İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

NESNEYE DAYALI YAZILIMLARI SERVİS ODAKLI MODÜLLERE AYRIŞTIRMA İÇİN ÖĞRENME TABANLI BİR YÖNTEM

DOKTORA TEZİ

Ural ERDEMİR

Bilgisayar Bilimleri Anabilim Dalı

Bilgisayar Bilimleri Programı

EKİM 2014

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

NESNEYE DAYALI YAZILIMLARI SERVİS ODAKLI MODÜLLERE AYRIŞTIRMA İÇİN ÖĞRENME TABANLI BİR YÖNTEM

DOKTORA TEZİ

Ural ERDEMİR
(704062004)

Bilgisayar Bilimleri Anabilim Dalı

Bilgisayar Bilimleri Programı

Tez Danışmanı: Doç.Dr. Feza BUZLUCA

EKİM 2014

İTÜ, Bilişim Enstitüsü'nün 704062004 numaralı Doktora Öğrencisi **Ural ERDEMİR**, ilgili yönetmeliklerin belirlediği gerekli tüm şartları yerine getirdikten sonra hazırladığı **"NESNEYE DAYALI YAZILIMLARI SERVİS ODAKLI MODÜLLERE AYRIŞTIRMA İÇİN ÖĞRENME TABANLI BİR YÖNTEM"** başlıklı tezini aşağıda imzaları olan jüri önünde başarı ile sunmuştur.

Tez Danışmanı :	**Doç.Dr. Feza BUZLUCA** İstanbul Teknik Üniversitesi	.....
Jüri Üyeleri :	**Prof.Dr. Nadia ERDOĞAN** İstanbul Teknik Üniversitesi	.....
	Doç.Dr. Banu DİRİ Yıldız Teknik Üniversitesi	.....
	Prof.Dr. Oya KALIPSIZ Yıldız Teknik Üniversitesi	.....
	Doç.Dr. Şule ÖĞÜDÜCÜ İstanbul Teknik Üniversitesi	.....

Teslim Tarihi : **31 Temmuz 2014**
Savunma Tarihi : **1 Ekim 2014**

Sevgili Aileme,

ÖNSÖZ

Mühendislik hayatım boyunca bilgi ve tecrübesinden faydalandığım ve bu tez çalışmasında birlikte çalışmaktan onur duyduğum değerli danışman hocam Sayın Doç.Dr. Feza BUZLUCA'ya, bilgi ve tecrübeleri ile çalışmalarıma yön veren değerli tez izleme komitesi üyelerim Sayın Prof.Dr. Nadia ERDOĞAN'a ve Sayın Doç.Dr. Banu DİRİ'ye, desteklerinden dolayı arkadaşlarım Dr. Umut TEKİN'e ve Sinan ESKİ'ye, katkılarından dolayı çalışmada incelediğimiz endüstriyel projelerin geliştirme ekibine ve hayatımın her safhasında benden yardımlarını ve desteklerini esirgemeyen sevgili aileme, candan teşekkürlerimi sunarım.

Temmuz 2014

Ural Erdemir
(Yüksek Bilgisayar Mühendisi)

İÇİNDEKİLER

Sayfa

KISALTMALAR

AST **:** Abstract Syntax Tree
CAM **:** Cohesion Among Methods Metric
CF **:** Coupling Factor Metric
CMMI : Capability Maturity Model Integration
CORBA **:** Common Object Requesting Broker Architecture
DIT **:** Depth of Inheritance Tree Metric
EJB **:** Enterprise Java Beans
IIOP **:** Internet Inter-ORB Protocol
ISO : International Organization for Standardization
JDT **:** Java Development Tools
LCOM **:** Lack of Cohesion Metric
LOC **:** Line of Code Metric
MHF **:** Method Hiding Factor Metric
NED **: N**on-Extreme Distribution
NOC **:** Number of Children Metric
NOM **:** Number of Method Metric
OMG **:** Object Management Group
OOP **:** Object Oriented Programming
RFC **:** Response For a Class Metric
RMI **:** Remote Method Invocation
SPICE : Software Process Improvement and Capability
SOA **:** Service Oriented Architecture
SOAP **:** Simple Object Access Protocol
UDDI **:** Universal Description, Discovery and Integration
UML **:** Unified Modeling Language
WMC **:** Weighted Method per Class Metric
WSDL **:** Web Service Definition Language
XMI **:** XML Metadata Interchange
XML **:** Extensible Markup Language

ÇİZELGE LİSTESİ

Sayfa

ŞEKİL LİSTESİ

Sayfa

NESNEYE DAYALI YAZILIMLARI SERVİS ODAKLI MODÜLLERE AYRIŞTIRMA İÇİN ÖĞRENME TABANLI BİR YÖNTEM

ÖZET

Günümüzde donanım hata oranları ve geliştirme maliyetleri azalırken, yazılım geliştirme ve bakım maliyetleri giderek artmaktadır. Birçok yazılım projesi, zaman ve bütçe kısıtlarını aşarak başarısızlıkla sonuçlanmaktadır. Araştırmalara göre yazılım geliştirme maliyetinin büyük çoğunluğunu ise yazılım bakım faaliyetleri oluşturmaktadır. Pratikte, herhangi bir yazılım sistemi hakkında bilgi edinmek yazılım bakımındaki en önemli gereksinimdir. Ancak çoğu zaman, yazılım tasarım dokümanları, güncelliğini yitirmiş, kısmen ya da tamamen eksik durumdadır. Bununla beraber yazılımın zamanla evrilmesiyle mimari tasarımı da zamanla değişmekte ve başlangıçta bulunduğu şeklinden tamamen farklı bir hale dönüşebilmektedir. Tüm bunlar yazılım sistemi anlamayı daha da zor ve maliyetli bir iş haline getirmektedir. Bu nedenle, yazılım sistemlerini daha kolay ve daha hızlı anlamayı sağlayacak yöntem, algoritma ve otomatik sonuç üreten araçlara gereksinim duyulmaktadır.

Yazılım geliştirme tecrübeleri, geride bıraktığımız yıllar içinde farklı geliştirme metodolojilerinin ortaya çıkmasına neden olmuştur. Her metodoloji geçişinde daha büyük ve daha karmaşık yazılımlara çözüm aranmaya çalışılmıştır. Karmaşıklık zamanla artarken, yazılım tasarımında daha geniş tanecikli yapılar kullanılarak karmaşık daha iyi yönetilmeye çalışılmıştır. Bu nedenle prosedürel dillerdeki fonksiyonun yanına, nesneye dayalı sistemlerde sınıf, sınıfın yanına da bileşen ve servis kavramları getirilmiştir. Yazılım sistemin mimarisi, yazılımın büyük parçalı yapılarından meydana gelir, sistemin bileşenlerini ve üst seviyede bu bileşenlerin etkileşimini tanımlar. Yazılım modülleri, davranışlarına ve verilerine bir arayüz üzerinden kontrollü erişim sağlayarak gerçeklemelerini saklarlar.

Yazılım mimarları ve geliştiriciler modüler, yeniden kullanılabilir yazılımlar üretmek için nesneye dayalı tasarım prensiplerine başvururlar. Bu nedenle nesneye dayalı sistemlerde modül adı verilen, belirli bir servise özel yazılım sınıf grupları ortaya çıkar. Bazı durumlarda yazılımın uygulama alanına göre, gerçek dünyada ilgili işler yapan nesneler grup oluşturduğu için, bunların yazılımdaki yansımaları da yazılım sınıf gruplarını oluşturabilir. Tasarım kalıplarının kullanılması programlama açısından, bu modüllerin daha esnek ve tekrar kullanılabilir olmasına yardımcı olur. Yazılımlardaki bu modülleri bulmak, yazılımı daha iyi anlaşılmasında, daha etkin bir şekilde yazılım mimarisini ortaya çıkarılmasında, servis adaylarının belirlenerek var olan sistemlerin servis odaklı mimariye geçirilmesinde ve bu servislerin bulut tabanlı platformlara ya da dağıtılmış sistemlere taşınmasında kritik önem taşımaktadır.

Bu tez çalışmasında nesneye dayalı sistemlerdeki servisleri belirlemek için özgün bir otomatik modül bulma yaklaşımı (**LBME**) sunulmuştur. Bu yaklaşım, statik analizlerle çıkarılan yapısal bağımlılıkları kullanan ve nesneye dayalı tasarım mimarilerinde rastlanan tasarım kalıpları ile tasarım prensiplerinden faydalanan,

öğrenmeye dayalı bir modül bulma yaklaşımdır. Önerilen yaklaşımda ilk olarak yazılım sistemi için düğümlerin sınıfları, ayrıtların ise sınıflar arası ilişkileri temsil ettiği ağırlıklı ve yönlü bir çizge oluşturulmaktadır. Ardından modülleri belirlemek için bu çizge üzerinde ağırlıklı kümeleme algoritmaları uygulanmaktadır. Kullanılan kümeleme algoritmaları nesneye dayalı sistemlerin çizge karakteristikleri göz önünde bulundurularak seçilmiştir. Çizge üzerindeki ayrıtların ağırlığı, ayrıtların modül içinde ya da modül dışında olma olasılıklarına göre belirlenmektedir. Bu konumsal olasılıkları mümkün olduğu kadar doğru şekilde belirmek için makine öğrenmesi tabanlı sınıflandırma sistemi kullanılmıştır ve bu sistem gerçek dünyadan nesneye dayalı referans bir sistem ile eğitilmiştir.

Bu bağlamda yazılımın analizinin yapılabildiği, otomatik modül bulma aracı geliştirilmiş ve sunulan özgün yaklaşım, çeşitli açık kaynaklı ve endüstriyel projeler üzerinde değerlendirilmiştir. Deneysel sonuçlar göstermiştir ki, sunulan yeni yaklaşım nesneye dayalı sistemler için gerçeğe çok yakın sonuçlar üretmekte ve literatürde bulunan mevcut metotlardan daha iyi başarım göstermektedir.

A LEARNING-BASED METHOD FOR EXTRACTING SERVICE-ORIENTED MODULES IN OBJECT-ORIENTED SOFTWARE

SUMMARY

Studies show that up to 75 percent of the total development cost is spent on maintenance activities through the full software development lifecycle. In practice, acquiring knowledge about a software system is the most crucial need in software maintenance. In most cases, documentation of software design is outdated, incomplete or absent, which make software systems difficult to understand. Therefore, automated tools and support of algorithms are required to understand software systems more quickly and easily.

Developers apply object-oriented (OO) design principles to produce modular, reusable software. Therefore, service-specific groups of related software classes called modules arise in OO systems. Extracting the modules is critical for better software comprehension, efficient architecture recovery, determination of service candidates to migrate legacy software to service-oriented architecture, and transportation of such services to distributed systems or cloud-based platforms. Correctly identifying software modules is also important for software evolution processes because with the increasing popularity of distributed systems and cloud computing, service-oriented architecture (SOA) and migration to SOA have received further stimulus. Developers attempt to identify modules in a software system to transform the underlying services into standard, independent and distributed services. Distributing these services by transporting them into cloud-based service-oriented platforms properly can improve the performance, reliability and security of the programs. In addition, service modules are reusable parts of the system that can be collected in libraries to use in future projects. Identifying software services automatically saves time and makes easier to ensure all these mentioned benefits to the software developers.

In this study, we propose a learning-based module extraction method for object-oriented (OO) systems **(LBME)** that considers the modular structure of the OO software. Studies on modularization in the literature generally represent an OO software system as a dependency matrix or a dependency graph comprised of classes and their relations. Clustering algorithms have been widely applied on the software graphs for automated architecture recovery. Modularization methods try to define and optimize a modularity objective function by some criteria. These criteria are based on the assumption that modules should be highly cohesive and loosely coupled; therefore, the density of the intra-module (internal) edges should be high, and inter-module (external) edge density should be low.

Our module extraction approach does not depend only on the density of the edges, besides we also consider the characteristics of the classes and the attributes of the relations represented by the edges to discover the modules. The characteristics of the classes and their relations are determined by the OO principles that are applied during the design. The architects and developers aim to develop reusable and flexible

components to create maintainable, high-quality software systems. Hence, they follow or "at least try to" follow OO design principles and design patterns. We believe that such concerns strongly affect the modularity of an OO system. For example, *Facade* classes usually take place on the boundary of a module and they have low cohesion, low complexity, and high coupling. On the other hand, classes inside a module (inner classes) implement the functionality of the module (service) and therefore they can have more complex structures and high level of coupling to libraries. If such inner classes of the modules were written properly, they should be highly cohesive.

Clustering algorithms group some "strongly-connected" nodes in the graph and cut some "weak" edges to form the modules. To improve the accuracy of our algorithm we assign weights to the edges according their probability being internal or external respect to the modules in the system. To determine the weights properly we use two machine learning-based classifiers in the form of decision trees. Using the first decision tree we estimate the probability of the positions (internal/external) of the edges according to the attributes of the relations they represent, such as inheritance, message call, association, aggregation, uses, etc. Considering only the properties of the relations is not sufficient to determine the correct position of an edge. Therefore, using the second decision tree we also classify the source and destination nodes of the edges as "inner class" (in a module) or "border class" (at the border of a module) according to their types such as abstract, interface and metrics such as complexity cohesion, and coupling. Then we combine the results of these two classifiers and determine the probability of the position (internal/external) of the edge more accurately. At the last step, high weight values are assigned to the edges with a high probability of being internal and low weight values are assigned to edges that are more likely external. We train our classifiers with data that are gathered from a real-world software system that is examined in advance.

The reference software system has been developed in a modular structure considering OO principles and patterns. The train data contains the attributes of selected classes and edges, and their real positions in the reference project. After the train phase, the classifiers "learn" the properties of the modular OO design, so that they can classify the positions of new nodes and edges correctly depending on their characteristics. To the best of our knowledge, there has been no mature study reported using OO design features for module extraction.

After the learning phase the proposed module extraction method that contains three main steps is ready to use. In the first step, we build the complete graph model of the software architecture by extracting the software entities (i.e., classes and interfaces) and their relations (i.e., extend and method call) directly from the Abstract Syntax Tree (AST) of the software project. We represent the architecture of the system as a simple directed and labeled graph. In the second step, we assign weight values to the edges depending on the probabilities of the edges being in a module or between different modules. Here, we use decision tree-based classifiers that were trained with the reference project to determine the positions of edges in the graph. Finally, for finding modules in these weighted and directed graphs we apply the agglomerative hierarchical clustering algorithm Fast Community that suits the properties of OO systems as we have shown in a preliminary study.

We implemented an automatic module extraction tool and evaluated the proposed approach on several open source and industrial projects. We evaluated our approach

on different industrial OO software systems by working closely with the development team of the projects and on many popular open-source projects. We compared automatically extracted modules by our method, with the modules identified by the developers of the projects or by software experts. We observe that the modules are realistically (close to real modules) extracted. Experimental results show that the proposed approach is highly accurate in modular service extraction for OO systems and outperforms existing methods. The results also show that the proposed method outperforms previously published approaches in the terms of stability, and non-extremity of the module size distribution.

1. GİRİŞ

Günümüzde, bilgisayar donanımlarının üretim maliyetleri ve hata oranları giderek azalırken, yazılımların maliyetleri ve hata oranları giderek artmaktadır. Yazılımların boyutlarının büyümesiyle birlikte, yazılım bakım masrafları (maintenance cost) ve geliştirme zamanları da artmıştır. Bugün birçok yazılım projesi başarısızlıkla sonuçlanabilmektedir. Amerikan ordusunun yazılım projeleri üzerine yaptığı bir araştırmaya göre, yapılan yazılım projelerinin [1]:

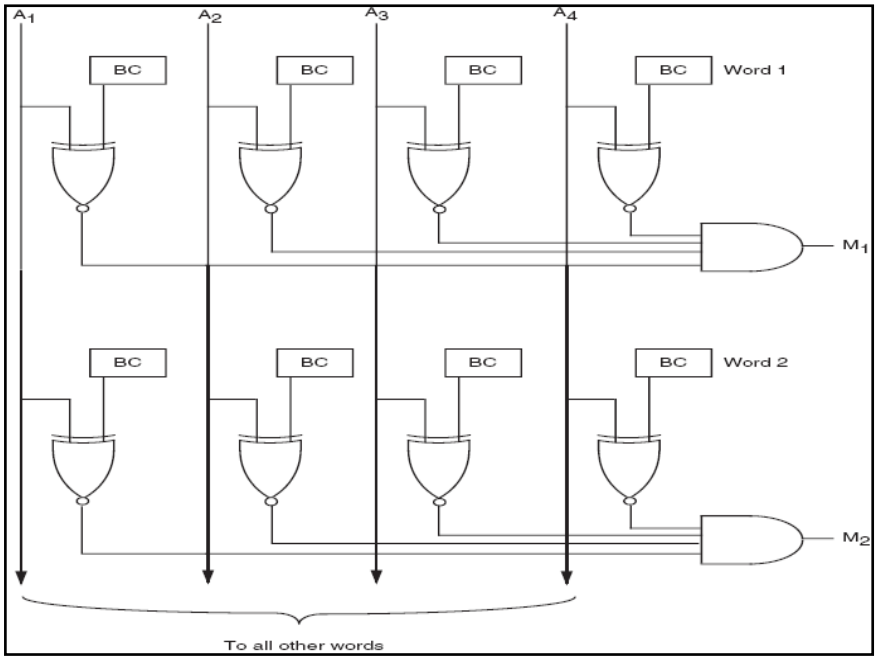
i. %47'si kullanılmamakta
ii. %29'u müşteri tarafından kabul edilmemekte
iii. %19'u başladıktan sonra iptal edilmekte ya da yeniden başlatılmakta
iv. %3'ü kimi değişikliklerden sonra kullanılmakta
v. %2'si ancak teslim alındığı gibi kullanılmakta olduğu görülmüştür.

2002 yılında NIST tarafından yapılan başka bir araştırmaya göre yazılım hatalarının Amerikan ekonomisine yıllık maliyeti 59 Milyar $ olarak tahmin edilmiştir ki o yıllarda satılan yazılımların toplam değeri 180 Milyar $ civarındaydı [2]. The Standish Group şirketinin son yıllarda yaptığı başka bir araştırmaya [3] göre ise büyük bilgi teknolojileri projelerinin %38'i kabul edilmeyecek seviyede başarısız bulunmakta, %52'si gecikmiş, bütçeyi aşmış, ya da istenilenden daha az işleve sahip olması nedeniyle yetersiz bulunmakta ve ancak %10'u başarılı bulunmaktadır.

Donanım üretim maliyetleri ve hata oranları azalırken yazılım maliyetlerinin ve hata oranlarının giderek artmasının nedeni temelde yazılım ve donanım sistemlerinin mimari farklılıklarından kaynaklanmaktadır [1] (Şekil 1.1 ve Şekil 1.2).

Donanım mimarilerinde:

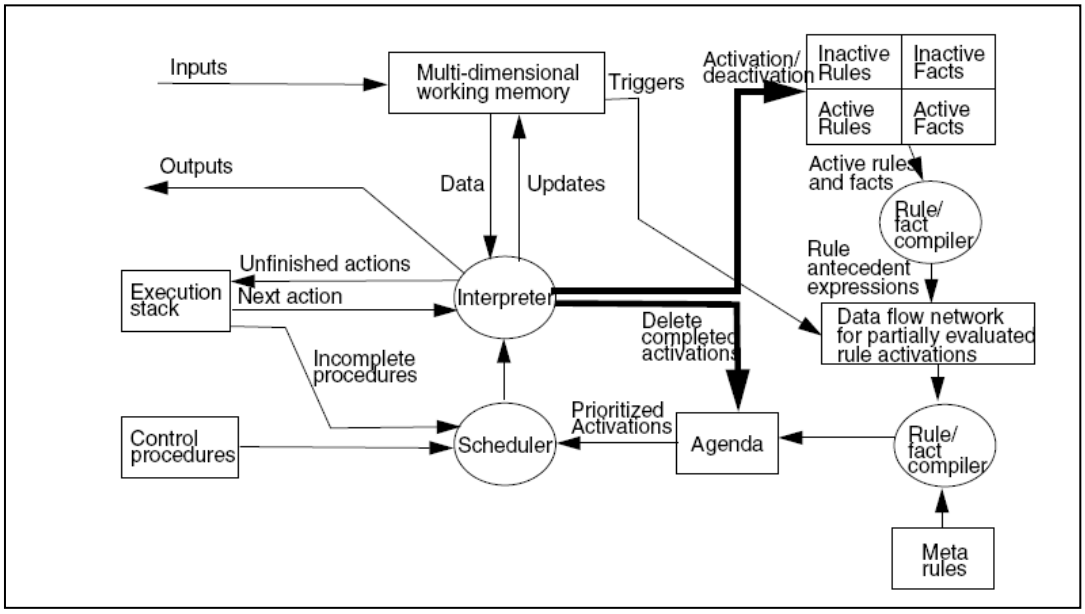
- Küçük sayısal alt sistemlerin çok sayıda kopyası vardır.
- Alt sistemler derinlemesine analiz/test edilmiştir.
- Tasarım hataları nadirdir.



Şekil 1.1 : Donanım mimarileri [1].

Yazılım mimarileri ise:

- Geniş ayrık durumlu alt sistemlerin az sayıda kopyasından oluşur.
- Tekrarlı yapı sayısı azdır.
- Derinlemesine analiz/test edilmesi zordur.
- Doğru yapıyı bulmak zordur.



Şekil 1.2 : Yazılım mimarileri [1].

Çalışmalar göstermektedir ki tüm yazılım yaşam döngüsü içerisinde toplam geliştirme maliyetinin %75'ine kadar olan bölümü yazılım bakım aktivitelerinde harcanmaktadır [4], [5]. Pratikte yazılım sistemi hakkında bilgi edinmek yazılım bakımında en kritik ihtiyaçtır [6], [7]. Ancak birçok durumda yazılım tasarım dokümanları, güncelliğini yitirmiş, kısmen ya da tamamen eksik durumdadır. Bu durum yazılımı anlamayı daha da güçleştirmektedir [8]. Bu nedenle yazılım sistemlerini daha hızlı ve daha kolay anlayabilmek için otomatikleştirilmiş araçlara ve algoritma desteğine ihtiyaç vardır. Yazılım sistemini anlamanın ana aşamalarından biri yazılım mimarisinin büyük resmini anlamak ve bu büyük resim içindeki modül ve servisleri belirleyebilmektir.

Yazılım modüllerini doğru bir şekilde belirleyebilmek yazılım gelişim evreleri için de önemlidir. Çünkü dağıtılmış sistemler ve bulut bilişimin artan popülerliği, servis odaklı mimari (Service Oriented Architecture [SOA]) ve servis odaklı mimariye geçiş (SOA migration) konularına olan talepleri daha da arttırmıştır. Yazılım geliştiriciler, yazılımın temelinde olan servisleri, standart bağımsız ve dağıtılmış servisler haline getirebilmek için, yazılım sistemi içindeki modülleri belirlemeye çalışırlar. Bu servislerin, bulut tabanlı servise dayalı platformlara dağıtılması sayesinde programların başarımı, güvenirliliği ve güvenliği arttırılabilir. Ayrıca bu servis modüleri, sistemin yeniden kullanılabilir parçalarıdırlar ve kütüphanelerde toplanmalarıyla, ileriki projelerde tekrar kullanılabilirler. Yeniden kullanım doğal olarak yazılım geliştirme şirketlerinin geliştirme maliyetlerini de düşürür. Bir sonraki alt bölümde yazılımda modüllerlik ihtiyacı, servis odaklı mimari ve yazılım modülerliğine etkileri kısaca anlatılmıştır.

1.1 Yazılımda Modüller ve Servis Odaklı Mimari

Bir sistemin yazılım mimarisi, yazılımın büyük parçalı yapılarından meydana gelir, sistemin bileşenlerini (Component) ve üst seviyede bu bileşenlerin nasıl etkileşimde bulunduğunu tarif eder. Bileşenlerin ve aralarındaki bağlantıların konfigürasyonu sistemin hem yapısal hem de davranışsal görüntüsünü oluşturur.

Geride bıraktığımız yıllar içinde yazılım geliştirme tecrübeleri çeşitli geliştirme metodolojilerinin ortaya çıkmasına neden olmuştur. Her metodoloji geçişinde daha karmaşık yazılımlara çözüm aranmaya çalışılmıştır. Karmaşıklık artmaya devam ederken, her seferinde daha geniş tanecikli yapılar (Örneğin fonksiyon, sınıf, bileşen

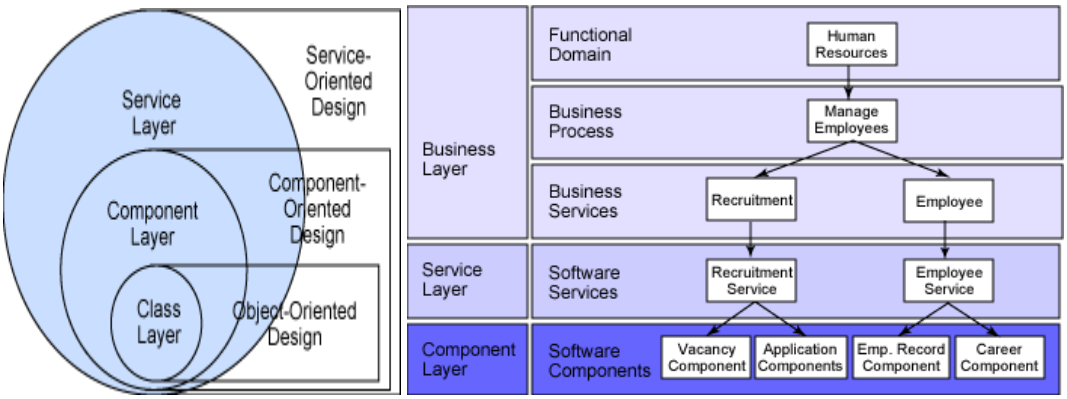
gibi...) kullanılarak karmaşıklık yönetilmeye çalışılmıştır. Bu yapılara yazılım modülleri ya da kara kutuları diyebiliriz.

Yazılım modülleri, davranışlarına ve verilerine bir arayüz (interface) üzerinden kontrollü erişim sağlayarak, iç yapılarındaki gerçeklemeleri diğer modüllerden saklarlar. Nesneye dayalı yazılımlarda uygun taneciklik (granularity) seviyesinde, davranışları ve verileri saklamak için nesneler kullanılır, aynı işi yapmak için bileşenler de kullanılır, bilgi saklamayı (information hiding) sadece nesne seviyesinde yapmak küçük sistemler için yeterlidir ve yazılımda gerçek dünya nesneleriyle eşleşen yapıların yaratılmasına izin verir. Burada problem çok sayıda nesnenin benzer amaçla gruplanmaya çalışılmasında ortaya çıkmaktadır. Nesnelere erişim arayüzleri üzerinden olmasına rağmen, nesne seviyesinde erişimin tanecikliği, büyük sistemlerde nesneler arasındaki bağımlılığın kontrolünü zorlaştırmaktadır. Bileşen (Component) kavramının ortaya atılmasıyla büyük sistemlerdeki bu bağımlılıkların daha iyi yönetilmesi sağlanmıştır. Bileşen, sistemin belirli bir işlevini yerine getirmek için daha küçük sayıdaki nesnelerin bir arada çalıştığı nesne gruplarıdır. Böylece bileşenler geniş sistem işlevleri düzeyinde başka bir "kara kutu" olarak karşımıza çıkmaktadır. Bileşenin içindeki nesneler sadece bileşen içindeki diğer nesneler tarafından bilinir, sistemin geri kalanına bu nesneler gösterilmez.

Günümüzde EJB, .NET ve CORBA gibi teknolojiler geliştiricilere bileşenleri gerçeklemenin etkin yollarını sunarlar. Bileşen tabanlı geliştirme metodolojisi, yazılım sistemi içindeki bağımlılıkların ve karmaşıklıkların daha iyi yönetilmesine izin verdiğinden, yazılım geliştiricilerin daha karmaşık, daha kaliteli sistemleri, daha hızlı oluşturabilmelerini sağlamaktadır. Ancak burada, bileşenler farklı teknolojiler kullanarak yapıldığında ve bu heterojen bileşenler bir arada kullanmak istenildiğinde ortaya çıkmaktadır. Örneğin, EJB metot çağrılarını RMI ile yaparken, CORBA IIOP'yi kullanmaktadır. Birçok bileşen Internet üzerinde, belki de bir güvenlik duvarının (firewall) arkasında bir yerde bulundurulmaktadır. Küçük tutarsızlıklar geliştiricilerin çözmesi gereken birçok soruna yol açabilir. Bu problemler olmasa bile bileşenlerin birbirleri hakkında çok fazla şeyi bilmesi gerekmesi de başka bir problemdir. Örneğin bir bileşenin kullanıcısı, o bileşen için hangi ulaşım ve taşıma tiplerini kullanacağını, aynı zamanda bileşenin kesin yerini de bilmesi gerekir. Ayrıca kullanıcının bileşenin arayüzü hakkında ayrıntılı bilgiye sahip olması gerekir ki arayüz değiştiğinde çağrı şeklini de değiştirebilsin. Bileşenler ağ adresli arayüzlere

sahip olduğundan ve oldukça fazla kullanıcısı olabileceğinden, bu durum çok geniş bir ölçekte bağımlılık oluşturur. Bu gibi problemler yazılım mimarisini de ciddi şekilde etkilemiş ve *Servis Odaklı Mimari (SOA)* kavramını ortaya çıkarmıştır. Yukarıda kısaca bahsedilen dezavantajlarıyla birlikte bileşen tabanlı modelleme sistemleri, SOA'ya geçiş için iyi bir başlangıç noktası olmaktadır. Servis, bir bileşen tarafından sağlanan ve diğer bileşenlerin arayüz sözleşmesine dayanarak kullandığı davranıştır. Servislerin ağ üzerinden adreslenebilir arayüzleri vardır. Servisler dinamik olarak keşfedilebilir ve kullanılabilirler.

Mevcut nesneye dayalı tasarımlarda olduğu gibi, iş modelleri soyutlamasını, sınıf seviyesine yapmanın tanecikliği, servis seviyesinde yapmaya göre daha düşük olmaktadır. Ayrıca kalıtım gibi sıkı bağlı ilişkiler de bağımlığı arttırmaktadır. Diğer yandan servis odaklı yaklaşım, esneklik ve çevikliği gevşek bağlı servislerle sağlamaya çalışmaktadır. Şekil 1.3 nesneye dayalı, bileşene dayalı ve servise dayalı sistemlerin görünürlük ve kapsama seviyelerini ve servis tanımlama hiyerarşisini göstermektedir.



Şekil 1.3 : Tasarım katmanları ve servis tanımlama hiyerarşisi [9].

Servis Odaklı Mimari güçlü bir servis katmanına sahiptir. Servis katmanındaki servisler ağ üzerinde çağrılabilirler. Servis arayüzlerini çağırmakta kullanılan teknolojilerin birlikte çalışabilirliğine önem verilmiştir. Aynı zamanda servis katmanındaki servislerin, konum şeffaflığına da önem verilmiştir. Böylece servisler dinamik olarak keşfedilebilir ve kullanılabilirler. Servis odaklı mimarinin en temel avantajları daha çok yeniden kullanılabilir, daha kolay bakımı yapılabilir daha ölçeklenebilir, daha güvenilir, daha güvenli, daha iyi test ve daha az hataya yatkın bir sistem mimarisi olmasıdır [9]:

Artan maliyetler nedeniyle yazılım şirketleri için yeniden kullanılabilirlik en önemli konulardan biri haline gelmiştir. Ancak dil ve platform uyumsuzlukları nedeniyle bunu başarmak oldukça zordur.

Ayrıca sınıfların tanecikliği ve sıkı bağlı olması da yeniden kullanabilirliklerini zorlaştırmaktadır. Oysa gevşek bağlı servislerin yeniden kullanılabilirliği çok daha kolaydır. Yazılım arkeolojisi [10] yazılımdaki hataların yerlerinin belirlenmesi ve düzeltilmesi işidir. İş akışının yeri olarak servis katmanına odaklanılması, hataların yerlerinin belirlenmesini ve düzeltilmesini kolaylaştıracağından, bakılabilirlik artacaktır.

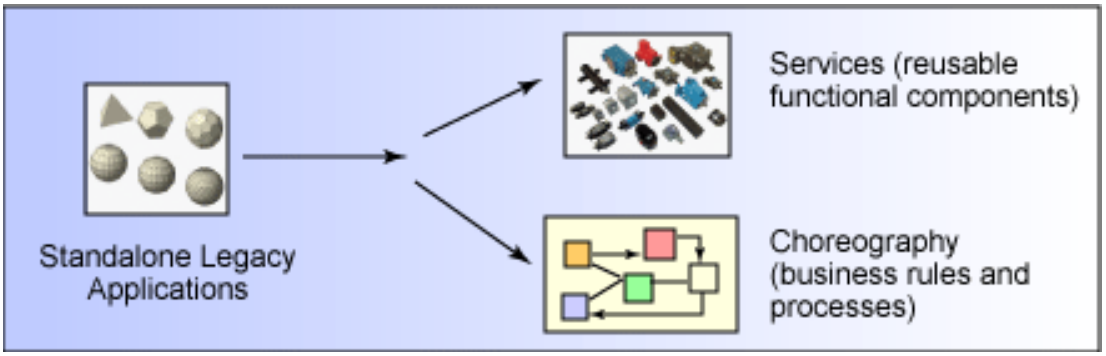
Çok katmanlı yapı sayesinde geliştiriciler bu katmanlarda paralel olarak çalışabilir. Projenin başında yazılan arayüz sözleşmeleriyle parçalar birbirine bağımsız olmadan geliştirilebilir. Servisler bir dizinde tutulduğundan ve uygulamalar bu servislere çalışma zamanı bağlandığından konum şeffaflığı sağlanır. Bu özellik sayesinde yük dengeleyiciler kullanılarak, servis kullanıcıların istekleri servis sunuculara arasında dengelenebilir. Konum şeffaflığı sayesinde çok sayıda sunucu aynı servisi üzerinde çalıştırabilir. Ağın bir bölümü ya da bir makine çökerse, kullanıcıların bilgisi olmaksızın istekler diğer sunuculara yönlendirilebilir.

Servislerle birlikte hem kullanıcı uygulamalar seviyesinde hem de servis seviyesinde kimlik doğrulama ile çok seviyeli güvenlik sağlanabilir. Servislerin açıklanmış arayüzleri (published interface) vardır [11]. Bu arayüzler birim testleri yazılarak kolayca test edilebilir ve bu testler yardımcı araçlarla otomatikleştirilebilir. Daha çok ve daha iyi test genellikle daha az hataya ve daha yüksek kalite demektir.

Servis Odaklı Mimari iki yaklaşımla gerçeklenebilir; *yukarıdan-aşağıya* (*top-down*) ya da *aşağıdan-yukarıya* (*bottom-up*). Yukarıdan-aşağıya yaklaşımda, sıfırdan sistem tasarlanırken servisler belirlenir ve aşağıda doğru servisleri gerçekleyecek sınıflar yazılır. CBM, SOA modellemesi için iyi bir başlangıç sağlayabilir. Ancak pratikte durum pek böyle değildir, yazılımcıların elinde uzun zamandır kullanılan, test edilmiş yazılım parçaları vardır ve bu sistemlerin bir kısmını servis odaklı mimariye geçirmeleri gerekir.

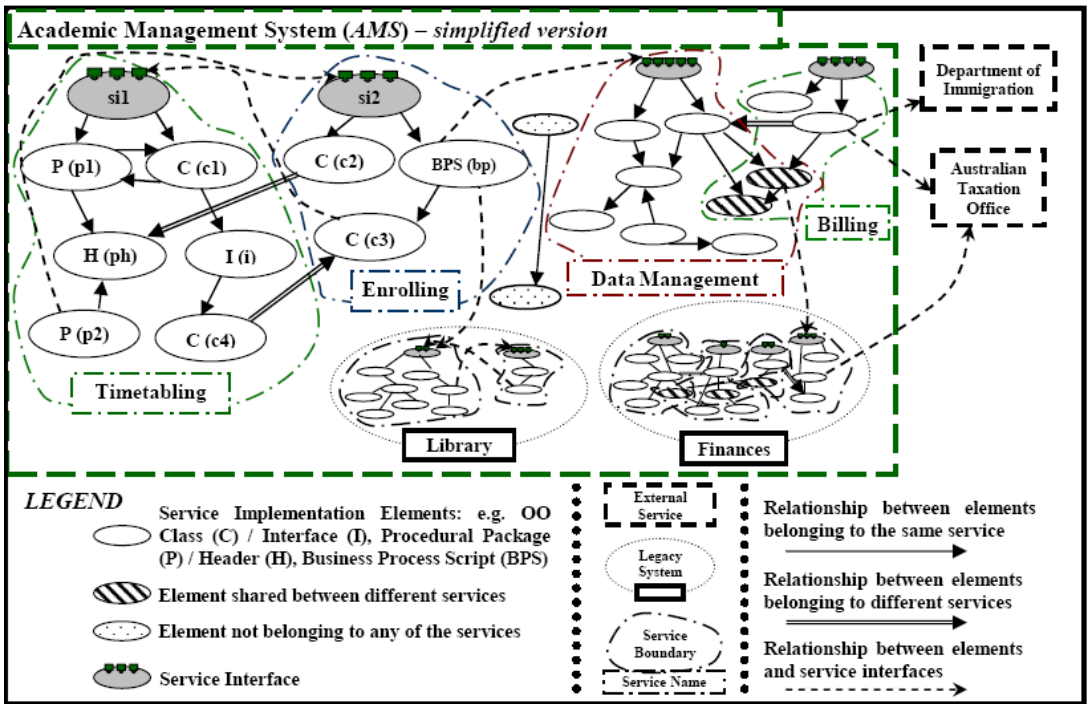
Var olan sistemden servisleri, işlemleri, iş süreçlerini ve iş kurallarını *"aşağıdan yukarıya"* ayrıştırmaları gerekir (*bottom-up*) (Şekil 1.4). Böylece yazılım hem servis

odaklı tasarımın avantajlarını sahip olur, hem de diğer yeni servis odaklı kısımlarla kolayca tümleştirilebilir.



Şekil 1.4 : Servis ayrıştırılması [9].

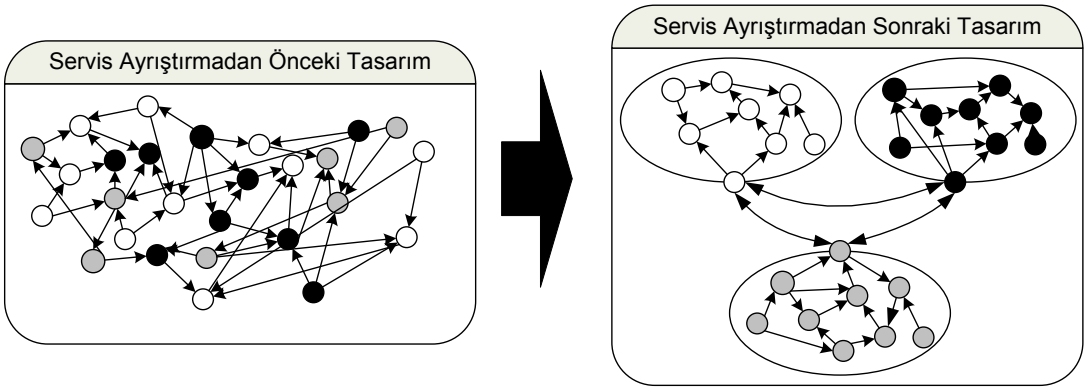
Tez çalışması kapsamında var olan yazılım sistemlerin içindeki servisleri bulan bir yöntem geliştirilmiştir, böylece Şekil 1.5'de olduğu gibi bir servis görüntüsü ortaya çıkartılmaktadır.



Şekil 1.5 : Örnek bir servis odaklı sistem tasarımı [12].

Yazılım servislerini, otomatik olarak belirlemek yazılım geliştiricilere zaman kazandırır ve servis odaklı mimarinin yukarıda bahsedilen faydalarının daha kolay edinilmesini sağlar. Var olan bir nesneye dayalı sistemin servis ayrıştırılması Şekil 1.6'da gösterilmiştir.

Bu tez çalışmasında, nesneye dayalı sistemler için öğrenme tabanlı yeni bir otomatik modül bulma metodu önerilmiştir. Bu metot nesneye dayalı sistemlerin modüler yapısına dayanmaktadır. Literatürdeki modül bulma çalışmalarında nesneye dayalı yazılım sistemi genellikle sınıflar ve aralarında ilişkilere göre oluşturulan bir bağımlılık matrisi ya da bağımlılık çizgesi olarak temsil edilir. Otomatik mimari çıkarımı için, yazılım çizgeleri üzerinde kümeleme algoritmalarına yaygın olarak başvurulmaktadır.



Şekil 1.6 : Var olan nesneye dayalı bir sistemin servis ayrıştırılması.

Modül bulma metotları, bir modülerlik objektif fonksiyonu tanımlayarak bunu belirli kriterlere göre eniyilemeye çalışır. Bu kriterler, modüllerin yüksek uyumlu ve gevşek bağımlı olması gerektiği varsayımına dayanır. Bu yüzden modül içi (iç) ayrıt yoğunluğunun fazla, modüller arası (dış) ayrıt yoğunluğunun az olması gerekir.

Bu çalışmada önerilen modül bulma yöntemi, sadece ayrıtların yoğunluğuna değil, aynı zamanda nesneye dayalı sınıfların karakteristiğini ve aralarındaki ilişkilerin niteliklerini de göz önüne alarak modülleri keşfetmektedir. Sınıfların ve aralarındaki ilişkilerin karakteristikleri, tasarımlarda başvurulan nesneye dayalı tasarım prensiplerine göre belirlenmektedir. Yazılım mimarları ve geliştiriciler yeniden kullanılabilir esnek bileşenler geliştirerek bakımı kolay yüksek kaliteli yazılım sistemleri üretmeyi amaçlarlar. Bu nedenle, nesneye dayalı tasarım prensiplerini ve tasarım kalıplarını izlerler ya da en azından izlemeye çalışırlar. Tez kapsamında bu kavramların nesneye dayalı sistemlerin modülerliğini ciddi şekilde etkilediği inancından yola çıkılarak, özgün bir modül bulma metodu geliştirilmiştir. Örneğin, Önyüz (Facade) sınıfları, genellikle modülün sınırında yer alır ve bu sınıfların, karmaşıklık ve uyumluluğu düşük bağımlılıkları ise yüksektir. Diğer yandan modülün iç kısmında yer alan iç sınıflar, modülün işlevsel görevini yerine getirir, bu

nedenle daha karmaşık ve daha yüksek kütüphane bağımlılığına sahiptirler. Bu iç sınıflar uygun şekilde yazılmışsa, yüksek uyumluluğu da sahiptirler.

Çizge kümeleme algoritmaları çizgedeki güçlü bağlı düğümleri gruplarken, zayıf bağlı ayrıtları keserek modülleri oluşturmaktadır. Önerilen yöntemde algoritmanın doğruluğunu arttırmak için ayrıtlara, "*modül içi*" (*iç*) ya da "*modüler arası*" (*dış*) olma olasılıklarına göre ağırlık atanmaktadır. Ağırlıkları uygun şekilde belirlemek için karar ağacı şeklinde iki makine öğrenmesi tabanlı sınıflandırıcı kullanılmaktadır. Birinci karar ağacı ayrıtların konum (*iç/dış*) olasılığını, temsil ettikleri ilişkinin niteliklerine (Örneğin kalıtım, metot çağrısı, nitelik değişkeni, sahip olma, kullanma vs) göre tahmin etmektedir.

Sadece ilişki özelliklerini göz önüne almak, ayrıtların gerçek konumunu tahmin etmek için yeterli değildir. Bu nedenle ikinci karar ağacı da, ayrıtın kaynak ve varış düğümlerini, sınıfların tiplerine (soyut [abstract], arayüz [interface]) ve uyumluluk, karmaşıklık, bağımlılık gibi kalite metriklerine göre "*iç düğüm*" ve "*kenar düğüm*" olarak sınıflandırmaktadır. Daha sonra bu iki sınıflandırma sonucu birleştirilerek ayrıtın olası konumu (iç/dış) daha doğru biçimde tahmin edilebilmektedir.

Son adımda modül içi olma olasılığı yüksek ayrıtlara yüksek ağırlık, modül dışı olma olasılığı yüksek ayrıtlara düşük ağırlık verilmektedir. Sınıflandırıcılar gerçek dünyadan derinlemesine incelenmiş bir yazılım sisteminden edinilen bilgilere göre eğitilmektedir. Bu referans yazılım sistemi, modüler yapıda, nesneye dayalı tasarım prensip ve kalıplarına göre geliştirilmiş bir sistemdir. Eğitim verisi sınıfların ve ayrıtların seçilen niteliklerini ve referans projedeki gerçek konum bilgilerini içermektedir. Öğrenme fazından sonra, sınıflandırıcılar modüler bir nesneye dayalı tasarımın özelliklerini "öğrenmektedir" öyle ki yeni düğümler ve ayrıtlar verildiğinde karakteristik özelliklerine göre ayrıtların konumlarını iç ve dış olarak sınıflandırabilir. Bildiğimiz kadarıyla, literatürde nesneye dayalı tasarım özelliklerini modül bulmada kullanan olgunlaşmış herhangi bir başka çalışma bulunmamaktadır.

Öğrenme fazından sonra, üç ana adımı içeren önerilen modül bulma metodu kullanıma hazırdır. İlk adımda, tüm yazılım mimarisinin çizge modeli doğrudan yazılımın Soyut Sentaks Ağaçları (SSA - Abstract Syntax Tree [AST]) işlenerek, yazılım varlıkları (sınıflar ve arayüzler) ve aralarındaki ilişkiler çıkarılarak oluşturulmaktadır. Sistemin mimarisi basit yönlü ve etiketli çizge olarak temsil

edilmektedir. İkinci adımda, ayırtlara modül içi ve modül arasındaki konum olasılıklarına göre ağırlık atanmaktadır. Burada öğrenme aşamasında referans proje çizgesindeki ayırtlarla eğitilen karar ağacı temelli sınıflandırıcılar kullanılmaktadır. Son olarak, bu yönlü ve ağırlıklı çizgelerdeki modülleri bulmak için, tez kapsamındaki çalışmalarda [13] nesneye dayalı sistemlerin özelliklerine uygun olduğu gösterilen hiyerarşik birleştirici (agglomerative) bir kümeleme algoritması (Fast Community [14]) kullanılmaktadır.

Tez çalışmasında önerilen özgün modül bulma yaklaşımı, çeşitli açık kaynaklı popüler projeler ve endüstriyel projelerde, geliştirme takımlarıyla yakın çalışılarak değerlendirilmiştir. Otomatik bulunan modüllerin, geliştiriciler ve yazılımın uzmanları tarafından belirlenen modüllere olan yakınlığı ölçülmüş ve otomatik bulunan modüller ile gerçeğe yakın sonuçlar elde edildiği gösterilmiştir. Deney sonuçları göstermiştir ki, önerilen metot doğruluk, kararlılık, modül boyunun düzgün dağılımı ve performans açısından literatürde yayınlanmış diğer yaklaşımlardan üstün sonuçlar üretmektedir.

1.2 Tezin Amacı

Bu tez çalışmasının ana hedefleri şöyle sıralanmaktadır:

- Yazılımlardaki modülleri ve ilişkileri, kaliteli tasarımlarda sıklıkla kullanılan nesneye dayalı tasarım kalıpları ve prensiplerine (inversion of control, GoF Kalıpları, gibi...) göre değerlendirmek. Bu perspektifle tasarımda belirli bir hizmeti yerine getiren servis kümelerini bulmak. Böylece:
    - Yazılımın daha iyi anlaşılmasını sağlamak (Software Comprehension).
    - Var olan sistemlerin SOA'ya geçirilmesini (Bottom-up) kolaylaştırmak (Refactoring/Migration).
- Gevşek bağlı parçalardan oluşan servislerin belirlenmesiyle bunların kolaylıkla dağıtılmasına yardımcı olmak ve dolayısıyla kaynak kullanımını etkin hale getirmek.
- Yazılım kalitesini servis odaklı değerlendirmek, böylece:
    - Servis içi ve servisler arası ilişkileri farklı ağırlıklandırmak.

- Yazılım mimarlarının büyük resmi görmesine yardımcı olacak yöntemleri ve araçları geliştirmek.
- Var olan sistemlerdeki kusurlu noktaları belirleyerek, tasarım kalitesini arttırılmasına yardımcı olmak.

Temelde modülerlik kalitesi, yüksek iç uyum, düşük dışa bağımlılık olarak tanımlanır. Bazı çalışmalarda iç ayrıt sayısı çok fazla olması iyi uyumlu olarak kabul edilmiştir, halbuki modül içindeki ayrıtları da mümkün olduğunca az tutmak gerekir. Ancak iç ve dış ayrıtları değerlendirmede, iç ayrıtların ağırlığı daha düşük olması gerekir. Yazılım mimarları, yazılımdaki iş bölümünü sınıf bazında değil, mantıksal hizmetler bazında tasarlarlar. Sınıfın tanecikliği bir üst seviyede çok düşük kalmaktadır. Bu nedenledir ki en basit tasarım kalıbı bile bir kaç sınıftan oluşmaktadır. Tasarıma servis düzeyinde bakmak büyük resmin görülmesini de kolaylaştıracaktır. Ayrıca servis düzeyinde yeniden kullanılabilirlik daha iyi değerlendirebilir.

Tez çalışması kapsamında önerilen yöntemin sağladığı özellikler şöyle sıralanabilir:

- Tasarımdaki farklı ilişkileri farklı ağırlandırılarak, (Örneğin modül içi ve modül dışı olanlar, kalıtım, içerme [composition], sahip olma [association] ilişkileri) değerlendirebilmektedir.
- Değerlendirme ölçütleri olarak nesneye dayalı tasarım prensipleri ve kalıpları, yazılım metrikleri referans noktası olarak kullanılmaktadır.
- Yöntem kullanıcılardan yardım alabilmektedir. Örneğin kullanıcı, bazı sınıfları servislerle kendi ilişkilendirebilir.
- Sadece saf servise dayalı mimarilerde değil, servise dayalı mimariden ufak sapmaları olan yazılım sistemlerinde de çalışmaktadır.
- Hem kod üzerinden hem de UML tasarımları üzerinden çalışılmasına izin vermelidir.

1.3 Tezin Konuya Katkıları ve Ana İskeleti

Tez çalışmasında nesne dayalı sistemler için özgün bir otomatik modül bulma yöntemi geliştirilmiştir. Bildiğimiz kadarıyla literatürde nesneye dayalı yazılımlar için özellikle geliştirilmiş otomatik bir modül bulma yaklaşımı yoktur. Bu çalışmaya

alternatif diğer otomatik modül bulma yöntemleri ile karşılaştırıldığında tez çalışması sonucu şu katkılar sağlanmıştır:

- Nesneye dayalı yazılımların tasarımlarını detaylı biçimde ifade eden bir çizelge modelinin tanımlanmıştır. Nesneye dayalı yazılım çizgelerinin özellikleri derinlemesine incelenmiş ve bu özelliklere uygun çizge kümeleme algoritmalarının çizgeye uygulanmasıyla, tasarım modüllerinin belirlenmesi özgün bir biçimde gerçekleştirilerek bu konuda öncül bir çalışma yapılmıştır.
- Önerilen otomatik modül bulma yaklaşımı sadece uygun çizge kümeleme algoritmasından değil, aynı zamanda nesneye dayalı yazılımlarda sıklıkla kullanılan tasarım kalıplarından ve prensiplerinden de faydalanarak daha doğru, daha etkin sonuçlar üretmektedir. Nesneye dayalı metriklerin, sınıf ve sınıflar arası ilişkilerin nesneye dayalı modeldeki özelliklerine göre, daha doğru modül bulma sonucu üreten ve bunu referans bir servis odaklı sistemden öğrenen yeni bir otomatik modül bulma yaklaşımı geliştirilmiştir.
- Önerilen yöntem, farklı sistemler için genişletmeye ve uyarlanmaya uygundur. Bu modelde seçilen nitelik ve metrikler, kullanılan makine öğrenmesi tabanlı sınıflandırma algoritmaları, uygulama uzayına özgü olarak genişletilip, uyarlanabilir.
- Önerilen bu özgün yöntem, açık kaynaklı ve endüstriyel projeler üzerinden denenmiş elde edilen sonuçlar, geliştiriciler ile birlikte analiz edilmiş ve otomatik bulunan modüller ile referans modüller arasındaki yakınlıklar ölçülmüştür. Geliştirilen yöntem literatürdeki diğer yaklaşımlarla, referans mimariye yakın sonuçlar üretme (doğruluk), yazılımın değişikliklerine karşı tutarlı sonuçlar üretme (kararlılık), üretilen modüllerin boyutunun düzgün dağılımı ve çalışma zamanı başarımı açısından karşılaştırılmış ve literatürdeki yöntemlerden daha başarılı sonuçlar ürettiği gösterilmiştir.
- Önerilen yöntem, geliştiriciler için oldukça zahmetli ve zaman alan, yazılım mimarisinin çıkarılmasında (Software Architecture Recovery) ve yazılım sistemini anlamakta (Software Comprehension) yardımcı olmakta, yazılımın bakımını ve yazılım sisteminin servis odaklı mimariye geçirilmesini kolaylaştırmaktadır.

Tez çalışmasının geri kalanı şöyle organize edilmiştir: Bir sonraki bölümünde literatürde modül bulma alanında yapılmış temel ve tez konusuyla ilgili güncel çalışmalar anlatılmıştır. Üçüncü bölümde yazılımın modül bulma açısından ifade eden yazılım çizge modeli anlatılmıştır. Dördüncü bölümde yazılımlarda sınıf ve ilişkilerin yerini modül mimarisindeki yerini belirlemede başvurulan ve bunların yazılım mimarisine olan etkilerinin anladığı nesneye dayalı kalite nitelikleri ve metriklere yer verilmiştir. Beşinci bölümde, Yazılım çizge kümeleme yöntemleri ve yazılım çizgelerinin karakteristiklerinin bu yöntemlerin başarıma etkileri aktarılmıştır. Altıncı bölümde, önerilen öğrenme tabanlı nesneye dayalı servis bulma yaklaşımı ayrıntılarıyla anlatılmıştır. Yedinci bölümde geliştirilen yöntemin, açık kaynaklı ve endüstriyel yazılım projelerine uygulanması ve diğer yöntemlerle kıyaslanmasına ilişkin sonuçlar verilmiş ve bu sonuçlar tartışılmıştır. Son bölümde ise yapılan çalışma özetlenerek ileriye dönük araştırmalar için çeşitli öneriler sunulmuştur.

2. LİTERATÜR ARAŞTIRMASI

Bu bölümde literatürde otomatik modül bulma amacıyla yapılmış başlıca çalışmalar anlatılmıştır. Yazılımlarda modül bulma amacıyla literatürde çeşitli algoritma ve yaklaşımlar geliştirilmiştir. Bu yaklaşımlarda, genellikle sınıflar ve kaynak kod dosyaları gibi yazılım varlıkları arasındaki bağımlılıklar kullanılır. Bu çalışmalarda kümeleme algoritmalarına sıklıkla başvurulduğundan, yazılımlarda modül bulma, literatürde yazılım kümeleme (Software Clustering) olarak isimlendirilmektedir. Modül bulma çalışmaları farklı şekillerde gruplanabilir; Örneğin kümelemede kullanılan bilgi (yapısal, kelime tabanlı), kullanılan varlıkların tanecikliği (kaynak dosyaları, sınıf, metot) bilginin edinilme şekli (statik, dinamik), kümeleme metotları (hiyerarşik, bölmelemeli, arama tabanlı) ve kullanıcı etkileşim gereksinimleri (otomatik, yarı otomatik) birer gruplandırma kriteri olabilir. İlerleyen alt bölümlerde ilk olarak yapısal bağımlılıklara dayalı modül bulma yöntemlerine yer verilmiştir, ardından sözcüksel analize dayalı yöntemler ve son olarak her ikisini de kullanan birleşik yöntemler anlatılmıştır.

2.1 Yapısal Bağımlılıklara Dayalı Teknikler

Literatürdeki temel yaklaşımların büyük çoğunluğu yazılım varlıkları arasındaki yapısal bağımlılıkları (Örneğin sınıflar ve kaynak dosyaları arasındaki ilişkiler) kullanırlar. Wiggerts [15] modül bulma için geliştirilen, yapısal bilgiye dayalı kümeleme algoritmaların bir özetini sunmuştur. Kümeleme algoritmaları temelde iki kategoriye ayrılabilir: hiyerarşik ve bölmelemeli. Hiyerarşik algoritmalar modülleri bir hiyerarşi düzeni içinde ayrıştırırken, bölmelemeli algoritmalar düz bir ayrıştırma yapısı sunarlar. Çeşitli hiyerarşik algoritmaların performansları daha önceki bazı çalışmalarda [16], [17] derinlemesine analiz edilmiştir. Bu çalışmalarda algoritmaların performansı, ürettikleri kümeleme sonuçlarının, uzman ayrıştırma ile karşılaştırılması ile ölçülmüştür. Anquetil ve Lethbridge [17], Wiggerts'in çalışmasında [15] yer alan beş algoritmanın farklı benzerlik metrikleriyle karşılaştırmalı analizini sunmuştur. Algoritmaları analiz etmek için iki büyük

prosedürel yazılım sistemini (Linux ve Mosaic) kaynak dosyaları arasındaki ilişkileri kullanmışlardır.

Maqbool ve Babri [16] yazılım modülerleştirme için çeşitli hiyerarşik kümeleme metotlarını gözden geçirmiş, kümelenecek varlıklar arasındaki benzerlik/uzaklık ölçülerinin karakteristiklerini araştırmış ve algoritma tarafından rastgele seçilen (arbitrary) kararların kümeleme kalitesine etkisini analiz etmişlerdir. Burada C dili yazılmış dört orta boyutlu (30 KLOC ve 70 KLOC arası) yazılım sistemi üzerinde deneyler yapılmıştır. Tzerpos ve Holt [18] kümeleme algoritmalarının kararlılığını, yazılımın ardışık versiyonları için ürettikleri sonuçların benzerliliği yönünden incelemiştir. Wu ve diğerleri [19] dört kümeleme yaklaşımını farklı konfigürasyonlarla yazılımın gelişimi bağlamında karşılaştırmışlardır. Bunlar:

1. Jaccard katsayısı ve tam bağ güncelleme kuralına (complete linkage update rule) dayalı birleştirici (agglomerative) bir kümele algoritmasıdır.
2. Jaccard katsayısı ve tek bağ güncelleme kuralına (single linkage update rule) dayalı birleştirici (agglomerative) bir kümele algoritmasıdır.
3. ACDC [20], yazılımda sık rastlanan modül yapılarının benzerliklerini bulmaya dayanan kalıp tabanlı bir kümeleme algoritmasıdır. Bu kalıplar nesneye dayalı tasarım kalıpları değil, başlık kaynak dosyası ayrımı gibi daha çok prosedürel dillerde görülen az sayıdaki temel kalıplardır.
4. Bunch [21], arama tabanlı algoritma kümesidir.

Bu yaklaşımlar beş büyük açık kaynaklı C/C++ sistem üzerinde denenmiş ve doğruluk, kararlılık, modül boyutlarının düzgün dağılımı açısından değerlendirilmiştir. Otomatik ayrıştırılan modül yapısı ile uzman modül ayrıştırması arasındaki uzaklığı ölçmek için MoJo [22] metriği kullanılmıştır. Bu çalışma sonucunda otomatik kümeleme yöntemlerinin ciddi iyileştirmeye ihtiyaç duyduğu belirtilmiştir.

Bittencourt ve Guerrero [23], yine aynı değerlendirme kriterlerini kullanarak dört Java sistemi üzerinde başka bir deneysel karşılaştırma çalışması yapmıştır. Bu çalışmada modül ayrıştırma sonuçlarının doğruluğunu ölçmek için MoJo metriğinin normalize edilmiş hali olan MoJoSim metriği kullanılmıştır.

2.1.1 Arama tabanlı yöntemler

Yapısal bilgilere dayalı tekniklerin bir alt grubu olan arama tabanlı yaklaşımlar, kümeleme problemine bir optimizasyon problemi olarak bakarlar. Bu yöntemler, geniş bir arama uzayında olası tüm çözümlerin arasından belirli bir objektif fonksiyonunda maksimum değeri verecek çözümü bulmaya çalışırlar. Çözüm uzayını taramak için çeşitli sezgisel yöntemlere başvurulmaktadır. Örneğin, Doval ve diğerleri [24] genetik algoritma kullanırken, Mancoridis ve diğerleri [25] tepe tırmanma ve genetik algoritmayı beraber kullanmıştır. Bu çalışmalarda modülerlik kalitesi (MQ) metriği [15] tek bir objektif fonksiyonu olarak kullanılmıştır. MQ'nun tanımı modülerlik prensiplerine, yani modül içi bağların yüksek (yüksek uyumluluk prensibi) modüller arası bağların az (az bağımlılık prensibi) olmasına dayanır. [24] ve [25]'de sunulan algoritmaların gerçeklemeleri Bunch kümeleme aracında sunulmuştur. [25]'den farklı olarak Mahdavi ve diğerleri [26] algoritmanın yerel bir tepe noktasında takılıp kalmasını önlemek için çoklu bir tepe tırmanma tekniği kullanmışlardır.

Seng ve diğerleri [27] yine arama tabanlı bir metot sunmuştur. Bu metot yine tek bir objektif fonksiyonlu genetik algoritma kullanır ancak bu uygunluk fonksiyonu beş metriğin birleşiminden oluşmaktadır. Uygunluk fonksiyonu bağımlılık ve uyumluluğa ek olarak karmaşıklık, çevrim sayısı ve dar boğaz metriklerini içerir. Son zamanlarda Praditwong ve diğerleri [28] çok objektif fonksiyonlu genetik bir algoritmaya dayalı bir yaklaşım önermiştir. MQ'yu en yükseğe çıkarmak, uyumluluğu en yükseğe çıkarmak, bağımlılığı en aza indirmek, yalıtılmış küme sayısını en aza indirmek, küme sayısını en yükseğe çıkarmak ya da en büyük ile en küçük küme arasındaki farkı en aza indirmek objektif fonksiyonlar arasında yer almaktadır. Çoklu değerlendirme kriteri olarak Pareto optimalliğini kullanmışlardır.

Yapısal kümeleme yaklaşımlarının büyük çoğunluğu kaynak koddan statik analiz ile elde edilen statik özellikleri kullansa da çalışma zamanı gözlenebilen dinamik özellikleri kullanan yöntemler de mevcuttur [29]. Ayrıca dinamik ve statik yöntemleri bir arada kullanan hibrit yöntemler de yazılım modüllerini bulmak için kullanılmıştır [30], [31].

2.2 Sözcüksel Analize Dayalı Teknikler

Son zamanlardaki çalışmalarda, yazılımlardaki sözcüksel özellikleri (Örneğin yorumlar, dosya adları, sınıf ve değişken adları gibi) kullanmanın kümeleme performansını iyileştireceği gösterilmiştir [32], [33], [34]. Kuhn ve diğerleri [35] yazılım modüllerini bulmak için semantik bir kümeleme yaklaşımı önermişlerdir. Belirteç adları ve yorumların içerikleri bir kelime hazinesine çıkarılmakta ve ilgili yazılım varlıkları (paketler, sınıflar ve metotlar) arasındaki benzerliği ölçmek için Latent Semantik İndeksleme (LSI) tekniğini kullanmışlardır. Yazılım varlıkları korelasyon matrisi üzerinde çalışan hiyerarşik bir kümeleme algoritması ile gruplanmaktadır. [35]'den farklı olarak Corazza ve diğerleri [36] Java sınıflarında bölge kavramını ortaya atmıştır. Sözcüksel bilgi sağlayan bu bölgeler yorumlar, JavaDocs, sınıf/metot belirteç adları ve değişken adları olarak tanımlanmıştır. Bu yaklaşımda bölgeler olasılıksal bir model ile ağırlıklandırılmakta ve K-Medoids algoritmasıyla [37] kümelenmektedir. İleride bu yöntem daha büyük bir veri seti ve altı bölge (sınıf isimleri, nitelik değişkeni isimleri, metot isimleri, parametre isimleri yorumlar ve kaynak ifadeleri) kullanılmasıyla genişletilmiştir [38]. Bu sözcüksel yaklaşımların dezavantajı yarı otomatik olmalarıdır yani en iyi ayrıştırmayı seçebilmek için kullanıcı müdahalesine ihtiyaç duymalarıdır. Ayrıca kaynak kodun yorum içermesine gerek duyulmaktadır. Bu problemleri gidermek amacıyla sözcüksel bilgilere dayalı ve yine LSI kullanan ancak bölgeler arasında ayrım yapmayan bir çalışma önerilmiştir [39]. Bu çalışmada en iyi ayrıştırmayı elde etmek için k-means kümeleme algoritması [40] uyarlanmıştır. Bu çalışma daha sonra benzerlik matrisinin arttırmalı hesaplanmasıyla hızlandırılmış ve daha büyük deney kümesi ile analiz edilerek genişletilmiştir [41].

2.3 Birleşik Teknikler

Hem yapısal hem sözcüksel bilgileri kullanan birleşik yöntemler de vardır. Andritsos ve Tzerpos [32] modül bulma sürecinde bilgi kaybını minimize etme kavramına dayanan hiyerarşik bir modül bulma yöntemi LIMBO'yu önermiştir. Algoritma yapısal ve yapısal olmayan (Örneğin dosya isimleri, geliştirici isimleri zaman mühürleri gibi) kullanabilmektedir. Scanniello ve diğerleri [34] katmanlı mimariye sahip yazılım sistemleri için bir kümeleme yaklaşımı önermiştir. Yazılım yapısal bilgiler kullanılarak bağ analizi algoritması ile katmanlara ayrıştırılır ve her katman

sözcüksel bilgilerin benzerliğini giriş olarak alan k-means algoritmasıyla kümelenir. [42]'de yazarlar nesneye dayalı bir yazılım sisteminin arttırmalı olarak yeniden modülerleştirilmesi için bir metot önermişlerdir. Burada yapısal ve semantik bilgiler paketleri ayrıştırmak için kullanılmaktadır. Sağlanan araç ile gerekli olduğunda daha uyumlu bir paket yapısı elde etmek için var olan paketlerin otomatik olarak bölünmesini önerebilmektedir. Bu teknik aynı pakette bulunması gereken ilgili sınıfları aralarındaki yapısal ve semantik uyumluluğu ölçerek belirlemektedir. Ek olarak son zamanlarda yapılan bir çalışmada Bavota ve diğerleri [43] modülerlik kalitesini arttıracak bir metot önermiştir. Bu metot ile kaynak koddan yapısal ve semantik bilgiler analiz edilerek bir sınıf daha uygun pakete taşınabilmektedir.

Yazılımda modül bulma çalışmaları yazılım modülerlik kalitesini arttırma amacıyla da kullanılmaktadır [44]. Yazılımlarda bulunan modüllerin ve servislerin değerlendirilmesi amacıyla bir kısım kalite metriği de tanımlanmıştır. Çünkü sistemin servis mimarisini anlamak için, sınıf seviyesi taneciklik çok düşük bir soyutlama seviyesinde kalmaktadır [9], [45]. Bu nedenle servis seviyesinde kalite analizi son zamanlarda daha çok ilgilenilen bir konu haline gelmiştir. Devam eden paragraflarda bu metrikler ve bunların kümeleme yöntemlerinde kullanımlarına dair literatürde bulunan çalışmalar anlatılmıştır.

Son yıllarda yapılan çalışmalarda servis odaklı tasarımların biçimsel modeli ortaya koyulmuş ve bu biçimsel modelin uygulaması olarak bazı metrik tanımları verilmiştir. Burada metriklerin kaliteyle ilişkisi ve metrik doğrulması yapılmayarak sadece biçimsel modelin nasıl kullanılabileceği gösterilmiştir [12]. [46] ve [47] çalışmalarında ise servis seviyesinde bağımlılık ve uyumluluk metrikleri tanımlanmış, bu metriklerin yazılım bakımına olan etkileri araştırılmıştır.

2008 yılında yayınlanan bir başka çalışmada [48], var olan nesneye dayalı modülerlik kalitesi metriklerinin, modül olarak sınıfı kabul ettikleri, ancak sınıfın tanecikliğinin çok küçük kaldığı, bu nedenle çok sayıda sınıfı barındıran süper-paketlerin modül olarak kabul edilmesi gerektiği belirtilerek, bu kabul ile bazı modülerlik metrikleri tanımlanmıştır. Süper paket (super-package) kavramı, Java diline de eklenmesi düşünülen bir kavramdır. Burada süper paketler, paketlerin hiyerarşik bir yapıda olduğu düşünüldüğünde üst seviye de kalan paketlerdir. Dolayısıyla süper paket kavramı servisten farklı olarak, mantıksaldan çok fiziksel bir

örtmeyi ifade eden bir kavramdır. Çalışmada metrikleri ölçme için süper paketler kullanıcılar tarafından paket adlarına bakarak elle tanımlanmaktadır.

Çalışmada tanımlanan metrikler bağımlılık metrikleri (kalıtım, sahip olma, doğrudan durum erişimi, yapısal metrikler), boyut düzgün dağılım metriği, uygulama arayüz (API) uyumluluk metrikleri, Sistem genişletilebilirlik metrikleri gibi gruplar halinde tanımlanmıştır. Çalışmada tanımlanan metriklerin dağılımları 0–1 arasında olması kolay yorumlanabilmeleri açısından oldukça yararlıdır. Bazı metriklerin birbirinden farklı olarak neleri ifade ettikleri ve nerelerde kullanılabileceği çok net değildir. Metriklerin doğrulanma yöntemi de, test edilen yazılımlara metrik değerlerinin düşüren rastgele kusurlar eklenmesi ve metrikler değerlerinin düştüğünün gözlenmesidir. Bu doğrulama yönteminin pratik açıdan metriklerin yararlı metrikler olduğunu göstermede yetersiz olduğunu görülmektedir. Ancak söz konusu çalışma çok büyük OO sistemler için yapıldığından, yazılım hakkında daha fazla bilgiye (çıkan hata oranı, değiştirilen kod sayısı, eklenen özellik vs) sahip olmadan daha yetkin doğrulama yapmak oldukça zordur.

3. NESNEYE DAYALI YAZILIM ÇİZGE MODELİ

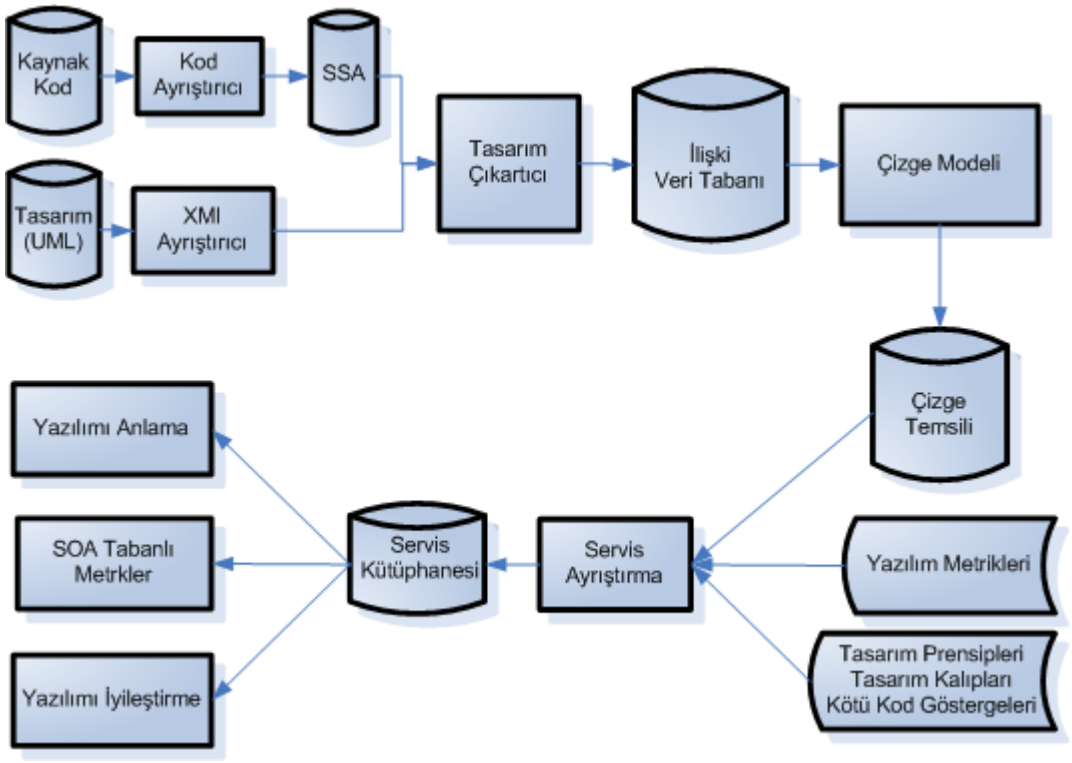
Nesneye dayalı bir sistemdeki modülleri otomatik bulmak için öncelikle modül bulma yöntemlerinin üzerinde koşacağı, yazılımı ifade eden uygun bir modelin oluşturulması ve verilen bir yazılımın kaynak kodundaki bilgilerin işlenerek yazılım için bu modelin otomatik çıkarılabilmesi gerekmektedir.

Modülleri belirlemek için en önemli giriş verisi elbette yazılım birimleri arasındaki bağımlılıklardır. Yazılımda birim kavramı ölçeğine göre, sınıf, paket ya da metot olabilir. Modül bulma çalışmalarında birimler arası bağımlılıklar temelde ya matris üzerinde ya da çizge üzerinde temsil edilir. Yazılım bağımlılık çizgelerindeki ortalama derecenin nispeten düşüklüğü (Genellikle 3 ile 5 arası) düşünüldüğünde, yazılımı çizge olarak ifade etmek, seyrek matris (sparse matrix) olarak ifade etmeye göre daha avantajlıdır.

Yazılım analizi için tez çalışmasında önerilen sistem mimarisi Şekil 3.1'de verilmiştir. Tezin giriş kısmında belirtildiği gibi uygun çizge modeli oluşturulduktan sonra, yazılımı değerlendirmek için çizge teorisi ve algoritmalarından yararlanılmaktadır. Çizge modeli yazılımın kaynak kodundan ya da UML tasarım dokümanlarından özel ayrıştırıcılar ile çıkarılabilir. Bu çalışmada bir yazılımdaki servisler, tasarım kalıplarından, tasarım prensiplerinden ve yazılım metriklerinden faydalanılarak, çizge kümeleme algoritmaları kullanılarak belirlenmeye çalışılmaktadır.

Belirlenen bu servisler yazılım evinin servis kayıt dizininde saklmanması ile yeni projelerde tekrar kullanılmasında, yazılım mimarisinin anlaşılmasında, yazılımın servis odaklı mimariye geçirilmesinde, servis odaklı bakış açısıyla kalitesinin ölçülmesinde ve gerekirse kod iyileştirmelerinin yapılmasında önemli rol oynayacaktır.

Takip eden alt bölümlerde sırasıyla yazılımın çizge modelinde temsili ve tez kapsamında bu modelin yazılımlardan nasıl çıkarıldığına dair bilgiler anlatılmıştır.

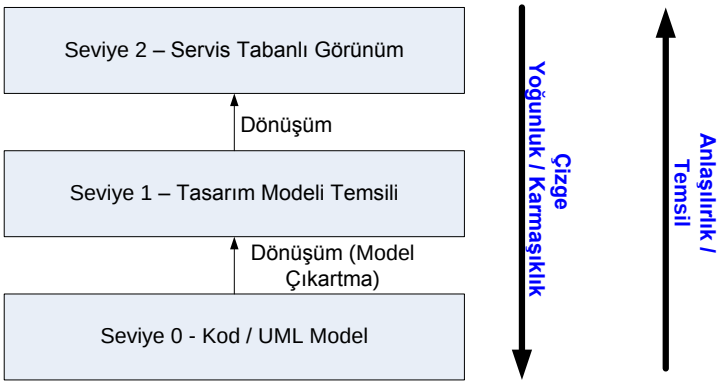


Şekil 3.1 : Önerilen yazılım analizi sistem mimarisi.

3.1 Çizge Modeli ve Çizge Temsili

Modül bulmada kullanılan modelin başlıca özellikleri şunlardır:

- Tasarım çıktılarını (UML [49] Sınıf ve Etkileşim Diyagramları) kullanabilir.
- Kaynak kod çıktılarını kullanılmaktadır (daha ayrıntılı sonuçlar için).
- Tasarımdaki farklı tipteki ilişkileri, ayırt edilmekte ve derecelendirilmektedir. (Örneğin modül içi ve modül dışı olanlar, kalıtım, içerme, nitelik değişkenine sahip olma ilişkileri)
- Değerlendirme ölçütü olarak nesneye dayalı tasarım prensipleri ve kalıpları, yazılım metrikleri referans noktası olarak kullanılmaktadır.
- Model üzerinde kullanıcılardan yardım alınabilmektedir. Örneğin kullanıcı, bazı sınıfları servislerle kendi ilişkilendirebilir ya da bazılarının ihmal edilmesini sağlayabilir.
- Sadece saf servise dayalı mimarilerde değil, servise dayalı mimariden ufak sapmaları olan yazılım sistemlerinde de çalışmaktadır.
- Genişletilebilir ve ölçeklenebilir özelliktedir.



Şekil 3.2 : Model katmanları.

UML, Tümleşik Modelleme Dili, OMG [50] tarafından geliştirilmiş, yazılımın modellenmesinde oldukça yaygın kabul görmüş standart bir modelleme dilidir. Tasarım bileşenlerinin modellenmesinde UML iyi bir başlangıç olsa da, servisleri bulmak ve değerlendirmek için bu modeli de içine alan daha ve ayrıntılı yeni bir çizge modeline ihtiyaç duyulmuştur.

Bu amaçla ortaya çıkartılan model katmanlı bir yapıda olup (Şekil 3.2), en alt katmanda yazılımın kaynak kodu ya da UML diyagramı bulunmaktadır. Bu katmanın üstünde kaynak kod üzerinde bazı dönüşümler yapılarak tasarım modeli temsili olan çizge modeli çıkartılmaktadır. En üst katmanda ise servis tabanlı kalite metrik görünümü bulunmaktadır. Katmanlar arasında aşağı doğru inildikçe çizge yoğunluğu ve karmaşıklık artarken, yukarı doğru çıkıldığında anlaşılabilir ve yorumlanabilirlik artmaktadır.

Bir yazılım tasarım elemanları ve bu elemanlar arasındaki ilişkilerden oluşur. Modelde bulunan başlıca tasarım elemanları şunlardır:

- Servisler/ Modüller
- Paketler/İsim Uzayları (Packages / Namespaces)
- Sınıflar ve Arayüzler (Classes / Interfaces)
- Metotlar (Methods)
- İlkel Tipler (Primitive Types)

Burada ilkel tiplerden kasıt, programlama dilinde olan *int*, *char*, *boolean* gibi tiplerin yanında uygulamaya özel (Domain Specific) ilkel tiplerdir. Örneğin bir grafik

programı için *Nokta* sınıfı ya da bir ağ protokol yazılımı için *Paket* sınıfı ilkel bir tip olarak kabul edilmektedir. Buna güzel bir örnek, C++'da *string* sınıfı bir ilkel tip değilken Java'da *String* sınıfının bir ilkel tip haline gelmesi verilebilir.

Tasarım elemanları arası ilişkiler ise şu şekildedir::

- Servis – Servis (Service – Service)
    - Servis İçi (interService)
    - Servisler Arası (intraService)
- Sınıf – Sınıf (Class – Class)
    - Kalıtım, sahip olma, içerme, iç sınıf, nitelik değişkeni, (inheritance, association, aggregation, composition, inner, attribute variable)
- Sınıf – Metot (Class-Method)
    - Üyesi olma
    - Parametre/geri dönüş tipi
    - Yerel değişken tipi
- Metot – Metot
    - Metot çağrısı

ELEMANLAR

Elemanlar, servisler, paketler, sınıflar, metotlar olabilir. Elemanlar çizgede düğümlerle temsil edilmektedir. Elemanlar arası ilişkiler ise düğümler arası ayrıtları temsil etmektedir.

- Eleman
    - Özellikler
- İlişkiler
    - Eleman A – Eleman B (tuple)
    - Tür (etiket)
    - Ağırlık (metrik değeri)
    - Yön

- İlişkilerin birbiriyle ilişkisi
    - Bir ilişkinin diğer bir ilişkiyi örtmesi, ya da bir arada bulunması gibi

Bunları oluşturulduktan sonra yazılımlar yönlü etiketlenmiş ve ağırlıklı (directed labeled weighted) bir çizge (tasarım ya da kod görüntü çizgesi) olarak temsil edilmektedir. Metrikler bu çizge üzerinde tanımlanıp, kümeler bu çizgede oluşturulmaktadır. Bu çizge üzerinde etiket ve ağırlığa göre servisleri belirleyen sezgisel bir yöntemler geliştirilmiştir. Örneğin belirli tip bağımlılığı çok fazla olanlar aynı kümede olacaktır. Burada önemli bir noktada ilişkileri birbiriyle beraber değerlendirme imkânıdır. Örneğin servis dışı, farklı türdeki ve ağırlıktaki ilişkiler önem arz edecektir.

SERVİSLER:

Servis, bir bileşen tarafından sağlanan ve diğer bileşenlerin arayüz sözleşmesine dayanarak kullandığı davranıştır Servisler paketlerden oluşur. Paketler sınıflardan oluşur. Her sınıf tek bir pakete aittir, bir sınıf tek bir servise aittir. Servisin gerçekleme sınıfları vardır. Gerçekleme Sınıflarına servis dışından erişimin olmaması beklenir. Servisin arayüz sınıfları vardır. Arayüz sınıflarının, tüm açık (public) metotları servis arayüzünü oluşturur. Bu sınıfların kullanımları da servis kalitesi hakkında fikir vermektedir.

PAKETLER:

Paketler diğer paketleri içerebilir. Paketler sınıfları içerir.

Paket Özellikleri:

- Genel
    - İsim
    - Ait olduğu Paket
    - Ait olduğu Servis
    - Dış/iç
- İlişkiler:
    - İçerme

SINIFLAR:

Sınıf Özellikleri:

- Genel
  - İsim
  - Ait olduğu Paket
  - Ait olduğu Servis
  - Hiçbir servise ait olmayabilir
    - Tek bir servise ait olabilir
    - Rolü
      - Servis arayüz sınıfı
      - Servis iç sınıfı (gerçekleme)
      - İkisi de
  - Soyut Sınıf (Abstract)
  - Arayüz (Interface)
  - Türetilemez Sınıf (Leaf)
  - Görünürlük (Visibility +,~,-)
  - Tekil Sınıf (Singleton)
  - Şablon (Template)
    - Parametreleri (Template Parameters)
- Nitelikleri (Attributes - Fields)
  - İsim
  - Tip (İlkel Tip/Sınıf Tipi)
  - Görünürlük (Visibility +,~,-)
  - Çokluk (Multiplicity)
  - Paylaşılan/Bileşik (Shared/Composite/Aggregation)
  - Statik
  - Sabit (Read Only-Const)

    - Türetilmiş (Derived)
    - Şablon (Template)
- Metotlar (Methods- Operations)
    - İsim
    - Görünürlük (Visibility +,~,-)
    - Soyut (Abstract)
    - Türetilemez Metot (Leaf)
    - Statik (Static)
    - Sabit (Read Only-Const)
    - Geri Dönüş Tipi (Return Type)
    - Parametre Listesi /Parametre Tipleri
    - Şablon Parametre (Template Parameter)
    - Yaratıcı/Yok edici (Constructor/Destructor –Create/Destroy)

İlişkiler:

Sınıflar arası doğrudan (Birinci dereceden) ilişkiler (Bağımlılıklar):

- Sınıflar diğer sınıf tiplerinde üye niteliklere sahiptir.* (Nitelik Görünürlüğü)
    - Sahip olma (Association) (Çift yönlü tek yönlü)
    - Grup erişimi (Multiplicity) (dizi vektör liste)
    - Parçası olma (Aggregation Association)
    - İçerme (Composition Association)
    - Sınıflar diğer sınıflardan türetilebilir (Generalization/Specification)
    - Sınıflar arayüzleri gerçekleyebilir (implements)

Sınıflar arası dolaylı (İkinci dereceden) ilişkiler (Bağımlılıklar): (2. Dereceden ilişkiler metotlar incelenerek bulunur.)

- Parametre Görünürlülüğü
- Yerel Görünürlülük

- Global Görünürlülük
- Yabancı Sınıflar (Yabancılarla konuşma prensibi...)
- Şablon sınıflar kullanılabilir.

Sınıflar ile diğer servislerdeki sınıflar arasındaki ilişkileri tiplerine ve niteliklerine göre ayırmak servisleri belirlemek açısından önemlidir. Çünkü yazılım mimarları da sınıflar arasındaki birinci ve ikinci dereceden ilişkileri ayrı değerlendirmektedir ve doğrudan ilişkiler dolaylı ilişkileri örtmekte, doğrudan (Birinci dereceden) ilişki varsa, dolaylı (ikinci dereceden) ilişkiye göz ardı etmektedir.

Sınıflar metotları içerir.

METOTLAR:

Özellikleri:

- İsim
- Ait olduğu sınıf
- Sabit (Const)
- Basit atama/edinme (Getter/Setter)
- Görünürlük (Visibility +,~,-)
- Statik
- Parametre listesi
- Sıra dışı durum atma Listesi (Exception Throw List)
- Doğal (Native)
- Yaratıcı/Yok edici (Constructor/Destructor)

Metot ilişkileri:

- Metotlar bulundukları sınıfın metotlarını çağırır.
- Metotlar diğer sınıfların metotlarını çağırır (ve çağrılır).
- Metotların aynı isimde, farklı imzalı görevdeş metotları vardır (Overloading).
- Metotlar üst sınıfın, çok şekilli (polymorphic) metotlarını örter (Overriding).
- Metotlar diğer sınıf tiplerinden parametre alır (parametre in/out olabilir).

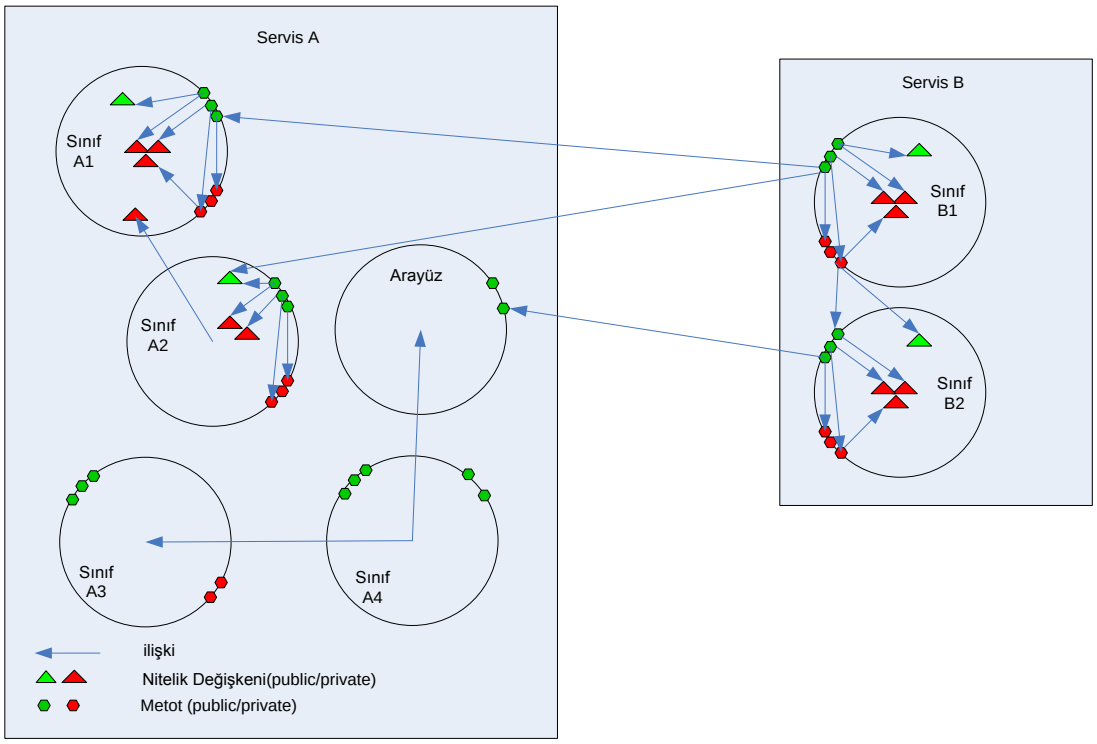
- Metotlar diğer sınıf tiplerinden geri dönüş (return type) değeri alır.
- Metotlar diğer sınıf tiplerinden nitelikleri parametre olarak aktarabilir.(parametre olarak aktarma, giriş/çıkış parametresi [in/out]).
- Metotlar diğer sınıf tiplerinden yerel değişkene (local variable) sahiptir. (2. Dereceden)
- Bir sınıftan nesne yaratma. Nesne yaratmak için (Creational Patterns – instantiation), sınıf hakkında daha çok şey bilmek gerekiyor. Güçlü bir bağ söz konusu bu nedenle sıradan bir metot çağrısından farklı değerlendirilebilir.
- Metotlar diğer sınıf tiplerinden global değişkenlere erişir.

Burada diğer sınıfın niteliği önemlidir. Diğer sınıf = Birinci derecede ilişkili olunan bir sınıf ise, ikinci dereceden ilişki göz ardı edilebilir.

Sınıfın ilişkili olduğu sınıf türleri şunlar olabilir:

- Türetildiği sınıf ile ilişki
- İç (inner) sınıf ile ilişki
- Servis içi bir sınıf ile ilişki
- Servis dışı Tip1 – iç servis sınıfı - dış servis gerçekleme sınıfı arasındaki ilişki (kötü tasarım)
- Servis dışı Tip2 - iç servis sınıfı - dış servis arayüz sınıfı arasındaki (daha kötü tasarım)
- Servis dışı Tip1 – servis arayüz sınıfı – dış servis arayüz sınıfı arasındaki ilişki
- Servis dışı Tip2 – servis arayüz sınıfı – dış servis gerçekleme sınıfı arasındaki ilişki

Model elemanları ve aralarındaki ilişkilere için basit bir örnek Şekil 3.3‘de verilmiştir. Şekilde görüldüğü gibi Servis A, A1, A2, A3 ve A5 sınıflarını içermektedir. B servisi ise B1 ve B2 sınıflarını içermektedir. Sınıfların açık ve kapalı nitelikleri ve metotları olabilir. Servisler arasında sınıflar üzerinden değişik ilişkiler olabilir.

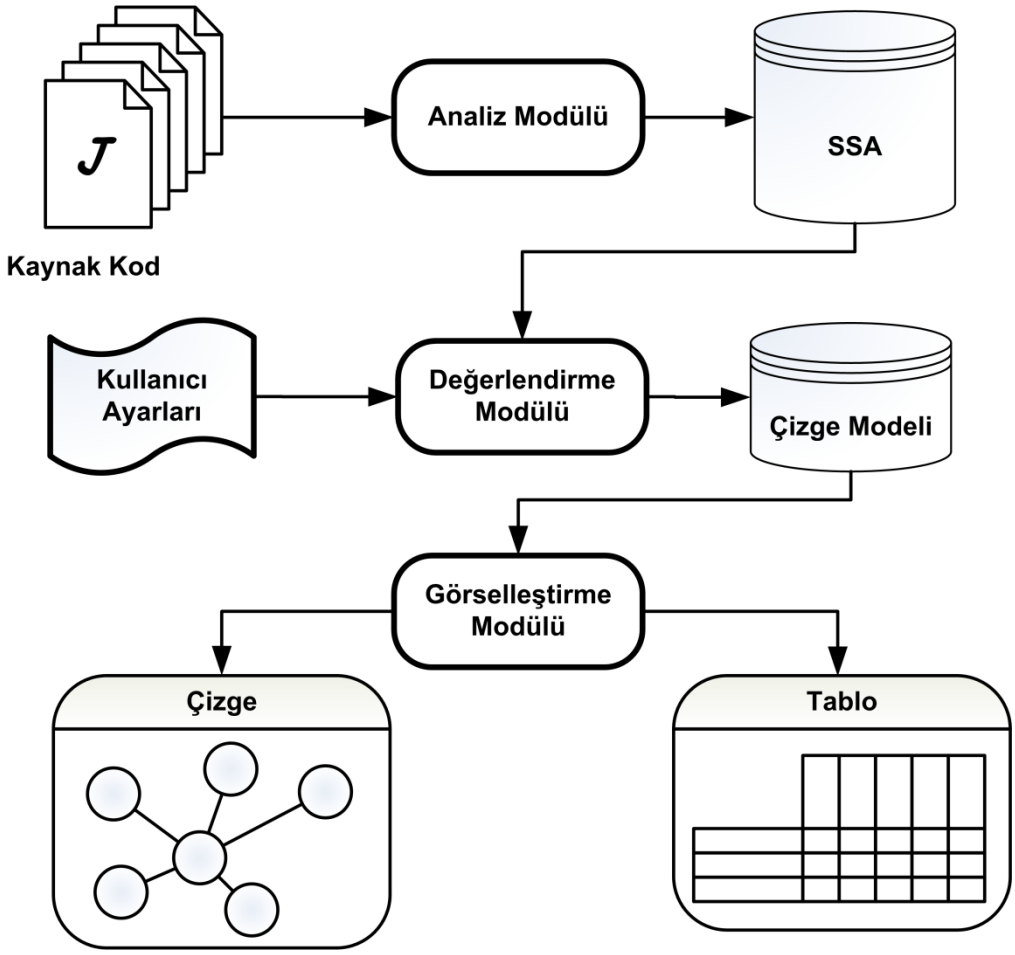


Şekil 3.3 : Model elemanları ve ilişkiler.

3.2 Yazılım Modelinin Çıkarılması

Kaynak kodundan yazılım sınıfları ve bunların arasındaki ilişkileri çıkarmak için bir takım hazır araçlar olsa da, tez kapsamında düşünüldüğü kadar derinlemesine bilgileri çıkaran ve bu bilgileri anlamlı veri yapılarında tutan hazır bir araç yoktur. Çıkarılan bu bilgilerin ayrıntı seviyesi, çizge temsili alt bölümünde ve bir sonraki yazılım kalite nitelikleri bölümünde daha detaylı olarak anlatılmaktadır. Bu amaçla tez kapsamında ortak geliştirilen yazılım analiz aracı E-Quality [51] Eclipse geliştirme ortamına bir eklenti olarak geliştirilmiştir.

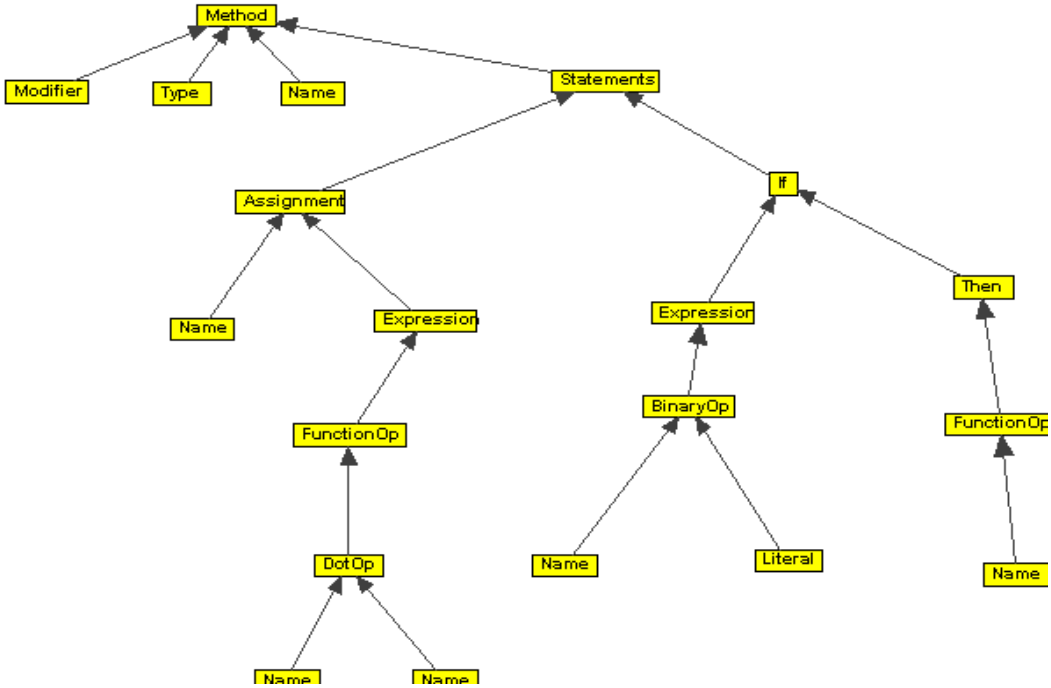
Yazılım analiz aracının sistem mimarisi Şekil 3.4'de verilmiştir. Analiz modülü Eclipse Java Development Tools (JDT) kütüphanesinin oluşturduğu, soyut sentaks ağaçlarını (SSA [Abstract Syntax Tree - AST]) analiz edilerek yazılımın çizge modeli çıkarmaktadır. Soyut sentaks ağaçları belirli bir program dilinde yazılmış kaynak kodun sentaks yapısının soyut bir ağaç temsilidir. Java kaynak kodları ayrıştırıcı (parser) ile soyut sentaks ağaçlarına (Abstract Syntax Tree - AST) çevrilir. Örnek bir SSA Şekil 3.5'de gösterilmiştir.



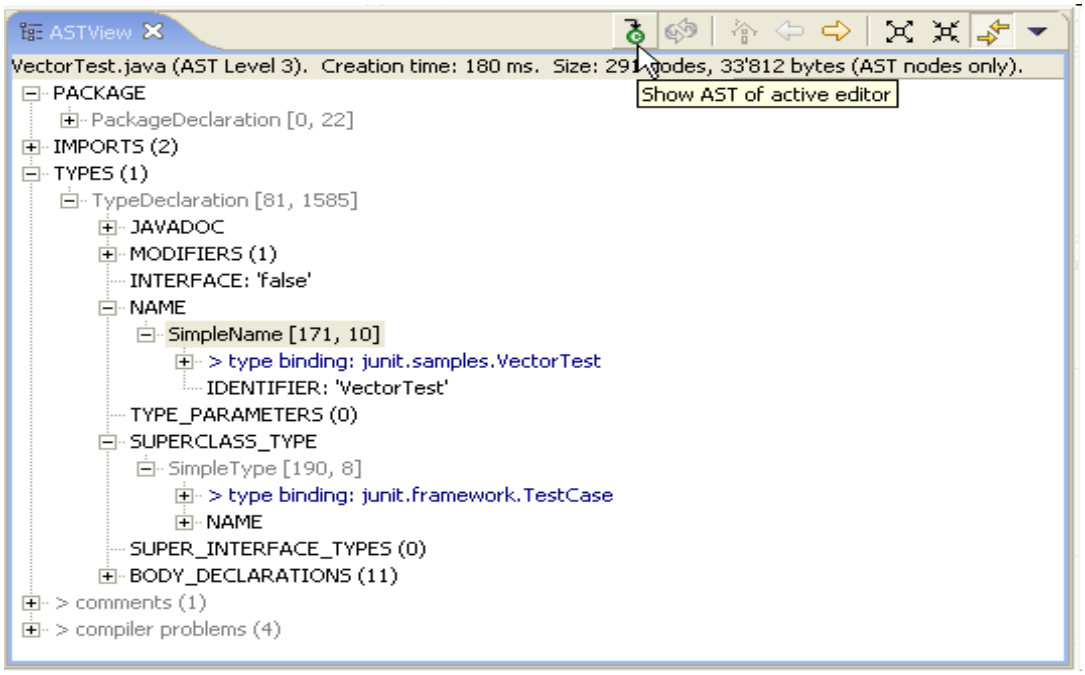
Şekil 3.4 : E-Quality analiz aracının sistem mimarisi.

Soyut sentaks ağacının daha iyi anlaşılması açısından *VectorTest.java* dosyasının Eclipse *ASTView* eklentisinde soyut sentaks ağacı Şekil 3.6'da gösterilmiştir. Yazılım ile ilgili her türlü metrik bu sentaks ağaçlarının uygun şekilde işlenmesi ile elde edilebilir.

E-Quality analiz aracıyla bu soyut sentaks ağacı işlenir ve modelde gerekli olan tüm sınıflar, metotlar, bunların özellikleri ve aralarındaki ilişkiler, ilişkilerin nitelikleri koddan çıkartılır. Analiz aracı bu bilgileri çeşitli tablo ve çizgelerde görselleştirmekte, kullanıcının etkileşimli olarak güncelleme yapmasına izin vermektedir.



Şekil 3.5 : Soyut sentaks ağacı**.**



Şekil 3.6 : ASTView.

4. NESNEYE DAYALI KALİTE NİTELİKLERİ VE METRİKLER

Yazılımlardaki sınıfların ve ilişkilerin yazılımın mimarisindeki yerini anlamak için, yazılım kalite niteliklerine başvurulabilir. Bu bölümde, yazılımın modüler yapısını anlamada ciddi önemi olan bağımlılık uyumluluk gibi modülerliği şekillendiren yazılım kalite nitelikleri ve bunları ölçmede kullanılan sınıf metrikleri anlatılmıştır. Ayrıca tasarım prensip ve kalıplarının kalite nitelikleriyle ilişkileri ve tüm bu kavramların, yazılımın mimari tasarımına etkileri tartışılmıştır. Önerilen yöntem, nesneye dayalı yazılım modelinden modülleri bulurken, sınıf ve ilişkilerin modül içi ve modül arası olma olasılığını belirlerken bu özelliklerden yoğun olarak faydalanmaktadır. Aynı zamanda modülerlik de bir kalite metriği olduğu için yazılım kalitesini değerlendirme, nesneye dayalı yazılım mimarisini anlamak için de iyi bir başlangıç noktasıdır [52].

4.1 Yazılımda Toplam Kalite Yönetimi ve Ölçme Kavramı

Toplam kalite yönetiminin [53], üretim bantlarında kullanılmaya başlamasıyla birlikte, endüstriyel ürünlerde kalitenin sistematik bir biçimde arttığı çeşitli tecrübelerle saptanmıştır. Günümüzde birçok firma toplam kalite sistematiklerini kullanmaktadır. Toplam kalite yönetimi süreçlerin etkin yönetimini amaçlamaktadır. Süreç kısaca, girdileri çıktılara dönüştüren işlemler kümesi olarak tanımlanabilir. Toplam kalite disiplinindeki temel esas, kaliteli ürünlerin ancak iyi tanımlanmış olgun süreçler sonucunda çıktığıdır. Mevcut süreçlerin sürekli geliştirilmesi ve iyileştirilmesi ile daha kaliteli ürünlerin ortaya çıkartılması hedeflenmektedir. Günümüzde toplam kalite prensiplerini temel alan SPICE, ISO ve CMMI gibi uluslar arası standartlar yazılım şirketleri tarafından süreç çalışmalarında sıklıkla kullanılmaktadır.

Sürekli gelişme ve iyileşmeye prensibinin temelinde sürecin planlaması, denetlemesi, çıktılarının ve performansının ölçülmesi en nihayetinde ise sürecin değerlendirmesi vardır. Tanımlı bir süreç boyunca birçok ara ürün veya ürün olarak nitelendirebileceğimiz çıktılar oluşturulmaktadır. Bir yazılım geliştirme sürecini ele

alacak olursak bu çıktılar örneğin; müşteri isterleri, tasarım dokümanları, yazılım kodu, test sonuçları ve bunun gibi diğer çıktılar olmaktadır.

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

4.2 Yazılımın İç Özellikleri

Üretilen bir yazılımın kullanıcılarına (müşterilerine) bakan dış özellikleri esasen yazılımın iç özelliklerinin bir yansımasıdır. Nesneye dayalı yazılımlarda bağımlılık, uyumluluk ve karmaşıklık yazılımın en önemli iç özellikleridir. Eğer bu iç özellikleri doğru ölçüp yorumlanabilinirse kalite arttırabilir. Böyle bir yaklaşımla, henüz yazılım bir ürüne dönüşmeden, bu özellikler değerlendirilerek erken öngörüler yapılabilir. En genel anlamda bağımlılık, yazılım parçaları arasındaki karşılıklı bağımlılığın derecesini, uyumluluk yazılım parçalarının kendi yaptıkları işlerdeki tutarlılığın derecesini, karmaşıklık ise yazılımın içyapısının kavranmasındaki zorluğun derecesini belirtir. Kaliteli yazılımların uyumluluğunun yüksek, karmaşıklığının ve bağımlılığının düşük olduğu yaygın olarak kabul görmüş bir olgudur [54].

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

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

4.2.1 Bağımlılık

Ontolojik terminolojide, bağımlılık (coupling) iki nesneden en az birinin diğerine etki etmesi olarak tanımlanır. A nesnesinin B nesnesine etki etmesi, B'nin kronolojik durumlarının sırasının A tarafından değiştirebilmesidir [55]. Bu tanıma göre Ma'nın Mb ya da Ib üzerinde herhangi bir eylemi, aynı şekilde Mb'nin Ma ya da Ia üzerinde eylemi, bu iki nesne arasında bağımlılığı oluşturur [56].

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

- A sınıfının içinde B sınıfı cinsinden bir üye (referans, işaretçi ya da nesne) vardır.
- A sınıfının nesneleri B sınıfının nesnelerinin metotlarını çağırıyordur.
- A sınıfının bir metodu parametre olarak B sınıfı tipinden veriler (referans, işaretçi ya da nesne) alıyordur ya da geri döndürüyordur.
- A sınıfının bir metodu B tipinden bir yerel değişkene sahiptir.
- A sınıfı, B sınıfının bir alt sınıfıdır [57].

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

- Bir sınıftaki değişim diğer sınıfları da etkiler (maintainability).
- Sınıfları birbirlerinden ayrı olarak anlamak zorlaşır (understandability).
- Sınıfları tekrar kullanmak zorlaşır (reusability).

```
import java.util.Calendar;
public class ResCancelPolicy implements CancellationPolicy{

public Calendar cancelReservation(ReservationType type, Calendar from) {
Calendar result = (Calendar)from.clone();
result.add(Calendar.DATE, 2);
return result;
}
public Calender getDueDate(ReservationType type,Calender from)
//...
Calendar result = (Calendar)from.clone();
result.add(Calendar.DATE, 2);
return result;
}
```

Şekil 4.1 : Örnek kod [58].

Nesneye dayalı programlardaki kapsülleme (encapsulation), bilgi saklama (information hiding) gibi temel özellikler doğrudan bağımlılığı azaltmaya yöneliktir.

Şekil 4.1'de verilen örnek kodda, *ResCancelPolicy* sınıfına ait birçok farklı tipte bağımlılıklara rastlanmaktadır. Örneğin sınıfın bazı metotları *Calendar* Sınıfından parametre almakta, *Calendar* tipinden yerel değişkenler kullanılmakta, bazı metotlarında ise *Calendar* tipinden bir değer geri döndürülmektedir, *Calendar* sınıfının *add(..)* metodu çağrılmakta, *Calendar* Tipinden bir *clone*() metoduyla *Object* tipinden bir nesne *Calendar* tipine dönüştürülmektedir (casting). Benzer şekilde *ReservationType* tipinden bir parametre alınmaktadır. Ayrıca sınıf *CancellationPolicy* sınıfından türediği için, bu sınıfla da bir bağımlılığı olduğu rahatlıkla söylenebilir.

Nesneler iş birliği içinde çalıştıklarından, modüllerin, bazı modüllerle bir tür bağımlılığı olması gerekir. İyi bir tasarımda gereksiz bağımsızlıklara rastlanmamalıdır. Burada önemli olan kaçınılmaz/zorunlu bağımlılıklar ile gereksiz bağımlılıkların nasıl ayırabileceğidir. Metriklerde rastlanan bir diğer bağımlılık konusu da, nesne bağımlılığıdır. Aynı sınıftan bile olsa diğer nesnelere erişiliyorsa bu da nesne bağımlılığını ölçerken hesaba katılır.

Bağımlılık ilgili şu temel soruların cevabı hala çok net değildir.

- Hangi ilişkiler, bağımlılık olarak nitelendirilmelidir?
- Aradaki bağın derecesi/kuvveti nedir?
    - Gözüktüğü yer sayısı
    - Tipi
    - Yeri (parametre, yerel değişken, nitelik değişkeni vs.)
- Nesne Bağımlığı (Object Coupling) ile sınıf bağımlılığı (class coupling) arasında bir fark var mı?
- Gerekli bağımlılık ile gereksiz bağımlılığı nasıl ayırabiliriz?

Tek bir sınıfın izole olarak bağımlılığını ölçmek, tasarım kalitesini ölçmede pek bir işe yaramayabilir. O nedenle bağımlığı servis içi ve servis dışı bağımlılık olarak ayrı değerlendirmek ve kaliteyi de buna göre ölçmek gerekir. Burada diğer bir referans noktası da nesneye dayalı tasarım prensipleri ve kalıplarıdır. Gerekli bağımlılık ile

gereksiz bağımlığı ayırmak için tasarım prensipleri ve tasarım kalıplarından yararlanılabilir. Örneğin yabancılarla konuşma prensibine aykırı bir bağlantı varsa bu gereksiz bağımlılık olarak nitelendirilebilir.

4.2.2 Uyumluluk

Uyumluluk (cohesion), bir sınıftaki metot ve niteliklerin birbiriyle ilgili olmasının ölçüsüdür. Bir sınıftaki metot parametrelerindeki ve nitelik tiplerindeki güçlü örtüşme iyi uyumluluğun göstergesidir. Eğer bir sınıf aynı nitelik değişkenleri kümesi üzerinde farklı işlemler yapan farklı metotlara sahipse, uyumlu bir sınıftır. Eğer bir sınıf birbiri ile ilgili olmayan işler yapıyorsa, birbiriyle ilgili olmayan nitelik değişkenleri barındırıyorsa veya çok fazla iş yapıyorsa sınıfın uyumluluğu düşüktür.

Bunge, benzerliği σ(), iki varlık arasındaki özellik kümesinin kesişimi olarak tanımlamaktadır [55]. $\sigma(X,Y) = p(x) \cap p(y)$. Benzerliğin bu genel tanımından yola çıkarak metotlar arasındaki benzerliğin derecesi, bu iki metot tarafından kullanılan nitelik değişkenleri (instance variables) kümesinin kesişimi olarak tanımlanmaktadır [56]. Elbette metodun kullandığı nitelik değişkenleri metodun özellikleri değildir ama nesnenin metotları, kullandığı nitelik değişkenlerine (instance variables) oldukça bağlıdır.

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

Metotların benzerlik derecesi, hem geleneksel yazılım mühendisliğindeki uyumluluk kavramı (ilgili şeyleri bir arada tutmak) ile hem de kapsülleme (encapsulation) (nesne sınıfındaki veriler ve metotları bir arada paketlemek) ile ilişkilidir. Metotların benzerlik derecesi, nesne sınıfının uyumluluğunun başlıca göstergesi olarak görülebilir. Düşük uyumluluk şu sorunlara yol açar:

- Sınıfın anlaşılması zordur (understandability).
- Sınıfın bakımını yapmak zordur (maintainability).
- Sınıfı tekrar kullanmak zordur (reusability).
- Sınıf değişikliklerden çok etkilenir.

Bağımlılık modüller arası fiziksel ilişkilerle ilgiliyken, uyumluluk tek bir modülle ilgili ve daha mantıksal bir olgudur. Modüllerin yüksek uyumlu olması tercih edilir, çünkü iç bağlar ne kadar ilişkiliyse modül o kadar kolay anlaşılır. Modül ile diğer modüller arasında ne kadar çok bağlantı varsa, modülü anlamak o kadar zorlaşır, yeniden kullanım zorlaşır, hataların izole edilmesi zorlaşır, bu açıdan düşük bağımlılık ve yüksek uyumluluk tercih edilir. Nesneye dayalı programlardaki kapsüleme (encapsulation), bilgi saklama (information hiding), soyutlama (abstraction) gibi özellikler doğrudan uyumluluğu arttırmaya yöneliktir.

4.2.3 Karmaşıklık

Karmaşıklık (Complexity) bir sınıfların iç ve dış yapısını, sınıflar arası ilişkileri kavramadaki zorluğun derecesidir. "Bir nesnenin karmaşıklığı: bileşiminin çokluğudur" [55] Buna göre karmaşık bir nesne çok özelliğe sahip olur. Bu tanıma göre (x, p (x)) nesnesinin karmaşıklığı özellik kümesinin kardinalitesi yani |p(x)|, olur [56]. Karmaşıklık arttıkça tasarımı anlamak zorlaşacak, tasarımda hata olma ihtimali artacak ve yazılımın bakımını yapmak zorlaşacaktır. Tasarım mimarisinin yapısal karmaşıklığı, servisler arası bilgi akışı, servislerin iç yapıları ve dışarıyla ile etkileşimine bakarak ölçüldüğünde, daha doğru ve yorumlanması kolay değerlendirmeler elde edilebilir.

4.3 Yazılım Tasarım Metrikleri

Yazılım tasarım metrikleri temelde yazılımın iç niteliklerini sayısallaştırılmasında kullanılır. Yazılım metrikleri çeşitli açılardan sınıflandırılabilir. Bilgilerin toplanma zamanı açısından statik ve dinamik iki tür metrik sınıfı vardır. Statik metrikler yazılım çalıştırılmadan, yazılımın yapısıyla ilgili belgelerden (Örneğin: kaynak koddan) elde edilebilirler. Dinamik metrikler ise yazılım çalışması sırasında toplanan verilere dayanır. Metrikler ölçmede kullanılan bilgilere göre de sınıflandırılabilir. Bazı metrikler metotların sadece parametre erişimlerine bakarken, bazıları metotlardan tüm veri erişimlerini dikkate alırlar. Metrikler ürettikleri verinin tipi ve aralığına göre de sınıflandırabilir. Bazı metrikler [0,+∞), bazıları sonlu bir aralıkta gerçel ya da tam sayı değerler üretirken, bazıları sınırlı bir aralıkta gerçel sayılar üretebilir. Özellikle [0–1] arası değer üreten metrikler karşılaştırılma açısından daha kullanışlıdır.

Metrikler yazılım kalitesini ölçmede ve dolayısıyla arttırmada, yazılımın bütçe, test, bakım maliyetlerinin tahmin edilmesinde, yazılımdaki hata ihtimali yüksek riskli kısımların bulunmasında, yazılımdaki tasarım kusurlarının belirlenmesinde kullanılır. Literatürde yazılımın bakılabilirliğini tahmin etmede [59],[60],[61], yazılımdaki kusurları bulmada [62],[63],[64], yazılımın hata bağışıklığını tahmin etmede [65], [66],[67],[68],[69] metrikleri kullanan çok sayıda çalışma mevcuttur. Bu bölümde, temel nesneye dayalı yazılım metrikleri incelenmiştir. Geleneksel işleve dayalı yazılım metrikleri (Örneğin: Kaynak kodu satır sayısı, Açıklama Yüzdesi, McCabe Çevrimsel Karmaşıklığı [70] (Cyclomatic Complexity) sınıfların metotlarına uygulanarak kullanılır. Devam eden alt bölümlerde literatürde yaygın olarak kabul görmüş üç nesneye dayalı yazılım metrik kümesi; Chidamber ve Kemerer metrik kümesi, MOOD metrik kümesi ve QMOOD metrik kümesi anlatılmıştır.

4.3.1 Chidamber ve Kemerer metrik kümesi

Chidamber ve Kemerer (CK) 6 temel metrik tanımlamışlardır. Bu metriklerin tanımları ve kısaca hangi özelliklerle ilişkili oldukları aşağıda açıklanmıştır [56].

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

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

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

Alt Sınıf Sayısı - Number of Children (NOC): Sınıftan doğrudan türetilmiş alt sınıfların sayısıdır. Eğer bir sınıf çok fazla alt sınıfa sahipse, bu durum kalıtımın yanlış kullanıldığının bir göstergesi olabilir. Bir sınıfın alt sınıf sayısı tasarımdaki

potansiyel etkisi hakkında fikir verir. Çok alt sınıfı olan sınıfların metotları daha çok test etmeyi gerektirdiğinden bu metrik sınıfı test etmek için harcanacak bütçe hakkında bilgi verir.

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

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

Metotların Uyumluğu - Lack of Cohesion in Methods (LCOM): Bu metriğin birden çok tanımı vardır. Chidamber ve Kemerer ilk olarak şu tanımı yapmışlardır. C1 sınıfının M1,M2, … ,Mn metotları olduğunu ve {Ii} kümesinin de Mi metodunda kullanılan nitelik değişkenleri kümesi olduğunu kabul edelim. Bu durumda LCOM bu n kümenin kesişiminden oluşan ayrık kümelerin sayısıdır (LCOM1 [71]). Daha sonra bu tanım yenilenerek şu tanım getirilmiştir. P hiçbir ortak nitelik değişkeni (I) paylaşmayan metot çiftlerinin kümesi, Q en az bir ortak nitelik değişkeni paylaşan çiftlerin kümesi olsun. Bu durumda LCOM= {|P|>|Q| ise P|-|Q| ; aksi durumda 0} olur (LCOM2 [56]). Literatürde farklı LCOM metrik tanımları da mevcuttur: LCOM3 [72], LCOM4 [73]. Sınıfın uyumluluğunun düşük olması, sınıfın 2 veya daha fazla alt parçaya bölünmesi gerektiğini gösterir. Düşük uyumluluk karmaşıklığı artırır, bu nedenle geliştirme aşamasında hata yapılma ihtimali yükselir. Ayrıca metotlar arasındaki ilişkisizliklerin ölçüsü sınıfların tasarımındaki kusurların belirlenmesinde de yardımcı olabilir.

4.3.2 MOOD metrik kümesi

MOOD metrik kümesi [74], nesneye dayalı yöntemin temel yapısal mekanizmalarına dayanır. Bunlar kapsülleme (MHF AHF), kalıtım (MIF, AIF), çok şekillilik (PF) ve mesaj aktarımı (CF) olarak sıralanır.

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

Nitelik Gizleme Faktörü - Attribute Hiding Factor (AHF): Sistemde tanımlı tüm sınıflardaki görünür (erişilebilir) niteliklerin, tüm niteliklere oranıdır. Bu metrikle sınıf tanımının görünürlüğü ölçülür. Burada kalıtım ile gelen metotlar hesaba katılmamaktadır.

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

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

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

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

4.3.3 QMOOD metrik kümesi

QMOOD metrikleri [75], Bansiya ve Davis tarafından yazılımın toplam kalite endeksi hesaplanması için 4 seviyeden oluşan hiyerarşik bir model içinde tanımlanmıştır. En alt seviyede sınıf, metot gibi nesneye dayalı yazılım bileşenleri vardır. 3. seviyede QMOOD metrikleri vardır. QMOOD metriklerinden, karmaşıklık, uyumluluk gibi bir üst seviyedeki yazılım özellikleri hesaplanır. Bu yazılım

özelliklerinden de anlaşılabilirlik, esneklik, tekrar kullanılabilirlik gibi en üst seviyedeki kalite nitelikleri hesaplanır. Son olarak kalite niteliklerinden toplam kalite endeksi hesaplanır. QMOOD çalışmasında tanımlanan seviyeler ve bağlar Şekil 4.2‘ de gösterilmiştir. Bu bölümde 11 QMOOD metriğinden 8 tanesi aşağıda verilmiştir:

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

Metotlar Arası Uyumluluk - Cohesion Among Methods (CAM): Metotların imzaları arasındaki, benzerliğin ölçüsüdür. Tam tanımı Bansiya'nın metrik kümesi çalışmasında yer almamaktadır. Literatürde metotların imzalarına bakarak uyumluluk ölçen farklı metrik tanımları bulunmaktadır. Bunlar temelde metotların parametre kullanım matrisi üzerinde çalışırlar ve imzalardaki uyumluluğu tam uyumlu olmaya yakınlıkla kıyaslayarak ölçerler. Metotlar ve tüm parametreler numaralandırılır, i. metodun imzasında, j. parametre kullanılıyorsa, matriste Mij 1 aksi halde 0 olur. CAMC [76] parametre matrisindeki 1 sayısına bakarken, NHD [77] matris satırları arasındaki Hamming uzaklıklarına bakar. SNHD [78], ise NHD metriğinin olası en küçük ve büyük değerine göre ölçeklenmiş halidir.

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

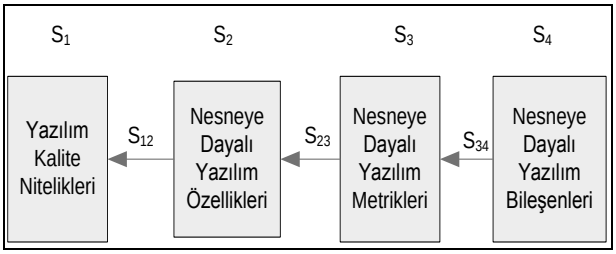
Veri Erişim Metriği - Data Access Metric (DAM): Sınıfın özel (private) ve korumalı (protected) niteliklerinin tüm niteliklere oranıdır. Bu metrik yazılımın kapsülleme özelliğini gösterir.

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

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

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

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



Şekil 4.2 : Seviyeler ve seviyeler arası bağlar.

4.4 Metriklerin Yazılım Modülerlik Kalitesini Arttırma Yöntemleriyle İlişkisi

Bu bölümde yazılımın modülerlik kalitesini iyileştirmeye yönelik yazılım dünyasınca kabul görmüş, sezgisel tecrübeler anlatılmıştır. Yazılım metriklerinin kalitesiz kod göstergelerini tespit etmede yardımcı olması beklenir. Aynı şekilde tasarım prensipleri ve tasarım kalıpları yerinde kullanıldığında yazılımın kalitesinin arttığı metriklerle de doğrulanmalıdır. Tasarım prensipleri ve tasarım kalıplarının kullanımı tez çalışmasında yazılım tasarımını anlamakta, sınıflar arasındaki ilişkileri değerlendirmede ve yazılımdaki servisleri bulmakta referans noktası olacaktır.

4.4.1 Düşük kaliteli kod göstergeleri

Kalitesiz kod göstergeleri (bad smells) kavramı ilk olarak, Martin Fowler tarafından var olan kodların iyileştirilmesi için ortaya koyulmuştur [79]. Yazılım yaşam döngüsü boyunca sürekli değişim halindedir. Bu değişiklikler bazen dağınık kodları toparlama bazen de uygulamanın yapısını değiştirme şeklinde olabilir. Yazılım geliştiriciler, kod üzerinde değişiklik yaparken kalitesizlik göstergelerini fark ederek gerekli düzeltmeleri yapmalıdırlar. Bu göstergelerin büyük bir çoğunluğu metrik değerlerini olumsuz yönde değiştirirken, bazıları metriklerle analiz sırasında gözden kaçabilir. Yazılım metriklerinden düşük kaliteli kod göstergelerini tespit etmekle ilgili başlangıç düzeyinde bir çalışma [80]'de verilmiştir. Aşağıda düşük kaliteli kod göstergelerinden bazıları verilmiştir [81].

Tekrar eden kod parçaları (yazılım klonları): Aynı kod parçasının birden çok yerde gözükmesidir. Bu kod parçalarının metot haline getirilmesi tavsiye edilir.

Ancak tanımlanan metriklerle bu tasarım kusurunu fark edecek bir yöntem bulunmamaktadır.

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

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

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

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

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

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

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

4.4.2 Tasarım kalıpları

Tasarım kalıpları (Design Patterns) ilk olarak mimar Christopher Alexander tarafından kaliteli mimari yapılardaki ortak özelliklerin sorgulanması ile ortaya çıkmıştır [82]. C. Alexander yaptığı araştırmalar sonucunda beğenilen (kaliteli) yapılarda benzer problemlerin çözümünde, benzer çözüm yollarına başvurulduğunu belirlemiş ve bu benzerliklere tasarım kalıpları adını vermiştir. 1980'lerde, Kent Beck bu çalışmalardan esinlenerek yazılım kalıplarını ortaya atmıştır [83]. Mimarlar gibi yazılım geliştiriciler de, tecrübeleriyle, birçok ortak problemin çözümünde uygulanabilecek, prensipler ve deneyimler (kalıplar) oluşturmuşlardır [84], [85]. Bu kalıplar yazılım mimarisinin şekillenmesinde de önemli rol oynamaktadır. Bu konudaki ilk önemli çalışma dört yazar tarafından hazırlanan bir kitap olmuştur [86]. Bu yazarlara dörtlü çete (Gang of Four) adı takılmış ve ortaya koydukları kalıplarda GoF kalıpları olarak adlandırılmıştır. GoF kalıpları üç gruptan oluşmaktadır:

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

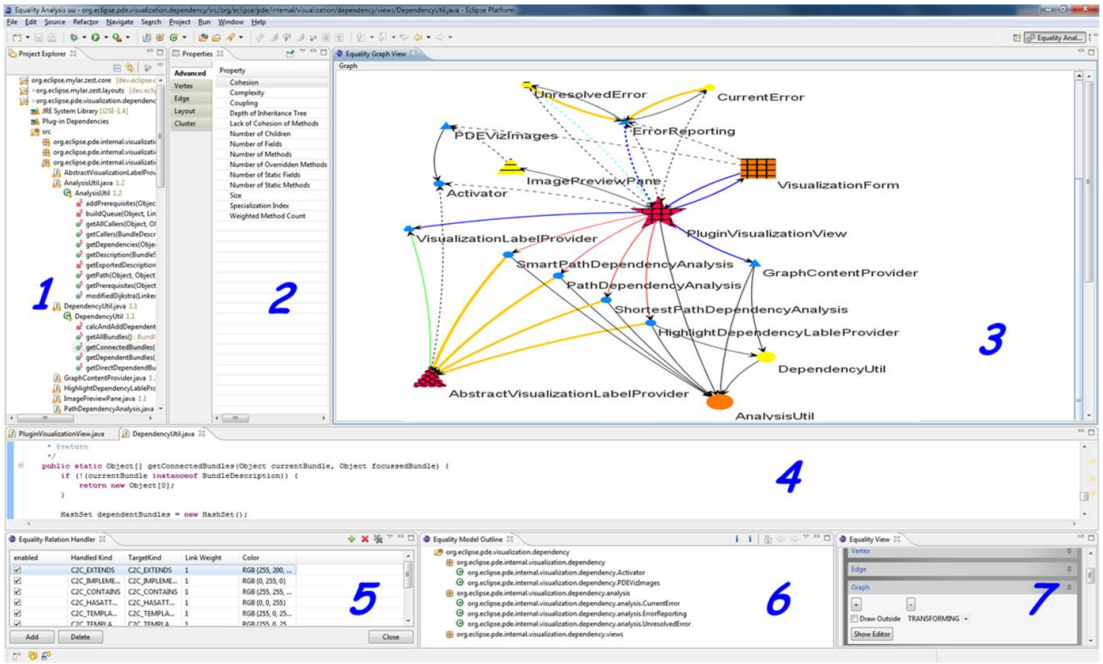
Yapısal Kalıplar: Bu gruptaki kalıplar, sınıf ve nesnelerin kompozisyonuyla ilgilidir. Genel olarak arayüzler oluşturmak için kalıtımdan faydalanırlar ve yeni işlevler kazandırmak için nesnelerin yapılarını oluşturacak uygun yöntemler önerirler. Adaptör, Köprü, Ön yüz (Facade) bu gruptaki bazı kalıplardandır.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

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

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

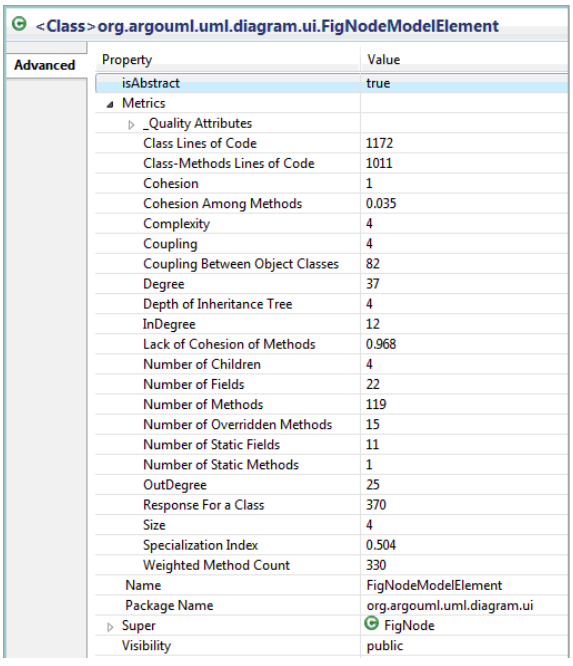
4.5 Analiz Aracında Üst Düzey Kalite Niteliklerinin Kullanımı

Analiz aracında sınıfların metrikleri ve diğer özellikleri Eclipse ortamında Şekil Şekil 4.3'deki gibi gösterilmektedir.



Şekil 4.3 : E-Quality yazılımın kullanıcı arayüzleri [51].

Burada 1 numaralı pencere yazılım kodunu ve projeleri gezmede kullanılan penceredir. 4 numaralı pencere kaynak kodunu düzenleme penceresidir. 3 numaralı pencerede ise yazılımın sınıf düzeyindeki çizgesi gösterilmektedir. Bu çizgede sınıflar düğümlerle, aralarındaki ilişkiler ise farklı tipteki ilişkilere karşılık düşen ayrıtlarla gösterilmektedir. Çizge modeli gerekli tüm bilgileri ve bağları mantıksal olarak tuttuğundan, bu üç pencere arasında, seçilen birimlere karşılık düşen kod parçası, düğüm, kaynak dosyası arasında etkileşimli olarak gezilebilir. Bu çizgenin görüntüsü, istenilen düğümlerin süzülmesi, konumlarının değiştirilmesi ile geliştiriciler tarafından düzenlenebilir. 5 numaralı pencerede hangi ilişkilerin görselleştirileceği, hangi renk ve kalınlıkla temsil edileceği belirlenebilmektedir. İki düğüm arasında birden fazla ilişki tipinin olması durumunda kural sırası ayrıtın rengini belirlemektedir. Ancak yine de ilişki bilgisi ayrıtın üzerine gelindiğinde kullanıcı tarafından erişilebilir. 6 numaralı pencere yardımıyla çizgede bulunan yazılım sınıflarının listesi çeşitli özelliklerine göre süzülerek hızlıca aranabilmekte ve toplu olarak seçilebilmektedir. 7 numaralı pencerede ise, farklı yöntemlerle otomatik düğüm konumlandırma, görüntüyü küçültme büyültme vb. çizgenin çizimiyle ilgili çeşitli ayarlar yapılmaktadır. 2 numaralı pencerede kodda ya da çizge üzerinde seçilen sınıf, paket, ya da tüm yazılımın koddan çıkarılan analiz metrikleri listelenmektedir. Sınıf için olan örnek bir metrik görünümü Şekil 4.4'de ve bu metriklerin açıklamaları Çizelge 4.1'de verilmiştir.



Şekil 4.4 : Metriklerin ve sınıf özelliklerinin analizde gösterilmesi**.**

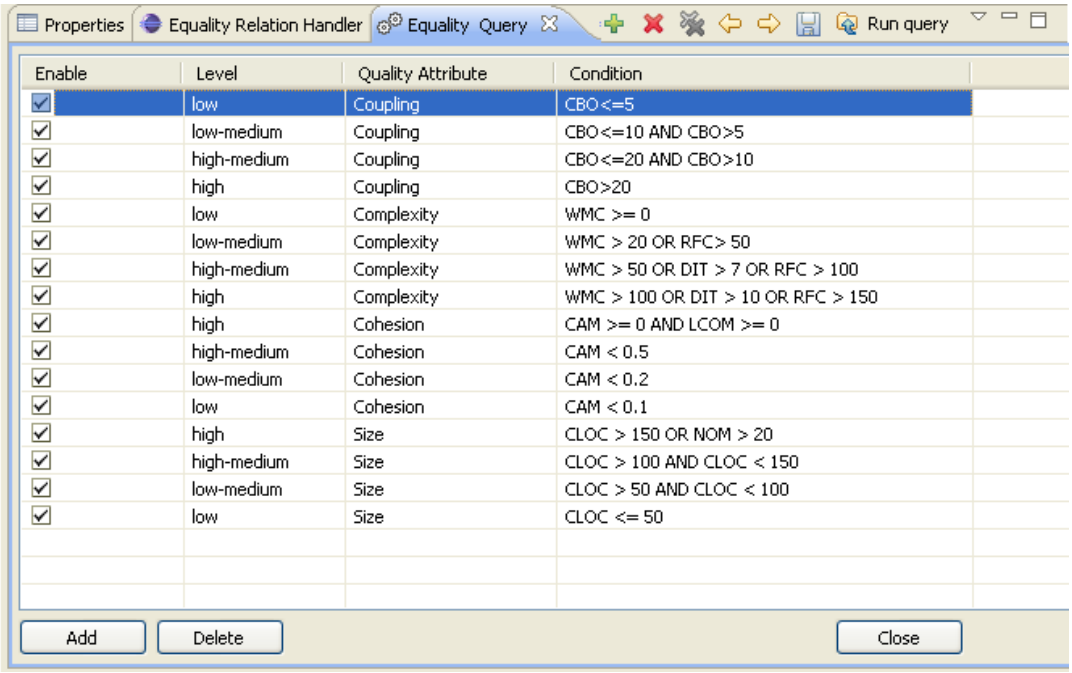
Çizelge 4.1 : Metrik listesi.

İsim	Açıklama
CAM	Metotlar arası uyumluluk
CBO	Nesne sınıfları arası bağımlılık
DIT	Türetim ağacının derinliği
In-Degree	Giren ayrıt sayısı
LCOM	Metotların uyumluluğu
LOC	Kod satır sayısı
NOC	Alt sınıf sayısı
NOF	Nitelik değişkeni sayısı
NOM	Metot sayısı
NORM	Örtülen metot sayısı
NOSF	Statik nitelik değişkeni sayısı
NOSM	Statik metot sayısı
Out-Degree	Çıkan ayrıt sayısı
RFC	Sınıfın tetiklediği metot sayısı
SI	Özelleşme endeksi
WMC	Sınıfın ağırlıklı metot sayısı

2 numaralı pencerede görüldüğü gibi, sınıflar arasında metriklerin geliştiriciler tarafından tek başlarına yorumlanması zordur, bu nedenle analiz aracında metrikler, dört temel üst düzey kalite niteliğine (Bağımlılık, uyumluluk, karmaşıklık, boyut) dönüştürülmektedir. Her bir kalite niteliği de düşük, düşük-orta, yüksek-orta ve yüksek olmak üzere dört kategoriye ayrılmaktadır.

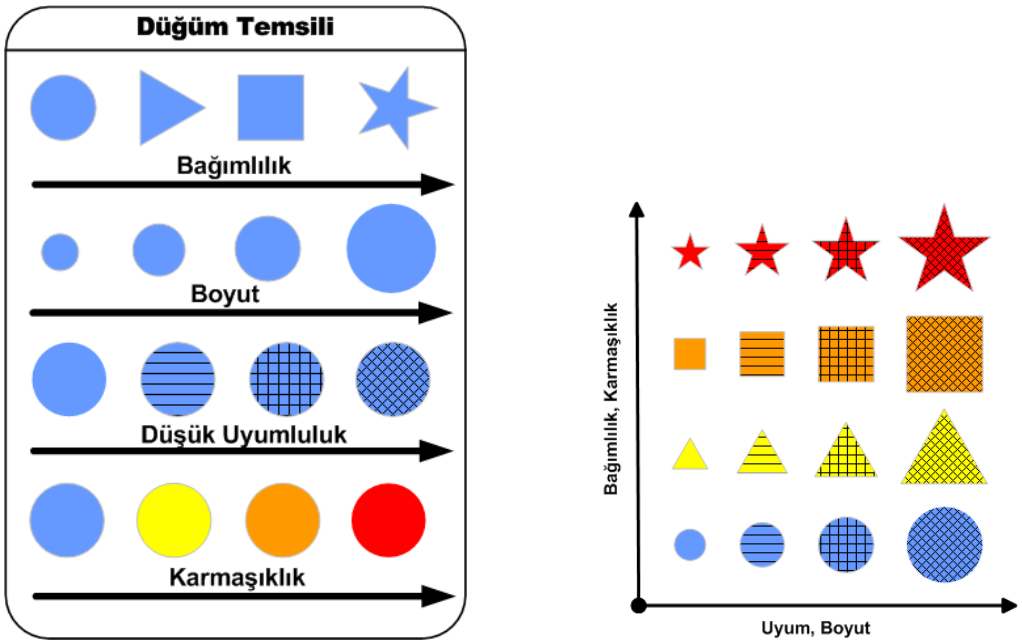
Metriklerin daha kolay anlaşılabilir üst düzey kalite niteliklerine dönüştürülmesi için metrikler üzerinde SQL benzer kurallar kullanıcı tarafından tanımlanabilmektedir. Şekil 4.5'de analiz aracındaki örnek metrik sorgu görünümü verilmiştir. Kurallar arasında çelişki olması durumunda, örneğin bir sınıf bir kurala göre yüksek bağımlılığa sahip iken, bir başka kurala göre düşük kurala sahip gibi gözükebilir, bu durumda üst düzey kalite nitelikleri kuralların sıralamasına göre belirlenmektedir.

Yazılım çizge şeklinde gösterilip kümelere ayrıldığında, kritik sınıfları ve sorumlulukların dağılımını daha iyi anlayabilmek için yazılım aracına üst düzey kalite niteliklerinin görselleştirilmesi de eklenmiştir. Bu düğümleri kolayca fark edilirse, modül yapısını bulmayı olumsuz yönde etkileyebilecek düğümler çizgeden kaldırılarak, modül bulma sonuçları iyileştirilebilir ya da modül bulma sonuçları daha iyi analiz edilebilir. Şekil 4.6'da görselleştirme özetlenmiştir.



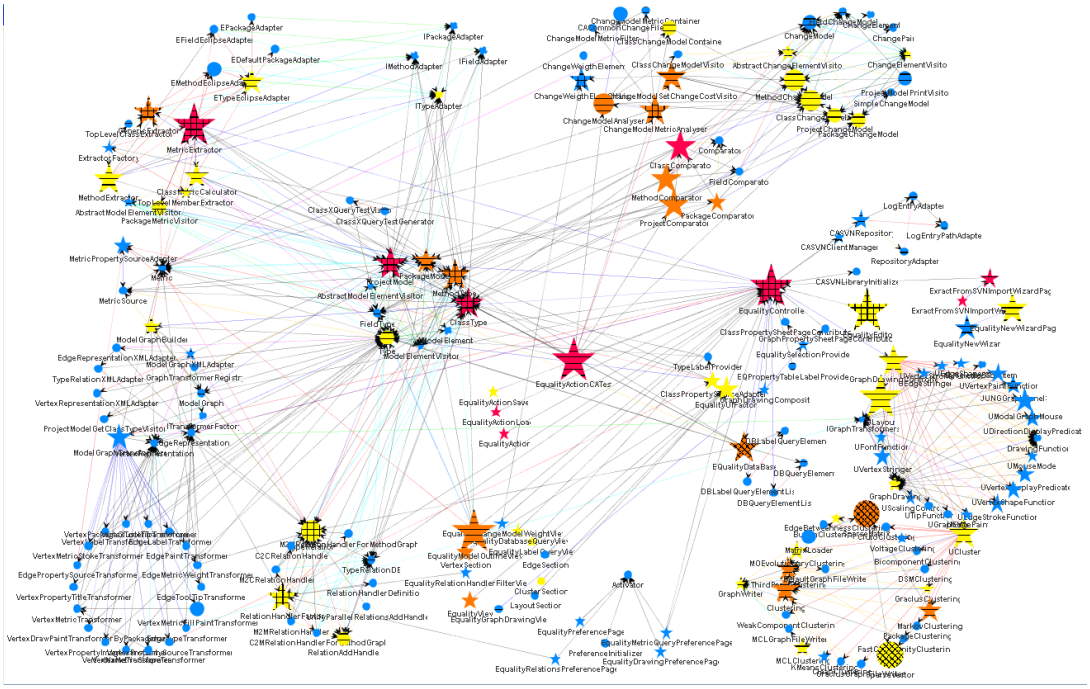
Şekil 4.5 : Metriklerin üst düzey kalite niteliklerine dönüştürülmesi.

Her bir kalite niteliğine 2 boyutlu şeklin fiziksel bir özelliği atanmıştır. Bu atama mümkün olduğunca insan algısına yatkın ve kolay hatırlanabilir Örneğin sınıfın kaynak kodunun boyutu büyüdükçe kademeli olarak, düğümün boyutu da artmaktadır. Benzer şekilde, şeklin kenar sayısı artması, etkileşime girdiği daha çok modül olduğu izlenimi vermekte ve bağımlığı artmaktadır. Uyumluluk şeklin içindeki dokunun artması ile karmaşıklık ise, soğuktan sıcağa renk değişimi ile gösterilmiştir.



Şekil 4.6 : Üst düzey kalite niteliklerinin görselleştirilmesi.

Örnek olarak geliştirilen yazılım aracının yaklaşık 200 düğümlü yazılım çizgesinin metriklerle görselleştirilmiş hali Şekil 4.7'de gösterilmiştir. Bu çizgede düğümler paketlere göre yerleştirilmiştir. Görüldüğü gibi çizgenin büyük bölümü küçük boyutlu, düşük bağımlılıklı (yuvarlak) ve az karmaşık (sarı-mavi) şekillerden oluşmaktadır. Az sayıda düğüm ise, yüksek bağımlılıklı karmaşık sınıflardır (büyük kırmızı ya da portakal yıldız şekiller). Seyrek dağılmış büyük yıldızlar genelde, modüllerin önyüz erişim noktaları (Facade), denetçi (Controller), test ve kullanıcı arayüz kısımlarıdır. Hatta bu özellikler çoğu zaman sınıfların adlarına da yansımaktadır. Merkezin hemen solunda bulunan 4 yıldız ise modelin çekirdeğini oluşturan kısımdır. Bu kısımları çizgeden çıkartmanın modül bulmaya etkisini incelemek oldukça yararlı olmaktadır.



Şekil 4.7 : Örnek bir yazılım çizgesinin metriklerle görselleştirilmesi.

5. YAZILIM ÇİZGE KÜMELEME

Nesneye dayalı yazılım sistemi uygun bir çizge modeliyle ifade edildikten sonra, yazılımda modül bulma problemi, matematiksel açıdan bu çizgedeki kümeleri bulma problemine dönüşür. Tez kapsamında bu amaçla literatürdeki çizge kümeleme kavramları derinlemesine incelenmiştir. Bu bölümde ilk olarak sosyolojiden moleküler biyolojiye farklı bilimsel literatürdeki çizge kümeleme kavramları ve uygulama alanları anlatılmıştır. Bu kavramların ardından temel çizge kümeleme yaklaşımları hakkında bilgiler verilmiştir. Bu kavramlar modüllerin ya da çizge uzayındaki yazılım kümelerinin anlaşılması açısından önemlidir. Bu yaklaşımlarda ilginç olan, neredeyse çizge kümeleme probleminin farklı uygulama alanı kadar farklı çizge kümeleme yöntemi olmasıdır. Bunun temel nedeni çizgedeki kümelerin ve çizge karakteristiklerinin uygulama alanına göre farklılık göstermesidir. Yazılım çizgelerine hangi kümeleme yaklaşımlarının uygun olduğunun belirlenmesi açısından, yazılım çizgelerinin özellikleri takip eden bölümde aktarılmıştır. Son olarak, tez kapsamında geliştirilen yazılım analiz aracında, modülerlik analizleri için yazılım kümelerinin nasıl etkin şekilde görselleştirildiği anlatılmıştır.

5.1 Çizge Kümeleme Kavramları

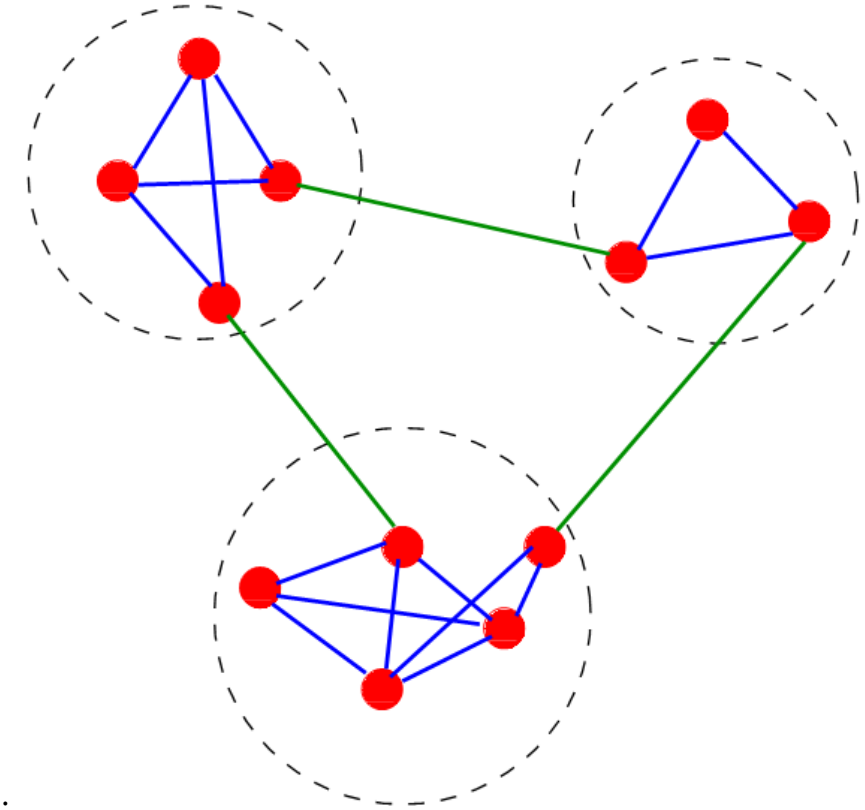
Bir çizge içinde küme (cluster) ya da topluluk yapısı (community structure) ilk olarak sosyal bilimlerde ortaya atılmıştır. Verilen bir çizgede $G=(V,E)$, küme bu çizgenin bir alt çizgesi olan $G'=(V', E')$ olarak tanımlanır. Bu alt çizgede düğümler sıkı bağlıdır ve düğümler arası uyumluluk yüksektir. Bir grup düğümün uyumluluğunu tanımlamanın farklı yolları olsa da genelde küme içi bağların sayısının çok, kümeler arasındaki bağların ise az olması şeklinde tanımlanır [87]. Bu kümeler çizgenin oldukça bağımsız parçaları olarak düşünülebilir. Kümeleri belirlemek sistemlerin genelde çizge şeklinde temsil edildiği, sosyoloji, biyoloji, bilgisayar bilimleri gibi disiplinlerde oldukça önemlidir. Çizge kümeleme problemi oldukça zor bir problemdir, son yıllarda üzerinde birçok disiplinler arası bilim insanı çalışmasına rağmen, henüz tatminkâr bir şekilde çözülememiştir. Çizgedeki kümeler

bazen topluluk ya da modül olarak da adlandırılır. Kümeler, ortak özellikleri paylaşan ya da çizgede benzer roller oynayan düğüm gruplarıdır.

Toplumlarda da bu tür grup organizasyonlarına sıklıkla rastlanır, aileler, çalışma ve arkadaş ağları, köyler, kasabalar, ülkeler… bu grupların bulunduğu örneklerdendir. İnternetin hızlı yaygınlaşması da sanal öründe (Web) yaşayan sanal grupların oluşmasını sağlamıştır [88],[89],[90],[91]. Topluluklara, biyoloji, bilgisayar bilimleri, mühendislik, ekonomi, politika gibi ağ tabanlı sistemleri içinde bulunduran alanlarda rastlanır. Örneğin protein etkileşimlerinde, hücrede aynı özel görevi yerine getiren proteinlerin aynı grupta yer aldığı görülmektedir [92],[93],[94]. Web üzerinde ilgili ya da aynı konuya ilişkin sayfalar aynı grupta yer almaktadır [95]. Pazarlamada benzer ürünleri satın alan müşteri profilerini belirleyerek kârlılığı arttırmada da kümelemeden faydalanılır.

Modülleri ve bu modüllerin sınırlarını belirlemek, düğümlerin modüllerin içindeki yapısal pozisyonlarına göre sınıflandırılmasına izin verir. Küme içinde merkezi bir noktada bulunan düğümler, örneğin diğer grup üyeleriyle çok fazla ayrıtı olan, önemli bir kontrol işlevini yerine getiriyor olabilir. Modüller arasında yer alan düğümler ise aracılık ve bilgi alışverişinin yönetiminde kritik rol oynayabilirler. Böyle bir sınıflandırma sosyal [96], [97] ve metabolik ağlarda mantıklı sonuçlar ürettiği araştırılmalarda görülmüştür [98]. Şekil 5.1'de örnek bir çizge üzerinde kümeleme gösterilmiştir [99].

Topluluk yapısıyla ilgili ilk çalışma Weiss ve Jacobson tarafından yapılmıştır [100]. Bu çalışmada amaç, bir devlet kurumundaki çalışma gruplarını bulmaktır. Matris olarak kurum üyeleri arasındaki çalışma ilişkileri kullanılmıştır. Çalışma grupları farklı gruplarla çalışan üyelerin silinmesiyle ayrıştırılmıştır. Buradaki gruplar arasındaki köprüleri kesme fikri, birçok modern algoritmanın da alt yapısını oluşturmaktadır. Daha önce başka bir çalışmada sosyal grupların sosyal bağları ifade eden matrisin satır ve sütunlarını uygun şekilde düzenleyerek ortaya çıkarılabileceğini göstermiştir [101]. Kümeleri bulmak bilgisayar bilimlerinde de oldukça kullanılmaktadır. Paralel hesaplamada, görevleri işlemcilere, aralarında iletişim minimum olacak ve hesaplamanın hızlı yapılmasına izin verecek şekilde dağıtmak oldukça önemlidir. Bu işleri eşit sayıda işlemciye, farklı işlemcilerdeki görevler arasındaki bağlantı minimum olacak şekilde sağlanabilir ki bu bir çizge parçalama problemidir.

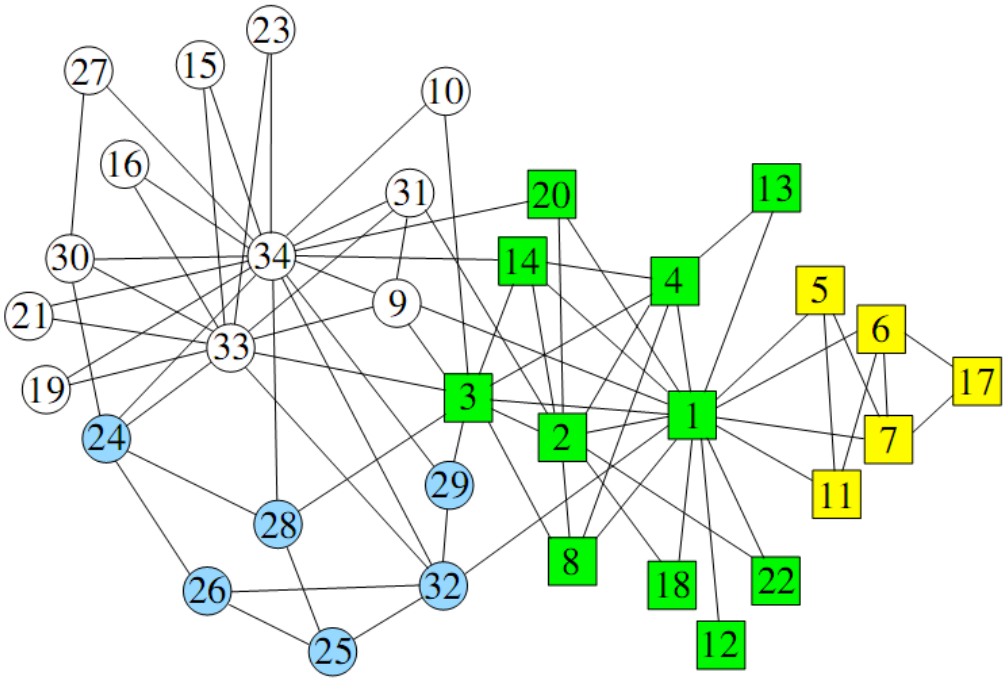


Şekil 5.1 : Örnek çizge kümeleri [99].

2002'de Girvan ve Newman, yeni bir algoritma önerdiler. Buna göre kümeler arasındaki ayrıtlar belirlenmekte ve bu ayrıtların ardarda silinmesiyle kümeler birbirinden izole olmaktadır. Kümeler arasındaki ayrıtları merkeziyet (centrality) değerlerine göre belirlenmektedir. Merkeziyet ölçüsü, ayrıtın arada bulunma ölçüsü olarak adlandırılmaktadır, öyle ki ayrıtların çizge üzerinde en kısa yoldan aktarılan sinyallerin iletimindeki rolleriyle ilişkilidir. Bu çalışmadan sonra bu alanda yapılan çalışmalar oldukça artmış; fizik, bilgisayar bilimleri, doğrusal olmayan (nonlinear) dinamik, sosyoloji ve ayrık matematik gibi farklı disiplinler konuya dâhil olmuştur. Her disiplinde konuya kendi yeni araç ve yöntemlerinin kullanımını açmıştır (spin modelleri, en iyilime, rastgele yürüyüşler, süzme, senkronizasyonlar gibi…).

Gerçek dünyada, küme yapısına sıklıkla rastlanır. En yaygın uygulama alanlarından birisi sosyal ağlardır. Şekil 5.2'de, kümeleme algoritmalarında değerlendirme kriteri olarak yaygın olarak kullanılan, Zachary karate kulübünün çizgesi verilmiştir. Çizge 34 düğüm içermektedir. Düğümler Amerika'daki bir karate kulübünün üyelerini temsil etmektedir. Ayrıtlar ise bu üyelerin 3 yıllık bir süre zarfında, kulüp dışında aktivitelerde beraber gözlemlendiklerini temsil etmektedir. Belirli bir zaman sonra kulübünün başkanı ve öğretmeni anlaşmazlığa düşmüş ve kulüp ikiye ayrılmıştır

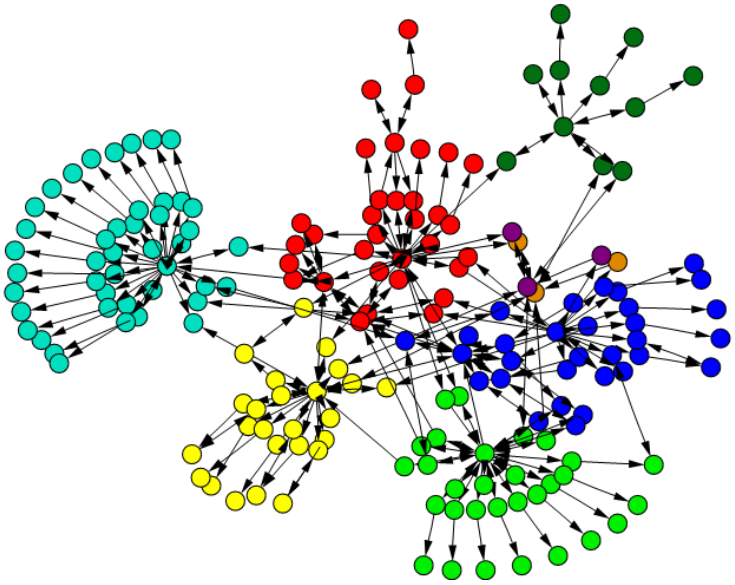
(Yuvarlak düğümler ve kare düğümler). Burada soru, bu grup ikiye ayrılmadan önce, verilen çizge üzerinden giderek bu ayrışmanın tahmin edilip edilemeyeceğidir. Bu çizgede, görsel olarak hemen göze çarpan özellikler bir grubun 33 ve 34 (başkan) numaralı düğümlerin etrafında diğer grubun ise 1 (öğretmen) numaralı düğümün etrafında toplandığıdır. Diğer yandan bazı düğümlerin iki ana grup arasında yer aldığı gözükmektedir (3, 9, 32, 14). Bu düğümler genelde kümeleme algoritmaları tarafından yanlış kümelere yerleştirilmektedir.



Şekil 5.2 : Zachary karate kulübü [102].

Bir sistemin birimleri arasındaki ilişkiler her zaman çift yönlü olmaz. Birçok durumda ilişkiler yönlüdür ve sistemi anlama açısından bu yönler önem arz eder. Şekil 5.3'de bu tür çizgelere örnek olarak bir örün ağının (WWW) çizgesi verilmiştir. Bu çizgede düğümler örün (web) sayfalarını, ayrıtlar ise sayfalar arasındaki bağlantıları (hyperlinks) temsil etmektedir. Daha öncekilerden farklı olarak bu çizgede ayrıtlar yönlüdür, yani sayfa X, sayfa Y'ye bağlantı içerirken, sayfa Y ters yönde bir bağlantı içermeyebilir. Bu durum öründe sıklıkla rastlanır. Burada kümeler benzer konudaki sayfa gruplarını oluştur ve arama sonuçlarında sayfalara sıra vermede yararlanılır [103]. Yönlü çizgelerde, ayrıtları yönsüz düşünerek kümeleme yapılabilse de, ayrıtların yönünü ihmal etmek hata payı çok yüksek sonuçlar üretebilir [104], [105]. Yönlü çizgeler üzerinde kümele yöntemi geliştirmek oldukça zordur. Örneğin, yönlü çizgeler asimetrik matrislerle temsil

edilirler, bu yüzden spektral analizleri çok daha zordur [106]. Yazılımı temsil eden çizge modelinide de ayrıtlar yönlüdür.



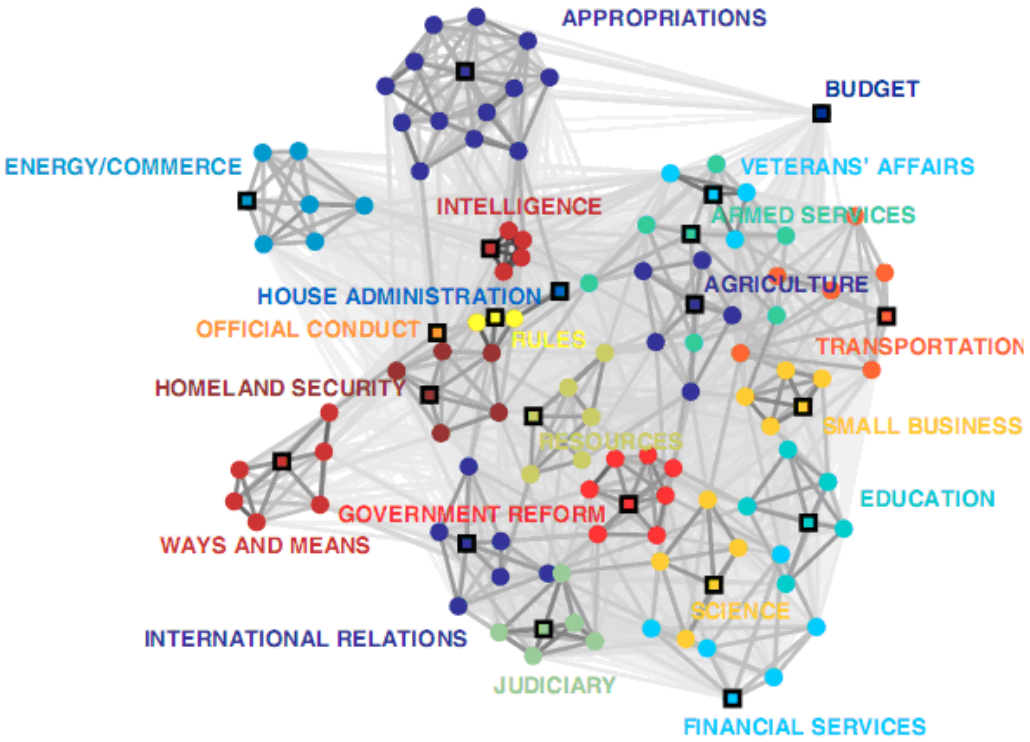
Şekil 5.3 : Teknolojik ağlardaki kümeleme [14].

Ayrıtların yönlü olmasının yanında diğer bir zorlayıcı olgu ise kümeler arasında örtüşme (overlapping) olmasıdır. Birçok gerçek ağda, düğümler birden fazla kümeye dâhil olabilir. Parçalamanın (Partition) tanımı gereği, bir düğüm birden fazla kümeye ait olamaz, dolayısıyla burada örtüşen kümeleri ifade etmek için örtü (cover) terimi kullanılır. Sosyal ağlarda örtüşme sıklıkla rastlanır. Yazılım çizgelerinde de örtüşme nadir de olsa rastlanabilecek bir durumdur.

Diğer yandan, çizgelerdeki düğümler çoğu zaman tek tiptir (tek parçalı çizge - unipartite). Ancak bazı durumlarda çizgedeki düğümler çok tipli olabilir (çok parçalı - multipartite) ve ayrıtlar farklı tipteki düğümleri birleştirebilir. Buna örnek olarak bir yazılımdaki dosyaları ve sürüm kontrol sisteminde yapılan değişim kayıtlarını verebiliriz. Eğer bir değişim kaydında (CVS commit), o dosya değiştirilmişse, dosya ile değişim kaydı arasında bir ayrıt vardır. Beraber değişen dosyaların kodda mantıksal parçaları ifade ettiği bu çizge üzerinde yapılacak kümeleme ile bulunabilir. Yazılım çizge modelimizde düğümler tek tip sınıfları ifade etse de, daha farklı denemelerde (mesaj çağrı zinciri [call graph] çıkarmak gibi) çok tipli çizgelerle de çalışıldığında, bu tip çizge kümele algoritmalarına ihtiyaç duyulacaktır ki bu kümeleme problemini daha da zor hale getirmektedir.

Tez kapsamında önerilen yazılım çizge modelinde kullanılan bir başka nokta da, ayrıtların etiketli (labeled) ve ağırlıklı (weighted) olmasıdır. Örneğin Şekil 5.2'deki Zachary karate kulübünde üyelerin dışarıda birlikte görülme sayısı ayrıtların ağırlığı olabilir. Ağırlık bilgisi çizgeyi daha doğru temsil ettiğinden, kümelemede yararlanılması doğruluğu daha hassas sonuçlar elde etmek açısından önemlidir. Birçok durumda ağırlıksız algoritmalar, ağırlıklı çalışabilecek şekilde uyarlanabilmektedir. Ancak ayırtların etiketli olmasını göz önünde bulunduran bir yöntem bulunmamaktadır. Bu nedenle tezin ilerleyen kısımlarında ayrıntılandırıldığı gibi bu çalışmada farklı etiket ve niteliklere göre uygun ağırlıklar verilmesi üzerinde derinlemesine çalışılmıştır.

Bir diğer önemli nokta çizgelerin hiyerarşik olabileceğidir, örneğin Şekil 5.4'de Amerikan hükümeti temsilciler meclisinde komiteler (kare düğümler) ve alt komiteler (daire düğümler) gösterilmiştir. Burada komiteler ile alt komiteler arasında bir hiyerarşi ilişkisi vardır. Ayrıtlar ise bu komitelere arasındaki ilişkileri göstermektedir.



Şekil 5.4 : Amerikan hükümet temsilciler meclisinde komite ve alt komiteler [107].

Çizge kümeleme ilk bakışta kolay anlaşılabilir bir kavram gibi görünse de, aslında kümenin tanımı üzerinde literatürde bir uzlaşı olduğu söylenemez. Çizgede

kümelemenin var olduğunu söylemek için, çizgelerin sığ (sparse) olması gerektiği açıktır. Eğer çizgedeki ayrıt sayısı, düğüm sayısından çok fazla ise, bu durumda kümeleme problemi daha çok veri kümeleme (data clustering) problemine yakındır. Veri kümeleme problemlerinde genellikle kümelerdeki eleman sayısı modülleri bulunmaya çalışılan yazılım çizgelerine göre çok daha fazladır.

Yazılımları büyüklüklerine göre 3 gruba ayrılabilir: Düğüm sayısı 100'den az olan küçük yazılımlar, 100-1.000 arasında olanları orta yazılımlar, 1.000 ve yukarıda olan büyük yazılımlar. Düğüm sayısı ancak çok büyük yazılımlarda 10.000 düğümün üzerine çıkmaktadır. Ayrıt sayısı (m) ise düğüm sayısının (n) bir katı olmaktadır (nxortalama düğüm derecesi). Ortalama düğüm derecesi genellikle 3 ile 5 arasındadır.

Bundan sonraki bölümde, farklı çizge kümeleme yaklaşımları temel özelliklerine göre sınıflandırılarak anlatılmıştır.

5.2 Çizge Kümeleme Yaklaşımları

Çizge bölmeleme: Çizge bölmeleme (graph partitioning) problemi çizgedeki düğümleri eşit boyda kümelere, kümeler arasındaki ayrıtların sayısı minimal olacak şekilde ayırmayı ifade eder. Gruplar arasında oluşan ayrıtların sayısı, kesit sayısı (cut size) olarak adlandırılır. Burada küme sayısını önceden bilmeye ihtiyaç vardır. Çizge parçalama paralel hesaplamada, devre parçalamada ve devre kartı yerleştirmede sıklıkla kullanılır. Birçok algoritma çizgeyi iki parçaya ayırmaktadır, ikiden çok parçaya ayırmak genelde yinelemeli şekilde tekrar ikiye ayırarak yapılır.

Elektronik devrelerin kartlara dağıtılması için geliştirilen Kernighan-Lin algoritması [108], bu konudaki ilk algoritmalardan biridir. Diğer bir yöntem Laplas matrisinin özelliklerine dayanan spektral ikiye bölmeleme yöntemidir [109].

Çizge parçalama algoritmaları yazılımdaki kümeleri ya da çizgedeki toplulukları bulmak için uygun yöntem oldukları söylenemez çünkü grup sayısının önceden belli olması ve kümeleri eşit boyda olması bu problem için mümkün değildir. Oysaki algoritmanın bunu bir giriş olarak alması değil çıkış olarak vermesi beklenmektedir. Yinelemeli olarak ikiye ayırma yönteminin de çok verimli olduğu söylenemez çünkü çizgeyi 3'e ayırmak için önce 2'ye ayırıp kalan parçalardan birini de 2'ye ayırmanın güvenilir sonuç üretmeyeceği açıktır.

Bölmelemeli (Partitional) Kümeleme: Diğer bir popüler kümeleme sınıfı bölmelemeli kümelemedir. Burada küme sayısı k önceden belirlenir. Her düğüme metrik uzayında bir nokta atanır ve bu noktalar arasındaki uzaklık ölçüsü tanımlanır. Uzaklık düğümler arasındaki benzersizliğin (dissimilarity)'nin bir ölçüsüdür. Amaç verilen bir maliyet fonksiyonunu maksimum/minimum yapacak şekilde, noktaların uzaklıklarına göre noktaları k kümeye ayırmaktır. En popüler yöntemlerden biri k-means kümelemedir [110]. Burada maliyet fonksiyonu toplam küme içi uzaklıktır.

Hiyerarşik Kümeleme: Bu kümeleme yöntemlerinde, yinelemeli olarak bir önceki aşamada bulunan kümelerden yararlanılarak yeni kümeler çıkarılır. Yukarıdan aşağıya parçalayıcı (divisive) ya da aşağıdan yukarıya birleştirici (agglomerative) olabilir. Aşağıdan yukarıya yöntemlerde ilk başta her düğüm ayrı bir kümedir ve her adımda kümeler birleştirilerek daha büyük kümeleri oluştururlar. Yukarıdan aşağıya yöntemlerde ise ilk başta tüm düğümler aynı kümededir ve her adımda bu kümeler daha küçük kümelere ayrılır.

Parçalayıcı (Divisive) Algoritmalar: Çizgedeki kümeleri bulmanın bir yolu da kümeler arası olan ayrıtları belirleyip bu ayrıtları çizgeden silerek kümeleri bağlantısız hale getirmektir. Parçalayıcı algoritmaların mantığında da bu yatmaktadır. Burada kritik nokta çizge kümeleri birbirine bağlayan ayrıtları belirlemektir. Bu konudaki en iyi bilinen algoritma Girvan ve Newman ayrıt aradalığı algoritmasıdır [87], [14]. Burada ayrıtlar, ayrıtların çizgedeki önemini belirten ve ayrıt merkezcilliği olarak adlandırılan bir ölçüye göre seçilir. Algoritma şu şekilde çalışmaktadır:

1. Çizgedeki tüm ayrıtlar için merkezcillik değeri hesaplanır.
2. En yüksek merkezcillik değerine sahip düğüm çizgeden silinir.
3. İstenen sayıda ayrıt silinene kadar 1. adıma dönülür.

Algoritma sonlanma koşulu olarak modülerlik kriterini kullanmaktadır. Ayrıt merkezcilliği ölçüsü olarak farklı parametreler kullanılabilir. Girvan Newman ilk olarak ayrıtın aradalığına göre (edge betweenness) bir değer belirlemiştir. Bu değer ayrıt üzerinden geçen tüm en kısa yolların sayısıdır (geodesic betweenness). Bilginin genelde en kısa yoldan aktığı düşünülerek bu değerin ayrıtın önemi için bir ölçü olabileceği düşünülmüştür [111]. Kümeler arasındaki ayrıtların da aradalık

değerlerinin yüksek olması beklenir çünkü farklı kümelerdeki düğümler arasındaki birçok en kısa yol bu ayrıtlar üzerinden geçer.

Bilginin yayılımı düşünüldüğünden akışın her zaman en kısa yoldan gitmediği, coğrafi yollar yerine rastgele yollar izleyebileceği de söylenebilir. Bu durumda ayrıtın aradalık değeri çizge üzerinde rastgele yürüyüşte bu ayrıt üzerinden kaç kez geçildiği olur (random-walk betweenness). Rastgele yürüyüşte bir düğümün üzerinden geçerken akış ayrıtlara eşit olasılıklı olarak dağıtılır.

Akım aradalığı (current-flow betweenness) yönteminde ise çizge bir elektrik devresi gibi düşünülür. Ayrıtlar birim dirence sahiptir. Herhangi iki düğüm arasında gerilim uygulandığında her ayrıttan geçecek akım miktarı kirchoff denklemleri ile hesaplanabilir. Bu hesaplama tüm olası düğüm çiftleri için yapılır ve ortalaması alınarak ayrıtın akım aradalığı değeri hesaplanır.

Bu yöntem diğer birçok çalışmaya ilham olan yeni bir bakış açısı getirdiğinden önemlidir [106]. [112]'de ise ayrıt silme yerine, düğüm merkezciliğine göre yinelemeli olarak düğüm ve bu düğümün bağlı olduğu ayrıtları silme denenmiş ve başarılı sonuçlar elde etmiştir.

Modülerlik Tabanlı Algoritmalar: Girman ve Newman'in algoritmalarında sonlanma koşulu olarak kullandıkları modülerlik zamanla birçok algoritmanın temel parçası olmuştur. Modülerliğin yüksek değeri iyi kümelemenin bir göstergesidir, bu nedenle çizge üzerinde maksimum modülerlik değerini veren kümeleme en iyi kümelemedir denebilir. Ancak çizgeyi parçalamanın birçok yolu olduğundan tüm kombinasyonları deneyerek en iyi parçalamaya ulaşmak mümkün değildir. Modülerlik eniyilemesinin NP-complete olduğu ispatlanmıştır [113]. Bununla beraber modülerlik eniyilemesi için kabul edilebilir sonuçlar üreten sezgisel algoritmalar mevcuttur.

Spektral Algoritmalar: Çizge matrislerinin spektral özellikleri de çizge kümelerini bulmakta sıklıkla kullanılmaktadır. Bunun bir örneği Laplas matrisi üzerinde özdeğerleri kullanmaktır. Burada ana mantık düğümler arasındaki yapısal ilişkileri çizge matrisinin özdeğerlerinde karşı düşürülen parçaları bulmaktır [114]. Özdeğerler kullanılarak düğümlere M boyutlu uzayda bir nokta karşı düşürülür. Daha sonra bu noktalar arasındaki Öklid uzaklığına göre düğümler kümelere ayrılır. Bu tür

algoritmalar genelde görüntü işlemede farklı nesneleri ayırt etmekte işe yaramaktadır, ancak yönlü ve ağırlıklı çizgelerde kullanılması zordur [115].

Dinamik Algoritmalar: Dinamik algoritmalar çizge üzerinde süreç işleterek çalışırlar. Bu tür algoritmalara örnek olarak rastgele dolaşı verilebilir. Eğer bir çizge güçlü bir topluluk yapısı (community structure) taşıyorsa, bu çizge üzerinde rastgele yüründüğünde, küme içi ayrıtlar daha yoğun olduğundan, bir küme içinde uzun zaman harcanacaktır. Bu tür algoritmaların avantajı hepsinin ağırlıklı olarak çalıştırılabilir hale getirmenin mümkün olmasıdır.

Ağırlıklara göre ayrıtlarda rastgele dolanma olasılığı verilerek ağırlıklı çizgelerde çalışabilir hale getirilebilirler. [112]'de rastgele dolaşı düğümler arasındaki uzaklığı tanımlamada kullanılmıştır. i ve j düğümü arasındaki uzaklığı rastgele yürüyüşte j düğümünden i'ye ulaşmak için ortalama kat edilmesi gereken ayrıt sayısı olarak vermiştir. Yakın düğümlerin aynı küme içinde yer alması beklenir. Buna göre iki tanım daha yapmıştır: küresel çekici (global attractor) düğüme en yakın düğüm, yerel çekici (local attractor) i'ye en yakın komşu düğümdür. Bir düğüm kendi çekimcisi ile aynı kümeye koyulmalıdır ve kendisinin çekicisi olduğu düğümlerle aynı küme de olmalıdır.

Tezin başlangıç aşamasında her yaklaşımdan önde gelen aşağıdaki algoritmalar yazılım çizgeleri üzerinde denenmiştir.

Zayıf bağlı parça kümeleme algoritması: Zayıf bağlı parçaları bulma diğer algoritmalarında kullandığı oldukça temel bir yöntemdir. Burada çizge kendi içinde bağlı bileşenlere ayrılır. Örneğin ayrıt aradalığı (edge betweenness) yönteminde ayrıtlar silindikten sonra oluşan çizge üzerinde zayıf bileşen kümeleme yapılarak kümeler bulunur. Bazen de algoritmaların doğası gereği, oluşan bağımsız parçaları ayrı ayrı analiz etmek gerektiğinde kullanılır. Algoritma giriş olarak sadece çizgeyi almaktadır.

k-means kümeleme algoritması [110]: d boyutlu uzayda düğümler noktalara karşılık düşürülür. Düğümler uzayda birbirine yakınlıklarına göre kümelere ayrılır. K-means'de zayıf parçalara ayırma gibi diğer algoritmaların sıklıkla kullandığı bir yöntemdir. Algoritma giriş olarak küme sayısını almaktadır. Enerji tabanlı bir yerleştirme algoritmasıyla düğümler 2 boyutlu uzayda yerleştirilerek k-means algoritması uygulanabilir. Bir diğer uygulama şekli de, düğümleri n boyutlu uzayda

tanımlayıp, aralarındaki uzaklığı aralarında ayrıt olup olmamasına göre belirli bir benzerlik katsayısına göre (Örneğin Jaccard katsayısı) belirlemektir. Küme sayısı önceden bilinemeyeceği için deneylerde farklı küme sayısı parametreleriyle oluşan kümelerin ayrı ayrı incelenmesi gerekir.

Ayrıt aradalığı kümeleme algoritması (Edge Betweeness) [87]: Parçalamaya dayalı (Divisive) bir algoritmadır. Algoritma giriş olarak silinecek ayrıt sayısını almaktadır. Burada problem silinecek ayrıt sayısının önceden bilinmesidir. Deneylerde farklı parametrelerle oluşan kümeler incelenmiştir.

K-way hiyerarşik kümeleme algoritması [116]: Hiyerarşik bir kümeleme algoritmasıdır. Çizge yönlü ve ağırlıklı olabilir. Algoritma doküman verilerinin kümelenmesinde kullanılmıştır. Bu kümeleme için Cluto [117] kütüphanesinden yararlanmıştır.

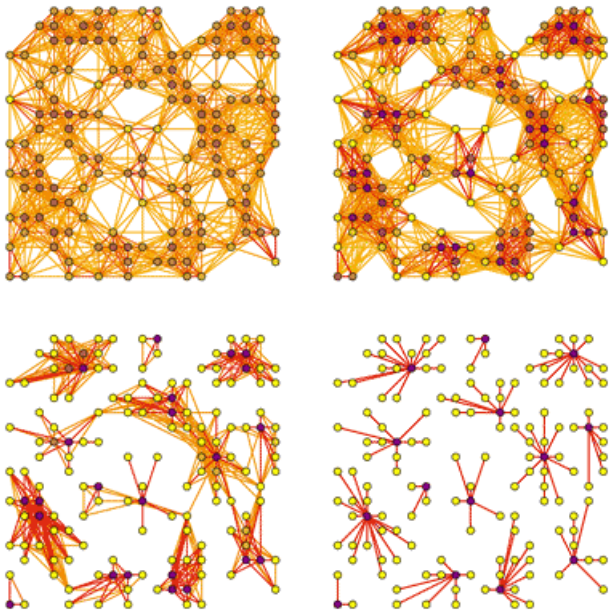
Bicomponent kümeleme algoritması: Çizge ikili bağlı bileşenlere (biconnected components) ayrılarak kümelenir. Algoritmada çizge yönsüz ve ağırlıksız olarak değerlendirilmektedir.

Markov kümele algoritması (MCL) [118]: Akış benzetimine dayalı dinamik bir kümeleme algoritmasıdır. Bu algoritmada dinamik olarak çizge üzerinde akış dağılımı benzetimi yapılır. Algoritmanın görsel olarak çalışması Şekil 5.5'de gösterilmiştir. Algoritma ağırlıklı olarak çalışsa da, ağırlık değeri bir düğümden çıkan ayrıtlar arasında normalize edilerek kullanıldığından ağırlık küresel değil yerel bir önem ifade etmektedir. Yönlü olması durumunda, yazılım çizgelerinde bazı düğümlere sadece giren ya da sadece çıkan yönde ayrıtlar olduğundan, benzetim süreci bu düğümlerde kilitlenebilir ve yanıltıcı sonuçlar üretebilir. Aynı zamanda algoritmanın, kalan enerji ve matris çarpımında kullanılan çarpan olarak kullandığı parametreler çizgeden çizgeye değişmekte ve nasıl bir değer alması gerektiği net bir şekilde bilinmemektedir.

Voltaj kümeleme algoritması [119]: Bu algoritmada, çizge bir elektrik devresi olarak düşünülür ve Kirchoff denklemlerine göre düğümlerin voltaj değeri hesaplanır. Daha sonra bu voltaj değerleri üzerinden k-means kümeleme uygulanarak düğümler kümelere ayrılır.

Ağırlıklı çizge kesitlerine göre kümeleme [120],[121]: Algoritma ağırlıklı yönsüz çizgelerde çalışmaktadır. Algoritma giriş olarak küme sayısını almaktadır. Algoritma

geleneksel spektral algoritmaların daha hızlı bir alternatifi olarak geliştirilmiştir. İmge parçalarını ayırt etmede (Image segmentation), sosyal ağların analizinde, gen ağlarının analizinde başarılı sonuçlar verdiği gösterilmiştir. Bu algoritma için Graclus [122] kütüphanesinden yararlanılmıştır.



Şekil 5.5 : Markov kümeleme algoritmasının çalışması [118].

5.3 Yazılım Çizgelerinin İncelenmesi

Tez çalışmasında denenen çok sayıda temel çizge kümeleme algoritmasının yazılım çizgeleri üzerinde şaşırtıcı bir şekilde çok başarısız sonuçlar verdiği gözlemlenmiştir. Bunun üzerine sonuçların nedenlerini incelemek ve uygun algoritmaları belirleme amacıyla yazılım çizgelerinin özellikleri incelenmiştir. Takip eden alt bölümlerde yazılım çizgelerinin özellikleri ve temel algoritmaların neden başarısız sonuçlar ürettiği anlatılmaktadır.

5.3.1 Yazılımlarda yüksek kümeleme katsayısı ve dünya küçük davranışı

Kümeleme sonuçlarının çizgenin nasıl temsil edildiğine ve çizgenin niteliklerine göre çok farklılık gösterebildiği, uygun kümeleme algoritmasının uygulama problemine göre seçilmesi gerektiği ve ilk olarak yazılım çizgelerinin özelliklerinin belirlenmesinin seçilecek yöntemin belirlenmesinde ciddi rol oynayabileceğinden önceki bölümlerde belirtilmişti. Bu bölümde yazılım çizgelerinin özellikleriyle ilgili

literatürdeki araştırmalar ve geliştirilen yazılım analiz aracına eklenen özellikler yer almaktadır.

"Hierarchical Small-Worlds in Software Architecture" [123] çalışmasında yazarlar çok sayıda nesneye dayalı C++ ve Java projesi üzerinde incelemeler yapmış ve yazılım çizgeleri üzerinde ortak kalıplara ve istatistiksel bilgiler bulmuştur. İncelenen tüm nesneye dayalı sistemler sosyal ağlarda sıklıkla rastlanan "Dünya Küçük Davranışı" ("Small-World Behavior") göstermiştir. Buna göre sınıf çizgelerinde, sistemdeki herhangi iki sınıf arasındaki ortalama yol uzunluğu, düşük bağımlı ve yüksek uyumlu sistemlerde bile, oldukça küçük çıkmıştır. Bununla birlikte tüm çizgelerin oldukça heterojen ağ karakteristiğinde olduğu, düğümlerin derece dağılımının güç kanunu dağılımına (power law distribution) uyduğu ve dağılımın üstel parametresinin (alfa) yazılımlarda benzerlik gösterdiği belirlenmiştir. Buna göre her yazılımda ortalama düğüm derecesinin çok üstünde iç ve dış ayrıt içeren az sayıda düğüm vardır. Son olarak yazarlar yazılım çizgelerinin bu özelliklerinin neden kaynaklandığını açıklamaya çalışmışlardır. Çizelge 5.1'de bu çalışmada farklı yazılımlarda yapılan bazı çizge ölçümleri verilmiştir. Burada çizelge değişkenler şu şekilde tanımlanmıştır: N: Düğüm Sayısı, L: Ayrıt sayısı,

$L \sim N^{1.17}$, ayrıt sayısı neredeyse doğrusal olarak düğüm sayısıyla ölçeklenmektedir.

d: ortalama yol uzunluğu,

d_{rand}: aynı N ve L ye sahip bir rastgele çizgedeki (random graph), ortalama yol uzunluğu

C: Kümeleme katsayısı,

C_{rand}: aynı N ve L ye sahip bir rastgele çizgedeki, kümeleme katsayısı

Bir düğümün kümeleme katsayısı şu şekilde tanımlanmaktadır:

$$C_i = \frac{t_i}{k_i(k_i-1)} \quad \textbf{(5.1)}$$

Burada k_i, i düğümün derecesi, t_i i düğümünün katıldığı üçgenlerin sayısıdır. Çizgenin kümeleme katsayı ise şu şekilde hesaplanmaktadır:

$$C = \langle \frac{2}{k_i(k_i-1)} \sum_{j=1}^{N} A_{i,j} \left[\sum_{k=1}^{N} A_{j,k} \right] \rangle \quad \textbf{(5.2)}$$

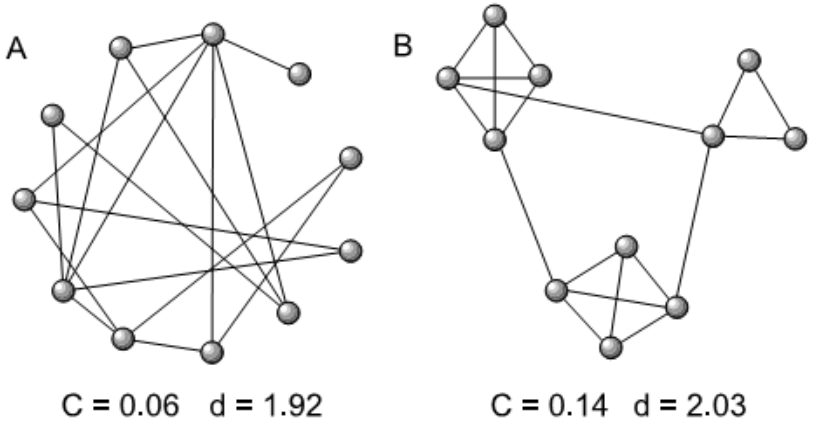
Çizelge 5.1 : Farklı nesneye dayalı yazılımlarda yapılan çizge ölçümleri [123].

Veri Kümesi	N	L	d	drand	C	C_{rand}
Mudsi	168	241	2,88	4,95	0,244	0,017
JDK-B	1364	1947	5,97	6,8	0,225	0,002
JDK-A	1376	2162	5,4	6,28	0,159	0,002
Prorally	1993	4987	4,85	4,71	0,211	0,003
Striker	2356	6748	5,9	4,46	0,282	0,002
gchempaint	27	41	2,85	3,26	0,204	0,102
4yp	54	90	3,28	3,44	0,069	0,059
Prospectus	99	168	3,8	3,77	0,14	0,034
eMule	129	218	3,87	4,16	0,237	0,025
Aime	143	319	2,66	3,34	0,413	0,031
Openvrml	159	335	3,53	3,53	0,08	0,026
gpdf	162	300	4,02	3,93	0,303	0,022
Dm	162	254	4,32	4,45	0,304	0,019
Bochs	164	339	3,15	3,6	0,335	0,025
Quanta	166	239	4,31	5,03	0,198	0,017
Fresco	189	277	4,73	4,89	0,228	0,015
Freetype	224	363	4,29	4,71	0,193	0,014
Yahoopops	373	711	5,57	4,47	0,336	0,010
Blender	495	834	6,54	5,14	0,155	0,007
GTK	748	1147	5,87	5,91	0,081	0,004
OIV	1214	3903	3,99	3,82	0,122	0,005
wxWindows	1309	3144	4,03	4,62	0,235	0,004
CS	1488	3526	3,92	4,74	0,135	0,003

Şekil 5.6'da aynı N ve L değerlerine sahip iki farklı çizge verilmiştir. A çizgesi rastgele bir çizgedir, B çizgesi ise modüler bir çizgedir, ortalama uzaklık d, birbirine çok yakın olmasına rağmen B çizgesinin kümeleme katsayısı A'nın 2 katından fazladır. Şekil 5.6'da görüldüğü gibi nesneye dayalı yazılımların türü ve büyüklükleri çok farklı olmasına rağmen hepsinde kümeleme katsayısı rastgele çizgeye göre oldukça büyüktür ($C >> C_{rand}$).

Yazılım çizgelerinin bu özellikleri bir önceki aşamada denenmiş olan MCL [118] ve ayrıt aralığı (edge betweenness) [87] algoritmalarının neden kötü sonuç verdiğini açıklamaktadır. MCL algoritması rastgele dolaşıda, aynı küme içinde kalma ihtimalinin yüksek olduğuna dayanıyordu, oysa yazılım çizgelerinde ortalama yol uzunluğu kısa olduğundan, az sayıda rastgele düğüm üzerinden geçerek çizgede başka kümelere geçme ihtimali oldukça yüksektir. Diğer yandan, ayrıt aradalığı algoritması da kümeleri birbirine bağlayan ayrıtların üzerinden çok sayıda en kısa yol geçeceği varsayımına dayanıyordu, oysa çizgelerin dünya küçük özelliğinde olması

bu varsayımı da geçersiz kılmaktadır. Diğer bir özellik ise yazılım modül boylarının değişken olmasıdır, bu nedenle çizgeyi eşit küme boylarına bölen parçalama tabanlı algoritmalar da yazılım çizgeleri için uygun değildir.



Şekil 5.6 : Aynı N ve L değerlerine sahip iki çizgenin kümeleme katsayıları [123].

5.3.2 Yazılım çizgelerinde kümeleme sonucunu olumsuz etkileyen özellikler

Bir önceki bölümde aktarıldığı gibi yazılımda bazı düğümler çok sayıda ayrıta sahiptir. Bu düğümler genellikle, çok sayıda farklı modülden girişi olan, dışarı çıkışı olmayan kütüphane (libraries/utilities) modülleridir. Bu modüller az sayıda olmalarına rağmen, çok sayıda farklı kümeden ayrıt içerdikleri için kümeleme sonuçlarını olumsuz yönde etkileyebilmektedir [124].

Yazılım çizgelerinde sıklıkla görülen bir diğer kalıp (pattern) ise, heryere yayılmış (omni-present) düğümlerdir. Bu düğümlerin çok fazla sayıda modülden içeri yönde ya da dışarı yönde ayrıt içerirler. İçeri yönde ayrıt sayısı, ortalama düğüm derecesinin k katından fazla ise, bu düğümler yayılmış tedarikçi (omni-present supplier) olarak; dışarı yönde ayrıt sayısı ortalama düğüm derecesinin k katından fazla ise, yayılmış istemci (omni-present client) olarak; hem içeri hem dışarı yöndeki ayrıt sayısı ortalama düğüm derecesinin k katından fazla ise, merkezi yayılmışlar (omni-present central) olarak adlandırılırlar. Bu düğümler her modüle dağıldığından, çizge kümeleme sonuçlarını olumsuz yönde etkilemektedir. Deneylerde yazılımda Önyüz (Facade Pattern), Denetçi (Controller pattern) ya da test işlevi gören sınıfların genelde bu özelliği taşıdığı gözlenmiştir.

Bu durum son yıllarda yapılan bir araştırmada da ortaya koyulmuş ve yazılım mimarisini anlama yöntemlerinde bu tarz modüllerin problem yarattığı belirtilmiştir

[124]. Bu çalışmada yazılım mimarisi anlama tekniklerinin (Software Comprehension) büyük ölçüde yazılımın kalitesine bağlı olduğu, sistem parçaları arasında bağımlılık artıkça yazılımı anlamanın zorlaştığı, ancak bütün bağımlılıkların aynı seviyede öneme sahip olmadığı belirtilmiştir. Özellikle çok sayıda modül tarafından çağrılmaya meyilli kütüphane işlevi gören parçaların, sistem yapısına çok sayıda bağımlılık eklerken, anlaşılırlık açısından sisteme bir şey kazandırmadığı gözlenmiştir. Bu nedenle kümeleme algoritmalarının başarımı ilk olarak bu modüllerin belirlenebilme başarısına bağlıdır. Bu sonuç literatürdeki birçok araştırmada da geçmekte ve bu modüllerin belirlenmesinin önemi sıkça ifade edilmektedir [20], [21], [125].

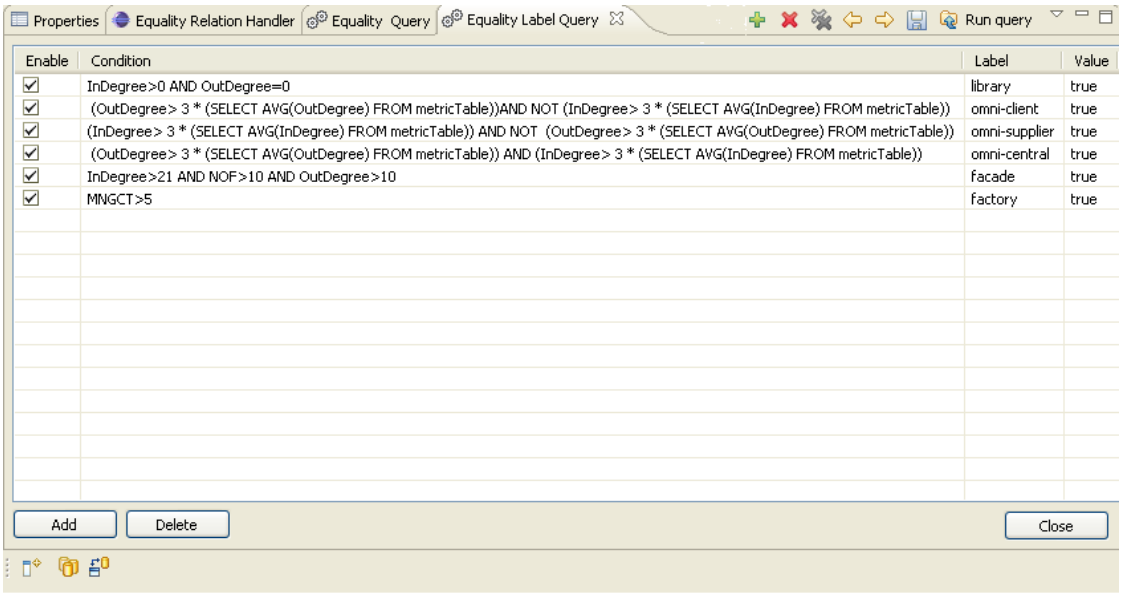
Bu nedenle tez kapsamında bu tür modülleri bulan ve kümelemeden önce filtreleyen yöntemler yazılım analizine dâhil edilmiş ve kümeleme filtrelenen çizgeler üzerinde yapılmıştır. Öte yandan kümele algoritmalarının bir kısmı ise bu özelliği kendi içinde yaparak çalışmaktadır [126].

Bu modüllerin bulunması giriş yelpazesi (fan-in) ve çıkış yelpazesi (fan-out) analizlerine dayanmaktadır. Farklı yerlerden çağrı alan bir modülün (çizge modelinde çok fazla içeri ayrıt bulunan) kütüphane modülü olma ihtimali oldukça yüksektir. Tersine dışarı yönde çok fazla ayrıtı bulunan modüllerinde kütüphane modülü olma ihtimali ise çok düşüktür. Bunun için bir eşik değeri kabul edilmekte, giriş ve çıkış düğüm derece metrikleri değerlendirilmektedir.

Diğer basit ama etkili yöntemde, isimlendirmedeki uyumluluklardır. Örneğin birçok kütüphane paketinin isminde “util.”, “utility”, "common", "lib" gibi kelimeler geçmektedir. Her ne kadar bu kullanılan dile oldukça bağımlı olsa da benzer yaklaşımla sistem mimarları bu modülleri rahatlıkla söyleyebilirler.

Bir diğer önemli noktada derece dağılımının güç dağılımına uygun olması nedeniyle, derecesi yüksek olan düğümlerinin sayısının oldukça az olması, buna neden olası modüller belirlendikten sonra, yazılım mimarları az sayıda modülü hızlıca analiz edip, kütüphane ya da yayılmış (omni-present) modül mü yoksa sistemin çekirdek bir işlevini yerine getiren bir modül mü, rahatlıkla belirleyebilir.

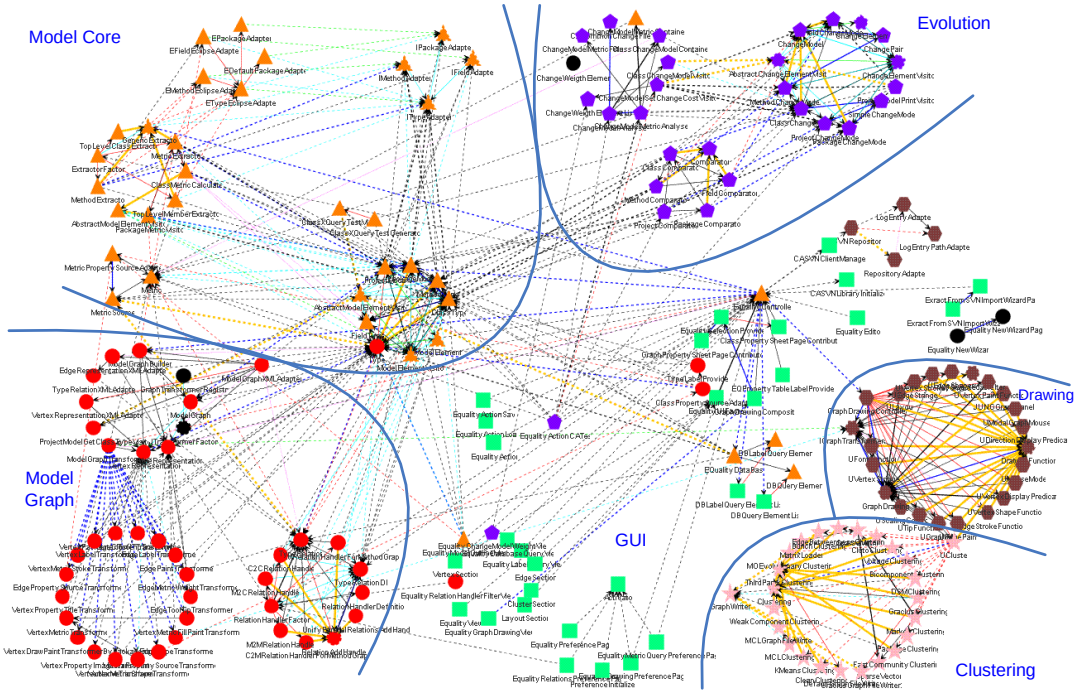
Bu amaçla analiz aracına belirli koşulları sorgulayarak düğümlere etiket ekleyebilme yeteneği eklenmiştir. Bu etiketlerden çalışma zamanı değiştirilebilir çizgede istenilen özelliklerdeki düğümler bulunabilir. Örnek bir etiket kümesi Şekil 5.7‘de verilmiştir.



Şekil 5.7 : Etiket sorgulama arayüzü.

5.4 Yazılım Kümelemelerinin Gösterimi

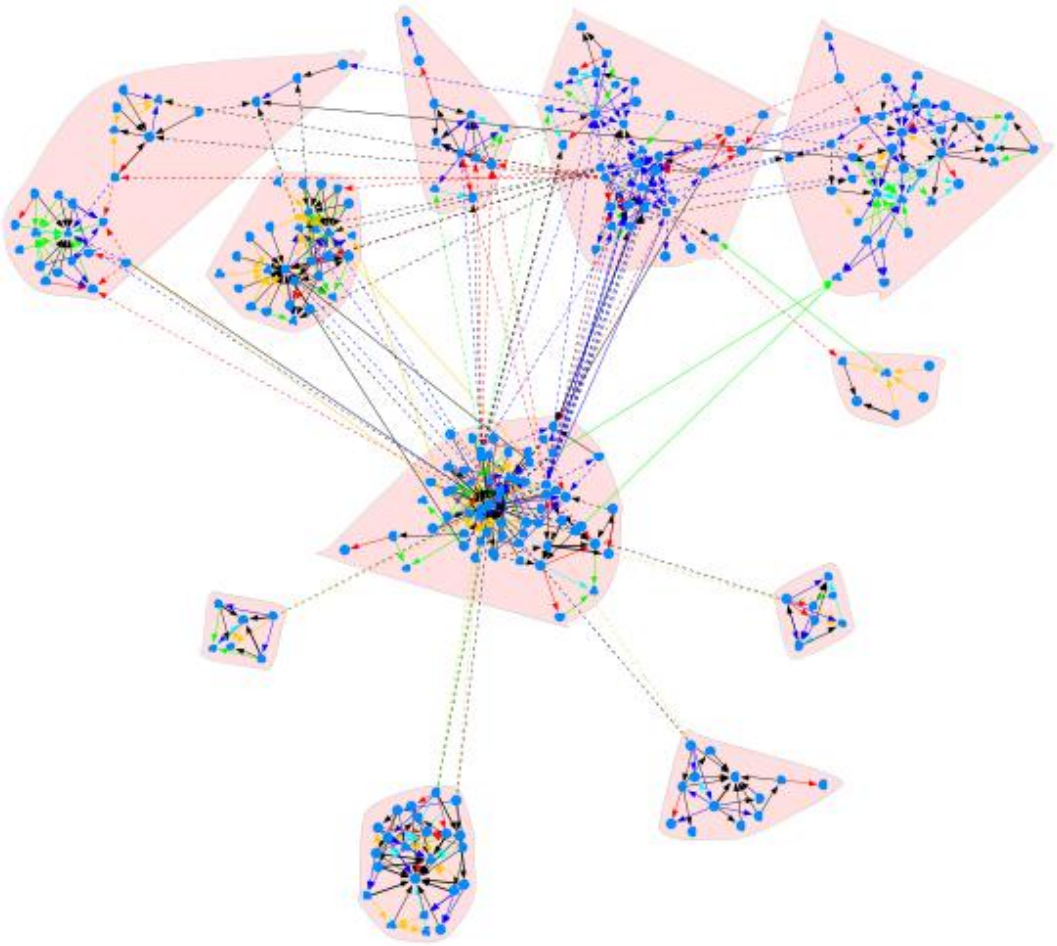
Deneyler sırasında karşılaşılan bir başka problem ise kümelemenin temsilidir. Genellikle kümeleme algoritmaları liste şeklinde kümeleri sıralamakta, ancak 20-30 sınıflı yazılımlarda bile bu kümeleme sonuçlarını anlamak ve analiz etmek oldukça zordur. Buna çözüm olarak, ilk etapta çizgedeki düğümleri paketlere göre yerleştirmeyi ve kümeleme algoritmasının ürettiği sonuçlara göre her kümeye farklı şekil, farklı renk ve farklı büyüklük atanarak kümeleme sonucunun daha iyi algılanabilmesi sağlanmıştır. Deneylerde fark edilen bir diğer özellik bazı algoritmaların çok sayıda sadece bir elemanlı kümeler üretmesidir. Bunlar algoritmaların zayıf yönü olarak değerlendirilmiştir ve bu kümeler küçük siyah renkte gösterilerek, küme gösterim uzayının verimli şekilde kullanılması sağlanmıştır. Farklı kümeleme algoritmaları üst üste çalıştırıldığında çizgedeki düğümlerin konumları korunmakta böylece algoritmaların ürettiği sonuçlar daha iyi karşılaştırılabilmektedir. Çok sayıda algoritma olduğu ve her birinin farklı parametrelere göre çalışma zamanı deneceği düşünülerek algoritmaların parametrelerinin girilebileceği bir arayüz de eklenmiştir. Şekil 5.8'de her bir paketin bir kümeyi oluşturduğu durumda çizge görüntüsü verilmiştir. Burada aynı kümedeki sınıflar ve aralarındaki ilişkileri açıkça gözükmektedir. Son olarak çizgeye hiyerarşik yerleşim özelliği kazandırılmış ve çizgenin kümelere göre hiyerarşik olarak çizilmesi sağlanmıştır (Şekil 5.8).



Şekil 5.8 : Kümelemenin paket yapısıyla gösterimi.

Analiz sonuçlarının daha hızlı ve doğru yorumlanabilmesi ve kümelemenin düzenlenebilmesinde en büyük zorluklardan biri kullanıcıya etkin bir görsel arayüz sunabilmektedir. Çalışma kapsamında bu işlemler kritik rol oynamaktadır. Bu nedenle çizge görselleştirme ile analiz aracının özellikleri arttırılmıştır.

Bunun yanında model ve görsel çizgeler ayrılmış, bir yazılım üzerinde farklı çizgelerin analiz edilmesi ve analizlerde kullanılacak referans kümelemenin ayrı ayrı tutulabilmesi özelliği sağlanmıştır. Geliştirilen analiz aracının örnek bir görünümü Şekil 5.9'da gösterilmiştir. Bu adımda düğümler ve kümeler aralarındaki ayrıtlara göre otomatik yerleştirilmektedir. Görüldüğü gibi modüler bir yazılımda düğüm ve kümeleri otomatik konumlandırma ile modül yapısını dikkate alarak çizme oldukça başarılıdır. Bu şekilde, dışarıdaki pembe poligonlar bir kümeleme yapısını belirtirken (Örneğin referans modül yapısını), içerideki düğümlerin şekil ve renkleri bir başka kümeleme yönteminin sonucunu (Örneğin denenen Kümeleme algoritmasının sonucunu) ya da sınıfların konumlarını anlamak için sınıfların kalite niteliklilerini gösterecek şekilde ayarlanabilir. Böylece üretilen sonuçlar daha iyi analiz edilebilir. Ayrıca kümelerdeki düğümler toplu olarak seçilip düzenlenebilir, kümeye ile içeri ya da dışarı yönde bağlantısı olan diğer kümeler etkileşimli olarak gösterilebilmektedir.



Şekil 5.9 : Kümelemenin otomatik gösterimi.

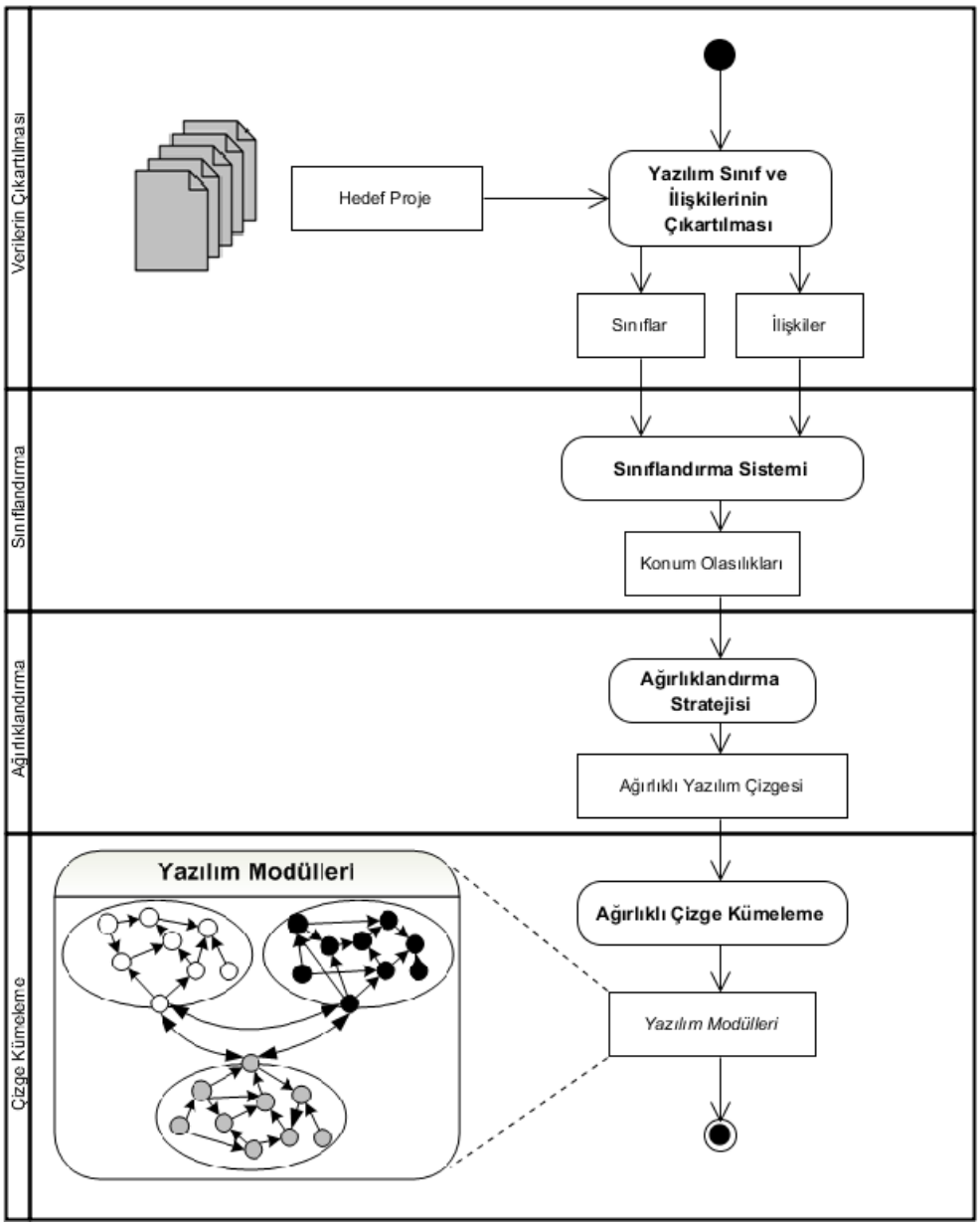
6. ÖNERİLEN NESNEYE DAYALI SERVİS BULMA YÖNTEMİ

Yazılım sistemlerinde modülleri otomatik bulmak için geliştirilen yöntemlerin büyük bölümü yazılım sistemini çizge şeklinde temsil etmektedir. Bu çizgede düğüm ve ayrıtlar sırasıyla sınıfları ve aralarındaki ilişkileri temsil etmektedir. Uygun tasarlanmış modüler yazılım sistemleri, düşük bağımlılık ve yüksek uyumluluk karakteristiklerini taşımalıdır. Bu nedenle yazılım bağımlılık çizgesinin, az yoğunlukta kümeler arası (dış) ayrıtlardan, çok yoğunlukta küme içi (iç) ayrıtlardan oluşması beklenir. Bu varsayım ile birçok çalışma, iç ve dış ayrıt oranına göre bir modülerlik objektif fonksiyon tanımlamakta ve bu fonksiyonu en yüksek değere taşıyacak optimal bir bölmeleme bulmaya çalışmaktadır.

Tez çalışmasında kapsamında önerilen yöntemin ana amacı, nesneye dayalı sistemler için daha doğru (uzman ayrıştırmasına daha yakın) sonuçlar elde edebilmektir. Bu nedenle, önerilen modül bulma yaklaşımı sadece ayrıtların yoğunluğunu değil, aynı zamanda sınıfların karakteristiklerini ve ilişki tiplerini de göz önünde bulundurmaktadır. Bu karakteristikler nesneye dayalı tasarımın doğasını yansıtırlar ve önerilen yöntemde yazılım çizgesindeki ayrıtların ağırlıklandırılmasında kullanılırlar. Ayrıtları ağırlıklandırmanın amacı, kümeleme algoritmasına kılavuzluk ederek daha doğru yazılım modül sonuçlarının üretilmesini sağlamaktır.

Önerilen modül bulma metodunun iş akışı Şekil 6.1'de kısaca gösterilmiştir. Birinci adımda, yazılım sisteminin çizgesi, sınıfların ve ilişkilerin hedef projeden çıkartılmasıyla elde edilir. Yazılım sisteminin çizge olarak temsil edilmesi Bölüm 6.1'de açıklanmıştır. İkinci adımda, ayrıtlar modül içi ve modül dışı olma olasılıklarına göre ağırlıklandırılırlar. Bu olasılıklar çizgedeki ayrıt ve düğümleri sınıflandıran, bir sınıflandırma sisteminin sonuçlarına göre belirlenir. Bu sınıflandırıcılar referans projeden edilen bilgilere göre bir kez eğitilerek oluşturulur. Daha sonra aynı sınıflandırıcılar farklı projelerin ağırlıklandırılmasında kullanıma hazır hale gelir. Sınıflandırma sistemi Bölüm 6.2'de ve ağırlıklandırma stratejisi Bölüm 6.3'de verilmiştir. Üçüncü adımda, uygun kümele algoritması Ağırlıklı Yönlü

Çizge (AYG) üzerinde uygulanır ve Bölüm 6.4'de açıklandığı gibi verilen yazılımın modül yapısı oluşturulur.



Şekil 6.1 : Nesneye dayalı modül bulma süreci.

6.1 Çizge Temsili

Bir nesneye dayalı yazılım sistemi ***G<V, E>*** çizgesi olarak temsil edilebilir. Burada ***V*** yazılım sınıflarının ve arayüzlere karşılık düşen düğümlerin kümesi $\{v_i\}$ ve ***E*** bu sınıflar arasındaki farkı tipteki (Örneğin kalıtım, metot çağrısı, gerçekleme, parametre görünürlüğü) ilişkilere karşılık düşen ayrıtların kümesidir $\{e_i\}$.

v_s ve v_d düğümleri, yazılım sınıfları *s* ve *d*'yi temsil ettiği varsayımıyla, yazılım sisteminde *s* sınıfından *d* sınıfına bir ilişki varsa, *G* çizgesinde v_s'den v_d'ye e_r ayrıtı vardır; { $e_r : (v_s; v_d) \in V \times V$ | s'den d'ye bir ilişki vardır}.

Eğer yazılım varlıkları arasında birden çok ilişki varsa, tüm paralel ayrıtlar basit bir çizge elde edebilmek amacıyla tek bir ayrıtta birleştirilmektedir. Bilgi kaybı olmaması için böyle birleşik ayrıtlar, tüm ilişkilerin etiketlerini içermektedir. Gerekli tüm bilgiler, örneğin sınıflar ve ilişkiler, Java kaynak kodundan soyut sentaks ağaçları işlenerek çıkarılmaktadır. Bu amaçla daha önceki bölümlerde anlatılan çalışma kapsamında geliştirilen yazılım analiz aracı E-Quality [51] kullanılmaktadır.

Yazılım bağımlılık çizgesinde bir M_i modülü, mantıksal olarak ilgili sınıfları ifade eden düğümler ve aralarındaki ilişkileri ifade eden ayrıtlar grubudur.

Bir modülün tanımı aşağıdaki gibidir:

$\boldsymbol{M_i}$*:* modül *i, i =1,2, ..., k*. *k* yazılım sistemindeki tüm modüllerin sayısı

$\boldsymbol{M_i}$ *G*'nin bir alt çizgesidir; $\boldsymbol{M_i} \subseteq G$. Öyle ki, $\boldsymbol{M_i} = <V_i, E(V_i)>$, V_i, M_i modülündeki düğüm kümesi, $V_i \subseteq V$ $E(V_i)$ ise V_i içindeki düğümlerin arasındaki ayrıtların kümesidir ve şu şekilde tanımlanmıştır: $E(V_i) \subset E$ and $E(V_i) = \{ e_r : (v_s; v_d) (v_s \in Vi \wedge v_d \in Vi)$.

Modüllerin aşağıdaki özellikleri taşıdığına dikkat edilmelidir.

1. Modüller boş olmayan kümelerdir; $V_i \neq \emptyset$, $E(V_i) \neq \emptyset$, $i=1,2,...,k$.
2. Modüller ayrık kümelerdir, iki farklı modül ortak bir düğüm ya da ayrıt içermez. $M_i \cap M_j = \emptyset$, $\forall((1 \leq i, j \leq k \wedge (i \neq j))$. Bu koşul $V_i \cap V_j = \emptyset \wedge E(V_i) \cap E(V_j) = \emptyset \forall((1 \leq i, j \leq k \wedge (i \neq j)$, koşullarını da gerektirir.

Son olarak sistemdeki tüm modüllerin kümesi *M* şu şekilde formüle edilebilir:

$$M = \bigcup_{i=1}^{k} Mi = \bigcup_{i=1}^{k} < V_i, E(V_i) > \qquad \textbf{(6.1)}$$

Modüllerine ayrılmış örnek bir yazılım bağımlılık çizgesi Şekil 6.2'de gösterilmiştir. Burada:

$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13} \}$

$E = \{ e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}\}$

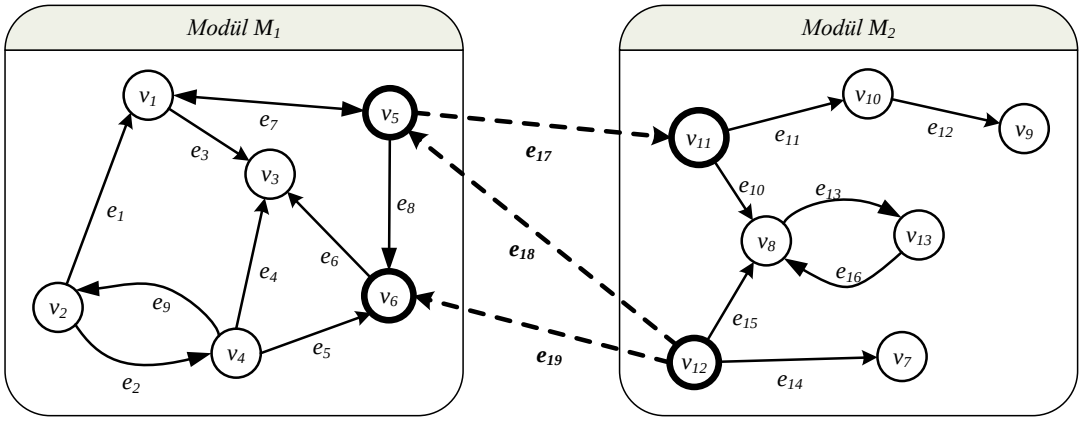
$M = \{M_1, M_2\}$,

$M_1 = <\{v_1,v_2,v_3,v_4, v_5,v_6\}, \{e_1,e_2,e_3,e_4,e_5,e_6,e_7,e_8,e_9\}>$

$M_2 = <\{v_7,v_8,v_9, v_{10},v_{11}, v_{12}\}, \{e_{10},e_{11},e_{12},e_{13},e_{14},e_{15},e_{16}\}>$

Yukarıda verilen tanım ve özellikler doğrultusunda, $G<V, E>$ k örtüşmeyen düğüm kümesine ayırmak şu şekilde formüle edilebilir:

$$V = \bigcup_{i=1}^{k} V_i \;, \forall((1 \le i, j \le k) \wedge (i \ne j)), V_i \cap V_j \ne \varnothing \qquad \textbf{(6.2)}$$



Şekil 6.2 : Modüllerine ayrılmış örnek bir yazılım bağımlılık çizgesi.

G çizgesindeki n adet düğümü k boş olmayan düğüm kümesine olası bölme sayısı 2. tipten Stirling numaraları ile verilmektedir ve farklı bölmeleme sayısı n ile üstel bir şekilde artmaktadır [129].

6.2 Sınıflandırma Sistemi

Yazılım geliştiriciler daha modüler ve yeniden kullanılabilir yazılım sistemleri üretmek için tasarım kalıpları ve tasarım prensiplerine başvururlar. Bunun sonucu olarak, sistemin belirli servisini yerine getiren, ilgili sınıf grubunun oluşturduğu modüller meydana gelir. Bu tarz ortak tasarım kararlarının, yazılımın bağımlılık çizgesinin modüler yapısını da etkileyeceği açıktır. Önerilen LBME yaklaşımı, nesneye dayalı yazılımlar için bu kalıpların etkisini bağımlılık çizgesini ağırlıklandırmada göz önüne alır ve çizge kümeleme algoritmasına kılavuzluk ederek, daha doğru sonuçlar üretilmesini sağlar. Hedef projenin yazılım çizgesindeki ayrıtlarına, uygun ağırlıklar atayabilmek için, bu yöntemde ayrıtların modüle göre konumları modül içi (iç) ve modüller arası (dış) olarak tahmin edilmeye çalışılır. Dış

ayrıtların kaynak ve varış düğümleri farklı modüllerde yer almaktadır. Eğer bir şekilde, tüm dış ayrıtların kümesi (*EX*) belirlenebilseydi, bu ayrıtların *G* çizgesinden çıkartılması ile elde edilen *G' = <V, E-EX>* çizgesindeki bağlı parçalar (connected components) *G* çizgesinin gerçek modülleri olurdu. Ancak tüm ayrıtları temsil ettikleri ilişkinin özelliklerine göre doğru şekilde kategorize etmek mümkün değildir. Örneğin, metot çağrısı ("method call") ve kullanma ("uses") ilişkilerine hem aynı modüldeki sınıflar arasında hem de farklı modüllerdeki sınıflar arasında rastlanabilir. Bu nedenle yöntemin doğruluğunu arttırma adına, ayrıtların konumlarını tahmin edebilmek için, kaynak ve varış düğümlerinin modül içindeki konumlarına ilişkin iç ve kenar olma kategorileri de göz önüne alınmaktadır. Bu amaçla, iki sınıflandırıcı (düğüm ve ayrıtlar) için karar ağacı formunda oluşturulmaktadır. Bu sınıflandırıcılar, referans bir yazılım sisteminden elde edilen veri kümesine göre eğitilmektedir. Veri kümesi, düğümler tarafından temsil edilen sınıfların ve ayrıtlar tarafından temsil edilen aralarındaki ilişkilerin niteliklerini içermektedir. Bu niteliklerin seçimi Bölüm 6.2.1 'de anlatılmaktadır. Yazılım uzmanlarının yazılımı modüler, servis-odaklı yapıya göre değerlendirmesiyle ayrıt kategorileri (iç/dış) ve düğüm kategorileri (iç/kenar) işaretlenmektedir.

Öğrenme aşamasından sonra, ayrıt sınıflandırıcılar herhangi bir hedef projedeki bir (e_i) ayrıtı için, $p_{internal}$ ve $p_{external}$ olasılıklarını üretebilir. Burada $p_{internal}$ iç ayrıt olma olasılığı, $p_{external}$ dış ayrıt olma olasılığıdır ve $p_{internal} + p_{external} = 1$ koşulu sağlanmaktadır. Benzer şekilde düğüm sınıflandırıcısı bir (v_i) düğümü için için, p_{inner} ve p_{border} olasılıklarını üretebilir. Burada p_{inner} iç düğüm olma olasılığı, p_{border} kenar düğüm olma olasılığıdır ve $p_{inner} + p_{border} = 1$ koşulu sağlanmaktadır. Bu koşullara göre ağırlıklandırma algoritması uygun ağırlıkları çizgenin ayrıtlarına atamaktadır. Sınıflandırma sisteminin ayrıntıları takip eden paragraflarda açıklanmıştır.

Düğüm ve ayrıt kategorileri için tanım ve gösterimler aşağıda verilmiştir:

EI: Tüm iç (modül içi) ayrıtların kümesi,$EI \subseteq E,\ EI = \bigcup_{i=1}^{k} E(V_i)$, Burada $E(V_i)$, M_i modülündeki tüm iç ayrıtların kümesi. İç ayrıtların varış ve kaynak düğümleri aynı modüle aittir.

EX: Tüm dış (modüller arası) ayrıtların kümesi *EX = E - EI*. Dış ayrıtların varış ve kaynak düğümleri farklı modüllerde yer alır.

VI: Tüm kenar (border) düğümlerin kümesi $VI \subseteq V$. Kenar düğüm en az bir dış ayrıta bağlıdır; $VI = \{v_i \mid, \exists e(v_s; v_d) \in EX, v_s = v_i \vee v_d = v_i\}$.

VX: Tüm iç (inner) düğümlerin kümesi, bu düğümler başka modüllerdeki düğümlere bağlı değildirler $VX = V - VX$. Örnek olarak, Şekil 6.2'de gösterilen çizge için *EI, EX, VI* ve *VX* kümeleri şu şekildedir:

$EI = \{ e_1,e_2,e_3,e_4,e_5,e_6,e_7,e_8,e_9,e_{10},e_{11},e_{12},e_{13},e_{14},e_{15},e_{16}\}$

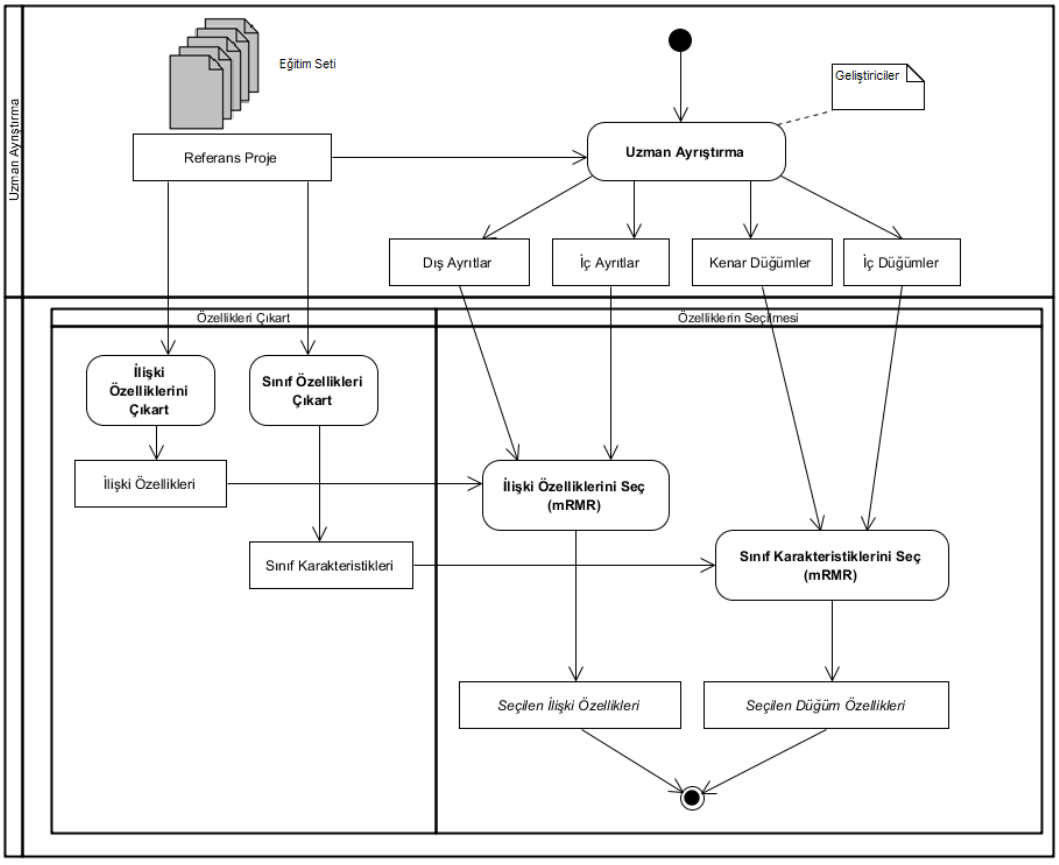
$EX = \{e_{17}, e_{18}, e_{19}\}$

$VI = \{v_1, v_2, v_3,v_4, v_7, v_8, v_9, v_{10}, v_{13}\}$

$VX = \{v_5, v_6, v_{11}, v_{12}\}$

6.2.1 Sınıflandırmalar için nitelik seçimi ve eğitim veri setini hazırlanması

Sınıflandırmada nesneye dayalı niteliklerin seçimi, sınıflandırma hatalarının azaltılması ve sonuçların veri kümesine daha az bağımlı olması için önemlidir. Fazlalık ve ilişkisiz nitelikleri süzmek aynı zamanda öğrenme ve sınıflandırma süresini de azaltılır. Nitelik seçme süreci Şekil 6.3'de gösterilmiştir.



Şekil 6.3 : Sınıflandırma için niteliklerin seçimi.

Önerilen sistemin nihai amacı, sistemi yeniden kullanılabilir servis odaklı modüllere ayırmak olduğu için, sistemin modüler yapısını yansıtabilecek, sınıf ve ilişki nitelikleri toplanmıştır.

İlk olarak, yazılım kaynak kodundan mümkün olduğunca çok nitelik çıkarılmaktadır, daha sonra bu özelliklerden en önemlileri bir seçme algoritması uygulanarak seçilmektedir. Ayrıt sınıflandırma için, ayrıtla temsil edilen ilişkilerin nitelikleri aşağıda açıklandığı gibi çıkarılmaktadır.

Ayırtın temsil ettiği ilişkinin tüm niteliklerinin kümesi $\boldsymbol{A}$ şu şekilde tanımlanmıştır. $\boldsymbol{A} = \{a_i\},\ i=1,\ldots,p,$ burada $\{a_i\}$ ayrıtın i niteliğidir p ise tüm özelliklerin sayısıdır. Eğer iki sınıf arasında aynı yönde birden fazla ilişki varsa, bu ilişkiler daha basit bir çizge elde etmek için tek bir ayrıtta birleştirilmektedir. Birleştirilen ayrıtların tüm nitelikleri aynı ayrıt veri kümesi içinde toplanmaktadır. Örneğin, iki sınıf arasında hem <creates> hem <calls> ilişkileri varsa, ilgili ayrıt her iki niteliği de taşır öyle ki "<creates>=true" ve "<calls>=true" olur.

Bu tez çalışmasında, toplam 25 ilişki niteliği $(p=25)$ çıkarılmıştır. Bu nitelikleri 2 grupta toplayabiliriz. Birinci grupta doğrudan ilişkiye ait 13 nitelik (a_1-a_{13}) yer almaktadır. İkinci grupta ise, kaynak ve varış sınıfları ile ilgili, 12 nitelik (a_{14}-a_{25}) yer almaktadır. Çıkarılan 12 doğrudan ilişki niteliği Çizelge 6.1'de verilmiştir. Bu çizelgedeki ilk sekiz satır (RA1-RA8), ayrıtın konumda belirleyici olabilecek, farklı tipteki bağımlılıklara işaret eden nitelikleri içermektedir. Bu nitelikler "Boolean" tipiyle ifade edilirler. Aralarındaki bağımlılık kuvvetli olan sınıfların aynı olacağı düşünülür, aynı şekilde kuvvetli bir bağımlılık ilişkisi iç bir ilişki olmalıdır. Örneğin, iki sınıf arasında çift yönlü bir ilişki, bu sınıfların aynı modül içinde olma olasılığının yüksek olduğunu gösterir. Diğer taraftan, bazı ilişki tipleri zayıf bağımlılığı ifade edebilir. Örneğin "isFrom/ToNested" niteliği eğer bir isimsiz (anonymous) ya da iç sınıf (nested) ile bir başka sınıf arasındaki ilişkilerde "true" değerini alır. Bu tarz sınıflar GUI bileşenleri ile kontrolün evrimi (inversion of control) için ya da türetilen dinleyici (listener) nesnelerin, yayıncı (publisher) nesnelere kayıt edilirken (Gözlemci kalıbı - the observer pattern) yaygın olarak kullanılırlar. Genellikle, yayıncı ve somut dinleyici sınıfları farklı modüllerde yer alırlar. Bu nedenle bu tarz isimsiz/iç sınıflarla olan ilişkiler ayrıtın konumu hakkında ipucu verebilirler. Bu tip ayrıtların genellikle dış ayrıt olması beklenir.

Çizelge 6.1 : İlişki nitelikleri (s→d).

Etiket	İsim	Açıklama
RA1	Extends	s, d den türemiş mi?
RA2	Implements	s, d'yi gerçekler mi?
RA3	Calls	s, d'nin metodunu çağırır mı?
RA4	Creates	s, d'yi yaratır mı?
RA5	Has Parameter	s'in, d'yi parametre olarak kullandığı metodu var mı?
RA6	Has Attribute	s'nin, d tipinden bir değişkeni var mı?
RA7	IsFrom/ToNested	İlişki iç sınıfa veya iç sınıftan mı?
RA8	İsBidirectional	s ile d arasındaki ilişki çift yönlü mü?
RA9	Number of Method Called	s'nin d'den çağırdığı metot sayısı
RA10	Number of Common Parameters	s ile d arasındaki ortak parametre sayısı
RA11	Public Method Utilization	Çağrılan açık metotların toplam metot sayısına oranı
RA12	Number of Overridden Methods	s tarafından d'nin örtülen metot sayısı
RA13	CycleLength	s ile d arasındaki döngünün boyutu

İlişki tipi ayrıtın konumunu belirlemede tek başına yeterli olmayacağından, diğer ilişki nitelikleri (RA9-RA13) de göz önüne alınmıştır. Bu nitelikler tam sayı değerleri almaktadır. Örneğin bir sınıf diğer bir sınıfın çok fazla sayıda metodunu çağırırsa, bu sınıfların aynı modülde yer aldığına ve aralarındaki ayrıtın iç bir ayrıt olduğuna bir işaret olabilir. Benzer şekilde metotlarında çok fazla sayıda ortak parametre olan sınıfların da aynı modülde olma olasılığı yüksek olabilir. *"Public Method Utilization"* kaynak sınıf tarafından başvurulan metot sayısının, varış sınıfı tarafından sağlanan tüm açık metot sayısına oranıdır. Eğer bir sınıf diğer bir sınıfın açık arayüzünü büyük oranda kullanıyorsa bu sınıflar arasında yüksek bağımlılık olduğunun göstergesi olabilir. Sonuç olarak, yüksek bağımlı sınıflar arasında ayrıtlar muhtemelen iç ayrıtlardır. *CycleLength* niteliği, sınıflar arasında döngüsel bir bağın varlığını ve derecesini belirlemek için kullanılır. Döngüsel bağ sınıflar arası sıkı bağımlılıkla ilgili bir karşı-kalıp (anti-pattern) olarak değerlendirilir. Bu yüzden, iki sınıf arasında küçük uzunlukta bir döngü varsa aynı modülde oldukları varsayılabilir. *ShortestPathLength(G, v_s, v_d)*'nin v_s'den v_d'ye olan eğer varsa en kısa yolu yoksa ∞ dönen bir fonksiyon olsun, buradan çıkışla, döngü uzunluğu *CycleLength(e(*v_s*;* v_d*))= ShortestPathLength(G, v_d, v_s) +1* şeklinde hesaplanabilir.

Ayrıtların doğru konumunu tahmin etmek için, ayrıtın kaynak ve varış düğümlerini oluşturan sınıfların temel özellikleri de göz önüne alınmalıdır. İkinci nitelik grubu 12

nitelik (a_{14}-a_{25}) içerir, bunlardan 6 tanesi kaynak sınıfla (a_{14}=SA1- a_{19}=SA6), 6 tanesi ise varış sınıfla ilgilidir (a_{20}=DA1- a_{25}=DA6). Bu nitelikler Çizelge 6.2'de verilmiştir. Örneğin Java programlama dilinde, arayüzler genellikle, gevşek bağlı yazılımlar oluşturmak ve gerçekleme detaylarını modüllerden yalıtmak amacıyla kullanılır. Ayrıca yazılım geliştiriciler gerçekleme detaylarını çocuk sınıflara sağlamayı tercih ederek yazılımın yeniden kullanabilirliğini sağlamak istediklerinde, soyut (abstract) sınıflar kullanır. Örneğin, kontrolün evrimi kalıbını (Inversion of Control pattern - IoC) [130] gerçeklemek için bağımlılığı azaltmak amacıyla Java arayüzleri (interfaces) kullanılır. Arayüz ve soyut sınıflara olan ya da bu sınıflardan olan ilişkiler ayrıtların konumsal olasılığını belirlemede ciddi rol oynayabilir. *"Number of Public Methods"* (NPM) niteliği uygulama program arayüzünün (Application Programming Interface - API) boyutunu ölçmek için kullanılabilir [131].

API boyu yüksek sınıflar servis kapısı rolü oynarlar. Bu sınıflar kendilerine dış ilişkiler üzerinden gelen istekleri alırlar ve kendilerinden çıkan iç ilişkilerle ilgili sınıflara dağıtırlar. Düğüm derece dağılımı açısından, nesneye dayalı çizgeler heterojendir ve güç-kanunu dağılımına göre karakterize edilirler. Bu nedenle, düğümler, derece, giren ve çıkan derece nitelikleri de ayırt edici özellik olabilmeleri açısından nitelik kümesine eklenmiştir.

Çizelge 6.2 : Kaynak ve varış nitelikleri (s→d).

Etiket	İsim	Açıklama
SA1	isInterface_S	s bir arayüz mü? (Boolean)
SA2	isAbsract_S	s bir soyut sınıf mı? (Boolean)
SA3	InDegree_S	s'ye giren toplam ayrıt sayısı
SA4	OutDegree_S	s'den çıkan ayrıt sayısı
SA5	Degree_S	s'nin toplam ayrıt sayısı
SA6	NPM_S	s toplam açık metot sayısı
DA1	isInterface_D	d bir arayüz mü? (Boolean)
DA2	isAbsract_D	d bir soyut sınıf mı? (Boolean)
DA3	InDegree_D	d'ye giren ayrıt sayısı
DA4	OutDegree_D	d'den çıkan ayrıt sayısı
DA5	Degree_D	d'nin toplam ayrıt sayısı
DA6	NPM_D	d'nin toplam açık metot sayısı

Düğüm sınıflandırma için, düğüm tarafından temsil edilen sınıfların karakteristikleri çıkarılmaktadır. $\boldsymbol{C}$ sınıfların tüm karakteristiklerinin kümesi, $\boldsymbol{C}=\{c_i\},\ i=1,\ldots,q,$ şeklinde tanımlanmıştır. Burada $\{c_i\}$ sınıfın i. karakteristiğini ifade eder ve q çıkarılan toplam karakteristik sayısıdır.

Bu çalışmada, düğümün konumu ile ilgili olabilecek 20 sınıf karakteristiği (c_1-c_{20}) Çizelge 6.3'de gösterilmiştir. Karakteristiklerin çoğu, sınıfın bağımlılık, uyumluluk, karmaşıklık ve boyut gibi kalite nitelikleri ile ilgilidir. Bu metriklerin ayrıntılı tanımları Bölüm 4.3'de anlatılmıştır. Farklı yazılım projeleri arasında daha tutarlı sonuçlar elde edebilmek için, yazılım projelerinin özelliklerine ve boyutuna çok bağlı olan gerçek metrik değerleri yerine, box-plot tekniği [132] ile metrikler normalize edilerek ve nicelenerek kullanılmıştır.

Çizelge 6.3 : Düğüm sınıflandırma için sınıf nitelikleri.

Etiket	İsim	Açıklama
c_1	isInterface	Arayüz mü? (Boolean)
c_2	isAbstract	Soyut sınıf mı? (Boolean)
c_3	CAM	Metotlar arası uyumluluk
c_4	DIT	Türetim ağacının derinliği
c_5	CBO_APP	Sınıflar arası bağımlılık (yazılım içinde)
c_6	CBO_LIB	Sınıflar arası bağımlılık (kütüphane sınıflarına)
c_7	Degree	Toplam ayrıt sayısı
c_8	In-Degree	Giren ayrıt sayısı
c_9	LCOM	Metotların uyumluluğu
c_{10}	LOC	Kod satır sayısı
c_{11}	NOC	Alt sınıf sayısı
c_{12}	NOF	Nitelik değişkeni sayısı
c_{13}	NOM	Metot sayısı
c_{14}	NORM	Örtülen metot sayısı
c_{15}	NOSF	Statik değişken sayısı
c_{16}	NOSM	Statik metot sayısı
c_{17}	Out-Degree	Çıkan ayrıt sayısı
c_{18}	RFC	Sınıfın tetiklediği metot sayısı
c_{19}	SI	Özelleşme endeksi
c_{20}	WMC	Sınıfın ağırlıklı metot sayısı

Box-plot algoritması sıralı sayılardan oluşan bir sayılar kümesini, kümenin ortanca (median) değerlerini kullanarak çeyreklikler (quartiles) halinde farklı seviyelere ayırmak için kullanılır. Ortanca değeri, sıralı sayılardan oluşan bir kümenin yüksek değerli elemanlarını, düşük değerli elemanlarından ayıran ve kümeyi yaklaşık olarak ikiye bölen ve kümenin ortasında bulunan elemandır. Sıralı sayı kümesi üzerindeki çeyreklikleri ifade etmek için üç değer (Q1,Q2 ve Q3) hesaplanır. İlk çeyreklik değeri Q1 kümenin en düşük değerli %25 elemanını kalanından ayırır, ikinci çeyreklik Q2 kümeyi ikiye böler ve üçüncü çeyreklik Q3 kümenin en yüksek değerli %25 elemanını ayırır. Daha sonra hesaplanan Q1 ve Q3 çeyreklik değerlerinin arasındaki fark olan IQR değeri, IQR =Q3-Q1 olacak şekilde hesaplanır. Bu fark

yardımıyla sayı kümesindeki aşırı yüksek ve aşırı düşük elemanlar belirlenir. Kümedeki aşırı yüksek değerli elemanlar QH = Q3 + (1.5)·IQR değerinden yüksek elemanlardır. Aşırı düşük değerli elemanlar ise QL = Q1 – (1.5)·IQR değerinden düşük elemanlardır. Buna göre bir yazılımdaki sınıflar için hesapladığımız metrik değerleri aşağıda gösterildiği gibi normalize edilmektedir:

MV sıralı bir metrik kümesi olmak üzere: MV = {mv_1, mv_2,… mv_n}, ve N(mv_i) kümesi mv_i değerlerinin normalize edilmiş halleri olsun. İlk aşamada MV kümesi için Q1, Q2, Q3, IQR, QL ve QH hesaplanır. Daha sonra N(mv_i) değerleri şu şekilde hesaplanmaktadır:

- N(mv_i) = 1 (en düşük) if mv_i <= QL
- N(mv_i) = 2 (düşük) if mv_i > QL ve mv_i <= Q1
- N(mv_i) = 3 (orta) if mv_i > Q1 ve mv_i <= Q3
- N(mv_i) = 4 (yüksek) if mv_i > Q3 ve mv_i <= QH
- N(mv_i) = 5 (en yüksek) if mv_i > QH

Çizelge 6.3'de verilen niteliklerin seçiminde, doğru tasarlanmış nesneye dayalı yazılım sistemleri göz önünde bulundurulmuştur. Modülerlik prensibine göre, yüksek uyumlu ve daha karmaşık sınıfların modülün içinde yer alması beklenirken, dışarıdan gelen farklı istekleri ilgili iç sınıflara dağıtan, daha az uyumlu önyüz (Facade) sınıflarının kenarlarda yer alması beklenir. Kenar sınıflar sorumlulukları iç sınıflara delege eden çok sayıda metoda sahip olabilirler. Bunun yanı sıra, kenar sınıflar, sistemde daha çok metot çağrısını tetikleyebilir ve daha yüksek *"Response for a Class"* (RFC) metrik değerine sahip olabilir. Bu nedenle, uyumluluk ve karmaşıklık metriklerinin sınıfları kenar ve iç sınıf olarak ayırmada önemli olduğu düşünülmüştür. Ayrıca, kütüphane ve uygulama bağımlılıkları [66] çalışmasında olduğu gibi ayrı olarak değerlendirilmiştir. Kenar sınıflar modülün kapısı olarak hizmet ederler, işi genelde kendileri yapmazlar ve kütüphane servislerine ihtiyaç duymazlar. Diğer yandan iç sınıflar modülün belirli hizmetlerini yerine getirmek için kütüphane sınıflarını daha yoğun kullanırlar.

Fazlalık ve ilgisiz nitelikleri elemek için mRMR [133] algoritmasına başvurularak, en önemli niteliklerin uygun (optimal) kümesi bulunmaya çalışılmıştır. mRMR sınıflandırma sistemlerini oluşturmak için en uygun özellikleri, maksimum

ilgililiklerine göre seçmeye yarayan bir algoritmadır. Algoritma iki farklı ayrık özellik kümesi için benzeşme miktarı ve bilgi uyuşması gibi değerlere bakarak filtreleme ve seçim yapmaktadır. Sınıflandırma için en uygun (highly relevant) değerler sınıflandırmanın başarımını arttıracaktır. Dolayısıyla algoritmanın ana hedefi doğal olarak sistemi en iyi şekilde sınıflandıracak özellikler alt kümesini seçmektir. Aynı zamanda algoritma sınıflandırma sisteminin karmaşıklığını ve boyutlarının azaltılması için özellik kümesinden bir biri ile çok alakalı özelliklerin veya gereksiz tekrarların filtrelenmesini sağlamaktadır. Algoritma ayrık ve sürekli değerlerden oluşan özellik kümeleri üzerinde çalışabilmektedir.

Ayrıtlar için nitelik seçme algoritması, referans projeden edinilmiş *p* özellikli örnek ayrıt kümesini alır ve istatistiksel olarak örnek kümeyi en iyi karakterize eden *p'* nitelik listesini *(p'<p)* verir. mRMR için eğitim verisi, referans küme için daha önceden sınıflandırılmış örneklerin kümesidir; $SE = \{se_1, se_2, ...se_m\}$, burada se_i *i.* ayrıtın nitelik vektörüdür ve m eğitim kümesindeki toplam ayrıt sayısıdır. Her bir se_i *(p+1)*-boyutlu bir vektördür $(a_{1,i}\ a_{2,i}\ ....\ a_{p,i},\ ec_i)$, burada $a_{j,i} \in$ A e_i ayrıtının *j.* niteliğini ve ec_i ayrıtın referans projenin geliştiricileri tarafından belirlenmiş kategorisini (iç ya da dış) ifade eder. Bu nedenle *SE*, *m* x *(p+1)* boyutlu bir matristir ve bu matrisin i. satırı se_i, e_i ayrıtının nitelik vektörünü içerir. Son sütün *(p+1)* hariç olan sütunlar da ayrıtın niteliklerini içerir. Son sütün ise ayrıtın kategorisini belirtir. En iyi niteliklerin *p'* boyutlu *A'* kümesini seçmek için SE kümesi üzerinde mRMR algoritması uygulanır (*p'<p* and $A' \subset A$; *(A'=mRMR(SE)).* Seçilen niteliklerle ayrıt sınıflandırıcısı için eğitim kümesi *SE'* oluşturulur. *SE'* boyutu *m* x *(p'+1)'dir* ve $SE' \subset SE$. *SE'* daha az nitelik içerir ama hala *m* ayrıt düğümü içerir.

Ayrıtlara benzer olarak, referans projeden elde edilen sınıflandırılmış örnek düğümlerin eğitim kümesi $SV = \{sv_1, sv_2, ..sv_n\}$ olsun, burada sv_i vi düğümü tarafından temsil edilen i. sınıfın karakteristik vektörüdür ve *n* kümedeki toplam sınıf sayısıdır. Her bir sv_i *(q+1)*-boyutlu vektördür $(c_{1,i}\ c_{2,i}\ ....\ c_{q,i},\ vc_i)$, burada $c_{j,i} \in C$ v_i düğümü tarafından temsil edilen sınıfın j. karakteristiği ve vc_i referans projenin yazılım geliştiricileri tarafından belirlenen kategorisidir (iç ya da kenar). Bu nedenle, SV *n* x *(q+1)* boyutlu bir matristir. mRMR ile nitelik seçme algoritması *q* karakteristikli SV matrisi alarak, örnek kümeyi en iyi karakterize eden *q' (q'<q)* karakteristikli *C'* vektörünü verir *(C'=mRMR(SV)).* Seçilen karakteristikler $C' \subset C$

kullanılarak, düğüm sınıflandırma için *SV'* eğitim kümesi oluşturulmaktadır. *SV'*nin boyutu *n x (q'+1)* olur ve $SV' \subset SV$ dir.

Ayrık değişkenler için mRMR algoritması şu şekilde tanımlanmıştır [133]:

$$\max_{i \in \Omega_S} \left\{ \frac{I(i,h)}{\left[\frac{1}{|S|} \sum_{j \in S} I(i,j)\right]} \right\} \tag{6.3}$$

Burada *S* nitelik kümesi, *h* hedef kategoriler, *I(i,j) i.* ve *j*. nitelik arasındaki karşılıklı bilgidir. Sürekli değişkenler için algoritma şu şekilde tanımlanmıştır:

$$\max_{i \in \Omega_S} \left\{ \frac{F(i,h)}{\left[\frac{1}{|S|} \sum_{j \in S} |co(i,j)|\right]} \right\} \tag{6.4}$$

Burada *F(i,h)* F-statistik ve *co(i,j)* korelâsyondur. Dikkat edilirse algoritma ilgililiği maksimum yapmaya çalışırken, fazlalığı minimum yapmaya çalışmaktadır.

Bu çalışmada, 25 ilişki niteliği çıkarılmış ve mRMR algoritması ile bunlardan 11 tanesi ayrıt sınıflandırma için seçilmiştir *(p=25, p'=11)*. Benzer şekilde düğüm sınıflandırma için 20 sınıf karakteristiğinden 11 tanesi mRMR ile seçilmiştir *(q=20, q'=11)*. Seçilen nitelikler büyük oranda eğitimde kullanılan veri kümesine dayandığından, seçilen nitelikler deneysel eğitim sürecini anlattığımız bölümde verilmiştir.

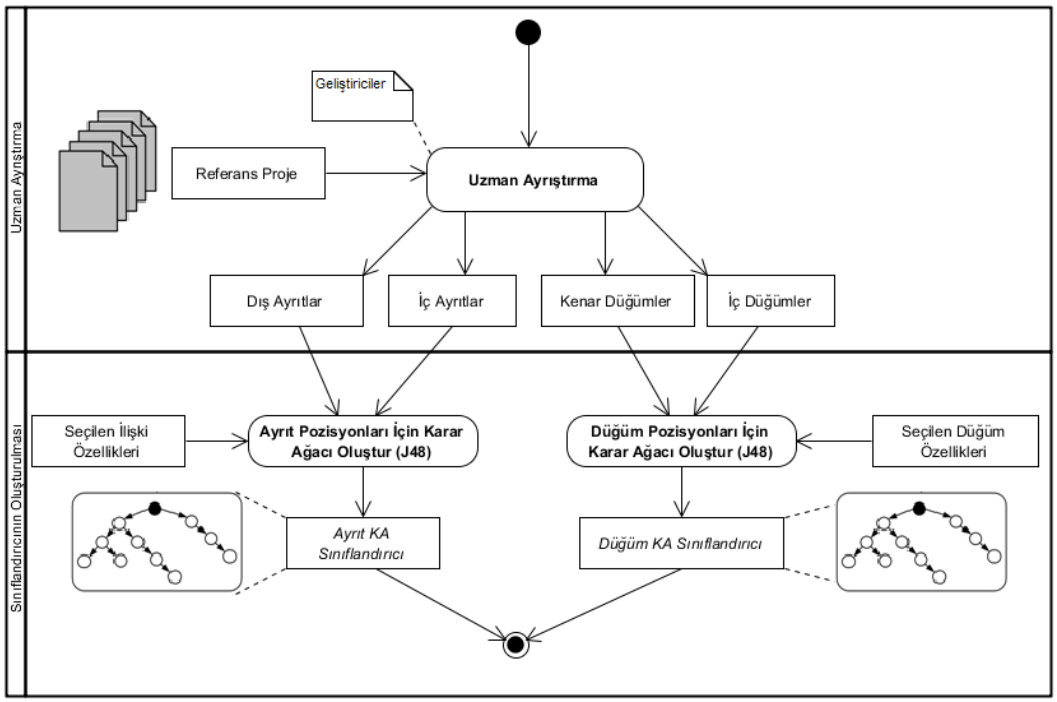
6.2.2 Sınıflandırıcıların Oluşturulması

Sınıflandırma aşamasının ilk amacı ayrıtları iç ve dış kategorisine ayırmak, ikinci olarak düğümleri kenar ve iç kategorisine ayırmaktadır. Sınıflandırıcılar Karar Ağacı (KA - Decision Tree [DT]) şeklinde inşa edilir ve bir önceki bölümde anlatılan nitelikleri içeren, daha önce incelenmiş örneklerin (düğüm ve ayrıtların) kategorileri üzerinden eğitim veri kümesiyle eğitilirler. Öğrenme aşamasından sonra, yeni bir örneğin niteliği sınıflandırma algoritmasına verildiğinde, algoritma bu örnek için kategori olasılıklarını üretir.

KA her bir düğümünün olası alternatifler içinde bir seçimi temsil ettiği yönlü bir ağaçtır. KA bir giren ayrıtı olmayan bir kök düğüme sahiptir. Yaprak düğümlerin ise çıkan ayrıtını yoktur ve bu düğümler sonuçta varılan olasılıksal sınıflandırma

sonuçlarını içerir. Sınıflandırma kök düğümden yaprak düğüme ulaşıncaya kadar yönlendirilerek devam eder. Yaprak düğümler bir sınıflandırma sonucuyla işaretlenmiştir.

Karar ağaçlarını oluşturma süreci Şekil 6.4'de gösterilmiştir. KA'ları oluşturmak ve eğitmek için, eğitim kümeleri *SE'* ve *SV'* üzerinde C4.5 algoritması [134] kullanılmıştır. Öğrenme aşamasında, algoritma eğitim verisini kategori alt kümelerine etkin şekilde ayırmak için örneklerin uygun niteliklerini seçer. Algoritma özyinelemeli (recursive) olarak veri kümesi üzerinde çalışır ve her bir adımda, kategoriyi daha spesifik temsil eden alt kümelere ayırır. Niteliklerin seçimi, entropi (normalize edilmiş veri kazancı) değerinin farkına göre yapılır. Yinelemenin her bir adımında, kategorize etme kararını vermek için en yüksek değerli nitelik seçilir. Bu amaçla C4.5 algoritmasının Weka [135]'deki gerçeklemesi olan J48 kullanılmıştır. Bu gerçekleme birçok araştırma projesinde de başarıyla kullanılmıştır. J48'in budanmamış ve budanmış karar ağaçları üretme yeteneği vardır. Budanmış karar ağaçları ile daha küçük ve daha az karmaşık ağaçlar oluşturulabilir [136].



Şekil 6.4 : Sınıflandırıcıların inşası.

Ayrıt sınıflandırma ayrıtların eğitim kümesi $SE' = \{se'_1, se'_2, ...se'_m\}$ üzerinde C4.5 algoritması uygulanarak bir kereliğine eğitilir *TrainEdgeClassifier(SE')*. Hatırlanacağı gibi, *SE'* bir referans projedeki ilişkilerin seçilen niteliklerini içerir.

Ardından, bu sınıflandırıcı (classifier) hedef projenin e_i ayrıtlarını kategorize etmek için şu şekilde kullanılır:

$\{p_{internal}, p_{external}\} = ClassifyEdge(A'_i)$, burada A', hedef projedeki e_i ayrıtı ile temsil edilen ilişkinin, seçilen p' adet niteliğini içeren nitelik vektörüdür. $p_{internal}$ ayrıtın iç ayrıt olma olasılığıdır. $p_{external}$ ayrıtın dış ayrıt olma olasılığıdır ve $p_{internal} + p_{external} = 1'$dir.

Benzer şekilde, düğüm sınıflandırma düğümlerin eğitim kümesi $SV' = \{sv'_1, sv'_2, ..sv'_n\}$ üzerinde C4.5 algoritması uygulanarak bir kereliğine eğitilir *TrainVertexClassifier(SV')*. Hatırlanacağı gibi, *SV'* bir referans projedeki ilişkilerin seçilen nitelerini içerir. Ardından, bu sınıflandırıcı (classifier) hedef projenin v_i ayrıtlarını kategorize etmek için şu şekilde kullanılır: $\{p_{inner}, p_{border}\} = ClassifyVertex(C'_i)$, burada C', hedef projedeki v_i ayrıtı ile temsil edilen ilişkinin, seçilen *q' adet* niteliğini içeren nitelik vektörüdür. p_{inner} düğümün iç düğüm olma olasılığıdır. p_{border} düğümün kenar düğüm olma olasılığıdır ve $p_{inner}+p_{border}=1'$dir.

6.3 Ağırlık Atama Stratejisi

Eğer düğüm ve ayrıt sınıflandırıcılar, yeni örnekleri %100 kesin doğru sonuç üretseydi, p_{inner} ve $p_{internal}$ için olasılık değerleri için sadece *0* ve *1* değerlerini üretirdi. İdeal bir sınıflandırıcının olası çıktılarına karşılık düşen olası ayrıt konumları Çizelge 6.4'de gösterildiği gibi sekiz farklı durumda (S0-S7) olabilir.

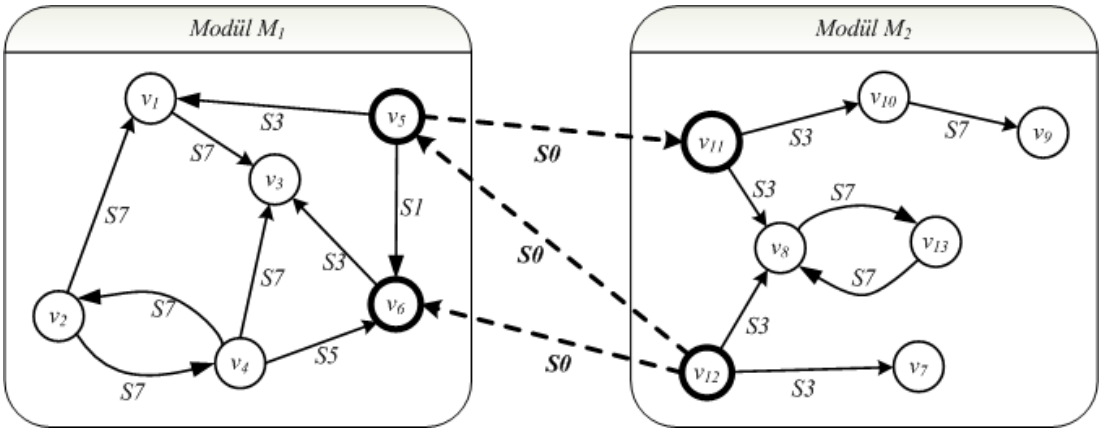
Çizelge 6.4 : İdeal sınıflandırcıların çıktılarına göre ayrıt konumları.

Durum	**Kaynak** p_{inner}	**Varış** p_{inner}	**Ayrıt** $p_{internal}$	**Ayrıt Konumu**
S0	0	0	0	DIŞ
S1	0	0	1	İÇ
S2	0	1	0	GD
S3	0	1	1	İÇ
S4	1	0	0	GD
S5	1	0	1	İÇ
S6	1	1	0	GD
S7	1	1	1	İÇ

Örneğin, eğer kaynak ve varış düğümleri kenar ($p_{inner}=0$) ise ve sınıflandırıcı aralarındaki ayrıtı dış ayrıt ($p_{internal} =0$) olarak kategorize edilmiş ise, ayrıt S0

durumundadır. Ayrıtının kaynak ve varış düğümleri iç ($p_{inner}=1$) ve ayrıt iç ($p_{internal}=1$) olarak kategorize edilmiş ise ayrıt S7 durumundadır. Örnek çizge için ayrıt durumları Şekil 6.5'de gösterilmiştir. Örneğin v12 ve v6 düğümleri arasındaki ayrıt S0 durumundadır ve v5 ve v6 düğümleri arasındaki ayrıt S1 durumundadır.

Eğer düğüm ve ayrıt sınıflandırıcılar tüm varlıkları %100 doğru olarak sınıflandırabilseydi, Çizelge 6.4'deki bazı durumlar oluşamazdı. Örneğin S2 durumunda, ayrıtını dış olarak sınıflandırılmış, varış düğümü ise iç olarak sınıflandırıldığından geçersiz bir durumdur. Çünkü bir iç düğüm dış ayrıta bağlı olamaz. S4 ve S5 durumları da böyle çelişkileri içeren diğer durumlardır. Bununla beraber, sınıflandırıcılar %100 doğru sonuç üretemeyeceği için, bu tür geçersiz durumlar pratikte oluşabileceğinden ayrıtlara ağırlık atanırken bu tür durumlar göz önüne alınmaktadır.



Şekil 6.5 : Ayrıt konum durumları.

Açıktır ki, eğer sınıflandırıcılar ideal olsaydı, o zaman sadece ayrıt sınıflandırma ayrıtını konumunu belirlemek için yeterli olurdu. Ancak, sınıflandırıcılar sadece ayrıtın ve düğümün olası konumu için (iç/dış, kenar/iç) bir takım tahmini olasılıklar üretmektedir. Bu olasılıkları ifade etmek için aşağıda verilen eşik değerleri kullanılmıştır:

p_{inner} *Yüksek eğer* ($p_{inner} >= HIGH\_INNER\_THRESHOLD$) *ve* $0.5 \leq HIGH\_INNER\_THRESHOLD \leq 1$.

p_{inner} *Düşük eğer* ($p_{inner} <= LOW\_INNER\_THRESHOLD$) *ve* $0 \leq LOW\_INNER\_THRESHOLD \leq 0.5$.

p_{inner} *Belirsiz eğer* ($LOW\_INNER\_THRESHOLD < p_{inner} < HIGH\_INNER\_THRESHOLD$).

$p_{internal}$ *Yüksek eğer* ($p_{internal} >= HIGH\_INTERNAL\_THRESHOLD$) *ve* $0.5 \leq HIGH\_INTERNAL\_THRESHOLD \leq 1$.

$p_{internal}$ *Düşük eğer* ($p_{internal} <= LOW\_INTERNAL\_THRESHOLD$) *ve* $0 \leq LOW\_INTERNAL\_THRESHOLD \leq 0.5$.

$p_{internal}$ *Belirsiz eğer* ($LOW\_INTERNAL\_THRESHOLD < p_{internal} < HIGH\_INTERNAL\_THRESHOLD$).

Sınıflandırıcılar ideal olmadığından dolayı, düğüm sınıflandırıcıların ürettiği sonuçlar da dikkate alınmakta ve ayrıtlara p_{inner} ve $p_{internal}$ değerlerine göre üç farklı ağırlık atanmaktadır. Bu ağırlıklar *GÜÇLÜ=ws, NÖTRAL=wn* ve *ZAYIF=ww* burada *ws>wn>ww'*dir.

Ağırlık atama stratejisi için, Çizelge 6.5'de gösterilen bir atama matrisi kullanılmaktadır. Büyük olasılıkla iç olan ayrıtlara (S1, S3, S5, S7) yüksek ağırlık değerleri eşleştirilmektedir. Kümeleme algoritması bu ayrıtlara mümkün olduğunca aynı kümeye koymaya çalışacaktır. Eğer bir ayrıt S0 durumda ise önerilen metot bu ayrıtı muhtemelen dış ayrıt olarak değerlendirir ve düşük ağırlık değerini atar. Kümeleme algoritması büyük olasılıkla bu tarz zayıf ayrıtları kesecektir. Eğer sınıflandırıcıların sonuçları arasında bir çelişki varsa (S2, S4, S6), önerilen yöntem ayrıtın konumunu belirleyemeyecek ve orta ağırlık derecesini atayacaktır (NÖTRAL). Çizelge 6.5'deki S8+ Durumunu en az bir olasılığın (p_{inner}, $p_{internal}$) belirsiz seviyede olduğu durumların grubunu ifade etmektedir. Bu durumda ağırlıklandırma metodu, ayrıta orta ağırlık derecesini atayacaktır (NÖTRAL).

Çizelge 6.5 : Ağırlık atama matrisi.

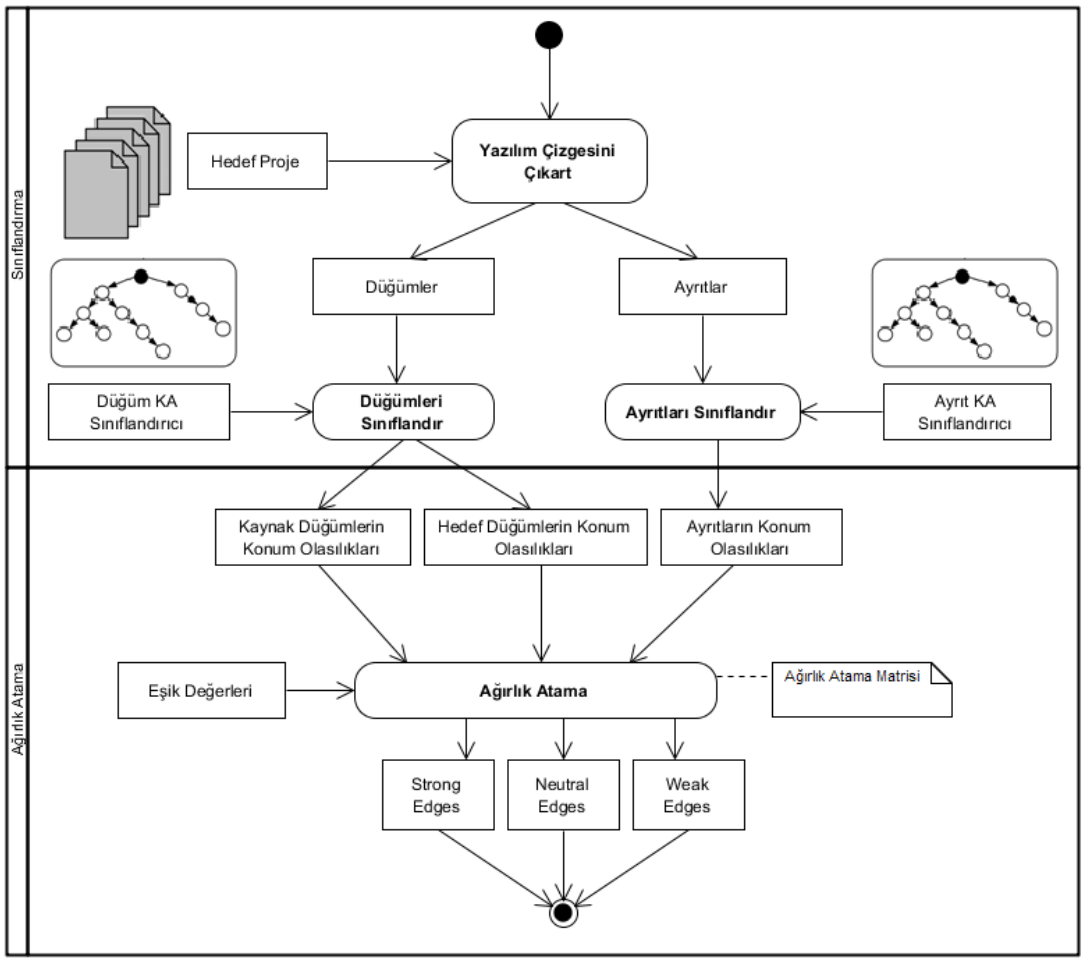
Durum	**Kaynak** p_{inner}	**Varış** p_{inner}	**Ayrıt** $p_{internal}$	**Ayrıt Konumu**	**Atanan Ağırlık**
S0	Düşük	Düşük	Düşük	Dış	***ZAYIF***
S1	Düşük	Düşük	Yüksek	İç	***GÜÇLÜ***
S2	Düşük	Yüksek	Düşük	Kararsız	***NÖTRAL***
S3	Düşük	Yüksek	Yüksek	İç	***GÜÇLÜ***
S4	Yüksek	Düşük	Düşük	Kararsız	***NÖTRAL***
S5	Yüksek	Düşük	Yüksek	İç	***GÜÇLÜ***
S6	Yüksek	Yüksek	Düşük	Kararsız	***NÖTRAL***
S7	Yüksek	Yüksek	Yüksek	İç	***GÜÇLÜ***
S8+	Belirsiz	Belirsiz	Belirsiz	Kararsız	***NÖTRAL***

Eşik ve ağırlık değerleri gibi uygun sistem parametrelerin seçimi gerçek dünya örnekleriyle önerilen metodun inşasını anlattığımız Bölüm 7.3'de anlatılmıştır. Ayrıt ağırlıklandırma stratejisi ayrıca Şekil 6.6'da gösterilmiştir.

6.4 Ağırlıklı Çizge Kümeleme

Çizge kümeleme algoritması, girdi olarak ağırlıklı bağımlılık çizgesini alır ve yazılım sisteminin modüllerinin listesini çıktı olarak verir. [13]'de belirtildiği gibi, bu amaçla nesneye dayalı yazılım bağımlılık çizgelerinin özelliklerine uygun, yönlü ve ağırlıklı

çizgelerde çalışabilen çizge kümeleme algoritmaları kullanılmalıdır. Böyle bir algoritma aynı zamanda, nesneye dayalı çizgelerin doğasına da uyarlanabilir olmalıdır (Dünya küçük davranışı, heterojen çizgeler, daha çok bilgi için bakınız: [137]). Bu nedenle Fast Community algoritmasının [14] ağırlıklı hali kullanılmıştır. Nesneye dayalı yazılım sistemleri birçok gerçek dünya sistemi (hücresel telefon ağları, sosyal ağlar, World Wide Web) gibi dünya küçük davranışı sergilemektedir [123], [137]. Algoritmanın bu tarz davranış gösteren sistemlere uygun olduğu çeşitli çalışmalarla gösterilmiştir [138]. Tez kapsamında yapılan bir alt çalışmada da Fast Community algoritması ağırlıksız olarak kullanılmış ve başarılı sonuçlar alınmıştır [13].



Şekil 6.6 : Ağırlıklandırma stratejisi.

FastCommunity [139] algoritması esasen çok geniş sosyal ağların eniyilemesi için tasarlanmış hızlı bir hiyerarşik birleştirici (agglomerative) algoritmadır (O(m d log n)), ancak yazılım çizgelerinin sosyal ağlara benzer davranışlar gösterdiği düşünülerek bu algoritmanın yazılım çizgelerinde ilginç sonuçlar verebileceği

düşünülerek seçilmiştir. Algoritmanın yönlü ve yönsüz, ağırlıklı ve ağırlıksız çalışması ve küme sayısını giriş olarak almaması bu tür algoritmalara göre büyük fayda sağlamaktadır. Algoritmanın en önemli avantajlarından biri büyük çizgeler için bile oldukça hızlı olmasıdır [140].

FastCommunity algoritması modülerlik düşüncesine dayanmaktadır ve bu modülerlik şu şekilde tanımlanmıştır [108]:

$$Qw = \sum (we_{ii-} wa_i^{\,2}) \tag{6.5}$$

Burada we_{ii} küme *i*nin içinde kalan ayrıtların fraksiyonu wa_i ise bir kenarı *i* kümesindeki düğümlerde kalan ayrıtların ağırlıklı fraksiyonudur. Eğer ayrıtların uçları rastgele dağılmışsa *i* kümesi içindeki düğümleri birbirine bağlayan ayrıtların ağırlıklı fraksiyonu wa_i^2dir. Yüksek *Qw* değeri, yüksek modülerlik ve iyi bir topluluk bölümü ifade eder. Ancak en yüksek *Qw* değerini verecek en iyi kümelemeyi çok geniş arama uzayında bulmak pratik olarak mümkün değildir. Bu nedenle bu eniyileme problemini çözecek sezgisel yaklaşımlar geliştirilmiştir [129].

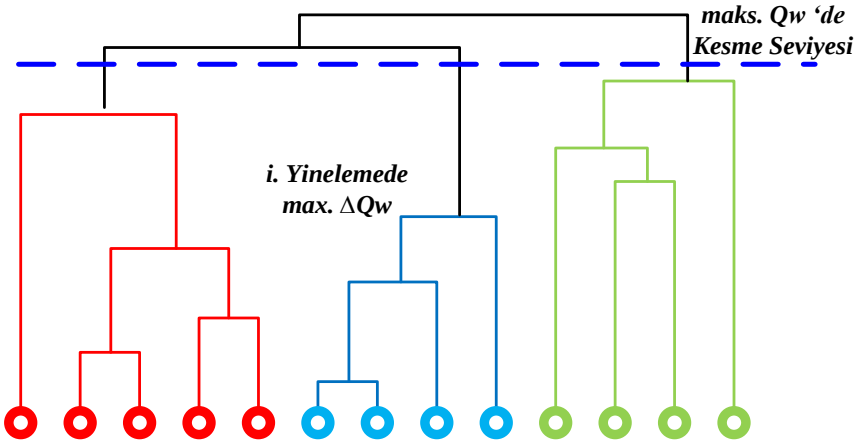
FastCommunity fırsatçı eniyilemesine (greedy optimization) dayalı hiyerarşik birleştirici bir algoritmadır. Algoritma temelde şu adımlarla açıklanmıştır:

1. ilk olarak tüm düğümleri ayrı bir kümeye ata.

2. *Qw* değerini maksimum arttıracak (ya da en küçük azaltacak) iki düğümü birleştir ($max\Delta Q$).

3. 2. adımı tek bir küme kalıncaya kadar tekrarla

4. Oluşan dendrogramda en büyük (maximal) *Q* değerine göre kesme seviyesini (cut level) belirle.



Şekil 6.7 : Örnek bir kümeleme dendrogramı.

Örnek bir kümeleme dendrogramı Şekil 6.7'de gösterilmiştir. *Qw*'nun artırmalı hesaplamasının çalışma zamanı karmaşıklığının $O(|E|dlog|V|)$ olacağı daha önceki çalışmalarda gösterilmiştir [141], burada *d* dendrogramın derinliğidir. Yazılım çizgeleri gibi yoğun olmayan (sparse) ve hiyerarşik yapıya sahip çizgelerde, E|~|V| ve d~log|V|, olduğundan çalışma zamanı karmaşıklık $O(|V|log^2|V|)$ olacaktır.

6.5 Yöntemin Algoritmik İfadesi

Bu bölümde önerilen yöntem algoritmik olarak kısaca ifade edilmiştir. Önerilen yöntemin öğrenme aşaması Algoritma 6.1'de verilmiştir. Öğrenme aşamasında, yöntem bir referans yazılım sistemi (RSS) ve bu sistem için uzmanlar tarafından

belirlenen modül listesini (ED) almaktadır. Bu sistem üzerinde yazılım bağımlılık çizgesi çıkarılmaktadır. Daha sonra uzman ayrıştırmada verilen modül yapısına göre, düğümler "*iç*" ve "*kenar*" olarak, ayrıtlar ise "*modül içi*" ve "*modüller arası*" olarak etiketlenmektedir. Ardından düğüm ve ayırt özellikleri yazılım kodundan çıkarılmaktadır. MRMR algoritmasıyla yardımıyla sınıflandırmada kullanılacak özellikler ayrı ayrı belirlenmekte ve sınıflandırıcılar referans veri kümesinin bu özelliklerle oluşturulan alt kümesi kullanılarak oluşturulmaktadır.

Algoritma 6.1 : Öğrenme aşaması.

```
Initialize EdgeClassifier
Initialize VertexClassifier
Initialize Characteristics Set C
Initialize Attributes Set A
Initialize Selected Characteristics Set C'
Initialize Selected Attributes Set A'

void function learn(SoftwareSystem RSS, ModuleList ED) {
      // input RSS is the Reference Software System
      // ED Expert Module Decomposition, list of vertex sets
      G = extractSoftwareGraph(RSS);
      markEdgesAndVertices(G,ED); // fill VI,VX,EI,EX sets
      SV = extractRelationAttributeSet(RSS, C, ED);
      SE = extractClassCharacteristicSet(RSS, A, ED);
      SV' = selectClassCharacteristics(SV, out C');  // mRMR feature selection algorithm
      SE' = selectRelationAttributes(SE, out A'); // mRMR feature selection algorithm
      VertexClassifier.train(SV'); // C4.5 DT Classifier algorithm
      EdgeClassifier.train(SE'); // C4.5 DT Classifier algorithm
}
```

Yazılım çizgelerindeki toplam ilişki sayısının (m), düğüm sayısının (n) k katı olması sebebiyle, m~n olarak kabul edilebilir. Benzer şekilde sınıflandırmada kullanılan, sınıf ve ilişki özelliklerinin sayısı (f) bir sabit olarak kabul edilebilir. Yöntemin pratik uygulamasında özellik kümesinin boyu f<25'dir. Öğrenme aşamasında kaynak koddan çizgedeki düğüm ve sınıfların özelliklerini çıkarma aşamasın süresi, kaynak kod boyuna göre değişse de, yazılımdaki sınıf sayısıyla doğru orantılı olduğu kabul edilebilir O(n). MRMR algoritmasının karmaşıklığı O(f·n) olarak verilmiştir [133], C4.5 sınıflandırıcılarının eğitilmesinin karmaşıklığı ise $O(f \cdot n \cdot \log n) + O(n \cdot (\log n)^2)$ verilmiştir [134]. Bu durumda eğitim aşamasının çalışma zamanı toplam karmaşıklığı $O(n) + O(n) + O(n \cdot \log n) + O(n \cdot (\log n)^2) = O(n \cdot (\log n)^2)$ olmaktadır. Ancak bu işlem sadece bir kez yapılmaktadır.

Modül bulma prosedürü Algoritma 6.2'de verilmiştir. İlk olarak verilen yazılım sisteminin (S), çizge modeli oluşturulmakta, ardından ayrıtlara ağırlık atanarak ağırlıklı çizgeye Fast Community çizge kümeleme algoritması uygulanarak modüller bulunmaktadır. Ağırlıklandırma işlemi Algoritma 6.3'de ayrıntılandırılmıştır. Ağırlık atanırken tüm ayrıtlar için, ayrıtın kendisi, kaynak ve varış düğümleri için sınıflandırma işlemi yapılmaktadır. Bu durumda ağırlıklandırma aşamasının karmaşıklığı m·(sınıflandırma karmaşıklığı·3) olur. m~n ve C4.5 algoritmasının sınıflandırma karmaşıklığı O(log n) kabulü ile, çizge ağırlıklandırma işleminin karmaşıklığı O(n·log n) olmaktadır. Fast community algoritmasının karmaşıklığı ise bölüm 6.4'de anlatıldığı gibi $O(n \cdot (\log n)^2)$ 'dir. Bu durumda modül bulma yöntemin toplam çalışma zamanı karmaşıklığı, $O(n \cdot \log n) + O(n \cdot (\log n)^2) = O(n \cdot (\log n)^2)$ olmaktadır.

Algoritma 6.2 : Modül bulma.

```
ModuleList function extractModules(SoftwareSystem S){
// returns extracted modules for a given software system
    //input S is the software system for module extraction
    G<V, E>: Software Dependency Graph
    Initialize M; // M is the data structure that contains extracted modules
    Initialize WE; // WE is the data structure that maps edges to weight values
    G = extractSoftwareGraph(S);
    WE = assignWeightsToEdges(G);
    M = clusterWeightedGraph(G,WE); // Fast community detection algorithm
    return M;
}
```

Algoritma 6.3 : Ağırlıklandırma.

```
Initialize thresholds HIGH_INTERNAL_THRESHOLD, LOW_INTERNAL_THRESHOLD
Initialize thresholds HIGH_INNER_THRESHOLD, LOW_INNER_THRESHOL
enum PositionalProb {LOW, HIGH, UNCERTAIN};
enum EdgeWeight {WEAK, NEUTRAL, STRONG};

Map function assignWeightsToEdges(Graph G<V,E>){
      For each e in E{
         PositionalProb edgeProb = getEdgePositionalProbabilty(e);
         PositionalProb sourceProb = getVertexPositionalProbabilty(G.getSource(e));
         PositionalProb destinationProb= getVertexPositionalProbabilty(G.getDest(e));
         WE[e] = getWeightAssignment(edgeProb, sourceProb, destinationProb);
      }
       return WE;
}
PositionalProb  function getEdgePositionalProbabilty(Edge ei){
       A'i = getVertexCharacteristics(ei,,A');
       pinternal = EdgeClassifier.classify(A'i);
      If (pinternal >=HIGH_INTERNAL_THRESHOLD)
         return HIGH;
      If (pinternal <= LOW_INTERNAL_THRESHOLD)
         return LOW;
      else
         return UNCERTAIN;
}
PositionalProb  function getVertexPositionalProbabilty(Vertex vi){
      C'i = getVertexCharacteristics(vi,,C');   // get characteristics of corresponding class
      pinner  = VertexClassifier.classify(C'i); //probability of being an inner vertex
      If(pinner >= HIGH_INNER_THRESHOLD)
          return HIGH;
      else If(pinner <= LOW_INNER_THRESHOLD )
         return LOW;
      else
         return UNCERTAIN;
}
EdgeWeight function getWeightAssignment(edgeProb, sourceProb, destinationProb){
      If(edgeProb == HIGH)
         return STRONG;
      If(edgeProb == LOW and  sourceProb == LOW and  destinationProb == LOW )
         return WEAK;
      else
         return NEUTRAL;
}
```

7. DENEYLER

Tez Çalışması kapsamında, önerilen modül bulma yaklaşımı gerçeklenmiş ve bir modül bulma ve yazılım analiz aracı oluşturulmuştur. Bu araç aynı zamanda çeşitli yazılımlar üzerinde farklı modül bulma yaklaşımlarının analiz ve değerlendirmesinde kullanılabilmektedir. Farklı modül bulma yöntemlerinin değerlendirme metotları bölüm 7.1'de anlatılmıştır. Karşılaştırılan diğer modül bulma yaklaşımları takip eden 7.2 bölümünde verilmiştir. Bölüm 7.3'de veri kümesinin inşa edilmesi ve sınıflandırma sisteminin inşası açıklanmıştır. Açık kaynaklı ve endüstriyel yazılım projeleri üzerinde yapılan deneyler ise Bölüm 7.4'de ayrıntılandırılmıştır.

7.1 Otomatik Yazılım Modülü Bulma Değerlendirme Yöntemleri

Çok sayıda modül bulma metodu olması bunların nasıl kıyaslanacağı sorusunu da beraberinde getirmektedir. Önerilen yöntem son zamanlardaki karşılaştırma çalışmalarında [19] da ortak olarak kullanılan doğruluk, kararlılık ve düzgün modül dağılımı kriterleri açısından değerlendirilmiştir. Temelde modül bulma yöntemleri bu üç kriterle karşılaştırılmaktadır. Bunlar şöyle tanımlanmaktadır:

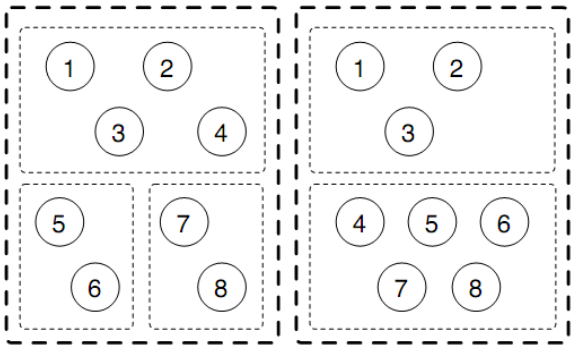
1-Doğruluk: Modül bulma yöntemi tarafından üretilen modüller, o yazılım konusunda uzman bir otorite tarafından üretilen referans modüllere yakın olmalıdır. Referans modül yapısı, yazılım mimari tarafından oluşturulabilir ya da kaynak kodun dizin yapısından çıkarım yapılabilir.

2-Modül dağılımında uç durumların azlığı: Modül bulma ile üretilen modüller arasında çok büyük ya da çok küçük (tekil) modüllerin sayısı az olmalıdır. Çok büyük ya da çok küçük modüller sıradışı uç durum olarak nitelendirilmektedir.

3-Kararlılık: Yazılımın benzer sürümlerindeki modül yapısı benzer olmalıdır. Yazılımın ardışık sürümlerinin oluşturduğu modül yapıları arasındaki fark az olmalıdır.

1. ve 3. kriterleri ölçebilmek için öncelikle iki modül yapısının (kümelemenin) birbirine benzerliği ölçebilmek gerekir. X ve Y iki modül yapısı (kümeleme) sonucu olsun:

MoJo [22]: Aynı yazılımın 2 farklı parçalaması arasındaki uzaklık, birini diğerine dönüştürmek için gerekli minimum taşıma (Move) ve birleştirme (Join) işlemlerinin sayısı olarak tanımlanmıştır. Taşıma bir düğümü alıp bir başka modüle taşımak, birleştirme ise iki modülün birleştirilmesidir. Modül ayırma (split) işlemi ise taşıma işlemleri ile yapılmak zorundadır. Çünkü birleştirme tek bir şekilde yapılabilirken, ayırma üstel sayıda şekilde yapılabilir. Ayırmaya atanan ağırlık, küçük kümenin eleman sayısına eşittir. Örneğin Şekil 7.1'de verilen iki küme için MoJo(X,Y)= 2, MoJo(Y,X)=3 olur.



Şekil 7.1 : Örnek X ve Y kümelemeleri.

Bu metriğin bir türevi olan MojoPlus'da ise bir grup düğüm bir yerden bir yere taşınabilir ve bu işlem 1 birim olarak hesaba katılır. Mojo ve MoJoPlus benzemezlik ölçüsüdür (dissimilarity), MoJoFM ise MoJo metriğinin benzerlik ölçüsüne dönüştürülmüş halidir ve şu şekilde tanımlanır: MoJoFM(X) = (1 − MoJo(X,Y)/ $\max_{\forall Z}$(MoJo(Z,Y))) × 100%

Burada $\max_{\forall Z}$(MoJo(Z,Y)): Y'ye olabilecek maksimum MoJo uzaklığıdır. Tez çalışmasında, literatürdeki birçok çalışmada olduğu gibi karşılaştırmalarda MoJo metrikleri kullanılmıştır.

7.1.1 Doğruluk

Otomatik yazılım modül bulma metodunun uzman bir otorite tarafından yapılan referans modül yapısına yakın bir modül yapısı üretmesi beklenir. Referans modül yapısı, yazılım mimari tarafından oluşturulabilir ya da kaynak kodun dizin yapısından çıkarım yapılabilir. Doğruluk (Authoritativeness) yazılımın mimarları tarafından yapılan ayrıştırma ile otomatik metot tarafından yapılan modül ayrıştırması arasındaki benzerliğin bir ölçüsüdür.

Çeşitli yazılım modül bulma yöntemlerinin sonuçları ile uzman modül yapısı (expert decomposition) arasındaki uzaklığı ölçmek için ***MoJo*** [22] uzaklık metriği kullanılmıştır. Bu metrik daha önceki birçok kümeleme çalışmasında (Örn. [19], [23]) da ortak olarak kullanılmıştır. ***X*** ve ***Y*** iki parçalama olsun, ***mno(X,Y) X*** parçalamasını ***Y*** parçalamasına dönüştürmek için gerekli en az sayıdaki taşıma (move) ve birleştirme (join) işlemleri olsun öyle ki taşıma bir düğümü bir kümeden diğer kümeye taşıma ve birleştirme de iki kümeyi tek bir kümede birleştirme işlemidir [22]. ***MoJo(X,Y)*** ve ***MoJoSim(X,Y)*** şu şekilde tanımlanmıştır:

$$MoJo(X,Y) = \min(mno(X,Y), mno(Y,X)) \quad \textbf{(7.1)}$$

$$MoJoSim(X,Y) = \left(1 - \frac{MoJo(X,Y)}{n}\right) \times 100\% \quad \textbf{(7.2)}$$

Modül yapısının doğruluğu ***MoJoSim(M, MA)*** şeklinde tanımlanmıştır, burada ***MA*** referans modül yapısını, ***M*** otomatik metot tarafından oluşturulan modül yapısını, ***n*** ise modül yapısı çıkartılan yazılım varlıklarının (sınıflar ve arayüzler) sayısını ifade eder.

7.1.2 Kümelemenin düzgün dağılımı

Otomatik modül bulma yaklaşımı çok büyük ya da çok küçük sınıf içeren uç boyutta kümeler oluşturmamalıdır çünkü bu iki durum mimari bileşenler arasında görülmez. Uç boyuttaki modüller uyumluluğunu düşürür ve bağımlılığı arttırırlar. NED (non-extreme distribution) [19] metriği modülleri boyutunun düzgün dağıldığını ölçmek için kullanılmıştır ve şu şekilde tanımlanmıştır:

$$NED = \frac{\sum_{i=1, M_i\ düzgünboyutlu}^{k} |M_i|}{n}, \quad M_i\ düzgünboyutlu\ eğğe 5 < |M_i| < 1.5 \times |MA_{\max}| \quad \textbf{(7.3)}$$

Burada ***k*** kümelerin sayısı, M_i küme ***i***, n yönlü çizgedeki toplam düğümlerin sayısı MA_{max} referans kümelemedeki en büyük kümedir. Yüksek NED değeri iyi bir dağılım göstermektedir.

7.1.3 Kararlılık

Otomatik modül bulma metodunun yazılımın benzer versiyonları için yakın sonuçlar üretmesi yani modül bulma sonucunun kararlı olması beklenir. Bu nedenle yazılımın

ardışık sürümlerinin modülleri arasındaki fark az olmalıdır. Kararlılık Stability(M^n) = MoJoSim(M^n, M^{n-1}), şeklinde tanımlanmıştır, burada M^n ve M^{n-1}yazılımın iki ardışık sürümü için üretilmiş kümele sonuçlarıdır.

7.1.4 Çalışma süresi

Otomatik modül bulma metodunun çalışma süresi kullanılabilirliği ciddi şekilde etkileyebilir. Özellikle kullanıcı modül bulma aracını yinelemeli ve arttırmalı bir yaklaşımla tasarımı çıkarmak için tekrar tekrar kullanıyor ve keşfedilen yapıya göre sistem üzerinde güncellemeler yapıyorsa sık kümeleme ihtiyacı nedeniyle yavaş bir algoritma ile bu işlem çok zaman alabilir. Bu nedenle metotların çalışma süreleri hesaplanmış ve başarımları karşılaştırılmıştır. Çalışma süresi testleri 2.8 GHz Intel i7 işlemciye 8 GB bellek ile yapılmıştır.

7.2 Karşılaştırılan Modül Bulma Yöntemleri

Tez kapsamında önerilen yöntemin başarımını değerlendirmek için son çalışmalardaki yüksek başarımlarına ve gerçeklemelerinin erişilebilirliğine göre, dört önemli modül bulma yaklaşımı seçilmiştir. Literatür taraması bölümünde ayrıntılı olarak anlatılan bu yöntemler aşağıda verilmiştir:

Bunch [126]: Bu yöntemde diğer kümeleme algoritmalarından farklı olarak, kümelemeye bir arama problemi olarak bakılır ve belirli bir MQ fonksiyonunu maksimum değere çekecek, kümeleme aranır.

Oluşturulan n elemanlı bir kümeyi farklı k parçaya ayırmak için Stirling sayıları [127] oluşturur ve olası parçalama sayısı şu şekilde hesaplanır:

$$S_{n,k} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n \qquad \textbf{(7. 4)}$$

n elemanlı bir kümenin olası tüm parçalamalarının sayısı ise şu şekilde hesaplanır:

$$B_n = \sum_{k=1}^{n} S_{n,k} \qquad \textbf{(7. 5)}$$

Bu fonksiyon üstel olarak büyür ve n=20 gibi küçük bir değer için bile parçalama sayısı 50 trilyonun üzerine çıkar dolayısıyla tüm parçalamaları incelemek pratikte imkânsızdır. Bu nedenle sezgisel yaklaşımlar bu sayıyı küçültmek için kullanılır. Arama uzayının küçültülmesi işlemi, kaynak kod hakkında bilgiler (isimler, dizin

yapısı, vs…), yapısal olarak fazla değer ifade etmeyen modüllerin (kütüphaneler, omni-present düğümler) silinmesi ile sağlanır. Sonuç en iyi altı (sub-optimal) olmakla birlikte genellikle yeterli olmaktadır.

Bunch sezgisel yaklaşımla, Modülerlik Kalite Fonksiyonunu (MQ) maksimum yapacak parçalamayı bulmaya çalışmaktadır. Bu fonksiyon şu şekilde tanımlanmıştır:

$$MQ = \begin{cases} \frac{\sum_{i=1}^{k} Ai}{k} - \frac{\sum_{i,j=1}^{k} E_{i,j}}{k(k-1)/2}, & k > 1 \\ A_1, & k = 1 \end{cases}, \quad A_i = \frac{\mu_i}{N_i^2}, \quad E_{i,j} = \begin{cases} 0, & i = j \\ \frac{\varepsilon_{i,j}}{2N_i N_j}, & i \neq j \end{cases} \qquad \textbf{(7. 6)}$$

Burada kümelemenin amacı yüksek uyumlu, gevşek bağlı kümeler elde etmektir. Yüksek uyumluluk küme içi ayrıt sayısının çokluğu ile (A_i), az bağımlılık kümeler arası ayrıtların azlığı ile (E_{ij}) tanımlanmaktadır. N_i i modülündeki eleman sayısını, N_j j modülündeki eleman sayısını, $\varepsilon_{i,j}$ i ve j modülleri arasındaki ayrıt sayısı, μ_i i modülü içindeki ayrıt sayısını, k ise modül sayısını göstermektedir. MQ fonksiyonu ağırlıklı da olabilir. Arama problemi şu temel adımlar halinde gerçeklenmektedir:

1. Rastgele bir parçalanma seçilir.
2. Bu parçalanma içindeki düğümler sistematik bir şekilde MQ değeri daha yüksek olan bir parçalanma olacak şekilde yeniden düzenlenir.
3. Eğer daha iyi bir parçalanma bulunursa süreç yinelenir.
4. Bu tepe tırmanma (Hill-Climbing) yaklaşımı ile sonunda daha iyi bir parçalanma bulunamayınca sonlanır.

İlk rastgele parçalama yerel maksimuma götürebilir bunu aşmak için çok sayıda ilk rastgele parçalanma seçilerek birçok deneme yapılır. Ne kadar çok denenirse en iyi altı (sub-optimal) çözümü bulma ihtimali o kadar artar.

Bunch bir yazılım aracı olarak sunulmakta ve kütüphane modülleri, her yere dağılmış (omni-present) modülleri otomatik olarak belirleyerek bunları kümelemede ayrı tutabilmektedir. Aracın bir diğer özelliği de istenilen modüllerin bir arada olduğunu kullanıcı tarafından belirtilmesine izin vermesidir, böylece araç yinelemeli (iterative) şekilde çalıştırılarak daha iyi sonuçlar elde edilebilir. Bunch çizgede düğüm olarak kaynak dosyaları kabul etmektedir. İlişkiler ise bu dosyalar arasındaki kullanım bağımlılıklarıdır. Yani yazılım bileşenleri arasındaki her türlü ilişkiyi tek tipten bir bağımlılık kabul etmektedir. Bu özellik yöntemin başlıca en büyük dezavantajlarından biridir.

ACDC [20]: Diğer yöntemlerden farklı olarak bu yöntemde, yazılımı anlamaya yönelik sık kullanılan kalıpları kullanarak kümeleme yapılır. Yazılım sistemlerinin elle ayrıştırılmasın kullanılan ortak kalıplardan yararlanır. Yalnız bu kalıplar nesneye dayalı tasarım kalıpları değil, başlık kaynak dosyası ayrımı gibi daha çok prosedürel dillerde görülen az sayıdaki temel kalıplardır. Yazarlar çalışmada yedi tane alt sistem kalıbı tanımlamışlardır.

İlk aşamada birçok modül bu kalıplara göre yerleştirilir ve sistemin iskeleti oluşturulur. Kalan modüller evlat edindirme (Orphan Adoption [128]) ile en güçlü bağlı oldukları modülle aynı kümeye yerleştirilirler. ACDC'de tanımlanan kalıplar şunlardır.

1. Kaynak Dosyası Kalıbı (Source File): Birçok dilde birbiriyle ilgili değişkenler ve prosedürler aynı kaynak dosyasında yer almaktadır bu nedenle aynı dosyadaki fonksiyon ve değişkenler aynı kümede olur. Kısacası ilk durumda çizgedeki atomik düğümler kaynak dosyalarıdır. Nesneye dayalı yazılımlarda kümeleme sınıf bazında yapıldığı için bu kalıbı doğrudan her algoritmayla kullanılmış olur.
2. Dizin Yapısı Kalıbı (Directory Structure): Önemli kümeleme bilgisi bazı durumlarda dizin yapısı içinde saklıdır yani bazı durumlarda dizin yapısı kümelemeyi ifade ediyor olabilir. Bu birçok durumda yanıltıcı olabildiğinden bu kalıp çok sınırlı kullanılır (Örneğin son aşamada evlat edindirmede). Bu kalıba ters bir örnek, C, C++ gibi dillerde kullanılan başlık dosyalarının bulunduğu ayrı dizinler olabilir.
3. Başlık-Gövde Kalıbı (Body Header): C/C++ gibi dillerde, modüller başlık dosyaları ve gövde dosyalarından oluşur (foo.h, foo.c gibi) bu kalıba göre bu dosyalar tek bir modül gibi düşünülmeli ve birleştirilmelidir. Nesneye dayalı yazılımlarda kümeleme sınıf bazında yapıldığı için bu kalıbı doğrudan her algoritmayla kullanılmış olur.
4. Yaprak Koleksiyonu Kalıbı (Leaf Collection): Yazılımda sık görülen bir kalıp da bir grup dosyanın birbirine bağlı olmadığı halde, benzer amaçlara hizmet etmesidir (Örneğin sürücü kodları). Bu dosyalar genellikse sistem çizgesindeki yapraklardır. Esasen bu kalıp paylaşılan komşu yaklaşımına benzemektedir.

5. Destek Kütüphanesi Kalıbı (Support Library): Yazılım sistemleri genellikle birçok modül tarafından erişilen prosedürler içerir. Bu modülleri tek bir modülde toplamak kalan sistemin yapısını keşfetmeyi daha kolay hale getirebilir.

6. Merkezi Sevkiyatçı Kalıbı (Central Dispatcher): Bu kalıp destek kütüphanesinin ikili olarak düşünülebilir. Büyük sistemler genellikle az sayıda yüksek dış-dereceli düğümler içerir, öyle ki bu düğümler çok sayıda modüle bağımlıdır. Örneğin diğer modülleri bir sırada çağırarak belirli bir işlevin yerine getirilmesini sağlarlar. Bu nedenle bunlar da ilk aşamada kümelemede dışarıda tutulur.

7. Alt Çizge Selefi Kalıbı (Subgraph Dominator): Buraya kadar olan kalıplar bizim tüm algoritmalara bir şekilde kazandırdığımız kalıplardı. Alt çizge selefi kalıbına göre çizgede bir grup düğüme erişmek için mutlaka belirli düğümlerden geçmek gerekir. Bu düğümler alt çizgenin selefi olurlar. Alt çizge selefi tek düğüm olduğu gibi bir grup düğümden de oluşabilir. Ancak selef düğüm kümesini bulmak karmaşıklığı çok artırdığı için bu yöntemde tek bir düğüm selefi vardır.

LIMBO [32]: Bilgi teorisine dayanan bir modül bulma yaklaşımıdır. Modül bulma sürecinde bilgi kaybını minimize etme kavramına dayanan hiyerarşik bir modül bulma yöntemidir. Hiyerarşik modül bulma algoritmalarının karşılaştırıldığı çalışmada [16], en başarılı algoritmalardan biri olduğu için karşılaştırmalarda yer verilmiştir.

Jaccard Uzaklığına Dayalı K-Means: K-Means [40], d boyutlu uzayda düğümler noktalara düşürülür. Düğümler uzayda birbirine yakınlıklarına göre kümelere ayrılır. Algoritma giriş olarak küme sayısını almaktadır. Düğümlerin bağımlılık matrisi üzerinde Jaccard uzaklığı temel alınarak ve n (eleman sayısı) boyutlu bir uzayda k-means algoritması çalıştırılmaktadır (d=n). i ve j modülleri arasında;

a: iki modülde de ortak olan özelliklerin sayısı (11)

b: i modülünde olan, j modülünde olmayan özelliklerin sayısı (10)

c: j modülünde olan, i modülünde olmayan özelliklerin sayısı (01)

d: hem i modülünde hem j modülünde olmayan özelliklerin sayısı (00)

olmak üzere, *Jaccard uzaklığı: (b+c)/(a+b+c)* şeklinde tanımlanır.

Tanımda a+b+c = 0 olduğu durumda uzaklık belirsizdir, bu çalışmada a+b+c = 0 durumda uzaklık 1 olarak kabul edilmiştir.

KMeans algoritmasının en büyük dezavantajı küme sayısını giriş parametresi olarak almasıdır. Bir yazılımdaki modül sayısının baştan doğru biçimde tahmin edilmesi oldukça zordur. Bu çalışmada yapılan deneylerde bu algoritmayla farklı küme sayısı parametreleriyle oluşan kümeler incelenmiştir. Benzerlik ölçüsü için, [16] çalışmasında yazılım kümeleme için iyi sonuç verdiği bildirilen Jaccard katsayısı kullanılmıştır.

7.3 Sınıflandırıcıların Eğitilmesi ve İnşası

Düğüm ve ayrıtların sınıflandırmasında karar ağaçlarını oluşturmak ve eğitmek için, Karniyarik Yazılımın sisteminin 1.0.0 sürümünden elde edilen veriler kullanılmıştır. Karniyarik ödüllü bir ticari servis odaklı projedir [142] ve tecrübeli yazılımcılar tarafından nesneye dayalı prensip ve kalıplarına uygun olarak geliştirilmiştir. Veri kümesi için uygun bir kaynaktır çünkü hem genelleyebilecek kadar büyük bir projedir hem de uzman ayrıştırılmasının güvenilirliği ve kesinliği yüksektir. Bu projenin geliştiricileri tez çalışması için yeniden kullanılabilir ve servis tabanlı modüllerini sağlamış ve yazılım çizgesinin verilen modüler yapısına göre geliştirilen araç tarafından düğüm ve ayrıtlar işaretlenmiştir (kenar/iç ve dış/iç). İlk öğrenme veri kümesi 504 ayrıt ve 1517 ayrıttan oluşmaktadır. Daha sonra mRMR algoritması veri setine uygulanarak, 11 en önemli ilişki niteliği 25 nitelik arasından seçilmiştir. Ayrıt sınıflandırma için seçilen ilişki nitelikleri Çizelge 7.1'de verilmiştir.

Çizelge 7.1 : Ayrıt sınıflandırma için seçilen ilişki nitelikleri (s→d).

Etiket	İsim	Açıklama
RA1	Extends	s, d den türemiş mi?
RA2	Implements	s, d'yi gerçekler mi?
RA7	isFrom/ToNested	İlişki iç sınıfa veya iç sınıftan mı?
RA8	isBidirectional	s ile d arasındaki ilişki çift yönlü mü?
RA9	Number of Method Called	s'nin d'den çağırdığı metot sayısı
DA2	isAbsract_D	d bir soyut sınıf mı? (Boolean
DA3	InDegree_D	d'ye giren ayrıt sayısı
DA4	OutDegree_D	d'den çıkan ayrıt sayısı
DA5	Degree_D	d'nin toplam ayrıt sayısı
SA1	isInterface_S	s bir arayüz mü? (Boolean)
SA3	InDegree_S	s'ye giren toplam ayrıt sayısı

Benzer şekilde mRMR algoritması ile 20 sınıf karakteristiğinden 11 tanesi düğüm sınıflandırma için seçilmiştir. Seçilen karakteristikler Çizelge 7.2'de verilmiştir. Nitelik sayısını azalttıktan sonra iki eğitim kümesi *SE'* ve *SV'* elde edilmiştir. 1517 ayrıt x (11 nitelik+ 1 kategori) boyutlu *SE'* ayrıt sınıflandırıcıyı eğitmek için kullanılmıştır. Benzer şekilde 504 düğüm x (11 karakteristik+ 1 kategori) boyutlu *SV'* düğüm sınıflandırıcıyı eğitmek için kullanılmıştır. Eğitim aşamasından sonra, hedef yazılım projesinin çizgesine sınıflandırıcılar uygulanır. Bu sınıflandırıcıların sonuçlarına göre incelenen yazılımların çizgelerindeki ayrıtlara ağırlık atanmaktadır.

Çizelge 7.2 : Düğüm sınıflandırma için seçilen sınıf karakteristikleri.

Etiket	İsim	Açıklama
$\mathbf{c_1}$	isInterface	Arayüz mü? (Boolean)
$\mathbf{c_2}$	isAbstract	Soyut sınıf mı? (Boolean)
$\mathbf{c_4}$	DIT	Türetim ağacının derinliği
$\mathbf{c_5}$	CBO_APP	Sınıflar arası bağımlılık (yazılım içinde)
$\mathbf{c_6}$	CBO_LIB	Sınıflar arası bağımlılık (kütüphane sınıflarına)
$\mathbf{c_7}$	Degree	Toplam ayrıt sayısı
$\mathbf{c_8}$	In-Degree	Giren ayrıt sayısı
$\mathbf{c_{13}}$	NOM	Metot sayısı
$\mathbf{c_{15}}$	NOSF	Statik değişken sayısı
$\mathbf{c_{17}}$	Out-Degree	Çıkan ayrıt sayısı
$\mathbf{c_{18}}$	RFC	Sınıfın tetiklediği metot sayısı

Ağırlıklandırma stratejisinde yüksek ve düşük eşik değeri olarak sırasıyla *0.9 ve 0.1* atanmıştır.

HIGH_INNER_THRESHOLD = HIGH_INTERNAL_THRESHOLD = 0.9 ve

LOW_INNER_THRESHOLD = LOW_INTERNAL_THRESHOLD = 0.1

0.1 ve *0.9* arasındaki olasılık değerleri (p_{inner} and $p_{internal}$) belirsiz olarak kabul edilmektedir. Daha önce de belirtildiği gibi sınıflandırmanın amacı kümeleme sürecine kılavuzluk etmek için uygun ağırlıkların atanmasıdır. Oldukça uç sayılabilecek eşik değerlerinin seçilmesindeki amaç sadece konumlarından emin olunan ayrıtların sınıflandırılmasıdır. Eğer olasılıklar çok düşük ya da çok yüksek değilse, sınıflandırma sistemi kararsız sonuç verecek ve kümeleme algoritmasını yanıltmamak için, orta bir ağırlık ilgili ayrıta atanacaktır. Eğer eşik değeri olarak *0* ve *1* değerlerini seçseydik sistem her zaman kararsız sonuç üretecek ve tüm ayrıtlara aynı değer atanacaktı. Bu da sınıflandırma sisteminin devre dışı bırakılması anlamına gelir. Bu durumda kümeleme algoritması ağırlıkların kılavuzluğu olmadan ağırlıksız

gibi çalışacaktır. Diğer bir uç durum her iki yüksek ve düşük eşik değerlerine *0.5* değeri atamak olabilirdi. Bu durumda her bir ayrıta düşük ya da yüksek ağırlık değeri atanacaktır. Bu belirsiz olasılıklara (*0.5*'e yakın) dayalı kararlar ise kümeleme algoritmasının yanlış yönlendirilmesine neden olabilirdi.

Bölüm 6.4'de anlatıldığı gibi FC algoritması modülerlik kalite fonksiyonu *Qw*'yi maksimum yapacak bir modül ayrıştırması aramaktadır. Algoritma arttırmalı olarak *Qw'*yi hesaplamakta ve her bir yinelemede ΔQw'de maksimum artışı (ya da minimum azalışı) yapan iki modülü birleştirmektedir. Newman çeşitli çizgeler üzerinde çalışmış ve $Qw \in [X-0.5, +1]$'nin tipik değerinin *0.3* ve *0.7* arasında değiştiğini bildirmiştir. Aynı zamanda, güçlü komite yapısı olan çizgelerde *Qw* değerinin *0.6* ve *0.7* arasında değiştiğini raporlamıştır [14]. Bu nedenle çalışılan yazılım çizgelerinde *Qw* değerinin bu aralıklara düşmesi için ağırlıklar şu şekilde belirlenmiştir: Güçlü ayrıtlar için $ws=2w$, nötral ayrıtlar için $wn=w$, ve zayıf ayrıtlar için $ww=0.5w$.

7.4 Açık Kaynaklı ve Endüstriyel Projeler Üzerinde Deneyler

Bu bölümde önerilen yöntemin diğer yöntemlerle kıyaslanması açısından deney sonuçları ve deneylerin yapıldığı yazılım projelerinin kümesi verilmiştir. İlk olarak incelenen yazılım projeleri ile bilgiler takip eden alt bölümde anlatılmış, ayrıntılı deney sonuçları ise ikinci alt bölümde aktarılmıştır.

7.4.1 İncelenen yazılım projeleri

Deneyler kapsamında önerilen yaklaşım, dokümanlarının ve farklı sürümlerinin erişilebilirliğine göre yedi farklı açık kaynaklı ve dört endüstriyel yazılım sistemi üzerinde değerlendirilmiştir. Bu yazılım sistemlerinin bir kısmı önceki kümeleme çalışmalarında da kullanılmıştır. Karşılaştırmaların adil olması açısından bu yayınlarda kullanılan açık kaynak kodlu projeler özellikle kullanılmıştır. İncelenen yazılımların özellikleri Çizelge 7.3'de verilmiştir. Her bir proje için kod büyüklüğü LOC metriği cinsinden verilmiştir. Ayrıca düğüm (sınıf) ve ayrıt (ilişki) sayıları gibi yazılım bağımlılık çizgesinin özellikleri de verilmiştir. Açık kaynaklı projelerin uzman ayrıştırması için eğer varsa demet tanım dosyaları (bundle definition files) kullanılmış ve bu projelerde kullanım tecrübesi olan geliştiricilere doğrulatılmıştır. Eğer açık kaynaklı projelerin demet tanım dosyaları yoksa, paket yapıları uzman

ayrıştırması olarak kullanılmıştır. Endüstriyel projeler için her bir projenin yazılım mimarları kaynak kodlarının uzman ayrıştırmasını sağlamıştır.

İncelenen tüm yazılımlar için öncelikle yazılım bağımlılık çizgeleri incelenmiş ve her yere dağılmış olan sınıflar (omni-present classes [143]) örneğin araçlar, ortak kütüphaneler (util, common, lib), problem uzayına has ilkeller (domain primitives) analiz aracındaki süzme yetenekleri kullanılarak çizgeden silinmiştir. Çünkü bu sınıflar hiç bir servise ait değildir. Aynı zamanda test kodları, örnek kodlar ile kullanılmayan ölü kodlar (dead codes) da çizgeden silinmiştir.

Aşağıda incelenen açık kaynaklı ve endüstriyel yazılımlar için kısa açıklamalar verilmiştir.

Çizelge 7.3 : İncelenen yazılımların özellikleri.

Yazılım	Kod Satır Sayısı (KLOC)	Düğüm Sayısı	Ayrıt Sayısı	Paket Sayısı	Sürüm
		Açık Kaynaklı Projeler			
ArgoUML	156K	1686	5586	93	0.3.4
GEF	63K	756	2349	45	3.8
JFreeChart	72K	401	1420	50	0.9.21
Lucene	285K	2090	11959	205	4.3.1
Solr	102K	777	2423	64	4.3.1
Tomcat	184K	972	3567	116	7.0.42
Weka	111K	455	1385	42	3.3.6
		Endüstriyel Projeler			
BMC	72K	522	1455	109	3.1
Equality	32K	355	1368	43	1.0.2
Karniyarik	54K	706	1805	156	2.0.9
SMC	115K	1057	2768	243	2.6

ArgoUML [144], bir UML modelleme aracı yazılımıdır, UML diyagramları ve örnek kodlar oluşturabilmektedir. Ticari olmayan ve Java diliyle geliştirilmiş bir yazılımdır.

GEF [145], Eclipse platformunun grafiksel düzenleme çerçevesidir. Analizlerde 4 adet alt projesi (Draw2D, GEF, Zest, Layout) kullanılmış ve bu projelerin her biri birer servis kabul edilmiştir.

JFreeChart [146], Java için geliştirilmiş açık kaynak kodlu bir grafik çizme alt yapısıdır. Uzman modül yapısı olarak paket yapısı kullanılmıştır ve birim test sınıfları çıkartılmıştır. Uzun süredir geliştirilen ve çok sayıda sürümü olan bir yazılımdır.

Apache Lucene [147], tamamen Java ile yazılmış yüksek performanslı, tam özellikli bir metin arama motoru kütüphanesidir. Özellikle tam metin arama, çapraz platform gerektiren hemen hemen her uygulama için uygun bir teknolojidir. Apache yazılım vakfı tarafından sağlanan oldukça büyük, esnek arayüzlü, modüler mimarisi ile özelleştirilebilir, kaliteli bir açık kaynaklı yazılımdır.

Apache Solr [148], çok hızlı çalışan açık kaynak kodlu kurumsal arama sunucusu platformudur. Oldukça büyük ve kaliteli bir açık kaynaklı yazılımdır. En önemli özellikleri güçlü tam metin arama, isabet vurgulama, yönlü arama, gerçek zamanlı indeksleme, dinamik kümeleme, veri tabanı entegrasyonu, zengin belge (Word, PDF vb.) işleme ve coğrafi aramayı desteklemesi olarak sıralanabilir. Solr indeksleme alt yapısı dağıtılmış, yedekli ve yük dengeli sorgulama yapabilen, otomatik yük devretme ve kurtarma, merkezi yapılandırma sağlayan, son derece güvenilir, ölçeklenebilir ve hataya dayanıklı bir alt yapıdır. Solr dünyanın en büyük internet sitelerinin çoğunun arama ve navigasyon özelliklerinde kullanılmaktadır. Solr Java ile yazılmıştır ve tam metin indeksleme ve arama için özünde Lucene Java arama kütüphanesi kullanır ve REST gibi HTTP / XML ve JSON API desteklediğinden hemen hemen tüm programlama dillerince kolayca kullanılabilir.

Apache Tomcat [149] açık kaynaklı bir örün (web) ve uygulama sunucu yazılımıdır. Birçok örün projesi tarafından kullanılmaktadır ve tamamen Java kodlarından oluşmaktadır. Karşılaştırmalarda uzman modül yapısı, projenin tanımlanmış jar modülleri baz alınarak oluşturulmuştur.

Weka [135] açık kaynak kodlu çeşitli sınıflandırma, regresyon teknikleri ve ilişkilendirme kuralları sağlayan bir makine öğrenmesi aracıdır (Machine Learning Tool). Uzman modül yapısı olarak paket yapısı temel alınmıştır ve az sayıda sınıf içeren paketler en güçlü oldukları paketlerle birleştirilmiştir.

E-Quality [51] tez çalışması kapsamında geliştirilen nesneye dayalı sistemler için yazılım kalitesi görselleştirme ve analiz aracıdır. Yazılım tasarımı çıkarma, sürüm değişiklik analizi, yazılım kümeleme ve çeşitli tasarım seviyesi klon belirleme yöntemlerini içermektedir. Karniyarik ticari Java tabanlı bir dikey arama motoru bir projesidir, Servis odaklı mimari olarak geliştirilmiştir. Karmaşık bir örün robotu (Crawler), web tabanlı sitelerden fiyat bilgilerini toplamak ve karşılaştırmak için tasarlanmıştır. Uzman modül yapısı, projenin yazılım mimarları tarafından

oluşturulmuştur. Yönlendirici yönetim merkezi (SMC) uzaktan yönlendirici, anahtarlayıcı gibi ağ cihazlarının güvenli olarak merkezi yönetimi için geliştirilmiş ticari bir yazılımdır. Uzman modül yapısı, projenin yazılım mimarları tarafından oluşturulmuştur. Baz istasyonu yönetim merkezi (BMC), GSM baz istasyonlarının örün arayüzüyle konfigürasyonlarının yapılması ve durumlarının izlenmesi için geliştirilmiş ticari bir yazılımdır. Uzman modül yapısı, projenin yazılım mimarları tarafından oluşturulmuştur. BMC ve SMC yazılımları aynı ekip tarafından geliştirilmiş Java tabanlı ticari yazılımlardır.

7.4.2 Deney sonuçları

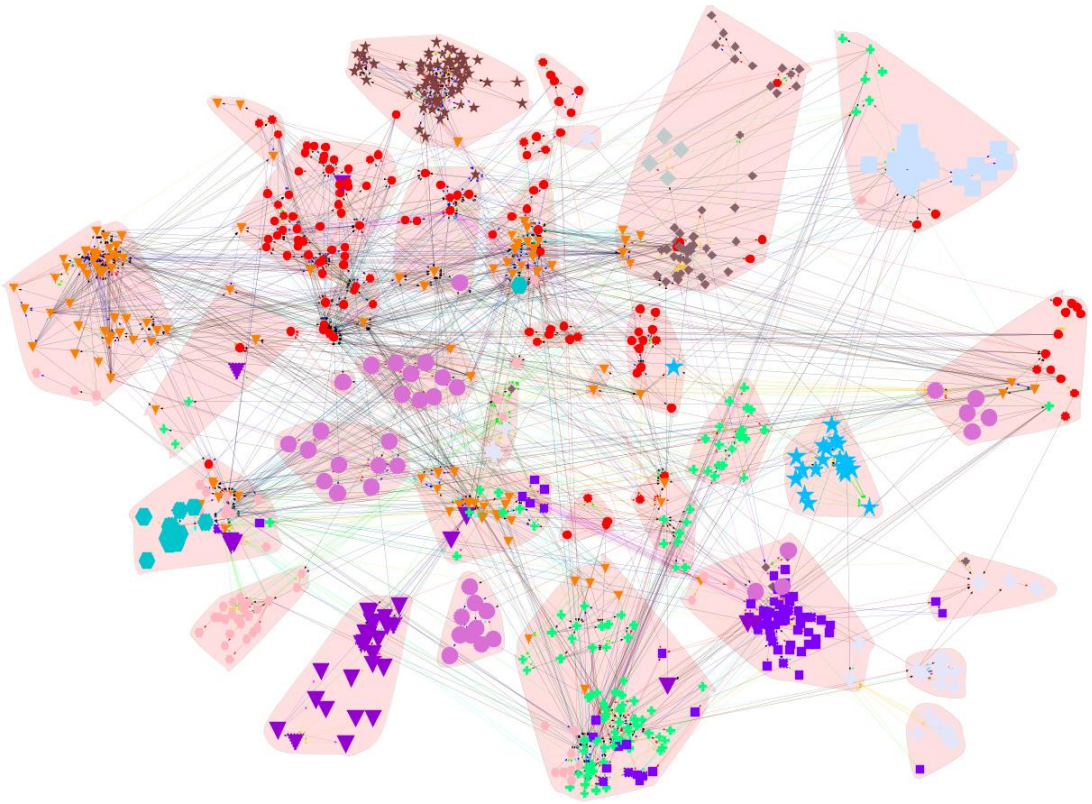
Çizelge 7.4 incelenen nesneye dayalı projelerin farklı algoritmalarla elde edilen doğruluk sonuçlarını göstermektedir. LBME sütunu tez kapsamında önerilen ağırlı yöntemi ve FC sütunu da tez kapsamında ilk geliştirilen ağırlıksız yöntemi göstermektedir [13]. Bu iki sütunu karşılaştırma, sınıflandırma ve ağırlıklandırma mekanizmaları tarafından sağlanan iyileşmeyi göstermektedir.

Çizelge 7.4 : MoJoSim(M,MA) ile ölçülen doğruluk sonuçları.

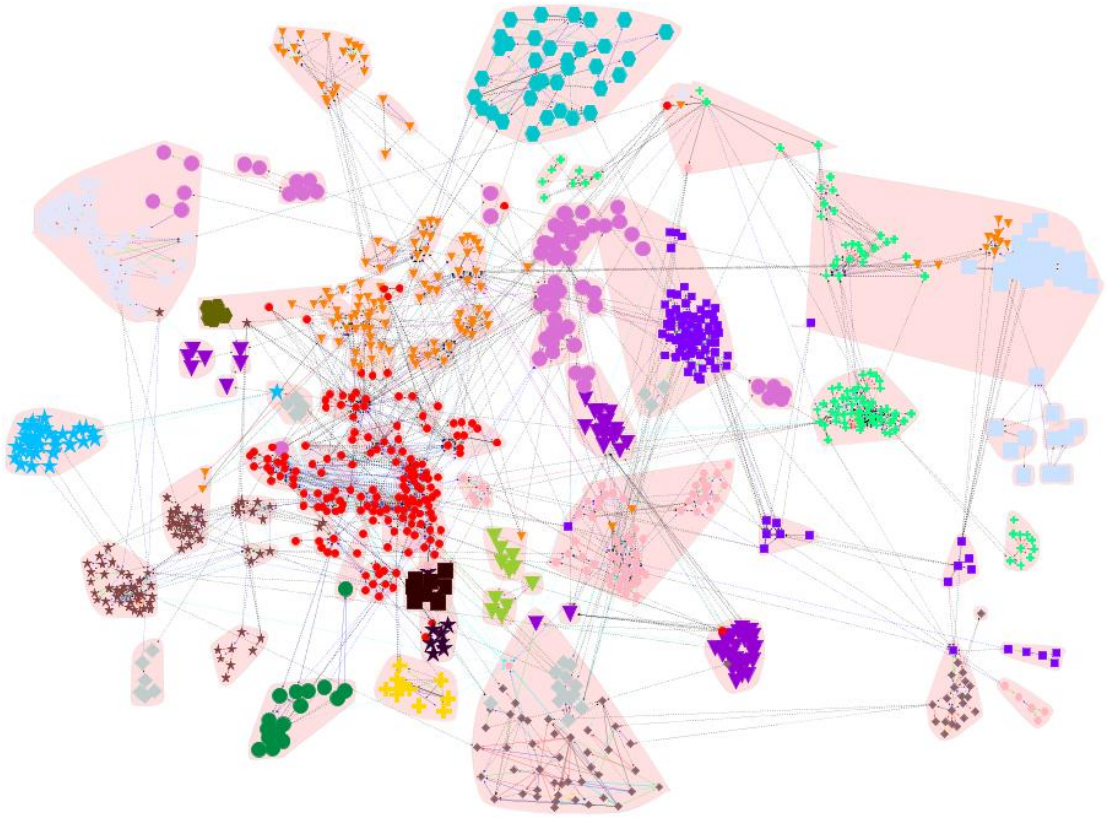
Yazılım	**Metot**					
	LBME	**FC**	**Bunch**	**ACDC**	**LIMBO**	**K-Means**
ArgoUML	**%65,42**	%62,75	%61,03	%60,02	%54,03	%56,05
GEF	**%83,93**	%76,79	%62,50	%54,71	%59,09	%56,49
JFreeChart	**%72,57**	%69,58	%54,61	%51,37	%56,86	%56,36
Lucene	**%73,01**	%68,42	%67,03	%59,43	%52,78	%55,02
Solr	**%75,55**	%68,08	%62,03	%52,12	%53,80	%59,07
Tomcat	**%83,74**	%75,41	%75,31	%64,81	%60,39	%64,09
Weka	**%78,02**	%75,16	%64,35	%58,02	%65,71	%61,10
BMC	**%77,78**	%72,41	%66,86	%66,09	%57,09	%63,03
Equality	**%84,51**	%74,37	%70,14	%72,11	%67,04	%55,21
Karniyarik	**%87,68**	%75,35	%70,11	%69,12	%56,09	%62,04
SMC	**%82,50**	%72,09	%68,31	%65,09	%58,09	%59,04

Çizelge 7.4'de görüldüğü gibi LBME doğruluk kriteri açısından diğer yöntemlerden daha yüksek başarımlı sonuçlar üretmektedir. Değerler %65 ile %87 arasında değişmektedir. ArgoUML projesinin diğer projelere nispeten çok yüksek doğruluk sonucu üretmediği gözükmektedir. Ancak önerilen yöntem yine de diğer metotlardan iyi sonuç üretmektedir. Kullanılan eğitim kümesi modüler ve servis odaklı bir yazılım sisteminden elde edildiği için, yeni önerilen yaklaşımın genel performansı daha modüler ve servis odaklı hedef projeler için artmaktadır. Örneğin, genel olarak

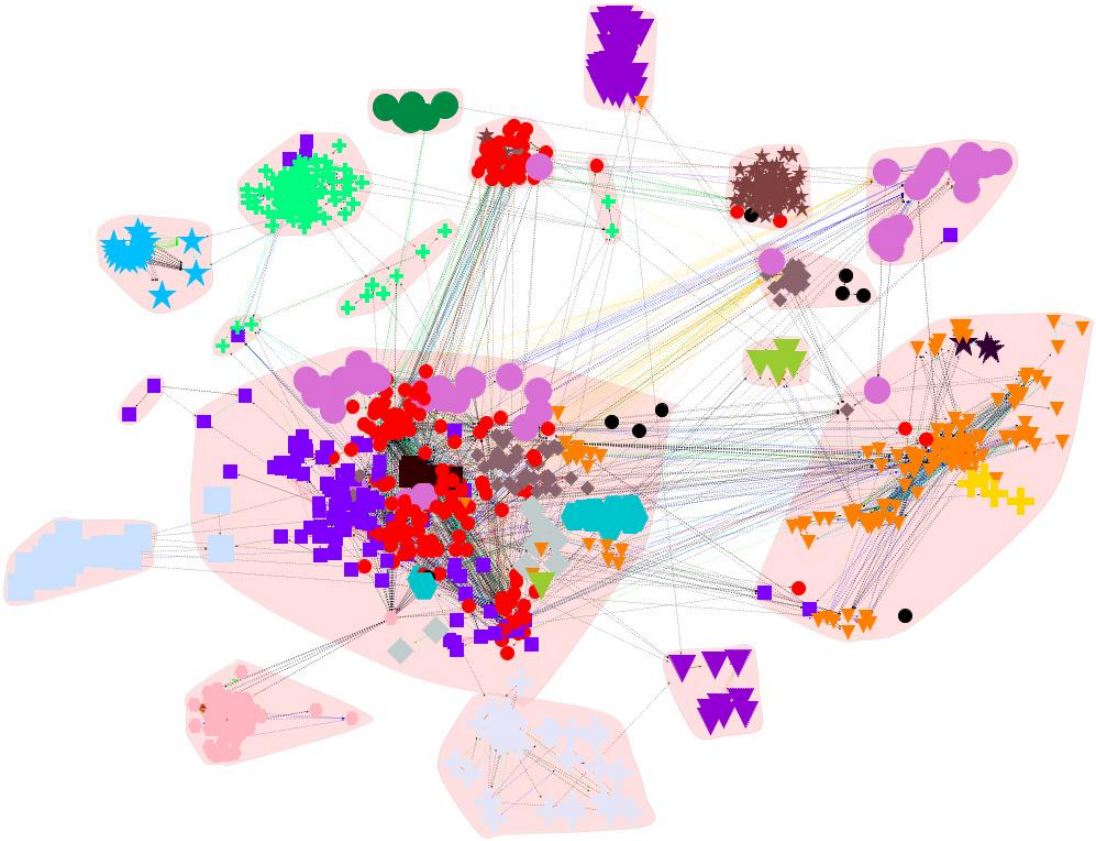
endüstriyel projelerdeki doğruluk sonuçları açık kaynaklı yazılım sistemlerine kıyasla daha iyi çıkmıştır. Bunun, ticari projeler için yeniden kullanılabilirliğin açık kaynaklı projelerden daha kritik olmasından kaynaklandığı düşünülmektedir. Rekabetçi bir pazarda kaynakların daha etkin kullanılmasına ihtiyaç olacağı açıktır. Çeşitli deneyler boyunca yazılım sistemlerindeki sınıfların (sınıflar ve arayüzler) iki farklı tipte olduğu gözlenmiştir. Bazı sınıflar belirli bir servisin üyesiyken bazıları ise sistemin ana kontrol lojiğinin üyesidir. Servisler yeniden kullanılabilir yazılım modülleridir ve güçlü bağlı yazılım sınıflarından oluşur. Kayıtlama (Log), kullanıcı yönetimi, görüntüleme, izleme modülleri yazılım sistemlerinde sık rastlanan ortak servis örnekleridir. Öte yandan, sistemin ana kontrol lojiği, projenin çözüm uzayına (application domain) özel görevleri içerir. Bu sınıflar genellikle yeniden kullanılabilir değildir ve sistemin geneline yayılmışlardır. Analiz edilen projelerin hiç biri %100 servis odaklı değildir ve her projenin belirli oranda bir ana kontrol lojiği vardır. Analizlerde, servislerin diğer sistemin kalanıyla az sayıda olan ilişkisinden dolayı, modül belirleme süreci servisler için daha kolay olduğu gözlenmiştir. Genellikle, bu servisler yazılım sisteminin büyük çizgesinde, görsel olarak fark edilebilen köprülü adalar şeklindedir [150]. Ancak ana kontrol lojiğini sınırlarıyla belirlemek oldukça zordur. Çünkü sistemin hemen her yerine yayılmış karmaşık ilişkiler vardır. İncelenen her sistemin modüler servisleri ve de farklı oranlarda modüler olmayan kontrol lojiği vardır. Bu oranlar aynı zamanda doğruluk sonuçları da etkilemektedir. Eğer servis oranı yüksekse, hedef sistemin modüler ve servis odaklı olduğu, değilse hedef sistem modüler yaklaşımla tasarlanmadığı söylenebilir. Tez çalışması kapsamında aynı zamanda, daha iyi fikir vermesi açısından otomatik çıkarılan modüller ile yazılım uzmanlarının belirlediği modüller, geliştirilen yazılım analiz aracıyla görselleştirilmiştir. Solr, SMC ve Tomcat'in modül yapıları sırasıyla Şekil 7.2, Şekil 7.3 ve Şekil 7.4'de verilmiştir. Bu şekillerde düğümlerin rengi ve şekilleri otomatik çıkarılan modülleri ayırt ederken, düğümleri içine alan dıştaki pembe poligonlar uzmanların belirlediği modülleri göstermektedir. Şekillere dikkat edilirse, sistemin otomatik olarak çıkarılan modüler yapısı, uzmanlar tarafından verilen yapıya oldukça yakındır. Aradaki gözle gözlenen küçük farklar, önerilen metodun uzman ayrıştırmaya göre bazı büyük modülleri iki ya da üç daha uyumlu alt modüllere böldüğü ve bazı çok küçük modülleri ise daha anlamlı tek bir modülde birleştirdiğidir.



Şekil 7.2 : Solr yazılımının modül yapısı.



Şekil 7.3 : SMC yazılımının modül yapısı.



Şekil 7.4 : Tomcat yazılımının modül yapısı.

Analiz edilen sistemler için modül boyutlarının düzgün dağılımına ilişkin NED sonuçları Çizelge 7.5'de verilmiştir. Yüksek NED değerleri istenilen bir dağılım gösterir. Bunch ve LBME için tüm çalışılan sistemlerde yüksek NED değerleri elde edildiği gözlenmektedir. Doğası gereği LBME çok küçük ya da aşırı büyük modüller oluşturmamaktadır. Diğer yandan bazı yazılım sistemleri için ACDC çok küçük modüller oluşturmaya meyilliyken, K-Means aşırı büyük modüller oluşturmaya meyillidir. Bu nedenle bu metotlar için elde edilen NED değerleri göreceli olarak düşüktür. Bazı yazılım sistemleri için FC metodu da çok büyük modüller oluşturmaktadır bu da daha kötü NED sonuçlarına neden olmaktadır.

Çizelge 7.5 : NED sonuçları.

Yazılım	Modül boyutunun düzgün dağılımı					
	LBME	**FC**	**Bunch**	**ACDC**	**LIMBO**	**K-Means**
ArgoUML	%99.17	%99.29	%100.0	%71.47	%100.0	%99.41
GEF	%100.0	%100.0	%100.0	%71.27	%100.0	%100.0
JFreeChart	%99.00	%71.82	%100.0	%74.06	%74.56	%98.75
Lucene	%99.62	%99.47	%100.0	%74.02	%100.0	%99.76
Solr	%99.49	%78.64	%100.0	%62.93	%78.89	%67.82
Tomcat	%98.87	%69.96	%100.0	%74.69	%70.47	%99.79
Weka	%100.0	%75.16	%100.0	%65.71	%100.0	%44.84
BMC	%99.04	%98.28	%100.0	%86.78	%76.25	%71.26
Equality	%100.0	%96.34	%100.0	%85.63	%100.0	%100.0
Karniyarik	%98.58	%98.16	%100.0	%74.93	%100.0	%71.67
SMC	%98.30	%77.34	%100.0	%81.17	%100.0	%75.69

Çizelge 7.6, farklı modül bulma metotlarının çalışma zamanlarını göstermektedir. Altı metot arasında LBME ve FC en zaman etkin yaklaşımlardır. Çünkü iki yaklaşımda düşük hesaplama karmaşıklığı olan Fast Community algoritmasını kullanmaktadır. LBME sınıflandırma süreci nedeniyle, FC'ye göre ihmal edilebilir derecede fazla çalışma zamanına sahiptir. Bu özellik modül bulmada, büyük sistemler için yinelemeli olarak iyileştirme yapıldığında LBME'nin kullanılabilirliğini artırmaktadır. Örneğin ArgoUML için LBME yaklaşık 1 sn'de sonuç üretirken, LIMBO ile bu projeyi modüllere ayrıştırmak 10.5 saati bulmaktadır.

Çizelge 7.6 : Metotların çalışma zamanı.

Yazılım	Çalışma Zamanı (ms)					
	LBME	**FC**	**Bunch**	**ACDC**	**LIMBO**	**K-Means**
ArgoUML	**1075**	***932***	180922	3449	37671462	14082
GEF	**336**	***329***	7154	1081	973026	847
JFreeChart	**214**	***210***	2200	576	210303	315
Lucene	**1441**	***1333***	376932	14061	101926344	31820
Solr	**359**	***353***	14610	1036	2321989	1629
Tomcat	**482**	***436***	24363	1547	45536586	3221
Weka	**278**	***210***	2480	660	326095	445
BMC	**286**	***226***	3347	637	4831871	583
EQuality	**120**	***116***	579	492	263626	81
Karniyarik	**333**	***307***	11602	831	1602393	1321
SMC	**520**	***455***	34218	1168	6688894	3883

Kararlılığın da değerlendirilmesi amacıyla JFreeChart projesinin 0.9.0'den 0.9.20'ye 21 ardışık sürümü analiz edilmiştir. 21 farklı sürüm için kararlılık sonuçları Çizelge 7.7'de verilmiştir. Her bir sürüm için koyu değerler en iyi sonucu ve yan yazılmış * ile biten değerler ise en iyi ikinci sonucu göstermektedir. JFreeChart için bu değerler %70 ile %100 arasında değişmektedir. Elde edilen kararlılık değerleri, LBME'nin ardışık sürümler arasındaki arttırmalı ve küçük değişiklikler olduğu durumlarda, benzer sonuçlar ürettiğini göstermektedir. Hemen hemen tüm sürümler için LBME en iyi ya da en iyi ikinci kararlılık sonuçları üretmektedir. Deneylerde ACDC'nin göreceli olarak yüksek kararlılık sonucu elde etmesinin nedeninin çok sayıda küçük modül ürettiğinden kaynaklandığı belirlenmiştir. Daha önceki bölümlerde de bahsedildiği gibi bu nesneye dayalı sistemlerin modül ayrıştırmasında uygun olmayan bir davranıştır.

Çizelge 7.7 : JFreeChart projesi için kararlılık sonuçları.

Sürümler	**Metot MoJoSim(M^n, M^{n-1})**					
	LBME	**FC**	**ACDC**	**Bunch**	**LIMBO**	**K-Means**
0.9.0	-	-	-	-	-	-
0.9.1	%97.42	%**100.00**	%93.39	%72.58	%*99.20**	%76.00
0.9.2	%84.57	%*87.20**	%86.78	%69.83	%**92.74**	%83.06
0.9.3	%90.45	%**98.45**	%88.00	%76.98	%*95.31**	%65.87
0.9.4	%78.05	%*80.18**	%**83.33**	%61.29	%65.44	%74.65
0.9.5	%*74.12**	%72.14	%**80.00**	%55.56	%69.72	%69.01
0.9.6	%**91.04**	%85.37	%*90.36**	%68.20	%74.91	%81.53
0.9.7	%71.20	%74.63	%*88.55**	%68.68	%82.16	%**89.96**
0.9.8	%**96.51**	%*92.90**	%81.29	%63.52	%74.61	%83.72
0.9.9	%77.01	%80.63	%*81.15**	%67.62	%**82.73**	%70.56
0.9.10	%*85.20**	%**88.85**	%*84.46*	%64.12	%75.17	%80.40
0.9.11	%*81.72**	%80.78	%**87.29**	%73.27	%66.45	%77.56
0.9.12	%**82.14**	%81.41	%*80.72**	%68.23	%71.29	%76.38
0.9.13	%83.45	%75.15	%**85.09**	%66.04	%75.15	%*83.69**
0.9.14	%*88.64**	%**91.64**	%84.10	%65.63	%65.86	%77.58
0.9.15	%*92.14**	%**97.43**	%89.24	%62.10	%65.23	%82.71
0.9.16	%84.20	%**98.00**	%*88.73**	%65.60	%66.67	%78.10
0.9.17	%*82.87**	%80.99	%**85.97**	%57.01	%63.53	%79.94
0.9.18	%**95.14**	%94.61	%*87.67**	%67.93	%58.81	%72.55
0.9.19	%75.29	%*85.29**	%**90.36**	%63.93	%61.31	%74.32
0.9.20	%*91.52**	%**93.64**	%90.98	%63.16	%83.67	%72.63

E-Quality projesinin oniki ardışık sürümü ve GEF projesinin onaltı ardışık sürümü yöntemlerin kararlılık değerlendirmesi maksadıyla incelenmiş ve sonuçları Çizelge 7.8 ve Çizelge 7.9'da verilmiştir. Elde edilen kararlılık sonuçları göstermektedir ki,

LBME yazılımın küçük değişimlerinde benzer sonuçları üretmektedir. Hemen hemen tüm sürümler için LBME oldukça kararlı sonuçlar üretmiştir.

Çizelge 7.8 : E-Quality projesi için kararlılık sonuçları.

Sürümler	**Metot MoJoSim(M^n, M^{n-1})**					
	LBME	**FC**	**ACDC**	**Bunch**	**LIMBO**	**K-Means**
0.1.0	-	-	-	-	-	-
0.1.1	%88.40	%*90.60**	%**93.75**	%87.29	%86.32	%81.20
0.1.2	%**89.12***	%**95.80**	%*88.32**	%79.17	%86.01	%79.02
0.1.3	%96.30	%**100.0**	%92.03	%75.34	%*98.62**	%78.62
0.1.4	%88.40	%**99.20**	%*89.43**	%73.64	%83.85	%73.08
0.1.5	%**100.0**	%**100.0**	%88.17	%85.80	%81.71	%81.71
0.1.6	%**94.22***	%97.14	%90.91	%84.62	%84.02	%81.96
0.1.7	%*86.45**	%84.85	%**92.75**	%82.67	%75.00	%75.00
0.1.8	%78.44	%79.22	%**89.54**	%*79.72**	%78.42	%74.30
1.0.0	%*80.72**	%78.62	%**86.12**	%72.34	%71.50	%70.40
1.0.1	%**83.41**	%80.96	%*82.25**	%76.16	%80.14	%72.58
1.0.2	%79.14	%*80.32**	%**82.12**	%74.14	%77.48	%78.80

Çizelge 7.9 : GEF projesi için kararlılık sonuçları.

Sürümler	**Metot MoJoSim(M^n, M^{n-1})**					
	LBME	**FC**	**ACDC**	**Bunch**	**LIMBO**	**K-Means**
v20110307	-	-	-	-	-	-
v20110408	%**91.64**	%*89.77**	%75.20	%64.01	%63.91	%70.34
v20110309	%*89.43**	%**90.67**	%77.71	%69.81	%72.43	%73.09
v20110208	%**89.72**	%*88.55**	%79.15	%49.30	%76.45	%75.47
v20110109	%*93.54**	%**96.04**	%74.73	%52.16	%68.35	%73.14
v20101212	%**91.83**	%*89.16**	%75.78	%56.94	%67.23	%76.21
v20101111	%**86.36**	%*85.04**	%77.74	%54.85	%64.47	%71.70
v20101011	%*92.15**	%**99.69**	%77.24	%51.93	%64.01	%66.72
v20100909	%*87.44**	%**91.90**	%78.56	%56.19	%68.61	%70.76
v20100809	%**93.34**	%*91.58**	%73.35	%57.98	%24.18	%65.58
v20100709	%*91.82**	%**94.64**	%70.69	%59.66	%21.63	%76.01
v20100609	%*94.41**	%**99.54**	%76.65	%53.63	%81.60	%73.99
v20100509	%*93.28**	%**99.54**	%78.56	%61.92	%63.75	%68.95
v20100409	%**94.59**	%*93.23**	%74.14	%61.06	%73.54	%65.00
v20100309	%*85.13**	%**86.14**	%76.64	%61.36	%71.34	%68.20
v20100209	%**92.82**	%*90.33**	%73.79	%61.69	%59.15	%76.15

8. SONUÇLAR VE ÖNERİLER

Bu tez çalışmasında, nesneye dayalı sistemlerdeki servisleri oluşturan yazılım modüllerinin otomatik bulunmasına ilişkin özgün bir model sunulmuştur. Bu model yazılım sistemlerinin bağımlılık çizgelerinin ağırlıklı hale getirilmesine dayanmaktadır. Sınıfların arasındaki ilişkileri temsil eden ayrıtların ağırlıkları, ayrıtları iç ve dış olarak kategorize eden bir sınıflandırma sisteminin sonuçlarına dayanmaktadır. Bu sınıflandırma sistemi, gerçek modüler ve servis odaklı referans bir sistemden elde edilen verilere göre eğitilmektedir. Nesneye dayalı yazılımlara uygun, ağırlıklı çizge kümeleme algoritması bağımlılık çizgesine uygulanarak yazılım modülleri bulunmaktadır. Bu bağlamda, çeşitli açık kaynaklı ve ticari sistemlerde önerilen yöntemin başarımını değerlendirme amaçlı otomatik bir modül bulma aracı da gerçeklenmiştir. Deney sonuçları göstermektedir ki, önerilen model, nesneye dayalı sistemler için uzman modül ayrıştırmasına yakın, doğruluğu yüksek sonuçlar üretmektedir. Sonuçlar aynı zamanda önerilen yaklaşımın, literatürdeki diğer yöntemlerden kararlılık, çalışma zamanı ve modül boyutlarının düzgün dağılımı açısından da üstün olduğunu göstermektedir.

Önerilen yaklaşımın ve gerçeklenen aracın yazılım tasarımcılarına ve mimarlarına getirdiği faydalar aşağıda özetlenmiştir. Modül çıkarma, yazılım sisteminin servis düzeyinde mimarisini kavrama için destek olmaktadır. Servis adaylarının belirlenmesini ve var olan eski sistemleri servis odaklı mimariye geçirilmesini sağlamaktadır. Bu modüller dağıtılmış sistemlere ya da bulut tabanlı platformlara aktarılarak sistemin performansı, güvenliği ve güvenilirliği arttırılabilmektedir.

Deney çalışmalarında, çıkarılan modül yapısının görselleştirmesinin, geliştirmecilerin sistemlerindeki, modüler tasarım prensipleri açısından problemli kısımları belirlemesinde yardımcı olduğu gözlenmiştir. Geliştiriciler ve mimarlar yanlış konumlandırılmış sınıflar, modüller arası modülerliği bozucu bağımlılıklar gibi çeşitli tasarım kusurlarını keşfedebilmektedirler. Bu bilgilerin ışığında, geliştiriciler sistemlerini daha modüler, daha anlaşılabilir ve yeniden kullanılabilir servis odaklı mimariler elde etmek için iyileştirebilirler. Önerilen metodun güvenirlilik seviyesi, daha büyük yazılım sistem kümeleri ile tecrübe edilip

arttırıldıktan sonra, bu model nesneye dayalı sistemlerin servis odaklı kalitesini ölçmede de kullanılabilir.

8.1 Bilimsel Katkılar

Bildiğimiz kadarıyla literatürde nesneye dayalı yazılımların çizge özelliklerine göre özel geliştirilmiş bir yazılım modül bulma yaklaşımı yoktur. Tez kapsamında bu özelliklere uygun çizge kümeleme algoritmalarının yazılım çizgelerine uygulanmasıyla, tasarım modüllerinin belirlenmesi özgün bir biçimde gerçekleştirilmiştir. Geliştirilen otomatik modül bulma yaklaşımı yazılım çizgelerine uygun kümeleme algoritmaları kullanmasının yanında, nesneye dayalı sistemlerde sıkça başvurulan tasarım prensip ve kalıplarına, kaçınılan kötü kod göstergelerine de başvurarak daha etkin sonuçlar üretilmiştir. Referans bir servis odaklı sistemden, nesneye dahalı metrikleri sınıf ve sınıflar arası bağımlılıkları, nesneye dayalı modeldedeki özelliklerini öğrenip, otomatik modül bulmada bunları daha etkin kullanan bir modül bulma modeli geliştirilmiştir. Geliştirilen bu model, seçilen nitelik ve metrik kümesinin değiştirilmesi, kullanılanılan makine öğrenmesi tabanlı sınıflandıcı algoritmalarının yazılım sistemine özel seçilmesi ile genişletmeye ve uyarlanmaya uygundur. Çalışma kapsamında önerilen modül bulma yaklaşımı bir analiz aracıyla beraber gerçeklenmiş, çok sayıda açık kaynaklı ve endüstriyel yazılım sistemi üzerinde başarımı diğer yöntemlerle karşılaştırılmıştır. Bu yöntemin, diğer yöntemlere kıyasla daha doğru ve hızlı sonuçlar üretmesinin yanında, daha kararlı ve düzgün modül boyutu dağılımına sahip sonuçlar ürettiği deneylerle gösterilmiştir. Böyle bir otomatik yazılım modül bulma yöntemi, yazılım bakım faaliyetlerinde oldukça büyük payı olan, yazılım mimarisinin çıkarılması, yazılım sisteminin anlaşılması ve var olan sistemlerin servis odaklı mimariye geçirilmesinde etkin rol oynayacaktır.

8.2 Gelecek Çalışmalar

Önerilen modelin uyarlanabilirliği ve genişletilebilirliği daha çok sayıda yazılım sisteminin kaynak kodunun ve tasarım mimarisinin üzerinde çeşitli yeni senaryolarının denemesine izin vermektedir. Tabi ki bu amaç için büyük yazılımlarda elde etmesi oldukça zor olan işlenmiş uygun veri kümesinin sağlanabilmesi şarttır. Yapılabilecek gelecek çalışmaların başında, öğrenme kümesinin modül bulma

amacıyla farklı şekillerde seçilmesi gelmektedir. Örneğin, bir yazılım ekibi tarafından geliştirilmiş yazılım projesi ile önerilen model eğitilerek, bu yazılım ekibinin geliştirdiği farklı yazılım projelerinde modül bulma yaklaşımı denenebilir. Benzer şekilde önerilen model, yazılım sisteminin önceki sürümlerinden öğrenilen bilgilerle eğitilerek, aynı yazılımın ilerleyen sürümleri üzerinde önerilen yöntem uygulanarak ilginç modül bulma sonuçları elde edinilebileceği düşünülmektedir. Böyle bir alt yapı yardımıyla yazılım sürüm konfigürasyon sistemleri otomatik izlenerek, yazılım mimarisindeki sıradışı ani değişimler ve sürümler geçişlerinde ortaya çıkacak test maliyetleri de saptanabilecektir. Model üzerinde gelecekte yapılabilecek diğer çalışmalar ise, yazılımın uygulama alanına özel olarak seçilmiş farklı yeni kümeleme ve sınıflandırma algoritmalarını farklı nitelik kümesi üzerinde uygulamak olabilir. Eğer modülü çıkarılan yazılımların kaliteleri hakkında da yeterli bilgi almak mümkün olursa, otomatik modül bulma başarımı ile yazılımın kalitesi arasındaki ilişki de ileride yapılacak önemli çalışmalar arasında yer almaktadır. Literatürde henüz başarılamamış yeni bir araştırma konusu da, örtüşen servisleri belirleyen yöntemlerin geliştirilmesi ve bu yöntemleri karşılaştıramada kullanılabilecek metriklerin geliştirilmesidir. Yeniden kullanılabilir servislerin belirlenmesi ile bu servislerin yazılım üretim maliyetlerine etkisini ölçebilecek modeller geliştirmek de ileri de bu alanda yapılabilecek önemli çalışmalar arasında yer almaktadır.

KAYNAKLAR

[1] **Kazman, R.** (2008). *An Introduction to Software Architecture* (ders notu). Alındığı tarih: 08.04.2008, adres: http://www.cgl.uwaterloo.ca/~rnkazman/746.html.

[2] **Tassey, G.** (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-3*. Prepared by RTI for the National Institute of Standards and Technology (NIST).

[3] **The Standish Group.** (2013). *CHAOS Manifesto 2013: Think Big, Act Small* (Rapor No: TSG: 2013/1). CHAOS report on the resolution of IT projects. Boston, Massachusetts: The Standish Group International, Incorporated.

[4] **National Institute of Standards and Technology.** (2002). "The economic impacts of inadequate infrastructure for software testing", May 2002. US Dept of Commerce.

[5] **Brady, J.** (1998). Automated Y2K Testing Special Focus on Year 2000. *Enterprise Systems Journal*, v13(6), 92-95.

[6] **Peeger, S. ve Atlee, J.** (2006). *Software engineering - Theory and practice, (3. ed.).* Ellis Horwood.

[7] **Corbi, T. A.** (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, *28*(2), 294–306.

[8] **Lange, C. F., Chaudron, M. R. ve Muskens, J.** (2006). In practice: UML software architecture and design description. *IEEE Software*, *23*(2), 40–46.

[9] **Zimmermann, O., Krogdahl, P. ve Gee, C.** (2004). Elements of service-oriented analysis and design. *IBM developerworks*.

[10] **Hunt, A. ve Thomas, D.** (2002). Software archaeology. *IEEE Software*, *19*(2), 20–22.

[11] **Fowler, M.** (2002). Public versus published interfaces. *IEEE Software*, *19*(2), 18–19.

[12] **Perepletchikov, M., Ryan, C., Frampton, K. ve Schmidt, H.** (2008). Formalising Service-Oriented Design. *Journal of Software (1796217X)*, *3*(2).

[13] **Erdemir, U., Tekin, U. ve Buzluca, F.** (2011). Object Oriented Software Clustering Based on Community Structure, *18th Asia Pacific Software Engineering Conference (APSEC 2011)*, Ho Chi Minh City, Vietnam: IEEE Computer Society, December 5-8.

[14] **Newman, M. E. ve Girvan, M.** (2004). Finding and evaluating community structure in networks. *Physical Review E*, *69*(2), 026113.

[15] **Wiggerts, T. A.** (1997). Using clustering algorithms in legacy systems remodularization, *Proceedings of the Fourth Working Conference on Reverse Engineering*, (pp. 33–43). Amsterdam, the Netherlands : IEEE Computer Society, October 6–8.

[16] **Maqbool, O. ve Babri, H. A.** (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, *33*(11), 759–780.

[17] **Anquetil, N. ve Lethbridge, T. C.** (1999). Experiments with clustering as a software remodularization method, *Proceedings of the Sixth Working Conference on Reverse Engineering*, (pp. 235–255). Atlanta, Georgia, USA : IEEE Computer Society, October 6–8.

[18] **Tzerpos**, **V. ve Holt, R. C.** (2000). On the stability of software clustering algorithms, *Proceedings. IEEE 8th International Workshop on Program Comprehension (IWPC)*, (pp. 211–218). Limerick, Ireland : IEEE Computer Society, June 10–11.

[19] **Wu, J., Hassan, A. E. ve Holt, R. C.** (2005). Comparison of clustering algorithms in the context of software evolution, *Proceedings of the 21st IEEE International Conference on Software Maintenance*, (pp. 525–535). Budapest, Hungary : IEEE Computer Society, September 25-30.

[20] **Tzerpos, V. ve Holt, R. C.** (2000). ACDC: An Algorithm for Comprehension-Driven Clustering, In Proceeding of the 7th Working Conference on Reverse Engineering (pp. 258–267). Brisbane, Queensland, Australia : IEEE Computer Society, November 23-25.

[21] **Mancoridis, S., Mitchell, B. S., Chen, Y. ve Gansner, E. R**. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures, *IEEE International Conference on Software Maintenance*, (pp. 50–59). Oxford, England, UK : IEEE Computer Society. August 30 - September 3.

[22] **Tzerpos, V. ve Holt, R. C. (1999).** MoJo: A distance metric for software clusterings, *Proceedings of the Sixth IEEE Working Conference on Reverse Engineering*, (pp. 187–193). Atlanta, Georgia, USA : IEEE Computer Society, October 6-8.

[23] **Bittencourt, R. A. ve Guerrero, D. D. S.** (2009). Comparison of graph clustering algorithms for recovering software architecture module views, *13th IEE European Conference on Software Maintenance and Reengineering*, (pp. 251–254). Fraunhofer IESE, Kaiserslautern, Germany : IEEE Computer Society, March 24-27.

[24] **Doval, D., Mancoridis, S. ve Mitchell, B. S.** (1999). Automatic clustering of software systems using a genetic algorithm, *IEEE Proceedings of the Software Technology and Engineering Practice (STEP'99)*, (pp. 73–81). Pittsburgh, Pennsylvania, USA : IEEE Computer Society, August 30–September 2.

[25] **Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. ve Gansner, E. R.** (1998). Using automatic clustering to produce high-level system organizations of source code, *IEEE International Conference on*

Program Comprehension, (pp. 45–52). Ischia, Italy : IEEE Computer Society, June 24–26.

[26] **Mahdavi, K., Harman, M. ve Hierons, R. M.** (2003). A multiple hill climbing approach to software module clustering, *IEEE International Conference on the Software Maintenance (ICSM)*, (pp.315–324).Amsterdam, The Netherlands : IEEE Computer Society, September 22–26.

[27] **Seng, O., Bauer, M., Biehl, M. ve Pache, G.** (2005). Search-based improvement of subsystem decompositions, *the Proceedings of the conference on Genetic and evolutionary computation*, (pp. 1045–1051). Washington, D.C , USA : ACM, June 25–29.

[28] **Praditwong, K., Harman, M. ve Yao, X.** (2011). Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, *37*(2), 264–282.

[29] **Xio, C. ve Tzerpos, V.** (2005). Software Clustering Based on Dynamic Dependencies, *the Ninth European Conference Software Maintenance and Reengineering*, (pp. 124-133), Manchester, United Kingdom : IEEE Computer Society, March 21–23.

[30] **Patel, C., Hamou-Lhadj, A. ve Rilling, J.** (2009). Software clustering using dynamic analysis and static dependencies, *the 13th IEEE European Conference on Software Maintenance and Reengineering*, (pp. 27 36). Kaiserslautern, Germany : IEEE Computer Society, March 24-27.

[31] **Andreopoulos, B., An, A., Tzerpos, V. ve Wang, X.** (2005). Multiple layer clustering of large software systems. *IEEE the 12th Working Conference on Reverse Engineering*, (pp. 33–43).

[32] **Andritsosand, P. ve Tzerpos, V.** (2005). Information-theoretic software clustering. *IEEE Transactions on Software Engineering*, *31*(2), 150–165.

[33] **Anquetil, N. ve Lethbridge, T. C.** (1999). Recovering software architecture from the names of source files. *Journal of Software Maintenance*, *11*(3), 201–221.

[34] **Scanniello, G., D'Amico, A., D'Amico, C. ve D'Amico, T.** (2010). Using the kleinberg algorithm and vector space model for software system clustering, *IEEE 18th International Conference on Program Comprehension (ICPC)*, (pp. 180–189). Braga, Portuguese : IEEE Computer Society, June 30–July 2.

[35] **Kuhn, A., Ducasse, S. ve Gírba, T.** (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, *49*(3), 230–243.

[36] **Corazza, A., Di Martino, S. ve Scanniello, G.** (2010). A probabilistic based approach towards software system clustering, *IEEE the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, (pp. 88–96). Madrid, Spain : IEEE Computer Society, March 15–18.

[37] **Manning, C. D., Raghavan, P. ve Schütze, H.** (2008). *Introduction to information retrieval* (Vol. 1). Cambridge University Press Cambridge.

[38] **Corazza, A., Di Martino, S., Maggio, V. ve Scanniello, G.** (2011). Investigating the use of lexical information for software system clustering, *IEEE the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, (pp. 35–44). Oldenburg, Germany : IEEE Computer Society, March 1–4.

[39] **Risi, M., Scanniello, G. ve Tortora, G.** (2010). Architecture recovery using latent semantic indexing and k-means: An empirical evaluation, *the 8th IEEE International Conference Software Engineering and Formal Methods (SEFM)*, (pp. 103–112). Pisa, Italy : IEEE Computer Society, September 13–18.

[40] **Hartigan, J. A. ve Wong, M. A.** (1979). Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 100–108.

[41] **Risi, M., Scanniello, G. ve Tortora, G.** (2012). Using fold-in and fold-out in the architecture recovery of software systems. *Formal Aspects of Computing*, *24*(3), 307–330.

[42] **Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.** (2013). Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, *18*(5), 901–932.

[43] **Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D. ve Lucia, A. de.** (2014). Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(1), 4.

[44] **Santos, G., Valente, M. T. ve Anquetil, N.** (2014). Remodularization Analysis Using Semantic Clustering. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, Antwerp, Belgium: IEEE Computer Society February 3-6**.**

[45] **Arsanjani, A.** (2004). Service-oriented modeling and architecture. *IBM developer works*, 1-15.

[46] **Perepletchikov, M. ve Ryan, C.** (2011). A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering*, *37*(4), 449–465.

[47] **Perepletchikov, M., Ryan, C. ve Tari, Z.** (2010). The impact of service cohesion on the analyzability of service-oriented software. *Services Computing, IEEE Transactions on*, *3*(2), 89–103.

[48] **Sarkar, S., Kak, Rama, A. C.** (2008). Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, *34*(5), 700–720.

[49] **Url-1 <** http://www.omg.org/spec/UML>, alındığı tarih: 16.05.2014.

[50] **Url-2 <** http://www.omg.org>, alındığı tarih: 16.05.2014.

[51] **Erdemir, U., Tekin, U. ve Buzluca, F.** (2011). E-Quality: A graph based object oriented software quality visualization tool, *the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT*), (pp. 1–8). Williamsburg, Virginia. USA : IEEE Computer Society, September 29-30.

[52] **Erdemir, U., Tekin, U. ve Buzluca, F.** (2008). Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi. *Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu*. İstanbul, Türkiye : Ekim 9-10.

[53] **Ishikawa, K.** (1985). *What is total quality control?: the Japanese way*. Prentice-Hall Englewood Cliffs, NJ.

[54] **Booch, G.** (2006). *Object Oriented Analysis & Design with Application*. Pearson Education India.

[55] **Bunge, M.** (1977). *Treatise on basic philosophy: Volume 3: Ontology 1: The furniture of the world*. Reidel, Boston.

[56] **Chidamber, S. R. ve Kemerer, C. F.** (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

[57] **Buzluca, F.** (2010). *Yazılım Modelleme ve Tasarımı* (Ders notu), Alındığı tarih: 17.08.2010, adres: http://www.buzluca.com.

[58] **Tempero, E.** (2009). *Software Measurement: Object-oriented Coupling* (Ders notu). Alındığı tarih: 09.03.2009, adres: http://www.cs.auckland.ac.nz/compsci702s1c/lectures/ewan/cs702-notes-lec07-oo-coupling.pdf

[59] **Oman, P. ve Hagemeister, J.** (1994). Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, *24*(3), 251–266.

[60] **Dagpinar, M. ve Jahnke, J. H.** (2003). Predicting maintainability with object-oriented metrics-an empirical comparison, *Proceedings of the tenth IEEE Working Conference on Reverse Engineering*, (pp. 155–164). Victoria, Canada : IEEE Computer Society, November 13-16.

[61] **Bandi, R. K., Vaishnavi, V. K. ve Turk, D. E.** (2003). Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, *29*(1), 77–87.

[62] **Marinescu, R.** (2001). Detecting design flaws via metrics in object-oriented systems, *the 39th IEEE International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, (pp. 173–182). Santa Barbara, California. USA : IEEE Computer Society, July 9-August 3.

[63] **Lague, B., Proulx, D., Mayrand, J., Merlo, E. M. ve Hudepohl, J.** (1997). Assessing the benefits of incorporating function clone detection in a development process, *Proceedings of IEEE International Conference on Software Maintenance,* (pp. 314–321). Bari, Italy : IEEE Computer Society, October 1-3.

[64] **Kontogiannis, K.** (1997). Evaluation experiments on the detection of programming patterns using software metrics, *Proceedings of the*

Fourth IEEE Working Conference on Reverse Engineering, (pp. 44–54). Amsterdam, The Netherlands : IEEE Computer Society, October 6-8.

[65] **Basili, V. R., Briand, L. C. ve Melo, W. L.** (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, *22*(10), 751–761.

[66] **Briand, L. C., Wüst, J., Daly, J. W. ve Victor Porter, D. (2000).** Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software*, *51*(3), 245–273.

[67] **Gyimothy, T., Ferenc, R. ve Siket, I.** (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897–910.

[68] **Subramanyam, R. ve Krishnan, M. S.** (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, *29*(4), 297–310.

[69] **Olague, H. M., Etzkorn, L. H., Gholston, S. ve Quattlebaum, S.** (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering*, *33*(6), 402–419.

[70] **McCabe, T. J.** (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4), 308–320.

[71] **Chidamber, S. R. ve Kemerer, C. F.** (1991). *Towards a metrics suite for object oriented design* (Vol. 26). ACM..

[72] **Li, W., S. Henry, D. Kafura, ve R. Schulman.** (1995). Measuring object-oriented design. *Journal of Object-Oriented Programming*, *8*(4), 48–55.

[73] **Hitz, M. ve Montazeri, B.** (1996). Chidamber and Kemerer's metrics suite: a measurement theory perspective. *IEEE Transactions on Software Engineering*, *22*(4), 267–271.

[74] **Brito e Abreu, F., Pereira, G. ve Sousa, P.** (2000). A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems, *Proceedings of the Fourth IEEE European conference on Software Maintenance and Reengineering*, (pp. 13–22). Zurich, Switzerland : IEEE Computer Society, February 29-March 3.

[75] **Bansiya, J. ve Davis, C. G. (2002).** A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17.

[76] **Bansiya, J., Etzkorn, L., Davis, C. ve Li, W.** (1999). A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, *11*(8), 47–52.

[77] **Counsell, S., Mendes, E., Swift, S. ve Tucker, A.** (2002). *Evaluation of an object-oriented cohesion metric through Hamming distances*. Tech. Rep. BBKCS-02-10, Birkbeck College, University of London, UK.

[78] **Counsell, S., Swift, S. ve Crampton, J.** (2006). The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *15*(2), 123–149.

[79] **Fowler, M. ve Beck, K.** (2000) *Bad Smells in Code,Refactoring: Improving the Design of Existing Code*, (pp. 75-88). Addison-Wesley, .

[80] **Salehie, M., Li, S. ve Tahvildari, L.** (2006). A metric-based heuristic framework to detect object-oriented design flaws, *the 14th IEEE International Conference on Program Comprehension*, (pp. 159–168). Athens, Greece : IEEE Computer Society, June 14-16.

[81] **Fowler, M.** (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[82] **Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fikdhl-King, I., ve Angel, S.** (1977). *A Pattern Language: Towns; Building, Constructions*. Oxford University Press.

[83] **Beck, K. ve Cunningham, W.** (1987). Using Pattern Languages for Object-Oriented Programs. (Rapor No. CR-87-43). Tektronix Technical.

[84] **Beck, K.** (1994). Patterns and software development. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, *19*(2), 18–23.

[85] **Riel, A. J.** (1996). *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc..

[86] **Gamma, E., Helm, R., Johnson, R., ve Vlissides, J.** (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

[87] **Girvan, M. ve Newman, M. E.** (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, *99*(12), 7821–7826.

[88] **Coleman, J. S.** (1964). Introduction to mathematical sociology. *London Free Press Glencoe.*

[89] **Freeman, L. C.** (2004). *The development of social network analysis: A study in the sociology of science* (Vol. 1). Empirical Press Vancouver.

[90] **Kottak, C. P.** (2011). *Cultural anthropology: Appreciating cultural diversity*. McGraw-Hill.

[91] **Moody, J. ve White, D. R.** (2003). Structural cohesion and embeddedness: A hierarchical concept of social groups. *American Sociological Review*, 103–127.

[92] **Chen, J. ve Yuan, B.** (2006). Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, *22*(18), 2283–2290.

[93] **Rives, A. W. ve Galitski, T.** (2003). Modular organization of cellular networks. *Proceedings of the National Academy of Sciences*, *100*(3), 1128–1133.

[94] **Spirin, V. ve Mirny, L. A.** (2003). Protein complexes and functional modules in molecular networks. *Proceedings of the National Academy of Sciences*, *100*(21), 12123–12128.

[95] **Flake, G. W., Lawrence, S., Giles, C. L. ve Coetzee, F. M.** (2002). Self-organization and identification of web communities. *Computer*, *35*(3), 66–70.

[96] **Burt, R. S.** (1976). Positions in networks. *Social Forces*, *55*(1), 93–122.

[97] **Granovetter, M**. (1973). The strength of weak ties. *American Journal of Sociology*, *78*(6), l.

[98] **Guimera, R. ve Amaral, L. A. N.** (2005). Functional cartography of complex metabolic networks. *Nature*, *433*(7028), 895–900.

[99] **Fortunato, S. ve Castellano, C.** (2007). Community Structure in Graphs. *arXiv:0712.2716 [cond-Mat, Physics:physics]*. Retrieved from http://arxiv.org/abs/0712.2716.

[100] **Weiss, R. S. ve Jacobson, E.** (1955). A method for the analysis of the structure of complex organizations. *American Sociological Review*, 661–668.

[101] **Homans, G. C.** (2013). *The human group* (Vol. 7). Routledge.

[102] **Zachary, W.** (1977). An Information Flow Modelfor Conflict and Fission in Small Groups1. *Journal of Anthropological Research*, *33*(4), 452–473.

[103] **Brin, S. ve Page, L.** (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, *30*(1), 107–117.

[104] **Leicht, E. A. ve Newman, M. E.** (2008). Community structure in directed networks. *Physical Review Letters*, *100*(11), 118703.

[105] **Rosvall, M. ve Bergstrom, C. T.** (2008). Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, *105*(4), 1118–1123.

[106] **Fortunato, S.** (2010). Community detection in graphs. *Physics Reports*, *486*(3), 75–174.

[107] **Porter, M. A., Onnela, J.-P. ve Mucha, P. J.** (2009). Communities in networks. *Notices of the AMS*, *56*(9), 1082–1097.

[108] **Kernighan, B. W. ve Lin, S.** (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, *49*(2), 291–307.

[109] **Barnes, E. R.** (1982). An algorithm for partitioning the nodes of a graph. *SIAM Journal on Algebraic Discrete Methods*, *3*(4), 541–550

[110] **MacQueen, J.** (1967). Some methods for classification and analysis of multivariate observations, *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, (Vol. 1, p. 14). California, USA.

[111] **Anthonisse, J. M.** (1971). The rush in a directed graph. *Stichting Mathematisch Centrum. Mathematische Besliskunde*, (BN 9/71), 1–10.

[112] **Zhou, H.** (2003). Distance, dissimilarity index, and network community structure. *Physical Review E*, *67*(6), 061901.

[113] **Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z. ve Wagner, D.** (2006). *On modularity-np-completeness and beyond*. Citeseer.

[114] **Donetti, L. ve Munoz, M. A. (2004).** Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, *2004*(10), P10012.

[115] **Fortunato, S.** (2007). Quality functions in community detection (Vol. 6601, pp. 660108–660108–10). doi:10.1117/12.726703**.**

[116] **Karypis, G. ve Kumar, V.** (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, *20*(1), 359–392.

[117] **Url-3** <http://glaros.dtc.umn.edu/gkhome/views/cluto>, alındığı tarih: 16.05.2014.

[118] **Dongen, S. Van.** (2000). *Graph clustering by flow simulation* (Doktora tezi), Utrecht University.

[119] **Wu, F. ve Huberman, B. A. (2004).** Finding communities in linear time: a physics approach. *The European Physical Journal B-Condensed Matter and Complex Systems*, *38*(2), 331–338.

[120] **Dhillon, I. S., Guan, Y. ve Kulis, B. (**2007). Weighted graph cuts without eigenvectors a multilevel approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *29*(11), 1944–1957.

[121] **Dhillon, I., Guan, Y. ve Kulis, B.** (2005). A fast kernel-based multilevel algorithm for graph clustering, *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, (pp. 629–634). Chicago, IL, USA : ACM, August 21-24..

[122] **Url-4** <http://userweb.cs.utexas.edu/users/dml/Software/graclus.html>, alındığı tarih: 16.05.2014.

[123] **Valverde, S. ve Solé, R. V.** (2003). Hierarchical small worlds in software architecture. *arXiv Preprint Cond-mat/0307278*.

[124] **Pirzadeh, H., Alawneh, L. ve Hamou-Lhadj, A.** (2009). Quality of the source code for design and architecture recovery techniques: utilities are the problem, *the 9th IEEE International Conference on Quality Software*, (pp. 465–469). Jeju, Korea : IEEE Computer Society, August 24-25.

[125] **Wen, Z. ve Tzerpos, V.** (2005). Software clustering based on omnipresent object detection, *Proceedings of the 13th IEEE International Workshop on Program Comprehension*, (pp. 269–278). St. Louis, Missouri, USA : IEEE Computer Society, May 15-16.

[126] **Mitchell, B. S. ve Mancoridis, S.** (2006). On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, *32*(3), 193–208.

[127] **Abramowitz, M., Stegun, I.** (1972). *Stirling Numbers of the First Kind, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, 9th printing. New York: Dover, p. 824.

[128] **Tzerpos, V. ve Holt, R. C.** (1997). The orphan adoption problem in architecture maintenance, *Proceedings of the Fourth IEEE Working Conference on Reverse Engineering*, (pp. 76–82). Amsterdam, The Netherlands : IEEE Computer Society, October 6-8.

[129] **Nijenhuis, A. ve Wilf, H. S.** (1978). Combinatorial algorithms. *New York*.

[130] **Fowler, M.** (2004). Inversion of control containers and the dependency injection pattern.

[131] **Lorenz, M. ve Kidd, J.** (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc.

[132] **Tukey, J. W.** (1977). *Exploratory Data Analysis*, Addison-Wesley.

[133] **Peng, H., Long, F. ve Ding, C.** (2005). Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *27*(8), 1226-1238.

[134] **Quinlan, J. R.** (1993). *C4. 5: programs for machine learning* (Vol. 1). Morgan kaufmann.

[135] **Weka 3** Data Mining with Open Source Machine Learning Software in Java, http://www.cs.waikato.ac.nz/ml/weka>, alındığı tarih: 16.05.2014.

[136] **Drazin, S. ve Montag, M.** (2012). Decision tree analysis using WEKA. *Machine Learning-Project II, University of Miami*, 1–3. http://www.samdrazin.com/classes/een548/project2 report.pdf

[137] **Concas, G., Marchesi, M., Pinna, S. ve Serra, N.** (2007). Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, *33*(10), 687–708.

[138] **Newman, M.** (1999). *Small Worlds: The Structure of Social Networks* (Working Paper No. 99-12-080). Santa Fe Institute. Retrieved from http://ideas.repec.org/p/wop/safiwp/99-12-080.html

[139] **Newman, M. E.** (2004). Fast algorithm for detecting community structure in networks. *Physical Review E*, *69*(6), 066133

[140] **Good, B. H., de Montjoye, Y.-A. ve Clauset, A.** (2010). Performance of modularity maximization in practical contexts. *Physical Review E*, *81*(4), 046106.

[141] **Clauset, A., Newman, M. E. ve Moore, C.** (2004). Finding community structure in very large networks. *Physical Review E*, *70*(6), 066111

[142] **Url-5** <http://www.killerstartups.com/review/karniyarik-com-online-search-in-turkey/>, alındığı tarih: 16.05.2014.

[143] **Müller, H. A., Orgun, M. A., Tilley, S. R. ve Uhl, J. S.** (1993). A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, *5*(4), 181–204.

[144] **Url-6** <http://argouml.tigris.org>, alındığı tarih: 16.05.2014.

[145] **Url-7** <http://www.eclipse.org/gef>, alındığı tarih: 16.05.2014.

[146] **Url-8** <http://www.jfree.org/jfreechart>, alındığı tarih: 16.05.2014.

[147] **Url-9** <http://lucene.apache.org>, alındığı tarih: 16.05.2014.

[148] **Url-10** <http://lucene.apache.org/solr>, alındığı tarih: 16.05.2014.

[149] **Url-11** <http://tomcat.apache.org>, alındığı tarih: 16.05.2014.

[150] **Ovatman, T., Weigert, T. ve Buzluca, F.** (2011). Exploring implicit parallelism in class diagrams. *Journal of Systems and Software*, *84*(5), 821–834.

[151] **Younas, M., Chao, K.-M. ve Laing, C.** (2004). SODA: service oriented design activities, *Proceedings of the 8th IEEE International Conference on Computer Supported Cooperative Work in Design*, (Vol. 2, pp. 419–424). Xiamen, China : IEEE Computer Society, May 26-28.

ÖZGEÇMİŞ

Ad Soyad: Ural ERDEMİR

Doğum Yeri ve Tarihi: Sapanca / 1982

E-Posta: uralerdemir@yahoo.com

Lisans: İstanbul Teknik Üniversitesi, Bilgisayar Mühendisliği

Yüksek Lisans: İstanbul Teknik Üniversitesi, Bilgisayar Mühendisliği

Mesleki Deneyim ve Ödüller: Ural ERDEMİR, Araştırmacı unvanı ile 2004 yılında çalışmaya başladığı TÜBİTAK – BİLGEM – UEKAE'de halen Başuzman Araştırmacı unvanıyla çalışmaya devam etmektedir.

TEZDEN TÜRETİLEN YAYINLAR/SUNUMLAR

- **U. Erdemir,** F. Buzluca, "A Learning-Based Module Extraction Method for Object-Oriented Systems", Journal of Systems and Software, Elsevier, Vol. 97, Issue: 11, pp. 156 - 177, November 2014.
- **U. Erdemir,** U. Tekin and F. Buzluca, "Object Oriented Software Clustering Based on Community Structure", The 18th Asia-Pacific Software Engineering Conference, IEEE Computer Society (APSEC 2011), Vietnam, 2011.
- **U. Erdemir,** U. Tekin, F. Buzluca, "E-Quality: A graph based object oriented software quality visualization tool", 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis IEEE Computer Society (VISSOFT 2011), Williamsburg, VA, USA, 29-30 September, 2011.
- **U. Erdemir,** U. Tekin, F.Buzluca, "Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi" Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu (YKGS08), İstanbul, 2008.

DİĞER YAYINLARIN LİSTESİ

- U. Tekin, **U. Erdemir**, F. Buzluca, "Mining Object-Oriented Design Models for Detecting Identical Design Structures", Sixth International Workshop on Software Clones (IWSC 2012), Zurich, Switzerland, June 4, pp. 43-49, 2012.
- U. Tekin, **U. Erdemir,** F.Buzluca, "Yazılım Klonları ve Klon Belirleme Yöntemleri", IV. Ulusal Yazılım Mühendisliği Sempozyumu (UYMS'09), 8-10 Ekim, İstanbul, 2009.
- **U. Erdemir**, U. Tekin, F. M. Atay, and A. Ercil. "Two-dimensional object recognition using power spectrum" In Proc. SIU 2002: The 10th Congress on Signal Processing and Applications (E. Panayirci, ed.), Vol. 2, 1044-1049, Pamukkale, Turkey, 2002.