T.C.

TRAKYA ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

YAZILIMIN EVRİMLEŞME SÜRECİNDE TASARIM ÖRÜNTÜLERİNİN YAZILIM KALİTESİ ÜZERİNDEKİ ETKİLERİNİN İNCELENMESİ

METİN İLHAN AKALIN

YÜKSEK LİSANS TEZİ

BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI

Tez Danışmanı: Yrd.Doç.Dr. Rembiye KANDEMİR

EDİRNE-2014

T.Ü. Fen Bilimleri Enstitüsü onayı

Prof. Dr. Mustafa ÖZCAN
Fen Bilimleri Enstitüsü Müdürü

Bu tezin Yüksek Lisans tezi olarak gerekli şartları sağladığını onaylarım.

Prof. Dr. Yılmaz KILIÇASLAN
Anabilim Dalı Başkanı

Bu tez tarafımca okunmuş, kapsamı ve niteliği açısından bir Yüksek Lisans tezi olarak kabul edilmiştir.

Yrd. Doç. Dr. Rembiye KANDEMİR
Tez Danışmanı

Bu tez, tarafımızca okunmuş, kapsam ve niteliği açısından Bilgisayar Mühendisliği Anabilim Dalında bir Yüksek Lisans tezi olarak oy birliği ile kabul edilmiştir.

Jüri Üyeleri İmza

Yrd. Doç. Dr. Rembiye KANDEMİR

Yrd. Doç. Dr. Özlem AYDIN

Yrd. Doç. Dr. Hilmi KUŞÇU

Tarih: 22/07/2014

T.Ü. FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİYÜKSEK LİSANS PROGRAMI
DOĞRULUK BEYANI

İlgili tezin akademik ve etik kurallara uygun olarak yazıldığını ve kullanılan tüm literatür bilgilerinin kaynak gösterilerek ilgili tezde yer aldığını beyan ederim.

22/07/2014

Metin İlhan Akalın

Yüksek Lisans Tezi

Yazılımın Evrimleşme Sürecinde Tasarım Örüntülerinin Yazılım Kalitesi Üzerindeki Etkilerinin İncelenmesi

T.Ü. Fen Bilimleri Enstitüsü

Bilgisayar Mühendisliği Anabilim Dalı

ÖZET

Bu araştırmanın genel amacı tasarım örüntülerinin yazılım kalitesi üzerindeki etkilerinin, yazılımın evrimleşme süreci içerisinde incelenmesidir. Evrimleşen yazılım, tasarım örüntüleri ve yazılım sistemlerinin kalite olguları üzerinde, model, metrik ve karakteristikler göz önünde bulundurularak yapısal ve işlevsel kapsamda çalışmalar geçekleştirilmiştir.

Araştırma kapsamında kullanılan ve incelenen yazılımlar, açık kaynak kodlu projelerdir ve kamu kullanımına izin veren lisanslara sahip yazılımlardır. Seçilen bu yazılımların, farklı tarihlerde piyasaya çıkan farklı sürümleri, kendi evrimleşme süreçleri içerisinde incelenmiş, yazılımların ihtiva ettikleri tasarım örüntüleri tespit edilmiş ve ortaya çıkarılmıştır. Yazılımların, belirlenen bir yazılım kalite olgusu çerçevesinde, model ve metrik incelemeleri yapılmış, her yazılımın farklı sürümlerine ait birer kalite endeksi hesaplanmıştır. Bu hesaplamalar sonrasında yazılımların farklı sürümlerinden elde edilen kalite endeksleri ile yine bu sürümlerin içerdikleri tasarım örüntüsü miktarları ile ilişkisi, birçok farklı istatistiksel yöntem yardımıylaaçığa çıkarılmıştır. Ve gerekli yorumlamalar yine bu yöntemler yoluyla gerçekleştirilmiştir.

Araştırma kapsamında, yazılım sistemlerindeki evrimsel sürecin, istenilen bir biçimde sürdürülebilmesi ve yazılım kalite standartlarına bağlı kalabilmesi amacı ile tasarım örüntülerinin kullanılmasının yanı sıra, evrimsel gereksinimlerde göz önünde bulundurulmuştur.Tasarım örüntülerinin bilinen sorunlara pratik çözümler sağlayarakverimli bir yazılım geliştirme sürecini desteklerler. Ancak yapılan

incelemeler bize göstermiştir kitasarım örüntülerinin, yazılımın kalite karakteristiğine tek başlarına yön verebilecek yeterliliğe sahip olduklarını söylemek mümkün değildir.

Yıl : 2014
Sayfa Sayısı : 92
Anahtar Kelimeler : Tasarım Örüntüleri, Yazılım Kalitesi, Yazılım Evrimi

Master's Thesis

An Examination for the Effects of Software Design Patterns

On Software Quality in Software Evolutionary Process

Trakya University Institute of Natural Sciences

Computer Engineering

ABSTRACT

This study aims to investigate the connection between design patterns and software quality metrics in software evolution. Evolving software, design patterns and software system quality concepts have studied within the scope of structurality and functionality by taking into consideration the models, metrics and characteristics.

The softwares that used in the study were selected among open source projects and general public licenced softwares. Those selected different software releases that released in the market in different dates were analysed and the containing design patterns have determined from their source codes. Softwares analysed within the frame of a defined software quality concept and quality indexes calculated from each releases of these softwares. After these calculations, with the help of several different statistical methods, the relationship has revealed among these calculated software quality indexes and the design patterns that softwares contains. And required interpretations has made via these methods.

For the purposes of sustaining the evolutional process as required and adhering to software quality standards, the evolutional necessity took in to consideration right along with the usage of the design patterns in the scope of this study. Design patterns supports an efficient software development process via providing practical solutions on common problems but the investigation shows us that it is not possible to say that the design patterns have sufficiency to dominate the software quality characteristic by themselves.

Year : 2014
Number of Pages : 92
Keywords : Design Patterns, Software Quality, Software Evolution

TEŞEKKÜR

Bu araştırmanın planlanması, deneysel çalışmaların yönlendirilmesi, sonuçların değerlendirilmesi ve yazımı aşamasında yapmış olduğu büyük katkılarından dolayı tez danışmanım Sn. Yrd. Doç. Dr. Rembiye KANDEMİR'e güveni, desteği, deneyim ve bilgisini paylaştığı için çok teşekkür ederim.

Yoğun iş tempolarına rağmen, bilgi ve tecrübelerini benimle paylaşan, çalışmalarımı pratik olarak destekleyen hocalarım Sn. Yrd. Doç. Dr. Özlem AYDIN'a ve Sn. Yrd. Doç. Dr. Hilmi KUŞÇU'ya destekleri için teşekkürlerimi sunarım.

Çalışmalarım süresince bana maddi ve manevi her türlü desteği veren ve zorlukları aşmamda yardımcı olan sevgili aileme yürekten teşekkür ederim.

İÇİNDEKİLER

KISALTMALAR LİSTESİ

A	Absrtaction
ANA	Average Nubmer of Ancestors
AST	Abstract Syntax Tree
CAM	Cohesion Among Methods
CIS	Class Interface Size
CISQ	Consortium of IT Software Quality
DAM	Data Access Metric
DCC	Direct Class Coupling
DIT	Depth of Inheritance Tree
DSC	Design Size in Classes
EC	Efferent Coupling
IEC	International Electrotechnical Commision
IEEE	Institute of Electric and Electronic Engineers
ISO	International Organization for Standardization
MFA	Meaasure of Functioanl Abstraction
MOA	Measure of Aggregation
NOC	Number of Classes
NOF	Number of Fields
NOH	Number of Hierarchies
NOM	Number of Methods
NOP	Number of Polymorphic Methods
OOPSLA	Object Oriented Programming Systems, Languages and Applications
QMOOD	Quality Model for Object Oriented Design
WMC	Weighted Methods per Class

ŞEKİLLER LİSTESİ

TABLOLAR LİSTESİ

BÖLÜM 1

GİRİŞ

Bilgisayarların yirminci yüzyılın ikinci yarısının başlarından itibaren ve elli yılı aşkın bir süredir hayatlarımızın içinde yer alması sonucunda, bugün birçoğumuz doğrudan veya dolaylı yollarla bilgisayar sistemleri ile ilişki halindeyiz ve günlük yaşantımızda neredeyse bilgisayarlara bağımlı hale gelmiş durumdayız. Birçok faaliyetin başarıyla gerçekleştirilebilmesi için, bilgisayarlara ve bilgisayarlar üzerinde çalıştırılacak yazılımlara gereksinim duyulmaktadır. Böylece bilgisayar teknolojilerine olan bağımlılığımız fazlalaşmakta ve yazılımların iş hayatındaki ehemmiyeti artmaktadır. İçinde bulunduğumuz yüzyılda yazılım sistemlerine duyulan gereksinim artarken, yazılım sistemleri de giderek çeşitlenmekte ve karmaşık bir hal almaktadırlar. Talepler ne denli karmaşık olursa olsun, her zaman yüksek kalite beklentisi mevcuttur. Yazılım geliştiricilerin bu yüksek kalite beklentisini karşılamak için daha verimli ve etkili geliştirme süreçleri ortaya koymaları gerekmektedir. Yazılım sistemlerinin geliştirme süreçlerinde ölçüm tekniklerinin kullanılması, sürecin belirlenen zamanda ve belirlenen bütçe dâhilinde tamamlanmasını ve dolayısıyla kaliteli bir yazılım ürününün üretilmesini sağlamaktadır.

Yazılım sektöründeki projelerin maliyetlerinin büyük çapta artmasıyla birlikte, yazılımın kalite olgusu, bugün üzerinde en çok çalışılan konulardan biri haline gelmiştir. Yazılım sektöründe kalite, kavram olarak herkesin farkını algılayabildiği fakat sonuç itibarı ile tam anlamıyla tanımlanamayan soyut ve öznel bir olgu olarak karşımıza çıkmaktadır. Bir disiplin çerçevesinde kalite kavramının tam olarak tanımlanabilmesi için gelişmiş ölçüm araçlarına ve sağlıklı karşılaştırmalar yapabilmek için tanımlanmış referans noktalarına ihtiyaç vardır.

Kalite olgusuna yazılım endüstrisi tarafından verilen önem ve üzerinde yapılan akademik çalışmaların sayısı gün geçtikçe fazlalaşmaktadır. Yazılım kalitesini,

müşterinin bakış açısından ve yazılım üreticisinin bakış açısından ele alarak iki farklı temel üzerinde incelemek mümkündür. Müşteri tarafından bakıldığında, genellikle satın alınan yazılımın kolay kullanılabilir olması, hatasız çalışması, tüm talepleri yerine getirmesi ve yeterli performansa sahip olması beklenir. Buna karşılık yazılım üreticileri tarafından bakıldığında ise geliştirme ve bakım maliyetlerinin düşük tutulması ve üretilen yazılımın bileşenlerinin gelecek projelerde de kullanılabilir olmasını beklenir. Yazılım sistemlerinde kalite olgusu çeşitli kategoriler halinde sınıflandırılabilir. ISO 9126, yazılım ürünlerinin kalitesi açıklayan ve kategorize eden uluslararası bir standarttır.

Bu çalışmada bahsedilen yazılım kalitesinin ise, uygulanabilir olan yapısal özelliklerinin ve ölçüm birimlerinin sınıflandırılması ve terminolojisi ISO 9126-3 ve onun devam niteliğindeki ISO 25000:2005 kalite modeli standartlarından türetilmiştir. Bu modellere dayanarak yapısal kalitenin karakteristikleri, Bilişim Teknolojileri Kalite Konsorsiyumu (CISQ: Consortium of IT Software Quality) tarafından açıkça ortaya konmuştur.

Kaliteli yazılımda en önemli unsur, yazılımın talep ve istekleri doğru ve eksiksiz bir biçimde karşılamasıdır. Ancak yazılım projelerinde gelen talep ve istekleri karşılamak adına hızlı ve kontrolsüz adımlar atmak yerine çözüm odaklı yazılım geliştiriyor olmak gerekir. Uzun vadede geliştirilen projelerde yazılımın anlık taleplere cevap verebiliyor olmasının yanında, sonradan talep edilebilecek değişikliklerin de öngörülebiliyor olması ve bu şeklide evrimleşecek yazılımın geliştiriliyor olması gerekmektedir. Bu amaç doğrultusunda, yazılım parçalarının tekrar kolayca kullanılabilir olmaları, kolayca genişleyebilir veya sistemden kolayca çıkarılabilir olmaları yani kısaca esnek olmaları beklenir. Yeni ihtiyaçların, evrimleşen yazılımın diğer kısımlarını en az biçimde etkileyerek yazılıma kolayca dâhil olmaları beklenir. İşte tasarım örüntülerinin, yazılımdaki bu prensipleri doğru bir şekilde uygulamamızda bize yardımcı olup olmadıkları bu çalışma kapsamında incelenecektir.

Tasarım örüntüleri, yazılımın tasarım aşamasında sıkça karşılaşılan genel sorunlara esnek, yeniden kullanılabilir, başarılı çözümler getiren bir takım hazır kalıplardır. Hazır olarak kodun içerisine konulup çalıştırılabilen, bitmiş tasarımlar değildir. Çeşitli durumlarda sorunların nasıl giderilebileceğini açıklayan, bunlara yol

gösteren açıklamalardır. Nesneye yönelimli programlamada, tasarım örüntüleri sınıf ve nesneler arasındaki ilişkilerin en iyi şekilde nasıl olmaları gerektiğini açıklayan yöntemlerdir. Algoritmalar, tasarım örüntüsü değildir. Çünkü bunlar hesaplama sorunlarına çözüm getirirler, oysaki tasarım örüntüleri yazılım tasarımı sorunlarıyla ilgilenir.

Nesne yönelimli programlamada sınıfların kendi içlerinde tutarlı, fakat diğer sınıflara asgari düzeyde bağımlı olmaları beklenir. Tasarım örüntüleri, nesne yönelimli programlamanın bu prensiplerini uygun bir şekilde yerine getirmemizi sağlarlar.

Bu çalışma içerisinde ele alınan konulardan bir tanesi de yazılım sistemlerinin geçirdikleri evrim süreçleridir. Yazılım evrimi, yazılımın ilk geliştirme süreci ve bunun akabinde çeşitli sebeplerden dolayı durmaksızın devam eden güncellemeler bütünü olarak adlandırılabilir. Yazılım geliştirme süreci sona erdikten sonra yapılan bu güncellemeler genel hatlarıyla yeni ihtiyaç ve istekler doğrultusunda yeni özellik ve yetenekleri sisteme dâhil etmek, farkedilen problemleri düzeltmek, değişmiş ya da değişen ortam şartlarıyla uyumluluğu devam ettirmek, yazılımın bakım yapılabilirliği ya da performans artışı konularında iyileştirmeler yapmak, yazılımdaki belirti göstermeyen hataları etkili sorunlara sebep olmalarından önce ortaya çıkarıp düzeltmektir.

Bu evrimsel gelişimin sağlıklı bir şekilde yürütülebilmesi, yani başka bir deyişle yazılım kalite standartlarına bağlılık çerçevesinde sürdürülebilmesi için, önümüze çıkabilecek sorunlar karşısında daha önceden benzer sorunları çözmek için geliştirilmiş ve işlerliği kanıtlanmış olan genel çözüm önerilerinin yani tasarım örüntülerinin göz önünde bulundurulması önem kazanmaktadır.

Yapılan literatür taraması kapsmaında tasarım örüntülerinin yaratımsal, yapısal ve davranışsal karakteristiklerinin anlaşılmasında Eric Gamma ve arkadaşlarının gerçekleştirdikleri, konunun öncüsü kabul edilen çalışmalarından faydalanılmıştır [2]. Ve tasarım örüntülerinin, Java, C++, C#, PHP, .NET ve Visual Basic .NET gibi farklı porgramlama dilleri ve uygulama çatıları üzerinde uygulandıklarında nasıl davranış sergiledikleri, çeşitli kitaplar üzerinden araştırılmıştır [1,3,4,5,6,7,9,10,11,12]. Bunun yanı sıra, tasarım örüntülerinin yazılımın evrimleşme süreci içerisindeki yazılım kalitesine olan artı ve eksi etkileri, zaman içerisinde uğradıkları bozulmalar ve evrim

sürecinin işleyiş mantığı, çeşitli makaleler ve tezler ışığında incelenmiştir [37, 38, 39, 40, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60]. Yazılım kalite olgusu kapsamında metrik ve modellerin analizinde Bansiya ve Davis ‘in 2002 yılında ortaya sürdüğü model temel alınmıştır [44]. Ve bu modelin yaptığımız araştırmaya uyarlanmasında Chawla ve Chhabra’nın 2013 yılında yayınladıkları çalışma etkili olmuştur [45].

BÖLÜM 2

TASARIM ÖRÜNTÜLERİ

Tasarım örüntüleri, başarılı tasarımların ve mimarilerin yeniden kullanılabilirliğini kolaylaştırır. Onaylanmış tekniklerin tasarım örüntüleri olarak kullanılması, onları yeni sistemlerin geliştiricileri için ulaşılabilir hale getirmektedir. Tasarım örüntüleri, sistemi yeniden kullanılabilir hale getiren tasarım alternatiflerinin seçilmesinde ve yeniden kullanılabilirlikten ödün vermeye sebep olabilecek alternatiflerden kaçınılmasına yardımcı olur. Ayrıca, sınıfların belirgin özelliklerini ve nesneler arası ilişkileri, bunların altında yatan nedenlerle birlikte ön plana çıkararak mevcut sistemin dökümantasyonunu ve bakımını kolaylaştırır. Basitçe söylemek gerekirse, tasarım örüntüleri tasarımcının hızlıca doğru tasarımı elde etmesini sağlar [2].

Tasarım örüntüleri, nesnelerin birbirlerinin veri modellerinin ve metodlarının birbirleriyle karışmadan nasıl iletişim halinde olduklarını tarif eder. Bu ayrımı sağlayabilmek, iyi bir nesne yönelimli programlamanın hedefidir.

Tasarım örüntülerinin, literatürde öne çıkan bazı tanımlamaları şöyledir:

- "Tasarım örüntüleri, sürekli karşılaşılan tasarım problemlerinin yinelenen çözümleridir." [Smalltalk Companion]
- "Tasarım örüntüleri, yazılım geliştirme dünyasındaki belirli görevlerin nasıl yerine getirileceğini tanımlayan kurallar kümesini oluşturur." (Pree, 1994)
- "Uygulama çatıları (framework) detaylı tasarım ve bunun uygulanması ile ilgilenirken, tasarım örüntüleri daha çok, yinelenen mimari tasarım konuları üzerine odaklanır." (Coplien & Schmidt, 1995)
- "Bir örüntü, belirli tasarım şartları altında ortaya çıkan yinelenen tasarım sorunlarını işaret eder ve bu sorunların çözümlerini temsil eder." (Buschmann, vd, 1996)

- “Örüntüler bir sınıf, sınıf örneği(instance) ya da sınıf bileşenleri seviyelerinin üzerinde bir soyutlama tanımlar ve belirtir.” (Gamma, d, 1993)

Tasarım örüntüleri, sıkça karşılaşılan proglamlama zorluklarına karşılık zarif, kolayca yeniden kullanılabilen ve bakım yapılabilir çözümler sunarak, nesneye yönelimli tasarımın ayrılmaz bir parçası haline gelmiştir [3].

Tasarım örüntüleri, temel olarak mevcut kodu iyileştiren tasarım araçlarıdır. Bunlar etkinliği arttıran araçlardır, fakat daha da önemlisi bir geliştiricinin genel tasarım yeteneklerinin gelişmesine, dolayısıyla proje kalitesinin artmasına ve daha geniş ölçekte beceriler kazanmasına izin verirler. Özelleşmiş ve sıkça rastlanan sorunların yanıtlarını görmemize izin verirler. Örüntülere aşina olan diğer geliştiriciler arasında çeviri görevi yapan alışıldık programlama modellerini tanımlarlar [1].

1970 lerin sonunda Christopher Alexander isimli bir mimar, örüntüler alanında bilinen ilk çalışmayı yaptı. Alexander ve iş arkadaşları, kalite tasarımının canlılık ve bütünlüğünü tanımlamak amacıyla, aynı sorunu çözmek için tasarlanan farklı yapılar üzerinde çalıştılar. Yüksek kaliteli tasarımlar arasındaki benzerlikleri tanımladılar. 1987 yılında Kent Beck ve Ward Cunningham, Alexander'ın yazılarından etkilenerek mimari örüntü fikirlerini yazılımın tasarımı ve geliştirilmesi için uyguladılar. Smaltalk kullanıcı arayüzleri için bir çalışma geliştirirken Alexander'ın fikirlerini kullandılar. Nesne Yönelimli Programlama Sistemleri, Diller ve Uygulamalar '87 konferansında(Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) ’87) yaptıkları çalışmanın sonuçlarıyla "Nesne Yönelimli Programlama için Örüntü Dillerinin Kullanımı" başlığı altında bir sunum gerçekleştirdiler. O günden sonra nesne yönelimli programlama camiasından birçok tanınmış isim tarafından, örüntülerle ilişkili birçok evrak ve sunum yayınlandı. 1994 yılında Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides, "Tasarım Örüntüleri: Yeniden Kullanılabilir Nesne Yönelimli Yazılımın Unsurları"(Design Patterns: Elements of Reusable Object-Oriented Software) adıyla yayınlanan kitapta örüntülerin faydalarını açıkladılar ve bu, tasarım örüntülerinin geniş çapta popülerliği ile sonuçlandı. Bu dört yazar birlikte "Dörtlü Çete"(Gang of Four) olarak anılırlar.

Tasarım örüntüleri, belgelenmiş en iyi yol ya da özel şartlar dâhilinde ortaya çıkan sorunları çözmek için birçok ortamda başarıyla uygulanabilen temel çözümlerdir.

Tasarım örüntüsü bir keşif değildir. Birçok yazılım sisteminin yapılandırılmasında tespit edilen ya da gözlemlenen problemlerin en iyi çözüm yolunun belgelenmiş ifadesidir [4].

Bir tasarım örüntüsü, yeniden kullanılabilir nesne yönelimli tasarımın yaratılması için faydalı olan alışılagelmiş tasarım yapısı görüşünü adlandırır, souyutlar ve tanımlar. Tasarım örüntüsü, bütünü oluşturan sınıfları ve örnekleri(instance), bunların rollerini ve işbirliklerini ve de sorumlulukların dağılımını tanımlar. Her bir tasarım örüntüsü ayrı bir nesne yönelimli tasarım problemi ya da sorunu üzerinde odaklanır.

Tasarım örüntüleri, nesne yönelimli tasarımlar olarak açıklanırlar.Bu sebeple Pascal, C, Ada gibi prosedürel programlama dilleri yerine Smalltalk, C++ gibi ana akım nesne yönelimli programlama dillerinde uygulanan pratik çözümler üzerinde temellendirilirler.

Örüntüler kullanım amaçlarına göre yaratımsal, yapısal ve davranışsal olarak sınıflandırılmıştır. Yaratımsal örüntüler, nesne yaratım süreçleri ile ilgilenirler. Yapısal örüntüler, sınıfların ya da nesnelerin yapılarıyla alakalıdırlar. Davranışsal örüntüler ise, sınıf ya da nesnelerin etkileşimleriyle ve sorumlulukların dağılımlarının karakterize edilmeleriyle ilgilenir[2].

2.1 Yaratımsal Tasarım Örüntüleri

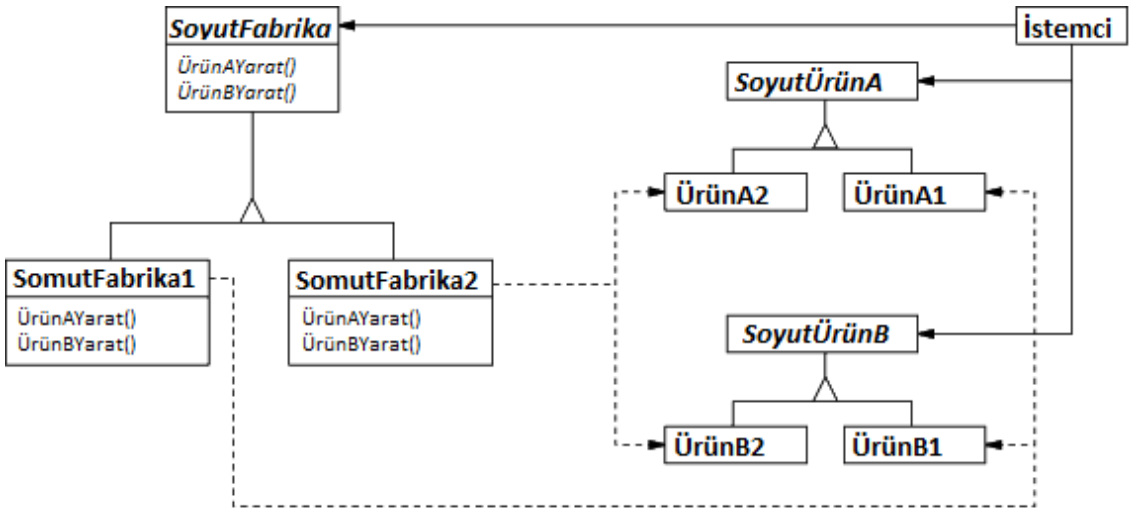
Yaratımsal tasarım örüntüleri ilk yaratım sürecini özetlerler. Bir sistemi, nesnelerinin yaratılmalarından, düzenlenmelerinden ve temsil edilmelerinden bağımsız kılar. Yaratımsal sınıf örüntüsü, yaratılan sınıfı çeşitlendirebilmek için kalıtımı kullanır. Yaratımsal örüntüler tasarımı gereksiz yere daraltmak yerine nasıl daha esnek olabileceğini gösterirler. Özellikle karmaşıklığın parçaları olan sınıfları değiştirmeyi kolaylaştırırlar [2].

2.1.1 Soyut Fabrika (Abstract Factory)

Bir soyut fabrika, birbirleriyle ilişkili ya da bağımlı nesnelerden aileler yaratırken, somut sınıflar ve bunların yaratılmasıyla alakalı tanımlamalar yapmadan tutarlı bir arayüz sağlar. İstemci, fabrikadan edinilen somut sınıf detaylarından ayrıştırılır [5].

Bu örüntü, ürün tanımlarıyla sınıf isimlerini istemciden soyutlar ve böylece bu ürün ve sınıflara ulaşmanın tek yolu bir fabrikadır.Bu sebepten istemci yapısı bozulmadan, ürün aileleri değiştirilebilir ve güncellenebilir [6].

Soyut Fabrika tasarım örüntüsünün yapısı Şekil 2.1'de gösterildiği gibidir.



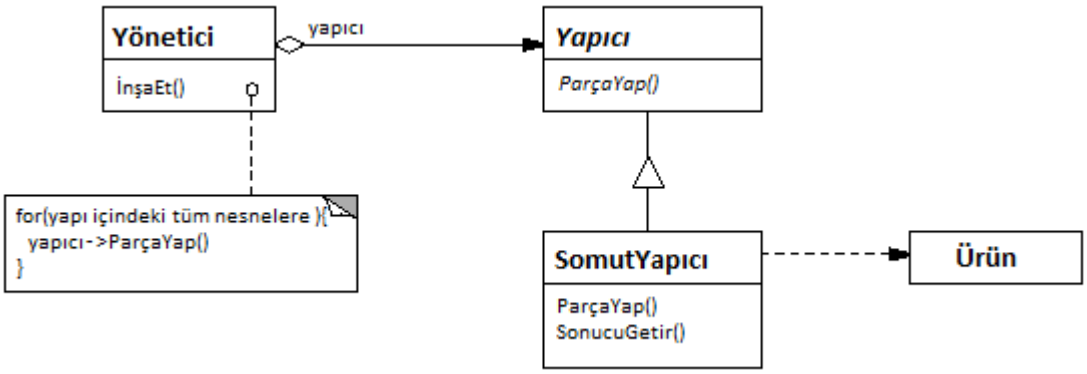
Şekil 2.1 Soyut Fabrika örüntüsünün yapısı.

2.1.2 Yapıcı (Builder)

Yapıcı örüntü, nesnelerin karmaşık yaratım sürecini ortadan kaldırmayı amaçlar. Yapıcı örüntünün kullanılması sadece en iyi çözüm yolu olmakla kalmaz,bir nesnenin yapıcı ve biçimlendirici metodlarında değişimler olduğu zaman, kod parçaları üzerinde ardı ardına değişiklikler yapılması ihtimalini de azaltır [7].

Yapıcı örüntü, karmaşık nesnelerin yaratılmasıyla temsilini birbirinden ayırarak aynı yaratım sürecinde farklı nesnelerin yaratılabilmesini sağlar. Ayrıca, istemci nesnesinin sadece tip ve içerik tanımlayarak karmaşık nesneler yaratmasına izin verir. İstemci, nesne yaratımının detaylarından sakınılmıştır [8].

Yapıcı tasarım örüntüsünün yapısı Şekil 2.2'de gösterildiği gibidir.



Şekil 2.2 Yapıcı tasarım örüntüsünün yapısı

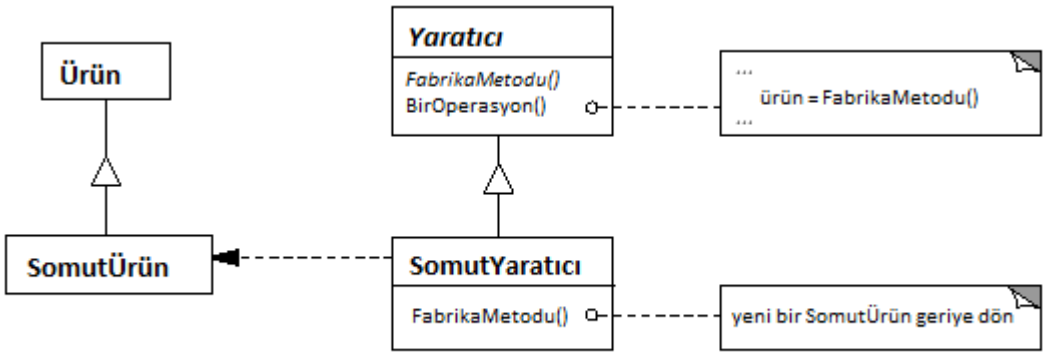
2.1.3. Fabrika (Factory)

Bu örüntü, Soyut Fabrika örüntüsüne "Somut Fabrika" olarak hizmet eder ve nesne yaratılması için bir arayüz tanımlar. Fabrika metodu kendisi bir sınıf yaratmaz fakat nesnenin yaratımını alt sınıflarına devreder. Bu örüntü Sanal Yaratıcı(Virtual Constructor) örüntü olarak blinir [9].

Birçok alanda, çeşitlilik arzeden nesnelerin yaratılmasında ve özel amaçlı alt sınıf kümelerini yerelleştirilmesinde, esneklik bir gerekliliktir [10].

Gücü ve esnekliği Fabrika ve aslında Soyut Fabrika yöntemleri yardımıyla kazanmak önem arzetmektedir [11].

Fabrika tasarım örüntüsünün yapısı Şekil 2.3‘de gösterildiği gibidir.



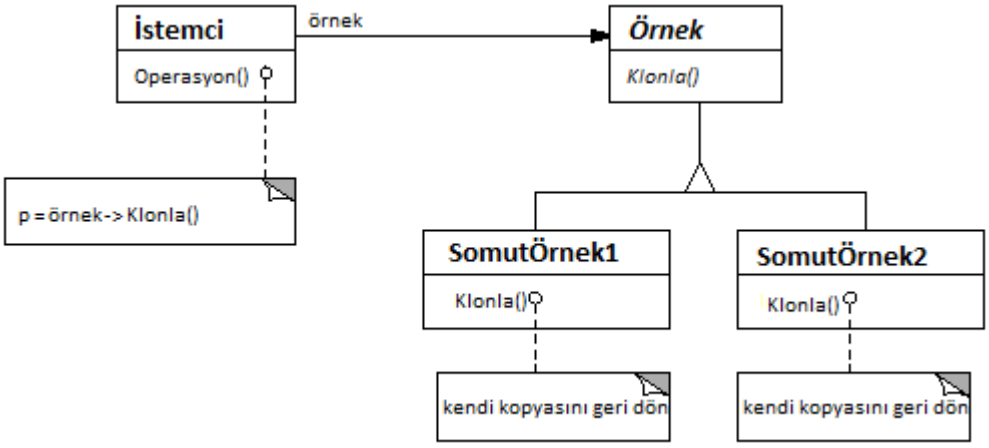
Şekil 2.3 Fabrika tasarım örüntüsünün yapısı.

2.1.4. Örnek (Prototype)

Örnek örüntü tamamen sanal olan bir kopya metod tanımlar ve türetilen somut sınıfların, bir işleyiş uygulayarak kendisini kopyalayan bu kopya metodu ezmesine izin verir [12].

Örnek örüntü faydalıdır. Çünkü sistemlerin yapıcısı tarafından tanımlanmış bir başlangıç durumuna bağımlı olmayıp hâlihazırda anlamlı değişken değerlerine sahip olan kullanılabilir nesne kopyaları üretmelerine izin verir [13].

Örnek tasarım örüntüsünün yapısı Şekil 2.4'de gösterildiği gibidir.



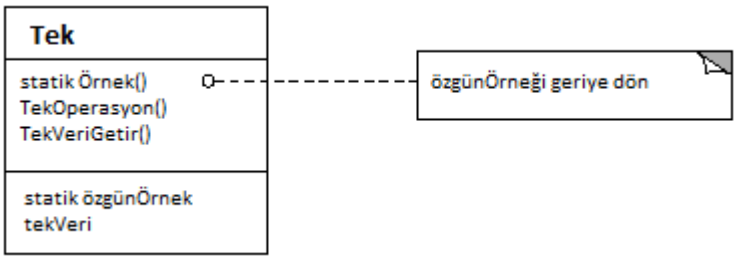
Şekil 2.4 Örnek tasarım örüntüsünün yapısı.

2.1.5. Tek (Singleton)

Tek örüntü, bir sınıfın yalnızca bir tane örneğinin var olduğu garantilemek ve bu örneğe evrensel bir erişim noktası sağlamak durumundadır [12].

Bazı durumlarda istenen, bir sınıfın bütün sistem tarafından erişilebilen yalnızca bir adet örneğinin olmasıdır [9].

Tek tasarım örüntüsünün yapısı Şekil 2.5'de gösterildiği gibidir.



Şekil 2.5 Tek tasarım örüntüsünün yapısı.

2.2. Yapısal Tasarım Örüntüleri

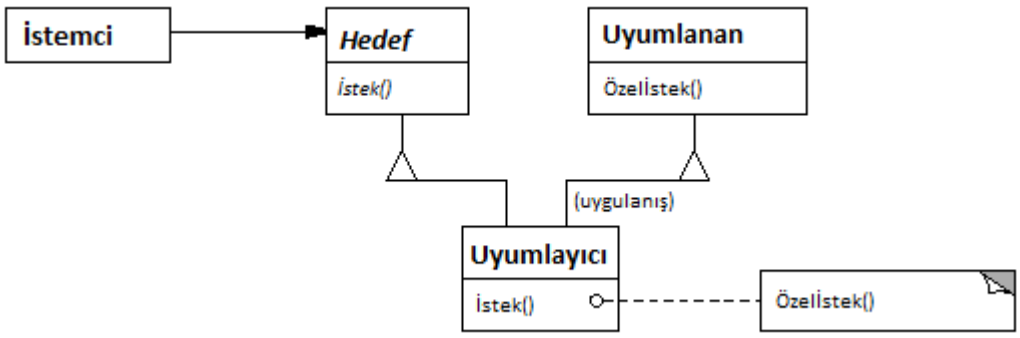
Yapısal örüntüler sınıfların ve nesnelerin daha büyük yapıları şekillendirmek için nasıl yapılandıklarıyla ilgilenir. Yapısal sınıf örüntüleri, arayüzleri ve bunların uygulanışların düzenleyebilmek için kalıtımı kullanır. Basitçe örneklemek gerekirse, çoklu kalıtımın iki ya da daha fazla sınıfın nasıl tek bir sınıf içerisinde birleştirebildiğini inceler. Sonuç olarak, üst sınıflarının özelliklerini birleştiren bir sınıf ortaya çıkar. Bu örüntü, bağımsız geliştirilmiş kütüphanelerin birlikte çalıştırılabilmesi bakımından kullanışlıdır [2].

2.2.1. Uyumlayıcı (Adapter)

Uyumlayıcı örüntü tabiri caiz ise iki farklı tipteki nesneyi uyumlu hale getirir ve birbirleriyle sorunsuz bir şekilde çalışmalarını sağlar [5].Bu örüntünün amacı, kodun beklentide bulunduğu arayüze özel bir uyumlayıcı sınıf yaratmaktır [14].

Uyumlayıcı, başka bir biçimde etkileşimde bulunamayacak iki veya daha çok sayıdaki nesnelere, oldukça iyi bir biçimde yeniden kullanılabilirlik imkânı sunar [13].

Uyumlayıcı tasarım örüntüsünün yapısı Şekil 2.6'da gösterildiği gibidir.



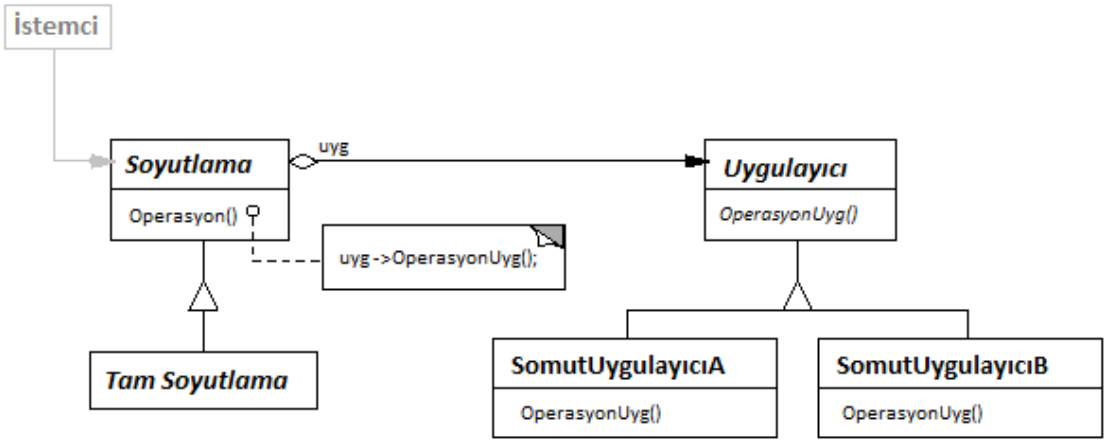
Şekil 2.6 Uyumlayıcı tasarım örüntüsünün yapısı

2.2.2. Köprü (Bridge)

Köprü örüntüsü bir uygulanışın(implementation) soyutlamasından ayrıştırırken, onların bağımsızca çeşitlenebilmelerine olanak sağlar [6].

Hiyerarşik soyutlamalar ve karşılığı olan hiyerarşik uygulanışlarmevcut olduğu zaman köprü örüntüsü fayda sağlar.Köprü örüntü, soyutlamaları ve uygulanışlarıbelirgin sınıflar haline birleştirmek yerine onları dinamik olarak birleştireceği bağımsız sınıflar halinde uygular [8].

Köprü tasarım örüntüsünün yapısı Şekil 2.7'de gösterildiği gibidir.



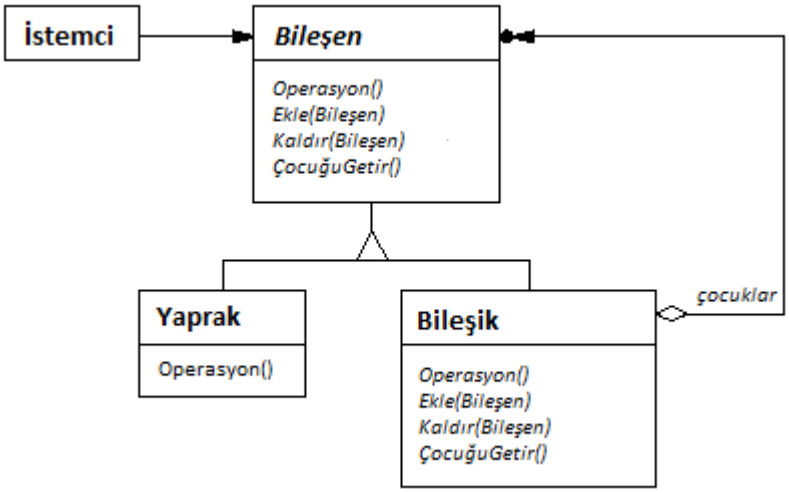
Şekil 2.7 Körpü tasarım örüntüsünün yapısı.

2.2.3. Bileşik (Composite)

Bileşik örüntü parça bütün ilişkisini göstermek için, bağımsız nesnelerin ve bunların taşıyıcılarının istemciler tarafından tekdüze bir biçimde işlenebilmesine izin veren ağaç yapısındaki bileşik nesneleri kullanır [12].

Bileşik örüntü tek bir birimin ve bileşik birimlerin aynı arayüzü ortaya çıkarmayı sağlar [10].

Bileşik tasarım örüntüsünün yapısı Şekil 2.8'de gösterildiği gibidir.



Şekil 2.8 Bileşik tasarım örüntüsünün yapısı.

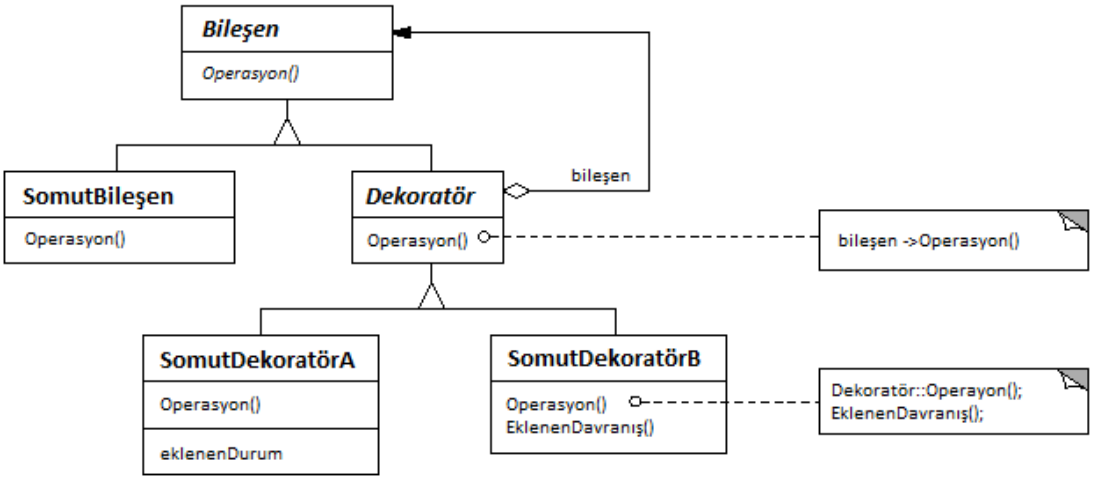
2.2.4. Dekoratör (Decorator)

Dekoratör örüntü, orijinal nesnenin yapısını değiştirmeden mevcut nesnenin içeriğini ve işlevselliğini değiştirmek ya da dekore etmek için uygundur [7].

Soyut dekoratör üst sınıfı, kendisiyle aynı tipte olan bir bileşene referans olur. Dekoratöre gelen istekler bir bileşenine ve o bileşenin bileşenine aktarılabilir vezincir böyle devam eder [14].

Bu örüntünün amacı, uygulamanın herhangi bir anında ya da yapımının tamamlanmasından sonraki bir zamanda yeni işlevsellik (ya da sorumluluklar) eklenebilmesini kolaylaştırmaktır [11].

Dekoratör tasarım örüntüsünün yapısı Şekil 2.9'da gösterildiği gibidir.



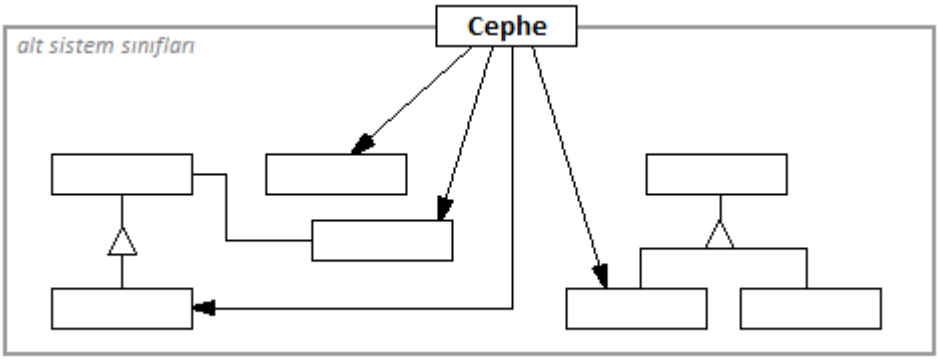
Şekil 2.9 Dekoratör tasarım örüntüsünün yapısı.

2.2.5. Cephe (Facade)

Cephe örüntüsü, bir alt sistemdeki farklı arayüzler kümesi için birleştirilmiş bir arayüz sağlanmasına yardımcı olur. Cephe örüntü, alt sistemlerin karmaşıklığını azaltarak ve aralarındaki iletişim ve bağımlılıklarını gizleyerek bu alt sistemlerin kullanılmalarını kolaylaştıacak üst seviye bir arayüz tanımlar [5].

Büyük çaptaki uygulamalarda kullanılan birçok sınıfta ve alt sistemlerde farklıkatmanları ayrı tutmak ve bağlaşımları(coupling) azaltmak önemlidir. Cephe örüntüsü bunu birleştirilmiş bir arayüz aracılığıyla gerçekleştirmeyi amaçlar [9].

Cephe tasarım örüntüsünün yapısı Şekil 2.10'da gösterildiği gibidir.



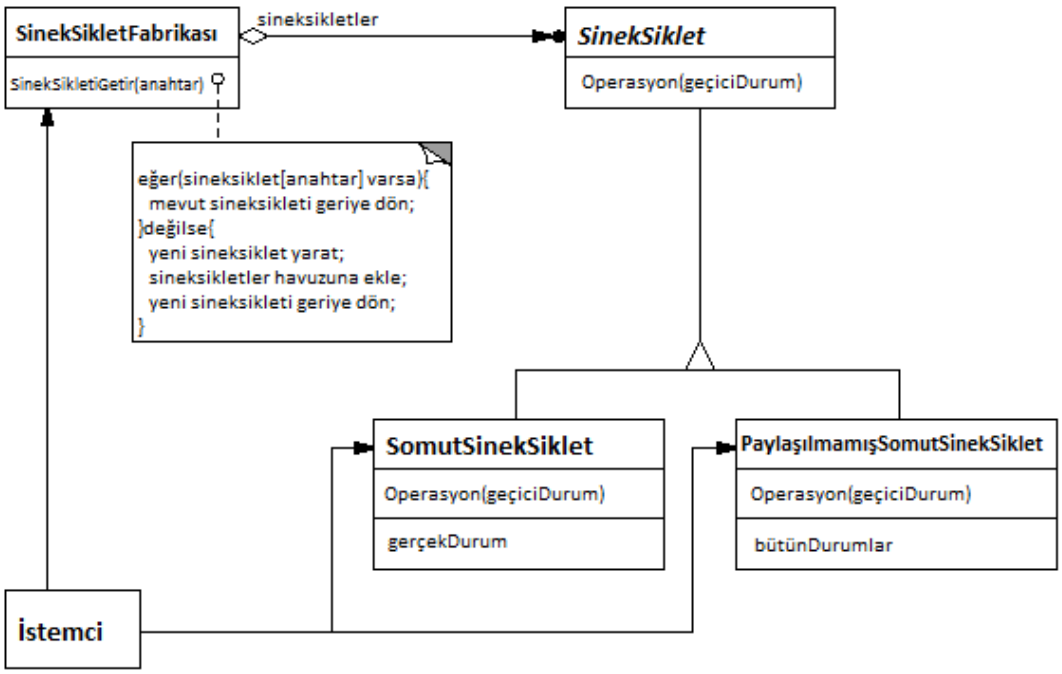
Şekil 2.10 Cephe tasarım örüntüsünün yapısı.

2.2.6. Sinek Siklet (Flyweight)

Sinek siklet örüntü, sistem içinde fazla sayıda gözlemlenen, genel bilgileri taşıyan küçük nesnelerin paylaştırılmasının etkili bir yoludur. Birçok değerin gereksiz yere kopyalandığı durumlarda depolama gereksinimlerini azaltır [6].

İstemciler, kaynak bilgisini(context information) sağlamaktan ve/veya hesaplamaktan sorumlu olan paylaşılan nesneyi kullanırlar. Bu bilgi ihtiyaç duyan paylaşılan nesnelere aktarılır [13].

Sinek Siklet tasarım örüntüsünün yapısı Şekil 2.11'de gösterildiği gibidir.



Şekil 2.11 Sinek Siklet tasarım örüntüsünün yapısı.

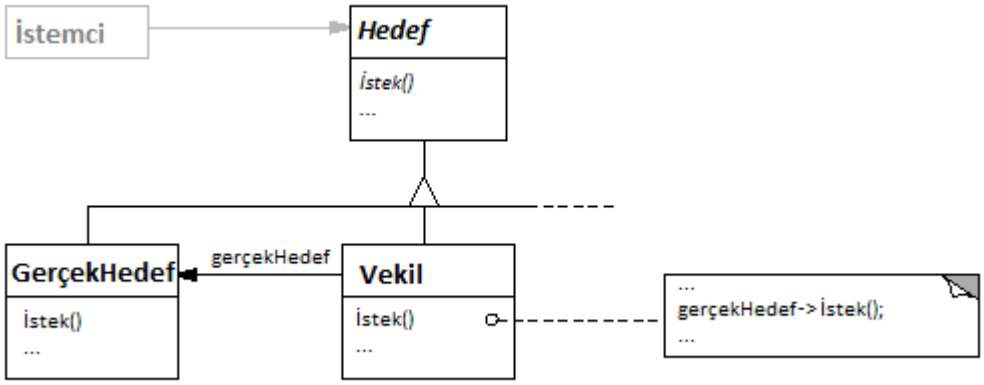
2.2.7. Vekil (Proxy)

Vekil tasarım örüntüsü, iki nesne arasında direk iletişimi ve ulaşımı keserek aracı olması için görünür bir biçimde konumlandırılmış bir nesne üretir [7].

Vekil örüntü, aynı arayüzü uygulayan bir vekil sınıfı ve bir hedef sınıftan oluşur. İlk önce, asıl hedef ile irtibat kurulup kurulmayacağına karar veren vekil nesne ile iletişime geçmek, istemci için alışılagelmiş bir durumdur [14].

İstemci kodunun kaynağa serbest erişim izninin verilmesi birçok alanda sorun yaratır ya da yaratacaktır. Bu gereklilikleri sağlamak adına Vekil örüntü programın tasarımına dahil edilir [10].

Vekil tasarım örüntüsünün yapısı Şekil 2.12'de gösterildiği gibidir.



Şekil 2.12 Vekil tasarım örüntüsünün yapısı.

2.3. Davranışsal Örüntüler

Davranışsal örüntüler algoritmalar ve nesneler arası sorumlulukların dağıtılması ile ilgilidir. Davranışsal örüntüler, sadece sınıfları ya da nesneleri değil bunların aralarındaki ilişkileri açıklayan örüntülerdir. Bu örüntüler, çalışma esnasında takip edilmesi güç olan karmaşık kontrol akışını karakterize ederler. Dikkatleri akış kontrolünden uzaklaştırarak, sadece nesnelerin birbirlerine bağlanma şekilleri üzerinde odaklanılmasını sağlar [2].

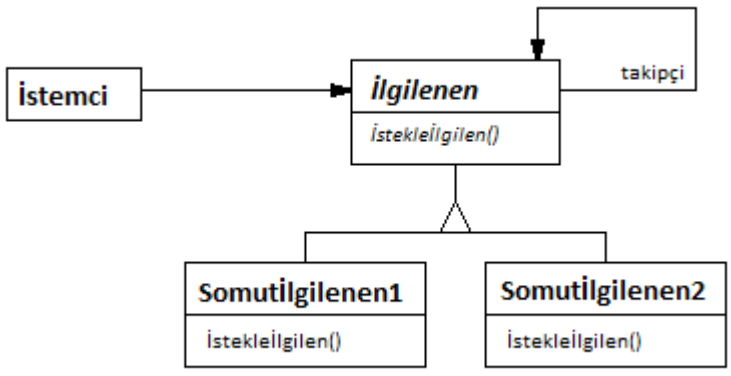
2.3.1. Sorumluluk Zinciri (Chain of Responsibility)

Sorumluluk zinciri örüntüsü bir sınıflar zinciri kullanır. Böyle bir durumda zincirin bir ya da birden fazla üyesi, istemciden gelen talebi karşılamaya vakıf olacaktır.Bir talep geldiğinde, talebi karşılayacak nesne bulunana kadar tüm zinciri gezer [14].

Aşağıda sorumluluk zinciri örüntüsünü kullanmanın sağlayacağı faydalar sıralanmıştır:

- Eşleşmeyi azaltır.
- Nesnelere sorumluluklar atanmasına esneklik getirir.
- Sınıflar kümesinin bir bütün olarak hareket etmesini sağlar, çünkü bir sınıfta oluşan durumlar talebi karşılayacak olan bir diğer sınıfa bileşik bir yapıiçerisinde gönderilebilir [8].

Sorumluluk Zinciri tasarım örüntüsünün yapısı Şekil 2.13'de gösterildiği gibidir.



Şekil 2.13 Sorumluluk Zinciri tasarım örüntüsünün yapısı.

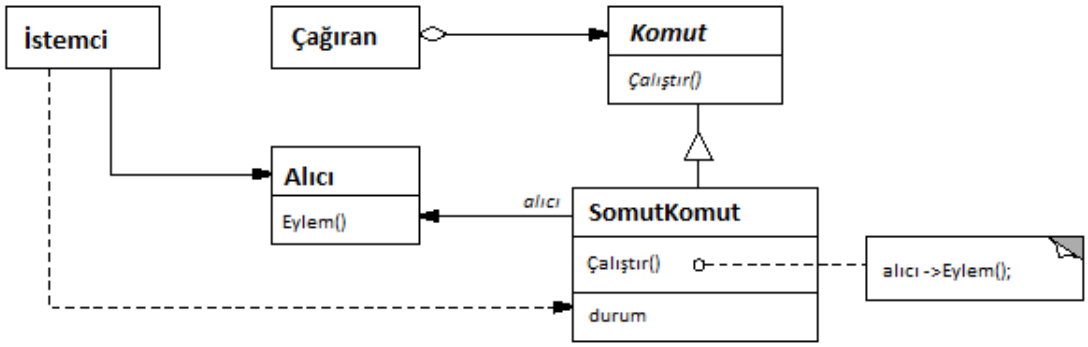
2.3.2. Komut (Command)

Komut nesnsesi hedef üzerinde talimatların nasıl uygulanması gerektiğine dair bilgiyi saklar. Böylece istemci hedef hakkında bilgi sahibi olmadan hedef üzerindemümkün olan işlemleri gerçekleştirir [5].

Komut örüntüsü, işlem talebinde bulunan istemci ve bu talebi yerine getirecek olan nesne arasında bir mesafe oluşturur. Bu örüntü bilhassa çok yönlüdür ve;

- İsteklerin farklı alıcılara gönderilmesini,
- İsteklerin sıraya konulmalarını kayıtlarının tutulmalarını(logging) ve geri çevrilmelerini,
- İlkel işlemlerden üst seviye ilişkisel işlemlerin oluşturulmasına,
- Tekrar etme ve geriye alma işlevselleiklerine katkıda bulunabilir [6].

Komut tasarım örüntüsünün yapısı Şekil 2.14'de gösterildiği gibidir.



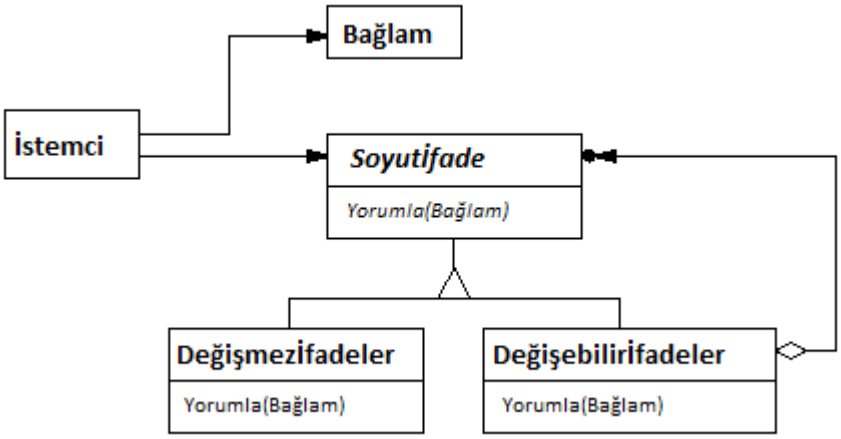
Şekil 2.14 Komut tasarım örüntüsünün yapısı.

2.3.3. Yorumlayıcı (Interpreter)

Yorumlayıcı tasarım örüntüsü, anahtar birimler için bir oluşumu inceler ve her bir anahtara karşılık gelen kendi yorumunu ya da etkisini ortaya koyar [7].

Yorumlayıcı örüntü, bir dildeki cümleleri bir takım temsiller kullanarak yorumlayan, tanımlı bir dilbilgisinin dil yorumlayıcısıdır. Bu örüntü örnek olarak, arama sorgularında kullanılır. Karmaşık bir algoritma yaratmak yerine kullanıcının kullanabileceği bir dilbilgisi tanımlanır (örneğin; VE/VEYA sorguları). Daha sonra kullanıcıların yazdığı cümleleri yorumlayacak bir dil yorumlayıcısı tanımlanır [9].

Yorumlayıcı tasarım örüntüsünün yapısı Şekil 2.15'de gösterildiği gibidir.



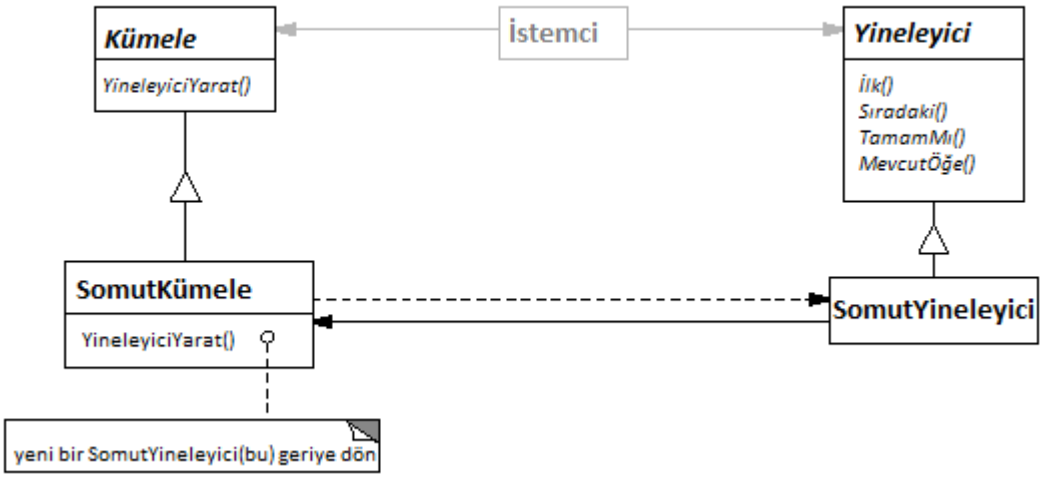
Şekil 2.15 Yorumlayıcı tasarım örüntüsünün yapısı.

2.3.4. Yineleyici (Iterator)

Yineleyici örüntü, bir koleksiyonun (collection) nasıl yapılandırıldığını bilmeden, koleksiyonun birimlerine ardışık bir biçimde ulaşılmasını sağlar [6].

Yineleyici örüntü, istemciye genel bir arayüz sağladığı için kullanışlıdır, çünkü istemcinin arka planda çalışan veri yapısını bilmesine ihtiyacı olmaz [9].

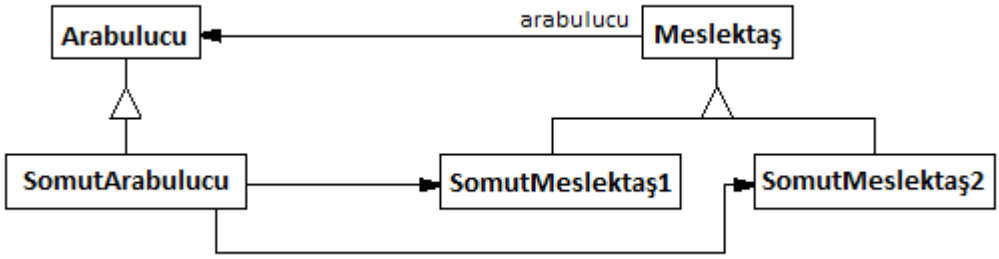
Yineleyici tasarım örüntüsünün yapısı Şekil 2.16'da gösterildiği gibidir.



Şekil 2.16 Yineleyici tasarım örüntüsünün yapısı.

2.3.5. Arabulucu (Mediator)

Arabulucu örüntü, objeler arasındaki mesaj dağılımlarını yöneten tek bir nesne tanıtarak bir sistemdeki nesneler arası iletişimi kolaylaştırır. Arabulucu örüntü, nesnelerin birbirlerini açıkça referans almalarını engelleyerek esnek bağımlılığı destekler ve birbirleri arasındaki etkileşimleri bağımsızca çeşitlendirmelerine izin verir [8].Arabulucu tasarım örüntüsünün yapısı Şekil 2.17'de gösterildiği gibidir.

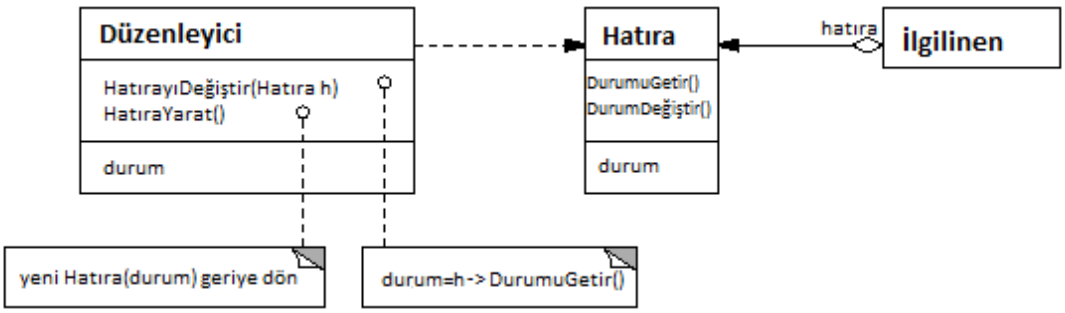


Şekil 2.17 Arabulucu tasarım örüntüsünün yapısı.

2.3.6. Hatıra (Memento)

İstemci kodu gerçek veri değerleriyle ilgilenmek yerine genellikle anlık durum kaydına ihtiyaç duyar (örneğin; kontrol noktası(checkpoint) ve geri dönme işlemleri).Bu davranışı desteklemek için, nesne kaydının dâhili verisi, Hatıra isimli yardımcı sınıftan elde edilebilir. İstemci kodu mevcut durumunu saklamak için Hatıra nesnesini kullanır ve hatıra nesnesi istemci nesnesine geri aktarılarak istemcinin önceki durumu yeniden elde edilmiş olur [9].

Hatıra tasarım örüntüsünün yapısı Şekil 2.18'de gösterildiği gibidir.



Şekil 2.18 Hatıra tasarım örüntüsünün yapısı.

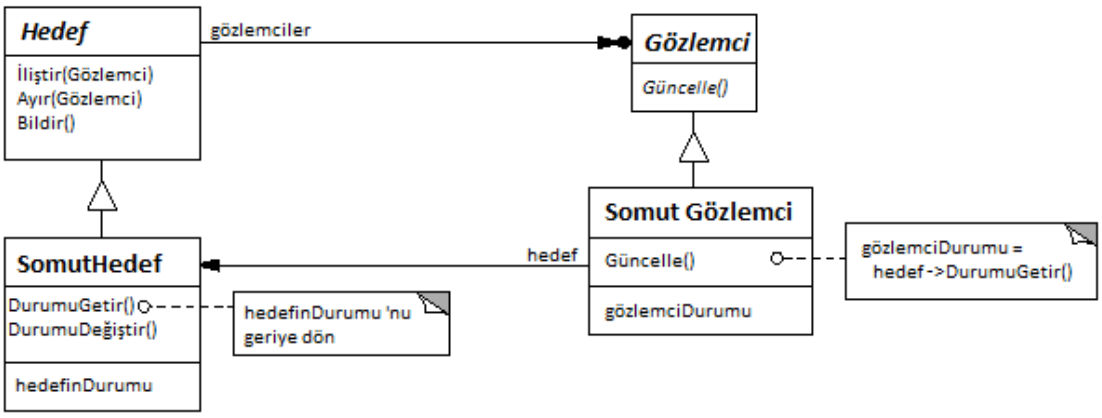
2.3.7. Gözlemci (Observer)

Gözlemci tasarım örüntüsü, hedef nesnenin durumunu izleyen nesnenin yaratılmasını kolaylaştırır ve çekirdek nesneden ayrıştırılan durum hedefli işlevselliği sağlar [7].

Gözlemci örüntü, kayıtlı gözlemciler topluluğuna yayınlanacak olan nesnenin durumundaki değişikliklere izin verir. Bu örüntü bazen sunucu baskısı ya da "yayın-abone" örüntüsü olarak da adlandırılır. Çünkü kaynak nesne, ihtiyaç duyulduğunda abonelere bilgiyi baskı yapar ya da yayınlar [14].

Gözlemci örüntü, nesneler arasında bir ilişki tanımlar böylece içlerinden birinin durumu değiştiğinde diğer nesneler bu durumdan haberdar olur. Genel olarak yeni bir durumun tanımlanabilir tek bir yayıncısı ve bilgi almak isteyen birden çok abonesi bulunur [6].

Gözlemci tasarım örüntüsünün yapısı Şekil 2.19'da gösterildiği gibidir.



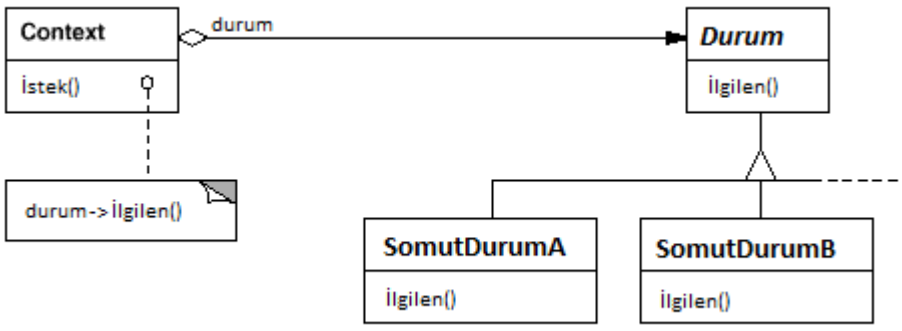
Şekil 2.19 Gözlemci tasarım örüntüsünün yapısı.

2.3.8. Durum (State)

Bir nesnenin davranışının kendi dâhili durumuna dayanarak değiştilmesi istendiğinde, durum örüntüsü kullanışlıdır. İstemciye, obje kendi sınıfını değiştirmiş gibi görünür. Durum örüntüsünün yararı, duruma özel bir mantığın durumu temsil eden sınıflar içerisinde yerelleştirilmiş olmasıdır [9].

Durum örüntüsü, nesnenin dahili durumu değiştiğinde, nesnenin davranışının da değişmesine izin verir. Nesne sınıfını değiştirmiş gibi görünür [8].

Durum tasarım örüntüsünün yapısı Şekil 2.20'de gösterildiği gibidir.

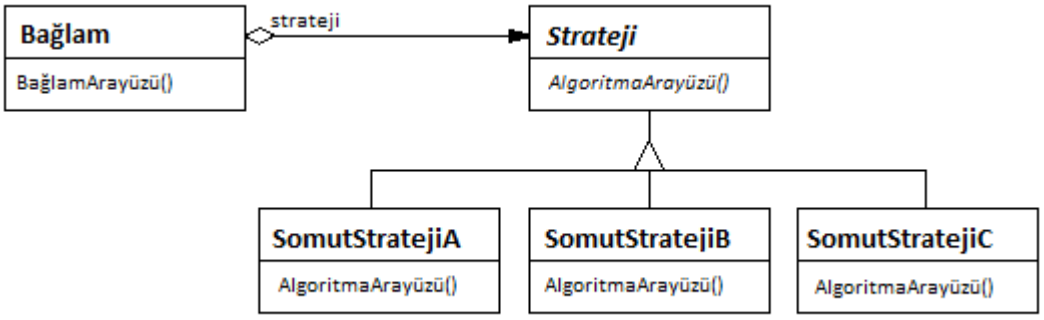


Şekil 2.20 Durum tasarım örüntüsünün yapısı.

2.3.9. Strateji (Strategy)

Strateji örüntüsü sınıflar içerisindeki algoritmaları yer değiştirebilir olmaları için sarmalar(encapsulate), böylece bu algoritmalar onları kullanan istemcilerden bağımsız bir biçimde çeşitlendirilebilirler [12].

Strateji tasarım örüntüsünün yapısı Şekil 2.21'de gösterildiği gibidir.



Şekil 2.21 Strateji tasarım örüntüsünün yapısı.

2.3.10. Şablon (Template)

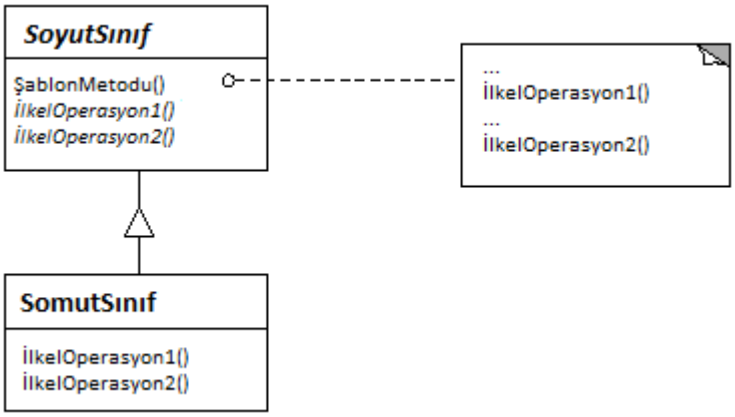
Bir şablon metot örüntüsü, uygulanışı alt sınıflara bırakılmış bir algoritma şablonunu içeren soyut bir sınıf içerisindeki tasarım özelliklerini saklar. İstemci kodunun altında yatan algoritmaları çeşitlendirecek olan alt sınıflar için, esnekliğe sahip bir varsayılan uygulayış ile genel bir arayüz çağırılması birçok durumdagereklidir [10].

Şablon örüntünün kullanıldığı durumlarda, her yeni bir sınıf tanımlanışında bütün algoritmanın yeniden kodlanmasına gerek kalmaz [14].

Şablon örüntüsü;

- Bir algoritmanın değişmez parçaları uygulandığı ve çeşitlenebilecek davranışları uygulamak için alt sınıfların kullanıldığı,
- Alt sınıflar arasındaki bilinen davranışların, bilinen bir sınıf içerisinde değişkenlerine ayrılarak ve yerelleştirilerek, gereksiz kod tekrarından kaçınıldığı durumlarda kullanılır [8].

Şablon tasarım örüntüsünün yapısı Şekil 2.22'de gösterildiği gibidir.



Şekil 2.22 Şablon tasarım örüntüsünün yapısı.

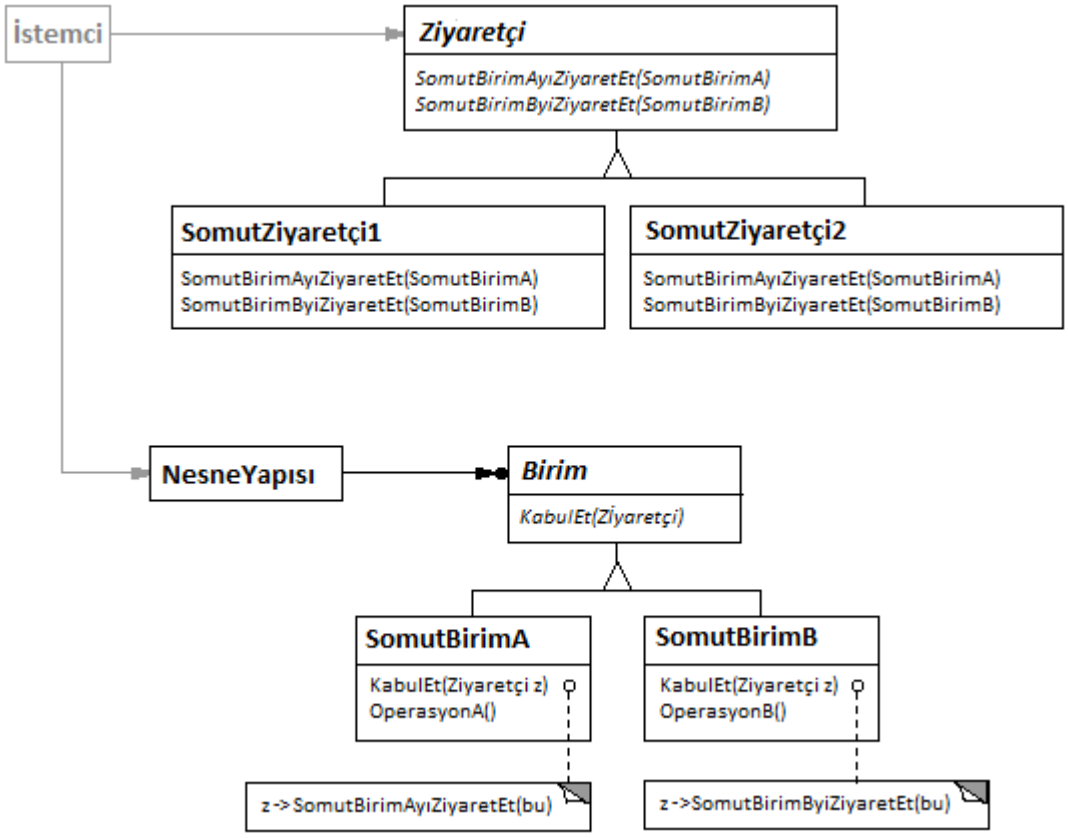
2.3.11. Ziyaretçi (Visitor)

Ziyaretçi örüntü, başka sınıflar içerisindeki veriyi etkileyebilmek için harici bir sınıf kullanır. Sınıf hiyerarşisi içerisine yerleşemeyen çokbiçimli(polymorhic)işlemler mevcut olduğu zaman, bu örüntü yaklaşımı kullanışlıdır [9].

Ziyaretçi örüntü, bir işlemin geniş bir yapı üzerinde uygulanması gerektiği ve bileşik sonuçların hesaplanması gerektiği durumlarda oldukça kullanışlıdır [13].

Ziyaretçi örüntü, temel nesnelerin karmaşıklığının azaltılmasına yardımcı olur. Temel olarak bu, nesnenin, uygulayacağı algoritmadan ayrıştırılmasıdır.

Ziyaretçi tasarım örüntüsünün yapısı Şekil 2.23'de gösterildiği gibidir.



Şekil 2.23 Ziyaretçi tasarım örüntüsünün yapısı.

BÖLÜM 3

YAZILIM KALİTESİ

İlk yıllarında Yazılım Mühendisliği Topluluğu (Software Engineers Community) ürünün kalitesi üzerinde odaklanmıştır. Sistemlerde ve projelerde tekrar eden hatalar,üzerinde durulan noktayı süreç ve ürün üzerindeki dikkate değer biçimdeki gelişmelere çevirmiştir. Kaçınılmaz bir şekilde kullanıcı, ürünün üretimindeki süreç veya maliyeti görmez. Kalite için tanımlamaların çok oluşu, bunun kendi içerisinde anlaşılabilir olmasının ve bir kavram olarak standartlaştırılmasının zorluğu gösterir. Sezgisel olarak herkes, kaliteyi ve bilhassa olmayışını farkeder. Kaliteyi tanımlama girişimleri, ölçülmesindeki zorluklar tarafından engellenir. Kalitenin geniş ölçüdeki iki tanımlaması şöyledir:

(1) Uluslararası Standatlaştırma Organizasyonu (ISO: International Standardization Organization) ; Tanımlanan yada kastedilen ihtiyaçların karşılanabilmesiyle ilgilenen özellik ve karakteristiklerin tümüdür.

(2) Elektrik ve Elektronik Mühendisleri Enstitüsünün(IEEE: Institute of Electric and Electronic Engineers) Yazılım Mühendisliği Standart Terimler Sözlüğü Terminolojisi (Standard Glossary of Software Engineering Terminology) ; Bir sistemin, bileşenin veya sürecin, tanımlı gereksinimlerle, müşteri veya kullanıcının ihtiyaç ve beklentileri ile buluşmasıdır.

Bu iki kalite yaklaşımı da, ürün üzerine odaklı gereksinimlerle ilişkili kullanıcı merkezli yaklaşımı gösterir. Kullanıcı kabulü ve müşteri memnuniyeti için, müşterinin bakış açısından kalite özelliklerinin ne denli önemli olduğu göz önünde bulundurulmalıdır. Tutarlı kalitenin, siparişlerin yinelenmesine ve itibarın oluşmasına sebebiyet veriyor olması genel olarak bilinen bir durumdur. Bu durumdan dolayı bütün

organizasyon boyunca sürekli iyileştirme yapılması fikri üzerinde temelli, mücadeleci bir avantaj sağlanması adına kalite hayati bir rol oynamaktadır.

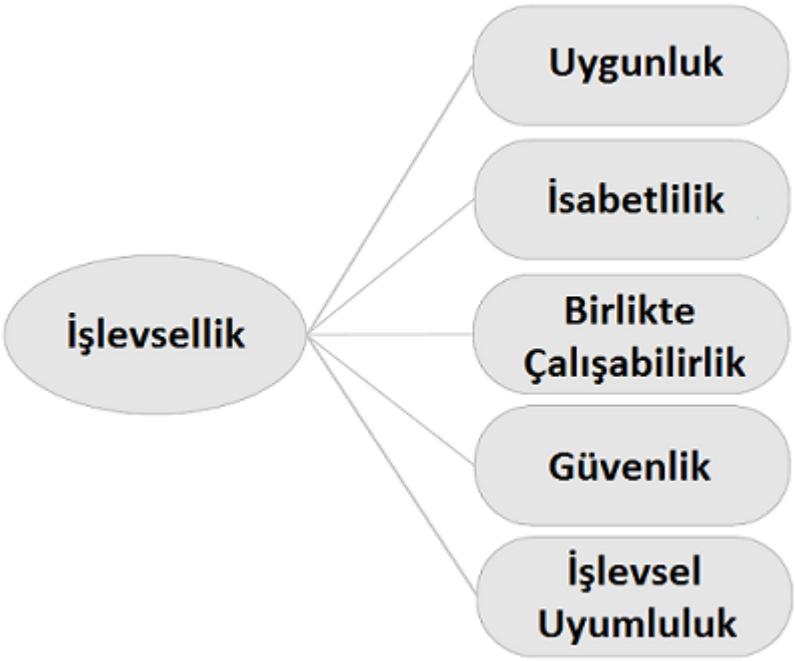
Bir kalite modeli, karakteristikler kümesi ve onların aralarındaki ilişkilerdir. Bu karakteristikler kalite gereksinimlerinin ve onların değerlendirilmesinin temelini oluşturur. ISO/IEC 9126 kalite modeli, işlevsellik, güvenilirlik, kullanılabilirlik, etkinlik, bakım yapılabilirlik ve taşınabilirlik isimleri altında altı adet üst seviye yazılım kalite karakteristiğini tanımlar. ISO/IEC TR 9126-4, yazılım kalite ölçütlerinin nasıl uygulanacağının açıklamasını, herbir karakteristik için ölçüt kümelerini ve yazılım ürünü yaşam döngüsünde ölçütlerin nasıl uygulanacağına dair verilen örnekleri içerir [36].

3.1. İşlevsellik (Functionality)

İşlevsellik sistemin istenilen işi yapabilme yeteneğidir. Birden çok ya da çoğu sistem bileşeni, bir işi tamamlayabilmek adına birbirleriye koordinasyon halinde çalışır. Bu nedenle eğer sistem bileşenlerine doğru sorumluluklar yüklenmez ise ya da diğer bileşenlerle uyum sağlayıcı doğru imkânlarla donatılmazlarsa (örnek olarak bileşenlerin görev içerisinde ne zaman devreye gireceklerini bilmeleri gösterilebilir), sistem gerekli işlevselliği yerine getirmekte başarısız olacaktır [21].

İşlevsellik, özel kullanım koşulları altında, kullanıcıların direk ya da dolaylı yollarla belirlenmiş ihtiyaçlarının karşılanmasını sağlayacak olan yazılım kabiliyetidir[18].

İşlevsellik Şekil 3.1'de gösterildiği gibi beş alt karakteristik altında incelenebilir; Uygunluk, İsabetlilik, Birlikte Çalışabilirlik, Güvenlik, İşlevsellik Uyumluluğu.



Şekil 3.1 İşlevsellik alt karakteristikleri.

3.1.1. Uygunluk (Suitability)

Uygunluk, belirlenmiş görevler ve kullanıcı hedefleri için uygun olan işlevler bütününü karşılamaya yarayan yazılım yeterliliğidir. Bu aynı zamanda görevin uygunluğunu ve yazılımın işletilebilir olduğunu gösterir [16].

Uygunluk, kullanıcıyı zorlamadan herhangi bir görevin yerine getirilmesinde, ihtiyaçları karşılayan uygulamanın işlevselliği anlamına gelir [22].

3.1.2. İsabetlilik (Accuracy)

İsabetlilik, doğru ya da kararlaştırılmış sonuca veya etkiye, gerekli doğruluk derecesi ile ulaşabilmeyi sağlayan yazılım kabiliyetidir [16].

Doğruluk oranı ya da hatadan bağımsızlıktır [19].

İsabetlilik, belirlenmiş sonuçların ya da etkilerin ortaya çıkmasını sağlayan yazılım özellikleridir [20].

3.1.3. Birlikte Çalışabilirlik (Interoperability)

Birlikte çalışabilirlik, bir sistemin diğer sistemlerle var olduğuna ve onlarla iş birliği yaptığına işaret eder. Birlikte çalışabilirlik sayesinde bir ürün sağlayıcı farklı ürünler üretebilir ve gerktiğinde kullanıcının bu ürünleri kombine edebilmesine izin verir. Bu durum ürün sağlayıcının ürünleri üretebilmesini kolaylaştırır ve hangi işlevsellik için ödeme yapılacağı ve hangi kombinasyonların elde edilebileği konularında kullanıcılara özgürlük sağlar. Birlikte çalışabilirliğe, arayüzlerin standartlaştırılması ile ulaşılabilir [15].

3.1.4. Güvenlik (Security)

Güvenlik, bilgi ve verinin, yetkisi olmayan insanlar ya da sistemler tarafından okunulmasına veya değiştirilmesine karşı koruma sağlayan, yetkisi olan insanlar ve sistemler tarafından erişilmesi esnasında da reddedilmemelerini sağlayan bir yazılım ürünü kabiliyetidir. Bu, aktarım esnasında veri üzerinde uygulanabilir bir durumdur. Güvenlik yalnızca kendi başına yazılımla ilişkili değil, sistemin tamamıyla ilişkilendirilebilecek bir kalite karakteristiğidir. Genellikle işletim sistemleri, dosya sistemleri, veritabanı yönetim sistemleri ve ağ yönetim sistemlerinin güvenliğinden emin olmak oldukça önemlidir [16].

3.1.5. İşlevsel Uyumluluk (Functionality Compliance)

İşlevsel uyumluluk, işlevsellikle ilişkili standartlara, sözleşmelere, kurallar dâhilindeki yönetmeliklere ve benzer talimatlara bağlı olan yazılım kabiliyetidir. Örnek

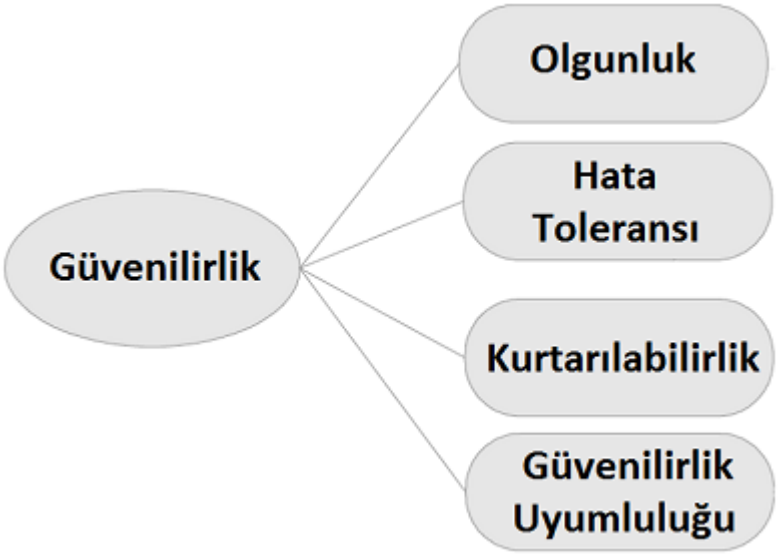
olarak muhasebe yazılımlarında bir girdinin fiziksel olarak silinmesine izin verilmez.Onun yerine, aynı kayda aynı miktarda ters yönde bir başka girdinin yaratılmasıyla mantıksal olarak silme işlemi gerçekleştirilir[16].

3.2. Güvenilirlik (Reliability)

Güvenilirlik, belirli zaman aralığında belirli koşullar altındaki performans seviyesinin elde edilebileceği, yazılım kabiliyetlerinin üzerinde ortaya çıkan özellikler kümesini işaret eden kalite karakteristiğidir(ISO/IEC,2001). Güvenilirlik, yönlendirmeler esnasındaki bilinçli ve hatadan uzak kullanıcı deneyimlerine işaret eder ve şişe ağzı denilen (bottle neck) özel durumlarda destek verir. Temel olarak, son kullanıcının hareketlerindeki sistem töleransına değinir ve ulaştırılmış bilginin isabetliliğini destekler[24].

Hali hazırda güvenilirlik, hatalar arasındaki zaman aralığının ortalamsaı, talep üzerindeki başarısızlık olasılığı, elverişlilik gibi birçok farklı yolla ölçülebilir.Güvenilirlik daha çok yazılım sistemlerindeki hata miktarına bağımlıdır [23].

Güvenilirlik dört alt karakteristik altında incelenebilir; Olgunluk, Hata Toleransı, Kurtarılabilirlik, Güvenilirlik Uyumluluğu. Şekil 3.2‘de güvenilirliğin alt karakteristikleri gösterilmektedir.



Şekil 3.2 Güvenilirlik alt karakteristikleri.

3.2.1. Olgunluk (Maturity)

Olgunluk yazılım ürünündeki hatalardan dolayı başarısızlık sıklığına işaret eder, ne kadar çok yazılım işin içerisine dâhil olursa o denli hatalar tespit edilir ve ortadan kaldırılır [15].

Bir yazılımdaki hataların çalıştırılma olasılığıdır[19].

Yazılımdaki hatalar yüzünden meydana gelen başarısızlığın görülme sıklığıyla ilişkili yazılım özellikleridir[20].

3.2.2. Hata Toleransı (Fault Tolerance)

Hata toleransı, yazılım veya donanım hatalarının varlığına rağmen, bileşenin ya da sistemin normal bir biçimde işleyebilmesi oranıdır[19].

Belirli arayüzlerde ihlaller meydana gelmesi ya da yazılım hatalarının mevcut olması durumlarında, belirli performans seviyelerinin elde edilmesi yeteneğiyle ilişkili yazılım özellikleridir[20].

Bir hata toleransı başarısızlıkları tespit etmek adına hesaplama kaynaklarını kullandığı için düşük performansa sebebiyet verir ve başarısızlık meydana geldiğinde program geri çekilme noktasına(recovery point) geri alınır(rollback) ve bileşenin ikinci bir versiyonu çağırılır[23].

3.2.3. Kurtarılabilirlik (Recoverability)

Kurtarılabilirlik, bir başarısızlık durumunda doğrudan etkilenen veriyi zamanında kurtarmak ve performans seviyesini geri kazandırma kabiliyeti ve bunun için gerekli olan çaba ile ilişkili yazılım özellikleridir[20].

Kurtarılabilirlik, elverişlilikle yakından ilişkilidir. Bir uygulama veya sistem başarısızlığından sonra etkilenen veriyi kurtarma ve beklenen performansı yeniden sağlama kapasitesine sahip olan bir uygulama kurtarılabilirdir. Kurtarma süreci esnasında uygulama kullanılabilir değildir ve bu nedenle kurtarmanın ortalama süresi ilgilenilen önemli bir ölçüttür [25].

3.2.4. Güvenilirlik Uyumluluğu (Reliability Compliance)

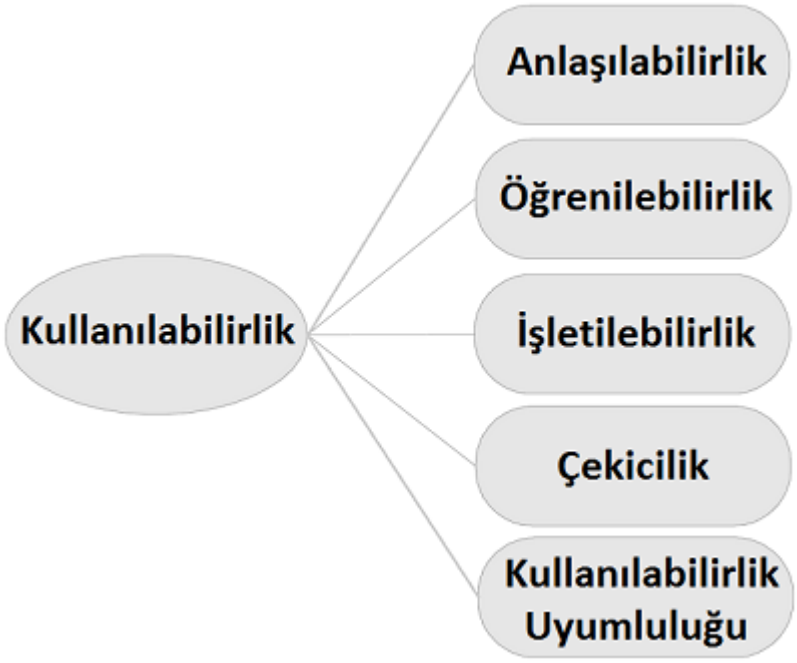
Güvenilirlik uyumluluğu, ürünün güvenilirliğine dair standartlara ve yönetmeliklere ne denli bağlı kaldığının oranıdır [19].

Gerekliliklerdeki hatalar üzere odaklanan yazılım güvenilirlik uyumluluk modeli, yazılım güvenilirliği ile uyumluluğa aykırı olan görüş arasındaki temel ilişkiyi açığa çıkarmayı önermektedir [26].

3.3. Kullanılabilirlik (Usability)

1991 yılında ISO 9126, kullanılılabilirliği; "belirli ya da dolaylı kullanıcıların kullanımda ihtiyaç duyduğu çaba ve kullanımın bireysel anlamda değerlendirilmesiyle ilgisi olan özellikler kümesi" olarak açıklamıştır. Bu, daha sonrasında ürün yönelimli kullanılabilirlik yaklaşımı olarak ileri sürülmüştür.Kullanılabilirlik yazılım kalitesinin bağımsız bir faktörüdür ve kullanımı kolaylaştıran arayüzler gibi yazılım özellikleri üzerine odaklanmıştır. Bununla birlikte kullanılabilirlik için gerekli olan özellikler kulanıcıya, yapılan işe ve ortama bağımlıdır. 2000 yılında yapılan ISO/IEC 9126-1 düzenlemesinde de belirtildiği gibiürün yönelimli yaklaşımda, kullanılabilirliğin yazılım kalitesine nispeten bağımsız katkısı vardır. Tanımlı koşullar altında kullanıldığı zaman, yazılım ürününü kullanıcı bakımından anlaşılabilir olma, öğrenilebilme ve beğenilir olma gibi kabiliyetlere sahiptir [17].

Kullanılabilirlik Şekil 3.3'de gösterildiği gibi beş alt karakteristik altında incelenebilir; Anlaşılabilirlik, Öğrenilebilirlik, İşletilebilirlik, Çekicilik, Kullanılabilirlik Uyumluluğu.



Şekil 3.3 Kullanılabilirlik alt karakteristikleri.

3.3.1. Anlaşılabilirlik (Understandability)

Bazı yazılım sistemleri diğerlerinden daha anlaşılabilirdir. Öyle ki bazı görevler doğal olarak diğerlerinden daha karmaşıktır. Verilen görevler doğal olarak benzerzorluklar taşır, daha anlaşılabilir tasarımlar üretmek ve daha anlaşılabilir programlar yazmak için güvenilir ilkeler takip edilebilir. Örneğin soyutlama ve birimselleştirme, bir sistemin anlaşılabilirliği arttırabilir [15].

3.3.2. Öğrenilebilirlik (Learnability)

Öğrenilebilirlik, bir kullanıcının(sistem geliştiricisi) uygulamayı öğrenebilmesini mümkün kılan yazılım birimi yeteneğidir. Öğrenilebilirlik ölçüsü, sistem geliştiricilerinin belirli işlevleri kullanmayı öğrenmelerinin ne kadar süreceğini ya da belgelendirme işlemlerinin ne denli etkili olacağını belirleyebilir nitelikte olmalıdır. Örneğin kullanıcı dökümanı ve yardım sistemi tamamlanmalıdır. Yardım içeriğe karşı hassas, yaygın bilinen görevlerin nasıl tamamlanacağını açıklıyor olmalıdır [27].

3.3.3. İşletilebilirlik (Operability)

İşletilebilirlik kullanıcıyı(sistem geliştiricisi), yazılımı işletebilir ve kontrol edebilir hale getiren yazılım bilimi yeteneğidir. Bir işletilebilirlik ölçüsü, sistem geliştiricilerinin, bileşeni işletebilir ve kontrol edebilir olup olmadığını belirleyebilmelidir. İşletilebilirlik ölçütleri bileşenin;

- görev için uygun olması,
- kendisini tanımlayabiliyor olması,
- kontrol edilebilirliği,
- kullanıcı beklentileriyle uyumluluğu,
- hata tolereansı,
- kişiselleştirilmeye uygunluğu

özellikleri ile kategorize edilebilir.

3.3.4. Çekicilik (Attractiveness)

Çekicilik, ürünün kullanıcıya çekici gelmesinin oranını ifade eder. Çekicilik, potansiyel üyenin ilgisini üzerine çekebilecek ve onun projeye dâhil olmasının sağlayabilecek proje yeteneği olarak tanımlanır.(Stewart & Gosain 2006) [28].

3.3.5. Kullanılabilirlik Uyumluluğu (Usability Compliance)

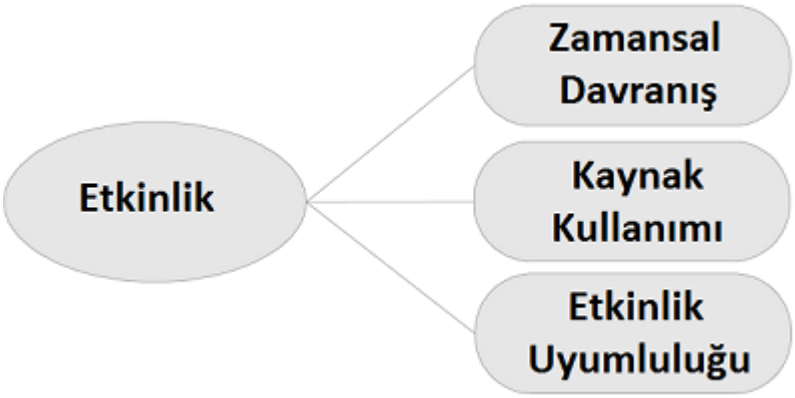
Kullanılabilirlik uyumluluğu, ürünün kullanılabilirliğine dair yönetmeliklere ne denli bağlı kalındığının oranıdır [27].

Kullanılabilirlik uyumluluğu, kullanılabilirlik seviyesine ulaşabilmek için yazılım ürününün uluslararası standartları ve sertifikaları takip edip etmediğini gösterir[29].

3.4. Etkinlik (Efficiency)

Etkinlik, uygulayış zorlukları olduğu kadar kavramsal güçlüklere de yol açan karmaşık bir kavramdır. Etkinlik, belirli şartlar altında kullanılan kaynak miktarı ile ilişkili olarak, sistemin uygun performansı sağlayabilme kabiliyetidir (ISO/IEC, 2001). Sistem işlevlerinin hem kullanılabilir hem de başarılı olduğu bir duruma işaret eder [24].

Etkinlik Şekil 3.4‘de gösterildiği gibi üç alt karakteristik altında incelenebilir; Zamansal Davranış, Kaynaksal Davranış, Etkinlik Uyumluluğu.



Şekil 3.4 Etkinlik alt karakteristikleri

3.4.1. Zamansal Davranış (Time Behaviour)

Zamansal Davranış, bir yazılım ürününün belirli şartlar altında işlevini yerine getirirken ki uygun cevap dönme ve işleme zamanları ve de çıktı oranları ile ilişkili olan özelliklerdir [30].

3.4.2. Kaynak Kullanımı (Resource Utilization)

Kaynak kullanımı,belirli şartlar altındaki bir yazılım ürününün kendi işlevini yerine getirdiği esnada, uygun miktarda ve tipte kaynak kullanabilme kabiliyetidir [20].

Kullanılan kaynaklar ve kaynakları kullanırken harcanan zamanı ilgilendiren bir ölçüt tarafından hesaplanan, özellik karmaşıklığını içerir. Mimari seviyede, özellikler her bir işlevsellik için ölçülebilir ve tanımlanabilir. Zaman ve mekân bileşen ile ilişkilendirilmiştir. Değerler, her bir işlevsellik için bir bileşen ve/veya bağlayıcı ile ilşkilidir [31].

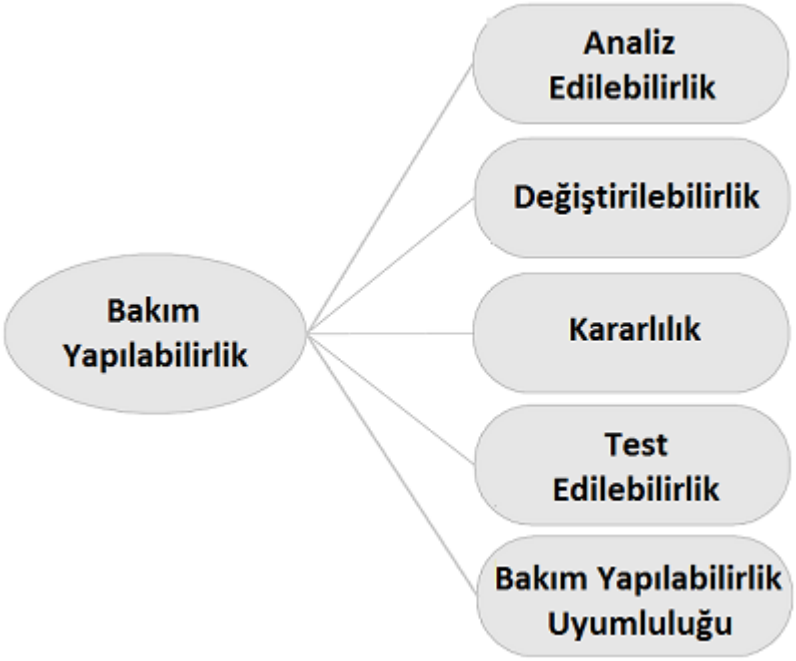
3.4.3. Etkinlik Uyumluluğu (Efficiency Compliance)

Etkinlik uyumluluğu, ürünün etkinliğine dair standartlara ve yönetmeliklere ne denli bağlı kalındığının oranıdır [27].

3.5. Bakım Yapılabilirlik (Maintainability)

Bakım Yapılabilirlik, yazılım sistemlerinin bakımının kolaylaştırılmasına işaret eder. Yazılım bakım işlemlerinin iki çeşidi mevcuttur. Bunlardan biri işlem esnasında ortaya çıkan hataların düzeltilmesidir. Böyle değişiklikler onarıcı bakım olarak adlandırılır. İkincisi ise, sistem yazılımının, işletim sisteminin veya veritabanı yönetim sisteminin güncellenmeleri gibi ortamsal değişikliklerden dolayı yapılanbakım işlemidir. Her iki tip bakım işleminde de hataların düzeltilmesi ve ortamsal değişikliklere uyum sağlanabilmesi için yazılım sisteminin nasıl çalıştığını bilen yazılım mühendislerini gereksinim vardır. İyi yapılandırılmış tasarım, sistemin anlaşılmasında sistem mühendislerine yardımcı olur. Bundan dolayı, bakım yapılabilirlik detaylı tasarıma ve arayüz tasarımına daha az bağımlı olduğu zaman, mimari tasarımın bakım üzerinde önemli bir rolü vardır [23].

Bakım yapılabilirlik Şekil 3.5 ‘de gösterildiği gibi,beş alt karakteristik altında incelenebilir; Analiz edilebilirlik, Değiştirilebilirlik, Kararlılık, Test Edilebilirlik, Bakım Yapılabilirlik Uyumluluğu.



Şekil 3.5 Bakım Yapılabilirlik alt karakteristikleri.

3.5.1. Analiz Edilebilirlik (Analyzability)

Analiz edilebilirlik, yazılım başarısızlıklarının nedenlerini teşhis eden veya geliştirilecek kısımların tanımlayan yazılım ürünü kabiliyetidir [32].

Geliştirilecek kısımların tanımlanmasında bilinen strateji, programın akış kontrolünü takip etmektir. Ancak bu strateji uygulamaya konulmadan önce, sistem mühendisinin tanımlı bir özellik için akış kontrolünü takip etmeye nereden başlayacağını tanımlaması gerekmektedir [33].

3.5.2. Değiştirilebilirlik (Changeability)

Yazılımlar mutlak değişim ihtiyacından muzdariptirler. Öyleki, insan elinden yapılmış olan hemen hemen herşey değişime tabidir. Ancak sürümü gerçekleşmiş bir yazılımın üzerinde yapılan değişiklikler farkı değerlendirilmelidirler. Bunun sebebi

kısmen, sistem yazılımının değişim baskısını en fazla hisseden sistem işlevini ihtiva etmesidir.Çünkü yazılım, üzerinde kolay değişiklik yapılabilir olarak algılanmalıdır [23].

3.5.3 Kararlılık (Stability)

Kararlılık, model değişikliklinden ötürü ortaya çıkan istenmeyen etkilere karşı dayanıklılıktır. Diğer bir deyişle, önemli derecede kararlılığa sahip olabilmek için model değişiklikliğinin etkileri mümkün olduğu kadar düşük tutulmalıdır [8].

Kararlılık, değişikliklerde ortaya çıkabilecek beklenmeyen etkilerden kaçınmayı sağlayan yazılım kabiliyetidir [31].

3.5.4. Test Edilebilirlik (Testability)

Test edilebilirlik özelliği, test hedeflerine ulaşılmasını kolaylaştıracak etkiye sahip yazılım karakteristiklerini içermektedir.

Bir yazılım sistemi eğer;

- bileşenleri ayrı ayrı test edilebiliyorsa,
- test tipleri sistematik ve tekrarlanabilir biçimde tanımlanabiliyorsa,
- test sonuçları gözlemlenebiliyorsa,

o yazılım test edilebilir olarak kabul edilir. Yazılım hatalarının açığa çıkmasının ilişkisel kolaylığı ve maliyetidir[34].

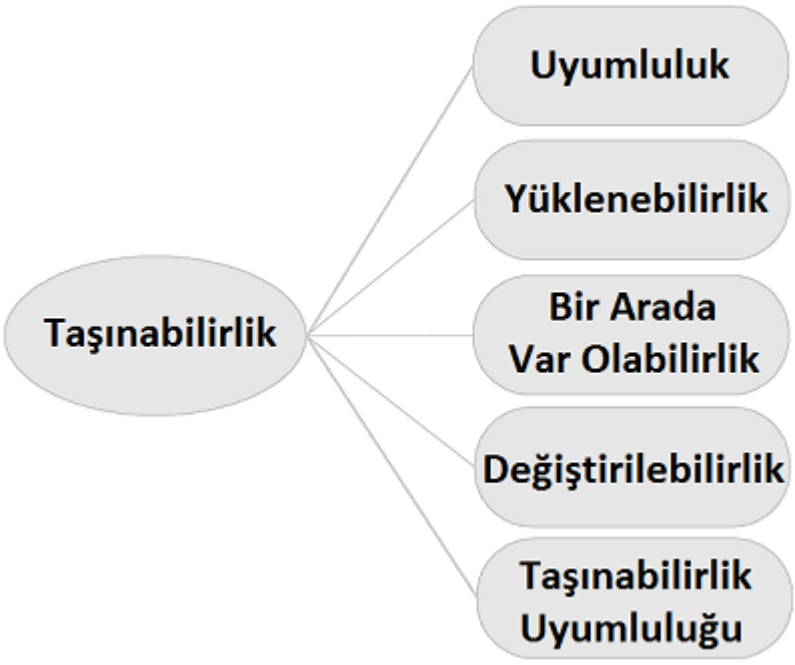
3.5.5. Bakım Yapılabilirlik Uyumluluğu (Maintainability Compliance)

Bakım Yapılabilirlik uyumluluğu, ürünün bakımının yapılabildiğine dair standartlara ve yönetmeliklere ne denli bağlı kalındığının oranıdır [19].

3.6. Taşınabilirlik (Portability)

Taşınabilirlik, bir yazılım sisteminin bir yazılım veya donanım ortamından bir başkasına kolayca taşınabilmesi özelliğidir. Bir ortamdan diğerine taşıma işlemi genellikle, sistem çağrıları gibi ortam tarafından sağlanan imkanlara bağlı olan kod parçalarının değiştirilmesini gerektirir. İyi yapılandırılmış bir tasarımda, bir başka ortama taşınacak olan kod değiştirilmek istendiğinde; ortama bağımlı olan kod az sayıdaki bileşenler halinde gruplanarak, bütün sistemin yeniden yazılması yerine sadece gerekli bileşenlerin değiştirilmesi sağlanır [25].

Taşınabilirlik Şekil 3.6 ‘da gösterildiği gibi,beş alt karakteristik altında incelenebilir; Uyumluluk, Yüklenebilirlik, Birarada Var Olabilirlik, Değiştirilebilirlik, Taşınabilirlik Uyumluluğu.



Şekil 3.6 Taşınabilirlik alt karakteristikleri.

3.6.1. Uyumluluk (Adaptability)

Yazılım uyumluluğu, kullanıcılara sistem karakteristiğini değiştirebilecek araçları sağlayabilen sistem yeteneğidir. Farklı tanımlı platformlara uyum sağlıyabiliyor olmaktır. Sistemin yeni ortama adapte oluşunu kolaylaştırma derecesi ile değerlendirilir. Bu nedenle uyumluluk faktörü, yazılım birimselliği, iletişim bütünselliği ve veri bütünselliği ile ilgilenir [29].

3.6.2. Yüklenebilirlik (Installability)

Yüklenebilirlik, bir ürünün ya da sistemin belirli bir ortama etkili ve etkin bir biçimde kurulabilirliğinin ya da kaldırılabilirliğinin derecesi olarak tanımlanabilir. Yüklenebilirliğin ölçümünde çok az sayıda ölçüt mevcuttur ve bunların hepsi deneysel olarak nitelendirilebilir. Bundan ziyade birçok ölçüt etkili bir biçimde aynı şeyi ölçmektedirler. Örnek olarak çabasız yükleme, yüklemenin kolaylaştırılması ve kullanıcının yükleme işlemini kolaylaştırma gibi bütün ölçütler, yükleme esnasındaki kullanıcı faaliyetletinin ihtiyaç boyutunu ölçerler [35].

3.6.3. BirArada Var Olabilirlik (Co-Existance)

Birarada var oluş, iki ya da daha fazla sistemin birbirlerinin farklılıklarına saygı gösterdiği ve çekişmelerin saldırgan olmayan bir tavırla çözüldüğü bir durumdur.Sistem entegrasyonu, çeşitli uygulama yazılımlarının çeşitli işlevlerinin bir uygulama içerisindeki kombinasyonudur. Eğer bir sistem diğerleri ile entegre olmaya elverişli ise, yazılım sisteminin bağımsızlığı ve makine bağımsızlığı olmak üzere göz önünde bulundurulacak iki adet karakteristik vardır. Yazılım sisteminin bağımsızlığı programın, standart olmayan programlama dili özelliklerinden, işletim sistemi karakteristiğinden ve diğer ortam kısıtlarından bağımsızlığının derecesini temsil eder. Makine bağımsızlığı(donanım bağımsızlığı), yazılımın onu işleten donanımdan ayrışmasıdır [29].

3.6.4. Değiştirilebilirlik (Replaceability)

Değiştirilebilirlik yazılımın, yazılım ortamındaki tanımlı diğerleri ile değiştirilebilirliğini ifade eder. Yazılımın bir önceki sürümü ile uyumlu olup olmadığınıyani yazılım ürününün büyük çabalar harcamadan önceki sürümünün yerine kolayca geçip geçemeyeceğini ölçer [29].

3.6.5. Taşınabilirlik Uyumluluğu (Portability Compliance)

Taşınabilirlik uyumluluğu, taşınabilirlik karakteristiklerini sağlayabilmek adına standartları veya uluslararası sertifikaları ne denli takip ettiğidir [29].

BÖLÜM 4

YAZILIM EVRİMİ

Yazılım ilk sürümünün ortaya çıkmasından çok sonra da evrimleşmeye devam eder. Günümüze kadar yapılan araştırmalar, yazılımın bakım ve evrim maliyetlerinin, yazılım sistemi ile ilişkili toplam maliyetin en az %50 'sini bazen de %90 'dan fazlasını oluşturduğunu göstermektedir. Bu maliyeti düşürmek adına, yöneticilerin ve geliştiricilerin, yazılımı evrime zorlayan etmenleri anlamaları, değişiklikleri kolaylaştıracak ve yazılımın yapısının bozulmamasını sağlayacak adımları atmaları gerekmektedir [37].

Yazılım evrimi olgusu 1970'lerde ilk defa büyük çapta yazılımlar geliştirildiğinde gözlemlenmiş ve 1990'larda yeniden ilgi çekmeye başlamıştır. Şimdi ise yazılım evrimi, yazılım mühendisliğinde bilinen bir tabir ve kabul gören bir alandır. Konuyla alakalı konferanslar ve atölyeler gerçeklenmekte, evrim yazıları, geleneksel yazılım mühendisliği konferanslarında ve bültenlerinde sıklıkla görülmeye başlamıştır.

Lehman ve Belady'nin klasik ve algılaması güçlü çalışmasından sonra yazılım evrimi, üzerinde çalışılmaya değer bulunmuş ve yazılım projelerinde ciddi sorunlara yol açan tavırların tanınmasını sağlamıştır. Yazılımlarda ortaya çıkan sorunlar karmaşıktır çünkübu sorunlar, yönetimsel ve ekonomik açılardan, programlama dilinden veya ortamdan kaynaklanabilirler. Daha da ötesi, yazılım mühendisliği ilerledikçe ve yeni teknolojiler, süreçler tanıtıldıkça, yazılım evrimi yeni sorunlarla ve zorluklarla karşı karşıya gelecektir. Aynı zamanda bazı yeni ilerlemeler, yazılımın evriminde yeni çözümleri de mümkün kılacaktır [38].

Yazılım evriminin en ön plandaki çalışmaları M.M. Lehman ve arkadaşları tarafından 1970'lerden 1990'lara kadar uzanan 30 yıllık bir aralıkta gerçekleşmiştir. Çalışmalar Lehman ve arkadaşları tarafından yazılım evriminin 8 kuralı formüle edilerek ve iyileştirilerek sonuçlanmıştır. Bu kurallar, ortak-temelli çeşitliliğe sahip

geniş ölçekli yazılım sistemlerinin evrimi üzerinde yapılan dikkatli ve deneysel çalışmaların sonucudur [39].

Lehman'ın yazılım evrimi kanunları;

1) *Süregelen Değişim*: Birgömülü tipteki yazılım sistemi devamlı olarak uyum göstermek zorundadır, aksi takdirde artan bir biçimde yeterli olmaktan uzaklaşır.

2) *Artan Karmaşıklık*: Birgömülü tipteki yazılım sistemi evrilir, bakım işlerinin tamamlanması veya azaltılması bitmezken, karmaşıklığı artma eğiliminde olur.

3) *Öz Düzenleme*: Gömülü tipteki yazılım sisteminin evrimleşme süreci, ürünün ve sürecin özellik ölçütlerinin normal dağılımına yaklaşan öz düzenlemedir.

4) *Örgütsel İstikrarın Korunması*: Evrimleşen bir gömülü tipteki yazılım sistemindeki ortalama etkili küresel faaliyet oranı, ürünün yaşam süresi ile değişmez.

5) *Aşinalığın Korunması*: Evrimleşen bir gömülü tipteki yazılım sisteminin faal hayatı süresindeki başarılı sürümlerinin içeriği, istatistiksel olarak değişmez.

6) *Süregelen Büyüme*: Birgömülü tipteki yazılım sisteminin işlevsel içeriği, ömrü boyunca müşteri memnuniyetini sağlamak için sürekli arttırılmalıdır.

7) *Azalan Kalite*: Birgömülü tipteki yazılım sisteminin, sıkı bir biçimde bakımı yapılmazsa ve değişen işlevsel ortama olan uyumu sağlanmazsa, kalitesinin azaldığı algılanır.

8) *Geri Besleme Sistemi*: Gömülü tipteki yazılım sistemleri çoklu döngülerden, çok seviyeli geri beslemelerden oluşur ve başarı ile değiştirilmiş veya iyileştirilmiş olarak ele alınmalıdır [40].

BÖLÜM 5

KULLANILAN YÖNTEMLER, YAZILIMLAR VE UYGULAMANIN GERÇEKLEŞTİRİLMESİ

5.1. Tasarım Örüntülerinin Tespit Edilmesi

Bu çalışmada incelenen açık kaynak kodlu yazılımlarda mevcut olan tasarım örüntülerini tespit etmek için, yine açık kaynak kodlu bir yazılım olan PINOT tercih edilmiştir.

PINOT, gömülü bir örüntü analiz motoru ile birlikte Jikes(C++ dilinde yazılmış bir açık kaynak kodlu Java derleyicisi) tarafından inşa edilmiştir. Bir örüntü tanıma aracında temel olarak bir derleyicinin kullanılmasının çok sayıda avantajı vardır. Bir derleyici, sınıf içi ve statik davranışsal analizi kolaylaştıran sembol tablolarını ve soyut sözdizim ağacını(AST: abstract syntax tree) inşa eder. Derleyiciler aynı zamanda anlamsal kontroller yaparak örüntü analizine yardımcı olur. En önemlisi de derleme hataları, doğru olmayan örüntü tanımlama sonuçlarına yol açacak olan sembol tablolarının ve soyut sözdizim ağacının eksik kalmalarına neden olur. Hâlbuki FUJABA ve PTIDEJ gibi bazı araçlar, tamamlanmamış kaynaktan örüntüleri kısmen tanıyabilirler. Eğer örüntü tanıma, çalışma esnasında örüntülerin yapılandırılması ve birleştirilmesi gibi ileri mühendisliğin bir parçası olarak kullanılacaksa bu tarz araçlar tercih edilebilir. Bizim durumumuzda örüntü tanıma ters mühendislikte(reverse engineering) kullanılacağından, isabet oranının hayati önemi vardır.

Bir örüntü tanıma aracının bütünlüğü, örüntü uygulanış çeşitliliklerini tanıyabilmesi yeteneğiyle belirlenir. Makul sebeplerden ötürü PINOT, gerçekte kullanılmış bilinen uygulanış çeşitliliklerini bulmak üzerine odaklanmıştır. Bundan dolayı, bazı davranışsal analiz teknikleri, herbir davranışsal örüntü üzerinde tam olarak uygulanamamıştır.

PINOT, iki adet benzer araçla karşılaştırılmıştır; HEDGEHOG ve FUJABA HEDGEHOG, örüntü özelliklerini, kullanıcıların sınıf içi ilişkileri ve yol hassasiyeti olmayan diğer anlamsal analizleri tanımlamalarına izin veren SPINE(Prolog benzeri bir tanımlama dili) üzerinden okur. Fakat daha karmaşık bir anlamsal analiz, gömülü ifadeye sıkı sıkıya bağlıdır. Bu tüzden SPINE, HEDGEHOG tarafından sağlanan anlamsal analizlerin yeteneğine bağlıdır. Bu aracı kullanmak için kullanıcı, hedef sınıfı ve karşılığında soruşturacağı örüntüyü tanımlamalıdır.

FUJABA, yazılım tekrar mühendisliği(re-engineering) için zengin bir grafiksel kullanıcı arayüzüne sahiptir. Örüntü sonuç alma motoru, kullanıcı tarafından tanımlanmış örüntüler için UML (Unified Modelling Language) benzeri görsel bir dil sağlar. Bu dil sınıf içi ilişkilerini ve "yaratan" ilişkilerinin tanımlanmasına izin verir. FUJABA'nın kullanımı kolaydır; kullanıcı basit bir biçimde kaynak kodun yerini belirtir ve örüntü sonuç alma motorunu çalıştırır. FUJABA sonucu grafiksel olarak gösterir. FUJABA, tamamen otomatik veya etkileşimli kullanıcı yönlendirmesi ile arama alanınını daraltabilir. PINOT tamamen otomatikleşmiştir; kaynak paketi alır ve örüntü örneklerini tespit eder. Yapısal ve davranışsal örüntülerin doğruluğunu kanıtı olarak, bütün tanımlama algoritmaları kodun üzeride gömülü haldedir.

Bu üç araç farklı kullanımlar için inşa edilmiş olmalarına rağmen hepsi örüntü örneklerini tanımlama işine müdahil olmuşlardır. Bundan dolayı bu araçlar isabetliliklerine göre kıyaslanmıştır. Tablo 5.1 ‘de, Uygulamalı Java Örüntüleri(Applied Java Paterns[13]) kitabındaki örnek kaynak kodunun her bir araç tarafından yapılan testlerin sonuçları gösterilmektedir.

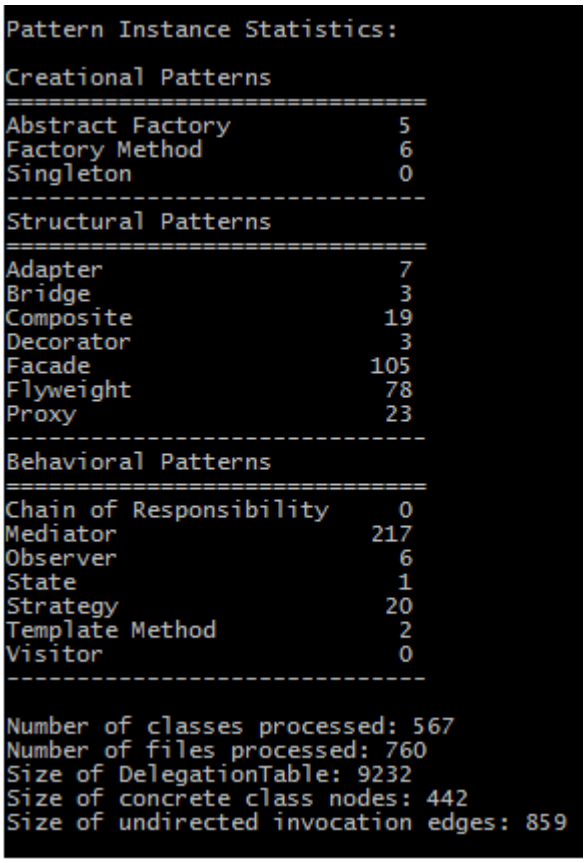
Tablo 5.1 Örüntü Tanıma Araçlarının Örnek Kod Üzerinden Elde Ettiği Sonuçlar

	TASARIM ÖRÜNTÜLERİ	PINOT	HEDGEHOG	FUJABA
Yaratımsal	*Soyut Fabrika*	EVET	EVET	HAYIR
	Yapıcı			
	Fabrika	EVET	EVET	HAYIR
	Örnek		HAYIR	
	Tek	EVET	EVET	EVET
Yapısal	*Uyumlayıcı*	EVET	EVET	HAYIR
	Köprü	EVET	EVET	EVET
	Bileşik	EVET	EVET	HAYIR
	Dekoratör	EVET	EVET	HAYIR
	Cephe	EVET		EVET
	Sinek Siklet	EVET	EVET	HAYIR
	Vekil	EVET	EVET	
Davranışsal	*Sorumluluk Zinciri*	EVET		EVET
	Komut			
	Yorumlayıcı			
	Yineleyici		EVET	HAYIR
	Arabulucu	EVET		HAYIR
	Hatıra			HAYIR
	Gözlemci	EVET	EVET	HAYIR
	Durum	EVET	HAYIR	
	Strateji	EVET	EVET	EVET
	Şablon	EVET	EVET	EVET
	Ziyaretçi	EVET	EVET	

Tablo 5.1 ‘de [41] görülmektedir ki, PINOT örnek kod üzerindeki tüm yapısal ve davranışsal örüntüleri tespit etmiştir. Çünkü PINOT bir örüntü tanımlama aracıdır,

bir sınıfın herhangi bir örüntü barındırabileceğini farzeder. Bundan ötürü PINOT bir sınıf üzerindeki bütün örüntü tanımlamalarını test eder. FUJABA da yaklaşık buna benzer bir yolla test yapar. HEDGEHOG otomatik doğrulayan bir araç değildir ve kullanıcılar hedef sınıf üzerindeki örüntüleri seçmekten sorumludurlar. Bundan ötürü, Tablo 5.1 ‘de gösterilen sonuçlar kaynak kod hakkında edinilen temel bilgi üzerine temellendirilmiştir ve sınıf üzerindeki muhtemel örüntüler doğrulamıştır [41].

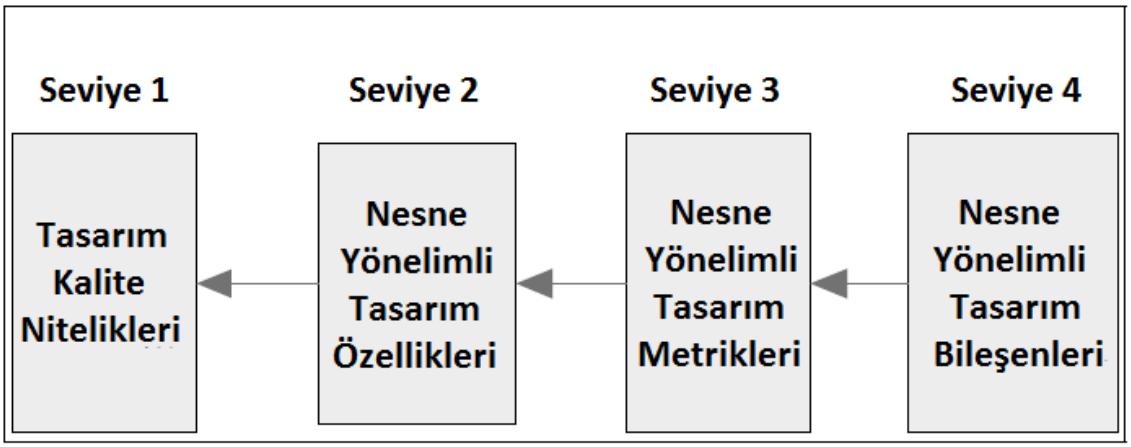
İncelen örüntü tanıma araçları arasından, PINOT yazılımı, kıyaslanan diğer yazılımlara nazaran daha geniş ölçekte tasarım örüntülerini tespit edebildiği için, bu çalışma içerisinde kullanılması uygun görülmüştür. PINOT yazılımının verdiği sonuçların ekran görüntüsü örneği Şekil 5.1‘de aşağıda gösterilmiştir.



Şekil 5.1 PINOT yazılımının sonuç ekranı görüntüsü

5.2 Yazılım Kalitesinin Ölçülmesi

Bu çalışmada yazılımın kalitesini ölçmek için Bansiya ve Davis 'in ileri sürdüğü, nesne yönelimli tasarım için kalite modeli olan QMOOD (Quality Model for Object Oriented Design) yaklaşımı kullanılmıştır. Bu yaklaşım dört farklı hiyerarşik seviyeden oluşmaktadır. Bu hiyerarşik seviyeler Şekil 5.2'de gösterilmiştir [42].



Şekil 5.2 QMOOD Hiyerarşi Seviyeleri

Dördüncü yani en alt seviyede, metotlar gibi nesne yönelimli bileşenler mevcuttur. Bu seviyenin bir üstünde bulunan üçüncü seviyede QMOOD modelinin metrikleri bulunmaktadır. QMOOD modelinin üçüncü seviyesindeki model metrikleri ve tanımlamaları Tablo 5.2 'de [42] verilmiştir.

Tablo 5.2 QMOOD Üçüncü Seviye Metrik ve Açıklamaları.

QMOOD Metrikleri	Açıklama
DSC (Design Size in Classes) : *Sınıflardaki Tasarım Boyutu*	Tasarım içerisindeki sınıfların toplam sayısıdır.
NOH (Number of Hierarchies) : *Hiyerarşi Sayısı*	Tasarım içerisindeki sınıf hiyerarşilerinin sayısıdır.
ANA (Average Number of Ancestors) : *Ortalama Ata Sayısı*	Bu metrik değeri, bilgi kalıtımı bırakan sınıfların ortalama sayısını temsil eder. Bir kalıtım yapısı içerisinde, kök (root) sınıf ile her bir sınıf arasında kalan sınıf sayılarının belirlenmesiyle hesaplanır.
DAM (Data Access Metric) : *Veri Erişim Metriği*	Sınıf içerisinde tanımlanan özel değişkenlerin (private attributes) tüm değişkenlere oranıdır. 0 ila 1 arasında değişen bu değerin olabildiğince yüksek olması istenir.
DCC (Direct Class Coupling) : *Doğrudan Sınıf Bağımlılığı*	Bir sınıf ile doğrudan ilişkili sınıfların sayısıdır. Bu metrik, değişken tanımlamaları ve metotlardaki parametre geçişleri ile doğrudan ilişkili sınıfları da kapsar.
CAM (Cohesion Among Methods of Class) : *Metotlar Arası Uyumluluk*	Bu metrik, metot parametre listeleri üzerinde temellenmiş sınıf metotları arasındaki ilişkiselliği hesaplar. Metot parametrelerinin, sınıftaki bağımsız parametre tipleri ile kesişimlerinin toplamı kullanılarak hesaplanır. 0 ila 1 arasında değişen bu değerin olabildiğince 1.0 a yakın olması istenir.
MOA (Measure of Aggregation) : *Kümeleme Ölçüsü*	Bu metrik, değişken kullanımı ile gerçekleştirilen parça bütün ilişkisi kapsamını ölçer. Bu metrik, tipleri kullanıcı tanımlı sınıflar olan, veri tanımlamalarının sayısıdır.
MFA (Measure of Functional Abstraction) : *İşlevsel Soyutlama Ölçüsü*	Bu metrik, bir sınıf tarafından kalıtılan metot sayısının, sınıfın üye metotları tarafından ulaşılabilen toplam metot sayısına oranıdır. Değeri 0 ila 1 arasında değişir.
NOP (Number of Polimorphic Methods) : *Çok Biçimli Metotların Sayısı*	Çok biçimli davranış sergileyen metotların sayısıdır.
CIS (Class Interface Size) : *Sınıf Arayüz Boyutu*	Sınıf içerisindeki genel (public) metotların sayısıdır.
NOM (Number of Methods) : *Metot Sayısı*	Bir sınıf içerisinde tanımlı bütün metotların sayısıdır.

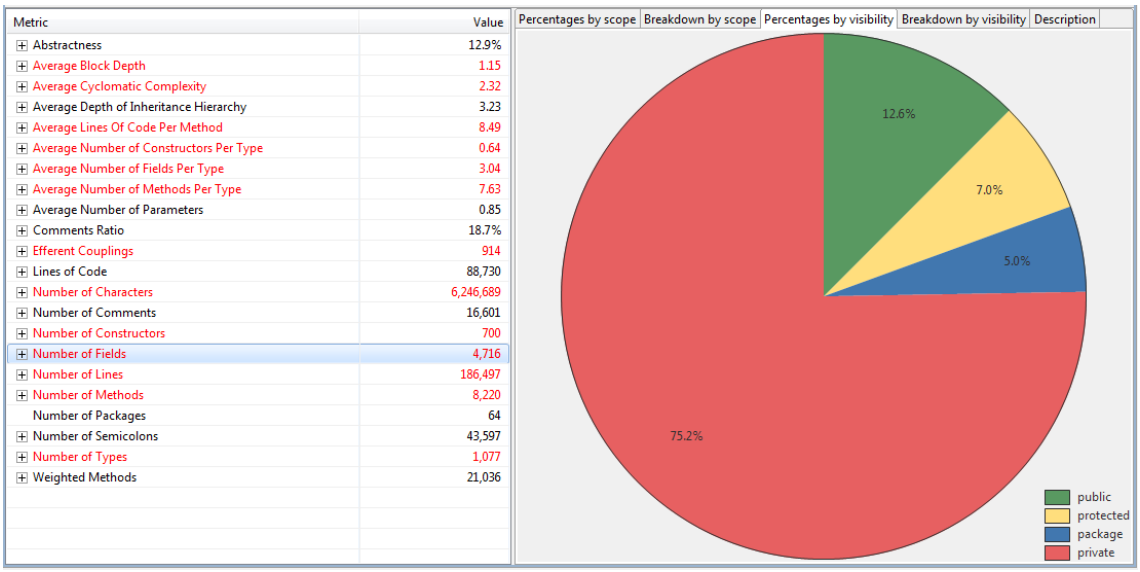
Bu QMOOD metriklerinden, Tablo 5.3 [42] içerisinde gösterilenmodelde bir üst seviyeye ait olan yazılım özellikleri hesaplanır.

Tablo 5.3 QMOOD İkinci Seviye Yazılım Özellikleri.

Yazılım Özellikleri	**Açıklama**
Tasarım Boyutu (Design Size)	Tasarımda kullanılan sınıfların miktarının ölçüsüdür.
Hiyerarşiler (Hierarchies)	Hiyerarşiler bir tasarımda farklı genelleştirme - özelleştirme kavramlarını tanımlamak için kullanılırlar. Tasarımdaki başka bir sınıftan kalıtımı olmayan ve alt sınıfları olan sınıfların sayısıdır.
Soyutlama (Abstraction)	Tasarımın genelleştirme - özelleştirme bakış açısının ölçüsüdür. Tasarımda bir veya daha fazla torun sınıfa sahip sınıflar soyutlama özelliğini gösterirler.
Sarmalama (Encapsulation)	Veri ve davranışın tek bir yapı içerisinde çevrelenmesi olarak tanımlanır. Nesne yönelimli tasarımlarda, değişken tanımlarına erişimin engellenmesi için, tasarlanan sınıfların özel (private) olarak tanımlanması bu özelliğe işaret eder ve böylece nesnenin iç temsili korunmuş olur.
Bağımlılık (Coupling)	Bir tasarımdaki bir nesnenin diğer nesneler ile olan karşılıklı bağımlılıkları olarak tanımlanır. Bir nesnenin düzgün işleyebilmesi için erişmesi gereken diğer nesnelerin sayısının ölçüsüdür.
Uyumluluk (Cohesion)	Bir sınıf içerisindeki metotların ve değişkenlerin ilişkiselliğini belirler. Metot parametrelerindeki ve değişken tiplerindeki kuvvetli çakışmalar (overlap), kuvvetli uyumluluğun belirtisidir.
Birleşim (Composition)	Nesne yönelimli tasarımda kümeleme ilişkileri olan, "parçası olma" (part-of), "sahip olma" (has), "'den oluşma" (consist-of) ya da "parça-bütün" (part-whole) ilişkilerini ölçer.
Kalıtım (Inheritance)	Sınıflar arası "bu-bir" (is-a) ilişkisinin ölçüsüdür. Bu ilişki, sınıfların kalıtım hiyerarşisi içerisinde gömülme seviyeleri ile ilişkilidir.
Çok Biçimlilik (Polymorphism)	Ara yüzleri bir diğeri ile eşleşen nesnelerin, çalışma anında birbirleriyle değiştirilebilme yeteneğidir. Nesne içerisindeki, çalışma anında dinamik olarak belirlenen servislerin ölçüsüdür.
Mesajlaşma (Messaging)	Diğer sınıflar için servis olabilecek genel (public) metotların sayısıdır. Sınıfın sağladığı servislerin ölçüsüdür.
Karmaşıklık (Complexity)	Sınıfların dâhili ve harici yapılarını ve bunların ilişkilerini anlayabilme ve kavrayabilmedeki zorluk derecesinin ölçüsüdür.

Bu çalışmada kullanılan tasarım boyutu, hiyerarşiler, soyutlama, bağımlılık, birleşim, mesajlaşma ve karmaşıklık gibi kalite özelliklerine ait metrikler*CodePro Analytix* isimli yazılım kullanılarak, Java kaynak kodu üzerinden elde edilmişlerdir.

CodePro Analytix, yazılım geliştirmede karşılaşılabilecek tipik kod kalite sorunlarını tespit edip düzeltmek amacıyla kullanılan, endüstride lider bir yazılım çözümüdür. *CodePro Analytix* çözümü, özellikle kod kalitesi, kodu yeniden gözden geçirme ve kod bağımlılık sorunları üzerinde çalışan organizasyonlar için özel olarak tasarlanmıştır. Eclipse yazılım geliştirme ortamı için geniş bir ölçekteki analiz araçlarını bünyesinde toparlamaktadır. *CodePro Analytix*, Eclipse temelli Java geliştirme ortamı için kod denetimini, ölçütleri, test üretimini, kod kapsamını ve takım çalışması özellikleri ile işlevlerini düzgün bir biçimde bir araya getirmektedir. *CodePro Analytix* yazılımının örnek ekran görüntüsü aşağıda Şekil 5.3'de gösterilmiştir.



Şekil 5.3 CodePro Analytics metriklerin sonuç ekranı görüntüsü

Bu çalışma içerisinde gerçekleştirilecek olan CodePro Analytics isimli uygulamada kullanacağımız araçlar, QMOOD modelinin üçüncü seviyesinde belirtilen metriklerden bazılarını desteklememektedir. Yine kullanacağımız uygulama kapsamındaki bazı metrikler buna benzer bir takım başka sebeplerden ötürü QMOOD metriklerinin yerine tercih edilmişlerdir. Chawla ve Chhabra 'nın çalışmalarından esinlenerek [43] QMOOD metriklerinin yerine tercih edilen bu metrikler, hangi tasarım özellikleri ile eşleştikleri de belirtilerek, tercih sebepleri ve açıklamaları ile birlikte aşağıda Tablo 5.4 [43] içerisinde gösterilmiştir.

Tablo 5.4 Tasarım Özellikleri ile Eşleşen Metrikler ve Uygulamada Kullanılacak Metrikler.

Yazılım Özellikleri	QMOOD Metrikleri	Uygulanan Metrikler	Açıklama
Tasarım Boyutu	*DSC*	*NOC*	Kullanılan NOC metriği DSC metriğine eşdeğer olup tasarımda bulunan sınıfların sayısını vermektedir.
Hiyerarşiler	*NOH*	*DIT*	Tasarım içerisindeki sınıf hiyerarşilerinin sayısını ölçen NOH metriği yerine, bir sınıftan kök sınıfa kadar olan ara sınıf sayısını ölçen DIT metriği kullanılmıştır.
Soyutlama	*ANA*	*A*	Soyutlamayı ölçmek için, bir paket içerisindeki tiplerin toplam sayısına göre ayrılmış soyut sınıfların ve arayüzlerin sayısını tespit eden A metriği kullanılmıştır.
Sarmalama	*DAM*	*1*	Seçilen uygulama aracı, bu metriği türetmek için yeterli desteğini sağlayamadığı için bu çalışmada bu metrik için "1" sabit değeri doğal değer olarak kullanılmıştır.
Bağımlılık	*DCC*	*EC*	EC metriği, incelenen paket içinde bulunan sınıfların bağımlı olduğu diğer paketlerin adedini verdiği için bağımlılık değerini temsilen kullanılabilir
Uyumluluk	*CAM*	*1*	Seçilen uygulama aracı, bu metriği türetmek için yeterli desteğini sağlayamadığı için bu çalışmada bu metrik için "1" sabit değeri doğal değer olarak kullanılmıştır.
Birleşim	*MOA*	*NOF*	Tespit edilecek ifadelerin, ilkel veri tipleri ile birlikte kullanıcı tanımlı değişkenleri de içerdiği varsayımı altında, NOF metriği birleşim özelliğini ifade edebilir.
Kalıtım	*MFA*	*1*	Seçilen uygulama aracı, bu metriği türetmek için yeterli desteğini sağlayamadığı için bu çalışmada bu metrik için "1" sabit değeri doğal değer olarak kullanılmıştır.
Çok Biçimlilik	*NOP*	*1*	Seçilen uygulama aracı, bu metriği türetmek için yeterli desteğini sağlayamadığı için bu çalışmada bu metrik için "1" sabit değeri doğal değer olarak kullanılmıştır.
Mesajlaşma	*CIS*	*NOM*	Aslında CIS metriği bir sınıf içerisindeki genel metotları içerir. Ölçüm kısıtlarından dolayı öncelikli olan CIS metriği yerine NOM metriği kullanılabilir
Karmaşıklık	*NOM*	*WMC*	WMC metriği, bir sınıf içerisindeki metotların karmaşıklıklarının toplamıdır (cyclomatic complexity). Bütün sınıflar eşit olarak ağırlıklandığında WMC metriği, sınıf içerisindeki NOM metriği ile aynı ölçüye sahiptir. Ancak karmaşıklıkların toplamı, bu metriği daha iyi resmetmektedir. Bu sebepten ötürü WMC metriği kullanım için tercih edilmiştir.

Bu hesaplanan özelliklerde de, en üst seviyede bulunan kalite nitelikleri hesaplanır. Bunlar yeniden kullanılabilirlik, esneklik, anlaşılabilirlik, işlevsellik, genişletilebilirlik ve etkililiktir.

ISO 9126 özellikleri olan işlevsellik, güvenilirlik, etkinlik, kullanılabilirlik, bakım yapılabilirlik ve taşınabilirlik, QMOOD modelinde başlangıçta seçilen kalite özellikler kümesidir. Bu özellikler kümesindeki her bir özellik, tasarım kalitesini doğru tanımlayabilme ve tüm yönleriyle başarılı bir biçimde temsil edebilme yetenekleri yönünden bireysel olarak incelenmiştir. Açıkça gözlemlenecek biçimde, tasarım aşamasından çok uygulama tarafına olan eğilimleri sebebiyle güvenilirlik ve kullanılabilirlik özellikleri bu kümeden çıkartılmıştır. Taşınabilirlik terimini ise, yazılımın uygulama esnasındaki kalitesi kapsamında incelemek daha uygun olacağından bu özellik, tasarımın karakteristiğini daha iyi bir şekilde yansıtacak olan genişletilebilirlik özelliği ile değiştirilmiştir. Buna benzer bir şekilde, etkinlik teriminin yerine, tasarımı daha iyi tanımlayacağından dolayı etkililik terimi getirilmiştir. Bakım yapılabilirlik terimi de aynı biçimde, yazılım ürününün varlığıyla alakalı bir imada bulunduğu için, tasarım karakteristiği üzerinde daha çok yoğunlaşan anlaşılabilirlik terimi ile yer değiştirilmiştir. Yazılım sistemlerinin esneklik özelliği de, önemli bir geliştirme ve son kullanıcı karakteristiğidir. Yazılım sistemlerinin bu kalite karakteristiğini gösterebilmeleri için, bu karakteristik hedefini destekleyecek yazılım mimarileri ile işbirliği içinde olmaları gerekmektedir. Dolayısıyla esneklik özelliğinin de bu tasarım modeline dâhil edilmesine karar verilmiştir. Sonuç itibarı ile QMOOD yaklaşımı için düzenlenen ve kullanılacak olan tasarım özellikleri kümesi, işlevsellik, etkililik, anlaşılabilirlik, genişletilebilirlik, yeniden kullanılabilirlik ve esneklik özelliklerinden oluşmaktadır.

Bu hesaplanan kalite niteliklerinden de son olarak bir kalite endeksi ortaya çıkarılır. Bu kalite endeksinin hesaplanma yöntemi aşağıda Tablo 5.5 [42] içerisinde gösterilmiştir.

Tablo 5.5 Kalite Niteliklerinden Kalite Endeksinin Hesaplanması.

Kalite Nitelikleri	**Endeks Hesaplama**
Yeniden Kullanılabilirlik	*-0.25 * Bağımlılık + 0.25 * Uyumluluk +0.5 * Mesajlaşma + 0.5 * Tasarım Boyutu*
Esneklik	*0.25 * Sarmalama - 0.25 * Bağımlılık + 0.5 * Birleşim + 0.5 *Çok Biçimlilik*
Anlaşılabilirlik	*-0.33 * Soyutlama + 0.33 * Sarmalama - 0.33 * Bağımlılık + 0.33 * Uyumluluk -0.33 * Çok Biçimlilik -0.33 * Karmaşıklık -0.33 * Tasarım Boyutu*
İşlevsellik	*0.12 * Uyumluluk + 0.22 * Çok Biçimlilik + 0.22 * Mesajlaşma + 0.22 * Tasarım Boyutu + 0.22 * Hiyerarşiler*
Genişletilebilirlik	*0.5 * Soyutlama - 0.5 * Bağımlılık + 0.5 * Kalıtım + 0.5 * Çok Biçimlilik*
Etkililik	*0.2 * Soyutlama + 0.2 * Sarmalama + 0.2 * Birleşim + 0.2 * Kalıtım + 0.2 * Çok Biçimlilik*

Kalite endeksinin hesaplanmasında kullanılan yazılımözelliklerinin, kalite niteliklerine olan olumlu ve olumsuz yöndeki etkileri aşağıda Tablo 5.6 ‘da gösterilmiştir.

Tablo 5.6 Yazılım Özelliklerinin Kalite Niteliklerine Etkileri

	Yeniden Kullanılabilirlik	Esneklik	Anlaşılabilirlik	İşlevsellik	Genişletilebilirlik	Etkililik
Tasarım Boyutu	⇧		⇩	⇧		
Hiyerarşiler				⇧		
Soyutlama			⇩		⇧	⇧
Sarmalama		⇧	⇧			⇧
Bağımlılık	⇩	⇩	⇩		⇩	
Uyumluluk	⇧		⇧	⇧		
Birleşim		⇧				⇧
Kalıtım					⇧	⇧
Çok Biçimlilik		⇧	⇩	⇧	⇧	⇧
Mesajlaşma	⇧			⇧		
Karmaşıklık			⇩			

5.3 Uygulamanın Gerçekleştirilmesi

Bu çalışma içerisinde gerçekleştirilen uygulama ile ulaşılmak istenen durum, zaman içerisinde evrimleşen yazılımlar içerisinde uygulanmış olan tasarım örüntülenin, yazılımın kalitesi üzerinde nasıl bir etkisinin olduğunu gözlemleyebilmektir. Bu amaca ulaşabilmek için, kamusal kullanıma izin veren lisanslara sahip, açık kaynak kodlu, Java dilinde yazılmış yazılımlar incelenmiştir. Bu yazılımların incelenmesinde kullandığımız yazılımlar da yine benzer biçimde, lisansları bakımından kamusal kullanım namına herhangi bir engel teşkil etmeyen yazılımlardır.

Uygulama çerçevesinde seçilen iki farklı yazılımın, belirli tarihsel dönem aralıklarıyla piyasaya çıkan farklı sürümleri incelenmiştir. Bu yazılımlardan birincisi, Apache Software Foundation lisansı altında olan "Apache Tomcat" uygulama sunucusudur. Bu uygulama sunucusunun sekiz farklı sürümü incelenmiştir. İncelen ikinci yazılım ise yine Apache Software Foundation lisansı altında bulunan "Apache Ant" isimli yazılım inşa aracıdır ve bu aracın da yine farklı tarihlerde piyasaya sürülmüş olan altı farklı sürümü incelenmiştir.

Öncelikle incelenen yazılımların ihtiva ettikleri tasarım örüntülerinin tespit edilmesi gerekmektedir. Bu örüntüleri tespit etmek için PINOT isimli yazılımdan faydalanılmıştır. PINOT, incelenen bu yazılımlar içerisinde bulunan tasarım örüntülerinden hangilerinin ve ne miktarda var olduklarını bize göstermektedir. Daha sonra elde edilen bu miktarlara göre, yazılım kalitesi kalitesi olgusu ile ilişkiler kurulmaya çalışılmıştır.

PINOT yazılımı ile incelenen Tomcat ve Ant yazılımlarının farklı sürümlerinin, sürüm tarihleri ile birlikte, içerdikleri tasarım örüntülerinin adetleri aşağıdaki tablolarda belirtilmektedir. Tomcat için yapılan inceleme Tablo 5.7 'de, Ant için yapılan inceleme ise Tablo 5.8 'de gösterilmiştir.

Tablo 5.7 Apache Tomcat Yazılımının İçerdiği Tasarım Örüntüleri

	Tomcat 7.0.0 (13.06.10)	***Tomcat 7.0.6 (13.01.11)***	***Tomcat 7.0.8 (04.02.11)***	***Tomcat 7.0.10 (05.03.11)***	***Tomcat 7.0.20 (09.08.11)***	***Tomcat 7.0.30 (05.09.12)***	***Tomcat 7.0.40 (09.05.13)***	***Tomcat 7.0.50 (08.01.14)***
Soyut Fabrika	2	2	2	2	0	0	0	0
Fabrika	2	2	2	2	0	0	0	0
Tek	1	1	1	1	1	0	0	0
Uyumlayıcı	2	0	0	0	0	0	0	0
Köprü	4	2	2	2	1	1	1	1
Bileşik	1	1	1	1	1	1	1	1
Dekoratör	0	0	0	0	0	0	0	0
Cephe	6	5	5	5	1	1	1	1
Sineksiklet	27	20	20	20	19	18	19	21
Vekil	0	0	0	0	0	0	0	0
Sorumluluk Zinciri	1	0	0	0	0	0	0	0
Arabulucu	3	0	0	0	0	0	0	0
Gözlemci	13	10	10	10	10	10	10	10
Durum	0	0	0	0	0	0	0	0
Strateji	16	14	14	12	12	12	12	12
Şablon	2	2	2	2	1	1	1	1
Ziyaretçi	0	0	0	0	0	0	0	0
TOPLAM	80	59	59	57	46	44	45	47

Tablo 5.8 Apache Ant Yazılımının İçerdiği Tasarım Örüntüleri

	Ant 1.8.3 *(26.02.12)*	***Ant 1.8.4*** *(22.05.12)*	***Ant 1.9.0*** *(05.03.13)*	***Ant 1.9.1*** *(15.05.13)*	***Ant 1.9.2*** *(08.07.13)*	***Ant 1.9.3*** *(23.12.13)*
Soyut Fabrika	11	11	5	5	5	5
Fabrika	18	18	6	6	6	6
Tek	12	2	0	0	0	0
Uyumlayıcı	13	14	7	7	7	7
Köprü	15	15	3	3	3	3
Bileşik	48	48	20	20	20	19
Dekoratör	7	7	3	3	3	3
Cephe	177	178	106	107	107	105
Sineksiklet	123	123	77	78	78	78
Vekil	60	60	23	23	23	23
Sorumluluk Zinciri	4	4	0	0	0	0
Arabulucu	506	506	216	217	217	217
Gözlemci	10	10	6	6	6	6
Durum	5	5	1	1	1	1
Strateji	38	38	22	22	22	20
Şablon	9	9	2	2	2	2
Ziyaretçi	1	1	0	0	0	0
TOPLAM	1047	1049	497	500	500	495

Tespit edilen tasarım örüntüleri ile ilişkisini ortaya çıkarmak için, incelenen yazılımların yazılım kalite endekslerinin bulunması gerekmektedir. Bu endeksin hesaplanabilmesi için gerekli olan metrikler CodePro Analytix yazılımı aracılığıyla elde edilmiştir. CodePro Analytix yazılımı ile incelenen Tomcat ve Ant yazılımlarının farklı sürümlerinin, sürüm tarihleri ile birlikte, sahip oldukları kalite metrikleri aşağıdaki tablolarda belirtilmektedir. Tomcat için yapılan inceleme Tablo 5.9 ‘da, Ant için yapılan inceleme ise Tablo 5.10 ‘da gösterilmiştir.

Tablo 5.9 Apache Tomcat Yazılımınından Elde Edilen Kalite Metrikleri

	Tomcat 7.0.0 (13.06.10)	***Tomcat 7.0.6 (13.01.11)***	***Tomcat 7.0.8 (04.02.11)***	***Tomcat 7.0.10 (05.03.11)***	***Tomcat 7.0.20 (09.08.11)***	***Tomcat 7.0.30 (05.09.12)***	***Tomcat 7.0.40 (09.05.13)***	***Tomcat 7.0.50 (08.01.14)***
NOC	1716	1768	1768	1774	1793	1865	1869	2051
DIT	2,50	2,51	2,51	2,51	2,53	2,53	2,53	2,50
RMA	17,4	17,3	17,3	17,3	17,4	17,3	17,5	18,8
EC	1462	1500	1497	1501	1521	1589	1588	1739
NOA	7342	7459	7476	7473	7556	7911	7908	8544
WMC	39763	40057	40112	40142	40552	42490	42705	44945
NOM	14386	14311	14315	14334	14471	15274	15314	16200

Tablo 5.10 Apache Ant Yazılımınından Elde Edilen Kalite Metrikleri

	Ant 1.8.3 (26.02.12)	***Ant 1.8.4 (22.05.12)***	***Ant 1.9.0 (05.03.13)***	***Ant 1.9.1 (15.05.13)***	***Ant 1.9.2 (08.07.13)***	***Ant 1.9.3 (23.12.13)***
NOC	1077	1078	1099	1112	1113	1117
DIT	3,23	3,23	3,22	3,22	3,22	3,22
RMA	12,9	12,8	12,9	12,9	12,9	12,9
EC	914	915	912	917	919	921
NOA	4716	4727	4899	4917	4923	4945
WMC	21036	21090	21486	21534	21564	21606
NOM	8220	8230	8411	8434	8446	8459

Ölçüm birimi kullanımında, veri analizi etki altında kalabilir. Ölçüm birimlerinin değiştirilmesi; örnek olarak uzunluk için metre kullanılırken değişiklik yapılıp inç kullanılması ya da ağırlık ölçümlerinde kilo kullanılırken bunun değiştirilip pound kullanılması, çok değişik sonuçların alınmasına sebebiyet verecektir. Genel olarak, bir özelliği küçük birimlerle ifade etmek, o özelliği daha geniş bir aralığa taşır ve dolayısıyla özellik daha yüksek miktarda bir etkiye ve ağırlığa sahip olur. Ölçüm birimlerinin seçilmesinde bağımlılıktan kaçınmaya yardımcı olması adına verinin normalleştirilmesi ya da standartlaştırılması gerekmektedir. Bu durum, verinin

dönüştürülerek örneğin [-1, 1] veya [0.0, 1.0] gibi daha küçük bir aralık içerisinde olmasını sağlar[44].

Verileri normalleştirmek, tüm özelliklerin eşit ağırlıklarda olmasını sağlar. Normalizasyon için birçok yöntem mevcuttur ancak bu çalışma içerisinde z-değeri ile normalleştirme yöntemi kullanılmıştır. Verinin normalleştirilmiş değeri, veriden, verilerin ortalamasının çıkarılması ve çıkan sonucun serinin standart sapmasına bölünmesi ile hesap edilir. Formülsel açılımı aşağıda Eşitlik 5.1' de gösterildiği gibidir.

$$X_{i,1\sigma} = \frac{X_i - \overline{X_S}}{\sigma_{X,S}} \qquad (5.1)$$

CodePro Analytix yazılımı aracılığıyla, Tomcat ve Ant yazılımlarının farklı sürümlerinden elde ettiğimiz farklı kalite metrikleri için de doğal olarak verisel anlamda bir dengesizlik söz konusudur. Bu verisel anlamdaki dengesizliği gidermek için de, z – değeri ile normalleştirme yoluna gidilmiştir. Elde edilen kalite metriklerinin veri değerlerinin, normalizasyon işleminden sonraki hali, Tomcat yazılım sürümlerinden elde edilen metrikler için yapılan normalleştirme Tablo 5.11 'de, Ant yazılım sürümlerinden elde edilen metrikler için yapılan normalleştirme ise Tablo 5.12 'de gösterilmiştir.

Tablo 5.11 Apache Tomcat Yazılımınından Elde Edilen Normalleştirilmiş Kalite Metrikleri

	Tomcat 7.0.0 *(13.06.10)*	***Tomcat 7.0.6*** *(13.01.11)*	***Tomcat 7.0.8*** *(04.02.11)*	***Tomcat 7.0.10*** *(05.03.11)*	***Tomcat 7.0.20*** *(09.08.11)*	***Tomcat 7.0.30*** *(05.09.12)*	***Tomcat 7.0.40*** *(09.05.13)*	***Tomcat 7.0.50*** *(08.01.14)*
NOC	-1,046	-0,549	-0,549	-0,4923	-0,3106	0,3775	0,4158	2,1556
DIT	-1,1538	-0,3846	-0,3846	-0,3846	1,1538	1,1538	1,1538	-1,1538
RMA	-0,2668	-0,4609	-0,4609	-0,4609	-0,2668	-0,4609	-0,0728	2,4500
EC	-0,9888	-0,5600	-0,5938	-0,5487	-0,3230	0,4443	0,4330	2,1369
NOA	-0,9651	-0,6978	-0,6680	-0,6618	-0,4298	0,6195	0,7359	1,9488
WMC	-0,8570	-0,6978	-0,6680	-0,6520	-0,4298	0,6195	0,7359	1,9488
NOM	-0,6294	-0,7368	-0,7311	-0,7039	-0,5078	0,6420	0,6992	1,9678

Tablo 5.12 Apache Ant Yazılımınından Elde Edilen Normalleştirilmiş Kalite Metrikleri

	Ant 1.8.3 *(26.02.12)*	***Ant 1.8.4*** *(22.05.12)*	***Ant 1.9.0*** *(05.03.13)*	***Ant 1.9.1*** *(15.05.13)*	***Ant 1.9.2*** *(08.07.13)*	***Ant 1.9.3*** *(23.12.13)*
NOC	-1,2433	-1,1876	-0,0184	0,7055	0,7611	0,9838
DIT	1,2984	1,2984	-0,6395	-0,6395	-0,6395	-0,6395
RMA	0,0049	-0,0196	0,0049	0,0049	0,0049	0,0049
EC	-0,7004	-0,3998	-1,3016	0,2014	0,8026	1,4038
NOA	-1,3302	-1,2245	0,4274	0,6003	0,6579	0,8692
WMC	-1,3780	-1,1662	0,3940	0,5831	0,7013	0,8668
NOM	-1,3223	-1,2322	0,3998	0,6072	0,7154	0,8326

Elde edilen normalleştirilmiş kalite metrikleri, Tablo 5.4 içerisinde belirtildiği gibi bir takım tasarım özellikleri ile eşleşmektedirler. Bu eşleşmeler üzerinden gidilerek, Tablo 5.5 'te ifade edilen Kalite Endekslerinin hesaplaması gerçekleştirilmiştir. Hesaplamalardan elde edilen kalite endeksleri aşağıdaki tanlolarda gösterildiği gibidir. Tomcat yazılım sürümlerinden elde edilen kalite endeksleri Tablo 5.13 'de, Ant yazılım sürümlerinden kalite endeksleri ise Tablo 5.14 'de gösterilmiştir.

Tablo 5.13 Apache Tomcat Yazılımınından Elde Edilen Kalite Endeksleri

Kalite Endeksi	***Tomcat 7.0.0*** *(13.06.10)*	***Tomcat 7.0.6*** *(13.01.11)*	***Tomcat 7.0.8*** *(04.02.11)*	***Tomcat 7.0.10*** *(05.03.11)*	***Tomcat 7.0.20*** *(09.08.11)*	***Tomcat 7.0.30*** *(05.09.12)*	***Tomcat 7.0.40*** *(09.05.13)*	***Tomcat 7.0.50*** *(08.01.14)*
Yeniden Kullanılabilirlik	-0,3405	-0,2529	-0,2416	-0,21093	-0,07845	0,648675	0,69925	1,777475
Esneklik	0,51465	0,5411	0,56445	0,556275	0,61585	0,948675	1,0097	1,190175
Anlaşılabilirlik	1,372338	1,078341	1,079661	1,040787	0,768966	0,006468	-0,16893	-2,53813
İşlevsellik	-0,28242	-0,02749	-0,02623	-0,00778	0,413788	0,818126	0,839136	0,993312
Genişletilebilirlik	1,361	1,04955	1,06645	1,0439	1,0281	0,5474	0,7471	1,15655
Etkililik	0,35362	0,36826	0,37422	0,37546	0,46068	0,63172	0,73262	1,47976
TOPLAM	2,978688	2,756861	2,816951	2,797712	3,208934	3,601064	3,858876	4,059142

Tablo 5.13 Apache Ant Yazılımınından Elde Edilen Kalite Endeksleri

Kalite Endeksi	***Ant 1.8.3 (26.02.12)***	***Ant 1.8.4 (22.05.12)***	***Ant 1.9.0 (05.03.13)***	***Ant 1.9.1 (15.05.13)***	***Ant 1.9.2 (08.07.13)***	***Ant 1.9.3 (23.12.13)***
Yeniden Kullanılabilirlik	-0,8577	-0,85995	0,7661	0,856	0,7876	0,80725
Esneklik	0,26	0,2377	1,2891	0,9998	0,8783	0,83365
Anlaşılabilirlik	1,424544	1,245156	0,633963	-0,16332	-0,41907	-0,74557
İşlevsellik	0,061216	0,093292	0,283218	0,488104	0,52414	0,598918
Genişletilebilirlik	1,35265	1,1901	1,65325	0,90175	0,60115	0,30055
Etkililik	-0,1311	-0,09372	0,57194	0,6411	0,66414	0,74866
TOPLAM	2,10961	1,812578	5,197571	3,723434	3,03626	2,543458

Yazılımlara ait toplam kalite endeksleri de hesaplandıktan sonra, yine bu yazılımlardan tespit edilen ve Tablo 5.7 ve Tablo 5.8 ‘de gösterilen tasarım örüntülerinin toplam sayıları ile Tablo 5.12 ve Tablo 5.13 ‘de gösterilen kalite endeksleri arasında herhangi bir ilişkinin var olup olmadığı incelenmiştir. Bu kriterler arasındaki ilişkinin varlığı sorgulanırken, iki veri seti arasındaki korelâsyon katsayıları hesap edilerek sonuca ulaşmaya çalışılmıştır.

Korelâsyon, değişkenler arasındaki ilişkisellik derecesinin ölçüsüdür. Bu çalışmada kullanılan korelâsyon tipi doğrusal korelâsyondur. Doğrusal korelâsyon iki ayrı değişken veya değer kümeleri arasında ilişkiyi, grafiksel olarak doğrusal bir biçimde temsil ettiği için bu adı almıştır. Korelâsyon, bir değişkenin kendisine eşlik eden değişkenin artış ya da azalış boyutunda nasıl değişim gösterdiğine açıklık getirir. Ölçüm ya da değişken kümeleri arasındaki korelâsyon pozitif ya da negatif yönde olabilir. Korelâsyonun pozitif olduğu durumda, bir değişken değerinin artışı, kendisiyle ilişkili olan değişken değerlerinin de arttığını, bir değişkenin azalışı ise aynı şekilde ilişkili değerlerin azaldığını gösterir. Korelâsyonun negatif çıktığı durumlarda ise, bir değişken değerinin artışı, kendisiyle ilişkili olan değişken değerlerinin azaldığını, bir değişkenin azalışı ise yine aynı şekilde ters orantıyla, ilgili değişken değerlerinin arttığını gösterir. Bunların dışında korelâsyon değerinin sıfır çıktığı durumda, iki ölçüm ya da değişken kümesi arasında bir ilişkinin var olmadığı ortaya çıkar. n‘er adet

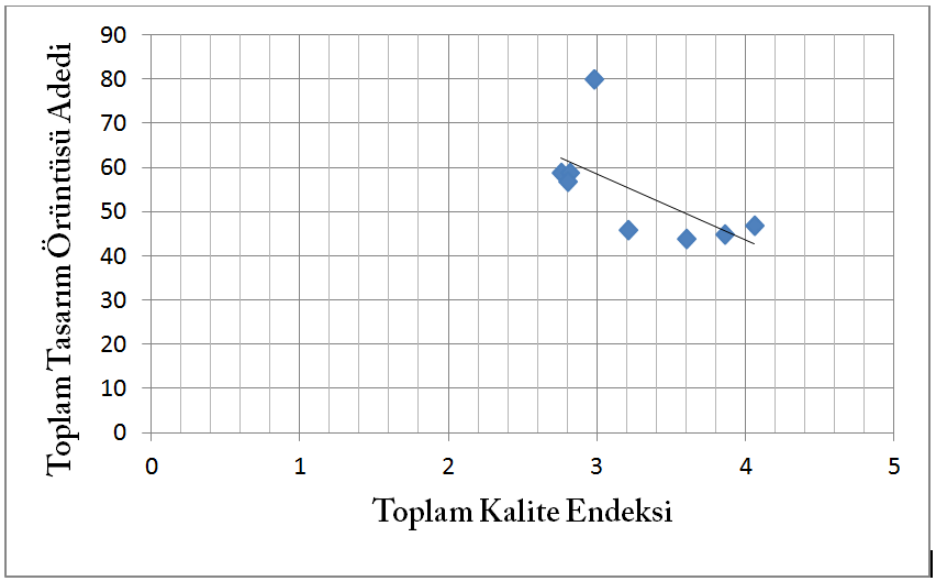
elemandan oluşan birbirinden farklı X ve Y değişken kümeleri için korelâsyon katsayısı ‘r’ aşağıda Eşitlik 5.2 ‘de gösterildiği gibi hesaplanır [45].

$$
\begin{aligned}
S_{XY} &= \sum XY - \frac{\sum X \sum Y}{n} \\
S_{XX} &= \sum X^2 - \frac{(\sum X)^2}{n} \\
S_{YY} &= \sum Y^2 - \frac{(\sum Y)^2}{n} \\
r &= \frac{S_{XY}}{\sqrt{S_{XX}.S_{YY}}}
\end{aligned}
\qquad (5.2)
$$

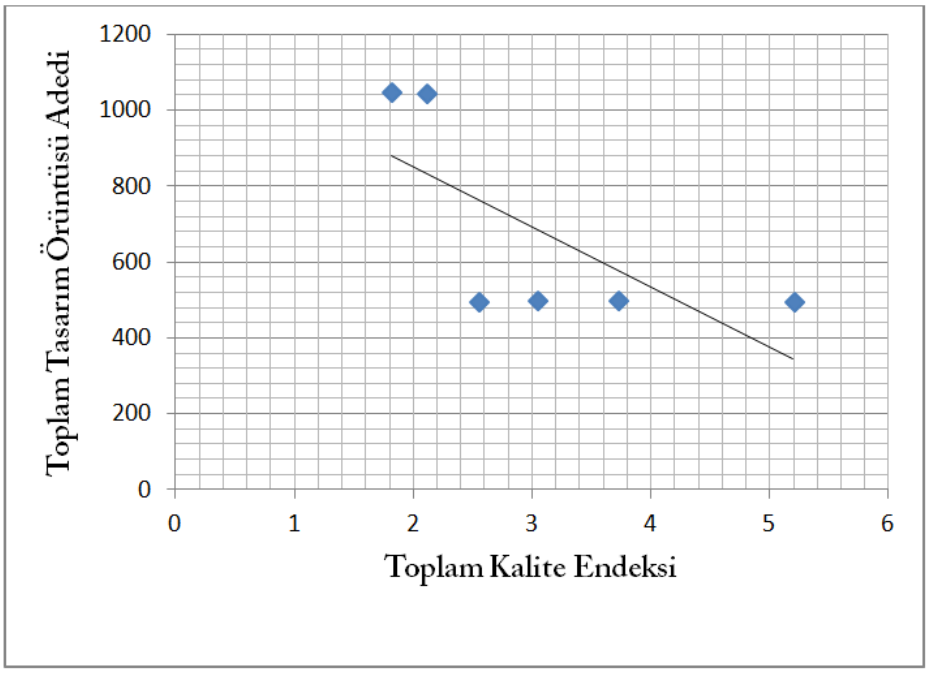
İki veri seti arasındaki korelâsyon katsayısı, ‘-1.0 ile -0.5’ veya ‘1.0 ile 0.5’ arasında hesaplanırsaçok kuvvetli bir ilişki, ‘-0.5 ile -0.3’ veya ‘0.3 ile 0.5’ arasında hesaplanırsa orta derecede kuvvetli bir ilişki, ‘-0.3 ile -0.1’ veya ‘0.1 ile 0.3 arasında hesaplanırsa zayıf bir ilişki, -0.1 ile 0.1 arsında hesaplanırsa herhangi bir ilişkinin olmadığı ya da çok zayıf bir ilişkinin var olduğu sonucu ortaya çıkacaktır.

İncelediğimiz yazılımlardan elde ettiğimiz toplam tasarım örüntüsü sayıları ile, hesap ettiğimiz toplam kalite endeksleri arasındaki korelâsyon katsayısını hesapladığımızda Apache Tomcat yazılımı için **-0,6279**, Apache Ant yazılımı için ise-**0,69064** değeri elde edilmiştir. Elde edilen bu değerlere göre her iki yazılım için de, ters yönde çok kuvvetli bir ilişkinin var olduğu ortaya çıkmaktadır. Bu demektir ki evrimleşen yazılım sürecinde, yazılım sürümlerinin içerisindeki toplam tasarım örüntülerinin sayıları azaldıkça, QMOOD kalite modeli anlayışına göre yazılım kalitesi artış göstermektedir.

Apache Tomcat ve Apache Ant yazılımlarından elde ettiğimiz araştırma sonuçlarına göre, yazılımdaki toplam tasarım örüntü sayıları ile toplam yazılım kalite endekslerinin dağılım grafikleri gösterilmiştir. Şekil 5.4 Apache Tomcat, Şekil 5.5 ise Apache Ant yazılımının ilgili dağıtım grafiğini temsil etmektedir.



Şekil 5.4 Apache Tomcat, Tasarım Örüntüsü ve Kalite Endeksi Dağıtım Grafiği



Şekil 5.5 Apache Ant, Tasarım Örüntüsü ve Kalite Endeksi Dağıtım Grafiği

BÖLÜM 6

SONUÇ

Bu çalışma kapsamında, evrimleşen yazılım süreci içerisinde, belli başlı bilinen sorunların üstesinden gelmek için kullanılan pratik çözümler olan tasarım örüntülerinin kullanım oranlarının, yazılımın kalite olgusu üzerinde nasıl bir etkiye sahip olduğunun farkına varılabilmesi adına bir araştırma gerçekleştirilmiştir. Bu araşırtmanın konusuna dâhil olan tasarım örüntüleri olgusu hakkında, olumlu görüşler olduğu kadar olumsuz görüşler de yazılım dünyasında yer almaktadır. Yazılım geliştirmede oldukça yaygın bir düzeyde kullanılan bir teknik olmalarına rağmen tasarım örüntüleri, evrimleşen yazılım sistemlerinin yapısal değişikliklerinden kaynaklanan olumsuz etkilerin engellenmesinde yetersiz kalabilmektedirler. Bu yapısal değişiklikler evrimleşen yazılım içerisinde sadece yazılım genelinde gerçekleşmeyip, hiyerarşik olarak sistem geneline göre alt düzeyde bulunan tasarım örüntülerinde de gerçekleşmektedir. Yazılım sistemlerine yeni işlevler ve özellikler eklendikçe, mevcut yapı değişerek genişlemek durumunda kalmaktadır ve bu da aslında tasarlanan mevcut yapının büyük ölçüde şekil değiştirerek, istenilen tasarım halinden farklı bir hal aldığını bize göstermektedir. Geniş ölçekte düşünüldüğünde, bu yeni işlev ve özelliklerin eklenmesiile genişleyen sistemlerle beraber tasarım örüntülerinin de sistem tasarımı içerisinde uğradığı değişiklikler, mevcut düzen ile istenilen tasarım anlayışına uyum sağlayamadıkları müddetçe,yazılım kalite olgusu üzerinde beklenildiği gibi olumlu etkileringerçekleşemeyeceğiaşikârdır.

Genişleyen sistemler içerisinde, mevcut tasarım örüntüleri de yapısal anlamda farklı biçimlerde değişme uğrayabilirler. Örneğin sistem içerisinde büyüyen veya çoğalan sistem birimlerinde mevcut olan tasarım örüntülerinin sayılarında artış gözlemlenebilir, ya da bunun tam aksi bir biçimde tasarım örüntü anlayışı, sistem hiyerarşileri içerisinde alt seviyelerden üst seviyelere taşınarak sayıca azalma gösterse de etki alanı açısındandaha geniş bir çerçevede varlığını sürdürdüğü sonucu çıkarılabilir. Farklı bir bakış açısından bakacak olursak, sistem içi yeni gereksinimleri

karşılayacak olan bir tasarım örüntüsü, sistemin mevcut yapısında bulunan tasarım örüntülerinin yerini alarak toplam tasarım örüntü sayısında bir azalmaya da sebebiyet veriyor olabilir.

Bu çalışma içerisinde gerçekleştirdiğimiz araştırmada da, benzer sonuçlar karşımıza çıkmıştır. Yazılım sistem ürünlerinin zaman içerisinde piyasaya çıkarılan farklı sürümleri üzerinde, mevcut tasarım örüntülerinin miktarları incelenmiş ve zaman içerisindekideğişimleri gözlemlenmiştir. Buna eş zamanlı olarak yine aynı sistem sürümlerinin kalite olgusu belirli prensipler altında incelenmiş ve hesap edilen kalite endeksleri ile tespit edilen tasarım örüntüleri arasındaki ilişki ortaya çıkarılmaya çalışılmıştır.

Elde edilen sonuçlar, tespit edilen toplam tasarım örüntülerinin sayısı ile incelenen kalite yaklaşımı kapsamında hesap edilen kalite endeklerinin ters yönde kuvvetli bir ilişki ortaya koyduğunu bize göstermiştir. Yani bu demektir ki, tasarım örüntülerinin toplam miktarında gözlemlediğimizazalma, yazılımın kalitesinin artışına engel teşkil eden bir durum değildir.Tasarım örüntü sayıları ve yazılım kalite olgusuna ait değişken kümeleri arasında ciddi miktarda ters yönde bir korelâsyonun ortaya çıkması, çalışma içerisinde yazılım sistemlerinin evrimleşmesi kapsamında incelenen, süregelen değişim, artan karmaşıklık, öz düzenleme gereksinimleri, yapısal ve örgütsel istikrarın korunması, süregelen büyümenin göz ardı edilmemesi ve sistem düzenine aşinalığın korunması gibi olguları ciddi bir şekilde göz önünde bulundurmamız gerektiğini bize göstermiştir.

Yazılım sistemlerinin evrimsel gelişiminin, istenilen bir biçimde yürütülebilmesi ve yazılım kalite standartlarına bağlılık çerçevesinde sürdürülebilir olması için, tasarım örüntülerinin kullanılmasının yanı sıra, yazılım evrim sürecinin çalışma kapsamında belirtilen evrimsel gereksinimlerin dikkate alınması gerekmektedir.

Yapılan çalışma bize göstermiştir ki tasarım örüntüleri, bilinen sorunlara pratik çözümler getirerek verimli ve etkili bir yazılım geliştirme sürecinin ortaya çıkmasını sağlasalar da, yazılımın kalite olgusunun karakteristiğine tek başlarına yön verebilecek yeterliliğe sahip değildirler.

KAYNAKÇA

[1] Lasater C. , Design Patterns, 2006

[2] Gamma E. , Helm R. , Johnson R. , Vlissides j. , Elements of Reusable Object-Oriented Software, 1994

[3] Cooper J. , Java Design Patterns: A Tutorial, 2000

[4] Kuchana P. , Software Architecture Design Patterns in Java, 2004

[5] Chung C. , Pro Objective-C Design Patterns for iOS, 2011

[6] Bishop J. , C# 3.0 Design Patterns, 2008

[7] Saray A. , Professional PHP Design Patterns, 2009

[8] Cade M., Roberts S. , Sun Certified Enterprise Architect for J2EE Technology Study Guide , 2002

[9] Rossberg J. , Redler R. , Pro Scalable .NET 2.0 Application Designs , 2006

[10] Horner M., Pro .NET 2.0 Code and Design Standards in C# , 2006

[11] Fischer T. , Slater J. , Stromquist P. , Wu C. , Professional Design Patterns in VB.NET: Building Adaptable Applications, 2002

[12] Gan G., Data Clustering in C++ An Object-Oriented Approach , 2011

[13] Stelting S. , Maassen O. , Applied Java Patterns, 2001

[14] Gardner H. , Manduchi G., Texts in Computational Science and Engineering , 2007

[15] Agarwal B. B. , Tayal S.P. , GUPTA M: , Software Testing & Engineering: An Introduction, 2008

[16] Homes B. , Fundamentals of Software Testing, 2012

[17] Abran A. , Khelifi A. , Suryn W. , Seffah A. , Consolidating the ISO Usability Models

[18] Fahmy S. , Haslinda N. , Roslina W. , Fariha Z. , Evaluating the Quality of Software in e-Book Using the ISO 9126 Model, 2012

[19] Immonen M., Tieto Software Product Quality Analysis Sistem , 2009

[20] Berander P. , Damm L., Damm O. , Eriksson J. , Gorschek T. , Henningsson K. , Jönsson P. , Kågström S. , Milicic D. , Mårtensson F. , Rönkkö K. , Tomaszewski P. , Software quality attributes and trade-offs , 2005

[21] Bachmann F. , Klein M., Software Architecture in Practice, Second Edition , 2008

[22] Spriestersbach A. , Springer T, Quality Attributes in mobile WebApplication Development

[23] Zhu H. , Software Design Methodology, 2005

[24] Stefani A. , Xenos M., E-Commerce System Quality Assessment using a Modelbased on ISO 9126 and Belief Networks, 2008

[25] Gorton I. , Essential Software Architecture, 2011

[26] Subrabaniam C. , Natarajan N., Software Reliability Compliance Model for Requirements Faults

[27] Bertoa M. F. Vallecillo A. , Usability Metrics for Software Components

[28] Santos Jr., Carlos Denner, Pearson, John, Kon, Fabio, Attractiveness of Free and Open Source Software Projects , 2010

[29] Al-Badareen A. B. , Selamat M. H. , Din J. , Jabar M. A. , Turaev S. , Software Quality Evaluation: User's View, 2011

[30] Panovski G., Product Software Quality , 2008

[31] Losavio F. , Chirinos L., Lévy N. , Ramdane-Cherif A. , Quality Characteristics for Software Architecture, 2003

[32] Bertoa M. F. , Vallecillo A. , Quality Attributes for Software Metamodels

[33] Bouwers E. , Correia J. P. , Deursen A. , Visser J, Quantifying the Analyzabilityof Software Architectures, 2011

[34] Lague B. , April A. , Mapping of Datrix. Software Metrics Set to ISO 9126 Maintainability Subcharacteristics

[35] Lenhard J. , Harrer S. , Wirtz G., Measuring the Installability of Service Orchestrations Using the SQuaRE Method

[36] Siakas K. V. , Georgiadou E. , PERFUMES: A Scent of Product Quality Characteristics, 2005

[37] Xie G., Chen J. , Neamtiu I. , Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software , 2009

[38] Mens T. , Demeyer S. , Software Evolution, 2008

[39] Scacchi W. , Understanding Open Source Software Evolution, 2003

[40] Godfrey M. V. , German D. M., On the Evolution of Lehman's Laws

[41] Shi N. , Olsson R. A. , Reverse Engineering of Design Patterns from Java Source Code

[42] Bansiya J. , A Hierarchical Model for Object Oriented Design Quality Assessment, 2002

[43] Chawla M. K. , Chhabra I. , Capturing Object Oriented Software Metrics To Attain Quality Attributes – A Case Study, 2013

[44] Han J., Kamber M., Pei J., Data Mining: Consepts and Techniques, 2012

[45] Mangal K., Statistics in Psycholohy and Education, 2002

[46] Budgen D., Design Patterns Magic or Myth?, 2013

[47] Izurieta C., Bieman J. m., A multiple Case Study of Design Pattern Decay, Grime,and Rot in Evolving Software Systems, 2012

[48] Schanz T. S., A Taxonomy of Modular Grime in Design Patterns, 2011

[49] Hegedus P., Ban D., Ferenc R., Gyimothy T., Myth or Reality? Analyzing the Effect of DesignPatterns on Software Maintainability, 2012

[50] Griffith I., Izurieta C., Design Pattern Decay: An Extended Taxonomy andEmpirical Study of Grime and its Impact on Design PatternEvolution, 2013

[51] Hassaine S., Evaluating Design Decay during Software Evolution, 2012

[52] Martin R. C., Design Principles andDesign Patterns, 2000

[53] Gustafsson J., Paakki J., Nenonen L., Verkamo A. I., Architecture-Centric Software Evolution by Software Metrics and DesignPatterns, 2002

[54] Hsueh N-L, Chu P-H., Chu W., A Quantitative Approach for Evaluating the Quality of Design Patterns, 2007

[55] Khomh F., Gueheneuc Y-G., An Empirical Study of Design Patterns and Software Quality, 2008

[56] Poornima U. S., Suma. V., Kumar V. H., Design Patterns as Quality Influencing Factor in Object Oriented Design Approach

[57] El-Wakil M., El-Bastawisi A., Boshra M., Object-Oriented Design Quality Models A Survey and Comparison

[58] Washizaki H., Yoshioka N., Fernandez E. B., Jurjens J., Overview of the 3rd International Workshopon Software Patterns and Quality (SPAQu'09), 2009

[59] Arora D., Khanna P., Tripathi A., Sharma S., Shukla S., Software Quality Estimation through Object OrientedDesign Metrics, 2011

[60] Panchenko O., Quality Metricsfor Maintainability of Standard Software, 2006