



**T.C.
İSTANBUL ÜNİVERSİTESİ-CERRAHPAŞA
LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ**



YÜKSEK LİSANS TEZİ

**YAZILIM TASARIM DESENLERİNİN KALİTE METRİKLERİ
YÖNÜNDEN ANALİZİ**

Yasemin YILMAZ

DANIŞMAN

Dr. Öğr. Üyesi Özgür Can TURNA

II. DANIŞMAN

Doç. Dr. Muhammed Ali AYDIN

Bilgisayar Mühendisliği Anabilim Dalı

Bilgisayar Mühendisliği Programı

İSTANBUL-2022

Bu çalışma, [7.01.2022] tarihinde aşağıdaki jüri tarafından [Bilgisayar Mühendisliği Anabilim Dalı, Bilgisayar Mühendisliği Programı]nda [Yüksek Lisans tezi] olarak kabul edilmiştir.

Tez Jürisi

[Dr. Öğr. Üyesi] [Özgür Can TURNA] (Danışman)
[İstanbul Üniversitesi-Cerrahpaşa
Mühendislik Fakültesi]

[Prof. Dr.] [Rüya ŞAMLI]
[İstanbul Üniversitesi-Cerrahpaşa
Mühendislik Fakültesi]

[Dr. Öğr. Üyesi] [Zeynep TURGUT AKGÜN]
[İstanbul Medeniyet Üniversitesi
Mühendislik ve Doğa Bilimleri]

20.04.2016 tarihli Resmi Gazete’de yayımlanan Lisansüstü Eğitim ve Öğretim Yönetmeliğinin 9/2 ve 22/2 maddeleri gereğince; Bu Lisansüstü teze, İstanbul Üniversitesi-Cerrahpaşa’nın abonesi olduğu intihal yazılım programı kullanılarak Lisansüstü Eğitim Enstitüsü’nün belirlemiş olduğu ölçütlere uygun rapor alınmıştır. |

ÖNSÖZ

Yüksek lisans öğrenimim sırasında ve tez çalışmalarım boyunca gösterdiği her türlü destek, yardım ve içtenliklerinden ötürü değerli danışman hocalarım Dr. Öğr. Üyesi Özgür Can TURNA ve Doç. Dr. Muhammed Ali AYDIN'a teşekkürü borç bilirim.

Eğitim hayatım boyunca her türlü destekleri ile yanımda olan aileme teşekkür ederim. |

Ocak 2022

Yasemin YILMAZ



İÇİNDEKİLER

Sayfa No

ÖNSÖZ	iv
İÇİNDEKİLER.....	v
ŞEKİL LİSTESİ	viii
TABLO LİSTESİ.....	ix
SİMGE VE KISALTMA LİSTESİ.....	x
ÖZET	xi
SUMMARY	xii
1. GİRİŞ	1
2. GENEL KISIMLAR.....	4
2.1 LİTERATÜR TARAMASI	5
2.2 YAZILIM KALİTE METRİKLERİ.....	8
2.2.1 SOLID	10
2.2.1.1 Tek Sorumluluk İlkesi (Single-Responsibility Principle)	11
2.2.1.2 Açık-Kapalı Prensibi (The Open-Close Principle)	11
2.2.1.3 Liskov İkame İlkesi (Liskov Substitution Principle).....	11
2.2.1.4 Arayüz Ayırma İlkesi (Interface Segregation Principle).....	11
2.2.1.5 Bağımlılık Tersine Çevirme İlkesi(Dependency Inversion Principle).....	12
2.2.2 SonarQube	12
2.2.2.1 SonarQube Kalite Metrikleri.....	12
2.2.2.2 Güvenilirlik (Reliability)	13
2.2.2.3 Güvenlik (Security).....	13
2.2.2.4 Sürdürülebilirlik (Maintainability)	13
2.2.2.5 Döngüsel Karmaşıklık (Cyclomatic Complexity).....	14
2.2.2.6 Bilişsel Karmaşıklık (Cognitive Complexity)	15
2.3 YAZILIM TASARIM DESENLERİ.....	15
2.3.1 Yaratımsal Tasarım Desenleri	17
2.3.1.1 Factory Method.....	18
2.3.1.2 Abstract Factory.....	19
2.3.1.3 Builder.....	20
2.3.1.4 Prototype	20

2.3.1.5	<i>Singleton</i>	21
2.3.2	Yapısal Tasarım Desenleri.....	22
2.3.2.1	<i>Adapter</i>	23
2.3.2.2	<i>Bridge</i>	24
2.3.2.3	<i>Composite</i>	26
2.3.2.4	<i>Decorator</i>	27
2.3.2.5	<i>Facade</i>	27
2.3.2.6	<i>Flyweight</i>	28
2.3.2.7	<i>Proxy</i>	28
2.3.3	Davranışsal Tasarım Desenleri.....	29
2.3.3.1	<i>Template Method</i>	30
2.3.3.2	<i>Chain of Responsibility</i>	31
2.3.3.3	<i>Command</i>	31
2.3.3.4	<i>Iterator</i>	32
2.3.3.5	<i>Mediator</i>	33
2.3.3.6	<i>Memento</i>	34
2.3.3.7	<i>Observer</i>	34
2.3.3.8	<i>State</i>	36
2.3.3.9	<i>Strategy</i>	38
2.3.3.10	<i>Visitor</i>	39
2.3.4	Desenler Arası İlişkiler.....	40
2.4	YAZILIM TASARIM DESENLERİ TESPİT ARAÇLARI.....	44
2.4.1	DPD (Design Pattern Detection).....	46
2.4.2	PINOT (Pattern INference recOverY Tool).....	47
2.4.3	Literatürdeki Diğer Araçlar.....	48
3.	MALZEME VE YÖNTEM.....	51
3.1	JHOTDRAW.....	51
3.2	DPD.....	52
3.3	PINOT.....	53
3.4	SONARQUBE.....	55
4.	BULGULAR.....	56
4.1	TASARIM DESENLERİ İNCELEMESİ.....	56
4.2	KALİTE METRİKLERİ İNCELEMESİ.....	58
5.	TARTIŞMA VE SONUÇ.....	59

KAYNAKLAR.....	60
EKLER	64
ÖZGEÇMİŞ	65

|



ŞEKİL LİSTESİ

	Sayfa No
Şekil 2.1: Yazılım Kalite Metrikleri.....	9
Şekil 2.2: Factory Method UML Diyagramı.....	18
Şekil 2.3: Singleton UML Diyagramı	22
Şekil 2.4: Adapter UML Diyagramı	23
Şekil 2.5: Bridge UML Diyagramı	25
Şekil 2.6: Observer UML Diyagramı.....	35
Şekil 2.7: State UML Diyagramı	37
Şekil 3.1: JHotDraw	51
Şekil 3.2: DPD Örnek Sonuç	53
Şekil 3.3: PINOT Örnek Sonuç.....	54

TABLO LİSTESİ

	Sayfa No
Tablo 2.1: GoF Tasarım Desenleri (Gamma ve diğ., 1994)	17
Tablo 4.1: JHotDraw 7.0.6 Tasarım Desenleri Analizi	56
Tablo 4.2: JHotDraw 7.6 Tasarım Desenleri Analizi	57
Tablo 4.3: JHotDraw 7.0.6 ve JHotDraw 7.6 Kalite Metrikleri Analizi	58



SİMGE VE KISALTMA LİSTESİ

Kisaltmalar	Açıklama
GoF	: Gang of Four (Dörtlü Çete)
DPD	: Design Pattern Detection (Tasarım Örüntü Algılama)
PINOT	: Pattern Inference Recovery Tool (Desen Çıkarım Kurtarma Aracı)
UML	: Unified Modelling Language (Birleşik Modelleme Dili)



ÖZET

[YÜKSEK LİSANS TEZİ]

[YAZILIM TASARIM DESENLERİNİN KALİTE METRİKLERİ YÖNÜNDEN ANALİZİ]

[Yasemin YILMAZ]

İstanbul Üniversitesi-Cerrahpaşa

Lisansüstü Eğitim Enstitüsü

[Bilgisayar Mühendisliği Anabilim Dalı]

[Danışman : Dr. Öğr. Üyesi Özgür Can TURNA

II. Danışman : Doç. Dr. Muhammed Ali AYDIN]

[Bu tez çalışmasında literatürde bilinen Gang of Four (GoF) tasarım desenleri araştırılmış ve bu tasarım desenlerinin yazılımlara uygulanmasının, yazılım kalite metrikleri yönünden analizi gerçekleştirilmiştir. Ayrıca GoF yazılım tasarım desenlerinin detaylarına ve yazılım kalite metriklerine değinilmiştir. Literatürde bilinen ve tersine mühendislik yöntemlerini kullanarak statik analiz ile yazılım tasarım desenlerini tespit eden ve yazılımın kalite metrikleri yönünden analizini yapan araçlar tanıtılmıştır. Açık kaynak kod olarak paylaşılan JHotDraw yazılımında kullanılan tasarım desenleri literatürde bilinen araçlardan DPD ve PINOT ile tespit edilmiştir. Araçlardan çıkan sonuçlar JHotDraw yazılımının farklı sürümleri için değerlendirilmiştir. Günümüzde kullanılan ve yazılımın kalite metrikleri yönünden analizinin yapılmasına yardımcı olan SonarQube aracı ile yine JHotDraw kalite metrikleri yönünden değerlendirilmiştir. Çıkan sonuçlara göre, problemlerin çözümü için yazılım tasarım desenlerinin kullanımının uygunluğu hakkında tahminlemeler yapılmıştır.]

Ocak 2022, [65] sayfa.

Anahtar kelimeler: [GoF yazılım tasarım desenleri, yazılım kalite metrikleri, fonksiyonel olmayan gereksinimler, SonarQube, DPD, PINOT]

SUMMARY

[M.Sc. THESIS]

**[ANALYSIS of SOFTWARE DESIGN PATTERNS in TERMS of QUALITY
METRICS]**

[Yasemin YILMAZ]

Istanbul University-Cerrahpasa

Institute of Graduate Studies

[Department of Computer Engineering]

Supervisor : [Assoc. Prof. Dr. [Özgür Can TURNA]

[Co-Supervisor : Assoc. Prof. Dr. Muhammed Ali AYDIN]

[In this thesis, Gang of Four (GoF) design patterns known in the literature were investigated and the application of these design patterns to software was analyzed in terms of software quality metrics. Details of GoF software design patterns and software quality metrics are mentioned. Tools that are known in the literature and that detect software design patterns with static analysis using reverse engineering methods and analyze the software in terms of quality metrics are introduced. The design patterns used in JHotDraw software, which is shared as open source code, were determined by DPD and PINOT, which are known in the literature. The results from the tools were evaluated for different versions of JHotDraw software. JHotDraw was evaluated in terms of quality metrics with the SonarQube tool, which is used today and helps to analyze the software in terms of quality metrics. According to the results, estimations were made about the appropriateness of using software design patterns for solving the problems.]

January 2022, [65] pages.

Keywords: [GoF software design patterns, software quality metrics, non-functional requirements, SonarQube, DPD, PINOT]

1. GİRİŞ

Desen kavramı ilk olarak Cristopher Alexander ve diğ. (1977) tarafından “A Pattern Language: Towns, Buildings, Construction” kitabında tanımlanmıştır. Alexander, “Her tasarım deseni, çevremizde tekrar tekrar ortaya çıkan bir sorunu tanımlar ve aynı şeyi iki kez yapmadan, bu çözümü milyonlarca kez kullanabileceğiniz şekilde sorunun çözümünün özünü tanımlar.” ifadesi ile desenlerin kullanımının önemini vurgulamaktadır. Kitap kentsel çevreyi tasarlamak için ortak bir dili tanımlasa da yazılım tasarım desenlerinin kullanımı yaklaşımı aynı şekilde ifade edilebilir. Yazılım tasarım desenleri kavramı ise ilk olarak, 1994 yılında, Gamma ve diğ. (1994) tarafından yazılan “Design Patterns - Elements of Reusable Object Oriented Software” kitabında ortaya atılmıştır. Gang of Four (GoF) tasarım desenleri olarak tanımlanan desenlerin kullanımı ve bu tasarım desenlerinin birbirleri arasındaki ilişki bu kitapta açıkça anlatılmıştır. GoF tasarım desenleri, yaratımsal tasarım desenleri, yapısal tasarım desenleri ve davranışsal tasarım desenleri olmak üzere üç ana başlık altına toplanmıştır. Bu desenler nesne yönelimli yazılım geliştirme alanında, nesnedeki belirli sorunlara basit ve iyi tasarlanmış çözümleri tanımlayan 23 tasarım deseninden oluşan bir yapı sunar.

Yazılım tasarım desenleri, yazılımların tasarlanmasında yaygın olarak ortaya çıkan sorunlara getirilen çözümlerdir. Kodun içerisinde yinelenen bir tasarım problemini çözmek ve bu sorunlarla karşılaşmamak için geliştirilmiş yol haritalarıdır. Yazılım geliştirme sürecinde önemli bir rol oynar. Nesne yönelimli yazılımdan tasarım deseni örneklerini çıkarmaya olan ilgi, son yirmi yılda oldukça artmıştır. Tasarım desenlerinin kullanımı, programın anlaşılmasını kolaylaştırır, sistemleri belgelemeye yardımcı olur ve tasarım değişimlerini yakalar (Al-Obeidallah, Petridis ve Kapetanakis, 2016). Bir tasarım deseni, farklı durumlarda kullanılabilen, bir problemin nasıl çözüleceğine ilişkin bir açıklama veya şablondur. Problemin çözümünün özünü, bu çözümün aynı şekilde iki kez yapılmadan defalarca kullanılabileceği şekilde tanımlamaktadır (Gamma ve diğ., 1994). Yazılım tasarım desenleri, doğrudan kodda kullanılabilen bir çözüm sağlamaz, bunun yerine bir sistemi tasarlarken izlenecek yaklaşımı ifade eder. Bu desenlerin bilgisi, bir geliştiricinin yalnızca bugünün gereksinimlerini karşılamakla kalmayıp aynı zamanda gelecekteki hedeflere de hitap edebilen daha esnek ve ölçeklenebilir bir sistem oluşturmaya yardımcı olur. Yazılım tasarım desenleri, sık karşılaşılan sorunlara karşı oluşturulan bu sorunların çözümündeki yeterliliği test edilen ve

sorunların çözümünde kullanılması önerilebilecek yeterliliğe sahip araç setidir. Belirtilen sorunlarla karşılaşılmasa bile yazılım tasarım desenlerini bilmek faydalıdır. Çünkü tasarım desenleri her türlü problemi nesne yönelimli tasarım ilkelerini kullanarak nasıl çözüleceğini öğretir (Gahlyan ve Singh, 2018).

Yazılım tasarım desenleri, ayrıntı düzeyleri, karmaşık yapılara uygulanabilirliği ve tasarlanan tüm sisteme uygulanabilirlik ölçeğine göre farklılık gösterir. Nesne yönelimli yazılımın yapısı, miras ve kompozisyon özelliklerinin yanı sıra sınıflar, yöntemler ve paketler olarak adlandırılan bir dizi tasarım varlığını içerir. Çevik yazılım geliştirmede, yazılım yapılarının temel yapısına yeni özellikler eklenebilmesi veya mevcut işlevselliği değiştirerek zamanın geçmesiyle yazılımın değiştirilebilir olması son derece önemlidir. Bu yaklaşım, yeniden kullanılabilirlik, sürdürülebilirlik, anlaşılabilirlik gibi sistem düzeyinde kalite özelliklerini etkiler ve iyileştirmeye yardımcı olabilir (Ampatzoglou ve diğ., 2011, 2015; Hussain ve diğ., 2017). Yazılım sistemlerinin kalitesinin artırılmasına yönelik iki yaygın pratik yaklaşım vardır. Birinci yaklaşım, yazılımın dış davranışında (Fowler, 1999; Neill, 2003) herhangi bir değişiklik olmaksızın iç yapısını değiştirerek bir sistemin tasarım kalitesini iyileştirmenin kesin bir yolu olarak kabul edilen yazılım yeniden düzenlenmesidir. İkinci yaklaşım ise tasarım desenlerinin kullanılmasıdır. GoF tasarım desenlerinin ortaya çıkması, bir sistemin karmaşıklık, uyum ve birleştirme gibi tasarım özelliklerini geliştirmiştir (Ampatzoglou ve Chatzigeorgiou, 2007).

Program verimliliği ve geliştirmenin üretkenliği, doğru desenlerin uygulanması ile %25-30 artmıştır, ancak bu tamamen geliştiricilerin becerilerine ve uzmanlık düzeyine bağlıdır (Riehle, 2011). Desenleri uygulamanın temel amacı iyi uygulama, iletişim ve dokümantasyondur (Stencel ve Wegrzynowicz, 2008). Tasarım bir deseni ve varyasyonları, hataya açık alanı tanımayı kolaylaştırır ve tersine mühendisliğe yardımcı olan otomatik algoritmalar yoluyla bilgi çıkarmada önemli bir rol oynar. Kaynak koddan tasarım örüntülerinin tespiti en zor görevlerden biridir çünkü tek bir desen, geliştiricinin geliştirme sırasında aynı amaçla uyguladığı farklı tekniklere, metodolojilere ve tasarımlara sahip olabilir (Gamma ve diğ., 1994) ve bazen geliştirici, kullanıcı ihtiyaçlarına göre çözümü doğru üretmek için birden fazla desen kullanabilir.

Tasarım desenleri genellikle algoritmalar ile karıştırılan bir kavramdır. İki kavram da problemlere çözüm getiren yaklaşımlardır. Ancak algoritmalar problemlerin çözümünü

sağlayan adımları net bir şekilde tanımlarken, tasarım desenleri bir çözümün daha üst düzey açıklamasıdır. İki farklı programa uygulanan aynı tasarım deseninin kodu farklı olabilir.

Bu tez çalışmasının organizasyonu şu şekildedir:

Genel Kısımlar bölümünde literatürdeki benzer çalışmalar incelenip sunulmuş, GoF yazılım tasarım desenlerinin detayları ve yazılım kalite metrikleri anlatılmış, tersine mühendislik yöntemleri kullanılarak statik analiz ile yazılımda kullanılan yazılım tasarım desenlerini belirleyen ve yazılımın kalite metrikleri yönünden analizini yapan araçlar tanıtılmıştır.

Malzeme ve Yöntem bölümünde anlatılan tersine mühendislik yöntemlerini kullanarak nesneye yönelik yazılımlarda hangi desenlerin kullanıldığı sonucunu veren araçlardan çıkan sonuçların karşılaştırılması verilmiştir.

Bulgular bölümünde araçlardan çıkan sonuçların değerlendirmesi ve kullanılan yazılım tasarım desenlerinin, yazılım kalite metrikleri yönünden değerlendirilmesi verilmiştir.

Tartışma ve Sonuç bölümünde ise yazılım tasarım desenlerinin kullanımının kalite metrikleri yönünden değerlendirmesi sunulmuştur.

|

2. GENEL KISIMLAR

Yazılım mühendisliği alanındaki desen terimi, yaygın olarak ortaya çıkan sorunların çözümlerini ifade eder; buna göre farklı geliştirme aşamaları ve uygulama alanları için farklı yazılım tasarım desenleri yazılmıştır. En çok kullanılan desenlerden bazıları, nesneye yönelik tasarım desenleri, mimarî desenler ve analiz desenleridir. Bu çalışmada 90'lı yılların ortalarında Gamma ve diğ. (1994) tarafından tanıtılan ve 23 yaygın nesne yönelimli soruna çözümler sunan tasarım desenlerinin analizi ve bu desenlerin kalite metrikleri yönünden değerlendirilmesi sunulacaktır. Bu desenler aynı zamanda GoF tasarım desenleri olarak da bilinir.

Yazılım Mühendisliğinde, ortamlarının ve kullanıcılarının ihtiyaçlarını karşılayan yazılım sistemleri üretmek için sürekli bir arayış vardır. Yazılım başarısı, büyük ölçüde tasarım sürecinde kullanıcıların gereksinimlerinin ne kadar iyi anlaşıldığına ve uygun işlemlere dönüştürüldüğüne bağlıdır. Yazılım projesi başarısızlıkları, şirketin imajını, müşterinin sadakatini olumsuz etkilediğinden ve müşterilerin memnuniyetini azaltabildiğinden günümüzün rekabetçi pazarında genellikle ekstra maliyet ve risk oluşturur (Hussain, Mkpojiogu ve Kamal, 2016). Yazılımdaki gereksinimler, analiz, şartname ve doğrulama gereksinim mühendisliği sürecinde tanımlanır. Gereksinim analizi, bir yazılım sistemi geliştirmek için gereksinim mühendisliği sürecindeki en karmaşık adımlardan biri olarak düşünülebilir. Analizden sonraki aşamada yazılım geliştirme sürecinde ihtiyaçlara karşı doğru yöntemi belirlemek oldukça önem arz etmektedir. Yazılım geliştirme döngüsünde yazılım sistemlerinin tasarım aşaması büyük önem kazanmaktadır. Genel olarak, tasarım aşamasının adımları doğru bir şekilde yürütülürse, yazılım geliştirmenin diğer yönleri üzerinde önemli bir etkisi olur. Bu nedenle yazılımın kalitesi, tasarım aşamasında doğru kararlar verilerek garanti edilmelidir (Hasheminejad ve Jalili, 2012; Sabatucci, Cossentino ve Susi, 2015). Şimdiye kadar, yazılım geliştirme uzmanları, yazılım tasarım aşamasında yaygın sorunlara birçok çözüm sunmuştur (Smith, 2011). Bu çözümler tasarım desenleri olarak bilinir. Tasarım desenleri, deneyimsiz yazılım tasarımcılarına yardımcı olmanın yanı sıra esnekliği ve yeniden kullanılabilirliği artırmaktadır (Zdun, 2007). Tasarım desenleri ayrıca yazılım uygulama ve bakım maliyetlerini azaltır (Zdun, 2007; Hasheminejad ve Jalili, 2012). Deneyimsiz tasarımcılar genellikle tasarım problemlerini çözmek için uygun kalıpları seçmekle ilgilenirler. Çok sayıda tasarım deseni olması nedeniyle, uygun olanı seçmek deneyimli tasarımcılar için de bir zorluktur (Intakosum

ve Muangon, 1970). Yazılım gereksinimlerine ve kullanıcı ihtiyaçlarına göre uygun tasarım desenlerini seçmek, kullanılan tasarım desenlerinin önündeki ana zorluklardan biridir.

Tasarım deseninin amacı, özellikle tasarımdaki problemleri çözmek, esnek ve tekrar kullanılabilir nesne yönelimli yazılımlar üretmektir. Tasarım desenleri, tasarımı yeniden kullanmak için doğrulanmış bir teknoloji olarak kullanılır ve sistem bakımını veya dokümantasyonunu iyileştirir. GoF tasarım desenleri yaratımsal, yapısal ve davranışsal olmak üzere üç ana başlık altında incelenmiştir. Yaratımsal tasarım desenleri, mevcut kodun esnekliğini ve yeniden kullanımını artıran nesne oluşturma mekanizması sağlar. Yapısal tasarım desenleri, yapıları esnek ve verimli tutarken nesnelerin ve sınıfların nasıl daha büyük yapılar halinde birleştirebileceğini açıklar. Davranışsal tasarım desenleri ise etkili iletişimi ve nesneler arasındaki sorumlulukların atanmasını sağlar (Gamma ve diğ., 1994).

2.1 LİTERATÜR TARAMASI

Bu bölümde literatürde yapılan çalışmalardan, tasarım desenlerinin sistematik bir incelemesini yapan veya desenlerin yazılım kalite metrikleri üzerindeki etkisini kataloglayan bazı önceki çalışmalar ele alınmıştır.

Ampatzoglou ve diğ. (2015) yaptıkları çalışmada, yazılım mühendisliği araştırma topluluğunun hem akademi hem de endüstrideki GoF tasarım desenlerine olan ilgisini bildirmişlerdir. 130'dan fazla bilimsel makale üzerinde yaptıkları haritalama çalışmalarında, GoF tasarım desenlerinin yazılım kalitesi özellikleri üzerindeki etkisinin, desen formülasyonları ve desen tespiti ile karşılaştırıldığında, önemli araştırma konusu olmaya devam ettiğini bildirmişlerdir. Benzer şekilde, haritalama çalışmalarında, Zhang ve Budgen (2013), bakım için bir çerçeve sağlamada desenlerin kullanımının yararlılığını bildirmişler ve araştırmacıların, etkiyi tanımlamak için kilit desenlere odaklanan vaka çalışmalarını kullanmalarını tavsiye etmişlerdir.

Ampatzoglou ve diğ. (2011), bileşen tabanlı açık kaynak yazılım projeleri için bir sınıf, desen veya paket olan ve yeniden kullanılabilir birimin hangisi olduğunu araştırmak için çok projeli bir vaka çalışması gerçekleştirir. Sonra yazarlar, desen tabanlı bileşenler olarak yeniden kullanılabilir bir çalışmaya yirmi üç bin sınıfı dahil etmişlerdir. Ayrıca, her durum için yazarlar; sınıfı yeniden kullanma, sınıfın ait olduğu kalıbı yeniden kullanma, sınıfın dahil olduğu paketi yeniden kullanma, desene katılan ve en az bir sınıf içeren tüm paketleri yeniden kullanma alternatiflerini araştırmışlardır. Son olarak, yazarlar, tasarım desenlerinin alternatif

yeniden kullanımının, diğer alternatiflere kıyasla en uygun seçeneğini sunduğu sonucuna varmışlardır. Benzer şekilde, Ampatzoglou ve diğ. (2015) başka bir çalışmada, nesne yönelimli uygulamalarda GoF tasarım desenlerinin sınıfların kararlılığı üzerindeki etkisini ampirik olarak araştırmış, altmış beş bin açık kaynaklı Java sınıfıyla çok projeli bir vaka çalışması yürütmüş ve diğer sınıflarda meydana gelen değişikliklerin yayılması nedeniyle bir sınıftaki değişim olasılığını araştırmışlardır. Çalışmalarının sonuçları, GoF tasarım desenlerinde tek rolü oynayan sınıfların, GoF tasarım deseni oluşumlarında, birden fazla rol oynayan sınıflardan daha kararlı olduğunu göstermektedir. Ayrıca, çalışmada sonuçlar, farklı GoF tasarım desenlerinin, onlara katılan sınıfların farklı kararlılık düzeylerini sağladığını göstermektedir.

Vokac ve diğ. (2004) ve Prechelt ve diğ. (2001) deneysel çalışmasını yinelemiş ve yazılım tasarım desenlerini kullanmanın yazılım bakımı açısından yararlılığını araştırmışlardır. Her iki çalışmada da yazarlar, tasarım desenli ve tasarımsız sistemlerin sürdürülebilirliğini karşılaştırmışlar ve profesyonellerden veri toplamak için aynı anketi kullanmışlardır. Ayrıca yazarlar, Abstract Factory, Composite, Observer, Visitor ve Decorator GoF tasarım desenlerini dikkate almış ve özellikle Visitor ve Observer desenlerinde farklı sonuçlar bildirmişlerdir. Son olarak, her iki çalışmanın deneysel sonuçları, tasarım desenlerinin etkisinin bakım açısından yararlı olduğunu göstermektedir.

GoF tasarım desenleri üzerine diğer bir çalışma, yazarların bir anket aracılığıyla tüm GoF desenlerinin yazılım kalitesi nitelikleri üzerindeki etkisini değerlendirdiği Khomh ve Gueheneuc (2008) tarafından yapılmıştır. Sonuçlar, ortak inançların aksine, uygulamadaki tasarım desenlerinin birkaç kalite özelliğini olumsuz etkilediğini ortaya koymuştur.

Zhang ve Budgen (2012), 2009 yılına kadar yayınlanan makaleler üzerinde yazılım tasarım desenlerinin etkinliği üzerine kullanılan araştırma yöntemi, tasarım desenleri ve yazılım kalitesi nitelikleriyle ilgili deneysel çalışmalar hakkında sistematik bir literatür taraması yapmıştır.

Diğer bir çalışmada ise (Ampatzoglou, Charalampidou ve Stamelos, 2013) GoF tasarım desenlerine ilişkin araştırma çabalarına genel bir bakış sağlamak için yaklaşık yüz yirmi çalışmanın haritalama çalışmasının sonuçlarını sunmaktadır. Bu çalışmanın araştırma soruları, tasarım deseni araştırmasının alt başlıklarda daha fazla kategorize edilip edilemeyeceği, tasarım deseninin anlaşılabilirliği, yeniden kullanılabilirliği, modülerliği, sağlamlığı ve tasarım desenlerinin birleştirilmesi alt konularından hangilerinin en aktif olanlar olduğu ve GoF

desenlerinin yazılım üzerindeki bildirilen kalite metriklerinin etkisinin ne olduğu ile ilgilidir. Sonuçlar, tasarım desenleri üzerindeki araştırmanın devam etmesi gerektiği ve tasarım desenlerinde, anlaşılabilirlik, yeniden kullanılabilirlik, modülerlik, sağlamlık ve tasarım desenlerinin birlikte kullanımlarının araştırılmaya devam edilmesi gereken konular olduğunu göstermiştir.

Rahmati ve diğ. (2019) yaptığı çalışmada GoF tasarım desenlerini seçmek için yeni bir yöntem sunulmaktadır. Önerilen yöntem, vektör uzayı modeline (VSM-Vector Space Model) dayalı olarak uygulanmaktadır. Bu yöntemde, Terim Frekans-Ters Belge Frekansı (TF-IDF - Term Frequency — Inverse Document Frequency) ağırlıklandırma algoritması, iki metin arasındaki benzerliği daha doğru bir şekilde belirlemek için geliştirilmiştir. Ayrıca, ağırlıklandırmada kelimelerin alt sözcüklerini ve eş anlamlılarını kullanmışlardır. Önerilen yöntemi 23 tasarım deseni, 29 nesneye yönelik tasarım problemi ve dokuz gerçek dünya problemi ile değerlendirmişler ve sonuçlarda diğer tasarım deseni tespit etme yöntemlerine göre iyileşme olduğunu gözlemlemişlerdir.

Nikolevea ve diğ. (2019), program kodunda tersine mühendislik yöntemini uygulayarak kaynak kodda desenlerin algılanması üzerinde yaptıkları çalışmalarını sunmuşlardır. Çalışmada C# kaynak kodunda tasarım desenlerinden en yaygın kullanımı olan Singleton tasarım deseni algılamak için basit bir yaklaşım sunmuşlardır. Gelecek çalışmalarında diğer tasarım desenlerinin de algılanması üzerinde çalışmalarını sürdürdüklerini belirtmişlerdir.

Hussain ve diğ. (2017), GoF tasarım desenlerinin sık kullanımının tasarım kalitesi nitelikleri üzerindeki etkisini ampirik olarak araştırmışlardır. Tasarım deseni kullanımı ile tasarım kalitesi nitelikleri arasında bir korelasyon olup olmadığını, sistem büyüklüğünün korelasyon üzerindeki etkisi ve kullanılan tasarım deseni örneklerinin sayısındaki değişikliğin, bir sistemin sonraki sürümlerinde tasarım kalitesini nasıl etkilediği üzerinde çalışmışlardır. Yeniden kullanılabilirlik, esneklik ve anlaşılabilirliği, kullanılan Template, Adapter-Command, Singleton ve State-Strategy tasarım desenleriyle önemli bir ilişkiye sahip olduğunu, ancak sistem boyutunun karıştırıcı etkisinden etkilendiğini göstermektedir. Daha sonra, velocity adlı açık kaynaklı bir projenin sonraki sürümlerinde, Singleton, Adapter-Command ve State-Strategy tasarım desenlerinin kullanım yoğunluğunun yeniden kullanılabilirlik ve esneklik özellikleri açısından tasarım kalitesini iyileştirebileceğini gözlemlemişlerdir.

Elish ve Mawal (2015), tasarım desenlerinin kullanımını ve yazılım kalitesi üzerindeki etkisini araştırmak için deneysel ve karşılaştırmalı bir çalışma yürütmüşlerdir. Bu çalışmanın amacı, tasarım, kategori, motif ve rol düzeyinde nesne yönelimli sistemdeki tasarım örüntülerindeki hata yoğunluğunu nicel olarak ölçmek ve karşılaştırmaktır. Katılımcı ve katılımcı olmayan sınıflara kıyasla yazar, bir tasarım motifinde katılımcı ve katılımcı olmayan sınıflar arasındaki hata yoğunluğu farkına yönelik net bir eğilim gözlemlenmemektedir. Benzer şekilde, farklı tasarım motifi kategorilerinde katılımcı sınıflarının hata yoğunluğu karşılaştırıldığında, yapısal tasarım motifine katılan sınıfların diğer kategorilere göre daha az hata yoğunluğuna sahip olduğunu gözlemlemiştir. Son olarak, tasarım motifindeki katılımcı sınıflarının hata yoğunluğunu karşılaştıran yazarlar, Builder, Adapter, Composite, Decorator ve Factory Method gibi tasarım desenlerinin önemli farklılıklar gösterdiğini gözlemlemiştir.

Feitosa ve diğ. (2019), desenlerin varlığının iyi kodlama uygulamalarına uyumunu incelemişlerdir. Bu hedefe ulaşmak için, GoF tasarım desenlerinin varlığı ile kaynak kodun doğruluğu, performansı ve güvenliği ile ilgili iyi uygulamaların ihlalleri arasındaki ilişkiyi statik analiz yoluyla araştırmışlardır. Sonuç olarak desenlere katılmayan sınıfların doğruluk, performans ve güvenlik için iyi kodlama uygulamalarını ihlal etme olasılığının daha yüksek olduğunu göstermektedir. Daha ayrıntılı bir analiz düzeyinde, belirli desenlere odaklanarak, daha karmaşık yapıya sahip desenlerin (örneğin, Decorator) ve değişime daha yatkın olan kalıp rollerinin (örneğin, Alt sınıflar) daha yüksek sayıda ihlalin muhtemel olduğunu gözlemlemiştir.

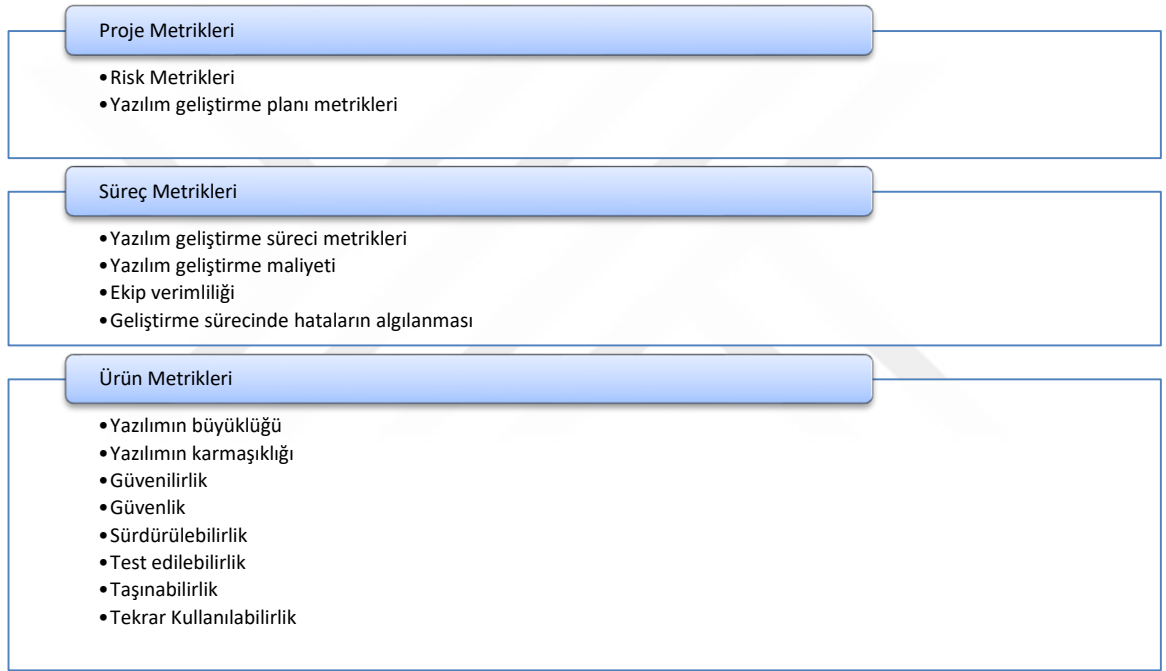
2.2 YAZILIM KALİTE METRİKLERİ

Yazılım kalitesine olan talep, endüstriyel alanda kullanılan yazılımlarda giderek artmaktadır. Bir hatanın yapılması, yazılımın gelişip büyümesi ve karmaşıklığının artmasından sonra daha fazla hataya neden olabilir. Bu nedenle, okunabilirliği ve bakımı sağlamak için kod karmaşıklığının daha düşük olması gerekir (Moon, Park ve Kim, 2016).

Yazılım kalitesi, bazı kalite faktörlerinin toplamı olarak görülebilir. Proje ne kadar çok faktörü kapsıyorsa o kadar başarılı olarak sınıflandırılabilir. Dikkate alınan faktörler projeden projeye değişebilir.

Yazılım kalite metrikleri üç ana başlık altında incelenmektedir. Bunlar; ürün metrikleri, süreç metrikleri ve proje metrikleridir (Belachew, Gobena ve Nigatu, 2018). Proje metrikleri,

potansiyel riskleri ölçmek ve istikrarlı bir plan yapmak için projenin erken aşamalarında kullanılır. Bu tür metrikler önceden yapılan benzer projelerdeki deneyimler baz alınarak proje durumunu ve maliyetini ölçümlemek için kullanılır. Süreç metrikleri, projenin gelişim sürecini gözlemlemek için kullanılır. Yazılımı programlarken, geliştirilmekte olan projede ya da geçmiş projelerdeki yapılan hatalar hakkında bilgi almak ve gelecekteki sorunları tahmin etmek için sonuçlar verebilir. Bu metrikler ekip verimliliği değerlendirilmesinde de kullanılır. Ürün metrikleri, en kapsamlı metriktir. Geliştirilen ürünün kendisini ele alır ve geliştirme aşamasının sonunda test edilirler (Mladenova, 2020).



Şekil 2.1: Yazılım Kalite Metrikleri

Tasarım desenlerinin tanıtımı doğrudan yazılım kalite ihlallerini ortadan kaldırmayı hedeflemese bile, daha iyi kodlama uygulamaları eşliğinde "daha temiz" ve iyi tasarlanmış bir mimariye sahip yazılımların tasarlanmasını sağlar.

Statik analiz, programı çalıştırmadan uygulama kodunu incelemek için kullanılır. Yazılım geliştirmenin ilk aşamalarında koddaki birçok kusuru belirlemek için kullanılır. Kodda daha fazla kusurun keşfedilmesine ve daha az yanlış uyarı yapılmasına yardımcı olan birçok statik analiz aracı mevcuttur. Yazılımın canlıya alınmasından önce kod kalitesini iyileştirmek ve hataları düzeltmek için yazılımı analiz etmek için statik analiz yaklaşımı kullanılır. Kaynak kod üzerinde statik analiz, kod incelemesi statik analiz araçları kullanılarak gerçekleştirilir. Yazılım

metrikleri, daha sürdürülebilir, yeniden kullanılabilir ve test edilebilir hale getirmek için bu teknikler kullanılarak elde edilebilir. Geliştirme sürecinin erken aşamalarında kusurları tespit etmek için çok iyi bir yaklaşımdır ve hatasız ve bakımı yapılabilir yazılım geliştirmede daha güvenilir ve verimli olduğunu kanıtlamıştır (Farooq, 2017).

Kodun manuel olarak gözden geçirilmesi zaman alan bir süreçtir ve kodu manuel olarak analiz etmek için programcılar, yazılım geliştirme sırasında kodda ne tür hataları teşhis etmeleri gerektiğini bilmelidir. Bu nedenle, statik analiz araçlarını insan denetimi ile birlikte kullanmak, kodu analiz etmenin ve statik analizden elde edilen sonuçlara dayalı olarak kod üzerinde iyileştirme gerçekleştirmenin çok verimli bir yoludur. Kod incelemesi ve analizi, yazılım geliştirme sürecinin herhangi bir aşamasında yapılmalıdır, ancak bunu erken aşamalarda yapmak daha iyidir. Yazılım geliştirme yaşam döngüsünün sonraki aşamalarında koddaki kusurları gidermek maliyetli hale gelir.

Bunların tümü, yazılım geliştirme yaşam döngüsünün sonraki aşamalarında sorun yaratan düşük kaliteli yazılımlara neden olur. Bakımı yetersiz kod, yazılım bakım maliyetinin artmasına neden olur. Yazılım tasarımıyla ilgili sorunların testlerle tespit edilmesi zordur. Sorunların kaynağı, yazılımın gereksinimleri ve tasarımında yatmaktadır (Gomes ve diğ., 2009). Tasarım yaparken yazılımda sonradan çıkabilecek problemlere karşı sağlanması gereken kriterler bulunmaktadır. Bunlardan SOLID kaliteli bir tasarım yapmak için beş ilke belirlemiştir. Sonraki bölümde bu beş ilke açıklanacaktır.

2.2.1 SOLID

Uygulama ve test aşamasındaki hataların sayısını azaltmak için kaliteli bir yazılım tasarımına ihtiyaç vardır. Kaliteli bir tasarım için bazı metriklerin sağlanması gerekir. SOLID beş ilkenin kısaltmasından oluşur. Bunlar; Tek Sorumluluk İlkesi (Single-Responsibility Principle), Açık-Kapalı Prensipleri (Open-Closed Principle), Liskov İfade İlkesi (Liskov Substitution Principle), Arayüz Ayırma İlkesi (Interface Segregation Principle), Bağımlılık Tersine Çevirme İlkesi (Dependency Inversion Principle)dir. SOLID tasarım ilkeleri; anlaşılabilirlik, esneklik, sürdürülebilirlik ve test edilebilirlik gibi yazılım kalite faktörlerini karşılamak için nesne yönelimli bir tasarım kılavuzu olarak kullanılmıştır (Oktafiani and Hendradjaya, 2018). SOLID tasarım ilkeleri, yazılım tasarımı kalite sorununun yönetimine izin verir. Bu ilkeler dizisi, kod karmaşıklığını azaltmaya, okunabilirliği, genişletilebilirliği, sürdürülebilirliği iyileştirmeye,

hataları azaltmaya, yeniden kullanılabilirliği uygulamaya, daha iyi test edilebilirlik sağlamaya ve "Kötü Tasarım" olarak bilinen kötü tasarım semptomlarını önlemek için tasarımın anlaşılmasını sağlayabilir.

2.2.1.1 Tek Sorumluluk İlkesi (Single-Responsibility Principle)

"Bir sınıfın değişmesi için asla birden fazla neden olmamalıdır." Tek Sorumluluk İlkesi (SRP), değişimin bir nedeni olarak kabul edilir. Bir sınıfı değiştirmek için birden fazla neden varsa, o sınıfın birden fazla sorumluluğa sahip olduğu varsayılır ve bu da yüksek eşleşme ile sonuçlanır. Bu tür bir bağlantı, herhangi bir değişiklik gereksinimi için beklenmedik şekillerde kırılan tasarımlara yol açar (Hassan, 2015).

2.2.1.2 Açık-Kapalı Prensibi (The Open-Close Principle)

"Sınıflar, modüller ve işlevler gibi yazılım varlıkları, genişletmeye açık, ancak değişiklik için kapalı olmalıdır." Bir programda yapılan tek bir değişiklik, bağımlı modüllerde bir dizi değişiklikle sonuçlandığında, o program, "kötü" tasarımıyla ilişkilendirdiğimiz istenmeyen nitelikleri sergiler. Program kırılan, katı, öngörülemez ve tekrar kullanılamaz hale gelir. Açık-kapalı ilke buna çok basit bir şekilde saldırır. Asla değişmeyen modüller tasarlamamanız gerektiğini söyler. Gereksinimler değiştiğinde, halihazırda çalışan eski kodu değiştirerek değil, yeni kod ekleyerek bu tür modüllerin davranışını genişletir (Hassan, 2015).

2.2.1.3 Liskov İkame İlkesi (Liskov Substitution Principle)

"Türetilen tür, temel türlerinin ikamesini tam olarak desteklemelidir." Temel sınıflara işaretçiler veya referanslar kullanan işlevler, türetilmiş nesneleri bilmeden kullanabilmelidir. Bu, ikame özelliği ile ilgilidir. Bu ilkenin önemi, onu ihlal etmenin sonuçlarını düşündüğünüzde ortaya çıkar. LSP'ye uymayan bir fonksiyon varsa, o zaman bir işaretçi veya bir temel sınıfa referans kullanan bu fonksiyon, o temel sınıfın tüm türevlerini bilmelidir (Hassan, 2015).

2.2.1.4 Arayüz Ayırma İlkesi (Interface Segregation Principle)

"İstemciler, kullanmadıkları arayüzlere bağlanmaya zorlanmamalıdır." İlişkisiz bir arayüzdeki bir değişiklik, istemci kodunda yanlışlıkla bir değişikliğe neden olabilir. Bu, tüm istemciler arasında yanlışlıkla bir bağlantıya neden olur. ISP, müşterilerin bunları tek bir sınıf olarak

bilmemeleri gerektiğini önerir. Bunun yerine, istemciler, uyumlu arabirimlere sahip soyut temel sınıfları bilmelidir (Hassan, 2015).

2.2.1.5 Bağımlılık Tersine Çevirme İlkesi(Dependency Inversion Principle)

"Üst düzey modüller alt düzey modüllere bağlı olmamalıdır. Her ikisi de soyutlamalara bağlı olmalıdır. Soyutlamalar ayrıntılara bağlı olmamalıdır. Ancak ayrıntılar soyutlamalara bağlı olmalıdır." Yüksek seviyeli modüller düşük seviyeli modüllerden ayrılmalı, yüksek seviyeli sınıflar ile düşük seviyeli sınıflar arasında bir soyutlama katmanı oluşturulmalıdır. Bağımlılığın tersine çevrilmesi ilkesine uymak için, bu soyutlamayı sorunun ayrıntılarından ayrılması gerekir (Hassan, 2015).

2.2.2 SonarQube

SonarQube (eski adıyla Sonar), 29 programlama dilinde hataları, kod kokularını(code smell) ve güvenlik açıklarını tespit etmek için statik kod analizi ile otomatik incelemeler yapmak üzere kod kalitesinin sürekli denetimi için SonarSource tarafından geliştirilen açık kaynaklı bir platformdur. SonarQube, statik analiz sonuçlarını raporlar ve kodun iyileştirilmesine katkıda bulunur. Temiz ve güvenli kod yazmak için çeşitli araçları mevcuttur (<https://www.sonarqube.org/>).

2.2.2.1 SonarQube Kalite Metrikleri

İşlevsel olmayan gereksinimler (Non functional requirements) bazen kısıtlamalar veya kalite gereksinimleri olarak bilinir. Performans gereksinimleri, sürdürülebilirlik gereksinimleri, güvenlik gereksinimleri, güvenilirlik gereksinimleri, güvenlik gereksinimleri, birlikte çalışabilirlik gereksinimleri veya diğer birçok yazılım gereksinimi türünden biri olup olmadığına göre ayrıca sınıflandırılabilirler (Society, 2014).

Bu bölümde SonarQube aracının değerlendirdiği fonksiyonel olmayan gereksinimler ve derecelendirme faktörleri anlatılacaktır.

2.2.2.2 Güvenilirlik (Reliability)

Güvenilirlik modelleri, yazılım testi sırasında veya hizmette olan yazılımdan toplanan arıza verilerinden oluşturulur ve bu nedenle gelecekteki arızaların olasılığını tahmin etmek ve testin ne zaman durdurulacağına ilişkin kararlara yardımcı olmak için kullanılabilir (Society, 2014).

SonarQube güvenilirlik metriğini aşağıdaki bulgulara göre değerlendirir.

A = 0 Hata (Bug)

B = en az 1 Küçük Hata

C = en az 1 Büyük Hata

D = en az 1 Kritik Hata

E = en az 1 Engelleyici Hata

2.2.2.3 Güvenlik (Security)

Güvenlik için tasarım, bilgilerin ve diğer kaynakların yetkisiz ifşasının, oluşturulmasının, değiştirilmesinin, silinmesinin veya erişimin engellenmesinin nasıl önleneceği ile ilgilidir. Ayrıca, hasarı sınırlayarak, hizmeti sürdürerek, onarım ve kurtarmayı hızlandırarak ve başarısız olma ve güvenli bir şekilde kurtarma yoluyla güvenlikle ilgili saldırılara veya ihlallere nasıl tolerans gösterileceği ile ilgilenir. Erişim kontrolü, temel bir güvenlik kavramıdır ve kriptolojinin doğru kullanımını da sağlamalıdır (Society, 2014).

SonarQube güvenlik metriğini aşağıdaki bulgulara göre değerlendirir.

A = 0 Güvenlik Açıkları

B = en az 1 Küçük Güvenlik Açığı

C = en az 1 Büyük Güvenlik Açığı

D = en az 1 Kritik Güvenlik Açığı

E = en az 1 Engelleyici Güvenlik Açığı

2.2.2.4 Sürdürülebilirlik (Maintainability)

IEEE 14764, sürdürülebilirliği, yazılım ürününün değiştirilebilme yeteneği olarak tanımlar. Değişiklikler, düzeltmeleri, iyileştirmeleri veya yazılımın ortamdaki değişikliklere uyarlanmasını ve ayrıca gereksinimlerdeki ve işlevsel özelliklerdeki değişiklikleri içerebilir. Birincil yazılım kalite özelliği olarak, bakım maliyetlerini azaltmak için yazılım geliştirme

faaliyetleri sırasında sürdürülebilirlik belirtilmeli, gözden geçirilmeli ve kontrol edilmelidir. Başarılı bir şekilde yapıldığında, yazılımın sürdürülebilirliği artacaktır. Alt özellikler genellikle yazılım geliştirme sürecinde önemli bir odak noktası olmadığı için sürdürülebilirliği sağlamak genellikle zordur. Geliştiriciler, tipik olarak, diğer birçok faaliyetle daha fazla meşgul olurlar ve sıklıkla bakımcının gereksinimlerini göz ardı etmeye eğilimlidirler. Bu da program anlama ve müteakip etki analizindeki zorlukların önde gelen nedenlerinden biri olan yazılım dokümantasyonu ve test ortamlarının eksikliğine neden olabilir ve sıklıkla olur. Sistematik ve olgun süreçlerin, tekniklerin ve araçların varlığı, yazılımın sürdürülebilirliğini artırmaya yardımcı olur (Society, 2014).

Varsayılan sürdürülebilirlik derecelendirme tablosu aşağıdaki gibidir.

A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1

Sürdürülebilirlik derecelendirmesi ölçeği, ödenmemiş iyileştirme maliyetinin aşağıdaki gibi olması durumunda dönüşümlü olarak ifade edilebilir:

$\leq 5\%$ Uygulamaya geçen sürenin %5'i, derecelendirme A'dır

6 ila 10 arasında derece B'dir

11 ila 20 arasında derece C'dir

21 ila 50 arasında derece D'dir

50'nin üzerindeki her şey E'dir

2.2.2.5 Döngüsel Karmaşıklık (Cyclomatic Complexity)

Döngüsel Karmaşıklık, yöntemleri değerlendirmek için matematiksel bir model kullanır. Döngüsel Karmaşıklık, kodun ne kadar hata üretebileceği, kodun testi, bakımı ve çıkan hataların çözümü için ne kadar maliyet çıkabileceği hakkında gösterge niteliği taşır. Bunları test etmek için gereken çabanın doğru ölçümlerini üretir, ancak bunları anlamak için gereken maliyetin yanlış ölçümlerini üretir. Yanlış ölçüm üretmesinin nedeni her yöntemin minimum Döngüsel Karmaşıklık puanının bir olmasıdır. Bu, toplam Döngüsel Karmaşıklık puanının herhangi bir sınıftan mı geldiğini yoksa karmaşık bir kontrol akışına sahip bir sınıf mı geldiğini bilmeyi imkansızlaştırır. Sınıf seviyesinin ötesinde, uygulamaların Döngüsel Karmaşıklık puanlarının kod toplamaları satırlarıyla ilişkili olduğu yaygın olarak kabul edilmektedir. Başka bir deyişle, Döngüsel Karmaşıklık, yöntem seviyesinin üzerinde çok az kullanışlıdır. Bu

sorunlara bir çare olarak Bilişsel Karmaşıklık, modern dil yapılarını ele almak, sınıf ve uygulama düzeyinde anlamlı değerler üretmek için formüle edilmiştir (Campbell, 2021).

2.2.2.6 Bilişsel Karmaşıklık (Cognitive Complexity)

Bilişsel Karmaşıklık, yazılımın sürdürülebilirliğini değerlendirmek için matematiksel modeller kullanma uygulamasını kullanmaz. Modern dil yapılarını ele alır, sınıf ve uygulama düzeyinde anlamlı sürdürülebilirlik sonuçları üretir. Döngüsel Karmaşıklık tarafından belirlenen emsallerden başlar, ancak yapıların nasıl sayılması gerektiğini değerlendirmek ve bir bütün olarak modele ne eklenmesi gerektiğine karar vermek için insan yargısını kullanır. Sonuç olarak, programcılara, önceki modellerde mevcut olandan daha adil göreceli bakım değerlendirmeleri olarak çarpan yöntem karmaşıklığı puanları verir (Campbell, 2021).

2.3 YAZILIM TASARIM DESENLERİ

Bu bölümde yazılım tasarım desenlerinden GoF tasarım desenleri incelenecektir.

Nesne yönelimli bir yazılım tasarlamak kolay değildir. Yeniden kullanılabilir ve esnek nesne yönelimli sistem tasarlamak daha zordur. Nesneleri ve bunların birbirleriyle olan ilişkilerini oluştururken ayrıntı düzeyine dikkat edilmesi gerekir. İyi bir tasarım, yalnızca belirli sorunu çözmemeli, aynı zamanda gelecekteki gereksinimleri de karşılayacak şekilde olmalıdır. İyi bir tasarımcının amacı, karşılaşılan problemlerin çözümlerine karşı daha önce denenmiş yöntemlerden yararlanmaktır. Her bir tasarım deseni, nesneye yönelik sistemlerde yeniden kullanılabilir bir tasarımı stratejik olarak adlandırır, tanımlar ve analiz eder (Gahlyan ve Singh, 2018).

Tasarım desenlerini öğrenmek, bunları kodda hemen kullanmak çekici görünmektedir. Ancak bir tasarım deseni seçmeden önce gerçekten dikkatli olunması gerekir. Önce problem anlaşılmalı ve sonra probleme çözüm sağlayabilecek desenler kullanılmalıdır. Bunun yanında desenleri kullanmanın faydasının, sisteme getireceği yükümlülüklerden fazla olduğuna emin olunmalıdır. Standart desenler geliştiricilerin ihtiyaçlarına ve tasarımlarına göre değişiklik gösterebilir, ancak temelde amaç kaliteli uygulamalar geliştirmek ve sorunları iyi bilmek için iyi bilinen çözümler geliştirmektir.

GoF tasarım desenleri üç ana başlık altında incelenmiştir. Yaratımsal tasarım desenleri, mevcut kodun esnekliğini ve yeniden kullanımını artıran çeşitli nesne oluşturma mekanizmaları sağlar. Yapısal tasarım desenleri, bu yapıları esnek ve verimli tutarken nesnelerin ve sınıfların nasıl daha büyük yapılar halinde birleştirileceğini açıklar. Davranışsal tasarım desenleri, algoritmalar ve nesneler arasındaki sorumlulukların atanması ile ilgilidir (Gamma ve diğ., 1994).

Tablo 2.1’de GoF tasarım desenlerinin ayrımı gösterilmiştir.



		Yaratımsal Tasarım Desenleri	Yapısal Tasarım Desenleri	Davranışsal Tasarım Desenleri
Kapsam	Sınıf	Factory Method	Adapter	Interpreter
				Template Method
	Nesne	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Tablo 2.1: GoF Tasarım Desenleri (Gamma ve diğ., 1994)

2.3.1 Yaratımsal Tasarım Desenleri

Yaratımsal tasarım desenleri, örnekleme sürecini soyutlar ve bir sistemin nesnelerini nasıl oluşturulduğundan ve temsil edildiğinden bağımsız hale getirmeye yardımcı olurlar. Bir sınıf yaratım modeli, örneklenen sınıfı değiştirmek için kalıtımı kullanır, oysa bir nesne oluşturma modeli, somutlaştırmayı başka bir nesneye devreder (Gamma ve diğ., 1994). Yaratımsal (Creational) tasarım desenleri, nesne oluşturma sürecinde yardımcı olmaktadır. Nesneyi yaratma, oluşturma ve tanımlama sürecini soyutlamaktadırlar. İki tür yaratımsal desen vardır. Sınıf yaratım desenleri; çalışma zamanında başlatılacak sınıfı seçmek için miras dikkate

alırken, nesne oluşturma desenleri; başka bir nesneye nesne somutlaştırmayı delege eder (Gahlyan ve Singh, 2018).

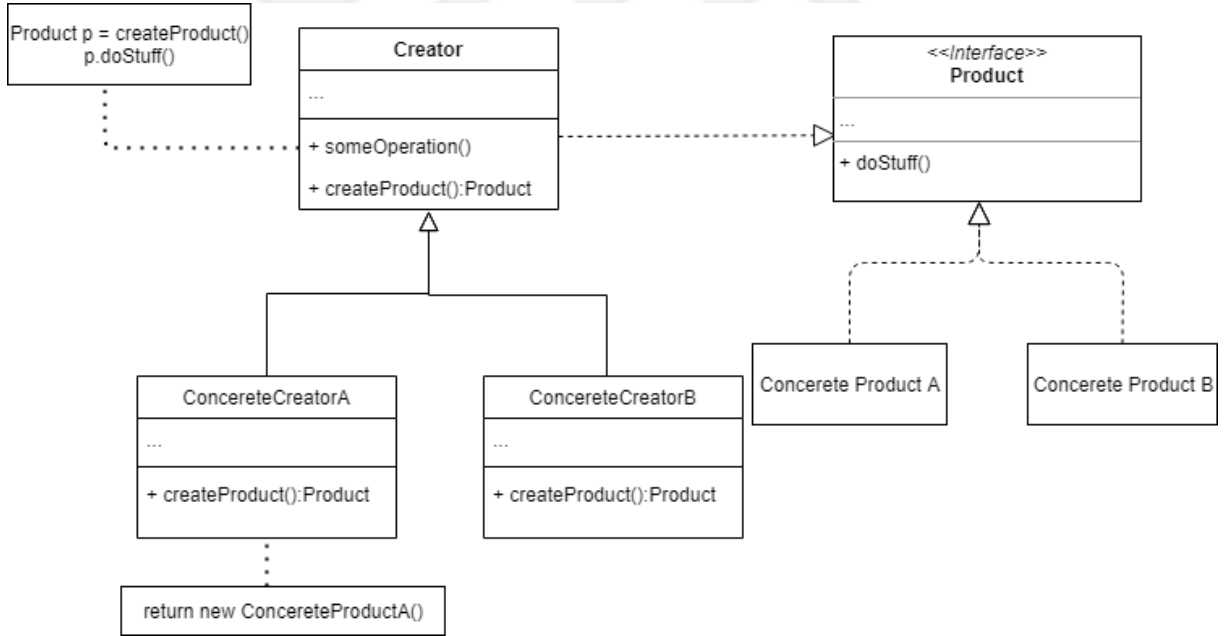
2.3.1.1 Factory Method

Factory Method, bir üst sınıfta nesneler oluşturmak için bir arayüz sağlayan, ancak alt sınıfların oluşturulacak nesnelerin türünü değiştirmesine izin veren yaratımsal bir tasarım desendir.

- **Uygulanabilirlik**

Kodun birlikte çalışması gereken nesnelerin türlerinin ve bağımlılıklarının ne olduğu önceden bilinmediğinde ve mevcut nesneleri her seferinde yeniden oluşturmak yerine yeniden kullanarak sistem kaynaklarından tasarruf etmek istenildiğinde kullanılır.

- **Yapı**



Şekil 2.2: Factory Method UML Diyagramı

1. Product, Creator sınıfı ve alt sınıfları tarafından üretilebilecek tüm nesnelerde ortak olan arayüzü ifade eder.
2. Concrete Products, Product arayüzünün farklı uygulamalarıdır.
3. Creator sınıfı, yeni ürün nesnelerini döndüren factory methodu bildirir. Bu yöntemin dönüş türünün ürün arayüzüne uyması önemlidir. Tüm alt sınıfları yöntemin kendi sürümlerini uygulamaya zorlamak için factory method soyut

olarak bildirilebilir. Alternatif olarak, temel factory method bazı varsayılan Product türlerini döndürebilir.

4. ConcreteCreator, temel fabrika metodunu geçersiz kılar, böylece farklı bir Product türü döndürür.

- **Avantajları**

- Creator ile Concrete Productlar arasında sıkı bağlantı yoktur.
- **Tek Sorumluluk İlkesi:** Ürün oluşturma kodu programda tek bir yere taşınarak kodun desteklenmesi kolaylaştırılabilir.
- **Açık-Kapalı Prensibi:** Mevcut müşteri kodu bozulmadan programa yeni ürün türleri eklenebilir.

- **Dezavantajları**

- Deseni uygulamak için birçok yeni alt sınıfın eklenmesi gerektiğinden kod daha karmaşık hale gelebilir. En iyi durum senaryosu, kalıbı mevcut Creator sınıfları hiyerarşisine sokulmasıdır.

2.3.1.2 Abstract Factory

Abstract Factory, somut sınıflarını belirtmeden ilgili nesnelerin ailelerinin üretilmesini sağlayan yaratımsal tasarım desendir.

- **Uygulanabilirlik**

Kodun çeşitli ilgili ürün aileleri ile çalışması gerektiğinde, ancak kodun bu ürünlerin somut sınıflarına bağlı olmasını istenmediğinde Abstract Factory'nin kullanımı uygun olabilir. Bunlar önceden bilinmeyebilir veya yalnızca gelecekteki genişletilebilirliğe izin vermek istenebilir.

- **Avantajları**

- Bir factoryden temin edilecek ürünlerin birbiriyle uyumlu olduğundan emin olunabilir.
- Ürünler ve kod arasında sıkı bağlantı yoktur.
- **Tek Sorumluluk İlkesi:** Ürün oluşturma kodu tek bir yere çıkarılabilir ve kodun daha kolay desteklenmesi sağlanabilir.

- **Açık-Kapalı Prensibi:** Müşteri kodu bozulmadan yeni ürün çeşitleri tanıtılabilir.
- **Dezavantajları**
 - Kod, olması gerekenden daha karmaşık hale gelebilir, çünkü desen ile birçok yeni arayüz ve sınıf tanıtılmıştır.

2.3.1.3 Builder

Builder, karmaşık nesneleri adım adım oluşturulmasına izin veren yaratımsal bir tasarım desendir. Desen, aynı yapı kodunu kullanarak bir nesnenin farklı türlerini ve temsillerinin oluşturulmasına olanak tanır.

- **Uygulanabilirlik**

Kodda bazı ürünlerin farklı temsillerini oluşturabilmesini istendiğinde Builder desenini kullanılabilir.

Bileşim ağaçları veya diğer karmaşık nesneleri oluşturmak için Builder kullanılabilir.

- **Avantajları**

- Nesneleri adım adım oluşturabilir, yapım adımlarını erteleyebilir veya adımları yinelemeli olarak çalıştırılabilir.
- Ürünlerin çeşitli temsillerini oluştururken aynı yapı kodu yeniden kullanılabilir.
- Tek Sorumluluk İlkesi: Karmaşık yapı kodu ürünün iş mantığından ayrılabilir.

- **Dezavantajları**

- Desen, birden çok yeni sınıf oluşturmayı gerektirir, bu yüzden kodun genel karmaşıklığı artar.

2.3.1.4 Prototype

Prototype, kodu sınıflarına bağımlı hale getirmeden mevcut nesnelerin kopyalanmasına izin veren yaratımsal bir tasarım desendir.

- **Uygulanabilirlik**

Kodun kopyalanması gereken somut nesne sınıflarına bağlı olmaması gerektiğinde Prototype deseni kullanılabilir.

Yalnızca ilgili nesnelerin başlatma şekillerinde farklılık gösteren alt sınıfların sayısı azaltılmak istendiğinde kullanılabilir.

- **Avantajları**

- Nesneler, somut sınıflarına bağlamadan klonlanabilir.
- Önceden oluşturulmuş prototipleri klonlamak için tekrarlanan başlatma kodunu kullanılmayabilir.
- Karmaşık nesneler daha kolay üretilebilir.
- Karmaşık nesneler için konfigürasyon ön ayarları ile uğraşırken kalıtıma bir alternatif elde edilebilir.

- **Dezavantajları**

- Dolaylı referanslara sahip karmaşık nesneleri klonlamak çok zor olabilir.

2.3.1.5 Singleton

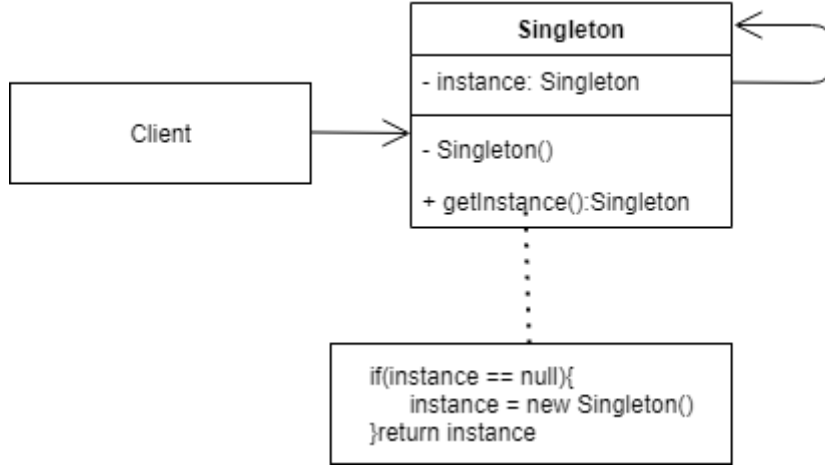
Singleton, bir sınıfın yalnızca bir örneğe sahip olmasını sağlarken bu örneğe genel bir erişim noktası sağlanmasına olanak sağlayan yaratımsal bir tasarım desendir.

- **Uygulanabilirlik**

Programda bir sınıfın tüm istemciler için yalnızca tek bir örneği olması gerektiğinde Singleton deseni kullanılabilir; örneğin, programın farklı bölümleri tarafından paylaşılan tek bir veritabanı nesnesi gibi.

Global değişkenler üzerinde daha sıkı kontrole ihtiyaç duyduğunuzda Singleton deseni kullanılabilir.

- **Yapı**



Şekil 2.3: Singleton UML Diyagramı

Singleton sınıfı, kendi sınıfının aynı örneğini döndüren getInstance statik yöntemini bildirir. Singleton'ın yapıcı fonksiyonu, Client kodundan gizlenmelidir. getInstance yöntemini çağırmak, Singleton nesnesini almanın tek yolu olmalıdır.

- **Avantajları**

- Bir sınıfın yalnızca tek bir örneğe sahip olduğundan emin olunur.
- Bu örneğe, global bir erişim noktası elde edilir.
- Singleton nesnesi yalnızca ilk kez talep edildiğinde atanır.

- **Dezavantajları**

- **Tek Sorumluluk İlkesi:** Tek sorumluluk ilkesini yok sayar. Desen, aynı anda birden fazla sorunu çözer.
- Singleton deseni, örneğin programın bileşenleri birbirleri hakkında çok şey bildiğinde kötü tasarımı maskeleyebilir.
- Desen, çok iş parçacıklı bir ortamda özel işlem gerektirir, böylece birden çok iş parçacığı birkaç kez tek bir nesne oluşturmaz.
- Singleton'ın istemci kodunu birim testi yapmak zor olabilir çünkü birçok test çerçevesi, sahte nesneler üretirken mirasa güvenir.

2.3.2 Yapısal Tasarım Desenleri

Yapısal tasarım desenleri, nesne yapılarının bileşimi ile ilgilidir. Yapısal nesne kalıplarını kullanarak nesneler oluştururken, arayüzleri veya uygulamaları düzenlemek için kalıtım kullanılarak yapısal sınıf desenleri kapsamına giren uygulamalara yeni işlevsellik katılabilir.

Statik sınıf kompozisyonunda olması muhtemel olmayan kompozisyonu dinamik olarak değiştirme esnekliği sağlar (Gamma ve diğ., 1994; Gahlyan ve Singh, 2018).

2.3.2.1 Adapter

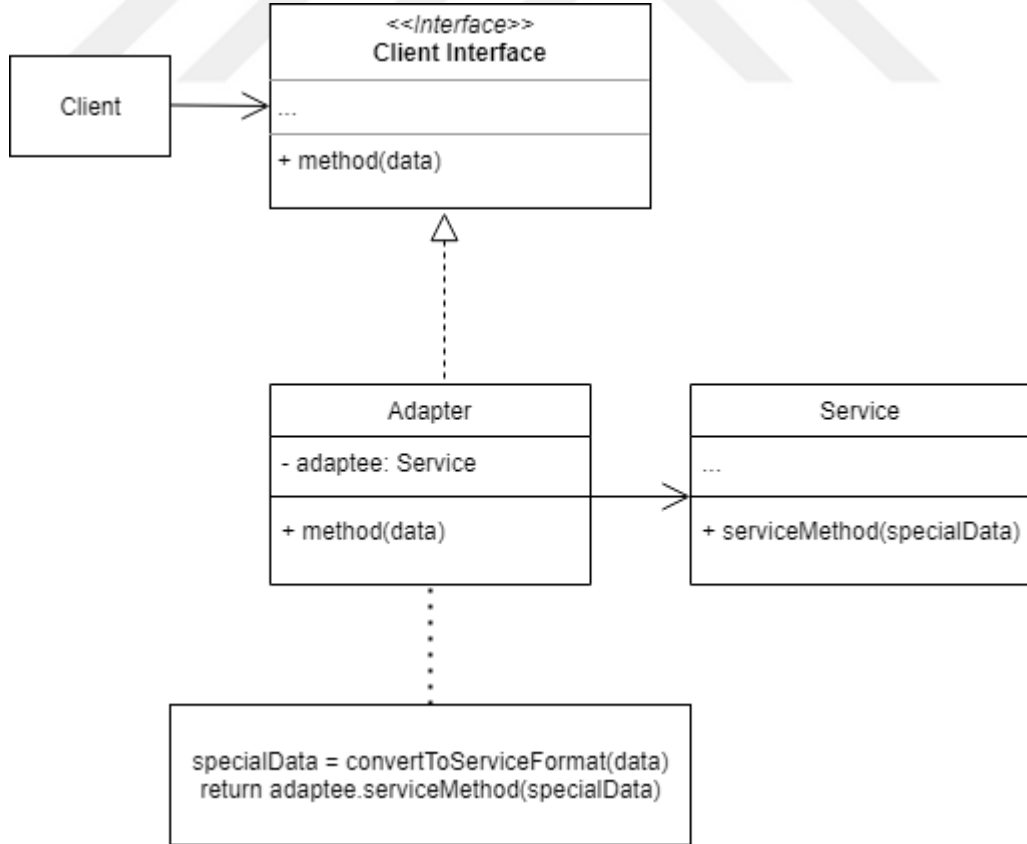
Adapter, uyumsuz arayüzlere sahip nesnelerin birlikte çalışmasına izin veren yapısal bir tasarım desendir.

- **Uygulanabilirlik**

Mevcut sınıflardan bazıları kullanmak istenildiğinde Adapter sınıfı kullanılabilir, ancak arayüz kodun geri kalanıyla uyumlu değildir.

Üst sınıfa eklenemeyen bazı genel işlevlerden yoksun olan birkaç mevcut alt sınıfı yeniden kullanılmak istendiğinde kullanılabilir.

- **Yapı**



Şekil 2.4: Adapter UML Diyagramı

1. Client , programın mevcut iş mantığını içeren bir sınıftır.
2. Client Interface, Client koduyla işbirliği yapabilmek için diğer sınıfların izlemesi gereken bir protokolü açıklar.
3. Service, genellikle üçüncü taraf olan yararlı bir sınıftır. Client, uyumsuz bir arayüze sahip olduğu için bu sınıfı doğrudan kullanamaz.
4. Adapter, hem Client hem de Service sınıfları ile çalışabilen bir sınıftır. Adapter, Client'tan Adapter arabirimi aracılığıyla çağrılar alır ve bunları anlayabileceği bir biçimde Service nesnesi çağrılarına çevirir.
5. Client kodu, istemci arabirimi aracılığıyla bağdaştırıcıyla çalıştığı sürece somut bağdaştırıcı sınıfına bağlanmaz. Bu sayede, mevcut istemci kodunu bozmadan programa yeni tip bağdaştırıcılar ekleyebilirsiniz. Bu, hizmet sınıfının arabirimi değiştirildiğinde yararlı olabilir: istemci kodunu değiştirmeden yalnızca yeni bir bağdaştırıcı sınıfı oluşturabilirsiniz.

- **Avantajları**

- **Tek Sorumluluk İlkesi:** Arayüz veya veri dönüştürme kodu programın birincil iş mantığından ayrılabilir.
- **Açık-Kapalı Prensibi:** Client arabirimi aracılığıyla bağdaştırıcılarla çalıştıkları sürece, mevcut istemci kodunu bozmadan programa yeni bağdaştırıcı türleri eklenebilir.

- **Dezavantajları**

- Bir dizi yeni arabirim ve sınıf oluşturulması gerektiğinden, kodun genel karmaşıklığı artar. Bazen, kodun geri kalanıyla eşleşecek şekilde service sınıfını değiştirmek daha kolaydır.

2.3.2.2 Bridge

Bridge, büyük bir sınıfı veya yakından ilişkili bir sınıflar kümesini birbirinden bağımsız olarak geliştirilebilen iki ayrı hiyerarşiye (soyutlama ve uygulama) ayrılmasına olanak tanıyan yapısal bir tasarım desendir.

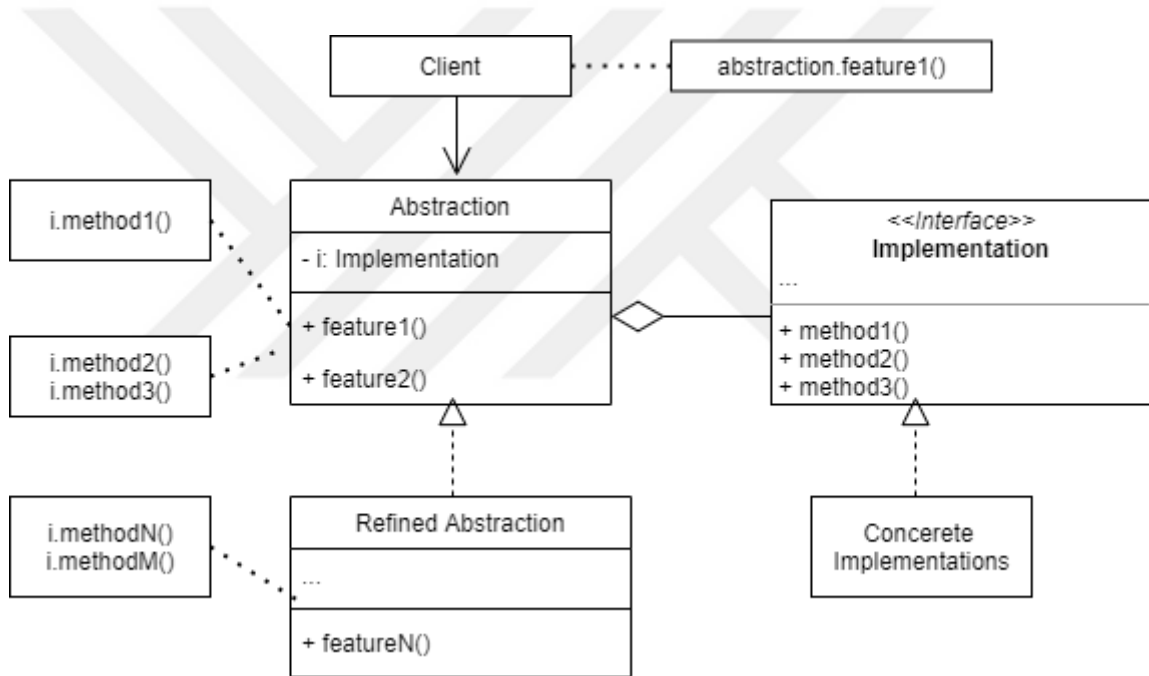
- **Uygulanabilirlik**

Bazı işlevlerin çeşitli varyantlarına sahip yekpare bir sınıf bölünmek ve düzenlenmek istendiğinde (örneğin, sınıf çeşitli veritabanı sunucularıyla çalışabiliyorsa) Bridge deseni kullanılabilir.

Bir sınıfı birkaç ortogonal (bağımsız) boyutta genişletmeniz gerektiğinde Bridge deseni kullanılabilir.

Çalışma zamanında uygulamaların değiştirilmesi gerekiyorsa Bridge deseni kullanılabilir.

- **Yapı**



Şekil 2.5: Bridge UML Diyagramı

1. Abstraction sınıfı, yüksek düzeyde kontrol mantığı sağlar. Gerçek düşük seviyeli çalışmayı yapmak için uygulama nesnesine dayanır.
2. Implementation arayüzü, tüm somut uygulamalar için ortak olan arayüzü bildirir. Bir soyutlama, yalnızca burada açıklanan yöntemler aracılığıyla bir uygulama nesnesiyle iletişim kurabilir.

Abstraction, uygulamayla aynı yöntemleri listeleyebilir, ancak genellikle soyutlama, uygulama tarafından bildirilen çok çeşitli ilkel işlemlere dayanan bazı karmaşık davranışları bildirir.

3. Concrete Implementations, platforma özgü kod içerir.
4. Refined Abstractions, kontrol mantığının varyantlarını sağlar. Ebeveynleri gibi, genel uygulama arayüzü üzerinden farklı uygulamalarla çalışırlar.
5. Genellikle, Client yalnızca soyutlama ile çalışmakla ilgilenir. Ancak, soyutlama nesnesini uygulama nesnelerinden birine bağlamak Client'ın görevidir.

- **Avantajları**

- Platformdan bağımsız sınıflar ve uygulamalar oluşturulabilir.
- Client, yüksek seviyeli soyutlamalarla çalışır. Platform ayrıntılarına açık değildir.
- **Açık-Kapalı Prensibi:** Birbirinden bağımsız olarak yeni soyutlamalar ve uygulamalar tanımlanabilir.
- **Tek Sorumluluk İlkesi:** Soyutlamada üst düzey mantığa ve uygulamada platform ayrıntılarına odaklanılabilir.

- **Dezavantajları**

- Desen kodu daha karmaşık hale getirebilir.

2.3.2.3 Composite

Composite, nesnelerin ağaç yapıları halinde oluşturulmasına ve ardından bu yapılarla tek tek nesnelermiş gibi çalışmaya olanak tanıyan yapısal bir tasarım desendir.

- **Uygulanabilirlik**

Ağaç benzeri bir nesne yapısı uygulanması gerektiğinde Composite desen kullanılabilir.

Client kodun hem basit hem de karmaşık öğeleri, aynı şekilde ele almasını istenildiğinde Composite desen kullanılabilir.

- **Avantajları**

- Karmaşık ağaç yapılarıyla daha rahat çalışılabilir.
- **Açık-Kapalı Prensibi:** Nesne ağacıyla çalışan mevcut kodu bozmadan uygulamaya yeni üye türleri tanımlanabilir.

- **Dezavantajları**

- İşlevselliği çok farklı olan sınıflar için ortak bir arabirim sağlamak zor olabilir. Bazı senaryolarda, bileşen arayüzünü aşırı genelleştirmeniz gerekir, bu da anlaşılmasını zorlaştırır.

2.3.2.4 Decorator

Decorator, nesneleri davranışları içeren özel sarmalayıcı nesnelerin içine yerleştirerek nesnelere yeni davranışlar eklenmesini sağlayan yapısal bir tasarım desendir.

- **Uygulanabilirlik**

Çalışma zamanında, nesnelere fazladan davranışlar atanması gerektiğinde Decorator deseni kullanılabilir.

Kalıtımı kullanarak bir nesnenin davranışını genişletmek kullanışlı veya mümkün olmadığında Decorator deseni kullanılabilir.

- **Avantajları**

- Yeni bir alt sınıf oluşturmadan bir nesnenin davranışı genişletilebilir.
- Çalışma zamanında bir nesneye sorumluluklar eklenip kaldırılabilir.
- Bir nesneyi birden çok decoratora sararak birkaç davranışı birleştirebilirsiniz.
- **Tek Sorumluluk İlkesi:** Muhtemel davranış çeşitlerini uygulayan yekpare bir sınıfı birkaç küçük sınıfa bölünebilir.

- **Dezavantajları**

- Wrapper yığınının belirli bir wrapper çıkarmak zordur.
- Bir decorator davranışı decoratorlar yığındaki sıraya bağlı olmayacak şekilde uygulamak zordur.
- Katmanların ilk yapılandırma kodu oldukça anlaşılmaz görünebilir.

2.3.2.5 Facade

Facade, bir kütüphaneye, çerçeveye veya diğer karmaşık sınıf kümelerine basitleştirilmiş bir arayüz sağlayan yapısal bir tasarım desendir.

- **Uygulanabilirlik**

Karmaşık bir alt sisteme, sınırlı ancak basit bir arayüze ihtiyaç olduğunda Facade deseni kullanılabilir.

Bir alt sistem, katmanlar halinde yapılandırılmak istediğinde Facade deseni kullanılabilir.

- **Avantajları**
 - Kodun karmaşıklığı, alt sistemlere bölerek azaltılabilir.
- **Dezavantajları**
 - Bir facade, bir uygulamanın tüm sınıflarına bağlı bir tanrı nesnesi (god object coupled) haline gelebilir.

2.3.2.6 Flyweight

Flyweight deseni, her bir nesnedeki tüm verileri tutmak yerine birden çok nesne arasında ortak durum parçalarını paylaşarak mevcut RAM miktarına daha fazla nesne sığdırmanıza olanak tanıyan yapısal bir tasarım desendir.

- **Uygulanabilirlik**

Flyweight deseni yalnızca programın mevcut RAM'e zar zor sığan çok sayıda nesneyi desteklemesi gerektiğinde tercih edilmelidir.
- **Avantajları**
 - Programın tonlarca benzer nesneye sahip olduğunu varsayarak RAM'den tasarruf edilebilir.
- **Dezavantajları**
 - Kod çok daha karmaşık hale gelebilir. Yeni ekip üyeleri her zaman bir varlığın durumunun neden bu şekilde ayrıldığını merak edeceklerdir.

2.3.2.7 Proxy

Proxy, başka bir nesne için, bir yedek veya yer tutucu sağlanmasına izin veren yapısal bir tasarım desendir. Proxy, orijinal nesneye erişimi kontrol ederek, istek orijinal nesneye ulaşmadan önce veya sonra bir şeyler gerçekleştirmenize izin verir.

- **Uygulanabilirlik**

Geç başlatma (sanal proxy). Sistem kaynaklarını her zaman açık kalarak boşa harcayan ağır bir service nesnesine sahip olduğunda kullanılır.

Erişim kontrolü (koruma vekili(protection proxy)): Yalnızca belirli istemcilerin service nesnesini kullanabilmesini istenildiğinde kullanılır.

Uzak bir hizmetin (remote proxy) yerel olarak yürütülmesi: Service nesnesinin uzak bir sunucuda tutulması istenildiğinde kullanılır.

Günlüğe kaydetme istekleri (logging proxy): Hizmet nesnesine yönelik isteklerin geçmişini tutmak istenildiğinde kullanılır.

İstek sonuçlarını önbelleğe alma (caching proxy): İstemci isteklerinin sonuçlarının önbelleğe alınması ve bu önbelleğin yaşam döngüsünün yönetilmesi gerektiğinde kullanılır.

Akıllı referans (smart reference): Ağır bir nesneyi kullanan hiç istemci olmadığında onu kaldırabilmeniz gereken zamandır.

- **Avantajları**

- Nesneler, istemcilerin haberi olmadan kontrol edilebilir ve yaşam döngüsü yönetilebilir.
- Proxy, hizmet nesnesi hazır olmasa veya mevcut olmasa bile çalışır.
- **Açık-Kapalı Prensibi:** Hizmeti veya istemcileri değiştirmeden yeni proxy'ler tanıtılabilir.

- **Dezavantajları**

- Çok sayıda yeni sınıf tanıtılması gerektiğinden kod daha karmaşık hale gelebilir.
- Servisten gelen yanıt gecikebilir.

2.3.3 Davranışsal Tasarım Desenleri

Davranışsal tasarım desenleri, algoritmalar ve nesneler arasındaki sorumlulukların atanması ile ilgilidir. Davranış desenleri sadece nesnelerin veya sınıfların kalıplarını değil, aynı zamanda bunlar arasındaki iletişim desenlerini de tanımlar. Bu modeller, çalışma zamanında takip etmesi

zor olan karmaşık kontrol akışını karakterize eder. Odağı kontrol akışından uzaklaştırarak sadece nesnelerin birbirine bağlı olduğu yola konsantre olunmasını sağlarlar (Gamma ve diğ., 1994).

2.3.3.1 Template Method

Template Method, üst sınıftaki bir algoritmanın iskeletini tanımlayan ancak alt sınıfların, yapısını değiştirmeden algoritmanın belirli adımlarını geçersiz kılmasına izin veren davranışsal bir tasarım desendir.

- **Uygulanabilirlik**

İstemcilerin bir algoritmanın yalnızca belirli adımlarını genişletmesine izin vermek, ancak tüm algoritma veya algoritmanın yapısı genişletilmek istenmediğinde Template Method deseni kullanılabilir.

Bazı küçük farklılıklarla neredeyse aynı algoritmaları içeren birkaç sınıf Template Method deseni kullanılabilir. Sonuç olarak, algoritma değiştiğinde tüm sınıfların değiştirilmesi gerekebilir.

- **Avantajları**

- Büyük bir algoritmanın yalnızca belirli bölümlerini geçersiz kılınmasına izin vererek, onları algoritmanın diğer bölümlerinde meydana gelen değişikliklerden daha az etkilenmeleri sağlanabilir.
- Yinelenen kod bir üst sınıfa çekilebilir.

- **Dezavantajları**

- Bazı istemciler, bir algoritmanın sadece gösterilen kısmını görebilirler. Bu durum sınırlandırıcı olabilir.
- **Liskov İkame İlkesi:** Bir alt sınıf aracılığıyla varsayılan bir adım uygulamasını bastırarak, Liskov İkame İlkesi ihlal edilebilir.
- Adımların sürdürülmesi zor olabilir.

2.3.3.2 Chain of Responsibility

Chain of Responsibility, istekleri bir işleyici (chain of handlers) zinciri boyunca iletilmesini sağlayan davranışsal bir tasarım desenidir. Bir istek alındıktan sonra, her işleyici, isteği işlemeye veya zincirdeki bir sonraki işleyiciye iletmeye karar verir.

- **Uygulanabilirlik**

Programın farklı türlerdeki istekleri çeşitli şekillerde işlemesi beklendiğinde, ancak tam istek türleri ve sıraları önceden bilinmediğinde Chain of Responsibility deseni kullanılabilir.

Belirli bir sırayla birkaç işleyiciyi yürütmek gerekli olduğunda desen kullanılabilir.

- **Avantajları**

- Talep işleme sırası kontrol edilebilir.
- Tek Sorumluluk İlkesi: İşlemleri gerçekleştiren sınıflardan işlemleri çağıran sınıflar ayrılabilir.
- Açık-Kapalı Prensibi: Mevcut istemci kodunu bozmadan uygulamaya yeni işleyici sınıfları eklenebilir.

- **Dezavantajları**

- Bazı istekler işlenmeden sonuçlanabilir.

2.3.3.3 Command

Command, bir isteği, istekle ilgili tüm bilgileri içeren bağımsız bir nesneye dönüştüren davranışsal bir tasarım desenidir. Bu dönüşüm, isteklerin bir yöntem argümanı olarak iletilmesine, bir isteğin yürütülmesinin geciktirilmesine veya sıraya koyulmasına ve geri alınamaz işlemlerin desteklenmesine olanak tanır.

- **Uygulanabilirlik**

İşlemlerle nesneler parametreleştirmek istendiğinde Command deseni kullanılabilir.

İşlemlerin sıraya konulması, bunların yürütülmesinin programlanması veya uzaktan yürütülmesi istenildiğinde Command deseni kullanılabilir.

Tersine çevrilebilir işlemler uygulamak istenildiğinde Command deseni kullanılabilir.

- **Avantajları**

- **Tek Sorumluluk İlkesi:** İşlemleri çağıran sınıflar, bu işlemleri gerçekleştiren sınıflardan ayrılabilir.
- **Açık-Kapalı Prensibi:** Mevcut istemci kodu bozulmadan uygulamaya yeni komutlar eklenebilir.
- Geri al / yinele undo/redo uygulanabilir.
- İşlemlerin yürütülmesi ertelenebilir.
- Bir dizi basit komut, karmaşık komutlarla birleştirilebilir.

- **Dezavantajları**

- Gönderenler ve alıcılar arasına yepyeni bir katman geldiği için kod daha karmaşık hale gelebilir.

2.3.3.4 Iterator

Iterator, bir koleksiyonun öğelerinin temelindeki temsilini (liste, yığın, ağaç vb.) açığa çıkarmadan geçiş yapılmasını sağlayan davranışsal bir tasarım desendir.

- **Uygulanabilirlik**

Koleksiyon karmaşık bir veri yapısına sahipse, ancak karmaşıklığın istemcilerden kolaylık veya güvenlik nedenleriyle gizlenmesi istenen durumlarda Iterator deseni kullanılabilir.

Uygulamanızda geçiş kodunun yinelenmesini azaltmak için Iterator deseni kullanılabilir.

Kodun farklı veri yapılarını geçebilmesini istediğinizde veya bu yapıların türleri önceden bilinmediğinde Iterator deseni kullanılabilir.

- **Avantajları**

- **Tek Sorumluluk İlkesi:** Büyük geçiş algoritmaları ayrı sınıflara çıkarılarak istemci kodu ve koleksiyonlar temizlenebilir.
- **Açık-Kapalı Prensibi:** Yeni koleksiyon türleri ve yineleyiciler uygulanabilir ve bunlar hiçbir şeyi bozmadan mevcut koda geçirilebilir.

- Her yineleyici nesnesi kendi yineleme durumunu içerdiğinden, aynı koleksiyon üzerinde paralel olarak yineleme yapılabilir.
- Aynı nedenle, bir yineleme ertelenebilir ve gerektiğinde devam edilebilir.
- **Dezavantajları**
 - Uygulama yalnızca basit koleksiyonlarla çalışıyorsa kalıbı uygulamak fazla olabilir.
 - Bir iterator kullanmak, bazı özel koleksiyonların öğelerinden doğrudan geçmekten daha az verimli olabilir.

2.3.3.5 Mediator

Mediator, nesneler arasındaki karmaşık bağımlılıkların azaltılmasına olanak sağlayan davranışsal bir tasarım desenidir. Desen, nesneler arasındaki doğrudan iletişimi kısıtlar ve onları yalnızca bir aracı nesne aracılığıyla işbirliği yapmaya zorlar.

- **Uygulanabilirlik**

Sınıflar arası bağımlılık fazla olduğunda ve bazı sınıfları değiştirmek zor olduğunda Mediator deseni kullanılabilir.

Bileşenler fazla bağımlı olduğundan, bir bileşeni farklı bir programda yeniden kullanılamadığında Mediator deseni kullanılabilir.

Çeşitli bağlamlarda bazı temel davranışların yeniden kullanılması için çok fazla bileşen alt sınıfı oluşturulması gerektiğinde Mediator deseni kullanılabilir.

- **Avantajları**

- **Tek Sorumluluk İlkesi:** Çeşitli bileşenler arasındaki iletişim tek bir yere çıkarılabilir, anlaşılması ve bakımı kolaylaşabilir.
- **Açık-Kapalı Prensibi:** Gerçek bileşenler değiştirilmek zorunda kalınmadan yeni araçlar tanıtılabilir.
- Bir programın çeşitli bileşenleri arasındaki bağlantı azaltılabilir.
- Tek tek olan bileşenler daha kolay yeniden kullanılabilir.

- **Dezavantajları**

- Zamanla bir Mediator bir Tanrı Nesnesine (God Object) dönüşebilir.

2.3.3.6 Memento

Memento, bir nesnenin önceki durumunu, uygulamasının ayrıntılarını açıklamadan kaydetmenize ve geri yüklemenize olanak tanıyan davranışsal bir tasarım desenidir.

- **Uygulanabilirlik**

Nesnenin önceki bir durumun geri yüklenebilmesi ve nesnenin durumunun anlık görüntülerini oluşturulması istediğinde Memento deseni kullanılabilir.

Nesnenin alanlarına / alıcılarına / ayarlayıcılarına(fields/getters/setters) doğrudan erişim, kapsüllemeyi (encapsulation) ihlal ettiğinde Memento deseni kullanılabilir.

- **Avantajları**

- Nesnenin kapsüllenmesini(encapsulation) ihlal edilmeden nesnenin durumu anlık olarak alınabilir.
- Caretaker sınıfının, oluşturanın durumunun Originator'un durum geçmişini korumasına izin vererek Originator'un kodu basitleştirilebilir.

- **Dezavantajları**

- İstemciler çok sık memento oluşturuyorsa, uygulama çok fazla RAM tüketebilir.
- Caretakers, eskimiş logları (mementoları) yok edebilmek için kaynak sahibinin yaşam döngüsünü izlemelidir.
- PHP, Python ve JavaScript gibi çoğu dinamik programlama dili, memento içindeki duruma dokunulmadan kalacağını garanti edemez.

2.3.3.7 Observer

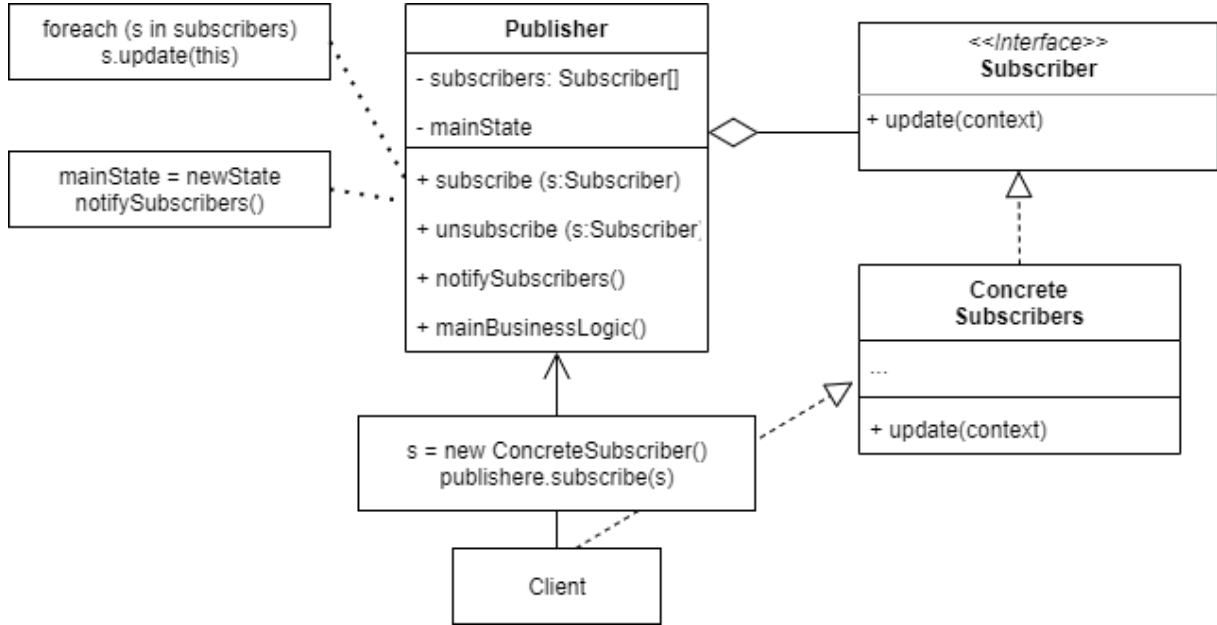
Observer, gözlemledikleri nesnede meydana gelen herhangi bir olay hakkında birden çok nesneyi bilgilendirmek için bir abonelik mekanizmasının tanımlanmasına izin veren davranışsal bir tasarım desenidir.

- **Uygulanabilirlik**

Bir nesnenin durumundaki değişiklikler diğer nesnelerin değiştirilmesini gerektirebilirse ve gerçek nesne kümesi önceden bilinmediğinde veya dinamik olarak değiştiğinde Observer deseni kullanılabilir.

Uygulamadaki bazı nesnelerin diğerlerini gözlemlemesi gerektiğinde, yalnızca sınırlı bir süre için veya belirli durumlarda Observer deseni kullanılabilir.

- **Yapı**



Şekil 2.6: Observer UML Diyagramı

1. Publisher, diğer nesnelere ilgi duyan olayları yayımlar. Bu olaylar, Publisher yayıncı durumunu değiştirdiğinde veya bazı davranışları yürüttüğünde meydana gelir. Yayıncılar(Publishers), yeni abonelerin katılmasına ve mevcut abonelerin listeden ayrılmasına olanak tanıyan bir abonelik altyapısı içerir.
2. Yeni bir olay meydana geldiğinde, yayıncı(Publisher) abonelik listesinin üzerinden geçer ve her abone(Subscriber) nesnesinde abone arayüzünde bildirilen bildirim yöntemini çağırır.
3. Subscriber interface Abone arayüzü, bildirim arayüzünü bildirir. Çoğu durumda, tek bir güncelleme yönteminden oluşur. Yöntem, Publisher yayıncının güncellemeyle birlikte bazı etkinlik ayrıntılarını iletmesine izin veren birkaç parametreye sahip olabilir.
4. Concrete Subscribers, Publisher tarafından verilen bildirimlere yanıt olarak bazı eylemler gerçekleştirir. publisher concrete classes ile eşlenmemesi için bu sınıfların tümü aynı arayüzü uygulamalıdır.

5. Genellikle Subscriber'ın güncellemeyi doğru bir şekilde işlemek için bazı bağlamsal bilgilere ihtiyacı vardır. Bu nedenle, yayıncılar genellikle bazı bağlam verilerini bildirim yönteminin argümanları olarak iletirler. Publisher, abonenin gerekli verileri doğrudan almasına izin vererek kendisini bir argüman olarak iletebilir.
6. Client, Publisher ve Subscriber nesnelerini ayrı ayrı oluşturur ve ardından aboneleri yayıncı güncellemeleri için kaydeder.

- **Avantajları**

- **Açık-Kapalı Prensibi:** Yayıncının kodunu değiştirmek zorunda kalmadan yeni abone sınıfları tanımlanabilir. Bir yayıncı arayüzü varsa bunun tersi de geçerlidir.
- Çalışma zamanında nesneler arasında ilişkiler kurulabilir.

- **Dezavantajları**

- Aboneler rastgele sırayla bilgilendirilir.

2.3.3.8 State

State, bir nesnenin iç durumu değiştiğinde davranışını değiştirmesine izin veren davranışsal bir tasarım desenidir. Nesne, sınıfını değiştirmiş gibi görünür.

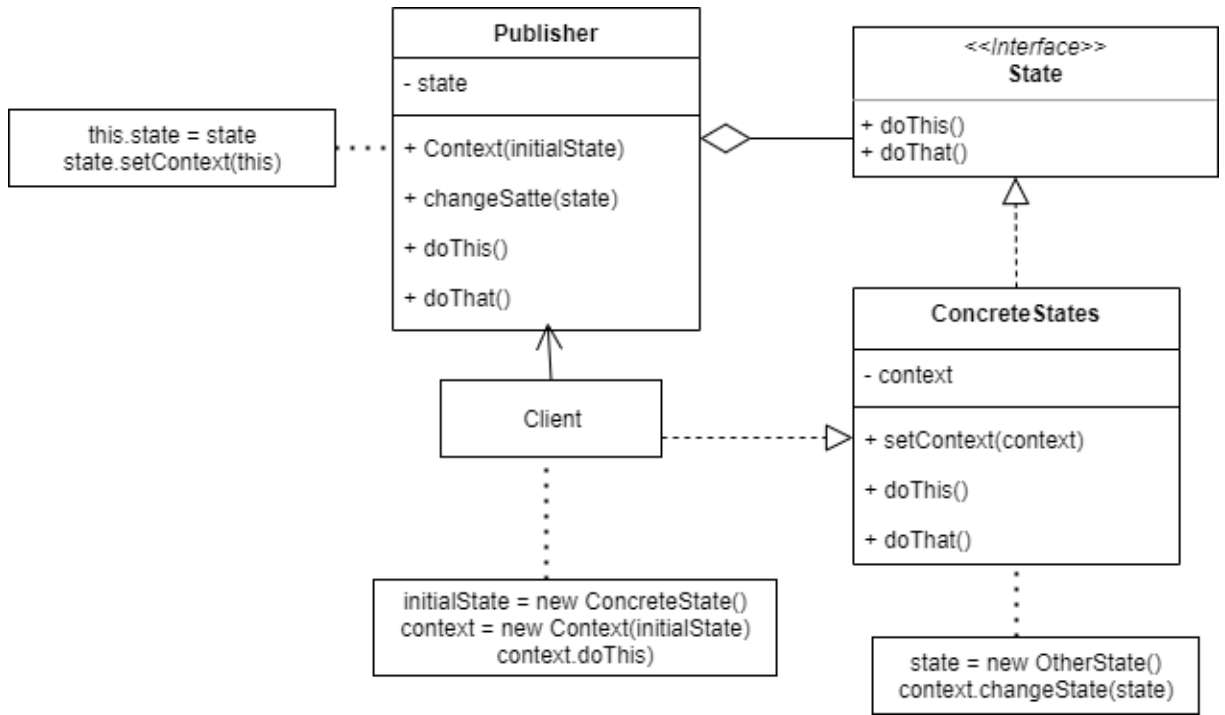
- **Uygulanabilirlik**

Mevcut durumuna bağlı olarak farklı davranan bir nesneye sahip olduğunda, durumların sayısı çok fazla olduğunda ve duruma özgü kod sık sık değiştiğinde State deseni kullanılabilir.

Sınıfın, sınıfın alanlarının mevcut değerlerine göre nasıl davrandığını değiştiren büyük koşullarla karmaşıklaşmış bir sınıf olduğunda State deseni kullanılabilir.

Koşul tabanlı bir durum makinesinin benzer durumları ve geçişleri arasında çok sayıda yinelenen kod olduğunda State deseni kullanılabilir.

- **Yapı**



Şekil 2.7: State UML Diyagramı

1. Context, somut durum nesnelerinden birine bir referansı saklar ve tüm duruma özgü çalışmalara delege eder. Context, State arabirimi aracılığıyla state nesnesiyle iletişim kurar. Context, kendisine yeni bir state nesnesi geçirmek için bir ayarlayıcıyı açığa çıkarır.
 2. State arabirimi, duruma özgü yöntemleri bildirir. Bu yöntemler tüm somut durumlar için mantıklı olmalıdır.
 3. ConcreteStates, duruma özgü yöntemler için kendi uygulamalarını sağlar. Benzer kodun tekrarlanmasını önlemek için, bazı ortak davranışları kapsayan ara soyut sınıflar kullanılabilir.
- State sınıfının nesneleri, bağlam nesnesine bir geri referans depolayabilir. Bu referans aracılığıyla durum, bağlam nesnesinden gerekli bilgileri alabilir ve durum geçişlerini başlatabilir.
4. Hem Context hem de ConcreteStates, bağlamanın bir sonraki durumunu belirleyebilir ve bağlama bağlı durum nesnesini değiştirerek gerçek durum geçişini gerçekleştirebilir.

- **Avantajları**

- **Tek Sorumluluk İlkesi:** Belirli durumlarla ilgili kod ayrı sınıflar halinde düzenlenebilir.
- **Açık-Kapalı Prensibi:** Mevcut durum sınıflarını veya bağlamı değiştirmeden yeni durumları tanımlanabilir.
- Durum makinesi koşulları ortadan kaldırılarak kod basitleştirilebilir.

- **Dezavantajları**

- Bir durum makinesinin yalnızca birkaç durumu varsa veya nadiren değişirse, kalıbı uygulamak aşırı olabilir.

2.3.3.9 Strategy

Strategy, bir algoritma ailesi tanımlanmasına, her birinin ayrı bir sınıfa koyulmasına ve nesnelerini birbiriyle değiştirilebilir hale getirilmesine olanak tanıyan davranışsal bir tasarım desenidir.

- **Uygulanabilirlik**

Bir nesne içinde bir algoritmanın farklı varyantları kullanılmak istediğinde ve çalışma süresi sırasında bir algoritmadan diğerine geçiş yapabilmek istendiğinde Strategy deseni kullanılabilir.

Strategy, yalnızca bazı davranışları gerçekleştirme biçimleri bakımından farklılık gösteren çok sayıda benzer sınıf olduğunda kullanılabilir.

Bir sınıfın iş mantığını, bu mantık bağlamında o kadar önemli olmayabilecek algoritmaların uygulama ayrıntılarından ayırmak için Strategy deseni kullanılabilir.

Sınıfın, aynı algoritmanın farklı türevleri arasında geçiş yapan çok büyük bir koşullu operatöre sahip olduğunda için Strategy deseni kullanılabilir.

- **Avantajları**

- Çalışma zamanında bir nesnenin içinde kullanılan algoritmalar takas edilebilir.
- Bir algoritmanın uygulama ayrıntıları, onu kullanan koddan ayrılabilir.
- Kalıtım, kompozisyon ile değiştirilebilir.

- **Açık-Kapalı Prensibi:** Bağlamı değiştirmek zorunda kalmadan yeni stratejiler sunulabilir.
- **Dezavantajları**
 - Yalnızca birkaç algoritma varsa ve bunlar nadiren değişiyorsa, programı desenle birlikte gelen yeni sınıflar ve arayüzlerle kod aşırı karmaşık hale gelebilir.
 - Uygun stratejinin seçilmesi için stratejiler arasındaki farkların farkında olunmalıdır.

2.3.3.10 Visitor

Visitor, algoritmaları üzerinde çalıştıkları nesnelerden ayrılmasına izin veren davranışsal bir tasarım desendir.

- **Uygulanabilirlik**

Karmaşık bir nesne yapısının tüm öğeleri (örneğin, bir nesne ağacı) üzerinde bir işlem gerçekleştirilmesi gerektiğinde Visitor deseni kullanılabilir.

Bir davranış, yalnızca bir sınıf hiyerarşisinin bazı sınıflarında anlamlıyken diğerlerinde anlamsız olduğunda Visitor deseni kullanılabilir.

- **Avantajları**

- **Açık-Kapalı Prensibi:** Sınıfları değiştirmeden farklı sınıflardaki nesnelerle çalışabilen yeni bir davranış sunulabilir.
- **Tek Sorumluluk İlkesi:** Aynı davranışın birden çok sürümü aynı sınıfa taşınabilir.
- Bir visitor nesnesi, çeşitli nesnelerle çalışırken bazı yararlı bilgiler toplayabilir. Bu, bir nesne ağacı gibi bazı karmaşık nesne yapılarını geçmek ve ziyaretçiyi bu yapının her bir nesnesine uygulamak istediğinizde kullanışlı olabilir.

- **Dezavantajları**

- Öğe hiyerarşisine her sınıf eklendiğinde veya buradan çıkarıldığında tüm ziyaretçileri güncellenmesi gerekir.
- Visitor arabirimleri, birlikte çalışmaları gereken öğelerin özel alanlarına ve yöntemlerine gerekli erişemeyebilir.

2.3.4 Desenler Arası İlişkiler

GoF yazılım tasarım desenleri ihtiyaçlar doğrultusunda birbirlerinden türetilmiştir. Ayrıca desenler birbirleri ile birlikte kullanılabilir. Bu bölümde desenlerin aralarındaki ilişkilere değinilecektir.

- Abstract Factory sınıfları genellikle bir dizi Factory Method desenini temel alır, ancak bu sınıflar üzerindeki yöntemleri oluşturmak için Prototype da kullanılabilir.
- Abstract Factory, Bridge ile birlikte kullanılabilir. Bu eşleştirme, Bridge tarafından tanımlanan bazı soyutlamalar yalnızca belirli uygulamalarla çalışabildiğinde yararlıdır. Bu durumda, Abstract Factory bu ilişkileri kapsayabilir ve karmaşıklığı istemci kodundan gizleyebilir.
- Abstract Factory, Builder ve Prototype desenleri, Singleton olarak uygulanabilir.
- Abstract Factory, yalnızca alt sistem nesnelerinin istemci kodundan oluşturulma şekli gizlemek istenildiğinde Facade'e bir alternatif olarak hizmet edebilir.
- Adapter, mevcut bir nesnenin arayüzünü değiştirirken, Decorator bir nesneyi arayüzünü değiştirmeden geliştirir. Decorator, Adapter kullandığınızda mümkün olmayan yinelemeli kompozisyonu destekler.
- Adapter, wrapped -paketteki nesneye farklı bir arabirim, Proxy nesneye aynı arabirimi sağlar Decorator ise nesneye gelişmiş bir arabirim sağlar.
- Prototype, Memento'nun daha basit bir alternatifi olabilir. Geçmişten saklamak istenilen nesne oldukça basitse ve dış kaynaklara bağlantıları yoksa veya bağlantıların yeniden kurulması kolaysa kullanılabilir.
- Bir Facade sınıfı, çoğu durumda tek bir Facade nesnesi yeterli olduğundan, genellikle bir Singleton'a dönüştürülebilir.
- Bir şekilde nesnelerin tüm paylaşılan durumlarını yalnızca bir Flyweight nesnesine indirgenebildiyse, Flyweight Singleton'a benzeyecektir. Ancak bu desenler arasında iki temel fark vardır: Yalnızca bir tane Singleton örneği olmalıdır, oysa bir Flyweight sınıfı farklı iç durumlara sahip birden çok örneğe sahip olabilir. Singleton nesnesi değiştirilebilir. Flyweight nesneler değişmezdir.
- Çoğu tasarım, Factory Method desenini kullanarak başlar ve Soyut Fabrika, Prototip veya Builder'a doğru gelişir.

- Bridge genellikle bir uygulamanın parçalarını birbirinden bağımsız olarak geliştirilmesine izin verecek şekilde önceden tasarlanır. Adapter ise uyumsuz sınıfların birlikte çalışmasını sağlamak için genellikle mevcut bir uygulamayla kullanılır.
- Bridge, State, Strategy (ve bir dereceye kadar Adapter) çok benzer yapılara sahiptir. Aslında, tüm bu desenler, çalışmayı diğer nesnelere devreden kompozisyona dayanmaktadır. Ancak hepsi farklı sorunları çözer.
- Builder sınıfları genellikle Factory Method desenini temel alır.
- Builder, Bridge ile birleştirilebilir; Director sınıfı soyutlamanın rolünü oynar, farklı kurucular ise uygulama görevi görür.
- Builder, karmaşık bileşim ağaçlar oluştururken kullanılabilir, inşa adımları yinelemeli olarak çalışacak şekilde programlanabilir.
- Builder, karmaşık nesneleri adım adım oluşturmaya odaklanır. Abstract Factory, ilgili nesnelerin ailelerini oluşturmada uzmanlaşmıştır. Abstract Factory ürünü hemen iade ederken, Builder ürünü almadan önce bazı ek yapım adımlarının çalıştırılmasına izin verir.
- Chain of Responsibility genellikle Composite ile birlikte kullanılır. Bu durumda, bir yaprak bileşen bir istek aldığı anda, bunu tüm ana bileşenlerin zincirinden nesne ağacının köküne kadar geçirebilir.
- Chain of Responsibility ve Decorator çok benzer sınıf yapılarına sahiptir. Her iki desen de yürütmeyi bir dizi nesneden geçirmek için yinelemeli kompozisyona dayanır. Bununla birlikte, birkaç önemli fark vardır. CoR işleyicileri, birbirinden bağımsız olarak keyfi işlemleri yürütebilir. Ayrıca herhangi bir noktada isteği iletmeyi bırakabilirler. Öte yandan, çeşitli Decorator nesnenin davranışını temel arayüzle tutarlı tutarken genişletebilir. Ayrıca, decorator'lerin talebin akışını kesmesine izin verilmez.
- Chain of Responsibility, Command, Mediator ve Observer, istekleri gönderenleri ve alıcıları birbirine bağlamanın çeşitli yollarını ele alır: Chain of Responsibility, bir talebi biri onu ele alana kadar, potansiyel alıcılardan oluşan dinamik bir zincir boyunca sırayla iletir. Command, gönderenler ve alıcılar arasında tek yönlü bağlantılar kurar. Mediator, gönderenler ve alıcılar arasındaki doğrudan bağlantıları ortadan kaldırarak, onları bir arabulucu nesnesi aracılığıyla dolaylı olarak iletişim kurmaya zorlar. Observer ise alıcıların istekleri dinamik olarak kabul etmesine ve abonelikten çıkmasına izin verir.

- Chain of Responsibility'deki işleyiciler(handler), Command olarak da uygulanabilir. Bu durumda, bir istekle temsil edilen aynı bağlam nesnesi üzerinde birçok farklı işlem yürütülebilir. Ancak, isteğin kendisinin bir Command nesnesi olduğu başka bir yaklaşım daha vardır. Bu durumda, aynı işlem bir zincire bağlı bir dizi farklı bağlamda yürütülebilir.
- Command ve Strategy benzer görünebilir, çünkü bir nesneyi bir eylemle parametreleştirmek için her ikisi de kullanılabilir. Ancak amaçları çok farklıdır. Herhangi bir işlemi bir nesneye dönüştürmek için Command kullanılabilir. İşlemin parametreleri o nesnenin alanları haline gelir. Dönüşüm, işlemin yürütülmesini ertelenmesine, sıraya koyulmasına, komutların geçmişinin kaydedilmesine vb. olanak tanır. Öte yandan, Strategy genellikle aynı şeyi yapmanın farklı yollarını açıklar ve bu algoritmaların tek bir bağlam sınıfında değiştirilmesine izin verir.
- Composite ağacın üzerinde bir işlem yürütmek için Visitor deseni kullanılabilir.
- Composite ve Decorator benzer yapı diyagramlarına sahiptir, çünkü her ikisi de açık uçlu sayıda nesneyi organize etmek için özyinelemeli kompozisyona güvenir. Composite ağaçtaki belirli bir nesnenin davranışını genişletmek için Decorator kullanılabilir.
- Composite ve Decoratordan yoğun şekilde yararlanan tasarımlar, genellikle Prototype kullanımından yararlanabilir. Deseni uygulamak, karmaşık yapıları sıfırdan yeniden oluşturmak yerine klonlamanıza olanak tanır.
- Decorator ve Proxy benzer yapılara sahiptir, ancak amaçları çok farklıdır. Her iki desen de, bir nesnenin işin bir kısmını diğerine devretmesinin beklendiği kompozisyon ilkesi üzerine inşa edilmiştir. Aradaki fark, bir Proxy'nin genellikle hizmet nesnesinin yaşam döngüsünü kendi başına yönetmesidir, oysa Decorator sınıflarının bileşimi her zaman client tarafından kontrol edilir.
- Durumunun geçmişte saklamak istenilen nesne oldukça basit ve dış kaynaklara bağlantıları yoksa veya bağlantıların yeniden kurulması kolay ise Prototype, Memento'nun daha basit bir alternatifi olabilir.
- Facade ve Mediator'ın benzer işleri vardır: Birbirine sıkı sıkıya bağlı birçok sınıf arasında işbirliğini düzenlemeye çalışırlar. Facade, bir nesne alt sistemi için basitleştirilmiş bir arayüz tanımlar, ancak herhangi bir yeni işlevsellik sunmaz. Alt sistemin kendisi Facadeden habersizdir. Alt sistem içindeki nesneler doğrudan iletişim

kurabilir. Mediator, sistemin bileşenleri arasındaki iletişimi merkezileştirir. Bileşenler yalnızca aracı nesnesini bilir ve doğrudan iletişim kurmaz.

- Facade, hem karmaşık bir varlığı tamponlaması hem de kendi başına başlatması açısından Proxy'ye benzer. Facade'den farklı olarak, Proxy, hizmet nesnesiyle aynı arabirime sahiptir ve bu da onları birbirinin yerine kullanılabilir kılar.
- Facade, mevcut nesneler için yeni bir arayüz tanımlarken, Adapter mevcut arayüzü kullanılabilir hale getirmeye çalışır. Adapter genellikle tek bir nesneyi sararken, Facade tüm bir nesneler alt sistemi ile çalışır.
- Factory Method, Template Method'un bir uzmanlığıdır. Aynı zamanda, bir Factory Method, büyük bir Template Method'da bir adım olarak hizmet edebilir.
- Flyweight, çok sayıda küçük nesnenin nasıl yapılacağını gösterirken, Facade, tüm bir alt sistemi temsil eden tek bir nesnenin nasıl yapılacağını gösterir.
- Iteratorlar, Composite ağaçlarda gezinmek için kullanılabilir.
- Karmaşık bir veri yapısında gezinmek ve her biri farklı sınıflara sahip olsalar bile öğeleri üzerinde bazı işlemler gerçekleştirmek için Iterator ile birlikte Visitor kullanılabilir.
- Koleksiyon alt sınıflarının, koleksiyonlarla uyumlu farklı yineleyici(iterator) türleri döndürmesine izin vermek için Iterator ile birlikte Factory Method kullanılabilir.
- Mediator ve Observer arasındaki fark genellikle anlaşılmazdır. Çoğu durumda, bu desenlerden herhangi biri uygulanabilir; ancak bazen her ikisini de aynı anda uygulanabilir. Mediator'un birincil amacı, bir dizi sistem bileşeni arasındaki karşılıklı bağımlılıkları ortadan kaldırmaktır. Bunun yerine, bu bileşenler tek bir aracı nesneye bağımlı hale gelir. Observer'ın amacı, bazı nesnelerin diğerlerinin astları olarak hareket ettiği nesneler arasında dinamik tek yönlü bağlantılar kurmaktır.
- Mevcut yineleme durumunu yakalamak ve gerekirse geri almak için Iterator ile birlikte Memento kullanılabilir.
- Prototype sınıfları genellikle Factory Method desenini temel alır.
- Prototype, Command'ların kopyalarının geçmişe kaydedilmesi gerektiğinde yardımcı olabilir.
- Prototype, mirasa dayalı değildir, bu nedenle dezavantajları yoktur. Öte yandan, Prototype, klonlanan nesnenin karmaşık bir şekilde başlatılmasını gerektirir. Factory Method, mirasa dayanır ancak bir başlatma adımı gerektirmez.

- RAM'den tasarruf etmek için Composite ağacın paylaşılan yaprak düğümleri Flyweight olarak uygulanabilir.
- State, Strategy'nin bir uzantısı olarak düşünülebilir. Her iki desen de kompozisyona dayalıdır, bazı işleri yardımcı nesnelere devrederek bağlamın davranışını değiştirirler. Strategy, bu nesneleri tamamen bağımsız ve birbirlerinden habersiz hale getirir. Bununla birlikte, Strategy somut durumlar arasındaki bağımlılıkları sınırlamaz ve bunların, istediği zaman bağlamın durumunu değiştirmesine izin verir.
- Template Method, kalıtıma dayanır: bu parçaları alt sınıflarda genişleterek bir algoritmanın parçalarını değiştirilmesine izin verir. Strategy, kompozisyona dayanır: Nesneye, o davranışa karşılık gelen farklı stratejiler sunarak davranışın bazı kısımları değiştirilebilir. Template Method sınıf düzeyinde çalışır, bu nedenle statiktir. Strategy ise nesne düzeyinde çalışır ve çalışma zamanında davranışların değiştirilmesine izin verir.
- Composite ağacın üzerinde bir işlem yürütmek için Visitor deseni kullanılabilir.
- Visitor, Command deseninin güçlü bir versiyonu olarak ele alınabilir. Visitor sınıfının nesneleri, farklı sınıflardan çeşitli nesneler üzerinde işlemler yürütebilir.

2.4 YAZILIM TASARIM DESENLERİ TESPİT ARAÇLARI

Tasarım örüntülerini algılama alanı hem akademi hem de endüstriden araştırmacıları cezbetmiştir. Tasarım desenleri, geliştirme ekibi tarafından alınan en eski tasarım kararlarını yansıtır. Tasarım desenleri, yazılım belgelerini iyileştirir, geliştirme sürecini hızlandırır, yazılım mimarîlerinin büyük ölçekli yeniden kullanılmasını sağlar, uzman bilgilerini yakalar, tasarım değişimlerini yakalar ve sistemleri yeniden yapılandırmaya yardımcı olur. Nesne yönelimli kaynak kodundan tasarım deseni örneklerini tespit etmek için son yirmi yılda birçok yaklaşım kullanılmıştır. Tasarım örüntüsü algılama yaklaşımlarının temel amacı, tasarım örüntülerinin örneklerini doğru bir şekilde çıkarmaktır. Bununla birlikte, tespit yaklaşımları girdileri, çıkarım metodolojisi, vaka çalışmaları, geri kazanılan modeller, sistem gösterimi, doğruluk ve doğrulama yöntemlerinde farklılık gösterir.

Farklı teknikler ve araçlar kullanarak kaynak koddan tasarım kalıpları ve varyantlarının tespiti, farklı uygulamaların karmaşık mimarîsini anlamak için anahtardır (Waheed ve diğ., 2016). Tasarım desenlerinin kaynak koddan tespit edilmesi ve kurtarılması Gamma ve diğ. (1994) ilk bölümde bahsedilen kitabının yayınlanmasıyla başlamıştır. Doğru ve akıllıca seçilip

uygulandığında tasarım desenlerinin birçok avantajı vardır (Zhang, 2011). Tasarım desenlerinin kaynak koddan tespit edilmesi bu modern çağda farklı teknikler ve araçların yardımıyla çok zor bir iş değildir ancak bir programcının ihtiyacına göre çeşitli varyasyonları kullanması desenlerin tespitini zorlaştırmaktadır. Şimdiye kadar standart tasarım desenlerinin çeşitleri hakkında kesin bir bilgi yoktur, çünkü tüm çeşitleri tespit etmek için herhangi bir kriter belirlenmemiştir. Farklı programlama dili yapıları ve geliştiricinin mantığı ve teknikleri nedeniyle varyasyonlar olabilir. Eski uygulamaların kaynak kodundan tasarım desenlerinin varyantlarının tespiti, tüm varyant tanımlarının uygun kataloğunun olmaması ve desen kurtarma araçlarının verimsizliği nedeniyle zor olmaya devam edecektir. Varyantlardaki varyasyonlar, standart desenler düşünüldüğünde evrim niteliğinde olabilir. Desenlerin varyantlarını tespit etmek için birden fazla araç ve teknik vardır, ancak varyantları tespit etmek için kullanılan teknikler, araçlar, yeni ve bilinmeyen varyantlara ilişkin bilgilere sahip değildir. Bu nedenle, çok fazla iş yaptıktan ve modern teknikleri kullandıktan sonra, tespitte doğruluk büyük bir zorluktur çünkü doğru ve esnek bir çözüm bulmaya yardımcı olabilecek tutarlı ve güvenilir değişken tanımları bulunmamaktadır. Özellik tabanı tespiti, desen tespitine yönelik en güçlü yaklaşımlardan biridir. Bu yaklaşım, desen tanımını tekrar eden özelliklere ayırır ve bu özellikler, bir desen oluşturan yapısal, ilişkisel ve davranışsal parçalar olabilir (Rasool ve Mader, 2011).

Tasarım deseni algılama yaklaşımları, desen tanıma için benzer ana adımları gerçekleştirir. Bu adımlar, kaynak koddan bilgi çıkarma, desen tespiti ve sonuç gösterimi ile ilgilidir. Bu çalışma yalnızca GoF tasarım desenlerine odaklanmaktadır. Elbette bu, GoF desenlerinin diğer kalıp türlerinden (örneğin, mimarî desenler ve deyim kalıpları) daha iyi olduğu anlamına gelmez.

Program kodunda, tasarım desenlerinin tespit edilmesi bir tersine mühendislik faaliyetidir. Yöntem hem yazılım sisteminin kalitesini tahmin etmekte hem de bakım ve geliştirme süreçlerinde kullanılabilir (Antoniol ve diğ., 1998). Tasarım ve performansını daha iyi anlamak için programı ve program kodunda değişiklikler yapmak için kaynak kodda tasarım desenlerini algılamak oldukça önemlidir (Gueheneuc ve diğ., 2009). Program kodunda tasarım desenlerini tanımlamak için birden fazla yaklaşım vardır. Herkese uyan tek bir yaklaşım maalesef yoktur, her yaklaşımın kendi avantajları ve dezavantajları vardır.

Literatürde tasarım desenlerini farklı yöntemlerle tespit eden birçok araç bulunmaktadır. Bu bölümde bu çalışma kapsamında kullanılan, DPD (Design Pattern Detection) ve PINOT

(Pattern INference recOverY Tool) anlatılacaktır. Son olarak literatürde var olan diğer araçlara değinilecektir.

2.4.1 DPD (Design Pattern Detection)

Tsantalis ve diğ. (2006) tarafından önerilen araç, örüntü oluşumlarını benzerlik puanlama algoritmasına (Similarity Scoring Algorithm - SSA) dayalı olarak tanımlar, örüntüler, orijinal olarak tanımlandıkları standart formların varyasyonları olsa bile desenleri analiz edebilir. Örneğin, yaklaşım, Strategy rolü (soyut sınıf veya arayüz) ile Somut Strateji alt sınıf rolü arasında bir orta düzey kalıtım seviyesi olsa bile, Strategy deseninin bir oluşumunu belirleyebilir.

Tsantalis'in yaklaşımı, statik yapılarının tüm önemli yönlerini temsil eden bir dizi matris kullanmaktır. Örüntülerin saptanması için hem sistemi hem de örüntü grafiğini girdi olarak alan ve köşeleri arasındaki benzerlik puanlarını hesaplayan bir grafik benzerlik algoritması kullanmışlardır. Bu yaklaşımın en büyük avantajı, kalıpları yalnızca temel formlarında (genellikle literatürde bulunan) değil, aynı zamanda bunların değiştirilmiş versiyonlarını da (modifikasyonun bir desen özelliği ile sınırlı olduğu göz önüne alındığında) tespit etme yeteneğidir. Bu önemli bir önkoşuldur çünkü herhangi bir tasarım deseni sayısız varyasyonla uygulanabilir.

Desen tespitinde en önemli zorluklardan biri, büyük yazılım sistemleri için keşif alanının boyutudur. Çok sayıda sistem sınıfı ve sınıfların belirli bir tasarım deseninde oynayabileceği çoklu roller nedeniyle bir birleşimsel patlama meydana gelebilir. Benzerlik algoritmasının tüm sisteme uygulanması, algoritmanın yavaş yakınsaması nedeniyle verimlilik sorunlarına yol açacaktır. Dahası, gerçek bir desen adayını oluşturan sonuçların birleştirilmesindeki zorluk, doğruluk açısından sorunlar yaratabilir. Bu sorunu ele almak için önerilen yaklaşım, çoğu desen en az bir soyut sınıf / arayüz ve onun soyundan gelenleri içerdiğinden, her tasarım deseninin bir veya daha fazla kalıtım hiyerarşisinde yer aldığı gerçeğinden yararlanır. Sonuç olarak, benzerlik algoritması tüm sistem yerine daha küçük alt sistemlere uygulanacak şekilde, sistem hiyerarşi kümelerine (iletişim hiyerarşi çiftleri) bölünür.

Bir diğer önemli konu ise tasarım desenleri listesinin sürekli genişliyor olmasıdır. Sonuç olarak, bir tespit metodolojisi belirli modellere dayanmamalıdır. Herhangi bir algoritma, şimdiye kadar icat edilmemiş olabilecek kullanıcı tanımlı desenlere uygulanabilirliğini genelleştirebilmelidir.

Kullanılan benzerlik algoritması, belirli bir statik yapıdan yararlanacak herhangi bir buluşsal yönteme dayanmadığından, önerilen metodoloji herhangi bir model girdisine uygulanabilir.

Önerilen metodoloji ilk olarak, tasarım desenleri kapsamlı ve sistematik olarak kullanan açık kaynaklı projeler olan JHotDraw, JRefactory ve JUnit üzerinde değerlendirilmiştir. Sonuçlar, bu sistemlerin dahili ve harici belgelerine göre doğrulanmıştır. İncelenen tasarım örüntüleri için yanlış negatif sayısı sınırlıyken, yanlış pozitif bulunamamıştır.

Daha sonra ilgili konuda yazılan makalelerde oldukça yoğun bir şekilde kullanıldığı görülmüştür.

2.4.2 PINOT (Pattern INference recOvery Tool)

Desen Çıkarım Kurtarma Aracı (PINOT) (Shi ve Olsson, 2006), GoF kataloğundaki tüm yapısal ve davranış desenlerinden oluşumları tanımlayabilen bir örüntü algılama yaklaşımıdır. Saptama, desenlerin yapısal yönlerine (sınıflar arası ilişkilerden türetilen) vurgu yapmakla kalmaz, aynı zamanda davranışsal yönlerini dikkate alma ihtiyacını da kabul eder. Arama alanını belirli yöntemlere daraltmak için sınıflar arası analiz yapıldıktan sonra, kontrol akışı ve veri akışı açısından her aday yöntemin gövdesine daha fazla statik davranış analizi uygulanır.

PINOT, gömülü bir desen analizi motoruyla Jikes'ten (C++ ile yazılmış bir açık kaynak Java derleyicisi) oluşturulmuştur. Bir desen tespit aracının temeli olarak bir derleyici kullanmanın birçok avantajı vardır. Bir derleyici, sınıflar arası ve statik davranış analizlerini kolaylaştıran sembol tabloları ve AST (abstract syntax tree) oluşturur. Derleyiciler ayrıca kalıp analizine yardımcı olan bazı anlamsal kontroller gerçekleştirir. Örneğin, yerel bir değişken bir genel değişkeni gölgelediğinde (ile aynı ada sahip olduğunda) Jikes, delegasyon ilişkilerinin belirsizliğini gidermeye yardımcı olan uyarılar yazdırır. En önemlisi, derleme hataları, sembol tablolarının ve AST'nin eksikliğini yansıtır ve bu da yanlış desen algılama sonuçlarına neden olur. Bununla birlikte, FUJABA ve PTIDEJ gibi bazı araçlar, tamamlanmamış kaynaktan kalıpları kısmen (belirsiz bir sayı ile) tespit edebilmektedir. Bu tür araçlar, desen tespiti, çalışma sırasında desen oluşturma ve dahil etme gibi ileri yazılım mühendisliğinin bir parçası olarak kullanılırsa arzu edilebilir. Normal şartlarda, desen tespiti, doğruluğun hayati önem taşıdığı tersine mühendislik için ayrılmıştır.

PINOT, belirli bir desen için tespit sürecini, bu modeli belirlemede en etkili olana göre başlatır. Desen tespiti, en düşük olasılıklı sınıfları veya yöntemleri budayarak arama alanını azaltır. Bir desen algılama aracının eksiksizliği, desen uygulama varyantlarını tanıma yeteneği ile belirlenir. Pratik nedenlerden dolayı PINOT, pratikte kullanılan yaygın uygulama varyantlarını tespit etmeye odaklanır. Bu nedenle, bazı davranış analizi teknikleri, her davranışa dayalı desene tam olarak uygulanmamaktadır.

PINOT, Java AWT, JHotDraw, Swing, Apache Ant ve diğer birçok program ve pakette kalıpları tespit etmede başarıyla kullanılmıştır.

2.4.3 Literatürdeki Diğer Araçlar

Tasarım modellerini tespit etmeye yönelik ilk yaklaşımlardan biri 1996'da Kraemer ve Prechelt (1996) tarafından sunulmuştur. Doğrudan C++ kaynak kodundan tasarım desenlerini tespit ederek yazılım sürdürülebilirliğini geliştirmeye çalışmışlardır. Tasarım desenleri, bir C++ kod deposunu sorgulamak için kullanılan Prolog kuralları olarak temsil edilir. Tespit sürecinde beş yapısal tasarım desenine odaklanmışlardır. Bunlar; Adapter, Bridge, Composite, Decorator ve Proxy'dir. Kraemer ve Prechelt yaklaşımı, NME, ACD, LEDA ve zApp sınıf kütüphanesi olan dört gerçek projeye uygulamışlar ve hassasiyetin %14-50 arasında olduğunu bildirmişlerdir.

CrocoPat, Cottbus Teknik Üniversitesi'nde (Beyer ve Lewerentz, 2003) geliştirilmiş bir tasarım desen tespiti aracıdır. İlişkiler açısından yazılım meta modelini temsil eder. Tasarım desenleri ilişkisel ifadelerle tanımlanır. CrocoPat'i oluşturma ana motivasyonu, önceki algılama araçlarının performans sorununu ele almaktır. CrocoPat tarafından sunulan metamodel, nesne yönelimli programı paketlere, sınıflara, yöntemlere ve özniteliklere ayırır. CrocoPat yalnızca performans açısından değerlendirilir. Bildirilen sonuçlar yalnızca Mozilla, JWAM ve wxWindows'daki Composite ve Mediator örneklerinin algılanmasını gösterir.

PTIDEJ (Pattern Traces Identification, Detection and Enhancement in Java) Montreal Üniversitesi'nde Eclipse platformu altında Java kullanılarak geliştirilmiştir. PTIDEJ tam bir tersine mühendislik aracına dönüşmüştür. PTIDEJ, tasarım desenleri, mikro desenler ve deyim desenleri (Gueheneuc ve Jussien, 1994; Gueheneuc ve diğ., 2004) için çeşitli tanımlama algoritmaları içerir. PTIDEJ, tasarım örüntü algılamasını, değişken atama aşamasında kararların alındığı bir kısıtlama tatmin problemi (CSP) olarak görür. PTIDEJ, tasarım motiflerine benzer mikro mimarîleri tanımlamak için açıklama tabanlı kısıt programlamayı

kullanır. Mikro mimarî, nesne yönelimli bir programın bir dizi sınıfının organizasyonunu(yapısını) tanımlar.

Örüntü Sorgulama ve Tanıma Sistemi (SPQR) (Smith ve Stotts, 2004), Carolina Üniversitesi'nde geliştirilen C++ kaynak kodunda temel tasarım örüntü algılaması için bir araç setidir. SPQR, sınıflar ve nesneler arasındaki yapısal ve davranışsal ilişkileri kodlamak için mantıksal bir çıkarım sistemi kullanır. SPQR yalnızca Killer Widget Uygulamasına uygulanır ve Dekoratör tasarım deseni çıkarılır. SPQR sonuçları yalnızca verimlilik (CPU süreleri ve bellek tüketimi) açısından doğrulanır.

DeMIMA (Gueheneuc ve Antoniol, 2008) (Tasarım Motifi Tanımlama: Çok Katmanlı Yaklaşım), kaynak koddaki tasarım motiflerine benzer mikro mimarîleri tanımlayan, Montreal Üniversitesi'nde geliştirilmiş yarı otomatik bir araçtır. DeMIMA, kaynak kodun soyut modeli ve sınıf ilişkileri oluşturmak için iki katman ve oluşturulan soyut modelden tasarım desenlerini tanımak için bir katman olmak üzere üç katman içerir. DeMIMA, PTIDEJ çerçevesinin üstüne Java'da uygulanmıştır. Ek olarak, DeMIMA kısıtlama tatmin problemini çözmek için açıklamaya dayalı kısıt programlamayı kullanır. DeMIMA, tasarım motiflerine benzer mikro mimarîleri, onları desenin katılımcı sınıfları arasındaki ilişkileri yansıtan kısıtlamalara dönüştürerek tanımlar. Kullanılan kısıtlamalar kalıtım kısıtı, katı geçişli kalıtım kısıtı, geçişli kalıtım kısıtı, kullanım kısıtı, cehalet kısıtı ve yaratım kısıtıdır.

Design Patterns Recovery Environment (DPRE), Salerno Üniversitesi'nde geliştirilmiş bir tasarım deseni tanıma prototipidir (Lucia ve diğ., 2009). DPRE, yapısal tasarım desenlerini nesne yönelimli koddan algılamak için iki aşamalı bir yaklaşım kullanır. DPRE birinci aşama, ön analiz sırasında çıkarılan sınıf diyagramı bilgilerinin analiz edilmesiyle tasarım desenini, adaylarının tanımlandığı kaba taneli bir seviye sağlar. İkinci aşamada, tasarım deseni tanımlamasına katılan sınıfların kodları, karşılık gelen GoF kalıplarının kaynak koduna uygunluklarını kontrol etmek için incelenir. DPRE'nin etkinliği, % 62 ile % 97 arasında değişen hassasiyetle karakterize edilmiştir.

Zanoni, hem tasarım deseni örneklerinin algılanmasını hem de yazılım mimarîsi yeniden yapılandırma etkinliklerini destekleyen MARPLE (Metrics and Architecture Reconstruction Plug-in) adlı bir Eclipse eklentisini tanıtmıştır (Zanoni, 2012). MARPLE, "temel elemanlar" olarak adlandırılan alt bileşenlerin tespiti yoluyla tasarım desenleri tespitinin çeşitli

problemlerini çözmeye çalışır. MARPLE'nin mimarîsi, XML veri aktarımı yoluyla birbiriyle etkileşime giren beş ana modül içerir.

Alnusair ve diğ. (2014) tarafından sunulan yaklaşım, Sempatrec (SEMantic PATtern RECOVERY), tasarım desenlerinin yapısını ve davranışını yakalamak için kaynak kodun kavramsal bilgisini ve anlamsal kuralları temsil etmek için ontoloji formalizmini kullanır. Yaklaşımı uygulamak için Eclipse IDE için bir eklenti olarak Sempatrec adlı bir araç geliştirildi. Sempatrec, hedeflenen yazılımın Java bayt kodunu işler, bir RDF (Kaynak Tanımlama Çerçevesi) ontolojisi oluşturur ve ontolojiyi yerel olarak bir havuzda depolar. Spesifik olarak, Sempatrec, kaynak kodda bulunan kavramsal bilgi yapısının açık bir temsili sağlamak için bir kaynak kodu temsil ontolojisi (SCRO) üretir. Bununla birlikte, geliştirilen SCRO, bir tasarım deseni ontolojisi alt modelinin yaratılacağı tasarım deseni kurtarması için bir temel görevi görür. Bu alt model, SCRO'nun kelime dağarcığını genişletir ve her tasarım deseni için belirli bir ontoloji ile daha da genişletilmiş bir üst tasarım deseni ontolojisini içerir.

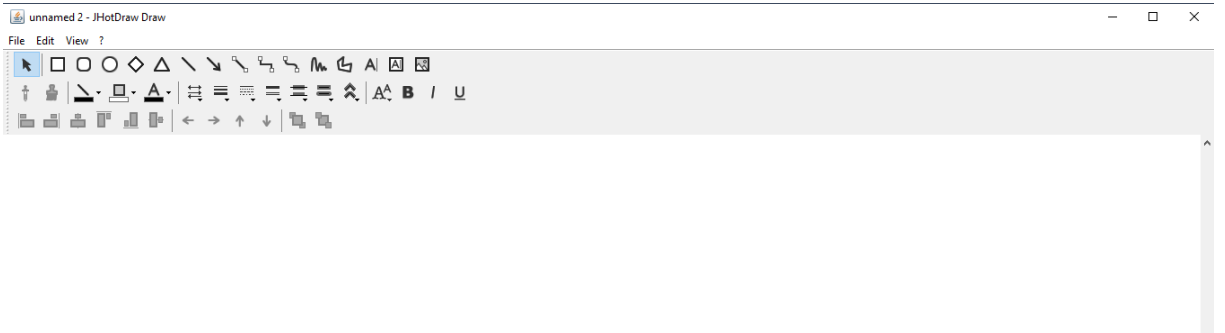
Bazı araçlar, JAVA AWT, JHotDraw, Swing, Apache Ant gibi iyi bilinen eski uygulamalar üzerinde deneyler yaptıktan sonra doğruluğunu iddia etmektedir. Bazı araçlar akıllıdır, karşılık gelen GoF modelinin varyantlarını otomatik olarak işlerler ancak performansları ve doğrulukları tartışmalıdır (Stencel ve Wegrzynowicz, 2008; Kniesel ve Binun, 2009). Kaynak kodun içinde, statik analiz, dinamik analiz, benzerlik puanlaması ve ayrıştırma gibi çoklu teknikler kullanarak çıkarabileceğimiz pek çok bilgi gizlidir. 23 iyi bilinen tasarım modelinin geliştirilmesi, amacı ve uygulanabilirliği aynı kalır, ancak çözüm yapısı ve değişim için çözüm uygulaması değişkenlik gösterebilir. Bu nedenle, yeni tasarım modellerini keşfetmek amaç değildir.

3. MALZEME VE YÖNTEM

Bu tez kapsamında JHotDraw 7.0.6 ve JHotDraw 7.6 versiyonları incelenmiştir. JHotDraw, yazılımında, ikinci bölümde açıklanan DPD ve PINOT araçları ile yazılımda kullanılan tasarım desenlerinin analizi yapılmıştır. Sonrasında yine aynı yazılımlar SonarQube aracı ile yazılım kalite metriklerinden güvenilirlik, güvenlik, sürdürülebilirlik ve kod karmaşıklığı yönünden değerlendirilmiştir.

3.1 JHOTDRAW

JHotDraw, GoF tasarım desenleri kullanılarak Java programlama dili ile yazılmış açık kaynaklı bir yazılımdır. Geliştiricilerin JHotDraw'ı açık kaynak kodlu yayınlamasının amaçlarından başlıcaları, JHotDraw'a dayalı yeni uygulamaların oluşturulmasını sağlamak, yeni ve gelişmiş özellikler ekleyebilmek, JHotDraw'ı yeni Java GUI araç setlerine taşıyabilmek, yeni tasarım kalıplarını ve yeniden düzenlemeleri belirleyebilmektir. JHotDraw yapılandırılmış çizim editörleri çerçevesi, eskizler, diyagramlar ve sanatsal çizimler için çizim editörlerini gerçekleştirmek için kullanılabilir. Çizimler hareketli ve etkileşimli olabilir. Bir veri modeliyle bir çizimi desteklemek mümkündür, bu da yapılandırılmış bir çizim düzenleyicisinin bir veri modeli için kullanıcı arayüzü olarak kullanılmasına izin verir. Çalışan JHotDraw aracı Şekil 3.1'de gösterilmiştir.



Şekil 3.1: JHotDraw

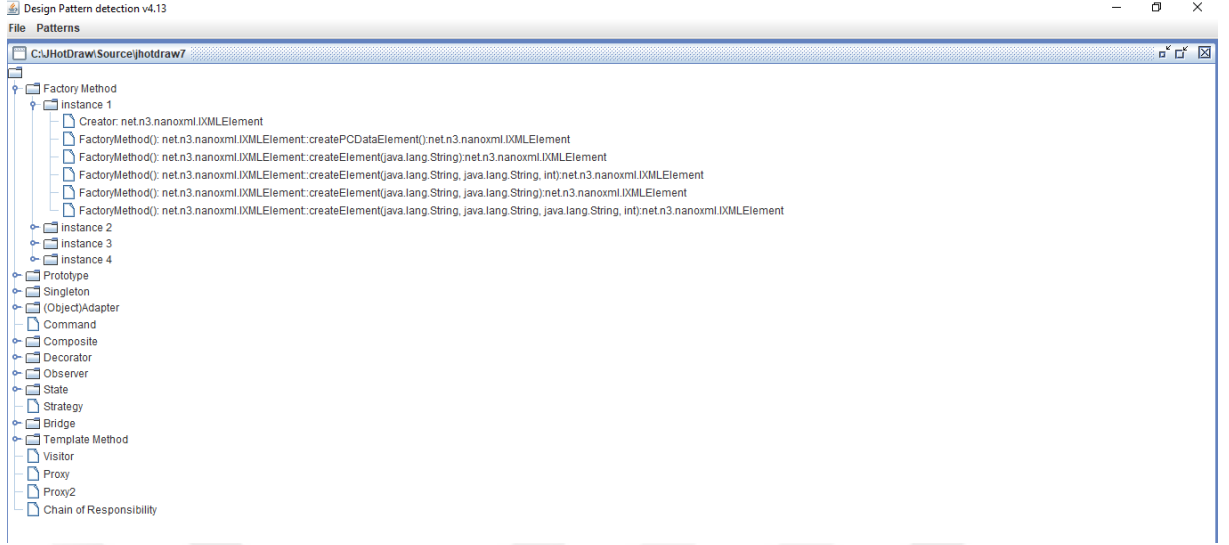
JHotDraw'ın yedi ana ve birçok ara sürümü yayınlanmıştır. Bu tez çalışmasında JHotDraw 7.0.6 ve JHotDraw 7.6 versiyonları üzerinde inceleme yapılmıştır. JHotDraw'ın bütün sürümleri <http://www.randelshofer.ch/oop/jhotdraw/> adresinde mevcuttur. Açık kaynak olarak

indirilen sürümler Apache NetBeans IDE 12.4'de derlenmiştir. Derlenen projeler kalite metriklerini ölçmek için SonarQube aracı, yazılım içerisinde kullanılan tasarım desenlerinin tespiti için DPD ve PINOT araçları kullanılmıştır.

3.2 DPD

Tsantalis ve diğ. (2006) tarafından, tasarım desenlerini tespit etme amacıyla geliştirilmiştir. Literatürdeki benzer çalışmalarda, aracın kullanımının öne çıktığı gözlemlenmiştir. DPD, yazılımda kullanılan tasarım desenlerini, kullanıldığı sınıfları ve aralarındaki ilişkilerinin ayrıntılı dökümünü, tersine mühendislik ve statik analiz yöntemlerini kullanarak çıkarır. Aracın tespit edebildiği tasarım desenleri aşağıdaki gibidir.

- Factory Method
- Prototype
- Singleton
- Adapter
- Command
- Composite
- Decorator
- Observer
- State
- Strategy
- Bridge
- Template Method
- Visitor
- Chain of Responsibility
- Proxy



Şekil 3.2: DPD Örnek Sonuç

DPD aracında bahsedilen JHotDraw’ın sürümlerinin derlenmiş versiyonlarının ayrı ayrı sonuçları alınmıştır. Alınan sonuçlar doğrultusunda tasarım desenlerinin kullanıldığı sınıflar incelenmiştir.

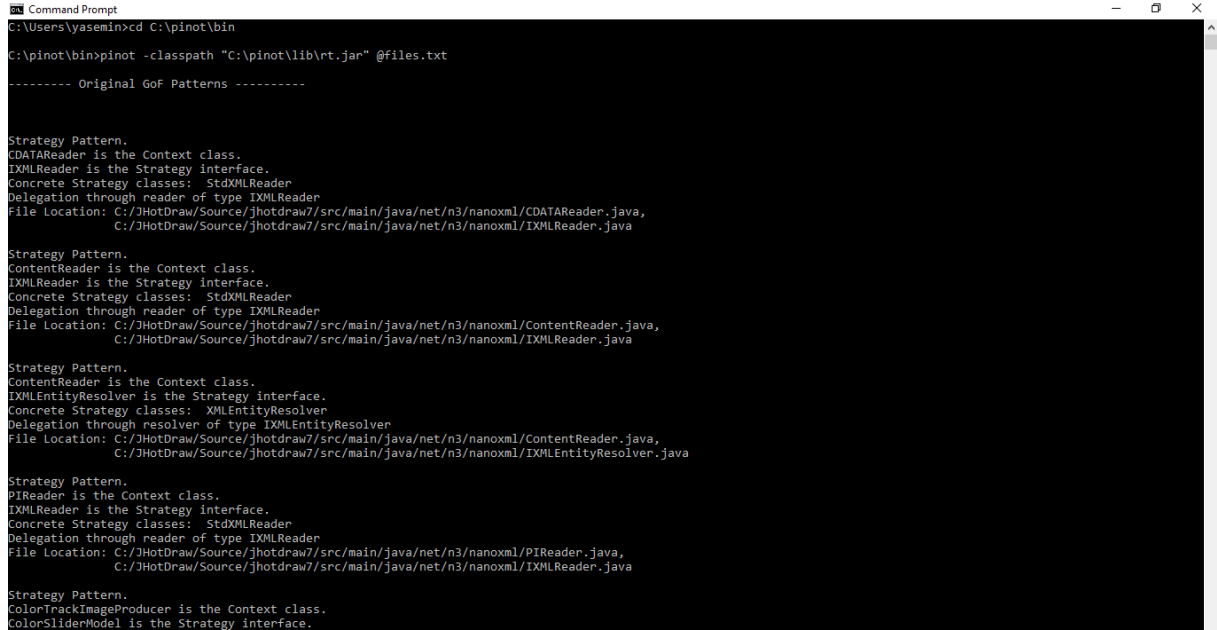
3.3 PINOT

PINOT tersine mühendislik tekniklerini kullanarak yazılımda kullanılan tasarım desenlerinin ayrıntılı sonucunu veren, Java yazılım dili ile yazılmış, açık kaynak kodlu bir yazılımdır. Aracın kaynak kodu, <https://www.cs.ucdavis.edu/~shini/research/pinot/> sitesinden indirildikten sonra derlenmiştir. Araç, gerekli yapılandırmalar yapıldıktan sonra komut satırı üzerinden projenin Java dosyalarını baz alarak sonuçları gösterir. DPD gibi literatürde yapılan çalışmalarda tasarım desenleri tespiti için öne çıktığı görülmüştür. Aracın tespit edebildiği tasarım desenleri aşağıdaki gibidir.

- Abstract Factory
- Factory Method
- Singleton
- Adapter
- Bridge
- Composite

- Decorator
- Facade
- Flyweight
- Proxy
- Chain of Responsibility
- Mediator
- Observer
- State
- Strategy
- Template Method
- Visitor

PINOT'ta bahsedilen JHotDraw'ın 7.6 ve 7.0.6 sürümlerinin ayrı ayrı sonuçları alınmıştır. Alınan sonuçlar doğrultusunda tasarım desenlerinin kullanıldığı sınıflar incelenmiştir.



```

C:\Users\yasemin>cd C:\pinot\bin
C:\pinot\bin>pinot -classpath "C:\pinot\lib\rt.jar" @files.txt
----- Original GoF Patterns -----

Strategy Pattern.
CDATAReader is the Context class.
IXMLReader is the Strategy interface.
Concrete Strategy classes: StdXMLReader
Delegation through reader of type IXMLReader
File Location: C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/CDATAReader.java,
C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/IXMLReader.java

Strategy Pattern.
ContentReader is the Context class.
IXMLReader is the Strategy interface.
Concrete Strategy classes: StdXMLReader
Delegation through reader of type IXMLReader
File Location: C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/ContentReader.java,
C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/IXMLReader.java

Strategy Pattern.
ContentReader is the Context class.
IXMLEntityResolver is the Strategy interface.
Concrete Strategy classes: XMLEntityResolver
Delegation through resolver of type IXMLEntityResolver
File Location: C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/ContentReader.java,
C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/IXMLEntityResolver.java

Strategy Pattern.
PIReader is the Context class.
IXMLReader is the Strategy interface.
Concrete Strategy classes: StdXMLReader
Delegation through reader of type IXMLReader
File Location: C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/PIReader.java,
C:/JHotDraw/Source/jhotdraw7/src/main/java/net/n3/nanoxml/IXMLReader.java

Strategy Pattern.
ColorTrackImageProducer is the Context class.
ColorSliderModel is the Strategy interface.

```

Şekil 3.3: PINOT Örnek Sonuç

3.4 SONARQUBE

SonarQube, yirminin üzerinde programlama dilindeki hataları, kod kokularını (code smell) ve güvenlik açıklarını tespit etmek için statik kod analizi ile otomatik incelemeler yapan açık kaynaklı bir platformdur. Günümüzde, büyük ölçekli firmalarda, kaynak koddaki hataların bulunması ve kod kalitesinin sağlanması için kullanılmaktadır. Bu tez kapsamında, Community versiyonu kullanılmıştır. SonarQube'ın kurulumu sunucu ve kod taraması olarak iki adımdadır. <https://www.sonarqube.org/> sitesinden indirilen kaynak kodda farklı işletim sistemlerinde çalışabilen dosyalar bulunur. Proje kapsamında Windows işletim sistemi kullanıldığından Windows için olan bat dosyası ile SonarQube sunucusu çalıştırılmıştır. Yerel bilgisayarda kurulumu sağlanan araçta, 9000 portu üzerinden projelerin statik analiz sonuçları görüntülenir. SonarQube, statik analiz sonuçlarını raporlar ve kodun iyileştirilmesine katkıda bulunur.

SonarQube ile JHotDraw 7.6 versiyonu için 76.886 kod satırı ve JHotDraw 7.0.6 için 28.551 kod satırı incelenmiş güvenlik, güvenilirlik, sürdürülebilirlik ve kod karmaşıklığı yönlerinden sonuçlar alınmıştır.

4. BULGULAR

Bu bölümde bulgular, tasarım deseni tespit araçlarından çıkan sonuçlar ve kalite metrikleri değerlendirmesinin bulunduğu iki başlık altında incelenecektir.

4.1 TASARIM DESENLERİ İNCELEMESİ

JHotDraw 7.0.6 ve JHotDraw 7.6 açık kaynak kodlu yazılımın farklı sürümlerinin DPD ve PINOT araçlarından çıkan, tasarım desenlerinin kullanımının sayısal sonuçları Tablo 4.1 ve Tablo 4.2'deki gibidir.

Tasarım Deseni	JHotDraw 7.0.6 DPD	JHotDraw 7.0.6 PINOT
Abstract Factory	-	6
Factory Method	3	6
Singleton	2	0
Adapter	26	1
Bridge	13	5
Composite	2	0
Decorator	3	1
Facade	-	5
Flyweight	-	1
Proxy	0	2
Chain of Responsibility	0	0
Mediator	-	18
Observer	2	0
State	51	11
Strategy	0	3
Template Method	12	1
Visitor	0	0
Prototype	9	-
Command	0	-

Tablo 4.1: JHotDraw 7.0.6 Tasarım Desenleri Analizi

Tasarım Deseni	JHotDraw 7.6 DPD	JHotDraw 7.6 PINOT
Abstract Factory	-	1
Factory Method	4	1
Singleton	11	-
Adapter	21	-
Bridge	20	-
Composite	6	-
Decorator	5	-
Facade	-	1
Flyweight	-	5
Proxy	-	-
Chain of Responsibility	-	-
Mediator	-	-
Observer	1	-
State	72	-
Strategy	-	5
Template Method	16	-
Visitor	-	-
Prototype	11	-
Command	-	-

Tablo 4.2: JHotDraw 7.6 Tasarım Desenleri Analizi

Tablodan da görüldüğü gibi büyük projelerde birçok problemin çözümü için uygun tasarım desenleri seçilip birlikte kullanılabilir. Günümüzde melez yaklaşım tekniği oldukça yaygın olarak kullanılmaktadır.

Çıkan analiz sonuçlarına bakıldığında bulunan sonuçların farklı olduğu görülmüştür. Her iki araç da statik analiz gerçekleştiriyor olsa da araçların sonuçlarının aynı olması beklenmez.

- DPD yöntem çağrılarını araştırırken PINOT araştırmaz.
- DPD, nesne oluşturmayı araştırırken PINOT bunu yapmaz.
- PINOT, kontrol akışını ve veri akışını araştırırken, DPD bunu yapmaz.
- PINOT, desen oluşumlarını yalnızca orijinal sürümlerinde tanımlar. DPD, sapmaları da tanımlar.

4.2 KALİTE METRİKLERİ İNCELEMESİ

Bölüm 2’de bahsedilen SonarQube aracının sonuçları Tablo 4.3’te gösterilmiştir. Güvenlik, güvenilirlik, sürdürülebilirlik ve karmaşıklık sonuçlarına bakılarak, tasarım desenlerinin yazılımlara doğru problemlere karşı doğru tasarım deseninin entegre edilmesi oldukça önemli olduğu görülmüştür. Karmaşıklığın artmış olması yazılım tasarım desenlerinin dezavantajları içerisinde değerlendirilir.

JHotDraw 7.6 sürüm sonuçlarına bakıldığında karmaşıklığın arttığı ancak diğer metriklerde oldukça iyi sonuç aldığı gözlemlenmiştir.

JHotDraw 7.0.6 sürümü sonucuna bakıldığında, tasarım desenlerinin doğru şekilde kullanılmasının kalite metrikleri yönünden oldukça önemli bir yere sahip olduğu gözlemlenmiştir. Tez çalışmasının diğer bölümlerinde değinildiği gibi tasarım desenlerini yazılımlara entegre etmenin tek başına yeterli olmadığı, problemlere uygun doğru tasarım desenlerinin kullanılmasının önemi görülmektedir. Karmaşıklık her ne kadar düşük olsa da güvenilirlik ve güvenlik metrikleri yönünden zayıf kaldığı görülmektedir.

Fonksiyonel Olmayan Kalite Metriği	JHotDraw 7.0.6	JHotDraw 7.6
Güvenilirlik (Reliability)	G	A
Güvenlik (Security)	E	A
Sürdürülebilirlik (Maintainability)	A	A
Döngüsel Karmaşıklık(Cyclomatic Complexity)	5,898	15,032
Bilişsel Karmaşıklık (Cognitive Complexity)	4,691	13,815

Tablo 4.3: JHotDraw 7.0.6 ve JHotDraw 7.6 Kalite Metrikleri Analizi

5. TARTIŞMA VE SONUÇ

GoF tasarım desenlerinin yazılımlara entegre edilmesi, yazılımın güvenliği, güvenilirliği, sürdürülebilirliği metrikleri yönlerinden önemli bir yere sahiptir. Bu metriklerin yanında temiz kod yazımı, kodun okunabilirliği, büyük yazılım projelerinde ortak çalışma kolaylığının sağlanabilmesi gibi yönler de oldukça önemlidir.

Günümüzde GoF tasarım desenlerinin tek başlarına kullanımı projelerin büyüklükleri ve farklı problemleri tek tasarım deseni kullanımı ile çözilememesi bakımından yaygın değildir. Bunun yerine melez yaklaşım yani birçok farklı tasarım deseninin bir arada kullanımı yaygın olarak kullanılmaktadır. İncelenen açık kaynak kodlu yazılımlarda GoF tasarım desenlerinin kullanıldığı çeşitli araçlarla analiz edilmiştir. Yazılım projelerinde tasarım desenlerinin kullanımını yaklaşımının doğruluğunu test etmek için günümüzde yaygın olarak kullanılan yazılımları kalite metrikleri yönünden değerlendiren SonarQube aracı kullanılmıştır. Bu araçtan çıkan sonuçlara bakıldığında, yazılım tasarım desenlerinin doğru şekilde kullanımının daha önemli olduğu görülür.

Gelecek çalışmalarda, GoF tasarım desenlerinden türeyen yeni tasarım desenleri yaklaşımları incelenebilir ve olası problemlere çözüm önerileri analiz edilebilir.

KAYNAKLAR

[Al-Obeidallah, M., Petridis, M. and Kapetanakis, S., 2016, A Survey on Design Pattern Detection Approaches, *International Journal of Software Engineering (IJSE)*, 7, 41–59.

Alexander, C., Ishikawa, S. and Silverstein, M., 1977, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, ISBN: 74-22874.

Alnusair, A., Zhao, T. and Yan, G., 2014, Rule-based detection of design patterns in program code, *International Journal on Software Tools for Technology Transfer*, 16(3), 315–334.

Ampatzoglou, A., Kritikos, A., Kakarontzas, G. and Stamelos, I., 2011, An Empirical Investigation on the Reusability of Design Patterns and Software Packages, *Journal of Systems and Software*, 84(12), 2265–2283.

Ampatzoglou, A., Chatzigeorgiou, S., Charalampidou, S. and Avgeriou, P., 2015, The Effect of GoF Design Patterns on Stability: A case study, *IEEE Transactions on Software Engineering*, 1 August 2015, IEEE, 41(8), 781–802.

Ampatzoglou, A., Charalampidou, S. and Stamelos, I., 2013, Research state of the art on GoF design patterns: A mapping study, *Journal of Systems and Software*, 86(7), 1945–1964.

Ampatzoglou, A. and Chatzigeorgiou, A., 2007, Evaluation of object-oriented design patterns in game development, *Information and Software Technology*, 49(5), 445–454.

Antoniol, G., Fiutem, R. and Cristoforetti, L., 1998, Design pattern recovery in object-oriented software, *6th International Workshop on Program Comprehension*, IEEE, 153–160.

Belachew, E. B., Gobena, F. A. and Nigatu, S. T., 2018, Analysis of Software Quality Using Software Metrics, *International Journal on Computational Science & Applications (IJCSA)*, 8(4), 11–20.

Beyer, D. and Lewerentz, C., 2003, CrocoPat: Efficient pattern analysis in object-oriented programs, *11th IEEE International Workshop on Program Comprehension*, IEEE, 294–295.

Campbell, G. A., 2021, *Cognitive Complexity*, SonarSource SA.

Elish, M. O. and Mawal A., M., 2015, Quantitative analysis of fault density in design patterns: An empirical study, *Information and Software Technology*, 66, 58–72.

Farooq, A., 2017, *Evaluating Impact of Design Patterns on Software Maintainability and Performance*, Thesis (MSc), University of Oslo Institute of Informatics.

Feitosa, D., Ampatzoglou, A., Avgeriou, P., Chatzigeorgiou, A., Nakagawa, E., 2019, What can violations of good practices tell about the relationship between GoF patterns and run-time quality attributes?, *Information and Software Technology*, 105, 1–16.

Fowler, M., 1999, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Canada, ISBN: 978-0201485677.

- Gahlyan, P. and Singh, S. N., 2018, Analysis of Catalogue of GoF Software Design Patterns, *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, IEEE, 814–818.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1994, *Elements of Reusable Object-Oriented Software*, Addison-Wesley, United States, ISBN: 0-201-63361-2.
- Gomes, I., Morgado, P. and Gomes, T., 2009, An overview on the Static Code Analysis approach in Software Development, *Faculdade de Engenharia da Universidade do Porto*, 1–16.
- Gueheneuc, Y. G. and Jussien, N., 2011, Using explanations for design patterns identification, *IJCAI Workshop on Modelling and Solving Problems with Constraints*, IJCAI, 57–64.
- Gueheneuc, Y. G. and Antoniol, G., 2008, DeMIMA: A Multilayered Approach for Design Pattern Identification, *IEEE Transactions on Software Engineering*, 34(5), pp. 667–684.
- Gueheneuc, Y. G., Guyomarch, J. Y. and Sahraoui, H., 2010, Improving design-pattern identification: A new approach and an exploratory study, *Software Quality Journal*, 18(1), 145–174.
- Gueheneuc, Y. G., Sahraoui, H. and Zaidi, F., 2004, Fingerprinting design patterns, *Proceedings - Working Conference on Reverse Engineering, WCRE*, 172–181.
- Hasheminejad, S. and Jalili, S., 2012, Design patterns selection: An automatic two-phase method, *Journal of Systems and Software*, 85(2), 408–424..
- Hassan, S. I., 2015, Effect of SOLID Design Principles on Quality of Software An Empirical Assessment, *International Journal of Scientific & Engineering Research*, 6(4), 1321–1324.
- Hussain, A., Mkpojiogu, E. and Kamal, F., 2016, The Role of Requirements in the Success or Failure of Software Projects, *EJ Econjournals*, 6(7), 6–7.
- Hussain, S., Keung, J. and Khan, A. A., 2017, The Effect of Gang-of-Four Design Patterns Usage on Design Quality Attributes, *Software Quality, Reliability and Security (QRS)*, 263–273.
- Intakosum, S. and Muangon, W., 2007, Retrieving Model for Design Patterns, *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 3(1), 51–55.
- Khomh, F. and Gueheneuc, Y. G., 2008, Do Design Patterns Impact Software Quality Positively?, *2008 12th European Conference on Software Maintenance and Reengineering*, 274–278.
- Kniessel, G. and Binun, A., 2009, Standing on the shoulders of giants - A data fusion approach to design pattern detection, *IEEE International Conference on Program Comprehension*, IEEE, 208–217.
- Kramer, C. and Prechelt, L., 1996, Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software, *4rd Working Conference on Reverse Engineering*, IEEE,

208-215.

Lucia, A., Deufemia V., Gravino, C. and Risi, M., 2009, Design pattern recovery through visual language parsing and source code analysis, *Journal of Systems and Software*, 82(7), 1177–1193.

Mladenova, T., 2020, Software Quality Metrics – Research , Analysis and Recommendation, *2020 International Conference Automatics and Informatics (ICAI)*, IEEE, 1-5.

Moon, S. Y., Park, B. K. and Kim, R. Y. C., 2016, Code Complexity on before and after Applying Design Pattern through SW Visualization, *2016 International Conference on Platform Technology and Service (PlatCon)*, IEEE, 2–6.

Neill, C. J., 2003, Leveraging object-orientation for real-time imaging systems, *Real-Time Imaging*, 9(6), 423–432.

Nikolaeva, D., Bozhikova, V. and Stoeva, M., 2019, A simple approach to design patterns identification in programming code', *2019 28th International Scientific Conference Electronics*, IEEE, 1-4.

Oktafiani, I., Hendradjay, B., 2018, Software Metrics Proposal for Conformity Checking of Class Diagram to SOLID Design Principles, *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, IEEE, 1–6.

Prechelt, L., Unger, B., Tichy, W. F., Brossler, P. and Votta, L.G., 2001, A controlled experiment in maintenance comparing design patterns to simpler solutions, *IEEE Transactions on Software Engineering*, 27(12), 1134–1144.

Rahmati, R., Rasoolzadegan, A. and Dehkordy, D. T., 2019, An Automated Method for Selecting GoF Design Patterns, *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, IEEE, 345–350.

Rasool, G. and Mader, P., 2011, Flexible design pattern detection based on feature types, *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, IEEE, 243–252.

Riehle, D., 2011, Lessons Learned from Using Design Patterns in Industry Projects, *Transactions on Pattern Languages of Programming*, 2, 1–15.

Sabatucci, L., Cossentino, M. and Susi, A., 2015, A goal-oriented approach for representing and using design patterns', *Journal of Systems and Software*, 110, 36–154.

Shi, N. and Olsson, R. A., 2006, Reverse engineering of design patterns from Java source code, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, IEEE, 123–132.

Smith, J. M. and Stotts, D., 2004, SPQR: flexible automated design pattern extraction from source code, IEEE, 215–224.

Smith, S., 2011, *Dynamically Recommending Design Patterns*, Thesis (BSc), Stetson University.

- Society, I. C., 2014, *Guide to the Software Engineering Body of Knowledge Version 3.0 (SWEBOK Guide V3.0)*, IEEE Computer Society, ISBN: 978-0-7695-5166-1.
- Stencel, K. and Wegrzynowicz, P., 2008, Detection of Diverse Design Pattern Variants, *2008 15th Asia-Pacific Software Engineering Conference*, 25–32.
- Tsantalis, N., Chatzigeorgiou, A., Stephanides, G. and Halkidis S.T., 2006, Design Pattern Detection Using Similarity Scoring, *IEEE Transactions on Software Engineering*, 32(11), 896–909.
- Vokac, M., Tichy, W., Sjoberg, D., Arisholm, E. and Aldrin, M., 2004, A controlled experiment comparing the maintainability of programs designed with and without design patterns - A replication in a real programming environment, *Empirical Software Engineering*, 9(3), 149–195.
- Waheed, A., Rasool, G., Ubaid, S. and Ghaffar, F., 2016, Discovery of design patterns variants for quality software development, *2016 International Conference on Intelligent Systems Engineering (ICISE)*, IEEE, 185–191.
- Zanoni, M., 2012, *Data mining techniques for design pattern detection*, Thesis(PhD), Universita degli Studi di Milano.
- Zdun, U., 2007, Systematic pattern selection using pattern language grammars and design space analysis, *Software - Practice and Experience*, 37(7), 983–1016.
- Zhang, C., 2011, *An Empirical Assessment of the Software Design Pattern Concept An Empirical Assessment of the Software Design Pattern Concept*, Thesis(PhD), Durham University.
- Zhang, C. and Budgen, D., 2012, What Do We Know about the Effectiveness of Software Design Patterns?, *IEEE Transactions on Software Engineering*, 38(5), 1213–1231.
- Zhang, C. and Budgen, D., 2013, A survey of experienced user perceptions about software design patterns, *Information and Software Technology*, 55(5), 822–835.

EKLER

|

