YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

TASARIM KALIPLARI KULLANARAK ÇERÇEVE GELİŞTİRME

Bilgisayar Mühendisi Rıza HORASAN

FBE Bilgisayar Mühendisliği Anabilim Dalında Hazırlanan

YÜKSEK LİSANS TEZİ

Tez Danışmanı : Prof. Dr. Oya KALIPSIZ (YTÜ)

İSTANBUL, 2007

İÇİNDEKİLER

Sayfa

KISALTMA LİSTESİ iv

ŞEKİL LİSTESİ v

ÇİZELGE LİSTESİ vi

ÖNSÖZ vii

ÖZET viii

ABSTRACT ix

1. GİRİŞ 1

2. YAZILIM ÇERÇEVELERİ 3

2.1 Yazılım Çerçevesi Nedir? 3
2.2 Yazılım Çerçevelerinin Özellikleri 4
2.2.1 Tekrar Kullanılabilirlik 4
2.2.2 Modülerlik 4
2.2.3 Genişletilebilirlik 5
2.2.4 Basitlik ve Kullanılabilirlik 5
2.2.5 Bakım Yapılabilirlik 5

3. YAZILIM ÇERÇEVESİ GELİŞTİRME 6

3.1 Çerçevelerin Teknik Özelliklerine Göre Sınıflandırılması 6
3.1.1 Beyaz-Kutu Çerçeveler 6
3.1.2 Siyah-Kutu Çerçeveler 7
3.1.3 Siyah-Kutu ve Beyaz-Kutu Çerçevelerin Karşılaştırılması 7
3.1.3.1 Miras Alma Yaklaşımı 8
3.1.3.2 Birleştirme Yaklaşımı 9
3.1.4 Gri-Kutu Çerçeveler 11
3.2 Çerçevelerin Etki Alanlarına Göre Sınıflandırılmaları 12
3.3 Çerçeve Katmanları 13

4. TASARIM KALIPLARI 14

4.1 Tasarım Kalıplarının Sınıflandırılması 14
4.1.1 Yaratımsal Kalıplar 15
4.1.2 Yapısal Kalıplar 17
4.1.3 Davranışsal Kalıplar 19
4.2 Tasarım Kalıpları Kullanmanın Etkileri 21
4.3 Tasarım Kalıpları ve Yazılım Çerçevesi Geliştirme 22

5. İLGİLİ ÇALIŞMALAR 24

6. ÖRNEK YAZILIM ÇERÇEVESİ 26

6.1 Üniversitelerin Finans Birimleri 26
6.2 Finans Çerçevesinin Temel Özellikleri 26
6.2.1 Genel Çerçeve Mimarisi 27
6.3 Çerçevenin Finans Alanına Özel Yapıları 27

6.3.1 Para Modülü 27
6.3.1.1 Para Modülü Tasarımı 28
6.3.2 Personel Modülü 29
6.4 Çerçevenin Alan Bağımsız Yapıları 31
6.4.1 Doğrulama ve Yetkilendirme 31
6.4.1.1 .NET ve Doğrulama ve Yetkilendirme Yapısı 31
6.4.1.2 Doğrulama ve Yetkilendirme ve Soyut Fabrika Tasarım Kalıbı 31
6.4.2 Günlük Tutma Sistemi 32
6.4.2.1 Günlük Tutma Sistemi Tasarımı 33
6.4.3 Raporlama Sistemi 34
6.4.3.1 Raporlama Yapısı ve Köprü Tasarım Kalıbı 35
6.4.4 Menü Yapısı 36

7. TASARIM KALIPLARININ ÇERÇEVE ÖZELLİKLERİNE ETKİLERİ 38

7.1 Modüller Arası İlişkiler ve Çerçevenin Sağlamlığı 38
7.2 Tekrar Kullanılabilirlik 39
7.3 Modülerlik 40
7.4 Basitlik ve Kullanılabilirlik 41
7.5 Bakım Yapılabilirlik 41

8. Sonuçlar ve Tartışmalar 42

KAYNAKLAR 44

ÖZGEÇMİŞ 46

KISALTMA LİSTESİ

ADS	Active Directory Servisi
CBO	Coupling Between Object Classes
DIT	Depth of Inheritance Tree
DYS	Doğrulama ve Yetkilendirme Servisi
İTÜ	İstanbul Teknik Üniversitesi
JFC	Java Foundation Classes
MFC	Microsoft Foundation Classes
NOC	Number of Children
VOD	Video on Demand

ŞEKİL LİSTESİ

Sayfa

Şekil 3.1 Beyaz kutu çerçeveler 6
Şekil 3.2 Siyah kutu çerçeveler 7
Şekil 3.3 Gri kutu çerçeveler 12
Şekil 3.4 Çerçeve katmanları 13
Şekil 4.1 Geometrik şekiller tasarımının UML gösterimi 16
Şekil 4.2 Geometrik şekillerin soyut fabrika kalıbı ile tasarımı 16
Şekil 4.3 Raporlama yapısı tasarımı 18
Şekil 4.4 Raporlama yapısının dekoratör kalıbı ile tasarımı 19
Şekil 4.5 Örnek kullanıcı arayüzü 20
Şekil 6.1 Para modülü UML gösterimi 29
Şekil 6.2 Personel listesi yapısı ve sorumluluklar zinciri kalıbı 30
Şekil 6.3 Doğrulama ve yetkilendirme yapısı ve soyut fabrika kalıbı 32
Şekil 6.4 Günlük tutma sistemi tasarımı 34
Şekil 6.5 Raporlama modülü ve köprü kalıbı 35
Şekil 6.6 Menü yapısı tasarımı 37
Şekil 7.1 Modüller arası ilişkiler 38

ÇİZELGE LİSTESİ

Sayfa

Çizelge 6.1 Finans birimleri arasındaki benzer yapılar 26
Çizelge 7.1 Modüller arası etkileşim 39

ÖNSÖZ

İlk olarak, yüksek lisans öğrenimim ve tez çalışması süresince gösterdiği yakın ilgi ve katkılarından dolayı Prof. Dr. Oya Kalıpsız' a teşekkürü bir borç bilirim.

Çalışma boyunca verdikleri destek nedeni ile başta Yazılım Geliştirme Grubu olmak üzere bütün İTÜ Bilgi İşlem Dairesi çalışanlarına teşekkür ederim.

Ayrıca iş ve günlük hayattaki yardımlarından dolayı Mahmut Meral ve İsmail Özgür Demirel'e teşekkür ederim.

Benim için en önemlisi, yıllardır benden sevgi, destek ve güvenlerini eksik etmeyen aileme teşekkür ederim.

Ocak 2007 Rıza Horasan

ÖZET

Mühendislik çalışmalarının temel amaçlarından biri olan kaliteli ürünü hızlı bir şekilde oluşturma ve süreçlerdeki değişikliklere göre güncelleyebilme işi, yazılım mühendisliği kapsamında da ele alınmaktadır. Belirli bir alandaki yazılım ürünlerine temel oluşturan yazılım çerçeveleri kullanımı, kaliteli programların hızlı bir şekilde oluşturulmasına ve bakımının yapılabilmesine olanak sağlamaktadır. Ancak belirli bir alandaki pek çok uygulamaya temel oluşturma görevine sahip çerçevelerin, doğası gereği tekrar kullanılabilir, bakımı yapılabilir ve uygulamalar özelinde genişletilebilir olması gerekir. Bu özelliklere sahip bir çerçeve tasarlamak zor bir iştir. Çerçeve geliştirme sürecinde tasarım kalıpları kullanımı, süreci kolaylaştıracaktır ve bahsedilen özelliklere sahip bir tasarıma ulaşmamıza yardımcı olacaktır.

Bu çalışmada üniversitelerin finans birimleri için geliştirilecek yazılımlara temel oluşturacak bir çerçevenin, tasarım kalıpları kullanılarak geliştirilmesi incelenmiştir. Geliştirilen çerçevede uygulanan tasarım kalıplarının amaçları ve tasarımın tekrar kullanılabilirlik, modülerlik, kullanılabilirlik ve bakım yapılabilirlik özelliklerini nasıl etkilediği anlatılmıştır. Geliştirilen ve incelenen çerçeveden yola çıkarak çerçeve ve tasarım kalıpları kullanımının kaliteli yazılım ürünleri oluşturmada önemli bir role sahip olduğu anlatılmak istenmektedir.

Anahtar kelimeler: Yazılım çerçeveleri, çerçeve geliştirme, tasarım kalıpları, çerçeve kalite özellikleri

ABSTRACT

Creating and maintaning a product of good quality quickly is one of the main goals of the engineering applications. This goal is also examined in software engineering concepts. Using software frameworks, which provide the basic components for the software applications of the same domain, helps in creating and maintaining applications of good quality. However as the frameworks provide the main concepts for lots of applications, a framework must be reusable, maintainable and expandable for each application. Developing a framework which has these properties is a hard job. Using design patterns in framework development process, makes the process easier and helps achieving reusable, maintainable and expandable frameworks.

In this work, developing a design patterns based framework for the applications, which will be developed for the financial departments of the universities, is examined and a sample framework is developed. In the framework modules, the purposes of the chosen design patterns are explained and the effects of using design patterns on the reusability, maintainability and usability characteristics of the framework is also explained.

Keywords: Software frameworks, design patterns, framework development using design patterns , framework quality characteristics

1. GİRİŞ

Sürekli gelişen ve değişen iş dünyasının hızlı ilerlemesindeki en büyük destek yazılım ürünlerinin kullanılmasından gelmektedir. Yazılım ürünlerinin bu hızlı gelişime ayak uydurması, programların geliştirme ve güncellenme sürelerinin kısalırken kalitelerinin de artması ile mümkün olmaktadır.

Herhangi bir problem uzayı (bankacılık, hastane ve sağlık, lojistik hizmetleri vb.) için geliştirilen uygulamaların, ilgili problem uzayı dahilinde, kendi içlerinde barındırması gereken yapılar bulunmaktadır. Bu yapıların belirli özellikleri, kanunlar ve yönetmelikler çerçevesinde belirlenmesi sebebiyle, geliştirilen uygulamadan bağımsız olarak ilgili problem uzayındaki her programda bulunmaktadır. Örneğin üniversitelerin finans birimleri için geliştirilecek bir programda üniversite personeli, para, banka, harcama kalemi, saymanlık ve döner sermaye yetkilileri gibi kavramlar her uygulamada bulunmalıdır. Bahsedilen yapıların geliştirilen her uygulama için geliştiriciler tarafından tekrar tekrar öğrenilmesi, tasarlanması, geliştirilmesi ve bakımının yapılması kaynakların verimsiz kullanılması anlamına gelecektir. Kaynakların verimsiz kullanılması problemini çözmenin bir yolu ilgili programların ortak özelliklerini içinde barındıran ve uygulamalara temel bir çatı belirleyen yazılım çerçevelerinin (software frameworks) kullanılmasıdır (Morisio vd., 1999; Vladimir ve Sylvia, 2005).

Yazılım çerçevelerinin uygulama geliştirme ve bakımını yapma işlemlerinde verim artışını sağlaması için çerçevelerin kolay kullanılabilir, geliştirilebilir, bakım yapılabilir ve sağlam bir yapıya sahip olması gerekmektedir. Çerçevelerin pek çok uygulamaya temel oluşturacak olması, çerçeve tasarlama ve geliştirme işini zorlaştırmaktadır. Çünkü çerçeveler, çerçeve temelinde geliştirilen uygulama özelinde belli bir seviyeye kadar geliştirilebilmelidir. Bu durum da çerçeve tasarımının esnek bir yapıya sahip olması gerekmektedir (Srinivasan, 1999; Krajnc ve Hericko, 2003).

Çerçevelerin uzun ömürlü, yazılım geliştiriciler tarafından tercih edilir ve kullanılabilir olması, çerçeve için hayati bir öneme sahiptir. Bahsedilen mimari özelliklerdeki tasarımlara sahip çerçevelerin geliştirilmesinde geliştiricilerin tecrübeleri oldukça önemlidir. Ancak her geliştirme sürecinde istenilen tecrübeye sahip insanların bulunması da zor bir iştir. Tecrübeli tasarımcılar tarafından geliştirilmiş ve kabul görmüş yapıların kullanılması çerçeve geliştirme sürecindeki zorlukların aşılmasında etkili olacaktır. Bu sebeple çerçeve geliştirme sürecinde tasarım kalıplarının (design patterns) kullanımı bu problemin aşılmasında etkili olacaktır. Tasarım kalıpları belirli tasarım problemlerinin çözümü için esnek ve geliştirilebilir yapılar önerilileridir (Gamma vd., 1995).

Bu tez çalışmasında üniversitelerin finans birimleri için geliştirilecek uygulamalara temel oluşturacak ve bu uygulamalardaki temel yapıları içinde barındıran bir çerçeve geliştirilirken tasarım kalıplarının kullanımı incelenecektir. Tüm üniversitelerde benzer uygulamaların geliştirilmesine rağmen bu uygulamaların belirli bir çerçeve temelinde geliştirilmemesi önemli bir kaynak israfına yol açmaktadır. Üniversitelerdeki yapıların yönetmelikler ile sık sık değiştirilmesi, özellikle bakım maliyetlerini arttırmaktadır. Bahsedilen geliştirme ve bakım maliyetlerini en aza indirebilmek için istenilen kalite özelliklerine sahip bir çerçevenin geliştirilmesi uygun bir çözüm olacaktır.

Geliştirilen çerçeve üniversitelerin finans birimlerini için genel olan yapıları kendi içinde barındırırken, bu yapıların çerçeveyi temel alan uygulamalar özelinde genişletilebilmesine olanak sağlamaktadır. Bu özellikteki tasarımın yapılabilmesinde tasarım kalıplarının kullanımı önemli bir yere sahiptir.

Çalışmanın birinci bölümdeki girişten sonra, ikinci bölümünde yazılım çerçeveleri, üçüncü bölümünde ise çerçevelerin geliştirilme aşaması anlatılacaktır. Dördüncü bölümde tasarım kalıpları üzerine incelemeler bulunmaktadır. Beşinci bölümde çerçeveler ve tasarım kalıpları ile ilgili önceki çalışmalar incelenecektir. Altıncı bölümde tasarım kalıpları temelinde geliştirilen çerçeve anlatılacaktır. Yedinci bölümde tasarım kalıpları kullanımının geliştirilen çerçeve üzerindeki etkilerinin incelenmesinden sonraki sekizinci bölümde sonuçlar ve tartışmalar yer almaktadır.

2. YAZILIM ÇERÇEVELERİ

Bu bölümde benzer uygulama programlarına temel oluşturması için geliştirilen yazılım çerçeveleri ve çerçevelerin özellikleri incelenecektir.

2.1 Yazılım Çerçevesi Nedir?

Yazılım mühendisleri, farklı müşteriler için geliştirdikleri ürünlerin, pek çok benzerliğe sahip olduğu ancak bu ortak noktalarının ortaya konmadığı hissine kapılırlar (Morisio vd, 1999). Belirli bir problem uzayı (doğrulama ve yetkilendirme, bankacılık, insan kaynakları vb. uygulamaları) için geliştirilen uygulamalarda pek çok ortak nokta bulunmaktadır.

Aynı problem uzayındaki uygulamaların ortak/benzer olan bileşenlerinin ve bileşenler arası ilişkilerin her yeni uygulama için tekrar geliştirilmesi büyük bir zaman ve enerji kaybına yol açmaktadır. Uygulamaların geliştirildiği alandaki kavramların tekrar tekrar tanımlanması ve ortaya çıkan problemlere değişik çözümler getirilmesi yazılım geliştirme sürecinin farklı aşamalarında sorun yaratmaktadır.

Analiz aşamasında müşterinin isteğinin tam olarak anlaşılıp kağıda dökülmesi zor bir iştir. Yazılım çözümleyiciler çoğu zaman uygulamanın geliştirileceği alan ile ilgili teknik bilgiye sahip olmadığı için, bu durum müşterinin isteği ile çözümleyicinin anladığı yapının aynı olmaması dolayısıyla benzer kavramların farklı uygulamalarda farklı yorumlanması ile sonuçlanır. Analiz aşamasındaki bu çeşitliliğe, tasarım ve geliştirme aşamalarındaki farklılaşmalar da eklendiğinde aynı işi yapan, tasarımı ve kod parçaları birbirinden farklı uygulamalar karşımıza çıkar. Analiz, tasarım ve geliştirme aşamalarında değişik geliştirme stilleri ve standartları kullanımı projenin gecikmesine ve müşterinin memnuniyetsizliğine yol açmaktadır. Her projenin en temelden başlanarak geliştirilmesi zaman alan, öğrenme ve öğretme sürecini uzatan bir iştir (Vladimir ve Sylvia, 2005).

Uygulamaların bakım süreçleri ise ayrı bir sorun oluşturmaktadır. Belirli bir alan için geliştirilen uygulamaların tamamını etkileyecek (iş sürecindeki temel bir değişiklik ya da yeni eklenen bir kavram) bir güncelleme isteği durumunda tüm uygulamaların ayrı ayrı güncellenmesi ve test edilmesi gerekecektir. Bu durum da bakımı zor ve maliyetli bir iş haline getirecektir.

Yukarıda bahsedilen yazılım geliştirme ve bakım sürecini olumsuz etkileyen durumların etkilerini en aza indirmek için yazılım çerçeveleri kullanılabilir. Nesne tabanlı yazılım çerçeveleri modern yazılım mühendisliğinin köşe taşlarından biridir. Özellikle tasarımın ve kaynak kodun tekrar kullanımını desteklediklerı için yazılım çerçeveleri yazılım dünyasında

artan bir öneme sahiptir (Krajnc ve Hericko, 2003). Yazılım çerçeveleri özellikle, mimarinin ve işlevsel parçaların tekrar kullanılmasının gerekli olduğu, benzer uygulamaların geliştirilmesinde önemlidir. Nesne tabanlı bir çerçeve, soyut ve somut sınıfların bir karışımından oluşur. Çerçeve, problem uzayındaki tüm uygulamaların genel özelliklerini içinde barındıran yarı-tamamlanmış bir yapıdır. Çerçeve bileşenlerinin bazıları uygulamalara özel olarak değiştirilebilmektedir (Srinivasan, 1999). Diğer bir çerçeve tanımı da geliştiricisi tarafından, geliştirilen uygulamaya göre özelleştirilebilen yazılım sistemi iskeletidir (Krajnc ve Hericko, 2003).

2.2 Yazılım Çerçevelerinin Özellikleri

Çerçeve tanımlarında tekrar kullanım ve genişletilebilirlik kavramlarının altı çizilmektedir. Bunlara ek olarak modülerlik, bakımının kolay olması gibi özellikler bir çerçevenin sağlaması gereken özellikler olarak sıralanabilir.

2.2.1 Tekrar Kullanılabilirlik

Bir çerçeve temel alınarak geliştirilen uygulama, çerçevenin küçük değişimler göstermesi ve üst seviyedeki tasarım özelliklerinin uygulama tarafından kabul edilmesi özelliklerini taşır. Diğer bir söyleyişle çerçevenin tekrar kullanılması hem tasarım hem de kod parçaları düzeyinde gerçekleşir (Morisio vd., 1999). İki seviyede gerçekleşen tekrar kullanımın yararı sadece kod parçalarının tekrarının engellenmesi ile sınırlı değildir. Özellikle uygulama programlarına bakım yapılırken, bakım yapılması gereken yapının tek noktada toplanması maliyeti azaltacaktır. Tasarımın tekrar kullanımı ise uygulama geliştiriciler tecrübesiz bile olsa, geliştiricileri çerçevede belirlenmiş ve kabul görmüş tasarımlar çevresinde uygulama geliştirmeye zorlayacaktır.

2.2.2 Modülerlik

Temel tasarım prensiplerinden biri olan Açık-Kapalı Özelliği (Open-Closed Principle) bağlamında iyi bir tasarımda değişen ve yeni eklenen yapıların önceki yapıları etkilememesi\çok az etkilemesi gerekmektedir. Birçok yapının birleştirilmesi ile oluşturulan çerçevenin yapılar arasındaki bağımlılıkları en aza indirmesi ve bu yapıların birbirlerinden bağımsız şekilde kullanılabilmeleri ve güncellenebilmeleri oldukça önemlidir. Bu özellik Modülerlik olarak adlandırılır. Modüler bir çerçevenin elemanları arasında sıkı bağımlılıklar (strong coupling) olmamalıdır çünkü bu durum çerçevenin herhangi bir parçasında değişikliğe gidildiğinde zincirleme olarak birçok parçanın da değiştirilmesi gerekliliğini ortaya çıkarabilir. Çerçeve geliştirilirken belirsiz olan ve daha sonradan gelecek değişiklik istekleri

karşılanırken bu zayıf bağımlılık, çerçevenin devamlılığının sağlanmasında çok kritik bir öneme sahiptir.

2.2.3 Genişletilebilirlik

Bir çerçeve üzerine geliştirilmiş bir uygulamanın, uygulama özelinde değiştirilebilecek ve geliştirilebilecek yapılara sahip olabilmesi yazılım çerçevelerin genişletilebilir ve geliştirilebilir olması problemini ortaya çıkartmaktadır. Çünkü uygulama geliştiriciler özel bir uygulama için çerçeveyi, kalıtım özelliğini ya da çerçeve sınıflarından türemiş nesneleri birleştirerek kullanırlar (Krajnc ve Hericko, 2003). Dolayısıyla yazılım çerçevesinin olabildiğince esnek ve genişletilebilir bir şekilde tasarlanması bir zorunluluktur (Gamma vd., 1995). Çerçeve geliştirme işinin doğasında, çerçevenin genel yapıları, belirli bir soyutlama derecesinde içermesi yatmaktadır. Her yazılım ürününde olduğu gibi çerçevelerin soyutluk derecesinde de bir sınırlama olmalıdır. Çünkü soyutlama derecesi arttıkça çerçeveyi temel alan uygulamalar için daha fazla kod yazmak ve çerçevenin detaylarının çok daha fazla öğrenilmesi gerekecektir. Dolayısıyla genişletilebilirlik ve soyutlama arasında bir denge sağlanmalıdır.

2.2.4 Basitlik ve Kullanılabilirlik

Bir çerçevenin basit ve kullanılabilir olması çerçevenin sağlaması gereken en temel özelliklerdendir. Teknik açıdan başarılı sayılan bir çerçevenin kullanımı zor ise bu durum uygulama geliştiricilerin performansını olumsuz etkileyecektir. Bir yazılım mühendisi çerçeveyi etkin olarak kullanarak, uygulama geliştirebilmek için gereken zamanın önemli kısmını çerçeveyi anlayabilmek\öğrenebilmek için harcamaktadır. Çünkü yazılım geliştirici uygulamanın kullanım alanını ve çerçevenin tasarım özelliklerini anlamalı ve kodlamayı etkili bir biçimde yapabilmelidir. Bu durumda çerçevenin öğrenilmesi faktörü, çerçeve üzerine uygulama geliştirmedeki üretkenliğin artmasını sağlamaktadır (Morisio vd., 1999).

2.2.5 Bakım Yapılabilirlik

Çerçeve geliştirme aşamasından sonra çerçevenin devamlılığının sağlanması, geliştirilen yeni versiyonların problem çıkarmadan var olan uygulama programlarında güncellenebilmesi hayati önem taşımaktadır. Aslında bu özellik modülerlik, basitlik ve kullanılabilirlik gibi çerçeve özelliklerinin bir toplamı olarak görülebilir.

3. YAZILIM ÇERÇEVESİ GELİŞTİRME

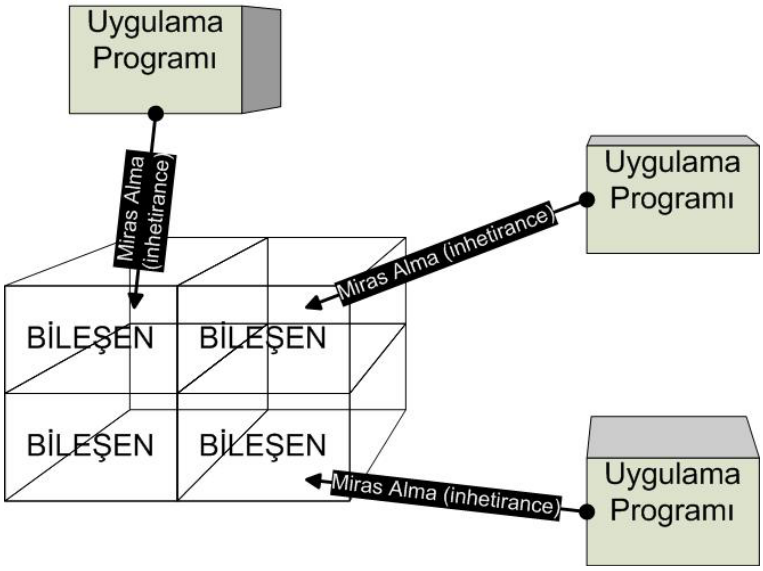
Yazılım çerçeveleri ve bir çerçeveyi oluşturan parçalar özelliklerine göre farklı sınıflarda incelenmektedir. Çerçeveler, teknik özelliklerine ve etki alanlarına göre, çerçeve bileşenleri ise uygulama programları ile olan ilişkilerine göre sınıflandırılır.

3.1 Çerçevelerin Teknik Özelliklerine Göre Sınıflandırılması

Bu bölümde kalıtım temeline dayanan Beyaz-Kutu (White-Box), birleştirme temeline dayanan Siyah-Kutu (Black-Box) ve bu iki türün avantajlı özelliklerinin birleştirilmesi mantığı üzerine geliştirilen Gri-Kutu (Gray-Box) Çerçeveler' in özellikleri incelenecektir.

3.1.1 Beyaz-Kutu Çerçeveler

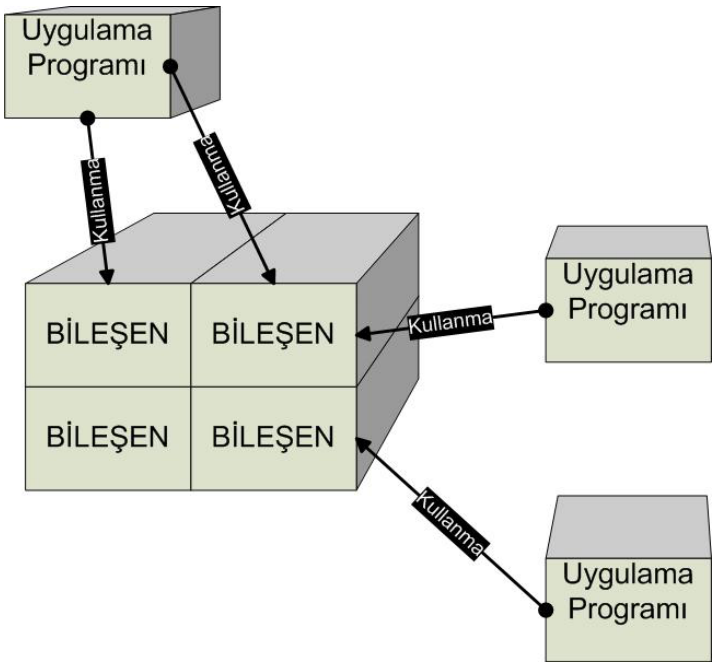
Şekil 3.1' de gösterildiği gibi Beyaz-kutu çerçeveler kalıtım temeline dayanmaktadır. Bu tip çerçevelerde uygulamalara özel yapılar, çerçevede tanımlanmış sınıflardan türeme ve sınıf metotlarının ezilmesi ile oluşturulur. Türeme ve metotların ezilmesi işleminin temelinde, türemiş sınıfların temel sınıflara ait bilgilere ulaşması gerekliliği yatar. Sınıf kalıtımı derleme zamanında ve statik olarak yapılır. Bu durum geliştirici açısından kolaylık sağlar. Ancak türemiş sınıf ile temel sınıf arasındaki sıkı bağımlılık tekrar kullanılabilirliği ve esnekliği olumsuz etkilemektedir. Çünkü temel sınıfta yapılan bir değişiklik, türemiş sınıfları doğrudan etkileyecektir. Bu yapıların kullanılabilmesi için sınıflar arası ilişkilerin ve sınıfların detaylarının bilinmesi gerekliliği beyaz-kutu çerçeve üzerine yazılım geliştirenler için bir dezavantajdır (Mattsson, 1999; Jung vd., 2000; Hayase vd., 2001).



Şekil 3.1 Beyaz kutu çerçeveler

3.1.2 Siyah-Kutu Çerçeveler

Siyah-kutu çerçeveler birleştirme (composition**)** temeline dayanmaktadır. Çerçeveyi temel alarak geliştirilen uygulamalar, yeni özellikteki yapıları nesneleri birleştirilerek kazanır. Nesne birleştirme işleminin dinamik olarak çalışma zamanında yapılabilmesi, siyah-kutu çerçevelerin en üstün yanıdır. Bunun yanında birleştirme işleminde (Şekil 3.2) nesnelerin içsel yapıları tamamen kapalıdır, birleştirilen nesneler sadece birbirlerinin arayüzlerinden (interface) haberdardır. Bu durum sınıflar arasındaki bağımlılığın gevşek olmasını sağlar. Sınıflar arası ilişkilerin arayüzler üzerinden tanımlanmış olması, herhangi bir sınıftaki içsel değişikliğin bu sınıfla uyumlu çalışan diğer sınıfların etkilenmesini engeller. Ancak gevşek bağımlılık ile kazanılan avantaj, nesneleri birleştirme işinin uygulama geliştiricilerin sorumluluğuna bırakılmasını beraberinde getirir (Mattsson, 1999; Jung vd., 2000; Hayase vd., 2001).



Şekil 3.2 Siyah kutu çerçeveler

3.1.3 Siyah-Kutu ve Beyaz-Kutu Çerçevelerin Karşılaştırılması

Çerçeveler yarı-tamamlanmış uygulamalardır. Bu bölümde aynı tip yapıları sıralamak için Kabarcık Sıralama Yöntemini kullanan bir yapının Beyaz-Kutu ve Siyah-Kutu çerçevelerde nasıl gerçeklenebileceği ve bu yapıların birbirlerinden farklılaşan özelliklerinden bahsedilecektir.

Kabarcık Sıralama Yönteminde her dizi elemanı kendisinden bir sonraki dizi elemanı ile karşılaştırılır ve eğer daha büyükse bu iki dizi elemanı yer değiştirir. Tüm elemanlar dizide

uygun yerlere geçene kadar yöntem çalışmaya devam eder. Bu yöntemde sıralama işlemi için büyüklük karşılaştırmasının ve yer değiştirme işlemi dışındaki işlemler tüm diziler için geçerli olacaktır. Ancak bu iki işlem diziyi oluşturan elemanların tipine göre farklılıklar gösterecektir. Bu durumda bahsedilen iki işlem dışındaki kod parçaları çerçeveye dahil edilerek uygulama programları açısından tekrar kullanım sağlanabilir.

3.1.3.1 Miras Alma Yaklaşımı

Miras alma temeline dayalı Beyaz-Kutu Çerçeveler için aşağıdaki gibi bir yapı tasarlanabilir.

```
// Kabarcık Sıralama uygulaması
public abstract class DiziSiralayici
{
  protected int _diziUzunlugu; // Sıralanacak dizinin uzunluğu
   protected void Sirala()
   {
      if (_diziUzunlugu == 1)
         return ;
      for (int index1 = _diziUzunlugu - 2; index1 >= 0; index1--)
         for (int index2 = 0; index2 <= index1; index1++)
            if (!SiraliMi(index1))
               Degistir(index1);
   }
   protected abstract bool SiraliMi(int index); // İlgili dizi elemanı uygun yerde mi?
   protected abstract void Degistir(int index); // Elemanın yerini değiştirir.
}
```

Bu yapıda SiraliMi() ve Degistir() metodlarının DiziSiralayici sınıfından türeyen sınıflar tarafından uygulanması, böylelikle temel sınıftaki Sirala() metodunun istenilen işlemi yapması beklenmektedir.

```
// Tamsayı (int)  için Kabarcık Sıralama uygulaması
public class IntSiralayici : DiziSiralayici
{
   private int[] intDizisi = null;

   public void DiziyiSirala(int[] gelendizi)
   {
      intDizisi = gelendizi;
```

```
        _diziUzunlugu = gelendizi.Length;
        Sirala();
        return;
    }
    protected override bool SiraliMi(int index)
    {
        return (intDizisi[index] > intDizisi[index + 1]);
    }

    protected override void Degistir(int index)
    {
        int gecici = intDizisi[index];
        intDizisi[index] = intDizisi[index + 1];
        intDizisi[index + 1] = gecici;
    }
}
```

IntSiralayici sınıfı için SiraliMi() ve Degistir() metodlarının uygulanması ile istenilen sıralama işlemi yapılacaktır. Ancak IntSiralayici sınıfı ile DiziSiralayici sınıfı arasında türeme işlemi sebebiyle sıkı bir bağımlılık kurulmuştur. Başka bir deyişle DiziSiralayici sınıfındaki bir değişiklik, IntSiralayici sınıfını doğrudan etkileyecektir. Bunun yanında IntSiralayici sınıfının düzgün çalışması için, temel sınıf olan DiziSiralayici sınıfının _diziUzunlugu elemana ulaşması gerekmektedir. Bu durum hem bilgiyi saklama (encapsulation**)** ilkesine aykırıdır hem de çerçeve detaylarının uygulama geliştirici tarafından öğrenilmesi gerekliliğini ortaya çıkarır.

3.1.3.2 Birleştirme Yaklaşımı

Birleştirme ilkesine dayalı Siyah-Kutu Çerçeveler için aşağıdaki gibi bir yapı tasarlanabilir.

```
public interface ISiralamaIsleri
  {
     bool SiraliMi(int index);
     void YerDegistir(int index);
     int DiziUzunluguGetir();
     void DiziyiBelirle(object dizi);
  }
  public class DiziSiralayici
```

```
{
    private int _diziUzunlugu;
    private ISiralamaIsleri _siralayici;

    public DiziSiralayici(ISiralamaIsleri gelen)
    {
        _siralayici = gelen;
    }

    public void Sirala(object siralanacekDizi)
    {
        _siralayici.DiziyiBelirle(siralanacekDizi);
        _diziUzunlugu = _siralayici.DiziUzunluguGetir();
        if (_diziUzunlugu == 1) // Dizi uzunluğu 1 ise işlem yapılmaz.
            return;
        for (int index1 = _diziUzunlugu - 2; index1 >= 0; index1--)
            for (int index2 = 0; index2 <= index1; index1++)
                if (!_siralayici.SiraliMi(index1))
                    _siralayici.YerDegistir(index1);
    }
}
```

DiziSiralayici sınıfı yine Kabarcık Sıralama Yönteminin ana parçalarını sunmaktadır, ancak bu sınıftan türemiş nesnelerin çalışabilmeleri için ISiralama arayüzünü uygulamış bir nesneyi kullanmaları gerekir.

```
public class IntSiralayici : ISiralamaIsleri
{
    private int[] dizi = null;

    #region ISiralamaIsleri Members
    public bool SiraliMi(int index)
    {
        return dizi[index] > dizi[index + 1];
    }
    public void YerDegistir(int index)
    {
```

```
        int gecici = dizi[index];
        dizi[index] = dizi[index + 1];
        dizi[index + 1] = gecici;
    }
    public int DiziUzunluguGetir()
    {
        return dizi.Length;
    }
    public void DiziyiBelirle(object gelenDizi)
    {
        dizi = (int[])gelenDizi;
    }
    #endregion
}
```

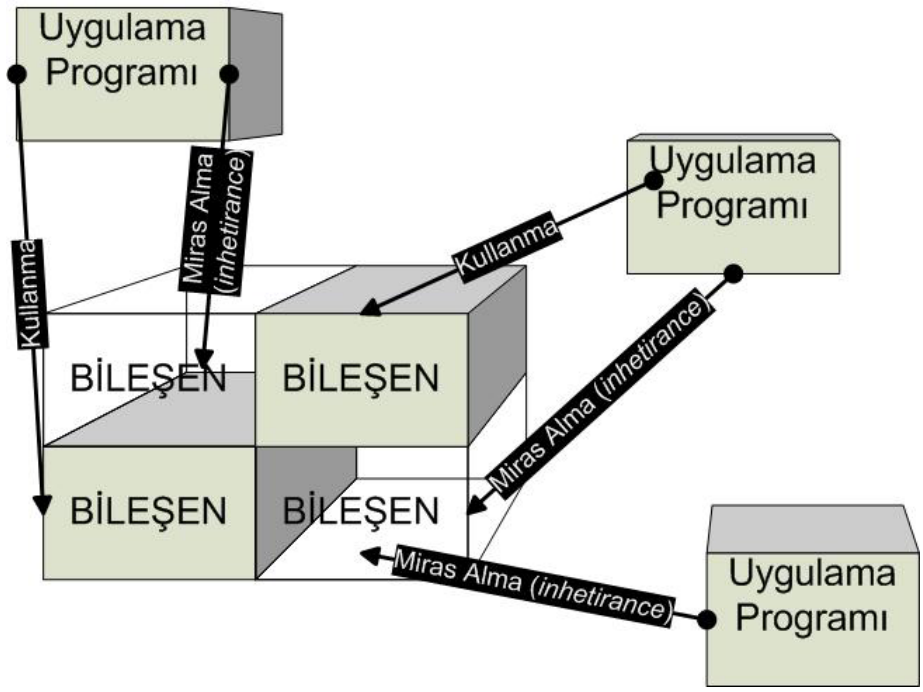
Yukarıda görülen IntSiralayici sınıfı, ISiralamaIslemleri arayüzünü uygulamaktadır. Bu sınıfın amacı dizi elemanlarının büyüklük kontrollerini ve yer değiştirme işlemlerini yapmaktır. Siyah-Kutu yaklaşımı ile DiziSiralayici sınıfı herhangi bir tipteki nesneleri içeren listeleri sıralayabilecektir. Birleştirme yaklaşımı ile DiziSiralayici ve IntSiralayici sınıfları arasındaki bağımlılık oldukça azalır. Çünkü DiziSiralayici sınıfı sadece ISiralamaIslemleri arayüzünü sağlayan bir nesne ile ilişki içindedir. Dolayısıyla DiziSiralayici sınıfı içindeki bir değişiklik IntSiralayici sınıfını etkilememektedir.

3.1.4 Gri-Kutu Çerçeveler

Beyaz-Kutu ve Siyah-Kutu çerçevelerin temelini oluşturan, sırasıyla miras alma ve birleştirme yaklaşımlarının doğası gereği birbirlerine göre avantajları ve dezavantajları vardır. Miras alma yöntemi ile belli fonksiyonlar ve yapılar temel sınıflarda tanımlanır. Ancak türemiş sınıflar ile temel sınıflar arasında oluşan sıkı bağımlılık bakım maliyetlerini arttıracaktır. Siyah-Kutu çerçevelerde ise sıkı bağımlılık problemi daha azdır. Bu tip çerçevelerde de sınıflar arası bağlantıları ve koordinasyonu sağlamak için ara katmanların yazılması gerekmektedir. Bu durum esneklik ve bakım yapılabilirlik özelliklerini desteklese de sınıf sayısının artması ile sonuçlanacağından çerçevenin karmaşıklığı artmaktadır (Chen, 2004).

Bu özellikler ışığında Gri-Kutu çerçeveler ortaya çıkmıştır. Gri-Kutu çerçevelerin temel amacı Beyaz-Kutu ve Siyah-Kutu çerçevelerin dezavantajlarını ortadan kaldıracak bir mimari oluşturabilmektir (Şekil 3.3). Kaliteli bir Gri-Kutu çerçeve yeterince esnek ve genişletilebilirken aynı zamanda da, miras alma mantığı ile gereksiz karmaşıklıkları çerçeve

kullanıcısından saklayabilmelidir (Krajnc ve Hericko, 2003).



Şekil 3.3 Gri kutu çerçeveler

Bu çalışmada geliştirilen Finans Çerçevesi Gri-Kutu çerçeve mimari özellikleri temel alınarak geliştirilmiştir.

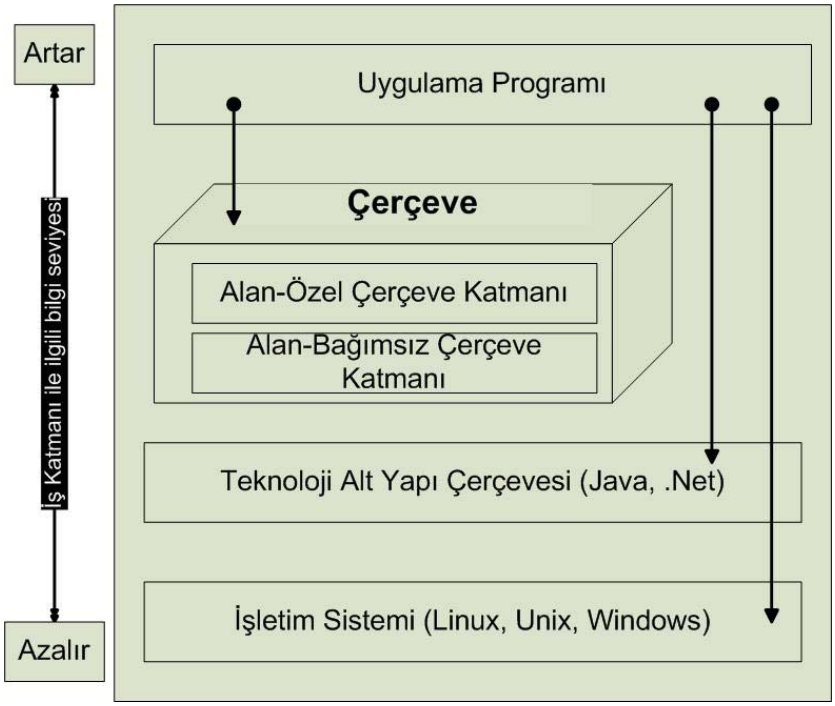
3.2 Çerçevelerin Etki Alanlarına Göre Sınıflandırılmaları

Çerçeveler etki alanlarına göre Uygulama, Alan-Özel ve Yardımcı Çerçeveler olarak üç gruba ayrılmıştır. Uygulama Çerçeveleri uygulama programlarına çok geniş ölçekte temel sağlayacak şekilde geliştirilmişlerdir. Bu çerçeveler tüm programlarda bulunabilecek grafik kullanıcı arayüzü, belge, veri tabanı işlemleri gibi çok temel yapıları sağlamaktadır. MFC (Microsoft Foundation Classes) ve JFC (Java Foundation Classes) bu tip çerçevelerdir. Alan-Özel Çerçeveler ise belirli bir problem uzayındaki, örneğin alarm sistemleri, bankacılık uygulamaları gibi, uygulama programlarına iskelet oluşturacak şekilde geliştirilmişlerdir. Yardımcı Çerçeveler ise bilgisayar dünyası ile ilgili çok özel alanlara destek vermek için tasarlanırlar. Bu tip çerçeveler bellek yönetimi, dosya sistemi yönetimi gibi oldukça özelleşmiş işlerle ilgilenir (Gurp ve Bosch, 2001; Prieto vd., 2003).

3.3 Çerçeve Katmanları

Çerçeveler de kendi içindeki katmanlı yapılar için bu gruplamaları kullanabilir. Hastane uygulamalarına temel oluşturacak bir çerçeveyi düşünürsek, çerçevenin alan-özel (domain-specific) katmanında, tüm hastane uygulamalarında bulunacak olan hasta, doktor, teşhis, ilaç vb. gibi yapılar ve aralarındaki ilişkiler gerçeklenecektir. Yapının alan bağımsız (cross-domain) katmanında ise hastane programlarının yanında diğer uygulamalarda da kullanılabilecek yetkilendirme ve doğrulama, e-posta gönderimi, dosya sistemi işlemleri gerçeklenecektir. Katmanlı yapının son parçası ise tamamen çerçevenin geliştirildiği ortam ile ilgili olacaktır. Bu katmanda (foundation framework) kullanılan programlama dilinin sağladığı kütüphaneler ve işletim sistemi ile ilgili yapılar bulunacaktır. Son katman, çerçeve geliştiricilerin hazır olarak kullanacağı ve değiştirilmesine izin verilmeyen yapılardan oluşacaktır.

Bu çalışmada geliştirilen çerçeve, Finans uygulamaları için özel yapıları içeren Alan Özel Katman, bu tip uygulamaların tamamında bulunabilecek yapıları içeren Alan Bağımsız Katman yapılarına sahiptir. Çerçevenin son katmanı ise C# ile uygulama geliştirmeyi sağlayan .NET ve MS Windows yapılarını içeren ve değiştirilmeden kullanılan yapıları içermektedir.



Şekil 3.4 Çerçeve katmanları

4. TASARIM KALIPLARI

Tasarım, elde edilmek istenen ürünün soyut halidir. Bu soyutlamada ürünü oluşturan parçalar ve bu parçalar arasındaki ilişkiler tanımlanmıştır. Belli bir problem uzayındaki soyutlamaları oluşturan parçalar kavramsal olarak aynıdır. Örneğin, evlerin mimari tasarımları düşünüldüğünde, fiziksel olarak hangi malzemeden ve nasıl yapıldığına bakılmaksızın, her ev için kapı, oda, pencere, mutfak, duvar vb. gibi bileşenler bulunmaktadır. Bunun yanında bir odanın, duvarları ve pencereleri içermesi; evin odaları içermesi gibi bileşenler arası ilişkiler de tasarımda yer alır. Aynı şekilde nesneye dayalı tasarımlar da aynı soyutlama özelliklerini gösterir. Nesneye dayalı tasarım çözümlerinde duvarlar ve kapılar yerine nesneler ve arayüzler vardır ancak her iki kalıp çeşidi de temelde, bir problem uzayındaki çözümlerdir. Her kalıp, kendi ortamında tekrar tekrar ortaya çıkan problemleri anlatır, sonra, aynı yöntemi tekrar yapmadan, milyonlarca defa kullanılabilecek bir şekilde, bu problemin temel çözümünü ortaya koyar (Mattsson, 1999; Chen, 2004).

Her nesneye dayalı yazılım ürününde tekrarlayan problemler vardır. Nesneleri kimin yaratacağı, çalışma zamanında nesnelerin özelliklerinin değişmesi gerekliliği gibi problemler, geliştirilen programlama dili ve ortamdan bağımsız olarak karşımıza çıkar. Bu problemlere tasarım kalıpları aracılığı ile genel çözümler bulunabilmektedir. Nesneye dayalı tasarımlar için kullanılan tasarım kalıpları özel bir bağlamda, genel tasarım problemlerinin çözümlerini ortaya koymamızı sağlayan tanımlamalardır. Bu tanımlamalar ile tasarım kalıpları tekrar kullanılabilen nesne tabanlı tasarımları ortaya çıkarır çünkü tasarım kalıpları genel tasarım yapısını isimlendirir, soyutlaştırır ve tanımlar (Srinivasan, 1999).

4.1 Tasarım Kalıplarının Sınıflandırılması

Sık sık kullanılan birçok tasarım kalıbının belgelendiği klasik kitap Tasarım Kalıpları: Tekrar Kullanılabilir Nesne Tabanlı Yazılım Elemanları kitabında (Gamma vd., 1995) tasarım kalıpları, kalıpların amaçlarına ve uygulama alanlarına göre

- Yaratımsal Kalıplar
- Yapısal Kalıplar
- Davranışsal Kalıplar

olmak üzere üç ana başlık altında toplanmıştır.

Yaratımsal Tasarım Kalıpları, nesnelerin yaratılma işlemlerini soyutlamak için kullanılmaktadır. Bu kümedeki kalıplar bir sistemin, nesnelerin yaratılma süreçlerinden

bağımsız olması amacıyla kullanılmaktadır.

Yapısal Tasarım Kalıpları, sınıfların ve nesnelerin daha büyük yapılar oluşturabilmek için nasıl bir araya getirileceği ile ilgili problemlere çözümler sunmaktadır.

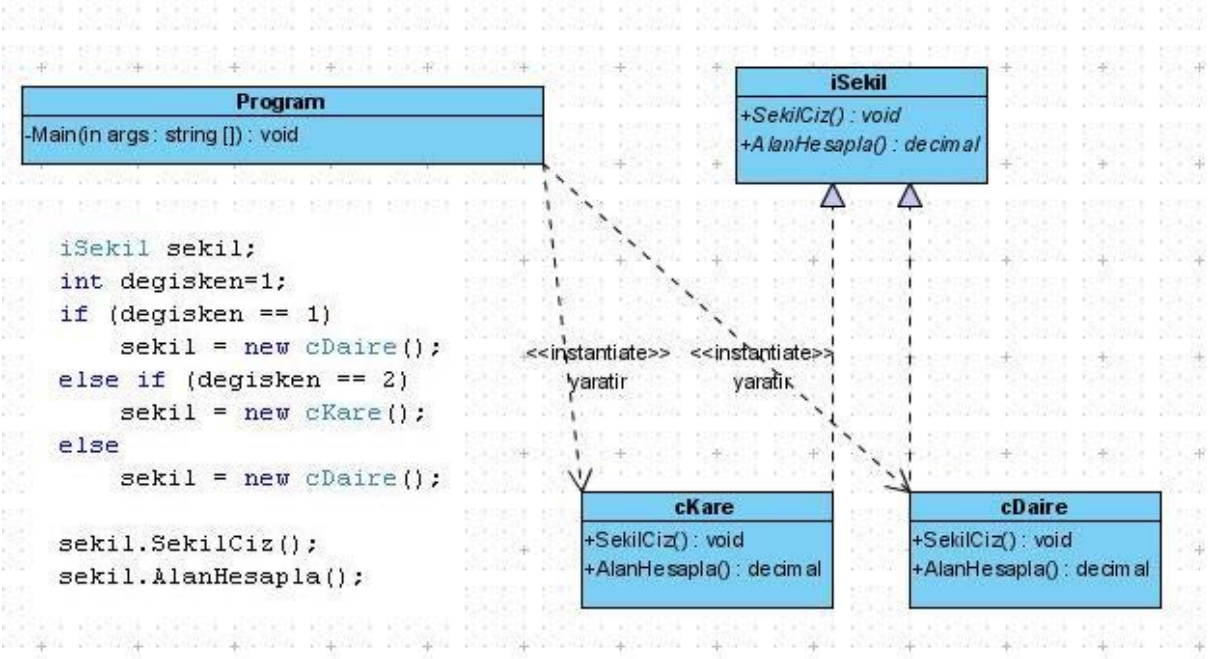
Davranışsal Tasarım Kalıpları da algoritmalar ve nesneler arasındaki sorumluluk atamaları ile ilgili konularla ilgilenmektedir (Gamma vd., 1995; Shalloway ve Trott, 2001). Bunun yanında farklı kaynaklarda, yine GoF (Gamma vd., 1995) kitabındaki sınıflandırmalar temel alınarak, daha özel başlıklar altında da sınıflandırmalar yapılmıştır. Shallow ve Trott kitabında tasarım kalıpları sınıflandırmasına Bağımlılık-Azaltan Kalıplar sınıfı da eklenmiştir. Metsker ise tasarım kalıplarını Arayüz, Sorumluluk, Yapısal, İşlemsel ve Eklenti başlıklarını verdiği sınıflar altında incelemiştir.

4.1.1 Yaratımsal Kalıplar

Yaratımsal kalıplar, nesne yaratma işini ve bu işlemdeki karmaşıklığı istemciden soyutlamak için kullanılmaktadır. Ancak bu tanım sadece kullanım kolaylığını çağrıştırmaktadır. Oysa yaratımsal kalıplar, istemcinin uygulama kodunu yazarken somut nesneler kullanmak yerine arayüzler ve soyut sınıflar kullanarak daha esnek yapılarla çalışabilmesini de sağlamaktadır. Bu sınıftaki kalıplar beş başlıkta toplanmaktadır.

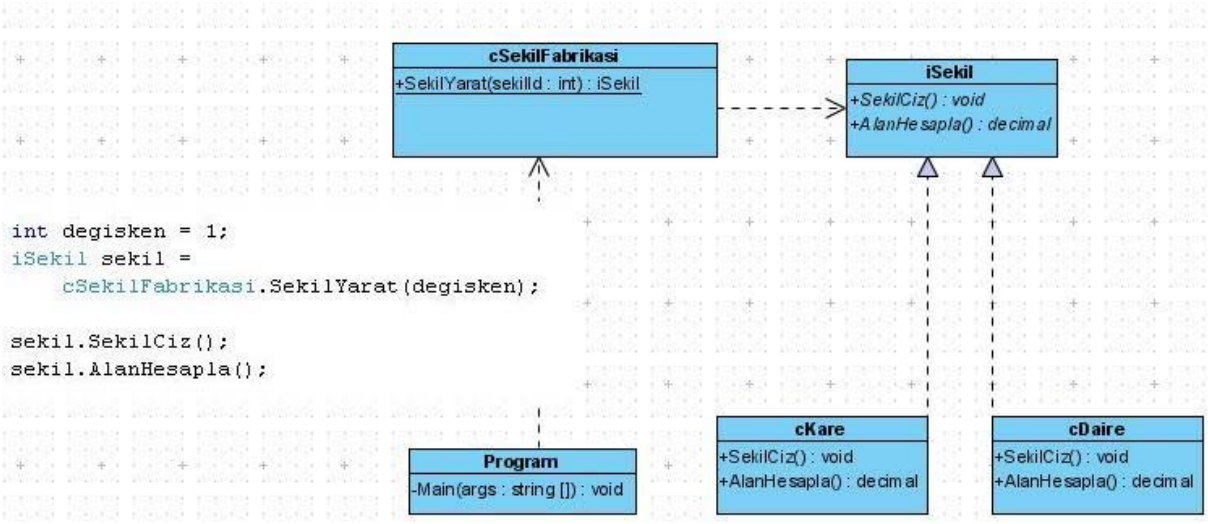
- Soyut Fabrika Kalıbı: Birbirine bağımlı ya da ilişkili nesne gruplarının yaratım işini istemciden soyutlamak için kullanılır.
- Yapıcı Kalıbı: Karmaşık bir yaratım sürecine sahip nesne gruplarının tek bir arayüz üzerinden yaratılmasını sağlar. Böylelikle aynı yaratım süreci ile farklı nesneler oluşturulurken diğer sistemler yaratım sürecinin farklılaşmasından etkilenmez.
- Fabrika Yöntemi: Nesne yaratım sorumluluğunu kalıtım yolu ile türeyecek sınıfa bırakmayı önerir. Böylelikle her sınıf kendi yaratım süreci ile ilgilenecektir.
- Örnek Kalıbı: Yaratım süreci masraflı olan nesnenin en baştan yaratılması yerine önceden yaratılmış örnek bir nesnenin parçalarının kullanılması ile kaynakların daha verimli kullanılmasını sağlar.
- Yegane Kalıbı: Uygulama boyunca bir sınıfın tek bir nesnesinin yaratılmasını garanti altına almak için kullanılır.

Yaratımsal Kalıpların daha iyi anlaşılabilmesi için aşağıda Soyut Fabrika Kalıbı'nın kullanımı ve uygulama koduna etkileri incelenmiştir.



Şekil 4.1 Geometrik şekiller tasarımının UML gösterimi

Görüldüğü gibi uygulama programı bir değişkenin değerine bakarak yaratılacak nesnenin tipine karar vermektedir. Burada nesneleri yaratma sorumluluğu programcıya verilmiştir ve bu durum cDaire ve cKare sınıflarının uygulama programı ile sıkı-bağımlı olmasına yol açmıştır. Bunun yanında ileride yeni bir geometrik şekil sisteme eklenirse görülen kod parçası güncellenme, test edilme vb. gibi aşamalardan geçmelidir. Oysa bu yapıdaki geometrik şekil nesnelerini yaratma işi kullanıcıdan alarak bu problemleri ortadan kaldırabiliriz. Şekil 4.1'deki tasarımın Soyut Fabrika kalıbı kullanılarak tekrar tasarlanmış hali Şekil 4.2' de gösterilmiştir.



Şekil 4.2 Geometrik şekillerin soyut fabrika kalıbı ile tasarımı

Görüldüğü gibi uygulama programı sadece cSekilFabrikasi sınıfı ile ilişkilidir ve geometrik şekillerin yaratılması işleri ile ilgili detaylardan soyutlanmış durumdadır. Dolayısıyla uygulama programı yeni bir şeklin sisteme etkilenmesinden ya da nesne yaratılma sürecinde meydana gelebilecek bir değişiklikten etkilenmeyecektir. Bu durum da bakım maliyetlerinin azalmasını sağlayacaktır.

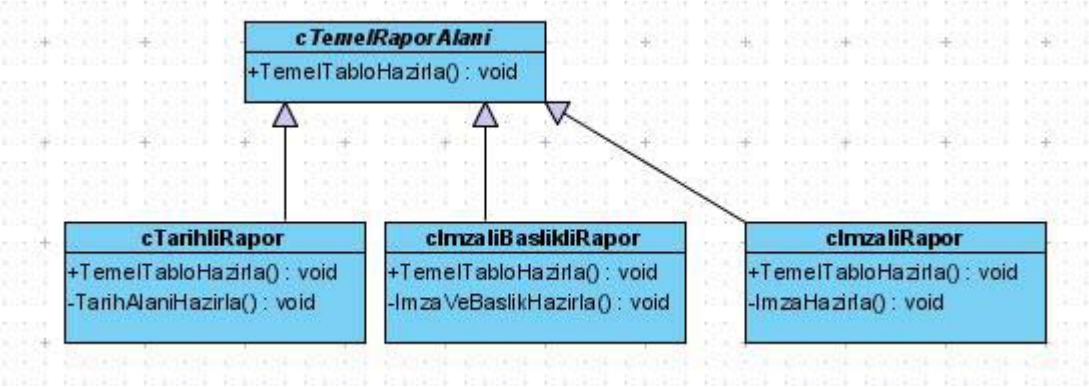
4.1.2 Yapısal Kalıplar

Sınıfların ve nesnelerin daha büyük yapılar oluşturabilmek için nasıl bir araya getirileceği ile ilgili problemlerin çözümleri Yapısal Kalıplar başlığı altında incelenmektedir. Nesne veya sınıfların birleştirilmesi ile belirli bir işi yapan sınıfın işlevi çalışma zamanında değiştirilebilir, işleve yeni özellikler kazandırılabilir.

Yapısal Kalıplar yedi madde altında toplanmıştır.

- Adaptör Kalıbı: İstenilen işi yapabilecek olmasına rağmen arayüz uyumsuzluğu nedeniyle kullanılamayan bir sınıfın, sisteme uyumlu arayüz altında kullanılabilmesini sağlar.
- Köprü Kalıbı: Hem arayüzün hem de sınıfların metotlarının uygulanışlarının birbirlerinden ayrılmasını sağlamak amacıyla kullanılır. Böylelikle arayüz ve sınıfın uygulanışı birbirinden bağımsız olarak değiştirilebilir.
- Bileşik Kalıbı: Nesnelere ve nesne gruplarına aynı arayüz üzerinden ulaşmamızı sağlar.
- Dekoratör Kalıbı: Bir nesneye, nesneyi değiştirmeden, yeni sorumluluklar eklemek için kullanılır.
- Önyüz Kalıbı: Karmaşık bir iç yapıya sahip sistemin basit bir arayüz üzerinden sunulmasını sağlar.
- Sineksiklet Kalıbı: Benzer nesnelerin pek çok defa yaratılması yerine bir örnek nesnenin pek çok yerde paylaşılarak kullanılmasını sağlamak için kullanılır.
- Vekil Kalıbı: Yaratımı masraflı özelliklerinin her zaman kullanılmadığı nesnelerin yerine aynı arayüzü sağlayan taklit nesnelerin kullanılmasını sağlar. Taklit nesneler kendilerine gelen istekleri gerekli olduğu zaman gerçek nesnelere iletir.

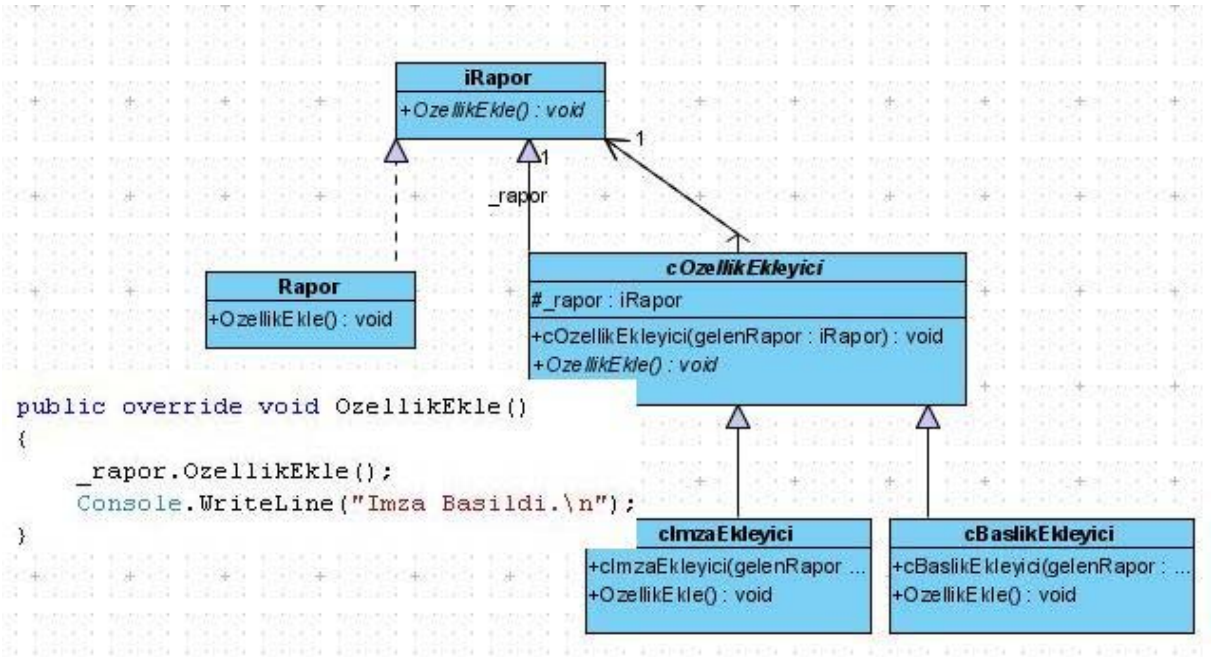
Yapısal Kalıpların çalışma mantığını anlatabilmek için bir raporlama problemi incelenecektir. Hazırlanacak raporların içeriği başlık, tarih, imza ve bilgilerin gösterildiği temel tablo yapılarından oluşabilmektedir. Tüm raporların ortak alanı temel tablodur. Ancak bazı raporlarda temel tabloya ek olarak, tarih ve imza; bazılarında ise başlık ve tarih bulunmaktadır. Yeni raporların içerikleri farklı alanlardan oluşabilir. Bu problem için aşağıdaki gibi kalıtıma dayalı bir çözüm tasarlanabilir.



Şekil 4.3 Raporlama yapısı tasarımı

Şekil 4.3' te kalıtım temelli tasarım ile kullanılan cTemelRaporalani isimli soyut sınıfın TemelTabloHazirla() metodu ile tekrar kullanım sağlanmıştır. Ancak bu yapının en problemli kısmı, yeni bir rapor çeşidi eklenmek istenirse yeni bir sınıfın eklenmesi gerekliliğidir. Örneğin rapora imza alanını eklemek için kullanılacak yapı cImzaliBaslikliRapor sınıfında tanımlanmıştır. Ancak sınıflar ve içerikleri arasındaki sıkı bağlantı sebebiyle cImzaliRapor gibi yeni bir sınıf yaratılmıştır.

Bu yaklaşımdaki temel sorun rapor içerikleri hazırlanırken kullanılan işlemlerin (imza, başlık, tarih) sınıflar ile çok sıkı-bağımlı olmasıdır. Dekoratör Kalıbı, gevşek-bağımlı bir tasarım sunar. Dekoratör Kalıbı'nın kullanımı ile rapor sınıfları, rapor özelliklerini kendi metotlarında uygulamayacaklardır. Bunun yerine raporların istenilen özellikleri sahip olması için bu özellikleri ortaya çıkaracak nesneleri kullanacaklardır.



Şekil 4.4 Raporlama yapısının dekoratör kalıbı ile tasarımı

Dekorator Kalıbı' nın kullanıldığı bu tasarımda (Şekil 4.4) Raporun temel tablosu ve eklenmek istenen diğer özellikler birbirinden ayrılmıştır. Böylelikle rapor çeşidine göre eklenecek özellikler çalışma zamanında da değiştirilebilecektir. Böyle bir yapıda yaratılacak rapor nesnesi aşağıdaki gibi olacaktır.

iRapor yeniRapor = new cBaslikEkleyici(new cImzaEkleyici(new Rapor()));
yeniRapor.OzellikEkle();

Farklı özelliklere sahip raporlar sadece istenilen özelliklerin, görülen nesne yaratma zincirine eklenmesi ile oluşturulabilecektir.

4.1.3 Davranışsal Kalıplar

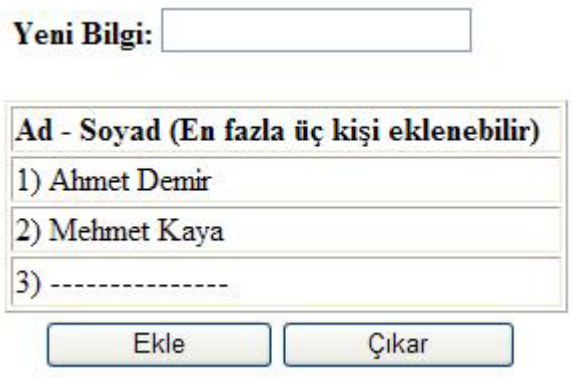
Bu kümedeki kalıplar nesneler arasındaki iletişim problemlerini çözmek için kullanılır. Nesneler arasındaki en temel iletişim bir nesnenin başka bir nesnenin metodunu çağırmasıdır. Bu temel yöntem nesneler arasında sıkı-bağımlı bir yapının oluşmasına neden olur. Bu bağımlılık sınıf sayısı arttığında bakım maliyetlerini de oldukça arttıracaktır.

Bu sınıfta incelenen tasarım kalıpları on bir başlık altında toplanmıştır.

- Sorumluluklar Zinciri Kalıbı: Bir isteği, duruma göre değişen farklı yapılar cevaplıyorken Sorumluluklar Zinciri Kalıbı, bu yapılara tek bir noktadan ulaşmamızı sağlar. Zincirdeki bir yapı gelen isteği karşılayamazsa, isteği bir sonraki yapıya iletir.

- Komut Kalıbı: Sistemlerin isteklerini nesne yapısı içerisinde işlemesine olanak sağlar. Böylelikle istekler kayıt altına alınabilir ve işlemlerin geriye dönüşü sağlanabilir.
- Yorumlayıcı Kalıbı: Karmaşık uygulamaların kurallarını sınıf yapısında tanımlayarak, kuralların genişletilebilmesine olanak sağlar.
- Yineleyici Kalıbı: Hiyerarşik bir yapıda, birçok nesneden oluşmuş bir kümenin elemanlarına sıra ile ulaşabilmemizi sağlar.
- Arabulucu Kalıbı: Birbirleri ile bağlantılı çalışan nesneler arası koordinasyonun tek bir arabulucu yapı üzerinde toplanmasını sağlar. Böylelikle beraber çalışan nesneler birbirlerinden haberdar olmak zorunda kalmaz.
- Yadigar Kalıbı: Nesnelerin önceki durumlarını saklamak ve gereğinde geri dönebilmek için kullanılmaktadır.
- Gözlemci Kalıbı: Bir nesne grubunun, bir veri kaynağındaki değişimden otomatik olarak haberdar olmasını sağlamak için kullanılır.
- Durum Kalıbı: Bir nesnenin davranışının belirli bir durumlara göre değiştirilebilmesine olanak sağlar. Son durumlu makinelerin modellenmesinde sıkça kullanılır.
- Strateji Kalıbı: Aynı problemi çözmek için kullanılabilecek farklı algoritmaların, aynı arayüz üzerinden sunularak, sistem tarafından fark edilmeden değiştirilebilmesini sağlar.
- Kalıp Yordamı: Bir metotta tanımlanmış ve alt adımlardan oluşan bir algoritmanın adımlarının, türemiş sınıflar tarafından değiştirilebilmesine olanak sağlar.
- Ziyaretçi Kalıbı: Birçok nesneden oluşan bir yapının, ziyaretçi nesne tarafından tek tek ziyaret edilerek bilgi toplanmasına ve sisteme sunulmasını sağlar.

Sınıflar arası bağımlılığı azaltmak için kullanılan davranışsal kalıplardan biri Arabulucu (*Mediator*) Kalıbıdır. Bu kalıbın çalışma mantığını anlatmak için farklı durumlara göre aktif ya da pasif durumlara geçen bazı elemanlardan oluşan bir kullanıcı arayüzü incelenecektir.



Şekil 4.5 Örnek Kullanıcı Arayüzü

Şekil 4.5’ te gösterilen arayüz için basit kurallar tanımlanmıştır. Bu kurallar,

- Listeye en fazla üç kişi eklenebilir
- Listedeki eleman sayısı üçten az ise Yeni Bilgi, Ekle ve Çıkar kontrolleri aktif durumdadır
- Listedeki eleman sayısı üç ise Yeni Bilgi ve Ekle kontroller pasif, Çıkar kontrolü ise aktif durumdadır

şeklinde tanımlanmıştır. Bu kuralları sağlamak için bir çözüm; örneğin Ekle düğmesi seçilerek listeye eleman eklendiğinde, toplam eleman sayısına göre Ekle düğmesine basıldığında çalışan kod parçasının Yeni Bilgi ve Çıkar düğmelerini pasif hale getirmesidir.

```
public void ElemanEkle(int elemanSayisi)
{
if(elemanSayisi >= 3)
Ekle.Enabled = false; YeniBilgi.Enabled = false; Cikar.Enabled = true;
}
```

Bu tasarımda görüldüğü gibi Ekle düğmesinin YeniBilgi ve Cikar kontrollerinin olduğunu bilmesi gerekmektedir. Bu durumda da kontroller arasında sıkı-bağımlı bir yapı oluşmuştur. Bu çözüme farklı bir seçenek olarak Arabulucu Kalıbı kullanılabilir.

```
public void ElemanEkle()
{
arabulucu.ElemanEkle();
}
.....
public void ElemanEkle()
{
if(ElemanSayisiGetir() >= 3)
Ekle.Enabled = false; YeniBilgi.Enabled=false; Cikar.Enabled = true;
}
```

Yeni tasarımda görüldüğü gibi Ekle düğmesi sadece Arabulucu sınıfının varlığını bilmektedir. Diğer kontroller ile ilgili bilgilerin tamamı hakkında bilgi sahibi olan tek yapı Arabulucu sınıfıdır. Böylelikle kontrollerde herhangi bir değişiklik olduğunda güncellenmesi gereken tek sınıf, Arabulucu sınıfı olacaktır. Sonuçta gevşek-bağımlı ve esnek bir yapı karşımıza çıkmaktadır.

4.2 Tasarım Kalıpları Kullanmanın Etkileri

Büyük sistemler için gerçeklenen uygulamalarda tekrar kullanılabilirlik, bakım yapılabilirlik ve yeni isteklerin sisteme dahil edilebilmesi, geliştirilen uygulamanın ömrü açısından oldukça

önemlidir. Tekrar kullanılabilirlik uygulamanın geliştirme aşamasında sağlanabilecek bir özelliktir. Ancak uygulamanın bakım yapılabilirlik ve eklenti yapılabilirliği yetenekleri geliştirme aşamasından sonraki zamanlarda kendini gösterecektir. Bu özelliklerin fiziksel olarak gerçeklenebilmesi için uygulamayı gerçekleyenler ile bakım yapacak insanlar arasındaki iletişim oldukça önemlidir. Tasarım kalıpları bu iletişimi sağlar. Tasarım Kalıpları geniş ölçekli sistemler geliştirilirken, temel zorlukları adresleyerek yazılım kalitesini arttırmaya yardımcı olur. Bu temel zorluklar, geliştiricilerin mimari yapı hakkındaki bilgi paylaşımı, yeni mimari stillere geçiş ve genellikle tecrübe ile aşılan geliştirme aşamasındaki problemlerdir (Shmidt, 1995).

Özellikle büyük sistemlerde çalışan, tecrübesiz geliştiricilerin yaşadığı sorunların başında sistemi anlamak için gereken eğitim gelmektedir. Tasarım kalıpları, yazılımdaki stratejilerin ve kullanılan taktiklerin yeni geliştiricilerin eğitilmesinde işe yarar. Tecrübeli geliştiricilerin, tecrübelerini somut olarak ortaya koymasına yardımcı olan tasarım kalıpları kullanımı, kritik mimari tasarımların belgelenmesini ve tartışılabilmesini beraberinde getirir (Shmidt, 1995).

Tüm bu avantajların yanında hem tasarım kalıpları kullanımının doğası gereği hem de tasarım kalıplarının yanlış kullanılmasından kaynaklanan olumsuz taraflar da bulunmaktadır. Tasarım kalıplarının getirdiği soyutlamanın yeni katmanlar (sınıflar) ile sağlanması, uygulamadaki sınıf sayısının artmasına neden olmaktadır. Bu durum göreceli olarak uygulamanın karmaşıklığını arttırmaktadır. Bunun yanında birleştirme tekniğinin kullanımının getirdiği esneklikle beraber uygulama yapısının öğrenilmesinin zorluğu da artar. Dinamik ve parametrik yapıların öğrenilmesi daha zordur. Ayrıca yaratılacak ve ilişkilendirilecek nesnelerin çalışma zamanında belirlenmesi, uygulamaların çalışma hızını olumsuz etkileyebilir (Gamma vd., 1995).

Bu bilgiler ışığında uygulama geliştirme esnasında tasarım kalıpları kullanılırken dikkatli karar verilmelidir. Bu anlamda tasarım kalıpları kullanımı için nesnel ve anlamlı bir kar-zarar değerlendirilmesi yapılmalıdır (Wendorff, 2001).

4.3 Tasarım Kalıpları ve Yazılım Çerçevesi Geliştirme

Bir çerçeve üzerine geliştirilmiş bir uygulamanın, uygulama özelinde değiştirilebilecek ve geliştirilebilecek yapılara sahip olabilmesi yazılım çerçevelerin genişletilebilir ve geliştirilebilir özelliklere sahip olmasını zorunlu hale getirir. Çünkü uygulama geliştiriciler özel bir uygulama için çerçeveyi, kalıtım özelliğini ya da çerçeve sınıflarından türemiş nesneleri birleştirerek kullanırlar (Krajnc ve Hericko, 2003). Dolayısıyla yazılım çerçevesinin

olabildiğince esnek ve genişletilebilir bir şekilde tasarlanması bir zorunluluktur (Gamma vd., 1995). Bu özelliklere sahip bir yazılım çerçevesi gerçeklenirken tasarım kalıplarından yararlanılması uygun olacaktır. Bu özellikleri sağlamak için, tasarım kalıplarını kullanan yazılım çerçevelerinin yüksek seviyede tasarım ve kod tekrar kullanımına ulaşması, tasarım kalıplarını kullanmayan çerçevelere göre daha olasıdır. Bu kalıplar, çerçeve mimarisinin, birçok farklı uygulama için tekrar tasarlanmadan kullanılabilmesine yardımcı olur (Gamma vd, 1995; Shalloway ve Trott, 2001).

Çerçeve tasarım özelliklerinin, çerçeve üzerine geliştirilen uygulamalarda da kendini göstermesi, aynı problem uzayındaki uygulamalarda belirli bir standardın yakalanmasını sağlar. Yazılım çerçeveleri ve tasarım kalıpları, birbirlerini karşılıklı olarak gerektiren standartlaşma ve esneklik gerektirmelerine bir cevaptır (Fach, 2001). Bu anlamda yazılım çerçevelerinde tasarım kalıpları kullanımı ile çerçeve ve geliştirilen uygulamalar arasındaki benzerlikler, hem çerçevelerin hem de uygulamaların bakım yapılabilirlik özelliklerini arttıracaktır. Çünkü yazılım çerçevelerinin tasarımları, sınıf, nesne ve metot ayrıntılarına girilmeden ve yararları bağlamında, tasarım kalıpları üzerinden tartışılabilir. Analiz, tasarım ve uygulama geliştirme aşamalarında genel bir soyutlama kümesi ve tasarım kalıplarının kullanımı, çerçeve üzerinde değişiklik yapılması aşamasında iyi bir iletişim mekanizması sağlar (Srinivasan, 1999).

Tasarım Kalıpları, yazılım geliştirilirken dinamik ve statik yapıları ve yazılımı oluşturan parçalar arasındaki işbirliğini ortaya koyması (Shmidt, 1995) sebebiyle çerçeve geliştirme ve geliştirilen uygulamaların dokümantasyon çalışmalarında da oldukça faydalı olacaktır.

5. İLGİLİ ÇALIŞMALAR

Bu başlık altında belirli bir alana ait uygulamalara temel oluşturmak için geliştirilmiş çerçevelerdeki tasarım kalıplarının kullanımı incelenecektir.

Yuan ve diğerleri VOD (*Video on Demand*) uygulamaları için genişletilebilir ağ yönetimi çerçevesi tasarlamışlardır. Çerçevede tasarım kalıpları, çerçeveye daha fazla esneklik ve genişletilebilirlik sağlamak için kullanılmıştır. Çalışmada birincil amaç ağ yönetimi uygulamalarındaki genişletilebilme probleminin aşılmasıdır. Çerçeve kullanılan Strateji Tasarım Kalıbı farklı algoritmaları aynı arayüz ile sunabilmek, Önyüz Tasarım Kalıbı karmaşık yapıları basit bir arayüz ile kullanabilmek, Arabulucu Tasarım Kalıbı nesneler arası bağımlılığı azaltabilmek ve Gözlemci Tasarım Kalıbı da veri kaynağının değişmesi durumunda, tüm nesnelere belirli bir hiyerarşide, haber verebilmek için kullanılmıştır. Sonuç olarak yüksek ölçekte genişletilebilir ve esnek bir çerçeve ortaya çıkmıştır (Yuan vd., 2006). VOD uygulamaları için geliştirilen çerçevede kullanılan Önyüz Tasarım Kalıbı, karmaşık bir yapıya sahip olan sistemlerin, kolay bir şekilde kullanılabilmesi için bir arayüz oluşturmayı önerir (Gamma vd., 1995). Böylelikle karmaşık sistem, içyapısını ortaya koymadan, sabit bir arayüz üzerinden kullanılabilecektir. Yuan ve diğerlerinin çalışmasında sistemler arası bilgi dönüşümünü sağlayan bir alt sistem bulunmaktadır. Alt sistem içinde tanımlanan on üç kural vardır ve her kural farklı bir grup için geçerlidir. Bu alt sisteminden yararlanacak bir kullanıcının tüm kuralları bilmesi gerekmektedir ve bu durum çerçeve kullanıcıları için karmaşıklık yaratabilir. Bu olumsuz durumu engellemek için alt sistem, Önyüz Kalıbı temelinde, FInterface arayüzü ile sunulmuştur. Bu arayüzdeki Convert() metodu kullanıcılar için bir kuralı seçecek ve ilgili nesneler arasındaki ilişkileri sağlayacaktır (Yuan vd., 2006). Böylelikle hem çerçeve kullanıcıları detayları öğrenmek zorunda kalmayacaktır, hem de sisteme yeni bir kural eklenmesi durumunda sistem arayüzünün değişmesine gerek kalmayacaktır.

Zhang ve diğerleri alan özel arama motoru uygulamaları için genişletilebilir bir çerçeve gerçeklemiştir. Bu çerçeve mimarisinin bazı bileşenlerinde tasarım kalıpları kullanılmıştır. *Focused Crawler* bileşeninde kullanılan Yapıcı Tasarı Kalıbı ile çerçeve kullanıcılarının diğer bileşenleri değiştirmeden yeni arama motorları eklemesi ve arama motoru nesnesi yaratılması işinin kullanıcılardan soyutlanması sağlanmıştır. *Text Filter* bileşeninde ise metin filtrelemede kullanılan yöntemin, filtreyi kullanan sınıf etkilenmeden değiştirilebilmesi için Strateji Tasarım Kalıbı kullanılmıştır. Çalışmada bahsedilen bileşenlerin mimarisinde kullanılan tasarım kalıpları sayesinde bileşenlere yeni eklenecek yapıların, diğer bileşenleri

etkilememesi sağlanmıştır (Zhang vd., 2003). Yapıcı Tasarım Kalıbı, yaratım süreci karmaşık bir nesnenin yaratım sürecini ile gösterilişini birbirinden ayırmayı, böylelikle aynı yaratım süreci ile farklı nesneler oluşturabilmeyi önerir (Gamma vd., 1995). Zhang ve diğerlerinin çalışmasındaki *Focused Crawler* bileşeninde farklı özelliklere sahip arama motorlarının aynı yaratım süreci ile oluşturulmalarını sağlayabilmek için Yapıcı Tasarım Kalıbı kullanılmıştır. Bu yapıda kullanıcı bir *Director* nesnesi yaratır ve bu nesneyi ilgili somut arama motoru nesnesi ile yapılandırır. Herhangi bir zamanda bilgi toplanmak istenirse *Director* nesnesi arama motoru ile haberleşir. Böylelikle kullanıcı arama motoru yaratma sürecinden bağımsızdır ve kullanıcının yapılandırma bilgisine göre farklı arama motorları kullanılabilir.

Srinivasan'a ait çalışmada ses tanıma alanında kullanılacak nesne tabanlı çerçeve tanıtılmıştır. Çalışma, tasarım kalıpları kullanılarak gerçeklenen çerçevenin, karmaşık yapıları ve sistem parçaları arasındaki ilişkileri tanımlamadaki yararları detaylandırılmıştır. Çerçevenin birincil amacı, ses uygulamaları geliştiricilerini, ses tanıma teknolojilerinden kaynaklanan karmaşıklıktan uzaklaştırmak ve geliştiricilerin çerçeveyi uygulamalarına özel olarak değiştirebilmelerini sağlamaktır. Bu amaca ulaşmak için Adaptör, Önyüz, Gözlemci ve Yegane Tasarım Kalıpları kullanılmıştır. Sonuç olarak çerçeve geliştirilirken kullanılan soyutlamalar ve tasarım kalıpları sayesinde çerçeve için gereken değişiklik istekleri için daha etkili bir tekniğe ulaşılmıştır. Ayrıca tasarım kalıplarının çerçevenin içsel tasarımını anlamaya, dokümantasyon çalışmalarına daha az bağımlı kalarak, yardımcı olduğu belirtilmiştir (Srinivasan, 1999). Adaptör Tasarım Kalıbı, var olan ancak arayüz uyuşmazlığı sebebiyle kullanılamayan yapıların, istenilen arayüz altında tanımlanarak kullanılabilir hale getirilmesi için kullanılabilecek bir yapı önerir. Srinivasan'ın çalışmasında var olan kullanıcı grafik sisteminin arayüz uyuşmazlığı sebebiyle karşılaşılan problemin çözümde Adaptör Tasarım Kalıbı kullanılmıştır. Adaptör sınıfı istenilen arayüzü sunarken kendisine gelen istekleri önceden varolan kullanıcı grafik arayüzüne iletmektedir. Böylelikle önceden arayüz uyuşmazlığı nedeniyle kullanılamayan alt sistem problemsiz bir şekilde kullanabilmiştir.

6. ÖRNEK YAZILIM ÇERÇEVESİ

Farklı üniversitelerin idari birimleri, yapıları ve işleyişleri açısından belirli kurallar dahilinde belirlendiğinden, ortak pek çok nokta barındırmaktadır. Üniversitelerdeki finansal işlerin takip edildiği Döner Sermaye, Saymanlık ve Strateji Geliştirme gibi birimlerde de ortak yapılar görülmektedir. Ancak ortak yapılar sadece birimlerin işleyişleri ile sınırlı değildir. Bu birimlerin kullandığı banka, maaş, personel, öğrenci, raporlama vb. gibi kavramlarda da benzerlikler vardır. Bu bölümde, bahsedilen benzer yapıları içinde barındıracak şekilde geliştirilmiş çerçevenin teknik özellikleri incelenecektir.

6.1 Üniversitelerin Finans Birimleri

Tüm üniversitelerin finans birimlerindeki uygulamalar belirli kanunlar ve kurallar dahilinde yapıldığı için bu uygulamaların ortak noktaları oldukça fazladır. Ne yazık ki neredeyse her üniversite kendi finans projesini, en temel adımlardan başlayarak geliştirmekte ve bu durum yazılım mühendisliğinin en temeli olan tekrar kullanımı azaltmakta ve kaynakların aynı işlemler için tekrar tekrar harcanmasına yol açmaktadır. Bu bölümde üniversitelerin finans birimleri için hazırlanacak projelere temel oluşturacak bir çerçevenin bazı bileşenlerinin geliştirilmesinde tasarım kalıplarının, tekrar kullanımı ve genişletilebilirliği desteklemek için, nasıl kullanıldığı anlatılmaktadır.

Çizelge 6.1 Finans birimleri arasındaki benzer yapılar

	Döner Sermaye	Saymanlık	Personel
Para	Dekont	Toplam Para Miktarı	Kullanabileceği Miktar
Personel	Proje Çalışanları	------	Proje Bilgileri
Doğrulama/ Yetkilendirme	Linux Sistemler	MS-Windows Sistemler	Veri Tabanı Üzerinden
Raporlama	X formatında belge	Y formatında belge	Z formatında belge

ALANA ÖZEL YAP

GENEL YAP

6.2 Finans Çerçevesinin Temel Özellikleri

Üniversitelerde kullanılan finans uygulamaların bakım sürecindeki en büyük güçlükler, finansal süreçlerde sık sık meydana gelen değişiklik istekleridir. İTÜ bünyesindeki finans

birimlerinde kullanılan programlar incelendiğinde, programların güncellenmesini ve\veya uygulamalara yeni eklentiler yapılmasını gerektirecek isteklerin, zamanla uygulamaları bakım yapılamaz hale getirdiği görülmüştür. Bu sebeple geliştirilen çerçevede esnekliği ve tekrar kullanımı en iyi seviyede kullanabilmek için Gri-Kutu Çerçeve yaklaşımı benimsenmiştir.

6.2.1 Genel Çerçeve Mimarisi

Finans Çerçevesi temel olarak iki ana bileşenden oluşmaktadır. Alan özel yapılar olarak adlandırılan kısımda finans birimleri ile ilgili Para ve Personel Yapıları toplanmıştır. Genel yapılar kısmında ise sadece finans projeleri ile ilgili olmayan Yetkilendirme\Doğrulama, Günlük tutma, Raporlama ve Menü Yapıları gerçeklenmiştir.

6.3 Çerçevenin Finans Alanına Özel Yapıları

Geliştirilen çerçevenin alan özel katmanında Para ve Personel Modülleri incelenmiştir. Bu modüller içinde gerçeklenen yapılar sadece üniversitelerin finans birimlerinde tanımlanmış kavramları içermektedir.

6.3.1 Para Modülü

Üniversitelerin finans birimleri tarafından kullanılan uygulamalardaki ortak bileşenlerden biri ise para miktarlarını, KDV oranlarını vb. bilgileri içeren banka dekontu, kasaya nakit yatırılan paralar için kasa dekontları ve bağış makbuzları gibi yapılardır. Bu yapılardan bundan sonra para tipleri olarak bahsedilecektir.

Uygulamanın kullanıldığı birime göre paranın sahip olacağı özellikler değişmektedir. Döner Sermaye için para ve paranın kaynağı bilgileri beraber kullanıldığında anlam taşımaktadır. Saymanlık tarafında ise toplam para miktarı ve ilgili harcama kalemleri ikilisi anlam taşımaktadır. Bunların yanında personel için paranın anlamı herhangi bir anda kullanabileceği para miktarıdır.

Uygulamalarda, bahsedilen para tipleri nesneleri tek tek kullanıldığı gibi, farklı para tipleri nesnelerinden oluşan kümeler de kullanılabilmektedir. Örneğin, bir işlem için seçilmiş paraların toplam miktarları gösterilmek istendiğinde, para tiplerine bakılmaksızın tüm paraların miktarlarının toplamı gösterilmelidir. Bu örnekte görüldüğü gibi uygulama tek bir para nesnesiyle ya da farklı tipteki paralardan oluşan, para nesneleri kümesiyle uyumlu çalışmalıdır. Bu problemi çözmek için Bileşik *(Composite)* Kalıbı'ndan yararlanılabilir.

Paralar ile ilgili bir diğer işlem de özellikle raporlamalarda kullanılan ve paranın düzeni ile ilgili yapıların hazırlanmasıdır. Bu yapılar paranın ondalıklı kısımlarının nasıl yuvarlanacağı, yazı ile nasıl gösterileceği ve birimi olarak sıralanabilir. Bir para nesnesinin uygulama

açısından anlamı, para miktarını gösteren ondalıklı bir değişken olmasıdır. Ancak bazı durumlarda, örneğin Yeni Türk Lirası biriminden bir para Amerikan Dolarına dönüştürüldüğünde uygulama açısından hiçbir şey değişmez çünkü paranın biriminin değişmesi, para miktarını ifade eden ondalıklı değişkenin bir sabit ile çarpılmasıdır. Bu durum para ile ilgili alınan raporlarda, paranın yazı ile gösteriliş şeklini ve düzenini etkileyecektir. Dolayısıyla Bu düzen değişikliği ile ilgili yapıların çalışma zamanında değiştirilebilmesi gerekmektedir. Bu problemi çözmek için Strateji (*Strategy*) Kalıbından yararlanılabilir.

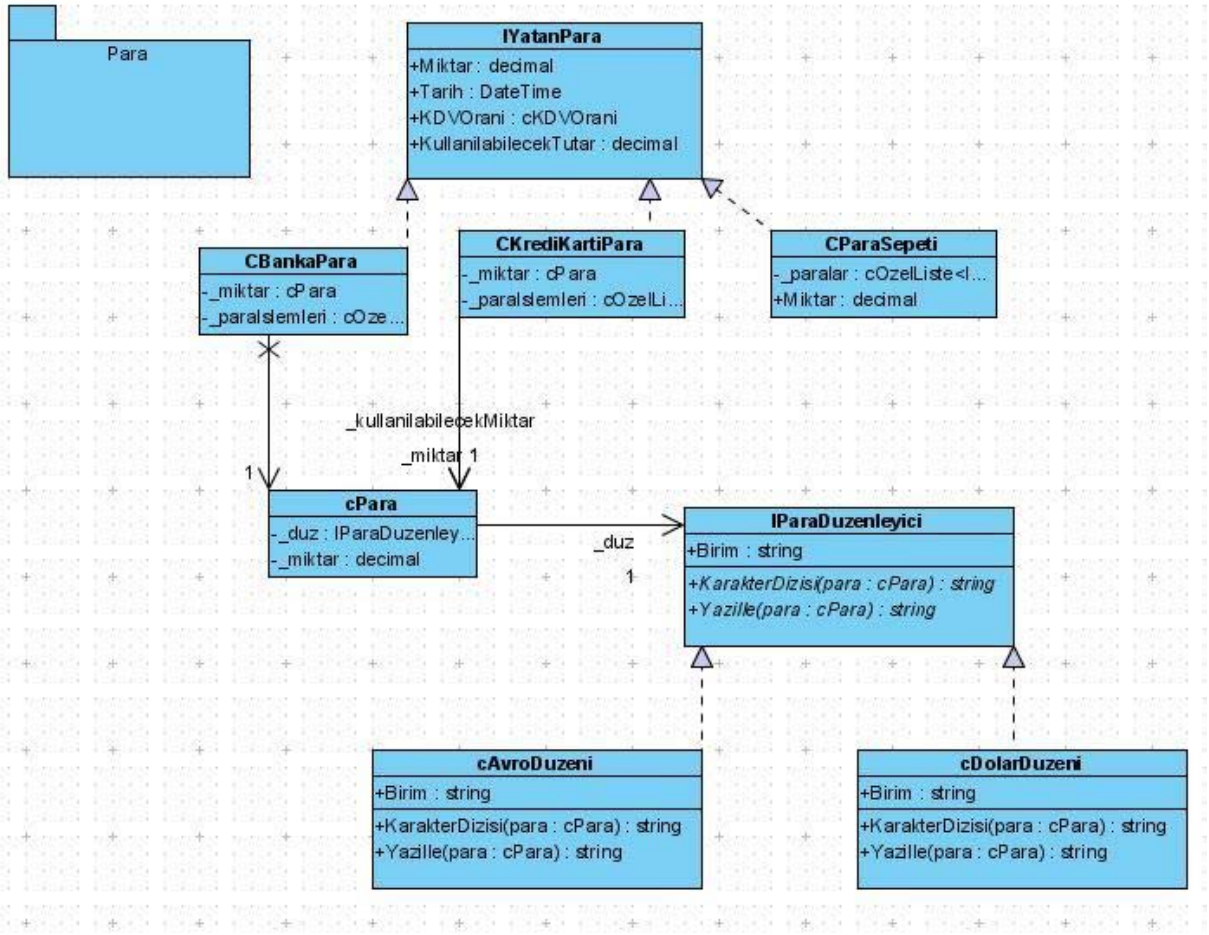
6.3.1.1 Para Modülü Tasarımı

Bileşik Tasarım Kalıbı, uygulamaların tek bir nesneye ya da nesnelerden oluşmuş kümelerle aynı şekilde çalışmasına yardımcı olmaktadır (Gamma vd., 1995). Diğer bir ifadeyle seçilmiş tek bir para ya da paraların toplam miktarını ekranda gösteren bir arayüz, farklı para tiplerinden ve para sayısından etkilenmemelidir. Bu durumun sağlanabilmesi için verilen örnekteki kulanıcı arayüzünün kendisine verilen nesnenin hangi sınıftan yaratıldığını bilmemelidir. Bu arayüzün bilmesi gereken tek şey kendisine verilen nesnenin, IYatanPara arayüzünü uygulamış bir nesne olacağıdır.

Şekil 6.1' de gösterilen tasarımda kullanılan Bileşik Kalıbı ile kullanıcı arayüzü ve CBankaPara, CKrediKartiPara ve CParaSepeti sınıfları arasındaki bağımlılık azaltılmıştır. Böylelikle arayüz hem her türlü tek ya da kümelenmiş para nesnesi ile nesnenin gerçek tipini bilmeden çalışabilmektedir hem de sisteme yeni bir para tipi eklendiğinde değiştirilmeden kullanılabilecektir.

Tasarımda incelenmesi gereken bir diğer yapı da CPara sınıfı ile para düzenlerinin belirlenmesi ile ilgilenen IParaDuzenleyici arayüzünü uygulayan sınıflar arasındaki ilişkidir. CPara sınıfından türemiş bir nesne kendi üzerindeki paranın miktarını bilmektedir ancak para miktarının raporlarda nasıl gösterileceği bilgisine sahip olmamalıdır. Çünkü bu özellikler paranın başka bir para birimine dönüştürülmesi durumunda ve bazen çalışma zamanında değişecektir. Bu durumda bu paranın düzeni ile ilgili işlerin, CPara nesnesi tarafından başka bir sınıfa yaptırılması uygun olacaktır. Bu tasarım özelliklerini sağlamak için Strateji Kalıbından yararlanılmıştır. Strateji Kalıbı aynı işi yaparken farklı yöntemler kullanan yapıların, bu yapıları kullanan istemcinin etkilenmeden değiştirilebilmesi amacı ile kullanılmaktadır (Gamma vd., 1995). CPara sınıfı nesnelerinin para düzeni ile ilgili işleri IParaDuzenleyicisi tipinden bir nesneye yaptırıyor olması, CPara sınıfı ile para düzeninden sorumlu sınıflar arasındaki bağımlılığın azaltılması anlamına gelmektedir. Böylelikle yeni bir para düzeni sınıfı eklendiğinde ya da para düzeninde meydana gelebilecek bir değişimden

CPara nesneleri etkilenmeyecektir. Bu durum bakım maliyetlerini azaltacaktır.



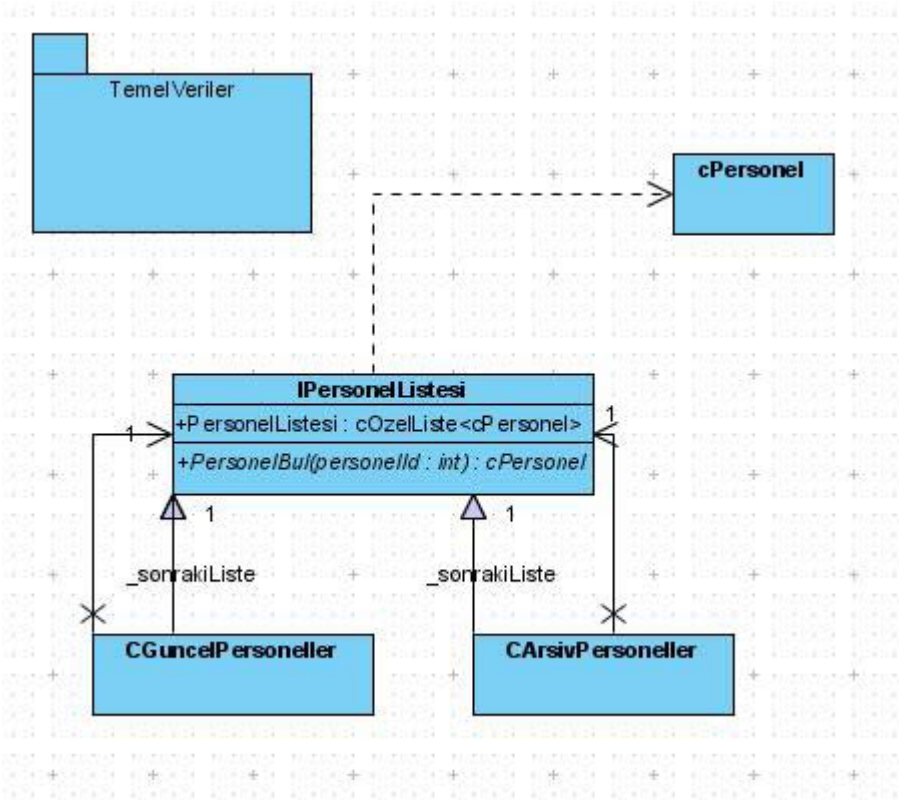
Şekil 6.1 Para modülü UML gösterimi

6.3.2 Personel Modülü

Üniversitelerde kullanılan finans uygulamalarında temel amaç para işlemlerinin hatasız şekilde tamamlanmasıdır. İşlemlerin pek çoğu üniversite personeli ile ilişkili iken bazı durumlarda üniversite dışı çalışanların da bu işlemler ile ilişkilendirilmesi gerekmektedir. Bu anlamda üniversite personel bilgilerinin alındığı temel bir liste var olsa bile uygulama özelinde oluşturulacak farklı bir listenin, üniversite dışı personeller gibi, de temel liste ile beraber uyumlu bir şekilde kullanılabilmesi gerekmektedir. Bazı uygulamalarda temel liste ile kullanılacak birden fazla farklı liste de olabilmektedir. Bu anlamda personel bilgilerinin tutulacağı yapının birden fazla liste ile uyumlu çalışmasının yanında yeni listelerin de kolay bir şekilde bu personel yapısına eklenebilmesi gerekmektedir.

Birbirine benzeyen yapıların beraber kullanılması için tüm listelerdeki elemanların aynı listede toplanması bir çözüm olarak sunulabilir ancak tüm liste elemanları benzer bir yapıya sahip olmayabilir. Bu problem göz önüne alınınca tüm listelerin toplaması yerine tüm

listelerin ortak olarak çalışması daha uygun bir çözüm olacaktır. Şekil 6.2' deki tasarımda Sorumluluklar Zinciri Kalıbı temel alınmıştır ve farklı listelerin ortak çalışması sağlanmıştır.



Şekil 6.2 Personel listesi yapısı ve sorumluluklar zinciri kalıbı

Şekil 6.2'deki ana unsur sistem nesnelerin CGuncelPersoneller ya da CArsivPersoneller nesneleri ile doğrudan ilişki kurmak yerine IPersonelListesi ile çalışmasıdır. Farklı listelerin bu yapıya katılmasının şartı, IPersonelListesi arayüzünü uygulamaları ve aranan bilgiye kendisi sahip değilse bir sonraki listeyi işaret etmeleridir. CGuncelPersoneller ve CArsivPersoneller sınıfları, IPersonelListesi arayüzünü uygulamıştır ve bir sonraki listeyi adreslemektedir. Böylelikle her liste kendi üzerinde arama yapar eğer personel bulunmazsa bir sonraki listede arama yapılmaya devam edilir çünkü her liste kendisinden sonraki aranacak listeyi bilmektedir. Bu işlem aranan personel bulunana ya da listeler bitene kadar devam eder. Bağlı liste yapısına benzeyen bu tasarımda Sorumluluklar Zinciri Kalıbı temel alınmıştır. Bu kalıp, istekte bulunan nesnenin, belirli bir arayüzü uygulayan tek bir nesne ile haberleşmesini dolayısıyla kendisine esas cevap verecek nesnenin hangisi olduğunu bilmesi zorunluluğunu ortadan kaldırır (Gamma vd., 1995). Her sınıfın kendisinden sonra gelen sınıfı işaret etmesi sebebiyle bu sınıflar listesi istenildiği kadar uzatılabilir. Bunun yanında sistem sadece IPersonelListesi ile haberleştiği için yeni bir listenin sisteme eklenmesi istemcinin değiştirilmesini gerektirmez.

6.4 Çerçevenin Alan Bağımsız Yapıları

Çerçevenin alan bağımsız yapıları Doğrulama ve Yetkilendirme, Günlük Tutma, Raporlama ve Menü Modüllerini içermektedir. Bu modüller finans alanı dışındaki uygulamalarda da kullanılabilecektir.

6.4.1 Doğrulama ve Yetkilendirme

Doğrulama, bir istemcinin kim olduğunun belirlemesidir. Bu işlem normal olarak bir kullanıcı adı ve parola kontrolü ile gerçekleştirilir. Yetkilendirme ise doğrulama aşamasını geçmiş bir kullanıcının sistemde bulunan kaynaklara ulaşabilme seviyesinin belirlenmesidir (Chen, 2004).

Bir *Windows* uzayında bulunan kullanıcının doğrulama ve yetkilendirme işlemi *Active Directory* Servisi (ADS) ile yapılabilir. Bu durumda ADS, verilen kullanıcı adı ve parola ile kullanıcının kimliği ve dahil olduğu güvenlik gruplarını sistem için hazırlar. Ancak bu işlemlerin her zaman ADS kullanılarak yapılması mümkün olmayabilir. Örneğin, sistem uzayında bir ADS bulunmayabilir, sistem bir *Windows* uzayı içinde olmayabilir ya da özel nedenlerle kullanıcı ad, parola ve rol bilgileri bir veri tabanında saklanıyor olabilir. Bu durumda geliştireceğimiz Doğrulama ve Yetkilendirme Sistemi'nin farklı yapılar üzerinden çalışabilmesi ve gelecekte bu sistemlerin değişmesine olanak vermesi gerekmektedir.

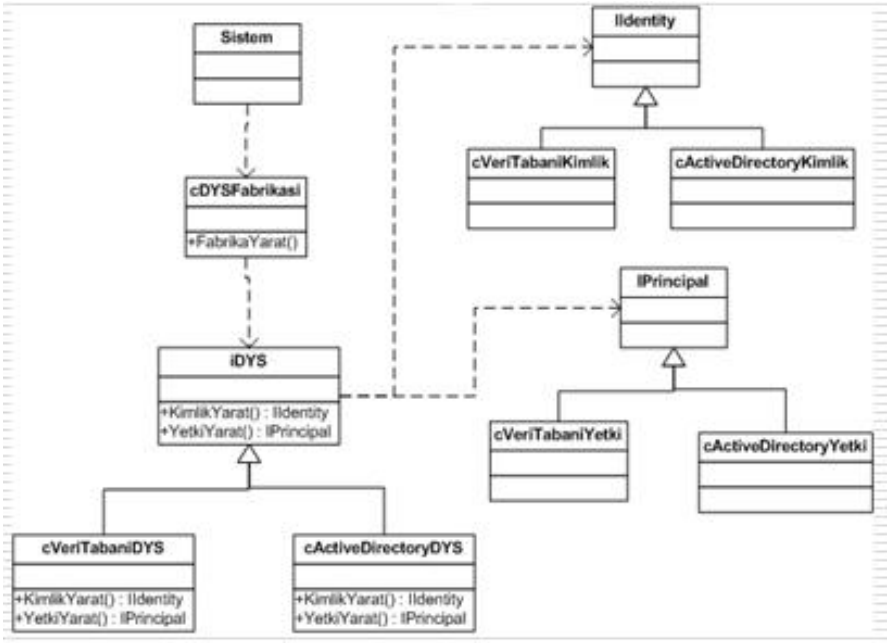
6.4.1.1 .NET ve Doğrulama ve Yetkilendirme Yapısı

.NET uygulamalarında kullanıcı kimliği sunan sınıfların *System.Security.Principal.IIdentity* arayüzünü uygulaması gerekmektedir. Kullanıcı rollerini içeren sınıfların ise kullanması gereken arayüz *System.Security.Principal.IPrincipal'* dır.

6.4.1.2 Doğrulama ve Yetkilendirme ve Soyut Fabrika Tasarım Kalıbı

Geliştirilen DYS' yi kullanacak bir uygulama geliştiricinin kullanıcı kimliğinin ve rollerinin hangi sistemler üzerinden belirlendiğini bilmesi, bilmek zorunda olması, geliştiriciye ek bir yük getirecektir. Bu sebeple kullanıcıyı bu işlemden soyutlamak için Soyut Fabrika Kalıbı' ndan yararlanılabilir. Soyut Fabrika Kalıbı, birbirine bağımlı ya da ilişkili nesne gruplarının yaratım sürecini istemciden soyutlamak için kullanılmaktadır (Gamma vd., 1995).

DYS'de kullanıcı kimliğinin ve rollerinin belirlenmesi için kullanılabilecek *Active Directory* ve veri tabanı yapılarının bulunduğu varsayılmıştır.



Şekil 6.3 Doğrulama ve yetkilendirme yapısı ve soyut fabrika kalıbı

Şekil 6.3' teki UML diyagramındaki dikkat edilmesi gereken ilk nokta Sistem'in sadece cDYSFabrikasi ile ilişki içinde olduğudur. cDYSFabrikasi tipinden bir nesne yaratıldıktan sonra bir değişkenden alınan bilgiye göre cVeriTabaniDYS ya da cActiveDirectoryDYS tipinden bir nesne yaratılacaktır. Yaratılan nesne de ilgili *IIdentity* ve *IPrincipal* tipinden nesneleri yaratacaktır.

Soyut Fabrika Kalıbının kullanıldığı bu tasarımda Sistem, *IIdentity* ve *IPrincipal* tipinden nesnelerin yaratılması ile ilgili ayrıntılarla ilgilenmek zorunda değildir. Sistem kodunun içinde *IIdentity* ve *IPrincipal* tipinden somut nesneler yaratılmasına gerek yoktur. Yeni bir doğrulama ve yetkilendirme yapısının kullanılması gerektiğinde ise Şekil 6.3' teki yapı kullanılarak cYeniDYS sınıfının yaratıldığında, Sistem kodunun güncellenmesine gerek kalmayacaktır çünkü Sistem sadece tanımlanmış arayüzlerle çalışmaktadır.

6.4.2 Günlük Tutma Sistemi

Uygulama programları sistemde gerçekleşen olayları, sonradan incelenmesi yararlı olacak kritik bilgileri saklamak ister. Toplanan veri kümesi genelde istatistiksel bilgi elde etme amacıyla kullanılmaktadır. Bunun yanında sistemde önceden belirlenmiş önemli işlemlerin takip edilmesi, işlemler zincirinin her adımının özel olarak incelenebilmesi ve saklanması için yine bir günlük mekanizmasının kullanılması gerekmektedir. İş katmanı ile ilgili bu tip günlük bilgilerini tutma işleminin son adımı, uygulama kullanıcılarına alarmlar üretmek ya da ilgili durumu içeren bir e-posta göndermek olabilir. Uygulama geliştiriciler açısından günlük

tutma mekanizması yine aynı amaçlarla kullanılabilir ancak programcıların tutuğu veriler genelde program ile ilgili teknik bilgiler olacaktır. Örneğin, kullanılan bir dış servise ulaşılamadığı durumda, bunun sebebi, zamanı, süresi gibi bilgiler bir günlük dosyasında tutulur. Daha sonradan bu bilgiler sistemin en iyilenmesi çalışmalarında kullanılabilir.

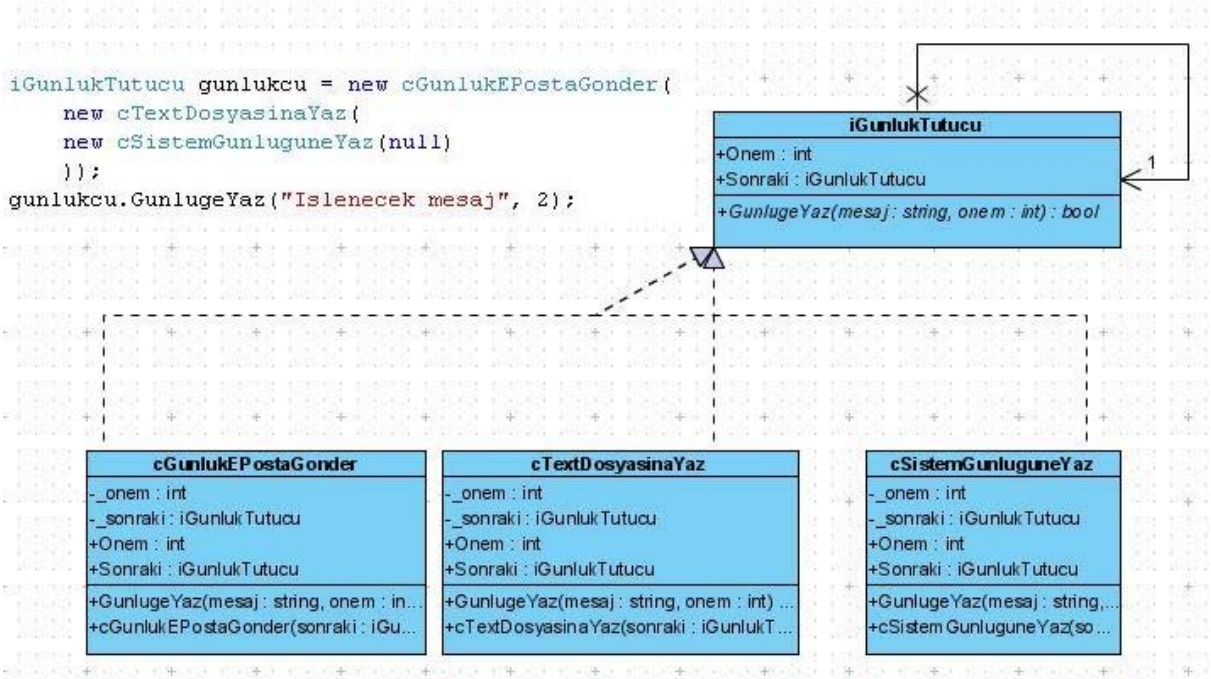
Bahsedilen tüm durumlarda günlük tutma işlemi iki aşamadan oluşmaktadır. Birinci adım kritik olaylarla ile ilgili verinin seçilmesi, ikinci adım ise bu verinin uygun bir ortama gönderilmesi ve işlenmesidir. İkinci aşamada önemli nokta uygun ortamın seçilmesidir. Bu anlamda uygun ortam bir kullanıcı arayüzü, e-posta sunucusu ya da bir metin dosyası olabilir. Daha detaylı bir günlük sisteminde uygun ortamın verinin anlam/önem seviyesine göre otomatik olarak seçilmesi gerekebilir. Bu bağlamda günlük olarak tutulacak verinin yapısı, saklanacağı ortamın belirlenmesi ve saklama işleminin yapılması gibi üç kritik işlem vardır.

Günlük tutma hizmetinin sağlaması gereken özellikler ve çerçeve yapısı içinde gerçeklenmesinin sebepleri aşağıda verilmiştir.

1) Günlük tutma hizmeti sorumluluğu uygulama programcısında olmamalıdır. Bu durumda uygulama programcısının, verinin işlenmesi ile ilgili alt yapıları bilmesi gerekmektedir (Dosya sistemi, e-posta sistemi gibi).
2) Saklanacak verinin düzeninin belirlenmiş standart bir yapıda olması gerekmektedir.
3) Bu hizmete yeni bir ortam eklendiğinde uygulama programı bu değişiklikten etkilenmemelidir.
4) Tüm finans uygulama programları için 1, 2 ve 3 numaralı özellikler geçerlidir.

6.4.2.1 Günlük Tutma Sistemi Tasarımı

Sorumluluklar Zinciri Kalıbı, benzer işler yapan nesneler zincirinin oluşturulması için kullanılır. Her nesnenin belirli durumlarda işlem yapması istenmektedir. Ayrıca bu nesneler zincirine yeni işlem nesneleri de eklenebilmelidir.



Şekil 6.4 Günlük tutma sistemi tasarımı

Şekil 6.4'te gösterilen yapı ile her iGunlukTutucu tipinden nesne kendisine gelen mesaj ile ilgili bir karar verecektir. Bu karar, mesaj ile birlikte gelen ve mesajın önemini belirten tamsayı tipinden değişken aracılığıyla verilir. Eğer mesajı alan nesne, mesajı işleyecekse gereken işlemleri yapar ve istemciye işlemin sonucunu döndürür ancak mesajı işlemeyecekse nesne üzerinde Sonraki özelliği ile belirlenmiş ve iGunlukTutucu tipinden nesneye mesajı ve tamsayıyı gönderir. Bu işlemler zinciri mesaj işlenene kadar devam eder.

Sorumluluklar Zinciri Kalıbı sayesinde günlük tutma hizmetini kullanacak uygulama programcıları basit bir arayüz üzerinden bu yapıyı kullanabilir. Bu zincire dahil olacak her işlem tipi için bir sınıfın yaratılmış olması sınıf sayısını arttırmaktadır; bunun yanında her sınıfın sadece tek bir işlem ile ilgileniyor olması bakım işlemlerini kolaylaştıracaktır.

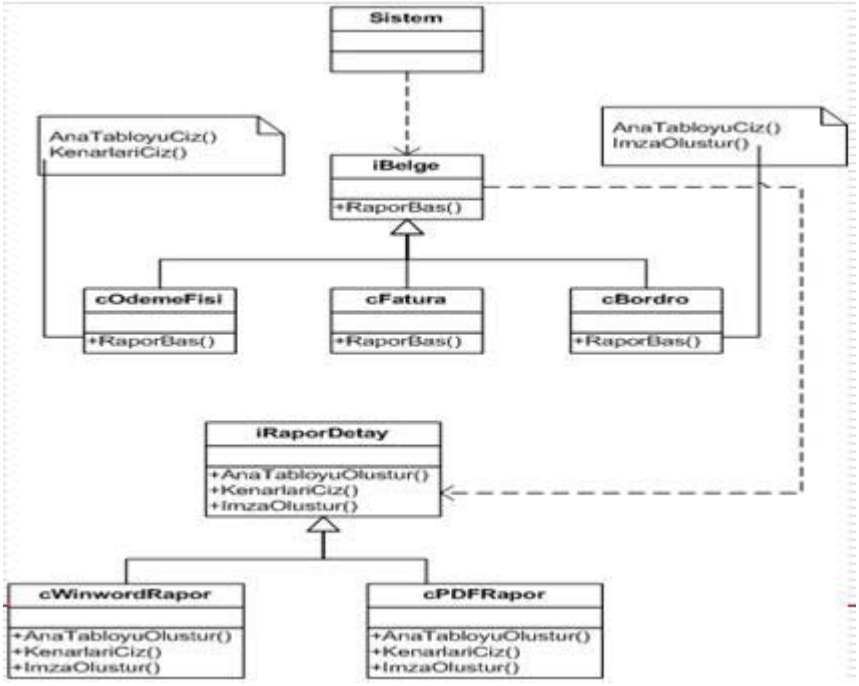
6.4.3 Raporlama Sistemi

Üniversitelerin finans birimlerinin kullanacağı uygulamaların temelini oluşturacak bir çerçevedeki bileşenlerden biri de Raporlama Modülü' dür. İşlem sonuçlarını kontrol eden birimlerin isteklerine göre raporların formatları (*Excel*, *Pdf*, *Word* vb.) ve içerdikleri bölümler (tarih, açıklama alanı, imza vb.) farklı olabilmektedir. Bunun yanında rapor formatlarının çalışma zamanında belirlenmesi gibi istekler de karşılanmalıdır. Dolayısıyla geliştirilecek sistemin bu isteklere ve gelecekte olabilecek değişiklik isteklerine cevap verebilecek yapıda tasarlanması gerekmektedir.

6.4.3.1 Raporlama Yapısı ve Köprü Tasarım Kalıbı

Finans uygulamalarındaki işlemlerin çoğu bir rapor ile sonlanmaktadır. Bu raporların bir dökümü de genelde kağıt üzerinde istenmektedir. Raporlarda bulunan alanlar ana tablo, kenar çizgileri ve özellikleri ve imza alanı olarak üç gruba ayrılabilir. Ancak her raporda bu üç alanın farklı kombinasyonları bulunabilmektedir. Bazı raporlar PDF, bazıları EXCEL ya da TXT dosyaları şeklinde kaydedilmek istenmektedir. Bu yapıdaki esas kritik nokta ise aynı rapor farklı zamanlarda farklı formatlarda istenebilmektedir. Bu problemi çözmek için hem farklı alanlardan oluşan rapor yapılarının hem de çalışma zamanında rapor formatlarının değiştirilmesini sağlayan bir yapı tasarlanmalıdır.

Köprü Tasarım Kalıbı, soyutlamanın ve bu soyutlamanın uygulanışlarını birbirinden ayırmak için kullanılır. Böylelikle soyutlama ve uygulama birbirinden bağımsız şekilde değiştirilebilir (Gamma vd., 1995).



Şekil 6.5 Raporlama modülü ve köprü kalıbı

Şekil 6.5' te gösterilen Raporlama yapısında dikkat edilmesi gereken temel nokta Sistem' in sadece iBelge arayüzü ile ilişki içinde olmasıdır. cFatura, cOdemeFisi ve cBordro sınıflarının RaporBas() metotları ve bu metotların raporları farklı formatlarda hazırlamak için kullanacakları farklı bileşenler birbirinden ayrılmıştır. Görüldüğü gibi her iBelge tipinden sınıf, hazırlanacak raporun hangi alanlardan oluşacağını bilmektedir ancak raporun hazırlanacağı ortam ile ilgili bilgiye sahip değildir. Her iBelge nesnesi farklı formatlarda rapor hazırlayabilmek için iRaporDetay tipinden nesneleri kullanacaktır. Böylelikle iRaporDetay

tipinden nesnelerin çalışma zamanında değiştirilmesi ile aynı iBelge nesnesi için farklı formatlarda raporlar, nesnenin iç yapısı değiştirilmeden alınabilmektedir. Ayrıca yeni bir rapor formatı ya da raporların içeriklerinin değiştirilmesi isteğine, istemci etkilenmeden cevap verilebilecektir.

6.4.4 Menü Yapısı

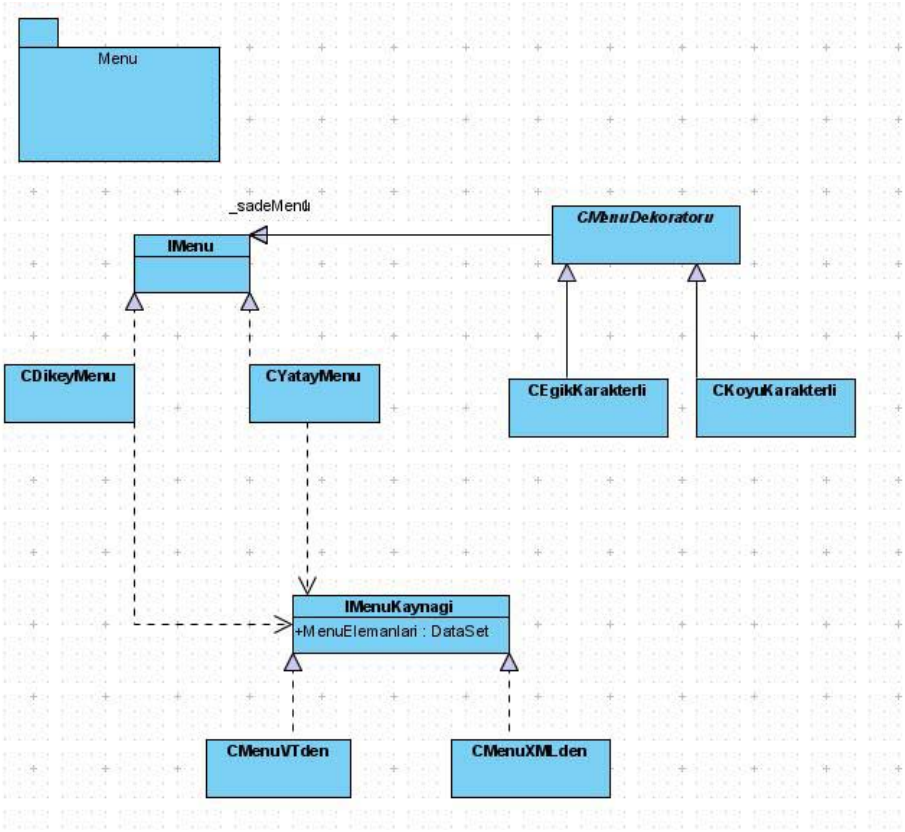
Web tabanlı finans uygulamalarının birçok parasal işlemi yerine getirmek için geliştirildikleri düşünülürse bu uygulamalarda tek bir web sayfasının bulunması yeterli olmayacaktır. Temel olarak sayfalar arası hiyerarşinin oluşmasını sağlayan menüler çoğu uygulamada kullanılmaktadır. Bu sebeple geliştirilen çerçeveye web tabanlı uygulamalar için kullanılabilecek bir menü yapısı da eklenmiştir.

Menülerin barındırması gereken en temel iki özellik, yapının menü ve alt menü kavramlarını içermesi ve seçilen menü elemanına göre yönlenilecek sayfanın belirlenmesidir. Bu işlem oldukça basit olduğu için çoğu zaman sabit menüler hazırlanmaktadır. Ancak finans uygulamalarına sürekli yeni elemanların (özellikle raporlama sayfaları) eklenmesi ve menülerin görünüm özelliklerinin farklı uygulamalarda değişebilir olması, sabit hazırlanan menülerin sürekli güncellenmesini gerektirmektedir. Bu durum da uygulamaların bakım maliyetlerini arttırmaktadır.

Çerçeve kapsamında geliştirilen menü yapısının en önemli özelliği pek çok menü özelliğinin dinamik olarak çalışma zamanında değiştirilebilmesidir. Bunun yanında menü elemanlarının görünüm adlarının ve yönlendirilecek sayfaların isimlerinin bir veri tabanından ya da XML dosyasından alınabilmesi bu dinamik yapıyı desteklemektedir. Bu dinamik özellikleri kazandırmak için menü yapısında Dekoratör ve Strateji Kalıplarından yararlanılmıştır.

Şekil 6.6’ da gösterilen Menü yapısının özelliklerinin dinamik olarak değiştirilebilmesini sağlayan ana yapı arayüz ve soyut sınıfların kullanılmasıdır. Menü elemanlarını veri tabanından okuyan CMenuVTden ve bir XML dosyasından okuyan CMenuXMLden sınıfları IMenuKaynagi arayüzünü uygulamaktadır. Strateji kalıbının kullanılması ile CDikeyMenu ve CYatayMenu sınıfları için menü elemanlarının okunacağı yapı önemini yitirmektedir, çünkü menüler için önemli olan IMenuKaynagi arayüzünü uygulamış bir nesne ile beraber çalışmaktır. Böylelikle herhangi bir menü herhangi bir kaynağı, menü elemanlarını oluşturmak için kullanabilecektir. Yeni bir menü kaynağı eklendiğinde menü oluşturan sınıfların güncellenmesine gerek kalmayacaktır. Bunun yanında yeni bir menü eklendiğinde ise yeni menü için kaynakların güncellenmesi gerekmeyecektir. Dolayısıyla menü ve kaynakları arasında gevşek bağımlı bir tasarım oluşturulmuştur.

Menü yapısının bir diğer özelliği de menünün görünüm özelliklerinin (menü elemanlarının eğik karakterli ve/veya koyu renkli yazılması gibi) çalışma zamanında değiştirilebilmesidir. Bu gereklilik farklı yetkilerdeki kullanıcılara aynı menü elemanlarının farklı özelliklerle sunulabilmesini sağlar ve kullanıcıların kendilerine göre özelleştirebilecekleri kullanıcı dostu bir menü oluşturur. Ayrıca menü görünümünün bir özelliği daha değiştirilmek istenirse CDikeyMenu ya da CYatayMenu sınıflarının güncellenmesine gerek kalmayacaktır. Uygulamadaki menünün sadece yaratılma sürecindeki küçük bir değişiklik sayesinde menü özellikleri değiştirilebilecektir.



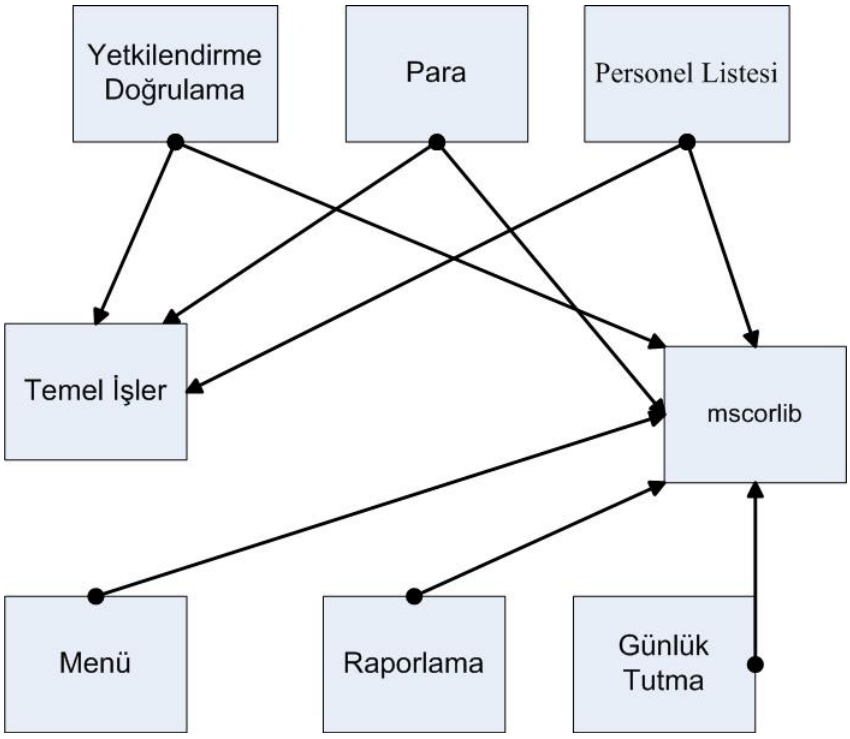
Şekil 6.6 Menü yapısı tasarımı

7. TASARIM KALIPLARININ ÇERÇEVE ÖZELLİKLERİNE ETKİLERİ

Bu bölümde kullanılan tasarım kalıplarının geliştirilen Finans Çerçevesi'nin kalite özelliklerini nasıl etkilediği anlatılacaktır. Kalite özellikleri 2.2'de Yazılım Çerçevelerinin Özellikleri başlığı altında anlatılan Tekrar Kullanılabilirlik, Modülerlik, Genişletilebilirlik, Basitlik ve Kullanılabilirlik ve Bakım Yapılabilirlik kapsamında ve Chidamber ve Kemerer ölçütlerine göre incelenecektir.

Nesne tabanlı bir uygulamanın sağlamlığını ve kalitesini ölçebilmek için kullanılan C&K (Chidamber ve Kemerer, 1994) ölçüt kümesinin miras alma ağacının derinliği (DIT), bir sınıftan doğrudan türetilen sınıf sayısı (NOC) ve nesneler arası bağımlılık (CBO) elemanları kullanılabilir. Bu ölçütler ile uygulamadaki sınıfların sağlamlığı, diğer bir deyişle bir sınıftaki bir değişikliğin diğer sınıfların da değiştirilmesi zorunluluğunu ortaya çıkarmaması, arasında negatif bir ilişki vardır (Elish ve Rine, 2003; 1).

7.1 Modüller Arası İlişkiler ve Çerçevenin Sağlamlığı



Şekil 7.1 Modüller arası ilişkiler

Şekil 7.1' de görüldüğü gibi çerçeve elemanlarının birbirlerine olan bağımlılıkları olabildiğince azaltılmıştır. Buradaki birincil amaç çerçeve elemanlarının farklı uygulamalarda birbirlerinden bağımsız olarak kullanılabilmesini sağlamaktır. Ancak bahsedilen modüler

yapıyı sağlamak adına tüm yapıları birbirinden ayırmaya çalışmanın olumsuz etkileri de olacaktır. Örneğin TemelIsler modülünün görevleri, çoğu uygulamada bulunan karakter dizileri ve listeler üzerindeki işlemlerdir. Bu işlevlerin bağımlılığı azaltmak adına farklı modüllerde tekrarlanması kazançlı bir çözüm olmayacaktır. Dolayısıyla belirli bir düzeyde kalmak şartıyla TemelIsler modülü ile YPara ve YetkilendirmeDogrulama modülleri arasında bir bağımlığa izin verilmiştir. Bu durum C&K ölçütlerindeki nesneler arası bağımlılığın artmasına yani çerçevenin sağlamlılığının bir miktar azalmasına yol açmaktadır. Özetlemek gerekirse kod parçalarının tekrar kullanımının sağlandığı bu durumda, bağımlılık bir ölçüde artmıştır. Miras alma ağacının derinliği ve bir sınıftan doğrudan türetilen sınıf sayısı çerçeve genelinde azdır. Bu durumun sebebi tasarım kalıpları kullanmanın getirdiği birleşim yaklaşımının benimsenmesidir.

Çizelge 7.1 Modüller arası etkileşim

Assembly	# Types	# Abstract Types	# IL instruction	Afferent Coupling	Efferent Coupling	Relational Cohesion	Instability	Abstractness	Distance
YYetkilendirmeDogrulama v1.0.0.0	10	2	333	0	12	1.6	1	0.2	0.2
YRaporlama v1.0.0.0	6	2	43	0	2	2	1	0.33	0.33
YPara v1.0.0.0	12	4	510	0	13	1.92	1	0.33	0.33
TemelIsler v1.0.0.0	7	3	489	11	8	2	0.42	0.43	0.15
Menu v1.0.0.0	9	3	436	0	4	1.33	1	0.33	0.33
GunlukTutma v1.0.0.0	7	2	236	0	13	1.14	1	0.29	0.29

Çizelge 7.1' de kodun tekrar kullanımını sağlamak adına, TemelIsler modülünün bağımlılığı artmıştır ve modülün 11 farklı yerde kullanıldığı görülmektedir (*Afferent Coupling*). Ancak TemelIsler modülü dışındaki modüller arasında herhangi bir sıkı-bağlı yapının olmaması çerçevenin ortalama sıkılık (*Instability*) ve soyutluk (*Abstractness*) özelliklerinin yüksek olmasını sağlamıştır. Böylelikle çerçeve değişikliklere kapalıyken geliştirilmeye açık hale gelmiştir.

7.2 Tekrar Kullanılabilirlik

Kaynak kodların tekrar kullanımı özelliği için Para modülünü oluşturan yapılar incelenebilir. Para modülünde Bileşik Kalıbının kullanımı ile farklı para sınıflarının tek bir liste (CParaSepeti) altında toplanıp kullanılabilmesini sağlamıştır. Bu durum CParaSepeti sınıfının herhangi bir para tipi ile sıkı bağımlı olmamasının bir sonucudur. Böylelikle CParaSepeti sınıfı farklı bir uygulamada farklı para tiplerinden oluşan nesneleri de kendi içinde tutabilecektir. Para miktarlarını ifade edebilmek için kullanılan CPara sınıfı ile, bu paraların farklı para birimlerine göre farklı görünüm özellikleri almasını sağlayan sınıflar (CAvroDuzeni, CdolarDuzeni) arasında kullanılan Strateji Kalıbı sayesinde ortaya çıkan gevşek bağımlı yapı sayesinde CPara, CAvroDuzeni ve CDolarDuzeni sınıfları farklı

uygulamalarda bağımsız şekilde kullanılabilirler. Mimari açıdan tekrar kullanım CParaSepeti ve diğer para tiplerinin IYatanPara arayüzünü uyguluyor olmalarından kaynaklanır. IYatanPara arayüzünde belirlenmiş alanlar ile ilgilenen yapıların, örneğin bu alanlardaki bilgileri gösteren bir ekranın, yeni bir para tipi oluşturulduğunda güncellenmesi gerekmeyecektir.

Mimari tasarımın tekrar kullanımına diğer bir örnek olarak Personel Listesi yapısı incelenebilir. 6.3.2'de anlatılan Personel Listesi yapısındaki Sorumluluklar Zinciri Kalıbı ile istenildiği kadar listenin Sistem tarafından sanki tek bir listeymiş gibi görülmesi sağlanmıştı. Bu yapı farklı bir uygulamada farklı listeler ile kullanılmak istendiğinde, yapı değiştirilmeden yeni listelerin eklenmesi ile kullanılır hale gelecektir. Bu durumda da her iki uygulamadaki Personel Listesi mimari olarak birbirinin aynısı olacaktır. Böylelikle farklı uygulamalarda kullanılan Personel Listesi yapıları için aynı test ortamları, dökümantasyonlar vb. kullanılabilecektir.

7.3 Modülerlik

Modülerliğin uygulamadaki karşılığı, 2.2.2'de bahsedildiği gibi, değiştirilen ya da güncellenen parçaların diğer yapıları etkilememesidir. Dekoratör kalıbının kullanıldığı ve menü elemanlarının görünüm özelliklerinin çalışma zamanında değiştirilmesini sağlayan menü tasarımı modülerliğe güzel bir örnektir. Özelliklerin değişmesini sağlayan CEgikKarakterli ve CKoyuKarakterli sınıflarına ek olarak yeni bir CRenkliKarakterli gibi menü elemanlarının renklerini değiştirmemizi sağlayan bir sınıf yazdığımızı varsayalım. Bu durumda özelliklerini değiştirmek için bu sınıflardan türemiş nesneleri kullanacak CDikeyMenu ve CYatayMenu sınıflarının değiştirilmesine gerek kalmayacaktır. Bunun nedeni CDikeyMenu ve CYatayMenu sınıflarının CMenuDecoratoru soyut sınıfı ile uyumlu çalışmasıdır. Bu modüler yapı Dekoratör kalıbı ile sağlanmıştır.

Modüler yapıya diğer bir örnek olarak Sorumluluklar Zinciri kalıbının kullanıldığı Günlük Tutma Yapısı gösterilebilir. 6.4.2' de incelenen bu yapıda konfigürasyon dosyasından okunan bilgiye göre bilginin yazılacağı günlük alanının seçilmesi tasarımı bulunmaktadır. Günlük tutma yapısından yararlanan Sistem'in, yapıya yeni bir alan eklendiğinde değiştirilmesi gerekmeyecektir çünkü Sistem, günlük tutma yapısının temelini oluşturan IGunlukTutucu arayüzü ile uyumlu çalışmaktadır. Yeni eklenen alan tamamen IGunlukTutucu arayüzünün arkasında kalacaktır.

7.4 Basitlik ve Kullanılabilirlik

Her mühendislik ürününde olduğu gibi gerçeklenen yapının basit ve kullanılabilir olması, ürünün müşteriler tarafından kabul edilmesini kolaylaştıracaktır. Geliştirilen Finans Çerçevesinde bu özellikler, çerçeve üzerinde uygulama geliştirenlerin, mümkün olduğunca nesne yaratma ve nesneler arasındaki ilişkilerin tanımlanma işlemlerinden soyutlanması ile sağlanmıştır. 6.4.1 başlığı altında incelenen YDogrulamaYetkilendirme modülünün temelini oluşturan Soyut Fabrika kalıbı sayesinde, doğrulama ve yetkilendirme işlemlerini yerine getirecek sınıflar ile bu sınıfların yaratılması süreci birbirinden ayrılmıştır. Böylelikle bu sınıfları kullanacak uygulama geliştiricilerin, sınıfların yaratım süreci ile ilgili detayları bilmelerine gerek kalmayacaktır.

7.5 Bakım Yapılabilirlik

Bakım yapılabilirlik ile modülerlik karşılık olarak sağlanan özelliklerdir. Uygulamanın bakım yapılabilir olması pratikte yeni versiyonların kolayca güncellenebilmesi ve bir parçadan yapılan değişikliğin diğer yapıları etkilememesi olarak ortaya çıkar.

Personel Listesi yapısında, gerçeklenen tasarımda IPersonelListesi arayüzünü uygulayan CGuncelPersoneller ve CArsivPersoneller sınıfları bulunmaktadır. Personel Listesini kullanan bir Sistem'in bilmesi gereken tek yapı IPersonelListesi arayüzüdür. Bu sayede farklı listeleri arka planda birleştirerek tek bir liste halinde sunan bu yapıya yeni bir listenin eklenmesi ya da liste elemanlarının farklı yapılardan oluşturulması durumlarında, Sistem tarafında herhangi bir değişikliğin yapılmasına gerek yoktur. Böylelikle Personel Listesi bakım yapılabilir bir tasarıma sahip olmuştur.

8. Sonuçlar ve Tartışmalar

Bu tez çalışmasında, yazılım dünyasında özellikle ürün geliştirme ve bakım maliyetlerini azaltmaya bir çözüm olarak kullanılan çerçeveler ve çerçevelerin tekrar kullanılabilirlik, bakım yapılabilirlik ve sağlamlık gibi özelliklere sahip olmasında tasarım kalıplarının nasıl yardımcı olabileceği incelenmiştir. Yazılım çerçevelerinin özelliklerini ve geliştirme sürecinde tasarım kalıplarının kullanımını somut bir şekilde anlatabilmek için üniversitelerin finans birimleri için geliştirilecek uygulamalara temel oluşturabilecek örnek bir çerçeve gerçeklenmiştir. Son olarak da çerçevenin kalite özellikleri incelenmiştir.

Çalışmada çerçeve temel olarak, finans alanına özel ve alan bağımsız yapılar olmak üzere ikiye ayrılmıştır. Finans alanına özel yapı Para ve Personel Modüllerinden oluşurken, alan bağımsız yapı Günlük Tutma, Doğrulama ve Yetkilendirme, Günlük Tutma ve Raporlama Modüllerinden oluşmaktadır. Tekrar kullanılabilirlik ve bakım yapılabilirlik özelliklerini en üst seviyeye çıkarabilmek için siyah-kutu ve beyaz-kutu çerçeve özelliklerini içinde barındıran gri-kutu çerçeve mimarisi temel alınmıştır. Gri-kutu yaklaşımının seçilmesinin en temel sebebi beyaz-kutu ve siyah-kutu çerçevelerin avantajlarını kullanırken beraberinde gelen dezavantajlarının da dengelenebilmesidir. Bu denge çerçevenin uzun ömürlü olmasındaki temel dayanak olacaktır çünkü üniversiteler bünyesinde geliştirilecek olan uygulamaların kanun ve yönetmelik değişiklikleri sebebiyle sık sık değiştirilmesi\güncellenmesi gerekmektedir.

Geliştirilen çerçevenin kalite özelliklerinin incelenmesi ile çerçevenin Açık-Kapalı Prensibine uygun olduğu görülmüştür. Sınıflar arası bağımlılığın az olması sebebiyle çerçeve üzerine yapılan eklentiler var olan yapıların güncellenmesini gerektirmeyecektir. Böylelikle çerçevenin ve çerçeve üzerine geliştirilmiş uygulamaların bakım maliyetleri azalacaktır. Ayrıca üniversitelerin finans birimleri için hazırlanacak uygulamaların temel özelliklerinin çerçeve kapsamında bulunması, uygulama geliştiricilerin sadece geliştirilecek uygulamanın iş katmanındaki yapılara yoğunlaşmasını sağlayacaktır. Bu durumun doğal sonucu olarak uygulama geliştirme maliyetleri de azalacaktır.

Bu çalışmanın uygulama kısmında gerçeklenen çerçevenin geliştirme aşamasında karşılaşılan en büyük problem uygulama geliştirme sürecindeki çerçeve kullanımı mantığının pratikte çok az kullanılmasıdır. Geliştirilecek çerçevelerin ekonomik karşılığının uzun vadede alınması sebebiyle ortaya çıkan bu durum çerçeve geliştirme işinin hep ötelenmesi ile sonuçlanmaktadır. Bir diğer problem de bir çok uygulamaya temel sağlaması planlanan çerçevenin sınırlarının belirlenmesi ve bu sınırlar dahilinde bir kaliteli tasarımın yapılmasıdır.

Kaliteli tasarımlar yapılırken yol gösterici olduğu bilinen tasarım kalıplarının varlığı bilinse de tasarım kalıplarının doğru kullanılmasındaki tecrübe gerekliliği yadsınamaz. Bu tecrübe ise zaman ve bilgi aktarımının sağlam bir şekilde yapılması ile kazanılacaktır. Geliştirilen çerçevenin bu problemlerin çözümüne yardımcı olması umulmaktadır.

Bu çalışmanın bahsedilen problemlerin aşılmasındaki çabalara destek ve üniversitelerin finans uygulamalarına temel oluşturacak ulusal bir çerçevenin geliştirilmesine ön ayak olabilecek olması bu konuda bir tez çalışmasının yapılmasındaki önemli motivasyonlardan biridir. Bu çalışmanın devamlılığının sağlanması için üniversiteler arası bir ekibin, üniversitelerde geliştirilecek uygulamaların temel yapılarını içinde barındıran bir çerçeve geliştirmek için, çalışması uygun olacaktır.

Yazılım geliştirme ve bakım maliyetlerini azaltabilmek için çerçeveler geliştirilmeli ve kullanılmalıdır. Ancak çerçeve geliştirme maliyetinin (zaman, para vb.) kısa sürede geri dönmemesi, yazılım geliştiricilerin çerçeve kullanımına kolay alışamaması ve çerçeve kullanma kültürünün yeterince yaygın olmaması nedeniyle çerçeveler ile ilgili çalışmalar için yeterli kaynak ayrılmamaktadır. Bu çalışmanın çerçeve geliştirilmesi ve kullanılması konusunda özendirici olması umulmaktadır.

Belirli bir alandaki uygulamalara temel sağlayacak, genişletilebilir ve bakımı yapılabilir mimariye sahip bir yapının oluşturulması zor bir süreçtir. Tasarım kalıpları kullanımı, bu süreç sonunda istenilen özelliklere sahip çerçevelerin oluşturulmasına yardımcı olacaktır. Ancak tasarım kalıpları kullanımını Türkçe olarak, gerçek bir uygulama üzerinde anlatan çalışmaların oldukça az olması, tasarım kalıplarının öğrenilmesini ve kullanımını yavaşlatmaktadır. Bu tezde anlatılanların, tasarım kalıplarının etkin bir şekilde kullanılmasına ve anlaşılmasına yardımcı olması çalışmanın bir diğer amacıdır.

KAYNAKLAR

Chen, X., (2004), Developing Application Frameworks in .NET, Apress, USA.

Chidamber, S. ve Kemerer C., (1994), "A Metrics Suite For Object Oriented Design", IEEE Trasactions on Software Engineering, 20:476-493.

Elish, M. O. ve Rine D., (2003), "Investigation of Metrics for Object-Oriented Design Logical Stability", Proceedings of the Seventh European Conference On Software Maintance And Reengineering, 103-200.

Fach, P. W., (2001), "Design Reuse through Frameworks and Patterns", Software, IEEE Volume 18(5):71 - 76

Gamma, E., Helm, R., Johnson R. ve Vlissides J., (1995), Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, Indianapolis.

Gurp, J. ve Bosch, J., (2001) "Design, implementation and evolution of object oriented frameworks: concepts and guidelines", Software-Practice and Experience Softw. Pract. Exper., 31:277–300.

Hayase, T., Ikeda, N. ve Matsumoto, K., (2001), "A Three-View Model for Developing Object-Oriented Frameworks", Technology of Object-Oriented Languages and Systems, 39th International Conference and Exhibition, 108–119.

Jung, H., Kim, D., Yang Y. ve Lee S., (2000), "A Design and Implementation of Object-Oriented Framework-based RAD Tool(INTRAD)", Systems, Man, and Cybernetics, 2000 IEEE International Conference, 3:2057 – 2061.

Krajnc, A. ve Hericko, M., (2003), "Classification of Object-Oriented Frameworks", EUROCON. Computer as a Tool. The IEEE Region 8, 2:57-61.

Mattsson, M., (1999), "Effort Distribution in a Six Year Industrial Application Framework Project", Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on 30 Aug.-3 Sept. 1999, 326 – 333.

Metsker, S. J., (2004), "Design Patterns in C#", Addison Wesley Professional Software Patterns Series, USA.

Morisio M., Romano D. ve Moiso C., (1999), "Framework Based Software Development: Invesigating the Learning Effect", Software Metrics Symposium, Proceedings.Sixth International, 260-268.

Prieto, F., Crespo, Y., Marques, J.M. ve Laguna, M.A., (2003), "Applying Formal Concepts Analysis to the Construction and Evolution of Domain Frameworks", Software Evolution, 2003. Proceedings.Sixth International Workshop on Principles of 1-2 Sept. 2003, 139 – 148.

Schmidt, D. C., (1995), "Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software", Communications of the ACM Special Issue on Object-Oriented Experiences, 38(10).

Shalloway, A. ve Trott, J. R., (2001), Design Patterns Explained a New Perpective on Object-Oriented Design, Addison Wesley Professional, USA.

Srinivasan, S., (1999), "Design Patterns in Object-Oriented Frameworks", Computer, 32(2):24-32.

Vladimir, L. ve Sylvia, I., (2005), "Towards Developmet and Use od In-house Component Framwork: Results and Expectations", Proceedings of the 2005 31st EUROMICRO

Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'05), 12-17.

Wendorff, P., (2001), "Assessment of Design Patterns During Software Reengineering: Lessons Learned from a Large Commercial Project", Software Maintenance and Reengineering, 2001. Fifth European Conference on 14-16 March 2001, 77 – 84.

Yuan, J., Xu, X. ve Gao Y., (2006), "An Expandable Network Management Framework for VOD", Proceedings of the 6th World Congress on Intelligent Control and Automation, June 21 – 23, Dalian.

Zhang, J., Qu, W., Du, L. ve Sun Y., (2003), "A Framework For Domain-Specific Search Engine: Design Pattern Perspective", Systems, Man and Cybernetics, 2003. IEEE International Conference, 4:3881 – 3886.

İnternet Kaynakları

[1] www.ndepend.com

ÖZGEÇMİŞ

Doğum tarihi	17.09.1981	
Doğum yeri	Aydın	
Lise	1995-1999	Aydın Lisesi Yabancı Dil Ağırlıklı Bölüm
Lisans	1999-2004	İstanbul Teknik Üni. Elektrik-Elektronik Fak. Bilgisayar Mühendisliği Bölümü
Yüksek Lisans	2004-Devam ediyor	Yıldız Teknik Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Yüksek Lisans Programı

Çalıştığı kurum(lar)

2001-2004	İTÜ Bilgi İşlem Merkezi, Yarı Zamanlı Asistan Öğr.
2004-Devam ediyor	İTÜ Bilgi İşlem Merkezi, Yazılım Geliştirme Uzm.