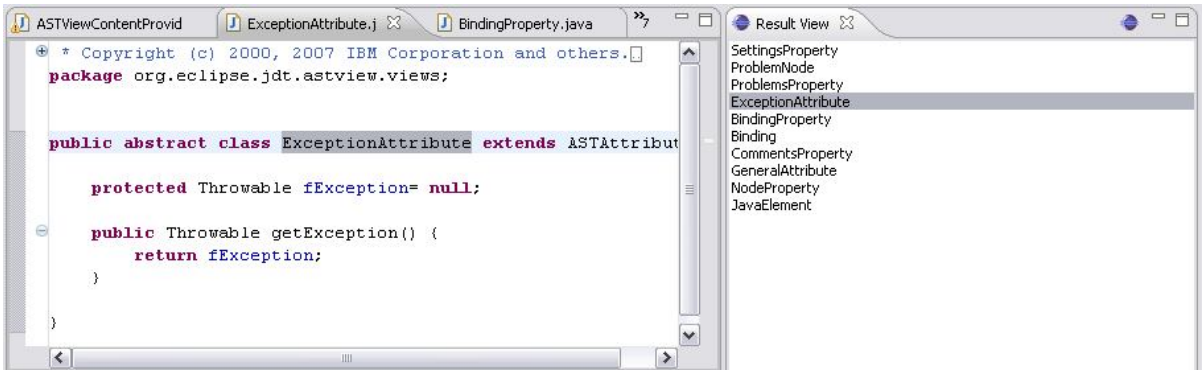
```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT ?ClassA ?ClassB
WHERE {
?ClassA abc:Class_SuperClassType_Class ?ClassB.
}
```



Şekil 4.31. Sınıflar arasındaki kalıtım yolu ile bağımlılığın tesbiti için SWT arayüzünde oluşturulan sorgu ve sonuç listesi.

Aynı sorguyu, kalıtım yolu ile bağımlı olunan Java sınıfı bağlamında özelleştirecek olursak, oluşan sorgu aşağıdaki gibi olacaktır.

```
PREFIX abc: <http://www.owl-ontologies.com/source_code.owl#>
SELECT ?ClassA ?ClassB
WHERE {
?ClassA abc:Class_SuperClassType_Class ?ClassB.
?ClassB abc:Class_name 'ASTAttribute'.
}
```



Şekil 4.32. *ASTAttribute* Java sınıfına kalıtım yolu ile bağımlı olan diğer sınıfların tesbiti için Eclipse sorgu görünümünde oluşturulan sorgu sonuç listesi ve sonuçların editör üzerinde görüntülenmesi.

Örnekteki *Class_SuperClassType_Class* ilişkisi geçişken olarak tanımlanmadığı için yukarıdaki sorgu sonucunda, ismi belirtilen sınıftan sadece doğrudan türemiş sınıflar sonuç listesinde yer alacaktır. Aynı ilişkiye geçişken özellik kazandırıldığında, ismi belirtilen sınıftan hem dolaylı hem de doğrudan türemiş sınıflar sonuç listesinde yer alacaktır.

5. SONUÇ VE ÖNERİLER

Kaynak kodun sorgulanabilirliği, yazılımın anlaşılabilmesinde ve yazılım üzerinde bazı analizlerin gerçekleştirilebilmesinde kullanılabilecek önemli bir gereksinimdir.

Bu çalışmada önceki çalışmalardan farklı olarak, kod bilgilerinin saklanmasında OWL-DL ile gösterilmiş ontoloji, sorgulanmasında ise SPARQL ve Pellet çıkarım motoru kullanılmıştır.

OWL-DL ile gösterilmiş ontoloji kullanımı ile nesneye yönelik bir programlama dili olan Java için temel kaynak kod kavramları (paket, sınıf, metod gibi) ve birbirleri arasındaki ilişkileri etkin bir şekilde ifade edilebilmiştir. Ayrıca ontoloji kullanımı ile Java kod bilgileri sorgulanabilir bir model üzerine oturtulmuştur. Geliştirilen kod ontolojisi, gereksinimler doğrultusunda değiştirilebilir ve yeni özellikler eklenebilir durumdadır.

OWL-DL'nin sağladığı geçişkenlik özelliği ile basit bir şekilde özyineleme gerektiren sorguların yazılabildiği görülmüştür.

Geliştirilen sorgulama arayüzü ile doğrudan sorgu yazmadan, sadece seçim yaparak ve gerekli parametreler girilerek SPAQL sorguları oluşturulabilmektedir. Sorgulama arayüzünde oluşturulan sorguların Eclipse'teki sorgu görünümüne kopyalama-yapıştırma yöntemi ile aktarılması, kullanımı zorlaştıran bir durumdur.

Geliştirilen aracın performans ve kullanım kolaylığı bakımından geliştirilmesi için önerilen çalışmalar şunlardır:

- Sorgulama arayüzü Eclipse geliştirme ortamına entegre edilebilir.
- Kod ayrıştırıcısı için iyileştirme yapılarak, ayrıştırma işlemi daha hızlı hale getirilebilir.
- Ayrıştırma işleminin el ile başlatılabilmesinin yanında, gerçek zamanlı olarak (kod değişikliklerinde Eclipse tarafından tetiklenmesi) kod değişiklikleri için, değişen kod kesimlerinin tekrar ayrıştırılması özelliği eklenebilir.

- Ayrıştırıcının ele aldığı özellik (kod metrikleri, fırlatılan hatalar) sayısı arttırılarak, kod hakkında daha yüksek çözünürlükte bilginin modellenmesi sağlanabilir.

KAYNAKLAR

[1] Önder Keskin, Ebru Sezer. Kaynak Kod Sorgulamada Ontoloji Kullanımı. Ulusal Yazılım Mühendisliği Sempozyumu, 2009.

[2] Brachman, R. J., Devanbu, P., Selfridge, P. G., Belanger, D., & Chen, Y. (1990). Toward a Software Information System. AT&T Technical Journal, 69(2), pp. 22-41.

[3] S. E. Sim, C. L. A. Clarke & R. C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," International Workshop on Program Comprehension (1998).

[4] Bajracharya, S., Ngo, T., Linstead, E., Rigor, P., Dou, Y., Baldi, P. ve Lopes, C. Sourcerer: A Search Engine for Open Source Code. International Conference on Software Engineering (ICSE 2007).

[5] Internet: http://www.eclipse.org/

[6] Internet: http://msdn.microsoft.com/en-us/vstudio/default.aspx

[7] Internet: http://www.w3.org/RDF/

[8] Internet: http://www.w3.org/TR/owl-ref/

[9] Internet: http://www.w3.org/TR/rdf-sparql-query/

[10] Internet: http://clarkparsia.com/pellet/

[11] Internet: http://www.eclipse.org/swt/

[12] Internet: http://www.eclipse.org/jdt/ui/astview/index.php

[13] Internet: http://www.eclipse.org/jdt/core/

[14] T. Berners-Lee, J. Hendler, ve O. Lassila, "The Semantic Web," Scientific Am., May 2001, sayfa 34–43.

[15] Internet: http://en.wikipedia.org/wiki/File:Semantic-web-stack.png

[16] Kaan Kurtel , “Web’in geleceği: Anlamsal Web”, Ege Akademik Bakış, (1)2008: 205-213.

[17] Internet: http://www.w3.org/

[18] Deconta, M.C.; Obrst, J.L.; Smith T.K. (2003): The Semantic Web, Wiley.

[19] T.R. Gruber, “A Translation Approach to Portable Ontologies,” Knowledge Acquisition, vol. 5, no. 2, 1993, sayfa 199–220.

[20] Ontological Engineering: with examples from Knowledge Management, e-Commerce, and the Semantic Web”, Springer, 2004.

[21] Internet: http://www.w3.org/TR/rdf-primer/

[22] Internet: http://www.w3schools.com/rdf/rdf_schema.asp/

[23] Dean Allemang, James Hendler, “Semantic Web for the Working Ontologist”, Morgan Kaufmann, 2008.

[24] Davies J., Studer R. & Warren P. (2006) Semantic Web Technologies: trends and research in ontology-based systems. John Wiley & Sons Ltd., Chichester, sayfa 4.

[25] Murat Osman Ünalır, “Ontolji Mühendisliği Eğitim Semineri notları”, Ulusal Yazılım Mühendisliği Sempozyumu, 2009.

[26] Internet: http://en.wikipedia.org/wiki/Web_Ontology_Language

[27] Internet: http://www.w3.org/Submission/RDQL/

[28] Internet: http://articles.techrepublic.com.com/5100-10878_11-6097805.html

[29] Internet: http://www.xml.com/pub/a/2005/07/20/versa.html

[30] Internet: http://www.cambridgesemantics.com/2008/09/sparql-by-example/

[31] Internet: http://www.daml.org/2003/01/periodictable/PeriodicTable#

[32] Internet: http://www.xml.com/pub/a/2005/11/16/introducing-sparql-querying-semantic-web-tutorial.html

[33] R.J. Brachman and J. Schmolze, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Sci* 9(2), 1985.

[34] Internet: http://www.isi.edu/isd/LOOM/LOOM-HOME.html

[35] Alexander Borgida , Ronald J. Brachman , Deborah L. McGuinness , Lori Alperin Resnick, CLASSIC: a structural data model for objects, Proceedings of the 1989 ACM SIGMOD international conference on Management of data, p.58-67, June 1989, Portland, Oregon, United States.

[36] V. Haarslev and R. Moeller. Racer: A core inference engine for the Semantic Web. In 2nd International Workshop on Evaluation of Ontology-based Tools (EON-2003), Sanibel Island, FL, 2003.

[37] E. Bozsak et al. Kaon - towards a large scale semantic web. In *Proceedings of EC-Web*, pages 304–313, Aix-en-Provence, France, 2002. LNCS 2455 Springer.

[38] Internet: http://en.wikipedia.org/wiki/Description_logic

[39] Yonggang Zhang, An Ontology-Based Program Comprehension Model, Phd Thesis.

[40] Internet: http://www.google.com/codesearch

[41] Internet: http://www.krugle.org

[42] Internet: http://www.koders.com

[43] Renuka Sindhgatta, Using an information retrieval system to retrieve source code samples, Proceeding of the 28th international conference on Software engineering, May 20-28, 2006, Shanghai, China.

[44] Internet: http://lucene.apache.org/java/docs/

[45] Mark A. Linton. Implementing relational views of programs. In Peter B. Henderson, editor, Software Development Environments (SDE), sayfa 132–140, 1984.

[46] Internet: http://en.wikipedia.org/wiki/Ingres_(database)

[47] Internet: http://en.wikipedia.org/wiki/QUEL_query_languages

[48] Yih Chen, Michael Nishimoto, ve C. V. Ramamoorthy. The C information abstraction system. IEEE Transactions on Software Engineering, 16(3):325–334, 1990.

[49] Ebert, J., Kullbach, B., & Winter, A. (1999). GraX - An Interchange Format for Reengineering Tools, Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99), pp. 89-98.

[50] Lange, C., Sneed, H. M., & Winter, A. (2001). Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools, Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC'01), pp. 209-218.

[51] Internet: http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/graph-repository-query-language-greql

[52] Shahram Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Tsuyoshi Ohira, Kazu Yasuda, Kazushi Kuse, Tsutomu Kamimura, ve Richard Helm. Architecture of the XL C++ browser. In CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, sayfa 369–379. IBM Press, 1992.

[53] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In USENIX Conference on Domain-Specific Languages, sayfa 229–242, 1997.

[54] Doug Janzen ve Kris de Volder. Navigating and querying code without getting lost. In 2nd International Conference on Aspect-Oriented Software Development, sayfa 178–187, 2003.

[55] The TyRuBa metaprogramming system. http://tyruba.sourceforge.net/.

[56] E. Hajiyev, M. Verbaere, ve O. de Moor. CodeQuest: scalable source code queries with Datalog. In D. Thomas, editor, Proceedings of ECOOP, volume 4067 of Lecture Notes in Computer Science, sayfa 2–27. Springer, 2006.

[57] Vijayan Sugumaran, Veda C. Storey, A semantic-based approach to component retrieval, ACM SIGMIS Database, v.34 n.3, p.8-24, Summer 2003.

[58] Internet: http://www.racer-systems.com/technology/contributions/2001/HaMo01e.pdf

[59] Haarslev, V., Moeller, R., Wessel, M.: Querying the semantic web with racer + nrql. In Bechhofer, S., Haarslev, V., Lutz, C., Moeller, R., eds.: CEUR Workshop Proceedings of KI-2004 Workshop on Applications of Description Logics (ADL 04), Ulm, Germany (2004).

[60] Internet: http://jena.sourceforge.net/

[61] Internet: http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

[62] Internet: http://protege.stanford.edu/

ÖZGEÇMİŞ

Adı Soyadı : Önder Keskin

Doğum Yeri : Eskişehir

Doğum Yılı : 1984

Medeni Hali : Bekar

Eğitim ve Akademik Durumu:

Lise 1998-2002 Eskişehir Kılıçoğlu Anadolu Lisesi

Lisans 2002-2007 Anadolu Üniversitesi Bilgisayar Mühendisliği Bölümü

Yabancı Dil: İngilizce

İş Tecrübesi:

2007-2009 Hacettepe Üniversitesi Bilgisayar Mühendisliği Bölümü
Araştırma Görevlisi

2009-... İnnova Bilişim Çözümleri A.Ş.

Yazılım Uzmanı