T.C
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

MOBİL PLATFORMLARDA KALİTELİ KOD GELİŞTİRİLMESİ VE MALİYETİN AZALTILMASI

Yüksek Lisans Tezi

Tezi Hazırlayan:

Nuri Gökhan ALP

İstanbul, 2019

T.C
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ BİLİM DALI

MOBİL PLATFORMLARDA KALİTELİ KOD GELİŞTİRİLMESİ VE MALİYETİN AZALTILMASI

Yüksek Lisans Tezi

Tezi Hazırlayan:

Nuri Gökhan ALP

Öğrenci No:

160820068

Danışman:

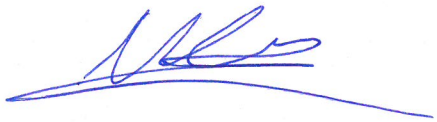
Dr. Öğr. Üyesi Atınç YILMAZ

İstanbul, 2019

YEMİN METNİ

Yüksek Lisans Tezi olarak sunduğum "Mobil Platformlarda Kaliteli Kod Geliştirilmesi ve Maliyetin Azaltılması" başlıklı bu çalışmanın, bilimsel ahlak ve geleneklere uygun şekilde tarafımdan yazıldığını, yararlandığım eserlerin tamamının kaynaklarda gösterildiğini ve çalışmamın içinde kullanıldıkları her yerde bunlara atıf yapıldığını belirtir ve bunu onurumla doğrularım. 13.06.2019

Nuri Gökhan ALP



T.C.
BEYKENT ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZ SAVUNMA SINAVI SONUÇ TUTANAĞI

Beykent Üniversitesi
Fen Bilimleri Enstitüsü Müdürlüğü'ne,

Aşağıda tez adı belirtilen yüksek lisans öğrencisi.160820068.no'lu ...Nuri Gencan A.P.....'in 13./06/2019 tarihinde yapılan tez savunma sınavı[1] sonucunda.45... dakika süreyle sunduğu ve savunduğu tezi hakkında[2] oybirliğiyle, başarılı... kararı verilmiştir.

Bilgilerinize saygılarımızla arz ederiz.

Anabilim Dalı : Bilgisayar Mühendisliği
Programı : Bilgisayar Mühendisliği
Tez Başlığı[3] : Mobil Platformlarda Kaliteli Kod Geliştirilmesi Ve Maliyetin Azaltılması

Tez Sınav Jürisi	Öğretim Üyesi	İmza
Danışman	: Dr. Öğr. Üyesi Atılay YILMAZ	
Üye	: Dr. Öğr. Üyesi Yücel Batu SALMAN	
Üye	: Dr. Öğr. Üyesi Ediz ŞAYKOL	

---

[1] Jüri üyeleri, söz konusu tezin kendilerine teslim edildiği tarihten itibaren en geç bir ay içinde toplanarak öğrenciyi tez sınavına alır. Tez savunma sınav süresi en az 45, en çok 90 dakikadır. Jüri üyeleri, sınav öncesi yapılacak toplantıda, kendi aralarından danışman dışında bir üyeyi başkan seçer. Tez sınavı, tez çalışmasının sunulması ve bunu izleyen soru-cevap bölümünden oluşur. Tez sınavı, öğretim elemanları, lisansüstü öğrenciler ve alanın uzmanlarından oluşan dinleyicilerin katılımına açık ortamlarda gerçekleştirilir. Belirlenen günde yapılamayan jüri toplantısı, katılanların hazırladığı bir tutanakla enstitü yönetimine bildirilir. Bu durumda, jüri en geç on beş gün içinde toplanarak adayı tez savunma sınavına alır. (05 Ağustos 2017 tarihli 30145 sayılı Resmi Gazetede Yayınlanan Değişiklik-Madde 29-3)

[2] Tez sınavının tamamlanmasından sonra jüri, tez hakkında salt çoğunlukla "kabul", "düzeltme" veya "ret" kararı verir. Jüri başkanı, jüri üyelerince imzalanmış karar tutanağını, tez sınavını izleyen üç gün içinde ilgili enstitü yönetimine teslim eder. Tezi hakkında düzeltme kararı verilen öğrenci en geç üç ay içinde gerekli düzeltmeleri yaparak ve birinci fıkradaki usule göre tezini aynı jüri önünde yeniden savunur. Süresi içerisinde "düzeltme" savunmasına girmeyen öğrencinin enstitü ile ilişiği kesilir. (Beykent Üniversitesi Lisansüstü Eğitim ve Öğretim Yönetmeliği-Madde 29-4)

[3] İleride doğabilecek aksaklıkların engellenmesi için tezin başlığının yazılması gerekmektedir.

Adı ve Soyadı : Nuri Gökhan ALP
Danışmanı : Dr. Öğr. Atınç YILMAZ
Türü ve Tarihi: : Yüksek Lisans, 2019
Alanı : Bilgisayar Mühendisliği
Anahtar Kelimeler : Mobil Platformlar, Kod Kalitesi, Yazılım Projesinde Maliyet, Yazılım Tasarım Örüntüleri

ÖZ

MOBİL PLATFORMLARDA KALİTELİ KOD GELİŞTİRİLMESİ VE MALİYETİN AZALTILMASI

Günümüzde mobil uygulama kullanımının daha fazla yaygınlaşmasıyla profesyonel yazılım ekiplerinde mobil yazılımcıların önemi artmıştır. Mobil platformda uygulama geliştirme hem arka plandaki iş kodları hem de ön yüz geliştirmelerini kapsar. Ön yüz geliştirme yapılırken planlama, geliştirme ve sonuç aşamalarında pek çok yoruma maruz kalınmaktadır. Dolayısıyla bir proje hayat bulana kadar ve hatta bulduktan sonraki sürümlerde dahil yorumlar neticesinde pek çok önemli değişiklikler yapılmaktadır. Bu değişimlere rahat ayak uydurabilmek için yazılımın kaliteli, kolay anlaşılır ve yeterli esneklikte olması gerekmektedir. Bu yetkinlikleri sağlamayan yazılımlar haliyle yükselen maliyetlerle karşı karşıya kalmaktadır.

Mobil yazılımın maliyetini arttıran faktörler ve maliyeti düşürmek için uygulanabilecek çözümler araştırılmıştır. İlk sürüm geliştirme, bakım ve devam eden sürümlerindeki geliştirmelerindeki durumları projenin büyüklüğü ve koşulları da göz önünde bulundurarak incelenmiştir. Tüm etken faktörler incelenerek problemler tespit edilmiş ve çözüm önerilerinde bulunulmuştur. Tez çalışması ile tasarım örüntüleri ve kaliteyi arttıran faktörler ile alakalı konular detaylandırılmıştır. Bu tez ile literatüre bir tanesi “Flex Pattern” olan üç yeni örüntü sunulmuştur.

Name and Surname : Nuri Gökhan ALP
Supervisor : Dr. Lecturer Atınç YILMAZ
Degree and Date : Master, 2019
Major : Computer Engineering
Key Words : Mobile Platforms, Code Quality, Software Project Costs, Software Design Patterns

ABSTRACT

DEVOLOPING HIGH QUALITY CODE AND REDUCING COST IN MOBILE PLATFORMS

With the increasing usage of mobile application, the importance of mobile software developers has increased in software teams. Application development on the mobile platform includes both background business code development and fronend development. Frontend development is most commented part of development during the plan, in development and final period. This is because, many important changes are made as a result of comments. In order to keep up with these changes, the software must be of high quality, easily understood and sufficient flexibility.

In this study, the factors that increase the cost of mobile software and the solutions that can be applied to reduce the cost have been investigated. The development of the first version, maintenance and next version are examined, taking into account the size and conditions of the project. All the factors were investigated and problems were determined and solution suggestions were made. With this thesis study, design patterns and factors that improve the quality were elaborated. In this thesis, three new patterns, one of which is “Flex Pattern”, are presented in the literature.

İÇİNDEKİLER

Sayfa No.

TABLOLAR LİSTESİ

Sayfa No.

ŞEKİLLER LİSTESİ

Sayfa No.

KISALTMALAR

DRY	: Don’t Repeat Yourself
JSON	: JavaScript Object Notation
KISS	: Keep It Short, Simple
MVC	: Model View Controller
MVP	: Model View Presenter
MVP-VM	: Model View Presenter View-Model
MVVM	: Model View View-Model
OOP	: Object Oriented Programming
SDK	: Software Development Kit
SOC	: Separation Of Concern
SOLID	: Single-Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle
TDD	: Test Driven Development
UI	: User Interface
UUID	: Universally Unique Identifier
UX	: User Experience
VIPER	: View Interactor Presenter Entity Router
WPF	: Windows Presentation Foundation
XIB	: XML Interface Builder
XP	: Extreme Programming
YAGNI	: You Aren't Gonna Need It

1. GİRİŞ

Gelişen teknoloji neticesinde mobil platformların önemi günden güne artmaktadır. Hatta pek çok üründe en çok kullanılan kanal mobil olmaya başlamıştır. Günümüzde bankalar, e-ticaret siteleri, sosyal platformlar, oyunlar, sigorta şirketleri, hastaneler ve bunun gibi pek çok farklı iş kolu veya kişi veya organizasyonlar mobil platformlarla ilgilenmektedir. Mobil uygulama kullanımının artmasıyla yazılım ekiplerindeki mobil yazılımcı ihtiyacı ve bu platforma verilen önem artmıştır.

Mobil platformların yayınlaşmasıyla uygulamaların özellikleri ve kapsamı daha da genişlemiştir. Artık pek çok mobil ürün büyük ölçekli olarak planlanmakta ve mobil ürünlere büyük yatırımlar yapılmaktadır. Projelerin büyümesi ile sorunlar da büyümektedir. En büyük sorunlar genellikle yeni geliştirmeler yapıldığı zaman veya hataların çıkması neticesinde ürünle ilgili ortaya çıkan hataların çözülmesi sırasında yaşanır. Eğer bir yazılım projesi kaliteli ve değiştirilmeye uygun bir şekilde geliştirilmediyse projeye bakım yapmak ve ek geliştirme yapmak daha da zorlaşacak ve maliyetler daha da artacaktır.

Günümüzde bir yazılım projesinin maruz kaldığı önemli problemleri giderebilmek için tasarım örüntüleri, prensipler ve çeşitli diğer çalışmalar yapılmıştır. Yazılımı geliştirmek için her yazılımcının bildiği belirli bir şekilde kod yazılması tasarım örüntüleri ile sağlanır. Örüntüler çeşidine göre tüm yazılımı veya bir kısmını ilgilendirebilirler. Belirli bir örüntü kullanarak kod yazımı birden fazla yazılımcının birlikte çalıştığı projelerde kodun herkesçe anlaşılabilmesi için önemlidir. Ancak proje koşul ve ihtiyaçlarına göre uygun örüntünün tercih edilmemesi durumunda büyük problemler ortaya çıkmaktadır.

Her yazılım projesinde olduğu gibi mobil yazılım projesinde de planlama çok önemlidir. Geliştirilecek ürünün ihtiyaçlarına ve koşullarına göre plan yapmak projenin geliştirildiği her dönemine etki edecek önemli bir unsurdur. En başından doğru kararların alınması maliyete ve kaliteye doğrudan doğruya etki edecektir. Hangi tasarım örüntüsünün kullanılacağı, mobil uygulamanın servis ile iletişiminin nasıl olacağı, mimari özelliklerinin ne olacağı ve yazılım geliştirme esnasında hangi yöntemlerin kullanılacağı çok önemlidir. İyi planlanmış bir proje hata çözümleri ile de

birlikte ya daha kısa sürede tamamlanmasına ya da uzun vadede geliştirmesinin ve bakımının yapılması için harcanacak sürenin daha da kısalmasını sağlayacaktır. Kötü planlanmış bir proje ise aksine daha fazla zaman harcanmasına ve daha fazla maliyete sebep olacaktır.

Mobil platform yazılım geliştirirken yazılımcılar hem arayüz hem de arka planda iş kodları yazmaktadır. Mobil uygulamalar bir kullanıcının doğrudan kullandığı son kaynak olması sebebiyle uygulama için çok fazla yorum yapılabilmekte ve akabinde de birçok karar değişiklikleri olmaktadır. Dolayısıyla mobil yazılım için değişime açık olmak ve değişiklikler için yeterli esnek yapıya sahip olmak çok önemlidir. Tezin ilerleyen bölümlerinde mobil uygulama projelerine esnekliği kazandırabilmek için çalışmalar yapılacak ve paylaşılacaktır.

Bu tez mobil platformlarda kod kalitesini arttırmayı ve maliyeti azaltmayı amaçlar. Kalitesiz kod yazımı, yanlış tasarım örüntüsü tercih edilmesi, yanlış planlama, öngörüsüz çalışma, değişime uygun esneklikte olmama, kod yoğunluğu ve karmaşıklık maliyeti arttıran asıl sebeplerdir. Tezde bu problemler üzerine odaklanılarak çözüm önerileri sunulacaktır.

Özetle proje planlamasındaki sorunlar, kod kalitesinin bahsedilen birtakım sebeplerden ötürü düşmesinden dolayı ortaya çıkan sorunlar ve projenin değişikliğe açık olamaması tezde çözümü araştırılacak problemlerdir.

Tezin "Kod Kalitesi Arttırma" bölümünde mobil yazılım projesinin kalitesinin nasıl arttırılabileceğine dair günümüzdeki bilinen paylaşılmış çalışmalara değinilecek ve tez çalışması ile önerilen yöntemler paylaşılacaktır. Tez çalışmasının önerdiği bu yöntemler tezin bazı bölümlerinde uygulanacaktır.

Tezin "Tasarım Örüntüleri" bölümünde günümüzdeki pek çok tasarım örüntüsü incelenecek ve bu örüntülerin kıyaslamaları yapılacaktır. Proje koşullarına, tipine ve büyüklüğüne göre hangi örüntülerin tercih edilmesi gerekildiğine değinilecek bu kapsamda konu detaylandırılacaktır. Örüntülerin olumlu ve olumsuz kısımları araştırılacak, örüntülerdeki olumsuz kısımları giderici bazı öneriler sunulacaktır.

Çalışma neticesinde örüntülerdeki veya bir yazılım projesindeki eksiklikleri ve sorunları gidermek amacıyla Flex Pattern ve Utils Based Pattern gibi tez çalışması neticesinde keşfedilecek yeni örüntüler ile ilgili çalışmalar detaylandırılacaktır.

Çalışma IOS platformu kullanılarak yapılacaktır. Ancak tezin odağı tek platforma katkı sağlamak olmadığından tezden elde edilen bilgilerin diğer mobil platformlarda kullanılması mümkün olacaktır.

2. LİTERATÜR TARAMASI

Bu tezde yayınlanmış makaleler, tezler, teknik raporlar, web sayfaları, öğreticiler, açık kaynak kodlar ve tartışma platformlarından elde edilen bilgilerle yazılım projesindeki kalitenin arttırılması ve maliyetin düşürülmesine yönelik çalışmalar yapılmıştır. Genel konuların bazı yerlerde kapsamı daraltılarak mobil platformlara odaklı çalışmalar yapılmıştır.

Günümüzde yazılım geliştirme ile kaynaklar genellikle internet ortamında açık kaynak sitelerde veya yazılım geliştiricilerim topluluklarını barındıran web siteleri üzerinde paylaşılmaktadır. Akademik olarak yapılmış çalışmalar ya bu kaynaklardan esinlenerek yapılmış ya da bir zaman sonra başkaları tarafından veya bazen kişilerin ya da toplulukların kendileri tarafından akademik ortamlara tez, makale, konferans veya kitap olarak eklenmişlerdir. Yaygınca biliniyor olsa da her bilgi doğrudan akademik ortamlarda değildir. Yazılım dünyasında yapılan çalışmalar genellikle ilk olarak web ortamında profesyonellere sunulur ve tartışılır ancak akademik olarak çalışma yapılma zorunluluğu yoktur.

Tez çalışmasında IOS platformuna odaklı örnekler sunduğundan, IOS platformunun kalite standartlarına Apple Developer Documentation uygun davranılmıştır [1]. Medium [2], Github [3], MSDN [4], Apple Developer Documentation [1] ve çeşitli yazılımlarla alakalı blog siteleri gibi kaynaklarından faydalanarak tez çalışması yapılmıştır.

Günümüzde literatüre geçmiş pek çok tasarım örüntüsü mevcuttur. Pek çok platformda olduğu gibi mobil platformlarda da MVC, MVP, MVVM, MVP-VM ve VIPER gibi mimari örüntüler kullanılmaktadır. Mimari tasarım desenleri ile ilgili çalışmaların bazılarının akademik, kitap, konferans, makale vb. çalışmaları da mevcuttur. Örneğin MVC için [5], MVP için [6], MVVM için [7] gibi çalışmalar mevuttur. Mimari tasarım desenlerini kıyaslayan pek çok çalışma da mevcuttur. Bunlardan bazıları; Native Android uygulaması MVC, MVP ve MVVM mimarisi kıyaslaması [8], MVC ve MVP kıyaslaması [9], tasarım deseni kullanılarak geliştiren yazılım ile kullanılmadan geliştirilen yazılımın performans analizi [10]. Literatürde mimari tasarım örüntülerinin kapsamlı kıyaslamaları bulunmaktadır. Diğer

çalışmalardan farklı olarak bu tezde mimari örüntüler mobil platformlara göre incelenmiş, kıyaslanmış, olumlu ve olumsuz yönleri incelenmiştir.

Kod kalitesini arttırmak için web sitelerinde ve literatürde yapılmış pek çok çalışma mevcuttur. Kaliteyi arttırmak için çeşitli yöntem ve prensipler belirlenmiştir. Tez de yer alan TDD ile ilgili bölüm ile ilgili daha önce Cumhur Ünlü'nün yapmış olduğu çalışmada TDD'nin yazılım üretkenliğine ve niteliğine etkileri araştırılmıştır. [11]. Yazılım kalitesini arttırmak için çeşitli yöntemler, prensipler ve örüntüler ile ilgili çok önemli bir kaynakta Rober Cecil Martinin Agile Software Development kitabıdır [12].

Aydınöz çalışmasında tasarım örüntülerinin nesne tabanlı metrikler ve hata eğilimi üzerine etkilerini incelemiştir [13]. Akalın yazılımın evrimleşme sürecinde tasarım örüntülerinin yazılım kalitesi üzerindeki etkilerinin incelemiştir [14].

Tez çalışması için kullanılan kaynaklar genellikle web ortamında akademik dokümanların haricindeki kaynaklarda detaylı bir şekilde bulunmaktadır. Literatürdeki çalışmalar genelleyici veya konu özelinde detaylandırılmış çalışmalardır. Bu tez diğer çalışmalardan farklı olarak doğrudan mobil platformdaki kalite ve maliyet problemlerini çözmeye odaklanır. Çözüm için yenilikçi çözümler sunar, mevcutta bilinen bazı çalışmaların kullanılması için referans gösterir, bilinen çalışmaların olumsuzluklarını giderici yöntem önerileri sunar.

3. KOD KALİTESİNİ ARTTIRMA

Kaliteli kod geliştirmesi projenin maliyetini doğrudan doğruya etkileyen faktördür. Kimi zaman kaliteli kod yazmak ilk geliştirme yapıldığı zaman maliyeti arttırabiliyor olsa da uzun vadede bir projenin bakım yapılabilirliğini arttırması ve kolayca değiştirilebilmesini sağlaması dolayısıyla maliyete önemli katkısı vardır.

Her ne kadar pek çok firma kaliteli iş çıkartmak istese de özellikle ülkemizde firmalar genellikle yapılan işin kalitesine değil sonucuna bakmaktadır. Bu bakış açısı yüzünden projenin ilk sürümlerinde hızlıca işi bitirebilmelerinden memnuniyet duysalar da sıra projede bakım yapmaya veya ilerleyen fazlarda değişiklik veya yenilik yapmaya geldiği zaman büyük problemler yaşanmakta, projenin maliyetleri çok ciddi oranda artabilmektedir. Kalitesiz iş yapılması neticesinde proje bakımdayken veya yenilik yapılacağı zaman değişiklikleri yapmak daha fazla zaman almakta ve harcanan zamana rağmen halen yeterli kaliteyi sağlayamamaktadır. Bunun asıl sebebi kalitesiz iş sonucu kodun esnekliğini, okunabilirliğini, değişime adapte olabilme yeteneğini ve tekrar kullanılabilirliğini kaybetmesidir.

Günümüzde kod kalitesini arttırmak için çeşitli prensipler, tasarım örüntüleri çalışmaları, araştırmalar ve elde edilmiş tecrübe üzerinden yapılan birtakım çalışmalar mevcuttur. Bu bölümde mevcut çalışmalardan bahsedilmekle birlikte tez çalışması ile tavsiye edilen önerilere de değinilecektir.

3.1. Kod Kalitesini Arttırmak İçin Bilinen Bazı Çalışmalar

Kod kalitesini arttırmak için çeşitli çalışmalar yapılmış ve yazılım topluluğuna paylaşılmıştır. Günümüzde bu konuda halen çalışılmaya devam edilmektedir. Pek çok mevcut çalışma artık olgunluğunu kazanmış ve birer standart haline gelmiştir. Bu bölümde bu konuda yapılmış çalışmalar incelenmiştir.

3.1.1. SOLID Prensipleri

SOLID prensibi Robert Martin tarafından Nesneye Yönelik Programlamanın ilk 5 maddesi olarak sunulmuştur. Yazılan kodun gelecekte istenebilecek değişiklik ya da yeniliklere karşı en az değişiklik ve zaman kaybıyla geliştirilmesi ve yeni bir özellik istenildiğinde minimum değişiklikle geliştirilebilmesi amaçlanmıştır [15].

SOLID ismi aslında bir kısaltmadır. Sırayla Tek Sorumluluk Prensibi (Single-Responsibility Principle), Açık Kapalı Prensibi (Open-Closed Principle), Liskov'un Yerine Geçme Prensibi (Liskov Substitution Principle), Arayüz Ayrımı Prensibi (Interface Segregation Principle) ve Bağımlılıkların Terslenmesi Prensibi (Dependency Inversion Principle) maddelerinden oluşur [15].

Tek Sorumluluk Prensibi ile her bir metoda tek bir görev ve sorumluluk verilir. Başka sınıfların işlerini yapmaz ve sadece kendi işinden sorumludur. Böylelikle tüm gereksinimler parçalara ayrılıp bağımsız bir şekilde çalışması hedeflenir. Böylece metot karmaşıklaşmaz ve yapılacak iş belli olduğundan başka faktörleri gözden kaçırma riski ortadan kalkar.

Açık Kapalı Prensibi ile nesneler geliştirmeye açık ama değişime kapalı olmaları anlamına gelir. Oluşturulan nesneler zaman içerisinde ek özellikler kazanabilirler ama temel nesne değişime kapalı tutulmalıdır [16]. Liskov'un Yerine Geçme Prensibi ile türeyen sınıfların kalıtım alınan sınıfın tüm özelliklerinin kullanılmasını şart koşar. Dolayısıyla türeyen sınıflar kalıtım alınan sınıfın kodlarını ezmemelidir. Bu konuyla alakalı önlem için tezin ilerleyen bölümlerindeki "4.3.1. Imitate Pattern" bölümü incelenebilir.

Arayüz Ayrımı Prensibi ile bir arayüze (interface) gerekli olmayan parçaların eklenmemesi sadece gerekli olan ve kullanılacak parçaların ekli olması gerektiği savunulur [15].

Bağımlılıkların Terslenmesi Prensibi ile üst sınıfların ve bu sınıfa ait metotların alt seviye sınıf ve bu sınıflara ait metotlara bağımlı olmaması şartı koşulmuştur.

3.1.2. YAGNI Prensibi

YAGNI prensibinin İngilizce açılımı "You Aren't Gonna Need It" Türkçe karşılığı ise ihtiyacın yoksa geliştirme anlamına gelir. Yazılımcıların ya ihtiyaç olursa düşüncesiyle geliştirme yapmalarına karşı olan ve "ihtiyaç yoksa geliştirme yapma" denilen bir prensiptir [17].

YAGNI prensibi arkasında "Do the Simplest Thing That Could Possibly Work" yani Türkçe karşılığıyla "Muhtemelen Çalışabilecek En Basit Şeyleri Yapın" XP pratiği olan bir prensiptir. Prensip bir yazılım projesine bazı ekstra özellikler katılmak istenildiğinde bu özellikleri eklemek yerine mevcutta bekleyen işlere odaklanılmasını savunur. Eklenip eklenmeyeceği belli olmayan, ileride değişime uğrayabilecek veya doğrulanamayacak işler üzerinde vakit kaybetmemeyi tavsiye eder [18].

3.1.3. KISS Prensibi

KISS prensibinin İngilizce açılımlarından biri "Keep It Short, Simple" olarak ifade edilir Türkçe karşılığı ise "Kısa ve basit tutun" gibi bir anlam ifade etmektedir. Kodu basitleştirmek için çabalamayı öneren bir prensiptir. Bir problemi çözmek için en yalın ve basit çözüm üzerinden giderilmesini önerir. Karmaşık çözümlerin zekice olduğu fikrini reddeder. Zor olanın bir işi basitleştirmek olduğu savunulur. Çözümler basitleştirilerek çözümün uygulanması kolaylaşır, hata riski azalır, bakım yapılması daha kolaylaşır dolayısıyla bakıma daha az zaman harcanması sağlanır ve basit çözümler daha kolay değiştirilebilirler [19].

3.1.3. DRY Prensibi

DRY prensibinin İngilizce uzun hali "Don't Repeat Yourself" Türkçe karşılığı ise "Kendini Tekrar Etme" anlamına gelmektedir. Prensip her bilgi parçasının bir sistem içinde tek, net ve yetkili bir temsiline sahip olması gerektiğini belirtir [20]. Kodları tek bir yerde yazıp buradan çalışmasını ve değiştirilmesi temeline dayanır. Özellikle büyük projelerde, çok fazla yazılımcının olduğu ve çok değiştirilen projelerde çok kullanışlı bir prensiptir [18].

3.1.4. SOC Prensibi

SOC uzun açılımı ile “Separation of Concern” Türkçe karşılığı “İşin Ayrılması” gibi bir manaya denk gelmektedir. Yazılımdaki parçaların sorumluluklarıyla birlikte kendilerine has olmalarını ve başka elemanlarla paylaşılmamasını söyler. Yazılım parçasında kendi içerisinde sorumluluğunu arttırır ve başka parçalara bağımlılığını yok eder. Böylelikle bir parça başka parçayla birlikte rahatça tekrar kullanılabilecektir. Bu sayede yazılımın genişletilebilir ve esnek olması sağlanır [21].

3.1.5. Fluent Interface

Fluent Interface Türkçe karşılığı ile “Akıcı Arayüz” kod karmaşasının önüne geçen kodu daha da basitleştiren bir yöntemdir. Daha okunaklı kodların yazılmasını, parçaya özgü kod oluşturulmasını ve kullanımı kolay kod yazılabilmesini sağlar [22]. Bu yapı metotların sınıfın kendisini dönmesi ile sağlanır. Böylelikle kod tekrardan nesne çağrılmaya gerek olmadan devamlılığını sürdürür. Bu sayede kolay okunur ve hızlı yazılır kod yazma imkanına kavuşulur.

Örneğin Ali isimli bir çocuğa bakkala gitme görevi verilsin. Bakkala gidebilmesi için yol tarifi yapılsın ve bakkala girmesi söylensin. Bunun koddaki karşılıkları aşağıdaki kod örneklerinde verilmiştir. Şekil 1’deki örnek klasik yani alışıldık yöntemle kodlanmıştır. Şekil 2’deki örnekte ise Fluent Interface yapısı ile kodlanmıştır. İkinci örneğin ne kadar daha basit olduğu gözlemlenebilir. Şekil 3 ve Şekil 4’te Şekil 1 ve Şekil 2 örneklerinin kullandıkları sınıf ve içerisindeki metotların nasıl yazıldıklarına dair örnekler paylaşılmıştır.

```
func alisildikYontem() {
    let ali = AliAlisildik()
    ali.sagaDon()
    ali.solaDon()
    ali.solaDon()
    ali.duzGit()
    ali.bakkalaGir()
}
```

Şekil 1. Klasik yöntem ile kodlama

```
func fluentInterfaceliYontem() {
    _ = AliFluentInterface().sagaDon().solaDon().solaDon().duzGit().bakkalaGir()
}
```

Şekil 2. Fluent Interface yöntemi ile kodlama

```
class AliAlisildik {
    func sagaDon() {
        print("sağa dön")
    }

    func solaDon() {
        print("sola dön")
    }

    func duzGit() {
        print("düz git")
    }

    func bakkalaGir() {
        print("bakkala gir")
    }
}
```

Şekil 3. Klasik yöntemle kodlama örneğinde çağrılan metotlar

```
class AliFluentInterface  {
    func sagaDon() -> Self {
        print("sağa dön")
        return self
    }

    func solaDon() -> Self {
        print("sola dön")
        return self
    }

    func duzGit() -> Self {
        print("düz git")
        return self
    }

    func bakkalaGir() -> Self {
        print("bakkala gir")
        return self
    }
}
```

Şekil 4. Fluent Interface yöntemi ile kodlama örneğinde çağrılan metotlar

3.1.6. Test Driven Development

Test Driven Development kısa ismi ile TDD, Türkçe karşılığı ile "Test Güdümlü Geliştirme" XP'nin bir parçası olarak Kent Back tarafından bulunmuş olan bir programlama tekniğidir. Beck, 1994 yılında Smalktalk için SUnit test kütüphanesini yazmış 1998 yılında XP'de "testleri çoğunlukla ilk sırada yazmalıyız" diye ifade ettiği Test First kavramını ortaya atmış 2002 yılında çıkardığı kitabında [23] Test Driven Development olarak daha da detaylı bir teknik olarak ele almıştır [24].

TDD'de test önceden yazılır sonrasında kodlama yapılır. Testi yazabilmek için planlı bir şekilde projeyi devam ettirmek gereklidir. İdeal yapıda ilk etapta ihtiyaçlar ve tasarım gözden geçirilir. Sonrasında ihtiyaçlara göre testler yazılır. Testler oluşturulduktan sonra kodların yazımına başlanır. Eğer kod testleri geçemiyorsa kodlar gözden geçirilir ve kodda gerekli kod düzenlemeleri yapılır. Kod testlerden

geçene kadar kodlar tekrar düzenlenir. Kod testten geçtikten sonra yeni testler ve kodlar yazılır [25].

TDD ile kod yazımı her ne kadar zorlaşıyor ve geliştirme aşamasında harcanan zamanda bir o kadar artıyor olsa da proje sonunda hatalı ve sorunlu bir ürün ortaya çıkmasını bir o kadar engellemektedir. TDD sayesinde projelerde hatalar çok ciddi azalmakta ve bir projede hata çözümü için harcanan zamanı çok ciddi azaltmaktadır.

3.2. Kod Kalitesini Arttırmak İçin Tezde Önerilen Yöntemler

Bu bölümde kod kalitesini arttırmak için tez çalışması neticesinde ortaya çıkarılan yöntemler anlatılmıştır.

3.2.1. Etkin Struct Kullanımı

Sturct yapısı değer türündendir doğrudan yığın üzerinde tutulurlar. Böylelikle doğrudan erişilebilirler. Sınıfa kıyas daha az maliyetli ve erişim hızları yüksektir [26]. Yapı ile birbiriyle ilişkili değişkenler veya yapılar (struct) iç içe tutulabilir.

Bu yapı sayesinde kullanılacak nesneler bir kategori altında rahatça toplanabilir. Bu yapı etkili kullanılması halinde önemli avantajlar yakalanmış olunur. Bir mobil proje ele alındığında, proje içerisinde genellikle resimlerin string ismi ile çekildikleri, dil kullanımı için yerelleştirme (localization) işlemlerinde de string'lerin olduğu görülür.

Bu yapının en büyük katkısı yerelleştirme, resim yükleme, bir anahtar (key) ile nesne çağırma veya observable bir yapı çağırma gibi string kullanılarak çağırma işlemlerinde hata yapılma riskinin giderilmesidir. Bu sayede string yazım hatalarından dolayı çağrımın başarısız olma ihtimali ortadan kaldırılmış olur. Öte yandan kullanıcı çok rahat bir şekilde ne yazması gerektiğini anlayabilir. Böylelikle kullanım kolaylığı da sağlanmış olur. Bu tip çağrım nesneleri tek yerde anlamlı şekilde toplandığından neyin nerede kullanılabildiği rahat tespit edilebilir, tekrarların önüne geçilebilir, değişiklik çok kolay ve hızlıca yapılabilir.

```
struct Localization {
 struct EffectiveStructUsagePage {
   static let title = Utils.localize("effectiveStructUsagePage.title")

   struct TopField {
       static let firstLabelText =
           Utils.localize("effectiveStructUsagePage.topField.firstLabelText")
       static let secondLabelText =
           Utils.localize("effectiveStructUsagePage.topField.secondLabelText")
   }

   struct BottomField {
       static let buttonText = Utils.localize("effectiveStructUsagePage.bottomField.buttonText")
       static let labelText = Utils.localize("effectiveStructUsagePage.bottomField.labelText")
   }
 }
}
```

Şekil 5. Etkin struct kullanımı örneğinde Localization için struct yapısı örnek kodu

```
struct Images {
    static let macbook = UIImage(named: "macbook")
    static let github = UIImage(named: "github")
}
```

Şekil 6. Etkin struct kullanımı örneğinde Images için struct yapısı örnek kodu

Şekil 5'te Localization ve Şekil 6'da Images için struct örnekleri paylaşılmıştır. Şekil 5'teki örnekte bir Localization yani yerelleştirme işi için struct yazılmıştır. Bu yapının görevi dil çevirilerini daha düzenli ve kolay bir şekilde çevrilmesini sağlamaktır. İçinde EffectiveStructUsagePage isimli örnek olarak oluşturulan sayfayı temsil eden bir struct mevcuttur. Sayfa başlığı title, üst kısımda kalan görüntü TopField ve alt kısımda kalan alt görüntü ise BottomField olarak tanımlanmıştır. Örneğe göre üst ve alt kısımda kalan görüntülerin (view) içinde tuş ve label objeleri vardır, bunlar içinde struct yapısı kod örneğindeki gibi ilgili alanlarda oluşturulmuştur. Yapı içindeki değişkenler string tipte değişkenlerdir. Utils.localize olarak tanımlanan statik sınıf ve metodu çalıştığında istenilen dile göre metnin çevirisini yapıp string olarak döner. Bahsedilen dil çevirisini yapan metoda ait kod Şekil 7'de paylaşılmıştır. Şekil 6'daki struct örneğinde uygulamada kullanılan resimlerin Images yapısı ile elde edilebilmesi sağlanmıştır.

```
static func localize(_ key: String) -> String {
    return NSLocalizedString(key, comment: "")
}
```

Şekil 7. Etkin struct kullanımı örneğinde dil çevirisi yapan Utils sınıfının localize metoduna ait kod

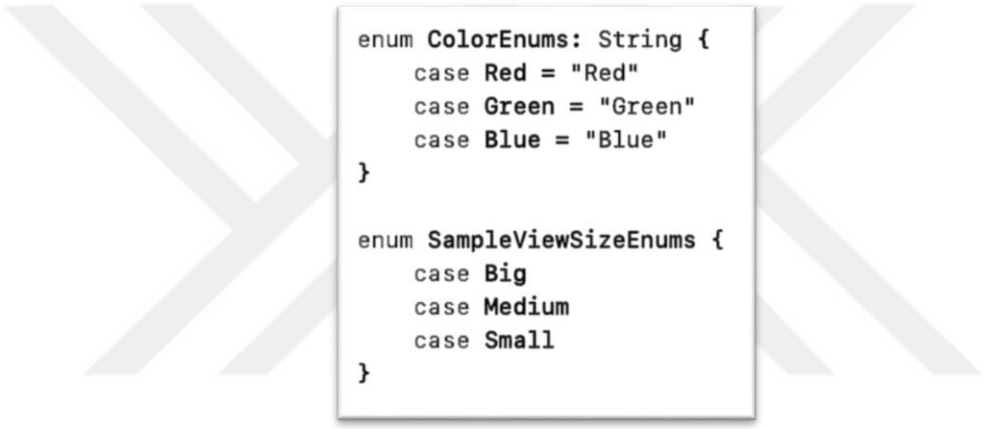
Şekil 8'deki kod örneğinde örneğin sayfasının kodlar paylaşılmıştır. Localization ve Images struct nesneleri kullanılarak ilgili yerlere değerlerin çok basit şekilde atandığı görülebilmektedir.

```
class EffectiveStructUsageViewController: UIViewController {

    @IBOutlet weak var topView: UIView!
    @IBOutlet weak var topViewFirstLabel: UILabel!
    @IBOutlet weak var topViewSecondLabel: UILabel!
    @IBOutlet weak var bottomView: UIView!
    @IBOutlet weak var bottomViewButton: UIButton!
    @IBOutlet weak var bottomViewLabel: UILabel!
    @IBOutlet weak var imageView: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.title = Localization.EffectiveStructUsagePage.title

        self.topViewFirstLabel.text =
            Localization.EffectiveStructUsagePage.TopField.firstLabelText
        self.topViewSecondLabel.text =
            "\(Localization.EffectiveStructUsagePage.TopField.secondLabelText)\nAppid:
            \(Constants.appid)"

            self.bottomViewButton.setTitle(Localization.EffectiveStructUsagePage
            .BottomField.buttonText, for: .normal)
        self.bottomViewLabel.text =
            Localization.EffectiveStructUsagePage.BottomField.labelText

        self.imageView.image = Images.macbook
    }

    @IBAction func buttonTapped(_ sender: Any) {
         self.imageView.image = Images.github
    }
}
```

Şekil 8. Etkin struct kullanımı örneğinde örnek sayfanın sınıfına ait kod

3.2.2. Etkin Enum Kullanımı

Bir projeyi daha anlaşılabilir kılmak için enum kullanımı da bir hayli önemlidir. Enum gönderilmiş bir bilginin neye karşılık geldiğini çok net bir şekilde gösteren etkili bir yapıdır. Enum kodu daha okunabilir kılar. Koddaki değerlerin kullanılması esnasında hatalı kullanılmasını engeller. Değerlerin ne olduğunu anlamak için yazılması gerekecek kod yazma ihtiyacını önler bu sayede gelen değerin doğrudan ne olduğu bilinebilir. Örneğin servisten 2 olarak integer değer şeklinde günün sayısı dönüldüğünde, günün Salı olduğu bilinmek isteniyorsa bu bilgi enum sayesinde çok rahat bir şekilde sağlanabilir.



```
enum ColorEnums: String {
    case Red = "Red"
    case Green = "Green"
    case Blue = "Blue"
}

enum SampleViewSizeEnums {
    case Big
    case Medium
    case Small
}
```

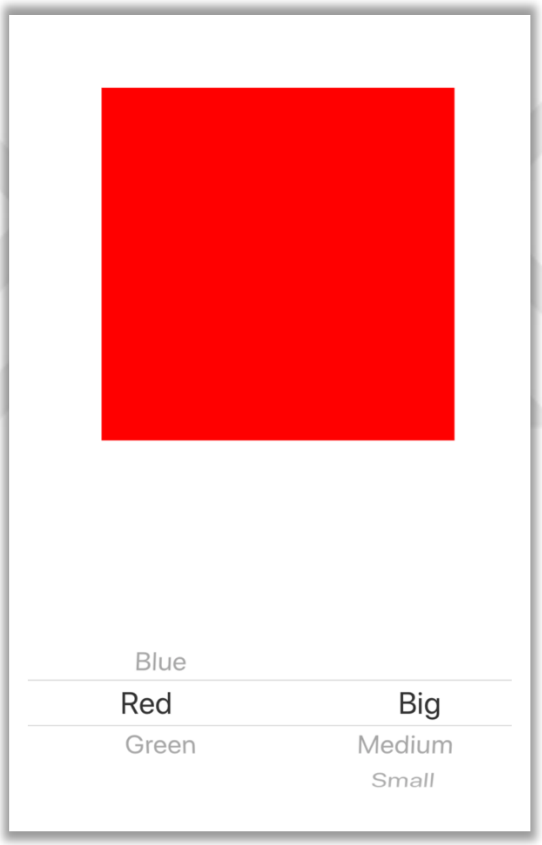
Şekil 9. Etkin enum kullanımı konusu için oluşturulan enum örneklerini

```
class EnumSampleViewController : UIViewController {

    @IBOutlet weak var colorPreviewConstraint: NSLayoutConstraint!
    @IBOutlet weak var colorPreviewView: UIView!
    @IBOutlet weak var pickerView: UIPickerView!

    let colors = [ ColorEnums.Blue,  ColorEnums.Red, ColorEnums.Green]
    let sizes = [ SampleViewSizeEnums.Big, SampleViewSizeEnums.Medium, SampleViewSizeEnums.Small ]

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

Şekil 10. Etkin enum kullanımı konusu için yapılan örneğin sayfasının sınıfına ait kod

Şekil 9'daki örnekte görüleceği üzere ColorEnums ve SampleViewSizeEnums isimli 2 adet enum oluşturulmuştur. ColorEnums belirlenen alandaki değişecek renkleri temsil eder, SampleViewSizeEnums ise bu alanın boyutunu temsil eder. Şekil 10'daki örnekte enum bilgileri colors ve sizes isimli değişkenlerde tutulmuştur.

Yukarıdaki bilgiler ışığında Şekil 11'de paylaşılan ekran görüntüsündeki akış şu şekilde olmaktadır; kullanıcı picker aracılığı ile soldaki renk seçimi ile renk seçtiğinde ekrandaki kare alanın rengi değişir, kullanıcı sağdaki boyut seçimini değiştirdiğinde karenin boyutları değişmektedir.



Şekil 11. Etkin enum kullanımı konusu örneği için yapılan sayfanın ekran görüntüsü

Yukarıdaki ekran görüntüsündeki picker'ların yazıları Şekil 12'deki örnekte gene enum'daki karşılıkları kullanılarak basılmıştır. Renkler için kullanılan colors olarak tutulan enum'lar string olduğundan doğrudan kullanılabilmiştir lakin boyutlarla

ilgili olan sizes değişkeninde tutulan enum'lar integer tipinde olduğundan picker'daki yazı karşılıklarının verilebilmesi için switch kullanılmıştır.

```
func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int)
    -> String? {
    if(component == 0 ) {
        return colors[row].rawValue
    } else {
        switch sizes[row] {
        case .Big:
            return "Big"
        case .Medium:
            return "Medium"
        case .Small:
            return "Small"
        }
    }
}
```

Şekil 12. Etkin enum kullanımı konusu için yapılan örneğin sayfasındaki sınıfta picker yazılarını ayarlayan kod

Şekil 13'deki kod örneği picker'da bir değer seçildiğinde çalışan metodu barındırır. Etkin enum kullanımı ile alakalı konuya fayda sağlayacak bir örnektir. Görüleceği üzere kod gayet anlaşılır, kısa ve nettir.

```
func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent
    component: Int) {
    if(component == 0) {
        switch colors[row] {
        case .Red:
            self.colorPreviewView.backgroundColor = UIColor.red
            break
        case .Green:
            self.colorPreviewView.backgroundColor = UIColor.green
            break
        case .Blue:
            self.colorPreviewView.backgroundColor = UIColor.blue
            break
        }
    } else {
        switch sizes[row] {
        case .Big:
            self.colorPreviewConstraint.constant = 270
            break
        case .Medium:
            self.colorPreviewConstraint.constant = 130
            break
        case .Small:
            self.colorPreviewConstraint.constant = 50
            break
        }
    }
}
```

Şekil 13. Etkin enum kullanımı konusu için yapılan örneğin sayfasındaki sınıfta picker'da değer seçildiğinde çalışan kod

3.2.3. Etkin Arayüz Oluşturucu Araçlar Kullanımı

Ekranda gösterilecek görüntülerin (view) kolayca anlaşılması önemlidir. Ekran görüntülerinin çok fazla çaba sarf etmeye gerek olmaksızın görülebilmesi hem ilk geliştirmesini yaparken büyük kolaylık sağlar hem de sonradan yapılacak değişikliklerin daha kolay yapılabilmesini sağlar. Oluşan görüntünün rahatça görülmesi ile görüntüyü (view) geliştiren yazılımcı haricinde başka bir yazılımcı bu görüntüye değişiklik yapması gerektiğinde mevcut halin kolayca anlaşılması sağlanarak geliştirmesini daha kolay ve risksiz yapılması sağlanır. Bunu sağlamanın en iyi yolu görsel arayüz tasarlama araçları kullanmaktır. Bunun IOS geliştirme tarafında karşılığı ise Interface Builder (Arayüz Oluşturucu) olarak tanımlanmıştır.

Arayüz oluşturucu araçların haricinde programmatic çözümle yani doğrudan kodlanarak da görüntüler oluşturulabilmektedir. Lakin bu şekilde geliştirme yapmak önemli problemlere yol açar ve şu problemler ortaya çıkar;

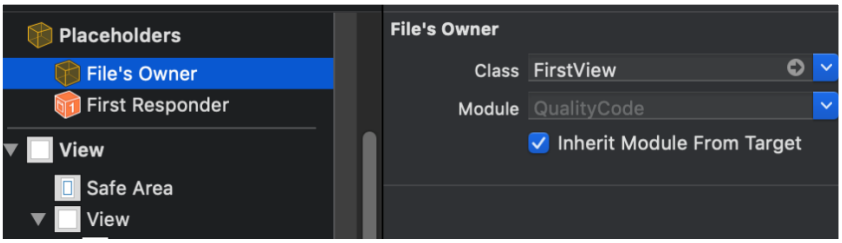
- Kalabalık kod içerisinde geliştirme yapmak ve düzenlemek zorlaşır. Özellikle bir görüntüde birden fazla görüntü iç içe olduğu durumda çok daha zor ve dikkatli bir şekilde geliştirme yapmak zorunda kalınır.
- Bir görüntü sınıfındaki kod miktarı çok artar. Bir View Controller için düşünülürse aşırı kod yazımı sonucunda Massive View Controller oluşur. Yani View Controller içindeki kod miktarı aşırılaşır ve karmaşıklaşır.
- Projenin parçasına o parçayı geliştiren yazılımcı haricinde bir başka yazılımcı müdahale etmek istediğinde çok zorlanır. Müdahaleleri sonucu kestirilemeyen farklı problemlerin de ortaya çıkması olasıdır.
- Herhangi bir müdahale yapmak yer yer imkansızlaşabilir. Çözüm kalmadığında veya o anki şartlara göre yazılımcı kod kopyalayarak kod tekrarlarıyla durumu idare etmek zorunda kalır. Dolayısıyla bu tip bir yöntem kod tekrarlarına yol açar.
- Sonradan araştırılma yapıldığında görselin nasıl gözüktüğü ve nasıl oluşturulduğu net olamadığından problemin kaynağını bulmak veya debug (ayıklama) yapılacak yeri kestirmek zorlaşır.
- Geliştirmesi yapılırken, istenilen görüntüyü oluşturmak daha zor olur. Görüntü sadece çalışma zamanında (runtime) oluşturulabildiğinden çok sayıda build almak gerekecek ve bütün bunlar çok daha fazla zaman kaybına yol açacaktır.
- Görüntü aslında dinamik oluşan bir görüntü olduğundan yani çalışma zamanında derlendiğinde oluşan bir görüntü olduğundan duruma göre görüntü tahmin edilemeyecek kadar farklılaşabilecektir. Dolayısıyla riski daha fazladır.
- Bu çözüm yolu daha özgür bir yöntemdir. Yazılımcı istediği şekilde yazarak görüntüyü oluşturabilir. Lakin bu durumda hata yapma ihtimalleri daha yüksek olur. Görüntüyü oluşturma yöntemleri çok ciddi sorunlar içerebilir. Görüntü ile alakalı işi testlerini yapıp tamamlasalar bile telefon cihaz farkı, ekran boyutu farkı gibi farklı etken faktörlerle karşılaşılması durumunda fark edilemeyen problemler ortaya çıkar.

Arayüz oluşturma araçları ile bu durum kıyaslandığında; arayüz oluşturucular düzenleme esnasında görüntüyü verebildiğinden ne gözükeceği tasarlarken görülebildiğinden ve programmatic çözümde ise çalışma zamanında değişen koşullara göre net olmayan sonuçlar ortaya çıkabileceğinden dolayı programmatic çözüm çok daha riskli bir yöntemdir. Dolayısıyla bir görüntüyü oluşturmak için arayüz oluşturma araçlarını kullanmakta büyük fayda vardır. Arayüz oluşturucular kullanıldığında; ekrana ne gösterileceği net anlaşılır, geliştirmeyi ilk yapan veya sonradan dahil olan yazılımcı görüntüyü daha net anlar ve geliştirmesini daha kolay yapar, problemlerin tespiti daha kolayıdır, görüntülerin kaynağını tespit etmek daha kolaydır, görüntüyü oluşturmak veya değiştirmek için harcanacak efor daha az olacaktır, daha risksiz bir şekilde geliştirilmesinin yapılması mümkün olacaktır.

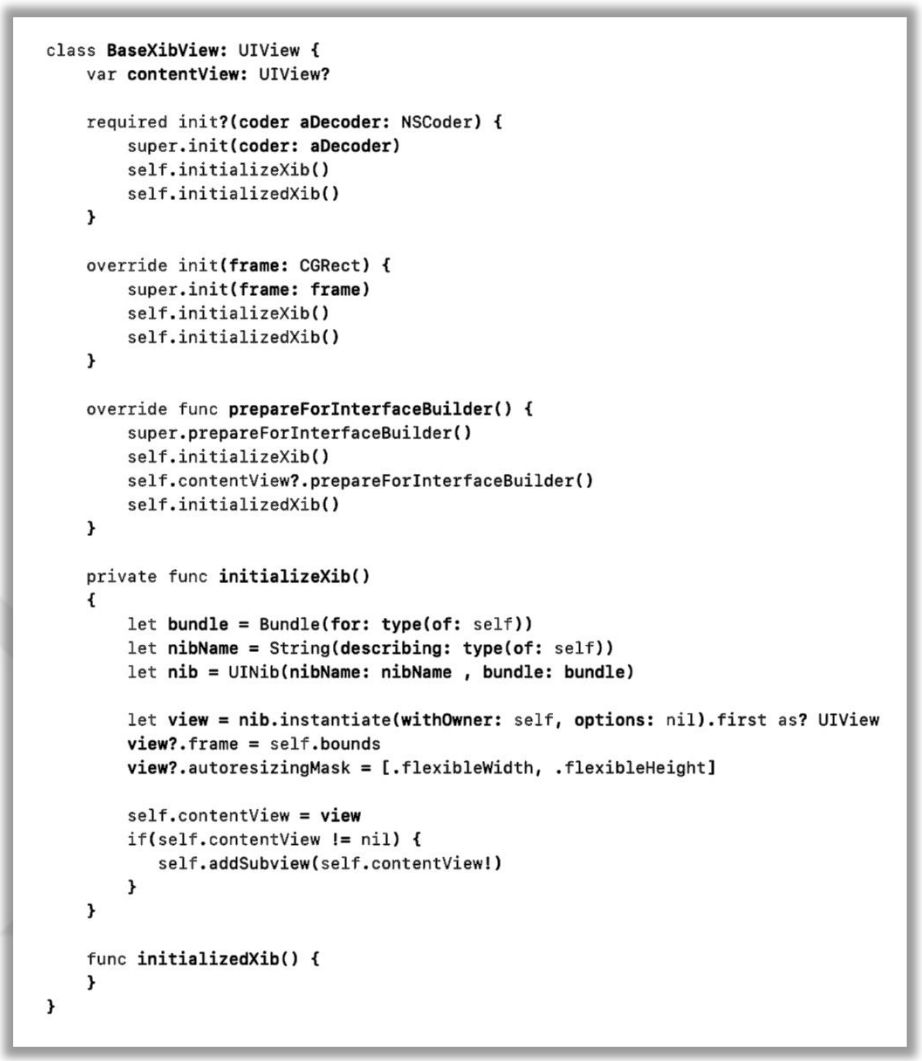
Bu bölümün ilerleyen paragraflarında daha etkin bir şekilde nasıl arayüz oluşturucu araçların kullanılacağı örneklerle birlikte anlatılacaktır. Aracın daha etkin kullanılabilmesi için arayüz tekrarlarının engellenmesine dair çalışma yapılmıştır. Bir görüntü içerisinde başka görüntülerin kolayca nasıl eklenebileceği gösterilmiştir. Görüntüyü kod ile çekmeden direkt bir arayüz içerisine nasıl eklenebileceği anlatılmıştır. Örneklerde görüntüler XIB'ler kullanılarak parçalara ayrılmış ve yer yer iç içe kullanılmıştır.

Şekil 15'de gösterilen BaseXibView koduyla örneklerde kullanılacak base bir sınıf oluşturulmuştur. BaseXibView ve bu yöntemi kullanan yerlerin sınıflarının Şekil 14'deki gibi File's Owner kısmına atanması gerekmektedir. BaseXibView sınıfın görevi Storyboard ve Xib gibi arayüz oluşturucu araçlarda doğrudan görüntülerin eklenebilmesini sağlamak ve bunu kullanan tüm diğer sınıflarda gerekli üst sınıf kodların barındırılmasını sağlamaktır. Görüntünün arayüz oluşturucuda çağrılabilmesini sağlayan kodlar initializeXib isimli metot içinde paylaşılmıştır. Bu metodun kodu incelenirse Xib'in UINib'in instiate metodu ile çekildiği ve akabinde yeni bir view'ın oluşturulduğu görülür. Ardından oluşan görüntünün (view) frame bilgileri mevcut olan frame'e göre ayarlanmış ekrana düzgün oturması için autoresizingMask flexibleWidth ve flexibleHeight bilgileriyle ayarlanmıştır. Bu görüntünün kullanılması için sınıf File's Owner'a eklenmesinden dolayı aslında oluşan bir görüntü olmadığından initializeXib metotu içinde oluşturulan görüntünün

contentView değişkeninde tutulması ve addSubView denilerek oluşan görüntünün eklenmesi gerekmiştir. Görüntüyü çağıran initializeXib metodu programmatic çağrılacaksa frame paremetresi alan init'ten, arayüz oluşturucular ile çağırılacaksa coder parametresi alan init'ten, çalışma zamanı harici geliştirme aşamasında arayüz oluşturuculardа gözükmesi içinse prepareForInterfaceBuilder metodundan çağrılmıştır. Arayüz oluşturucu aracı hazır etmek için prepareInterfaceBuilder çağrımları yapılmıştır. Görüntü oluşması için çağrımın tamamlandığının anlaşılması içinse initializedXib metodu kullanılmıştır.



Şekil 14. Etkin arayüz oluşturucu kullanımı konusunda önerilen yöntemin kullanılabilmesi için sınıfın atanması



```
class BaseXibView: UIView {
    var contentView: UIView?

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        self.initializeXib()
        self.initializedXib()
    }

    override init(frame: CGRect) {
        super.init(frame: frame)
        self.initializeXib()
        self.initializedXib()
    }

    override func prepareForInterfaceBuilder() {
        super.prepareForInterfaceBuilder()
        self.initializeXib()
        self.contentView?.prepareForInterfaceBuilder()
        self.initializedXib()
    }

    private func initializeXib()
    {
        let bundle = Bundle(for: type(of: self))
        let nibName = String(describing: type(of: self))
        let nib = UINib(nibName: nibName , bundle: bundle)

        let view = nib.instantiate(withOwner: self, options: nil).first as? UIView
        view?.frame = self.bounds
        view?.autoresizingMask = [.flexibleWidth, .flexibleHeight]

        self.contentView = view
        if(self.contentView != nil) {
           self.addSubview(self.contentView!)
        }
    }

    func initializedXib() {
    }
}
```

Şekil 15. Etkin arayüz araçları kullanımı konusu örneğinde BaseXibView kodu

Örneğe göre temelde FirstView ve SecondView olmak üzere 2 adet görüntü (view) oluşturulacaktır. Bu iki görüntünün birlikte kullanılmasından oluşan MergedView isimli yeni bir görüntü oluşturulacaktır. Görüntüyü oluşturan sınıflar BaseXibView'dan türeyerek arayüz oluşturuculardan doğrudan görüntü olarak eklenerek çalıştırılabilmeleri sağlanacaktır. Görüntülerin eklenecekleri yerlerde arayüz oluşturucu araçta boş bir görüntü eklemesi yapılmış ve görüntüye sınıf olarak BaseXibView kullanan görüntüye ait sınıf eklenmiştir. Şekil 16, 17, 18, 19, 20, 21, 22 ve 23'teki örneklerde görüntüler ve sınıf kodları paylaşılmıştır. Görüntünün arayüz oluşturucu araçta yansıması için sınıfların başına @IBDesignable isimli nitelik (attribute) ile kullanılmıştır.

Şekil 16. Etkin arayüz oluşturucu araç kullanımı örneğinde FirstView görüntüsü

```
@IBDesignable
class FirstView: BaseXibView {

    @IBOutlet weak var labelOne: UILabel!
    @IBOutlet weak var labelTwo: UILabel!

    override func initializedXib() {
        super.initializedXib()
    }
}
```

Şekil 17. Etkin arayüz oluşturucu araç kullanımı örneğinde FirstView sınıfı kodları

Şekil 18. Etkin arayüz oluşturucu araç kullanımı örneğinde SecondView görüntüsü

```
@IBDesignable
class SecondView: BaseXibView {

    @IBOutlet weak var imageView: UIImageView!

    override func initializedXib() {
        super.initializedXib()
    }
}
```

Şekil 19. Etkin arayüz oluşturucu araç kullanımı örneğinde SecondView sınıfı kodları

FirstView ve SecondView isimli 2 görüntünün birleştiği MergedView'a dair görüntü Şekil 21'de belirtilmiştir. Şekil 20'deki kod ekran görüntüsüne bakılırsa içerdiği 2 görüntü firstView ve secondView isimli 2 değişkende tutulduğu görülecektir.

```
@IBDesignable
class MergedView: BaseXibView {

    @IBOutlet weak var firstView: FirstView!
    @IBOutlet weak var secondView: SecondView!

    override func initializedXib() {
        super.initializedXib()
    }

}
```

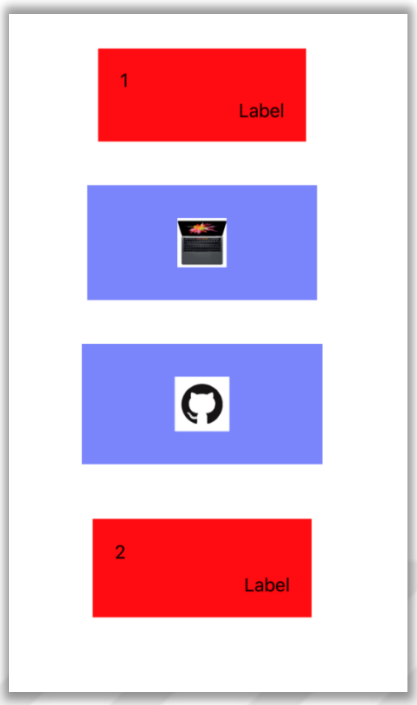
Şekil 20. Etkin arayüz oluşturucu araç kullanımı örneğinde MergedView sınıfı kodları

Şekil 21. Etkin arayüz oluşturucu araç kullanımı örneğinde MergedView görüntüsü

Bu görüntülerin (view) EffectiveInterfaceBuilderUsageViewController isimli bir sayfada kullanıldığı örnek yapılmıştır. Şeki 22'deki paylaşılan kodlarına bakılacak olursa görüntülere ait sınıfların mergedView, secondView ve firstView isimli değişkenlerde tutulduğu görülür. Tüm görüntülerin birleştiği mergedView'daki ilk label yazısına 1 değeri atanarak, ekran görüntüsünde en altta yer alan firstView'ın ilk label yazısına 2 değerini atanarak ve secondView isimli görüntünün resmi de değiştirilerek Şekil 23'de görülen ekran görüntüsü verilmesi sağlanmıştır.

```
class EffectiveInterfaceBuilderUsageViewController: UIViewController {

    @IBOutlet weak var mergedView: MergedView!
    @IBOutlet weak var secondView: SecondView!
    @IBOutlet weak var firstView: FirstView!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.mergedView.firstView.labelOne.text = "1"
        self.firstView.labelOne.text = "2"
        self.secondView.imageView.image = Images.github

    }
}
```

Şekil 22. Etkin arayüz oluşturucu araç kullanımı örneğinde EffectiveInterfaceBuilderUsageViewController sınıfı kodu



Şekil 23. Etkin arayüz oluşturucu araç kullanımı örneğinde EffectiveInterfaceBuilderUsageViewController ekran görüntüsü

3.2.4. Etkin Servis ve Mobil İletişimi

Mobil uygulamanın web servisi ile iletişimi son derece önemlidir. İyi planlanmış ve yazılmış bir servis, onu kullanan mobil uygulamanın kalitesini de doğrudan arttırır. Servisin iyi olması sadece kaliteyi arttırmakla kalmaz aynı zamanda maliyete de doğrudan etki eder. Bir servisin iyi olabilmesi için karmaşık veriler dönmemesi, akışının basit olması, gereksiz bilgiler içermemesi, mobil taraf için verilerin kullanımı esnasında iş katmanı yazılım efor ihtiyacı düşük olması, modeldeki hiyerarşinin karmaşık ve çok iç içe olmaması gerekmektedir. İyi yazılmış bir servis, servisi kullanan kaynakların bilgileri daha kolay kullanabilmelerini sağlar. Dolayısıyla daha kolay kullanılan bir servis anlaşılır bir servistir denilebilir. Servisi kullanan kaynak ihtiyacı olan bilgileri daha rahat kullanır ve ne işe yaradıklarını daha kolay anlar. Böylelikle bir iş yapılacağı zaman gelen bilgilerin bilinmemesi veya yanlış anlaşılması durumlarında doğacak hata riskleri giderilmiş olur.

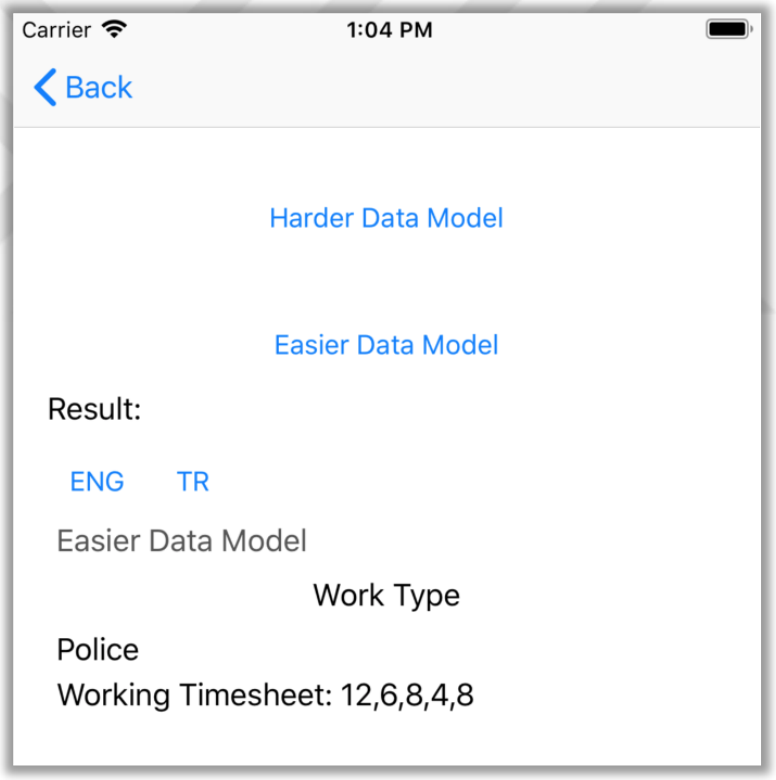
Günümüz koşullarında şirketler mobil platformlarda çalışma yapıyorlarsa IOS ve Android platformu tercih etmektedirler. Genellikle sadece bir platform tercih edilmez. Günümüzde bu iki platform yaygın kullanılmaktadır, bu iki platform haricinde diğer platformlara nadiren destek verilir. Genellikle en az iki mobil yazılım ile uğraşıldığı düşünülürse daha basit, iş katmanı eforu daha düşük olan bir servis ile uğraşıldığında mobil yazılım geliştirme için harcanacak zaman düşecek ve maliyet de düşecektir. Dolayısıyla servisin kalitesinin toplam zaman ve maliyete etkisi daha fazla olacaktır.

Günümüzde şirketlerin büyük bir çoğunluğu servis ile iletişim kuran profesyonel uygulamalar geliştirmektedir. Servis yazılımcıları ile mobil yazılımcılar birlikte çalışmaktadır. Lakin bazı durumlarda özellikle kopuk iletişim yaşanması durumunda gerçekten çok ciddi sıkıntılar ortaya çıkmaktadır. Servisi geliştiren ekip eğer mobil ekipten bağımsız tümüyle kendi tercihleri doğrultusunda geliştirme yaparlarsa mobil uygulama geliştirme eforunu ve verilerin kullanılmasında hata yapılması riskini doğrudan arttırırlar. Bir işte servis geliştirmesini önemli olduğu kadar mobil uygulama geliştirmeleri de bir hayli önemlidir. Unutulmamalıdır ki mobil platformlara yazılım yapılacağı zaman en az iki platforma destek verilmektedir. Dolayısıyla günümüzde IOS ve Android platformları gibi en az iki platformda çalışıldığı dikkate alınırsa, kötü yazılmış bir servis iki kat zaman alacak ve maliyeti olacaktır. Aynı şekilde iyi yazılmış bir servisin pozitif zaman ve maliyet katkısı da en az iki kat olacaktır.

Mobil platform ile çalışacak bir servis olabildiğince mobile özel olmalıdır. Mümkün olduğunca içinde sadece kullanılacak alanları barındırmalıdır. İç içe hiyerarşide uzayıp giden bir modele sahip olmamalıdır. Listeleme yapılacağı zaman liste mobilin ihtiyacına göre tasarlanmalıdır. Sıralama ve filtreleme iş katmanı eforu gerektirdiğinden maliyeti azaltmak ve hata riskini azaltmak için mümkün olduğunca servis tarafında yapılmalıdır. Unutulmamalıdır ki servisten gelen veriyi kullanan yazılımcılar serviste geliştirmeyi yapan kişi kadar veriyi bilmemektedir dolayısıyla olabildiğince basit, isimlendirmelerde amacı net anlaşılan dönüşler yapılmalıdır. Servis geliştirme aşamasında ve sonrasında işin mobil tarafında çalışan kişilerle iyi

iletişim halinde olunmalı ve servisin çalışma yöntemi, döndüğü bilgiler ve nasıl kullanılacağı doğru aktarılmalıdır. Mobil akışa uyumsuz kısımlar düzeltilmelidir.

Bu konuyla alakalı Şekil 24'teki ekran görüntüsü çıktısını veren bir sayfa yapılmıştır. Bu sayfadaki yazılar servisten gelen bilgilere göre doldurulmaktadır. Örnek aynı işi yapan ve aynı bilginin gösterilmesini sağlayan 2 farklı servis dönüşüne sahiptir. Birincisi zor olan servis dönüşüdür ki Harder Data Model isimli tuşa basıldığı zaman servisten bu veri gelir. İkincisi ise Easier Data Model tuşuna basıldığı zaman gelen çok daha basit bir dönüşe sahip olan servistir. Her iki tip serviste İngilizce ve Türkçe olmak üzere 2 dil desteğine sahiptir. ENG ve TR tuşlarına basılarak tercih edilen servisin seçili dile göre dönüşü ekrana basılmaktadır.



Şekil 24. Etkin servis ve mobil iletişimi konusunun örnek uygulamasına ait sayfa sonuç ekran görüntüsü

Şekil 24'deki ekran görüntüsünde görülebileceği üzere gelen veri ile iş tipi için Police, haftalık zaman çizelgesi için 12,6,8,4,8 yazılmıştır.

Şekil 25. Etkin servis ve mobil iletişimi konusu örneğinin sınıfının temel kodları

EffectiveServiceAndMobileCommunicationViewController isimi ile sınıfının temel kodları Şekil 25'deki kod ekran görüntüsünde paylaşılmıştır. 4 adet label ile sırasıyla verinin kaynağı, başlık, iş ismi ve zaman planı ile alakalı çıktıların verilmesi sağlanmıştır. Servisten veri elde edildiği zaman İngilizce veriler engResult, Türkçe veriler ise trResult isimli değişkenlerde tutulacaklardır. Bu değişkenler EffectiveServiceAndMobileCommunicaitionSampleModel isimli bir model kullanılarak ortaklaştırılmıştır.

Şekil 26. Etkili servis ve mobil iletişimi konusunun örneğinde ekrana yazıların basılmasını sağlayan modele ait kod

```
@IBAction func engTapped(_ sender: Any) {
    self.writeOutput(forLanguageCode: "en")
}

@IBAction func trTapped(_ sender: Any) {
    self.writeOutput(forLanguageCode: "tr")
}

func writeOutput(forLanguageCode languageCode: String) {
    var outputInfo: EffectiveServiceAndMobileCommunicationSampleModel?
    if(languageCode == "tr") {
        outputInfo = self.trResult
    } else {
        outputInfo = self.engResult
    }

    if(outputInfo != nil) {
        self.titleLabel.text = outputInfo!.title
        self.jobNameTitle.text = outputInfo!.jobName
        self.timesheetLabel.text = ""
        var isFirstItem: Bool = true
        for item in outputInfo!.timeSheetOfWeek {
            self.timesheetLabel.text?.append("\(isFirstItem ? "Working Timesheet: " :
                ",")\(String(describing: item))")
            isFirstItem = false
        }
    } else {
        self.titleLabel.text = "No Data Available!"
        self.jobNameTitle.text = "No Data Available!"
        self.timesheetLabel.text = "No Data Available!"
    }
}
```

Şekil 27. Etkili servis ve mobil iletişimi konusunun örneğinin sınıfında ENG ve TR tuşlarına basılınca devreye giren action metotları ve label'lara çıktıyı veren writeOutput isimli metodu içeren kod

Ekrana verinin basılması için kullanılan model ile belirli alanlara yazıların basılması sağlanmıştır. Şekil 27'deki örnekte iki dile destek veren writeOutput metodu ile belli bir model yapısından alınan belirli bilgilerle label'lara yazıların basılması sağlanmıştır. Model incelenecek olursa değişkinler opsiyonel tip değildir. Dolayısıyla içi boş değer atanamaz durumdadır. Eğer içi boş değer gelirse uygulama çökecektir. Tablo 1, 2 ve 3'te zor ve kolay dönüş verilerine yönelik örnekler paylaşılacaktır. İki örnekte de gelen veri ekrana görüntüyü oluşturan modele dönüştürülecektir ve mock data (test için kullanılan sahte veri) kullanarak servise gidiyormuş gibi bir senaryo üzerinden çalışacaktır.

İlk olarak zor veri dönüşü örneğine değinilmiştir. Tablo 1'deki örnekteki veri dönüşü zor olan karmaşık bir veri dönüşüdür. JSON tipinde dönüş örneği paylaşılmıştır. İç içe giren pek çok model ve dizin alanlarına sahiptir.

Tablo 1. Etkin servis ve mobil iletişimi konusunun örneğinde zor ve karmaşık olan modele sahip bir JSON örneği

```
{
 "place": 1,
 "titleLocaliztions": [
  {
   "tag": "title",
   "localization": {
    "text": "Work Type",
    "shortCode": "en",
    "longCode": "English"
   }
  },
  {
   "tag": "title",
   "localization": {
    "text": "İş Tipi",
    "shortCode": "tr",
    "longCode": "Türkçe"
   }
  }
 ],
 "workDetail": {
  "jobDetail": [
   {
    "tag": "jobDetail",
    "localization": {
     "text": "Police",
     "shortCode": "en",
```

```
        "longCode": "English"
       }
     },
     {
       "tag": "jobDetail",
       "localization": {
         "text": "Polis",
         "shortCode": "tr",
         "longCode": "Türkçe"
       }
     }
   ],
   "workHoursOfWeek": [ 12, 6, 8, 4, 8 ]
  }
}
```

Örneklerde servisten alanların gerçek hayattaki gibi boş dönebilme ihtimaline karşın data model opsiyonel olarak yani null olabilir değişkenler olarak tanımlanmıştır.

Şekil 28'deki zor veri dönüşü kodu örneğinde görülebileceği üzere ilgili metot olan harderDataTapped metodu içesinde çok yoğun bir kod vardır. Aslında çok basit 3 tane alanı doldurmak için çok yoğun efor gerektiren bir model olduğu Tablo 1'de paylaşılan JSON örneğinde ve Şekil 28'deki kod örneğinde bariz bir şekilde görülebilmektedir.

Servisten çekilen veri fetchedData değişkenine atanmıştır. İlk olarak başlığı çekebilmek için titleLocalizations dizin alanının içinden localization içine girmiş ve buradan da shortCode kontrolü yaparak içerideki text alanını uygun dildeki başlıkla alakalı title değerine atamasını sağlamıştır. Ardından workDetail içine girmiş buradan ilk workHoursOfWeek alanından iş planı için çalışma saatlerini dizin olarak almıştır. Sonrada gene workDetail içindeki jobDetail içine girmiş buradan iş ismini yazabilmek için gerekli metni gene benzer localization mantığıyla text alanından iç içe yapılardan elde etmiştir.

Görüleceği üzere yapılması gereken iş basit olsa da kötü tasarlanmış bir servis dönüşü yüzünden çok ciddi uğraş gerekmektedir. Bu işi tamamlayabilmek için gereğinden daha fazla zaman kaybedilmesine sebep olmuştur. Gerçek hayattaki gibi servisten alanların boş gelebileceği dikkate alınarak null kontrolleri yazılmıştır. Alanların boş gelebilme ihtimali bir projenin riskini arttırır. Dolayısıyla bu kadar iç içe karmaşık bir yapıda riski çok ciddi oranda artmaktadır

```
@IBAction func harderDataTapped(_ sender: Any) {
  // Assume that HarderDataModel is fetched from service..
  let fetchedData = HarderDataModel()

  self.trResult = EffectiveServiceAndMobileCommunicationSampleModel()
  self.engResult = EffectiveServiceAndMobileCommunicationSampleModel()

  self.trResult?.title = ""
  self.trResult?.jobName = ""
  self.trResult?.timeSheetOfWeek = [Int]()

  self.engResult?.jobName = ""
  self.engResult?.title = ""
  self.engResult?.timeSheetOfWeek = [Int]()

  if (fetchedData.titleLocaliztions != nil && fetchedData.titleLocaliztions!.count > 0) {
      for titleItem in fetchedData.titleLocaliztions! {
          if let shortcode = titleItem.localization?.shortCode,
              let text = titleItem.localization?.text {

              if(shortcode == "tr") {
                  self.trResult?.title = text
              } else {
                  self.engResult?.title = text
              }
          }
      }
  }

  if(fetchedData.workDetail != nil) {
      if let workHoursOfWeek = fetchedData.workDetail!.workHoursOfWeek {
         self.trResult?.timeSheetOfWeek = workHoursOfWeek
         self.engResult?.timeSheetOfWeek = workHoursOfWeek
      }

      if (fetchedData.workDetail!.jobDetail != nil && fetchedData.workDetail!.jobDetail!.count > 0) {
          for jobItem in fetchedData.workDetail!.jobDetail! {
              if jobItem.localization != nil,
                  let shortcode = jobItem.localization!.shortCode,
                  let jobname = jobItem.localization!.text {

                  if(shortcode == "tr") {
                      self.trResult?.jobName = jobname
                  } else {
                      self.engResult?.jobName = jobname
                  }
              }
          }
      }
  }

  self.sourceLabel.text = "Harder Data Model"
  self.writeOutput(forLanguageCode: "en")
}
```

Şekil 28. Etkin servis ve mobil iletişimi konusunda zor ve karmaşık olan veri dönüşü ile çalışan örneğe ait harderDataTapped metodu kodu

Eğer veri daha basit olsaydı hem riski azalacak hem daha kısa sürede yapılacağından maliyeti daha az olacaktı. Tablo 2, Tablo 3 ve Şekil 29'daki örnekte daha basit bir dönüş verisiyle bir örnek paylaşılmıştır.

Tablo 2. Etkin mobil ve servis iletişimi konusunun örneğinde Türkçe bilgiler içeren basitleştirilmiş servis dönüşü JSON örneği

```
{
    "place":1,
    "title":"Work Type",
    "worhHoursOfWeek":[12,6,8,4,8],
    "jobTitle":"Police"
}
```

Tablo 3. Etkin mobil ve servis iletişimi konusunun örneğinde İngilizce bilgiler içeren basitleştirilmiş servis dönüşü JSON örneği

```
{
    "place":1,
    "title":"İş Tipi",
    "worhHoursOfWeek":[12,6,8,4,8],
    "jobTitle":"Polis"
}
```

Tablo 2 ve Tablo 3'teki JSON örneklerinde daha basit bir şekilde dönülen basitleştirilmiş örnekler paylaşılmıştır. Görüleceği üzere bu yapıda okuması ve anlaması daha rahat olduğu gibi bu tip bir dönüşle çalışmak hem daha kolaydır hem de riski daha azdır.

```
@IBAction func easierDataTapped(_ sender: Any) {
    // Assume that EasierDataModel is fetched from service..
    let fetchedENGData = EasierDataModel(langCode: "en")
    let fetchedTRData = EasierDataModel(langCode: "tr")

    self.trResult = EffectiveServiceAndMobileCommunicationSampleModel()
    self.engResult = EffectiveServiceAndMobileCommunicationSampleModel()

    self.engResult?.title = fetchedENGData.title ?? ""
    self.engResult?.jobName = fetchedENGData.jobTitle ?? ""
    self.engResult?.timeSheetOfWeek = fetchedENGData.worhHoursOfWeek ?? [Int]()

    self.trResult?.title = fetchedTRData.title ?? ""
    self.trResult?.jobName = fetchedTRData.jobTitle ?? ""
    self.trResult?.timeSheetOfWeek = fetchedTRData.worhHoursOfWeek ?? [Int]()

    self.sourceLabel.text = "Easier Data Model"
    self.writeOutput(forLanguageCode: "en")
}
```

Şekil 29. Etkin mobil ve servis iletişimi konusunda basit veri dönüşü ile çalışan örneğe ait easierDataTapped metodu kodu

Daha önce harderDataTapped metodu içerisindeki kodlar paylaşılmıştı. Yukarıdaki easierDataTapped kodları ile kıyaslandığı zaman aradaki bariz fark görülebilmektedir. Bu çalışmada aynı işi ve aynı bilgiyi veren 2 farklı servis dönüş modeli tasarlanmıştır. İlk olarak tasarlanan zor model kullanıldığında mobil tarafta çok karmaşık kod yazılmak zorunda kalınmış ve bu da hem zaman kaybettirerek maliyeti arttırmış hem de hata riskini arttırmıştır. Ancak ikinci tasarlanan basit modelde hem daha kısa sürede iş tamamlanabilmiş, hem daha okunabilir kod yazılabilmesi sağlanmıştır. Böylece hem geliştirilirken daha az maliyetle iş bitirilmiş hem de ilerleyen dönemde yapılacak değişikliklerin daha kolay yapılabilmesi sağlandığından bakım maliyeti de azalmıştır. Mobil platform geliştirilen bir projede IOS ve Android gibi en az 2 platforma da geliştirme yapılacağı var sayılırsa bu farkın maliyete pozitif etkisi 2 kat olacaktır.

Mobile uygun servis geliştirilmesi ile alakalı anlatılmış örnek üzerinden süre sayılarak uygulamanın geliştirilmesi aşamasında harcanacak süreye ait sonuçlar elde edilmiştir. Bu sonuçlara göre karmaşık modelin sadece dönüş modelinin oluşturulması 6 dakika 15 saniye sürmüştür. Dönüş modeli oluşturulduktan sonra 6 dakika 44 saniye

harcanarak dönen verinin ekrana basılması sağlanmıştır. Karmaşık model ile tüm geliştirme toplamda 12 dakika 59 saniye sürmüştür. Basit model ile yapılan geliştirme süresi sayıldığında dönüş modelinin oluşturulmasının sadece 36 saniye sürdüğü, dönen verinin ekrana basılmasının ise sadece 1 dakika 19 saniye sürdüğü tespit edilmiştir. Basit model ile tüm geliştirme toplam 1 dakika 55 saniye sürmüştür. Sonuçlara göre basit model, zor modele göre yaklaşık 6.77 kat daha hızlı geliştirilmiş ve 11 dakika 4 saniye daha erken geliştirilmesi tamamlanmıştır. Böylesine olumlu bir katkı mobil projelerde genellikle en az 2 platformun olduğu varsayılırsa 2 kat pozitif katkı sağlayacak önemli bir sonuçtur. Uygulanan örnek ile elde edilen sonuçlarda geliştirme sırasında hata yapılmadan hızlıca ilerlenmiştir. Gerçek projelerde hatalar dolayısıyla kaybedilecek zamanlar da dikkate alınırsa karmaşık model ve basit model arasındaki zaman farkı basit model lehine daha da büyüyecektir.

4. TASARIM ÖRÜNTÜLERİ

Yazılım projelerinde tasarım örüntüleri kullanımı büyük önem arz eder. Projenin kalitesini arttırdığı gibi okunabilirliği arttırıp belirli bir standart oluşturması sebebiyle işi kolaylaştırmaktadır. Tasarım örüntüleri farklı platform ve yazılım dillerinde kullanılabilmektedir.

Tasarım örüntüsü kullanımı ile yazılan kod her ne kadar belirli bir standartta yazılıyor olsa da bazı noktalarda ucu açık kısımları da olabilmektedir. Ucu açık kısımlarda bazen şirketler belli bir standart getirip ona göre geliştirme yapsa da çoğunlukla o projeyi geliştiren kişinin oluşturmuş olduğu tarzda çalışılmaya devam edilmektedir. Projeye başka bir çalışan sonradan dahil olduğunda ucu açık olan kısımlarda çok ciddi adaptasyon sıkıntısı yaşanmaktadır ve çoğu durumda da sonradan katılan kişi önceden oluşturulmuş standart veya stile uymayarak tamamen kendi oluşturduğu apayrı bir yöntemle projeyi devam ettirerek yazılımın okunabilirliği azaltmakta ve karmaşıklığı arttırmaktadır. Durum dolayısıyla tasarım örüntülerinde ucu açık olan kısımlar için de belirli standart veya yöntemler geliştirilmesi ihtiyacı mevcuttur. Şirketler bunları kendi içlerinde oluştursa da dünyadaki tüm yazılımcıların öğrenebileceği belirlenmiş kurallar, yöntemler, standartlar ve yöntemler belirlenmelidir.

Tasarım örüntülerinde bir diğer sorun ise bazı örüntülerin yanlış kullanımı veya bazı noktalara dikkat edilmemesi sebebiyle önemli sorunlar yaşanabiliyor oluşudur. Örneğin Template Pattern'in kullanmasında yazılımcının yaygınca yaptığı hatalardan bir tanesi süper sınıftan ovverride method'un çağırılmamasıdır. Her ne kadar basit bir hata gibi gözükse ve yazılımcının böyle bir hata yapmaması gerektiği düşünülebilse de yaygınca yapılan ve hatta bazen yazılımcı tarafından önemsenmeyip göz ardı edilen bir sorundur. Bu tip problemlerin yaşanmasını engellemek için tasarım örüntüleri kullanımında olası sorunları veya dezavantajları gideren çözümlere ihtiyaç vardır.

Bu bölümde tasarım örüntülerinde ucu açık kalan kısımlarla ilgili belirli standart getiren veya öncesinde bahsedilen sorunlara çözüm bulan çözümler araştırılacaktır. Tasarım örüntülerinde dezavantajlı kısımların giderilmesi ve olası problemlerin çözülmesi için yöntemler belirlenecektir.

4.1. IOS Platformunda Tasarım Örüntüleri Kullanımı

Mobil platformlar kaliteyi sürdürebilmek için belirli standartlara sahiplerdir. Aynı zamanda mobil platformun SDK'sı içerisinde belirli çözümlere sahip özellikler hazır olarak gelmektedir. Bunlara örnek vermek gerekirse bir listeleme operasyonu, eposta gönderimi, galeriden fotoğraf alma vs. gibi pek çok özellik SDK tarafından desteklenerek yazılım geliştiricinin bunların kullanmasına olanak sağlar. Bu destek sayesinde aynı zamanda yazılım geliştirici ayrıca bu özellikleri sıfırdan yapmakla uğraşmaz ve uzunca süre üzerinde uğraşılmış ve stabil hale getirilmiş özellikleri kolay bir şekilde kullanma imkanına kavuşur.

Tasarım örüntüleri platformun varsayılan özelliklerinde de yaygınca kullanılmaktadır. Mesela Android'de Reclycle View kullanılırken Adapter pattern kullanılır, IOS platformunda bir sayfadaki görüntü (view) yönetilirken Template Pattern kullanır. Bunun gibi pek çok örneğe mobil platformda yazılım geliştirirken varsayılan olarak veya standart olarak rastlanır. Hatta bu örüntüler pek çok yerde opsiyonel olmak yerine belirli bir özelliğin kullanılabilmesi için varsayılan olarak zorunlu örüntüler olarak karşımıza çıkarlar.

IOS platformunda diğer mobil platformlar gibi belirli tasarım örüntülerini yaygınca kullanmaktadır. Bilinen yaygınca kullanılan tasarım örüntülerinden bazıları şunlardır; Abstract Factory, Adapter, Factory Method, Template Method, Chain of Responsibility, Command, Observer, Composite, Decorator, Facade, Iterator, Mediator, Memento, Proxy, Receptionist, Singleton, MVC

4.2. Tasarım Örüntüleri Detaylı İnceleme

Aşağıda belirtilen tasarım örüntüleri tipleri incelenmiştir:

- Oluşturucu Tasarım Örüntüleri (Creational Patterns)
- Yapısal Tasarım Örüntüleri (Structural Patterns)
- Davranışsal Tasarım Örüntüleri (Behavioral Patterns)
- Mimari Tasarım Örüntüleri (Architectural Patterns)

Bu bölümde avantaj ve dezavantajlara değinilecek, dezavantajlı kısımları giderici çözümler paylaşılacaktır. Araştırmada gerekli görülmeyen tasarım örüntüleri detaylandırılmayacaktır.

4.2.1. Oluşturucu Tasarım Örüntüleri

Nesneleri yaratmakla görevli tasarım örüntüsüdür. İngilizce ismi "Creational Pattern" olarak adlandırılır. Yaygın oluşturucu tasarım örüntülerinden bazıları şunlardır: Factory, Abstract Factory, Builder, Prototype, Singleton.

4.2.1.1. Prototype Pattern

Bu tasarım örüntüsü dofactory.com'a göre %60 oranında kullanım sıklığına sahiptir. Prototype tasarım örüntüsü genellikle üretim maliyetini ortadan kaldırmak için kullanılır. "Deep Copy" yani derin kopyalama şeklinde kopyalama işlemi yapar. Nesne birebir kopyalanarak yeni referans değerine atanır [27].

Yazılım geliştirirken bazı noktalarda aynı nesneleri az bir değişikliğe uğratarak tekrar kullanmak durumunda kalınır. Bu örüntü bunun yapılabilmesi için işi daha da kolaylaştırmaktadır. Lakin bu örüntü aynı nesnenin birebir aynısını farklı bir referans ile yaratmaktadır. Nesnelerin istenen bazı kısımlarını başka tipte modelde kopyalama işlemi yapmamaktadır.

Yazılımda mobil tarafın servis tarafıyla olan iletişiminde kullanılan data modelleri ayırmakta büyük fayda vardır. Geliştiriciler data modelleri manipüle ederek kod içerisinde tekrar tekrar normal bir modelmiş gibi kullanmamalıdır. Bu şekilde kullanım sonucu kodun anlaşılması zorlaşmakta ve servisten gelen modelde yanlışlıkla değiştirme yapılması gibi problemler ortaya çıkmaktadır. Yanlışlıkla data modelin değiştirilmesini veya hatalı kullanımını engellemek için data modeller tek seferde değiştirilemeyen tip olan let olarak tanımlanmalı ve bu sayede sonradan değişikliğe açık olmamalıdırlar.

Bu örüntü data modellerin bir kısmını veya istenildiği şekilde kullanabilme olanağını vermemektedir ayrıca bu yapıda data modeli önerilen şekilde kullanmamız mümkün olmamaktadır. Bu açıdan düşünürse bu örüntü ile kazanımlarımızı

kullanamıyor olduğumuz görülebilir. Bunun için yeni bir yaklaşım olarak "4.3.1. Imititate Pattern" bölümünde anlatılan örüntünün kullanılması tavsiye edilmektedir.

Ancak eğer model aynı şekilde kullanılacak ve sadece küçük bir değişiklik yapılacaksa bu durumu let nesnelerde de kullanmak mümkündür. Bunun için instance alındığı esnada gerekenin yapılması yeterli olacaktır. Şekil 30 ve Şekil 31'de bu konuyla alakalı örnek kodlar paylaşılmıştır.

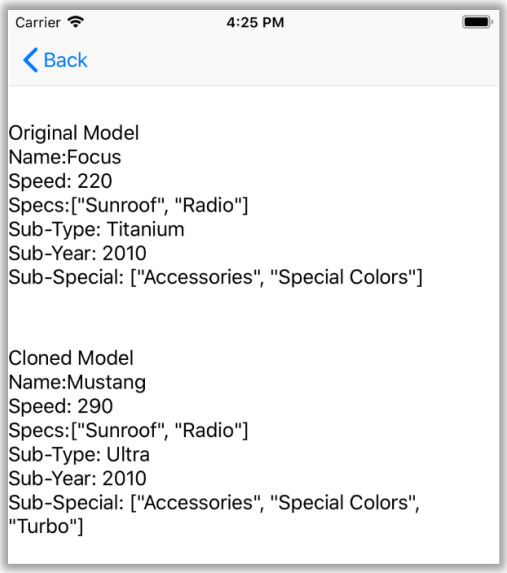
```
class PrototypeModel
{
    // Assume that this is data model and values are protecting with let. You want to send same data with
        little changes than you can do this with example solution.
    // This example you will see that we are giving oppurtunity to change the name field with cloning
    let name: String!

    // TRADITIONAL PART START
    var speed: Int!
    var brand: String!
    var specs: [String]!
    var subval: SubPrototypeModel!
    // TRADITIONAL PART END
    init(name: String, speed: Int, brand: String, specs:[String], subval: SubPrototypeModel) {
        self.name = name
        self.speed = speed
        self.brand = brand
        self.specs = specs
        self.subval = subval
    }

    convenience init(prototypeModel: PrototypeModel, newName: String? = nil)
    {
        let clonename: String! = newName != nil ? newName! : prototypeModel.name
        self.init(name: clonename, speed: prototypeModel.speed, brand: prototypeModel.brand, specs:
            prototypeModel.specs, subval: prototypeModel.subval.clone())
    }

    func clone(newName: String? = nil) -> PrototypeModel
    {
        return PrototypeModel(prototypeModel: self, newName: newName)
    }
}
```

Şekil 30. PrototypeModel kodu

```
class PrototypePatternViewController: UIViewController {

    @IBOutlet weak var result1: UILabel!
    @IBOutlet weak var result2: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        let orgmodel = PrototypeModel(name: "Focus", speed: 220, brand: "Ford", specs: ["Sunroof",
            "Radio"],
                                      subval: SubPrototypeModel(type: "Titanium", year: 2010, special:
                                          ["Accessories", "Special Colors"]))
        let clonedModel = orgmodel.clone(newName: "Mustang")
        clonedModel.speed = 290
        clonedModel.subval.type = "Ultra"
        clonedModel.subval.special.append("Turbo")

        result1.text = "Original Model \(getPrintString(model: orgmodel))"
        result2.text = "Cloned Model \(getPrintString(model: clonedModel))"
    }

    func getPrintString(model: PrototypeModel) -> String
    {
        return  "\nName:\(model.name!)\nSpeed: \(model.speed!)\nSpecs:\(model.specs!)\nSub-Type:
            \(model.subval.type!)\nSub-Year: \(model.subval.year!)\nSub-Special: \(model.subval.special!)"
    }

}
```

Şekil 31. PrototypePatternViewController kodu

Şekil 30'daki örnekte görüleceği üzere name değeri let ile koruma altına alınmıştır. Bu alan sadece init ile çağrıldığı zaman oluşturulabilmektedir. "clone" metodu ile farklı bir isim kullanılabilmesini görüldüğü üzere sağlayabildik. Bunun için metot çalıştığı zaman yeni bir instance yaratılma aşamasında name alanını değiştirmek yeterli olmuştur.



Şekil 32. Prototype örüntüsü örneğinin sonuç ekran görüntüsü

Şekil 32’deki Sonuç ekranında görülebileceği üzere bir araba nesnesi kopyalanarak aynı özelliklerin kullanılması sağlanmış ve farklılıklar ise sonradan yapılan değiştirilerek kullanılmıştır. Data model veya bunun gibi orijinalliğinin korunması ve tekrar kullanımlarda farklı nesne ile kullanılması gereken durumlarda önerilen yöntemle güvenli bir şekilde kopyalama yapılabilir. Eğer model belli parçalar halinde farklılaştırılarak kullanılmak isteniyorsa Imititate Pattern kullanılmalıdır.

4.2.1.2. Singleton Pattern

Singleton tasarım örüntüsü sınıfın tek seferde instance yaratılmasını ve her yerden tekrar tekrar yaratmaya gerek kalmaksızın tek seferde oluşturulmuş sınıftan işleyin devam edebilmesini sağlayan bir örüntüdür ve yaygın kullanılmaktadır. Singleton sayesinde sınıfa kolayca ulaşılır ve sınıfı çağrılan yerde oluşturma derdi yoktur.

Lakin bu tasarım örüntüsünde yaygınca yapılan hata ise geliştiricilerin sınıfı oluşturulmasını engellememeleridir. Eğer bir sınıfta singleton kullanılmışsa bu sınıfın normal yollarla oluşturulmaması gerekir. Eğer oluşturulursa singleton ile çalışan nesne ile birlikte çalışmasında sorunlar ortaya çıkartabilir. Aynı zamanda singleton olan yerlerde class func’ta kullanılmamalıdır. Bu problemi gidermek için basitçe instance yaratma yani init çağrımlarını private’a çekmektir. Böylelikle yanlışlıkla instance yaratılamayacaktır. Şekil 33’deki örnekte init private’a çekilerek bu problem giderilmiştir.

```
import Foundation

class SingletonClass {

    static let shared = SingletonClass()
    private var count = 0
    private var createDate:Date?

    private init() {
        self.count = 0
        self.createDate = Date()
    }

    func getSharedCount() -> Int
    {
        self.count = self.count + 1
        return self.count
    }

    func getCreateDate() -> Date?
    {
        return self.createDate
    }
}
```

Şekil 33. Problem riskini ortadan kaldıran Singleton sınıf örneğine ait kod

4.2.2. Yapısal Tasarım Örüntüleri

İngilizce ismi "Structural Patterns" nesnelerin birbirleriyle olan ilişkisini düzenleyen örüntülerdir [28]. Yaygınca kullanılan yapısal tasarım örüntüleri şunlardır: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Front Controller, Module, Proxy.

4.2.3. Davranışsal Tasarım Örüntüleri

Birden fazla sınıfın bir işi yerine getirirken nasıl davranacağını belirleyen örüntülerdir. İngilizce ismi "Behavioral Patterns" olarak adlandırılmıştır. Yaygınca kullanılan davranışsal tasarım örüntüleri şunlardır: Chain of Responsibility, Command, Mediator, Memento, Observer, Strategy, Template, Visitor

4.2.3.1 Template Pattern

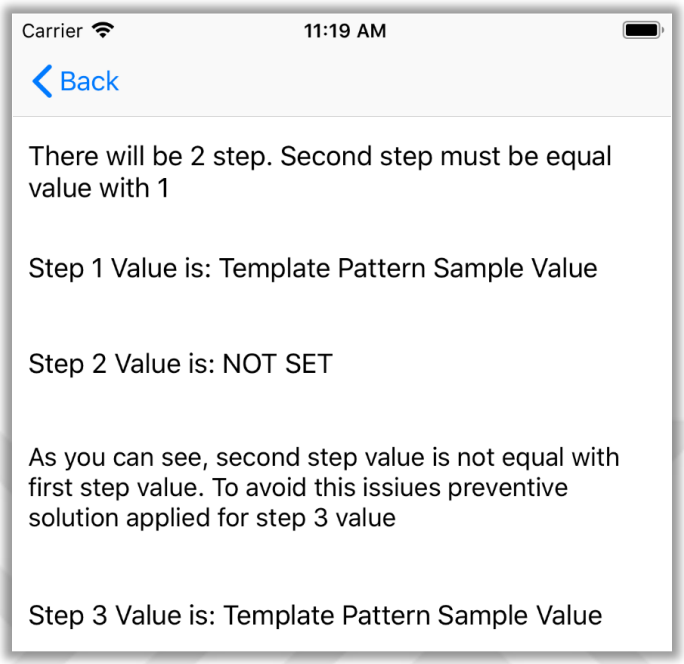
Bu örüntü alt sınıfların genel yapısını değiştirmeden bir algoritmanın belirli adımlarını yeniden tanımlamasına izin verir. Bu tasarım örüntüsü bir algoritmayı bir adım dizisine böler, bu adımları ayrı yöntemlerde tanımlar ve bunları tek bir şablon yönteminin yardımıyla ardışık olarak çağırır. Bu tasarım örüntüsü genel yapı değişmeden alt sınıflarda basit algoritma ile fonksiyonun genişletilmesi gerektiğinde veya bir sınıfı değiştirildiğinde diğer sınıflarında değişmesi gerektiğinde kullanılması tavsiye edilir [29].

Temelde bir şablon kullanımını baz alan bu örüntü IOS platformunda çok yaygınca kullanılmaktadır. Özellikle UIViewController sınıfı kullanıldığında bu sınıfın özelliklerinden olan bir view'ın (görüntünün) yüklendiğini, gözüktüğünü, artık gözükmediğini ve bunun gibi birtakım bilgileri template pattern sayesinde anlaşılması sağlanmıştır. Çok yararlı bu örüntü maalesef yazılım geliştiricilerin hatalı kullanması sebebiyle projede önemli sorunlara sebebiyet verebilmektedir. Yazılım geliştiriciler her ne kadar tecrübeli olsalar da bu hatayı çok kolay bir şekilde yapabilmekte ve durumu gözden kaçırabilmektedirler.

Bu örüntü delegasyonla veya parent-child ilişkisi ile rahatça kullanılabilmektedir. Yazılımcıların kolayca yapabilecekleri hataları gidermek adına tezin devam eden kısımlarında çözüm önerisi paylaşılmıştır ve bu önerinin standart olarak kullanılması tavsiye edilmektedir.

Bu bölümün devam eden kısımlarında Şekil 34, Şekil 35 ve Şekil 36 ile konuyla alakalı bir çalışma yapılmıştır. Örnekte TemplateViewController sınıfı üzerinden bir sayfa açılmaktadır ve bu sayfa TemplateBaseViewController isimli bir sınıftan türeyerek kendine bu sınıfı şablon almıştır. Bu sınıflar Template Pattern kullanmışlardır. Şablon sınıflardan step1Value, step2Value ve step3Value olarak değerler elde edilmektedir. Şekil 34'te görülen örnekte step1Value değeri beklenen değer olarak verilmiş, diğer steplerde step 1'deki değer ile aynı değerin yazılmış olması beklenmiştir. Template Pattern'in yanlış kullanılması sonucu step2Value değerinin yanlış geldiği görülmektedir. Bu problemi step3Value değerinde bu problem çözülmüştür. TemplateViewController sınıfı incelenirse step 2'deki değerin step 3 ile

aynı şekilde süper sınıftaki metod ile çağrılmadığı için hatalı olarak kullanıldığı görülecektir. Lakin buna rağmen step 3 değeri doğru gelmiştir.



Şekil 34. Template örüntüsü örneğinin sonuç ekran görüntüsü

```
class TemplateViewController: TemplateBaseViewController {
    @IBOutlet weak var topTitleLabel: UILabel!
    @IBOutlet weak var step1Label: UILabel!
    @IBOutlet weak var step2Label: UILabel!
    @IBOutlet weak var step3Label: UILabel!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.step1Label.text = "Step 1 Value is: \(super.step1Value)"
    }

    override func step2Setter() {
        self.step2Label.text = "Step 2 Value is: \(super.step2Value)"
    }

    // Preventive Solution Start
    override func step3Setter() {
        self.step3Label.text = "Step 3 Value is: \(super.step3Value)"
    }
    // Preventive Solution End
}
```

Şekil 35. TemplateBaseViewController'dan türeyen TemplateViewController sayfasının sınıf kodu

Şekil 35'te görüleceği üzere problemin çözümü için base sınıfta asıl işi yapan metot private olarak tanımlanmıştır ancak ovverride metodunun çalışabilmesi için step3Setter() alt sınıftan çağrılabilir halde bırakılmıştır. Burada asıl yapılan mutlaka çalışması gereken kodun izole edilerek yanlışlıkla çalışmasının engellenmesi ihtimalini ortadan kaldırmaktadır. İstenirse step3Setter() metodu içerisinde çalışması zorunlu olmayan kodlar eklenebilir lakin zorunlu kodların mutlaka çalışması sağlanmıştır. Bu çözümle bilerek ya da yanlışlıkla süper çağrımı yapmanın unutulması sorunu giderilmiştir. Kısaca anlatılmış bu çözüm aynı zamanda Protected Template Pattern isimli yeni bir örüntü olarak 4.3.2. bölümde anlatılmıştır.

```
class TemplateBaseViewController: UIViewController {

    var step1Value: String = "NOT SET"
    var step2Value: String = "NOT SET"
    var step3Value: String = "NOT SET"

    var expectedValue = "Template Pattern Sample Value"

    override func viewDidLoad() {
        super.viewDidLoad()
        self.step1Value = self.expectedValue
        self.step2Setter()
        self.secureStep3Setter()
    }

    func step2Setter()
    {
        self.step2Value = self.expectedValue
    }

    // Preventive Solution Start
    private func secureStep3Setter() {
        self.step3Value = self.expectedValue
        self.step3Setter()
    }

    func step3Setter()
    {

    }
    // Preventive Solution End
}
```

Şekil 36. Base alınan TemplateBaseViewController sınıfı kodu

4.2.4. Mimari Tasarım Örüntüleri

İngilizce ismi ile "Architectural Patterns" bir yazılım projesinin omurgasıdır. Projenin mimari yapısı kullanılan mimari örüntüsü ile temsil edilir. Yapılacak projeye başlamadan önce kullanılacak tasarım örüntüsüne karar verilir. Genellikle tercih edilen mimari örüntü değiştirilmez ve proje var olduğu sürece ilk seferde karar verilen mimari örüntü ile geliştirmesi yapılmaya devam edilir. Bu sebeple başlangıçta verilen karar çok önemlidir. Proje için doğru mimari tasarım örüntüsünü belirlemek projenin kalitesini ve maliyetini doğrudan doğruya etkileyen en önemli unsurdur. Bu karar verilirken geniş bir çerçevede değerlendirme yapmak ve bütün faktörleri göze alarak tercih yapmak gereklidir.

Mobil platformlarda kullanılan başlıca mimari tasarım örüntüleri şunlardır: MVC, MVVM, MVP, MVP-VM, VIPER. Destekleyici mimari örüntü olarak Coordinator Pattern örnek gösterilebilir

4.2.4.1. MVC Pattern

Günümüzde pek çok platformda en sık kullanılan mimari tasarım örüntüsü MVC'dir. IOS ve Android platformları temelde MVC'yi baz alır. MVC örüntüsü kolay bir mimaridir ve her seviyeden yazılımcının rahatça çalışabildiği, yazımı ve anlaması kolay mimarilerden biridir. Fazla tecrübeye sahip olmayan yazılımcılar diğer mimarileri daha zor bulduklarından ötürü MVC örüntüsü kullanmayı tercih edebilmektedirler.

Temelinde Model, View ve Controller parçalarından oluşur. Model katmanında veriler tutulur ve View'ın yani görünüm parçasının güncellenmesi model sayesinde sağlanır. View parçasında veri depolanmaz Model parçasından elde edilerek kullanılması sağlanır. Controller View ile Model arasındaki iletişime sağlayan aracı bir parçadır, görünümle alakalı bir şey değişmesi gerektiğinde modeli günceller ve değişimin sunulması talimatını verir [30].

MVC'nin başlıca avantajları şu şekildedir [31];

1. Yapısının basitliği, kolay anlaşılabilmesi neticesinde hızlı yazılım projesi geliştirmeye olanak sağlar. Diğer mimari örüntülerin genel yapısı bu örüntüye kıyas görece daha zordur bu sebeple MVC hızlı geliştirilen projelerde başlıca tercih edilen bir mimari tasarım örüntüsüdür.
2. Her seviyeden yazılımcı çok rahat bir şekilde MVC yapısını anlayabilir ve çalışabilir. Yapısı diğer örüntülerden daha kolay anlaşılabilecek şekilde daha basittir bu sebepten daha az tecrübeli yazılımcılar bu mimaride daha rahat çalışmaktadırlar.
3. Esnek bir örüntü tipidir. Bir değişiklik yapılması gerektiğinde kolayca değişiklik yapılabilir, birden fazla yerde farklı şekilde rahatça kullanılabilir.
4. Herhangi bir model rahatça değiştirilmeden kullanılabilir. Böylece aynı model farklı yerlerde de kullanılabilir.

MVC mimari örüntüsünün dezavantajları şunlardır;

1. Proje karmaşıklaşmaya başladıkça kod miktarı da artar. Artan kod miktarı ve karmaşıklık neticesinde bu mimari daha zor anlaşılabilir ve daha zor değişiklik yapılabilir duruma gelebilmektedir. Çalışmadaki karmaşıklık arttıkça kod satır miktarı daha fazla artmakta ve anlaşılması zorlaşmaktadır. Diğer bazı mimarilerde karmaşıklığa karşı ayrı katmanlarla çözüm bulunmuşken bu mimarideki view ve controller parçalarında işler ayrılmadığından karmaşıklık artmaktadır.
2. Business kod yani iş kodu için ayrılmış bir alan yoktur dolayısıyla iş kodları view controller katmanının içerisinde yazıldığından değişiklik yapmak daha zorlaşır.
3. Zaman içerisinde kod tekrarları bu yapıda daha fazla artmaktadır. Ortaklaştıran belirli bir yapısı olmaması ve karmaşık projelerde tek sınıfta satır sayısı çok fazla artmasından dolayı bir süre sonra yazılımcılar kod tekrarını önleyemez olurlar.
4. Bakım yapılabilirliği sıkıntılıdır. Çünkü proje devam ettikçe karmaşıklaşmakta ve değişikliğe uğramaktadır, bu yapıda karmaşıklık arttıkça yazılımı tekrar okuyup anlayıp değişiklik yapmak daha da zorlaşmaktadır.

MVC örüntüsü hızlıca geliştirilecek basit projeler için ideal bir örüntü tipidir. Özellikle az tecrübeli yazılımcılar ile çalışılıyorsa tercih edilebilecek en ideal örüntülerden biridir. Her ne kadar kolay ve hızlıca proje geliştirilebildiğinden projeyi tamamlamak için gerekli süreyi önemli oranda kısaltsa ve maliyeti de azaltsa da MVC ile yazılan projeler eğer kompleks projeler ise bakım yapılması çok ciddi oranda zorlaşmaktadır. Bu yapıda iş kodları veya yazılımın gerekli diğer öğeleri ayrılmamıştır dolayısıyla controller'daki kod miktarı bir kompleks işlerde diğer mimarilere kıyas

daha fazla olacaktır. Bu mimari kısa sürede yapılacak basit yap unut tipi tek seferde geliştirme yapılacak projelerde kullanılması daha uygundur.

Eğer komplike işlerde MVC örüntüsü kullanılması tercih edilecekse çeşitli destekleyici yapıların da kullanılmasında fayda olacaktır. Destekleyici yapılar özellikle bakım maliyetine önemli katkı sağlayacaklardır, projenin daha kolay anlaşılabilmesine destek vereceklerdir. MVC ile uyumlu çalışabilecek özellikle satın alma gibi işlemlerin yapıldığı birden fazla adımın olduğu birbirinden bağımsız yapılardaki karmaşayı önlemek için MVC ile birlikte mutlaka Coordinator Pattern kullanılmasında fayda vardır.

4.2.4.2. MVP Pattern

MVC mimarisinde business logic ve UI işleri arasında bir ayrım olmaması sebebiyle yoğun kod yığını sorunu ortaya çıkmaktaydı. Ayrıca kod yoğunluğu ve koddaki parçaların birbiriyle olan bağımlılıklarından ötürü MVC örüntüsü zor test edilebilir örüntüydü. MVP örüntüsü ile birlikte View (görüntü) işleri ve diğer işler birbirinden ayrıştırılmış oldu. Böylelikle business logic doğrudan View katmanında yazılmaması dolayısıyla sınıftaki kod sayısının aşırı artması engellenmiş ve daha kolay test edilebilir olması sağlanmış oldu.

Bu örüntü aslında MVC Ek olarak bir Presenter (Sunucu) katmanı eklenerek işlerin ayrıştırılması sağlanmıştır. MVP mimarisi basit bir mimaridir, az tecrübeli yazılımcıların bile kolay adapte olacağı hızlı geliştirme yapılabilecek güçlü bir mimaridir.

MVP'nin 2 farklı yaklaşımı vardır. Bunlar Passive View ve Supervising Contreller olarak geçer. Passive view'da View katmanından Model katmanına doğrudan erişim yoktur. Bütün her şeyi Presenter yönetir, View ve Model'i Presenter katmanı günceller. Supervising Controller'da data binding kullanılabilir böylelikle Model ve View katmanı arasında iletişim mümkün kılınır ve birbirlerini güncellemelerine imkân verilir.

Passive View yöntemi test edilebilirlik açısından çok daha ideal bir yöntemdir. Bu yöntemde View katmanındaki yükün hafifletilmiştir. Supervising yönteminde ise

data binding data çekme işlemlerini de kendi üzerine alabildiği için test edilebilirlik açısından zor ama data binding açısından işleri daha kolaylaştıran bir yöntem olarak öne çıkmaktadır [32].

MVC'ye kıyas biraz daha efor gerektiren bir mimaridir. Her bir View için Presenter'lar da oluşturmak gerekebilir bundan dolayı kod bazı noktalarda biraz daha komplikeleşebilmektedir.

Eğer ortalama bir proje geliştiriliyorsa, hızlı ve daha düzenli bir proje geliştirilmesi isteniyorsa MVP örüntüsü tercih edilebilir. MVC ve MVP örüntüleri arasında karar verilecekse değerlendirmedeki önemli faktör iş kodu yoğunluğu olmalıdır. Eğer basit ve hızlı bir proje geliştirilecekse ve bu projenin business logic (iş mantığı) işleri karmaşıksa MVP örüntüsünü tercih etmekte fayda vardır. Bu senaryonun aksine eğer business logic basitse daha fazla hız kazanmak adına MVC örüntüsü tercih edilebilir.

Daha önceden MVC örüntüsünde bahsedilen Coordinator Pattern bu örüntüyle birlikte de kullanılabilir. Coordinator Pattern kullanımıyla bahsedilen aynı probleme gene aynı şekilde çözüm bulunabilir.

4.2.4.3. MVVM Pattern

MVVM mimarisi ilk olarak John Gossman tarafından 2005 yılında önerilmiştir [33]. Bu yapı WPF ve Windows Phone uygulamalarında sıkça kullanılmaktadır. MVVM projede sorumlulukların ayrıştırılması yani Seperation of Concerns esasına dayalı bir tasarım kalıbı sunmaktadır. Bu yapı ile MVC'deki View Controller katmanındaki yoğunluğun ve MVP yapısında da Presenter katmanındaki yoğunluğun azaltılması sağlanmıştır.

MVVM mimarisi tıpkı MVP gibi işleri soyutlaştırarak birbirinden ayrılabilmesini sağlar. MVVM Model, ViewModel ve View katmanlarından oluşur. View katmanı ekrana görüntü veren işleri çalıştırırken ViewModel katmanı ekrana neyin gösterilmesi gerektiğine dair bilgileri ve talimatları verir. Kullanıcının View ile yaptığı etkileşimler neticesinde ilgili işlerin çalışabilmesini sağlar. View ve Model arasında doğrudan iletişim yoktur. Köprü görevi gören ViewModel aracılığı ile işlerin

yapılması sağlanır. ViewModel View katmanına direkt erişmez ve View ile ilgili hiçbir şey bilmez.

MVVM yapısı observable yapılarla kullanılabilecek en iyi yapılardan bir tanesidir. Binding işlemleri çeşitli yollarla yapılabilir. İster Apple'ın hazır olarak sunduğu Key Value Observing yapısı ile ya da 3. Parti Functional Reactive Programming kütüphaneleri ile istenirse kendin yap yöntemi ile binding işlemleri yapılabilir. Günümüzde RxSwift ve ReactiveCocoa gibi güçlü kütüphaneler sayesinde binding ve observable yapıları gücüne güç katmıştır. Özellikle RxSwift ve MVVM'in birlikte kullanıldığı çok sık görülebilmektedir. MVVM'in avantajlarından daha fazla yararlanabilmek için data binding kapsamlı bir şekilde kullanılmalıdır. Data binding reactive programming yapıları ile birlikte kullanılması ile proje mimarisi çok güçlü ve temiz bir yapıya kavuşacaktır [34].

MVVM yapısının getirdiği çok büyük faydalar vardır. Yapılar birbirinden ayrıştırıldığından birbirlerine daha az bağlı esnek bir yapı oluşturabilmemizi sağlar. Kod yoğunluğunun ve karmaşıklığın azalmasını sağlar. Görsel işler ile business logic birbirinden ayrıştırılmıştır. Bu sayede projede çalışan yazılımcılar birbirinden bağımsız olarak farklı işlerde rahatça çalışabilirler. Bu ayrıştırma test edilebilirliği ve bakım yapılabilirliği kolaylaştırmaktadır. Sorumlulukların ayrıştırılması neticesinde bu mimaride test edilebilirlik ve bakım yapılabilirlik kolaydır. Modelde herhangi bir değişiklik yapıldığı zaman UI tarafında hiçbir değişiklik yapılmadan kullanılabilmesini sağlamaktadır.

Tasarımda çok fazla köklü değişiklik yapılan projelerde pek tavsiye edilmez çünkü böylesi durumlarda View ile birlikte ViewModel'in de değiştirilmesi gerekeceğinden daha fazla zaman harcanmasına sebep olacaktır. View ve kodlarının tekrar kullanılabilirlikleri zayıftır. Bu mimariyi kullanan projelerde tasarımda çok değişiklik yapılması durumunda kod kalitesinin düşürmesi ve daha çok sorun yaşanması ihtimali daha fazladır.

MVVM projenin parçalarını modüllere böler, her ViewModel'de View'a aktarılacak verilerin filtrelenmesi ve işlenmesi işlemi yapılır. Mimari her ne kadar bu konuda bize temiz bir yapı sunmuş olsa da genel olarak yazılımcılar tarafından dikkat edilmeyen sorun bu ayrışan yapı ve modüller sebebiyle kod tekrarların ortaya çıkmasıdır. Tek yerden filtreleme ve birtakım işlemlerin yapılabilmesi mümkün olabilecek iken aynı işin yapıldığı kodların tekrar tekrar yazılmış olduğu görülebilmektedir. Bu tarz problemi çözebilmek için işleri tek yerden yönetme stratejisine gidilmelidir. Bunun için binding'te reactive programming kullanılabilir, Utils Based Pattern kullanılabilir, bazı yerlerde Coordinator Pattern kullanılabilir veya kodları ortaklaştıran bazı sınıflar yazılabilir. Problemlerle ilgili ek detay için Danielhall'ın yazısı incelenebilir [35].

Test edilebilirlik bazen daha zor olabilmektedir. Bazı durumlarda test çalıştırmak için tüm fonksiyonları çağırmak gerekebilmektedir. Debug yapılması daha zor olabilmektedir, tam olarak neyin nereden çağrıldığını bilmek daha zor olduğundan breakpoint konulması gereken yeri tespit etmek daha zor olabilmektedir. Observable yapılar veya reactive programming kullanıldığı zaman debug yapmak daha da zorlaşabilmektedir.

Bu mimari MVP'ye kıyas daha zor bir mimaridir. Daha çok tecrübe gerektirmektedir. İlk geliştirme esnasında maliyeti daha yüksek olabilse de bakım maliyeti daha düşük olacaktır. İyi planlanmış, çok fazla tasarımsal değişikliğe uğratılmayacak, daha yavaş geliştirilebilecek ve tecrübeli ekip üyelerine sahip orta veya büyük projelerde kullanılması daha uygundur. Basit projelerde maliyeti dolayısıyla, hız gerektiren projelerde daha zaman alması ve sık değişikliğe ideal bir yapısının olmaması sebebiyle bu tarz projelerde bu mimarinin kullanılması tavsiye edilmez.

4.2.4.4. MVP-VM Pattern

Bu mimariyi aslında MVP ve MVVM melezi bir mimari gibi görmek mümkündür. Bu iki mimarinin iyi taraflarını toparlayarak ortaya bazı açılardan daha iyi bir mimari ortaya çıkması hedeflenmiştir.

MVP-VM yapısı Model, View, Presenter ve View Model katmanlarından oluşur. View katmanı MVVM'deki gibi hem ViewModel ile hem de MVP yapısındaki gibi Passive View yaklaşımıyla Presenter ile iletişim kurabilir. ViewModel'in yapamadığı işlerin Presenter katmanında yapılabilmesine imkân verir. Presenter katmanının Model, ViewModel ve doğrudan View ile ilişkisi vardır.

Bu yapının en büyük avantajı hem data binding'in kullanılabiliyor olması hem de test edilebilirliğin kolay olmasıdır. Normal MVP yapısında Passive View kullanılması durumunda data binding kullanmak mümkün olmuyordu ancak MVP-VM mimarisi Passive View kullanımı sayesinde MVP ve MVVM'in olumlu yönleri bu mimaride bir araya gelmektedir.

MVVM mimarisinin dezavantajlarından bir diğeri ise ViewModel katmanının gereksiz karmaşıklaşıyor olmasıydı. Bu sebepten test ve bakım yapılabilirlik zorlaşıyordu. Ayrıca ViewModel'de aslında ekranda tuşa basıldı, ekran yüklenince rengini değiştir, business işlerini ilgilendiren diğer işler vb. gibi olması gerekmeyen bazı işlerde ViewModel veya View katmanlarında olabiliyordu. Presenter'ın bu yapıyla gelmesi ile birlikte akışı çok daha verimli yöneten temiz bir yapı oluşmuş oluyor. Bu sayede View ve ViewModel'deki kod sayısı daha azalmış ve basitleşmiş oluyor.

Bir diğer olumlu yanı ise MVVM tasarımsal değişikliklere kolay adapte olamıyordu eğer bu faktöre uygun bir geliştirme yapılırsa Presenter'ın kabiliyetleri kullanılarak ViewController'daki ve View'daki yükün azaltılması ile bu dezavantaj MVP-VM mimarisi sayesinde ortadan kaldırılabilir.

MVP-VM her ne kadar iki mimarinin de avantajlarını kazanmış bir mimari gibi gözükse de bazı sorunlu kısımları da vardır. Bunlardan birincisi internette yeterince doğru kaynak bulunmamasıdır. Tez için yapılan araştırma neticesince farklı farklı kullanımların olduğu görülmüştür.

İkinci dezavantajı ise bazı noktalarda ucu açık kısımların olmasıdır. Mimari her ne kadar Presenter ve ViewModel'in birlikte kullanılabilmesine olanak sağlayan bir yapıya kavuşturmuş olsa da bu iki katmanın hatalı kullanılması mümkündür. Yani

ViewModel'de Presenter'da olması gereken kodların, Presenter'da da ViewModel'de olması gereken kodların yazılması gibi veya Presenter veya ViewModel katmanlarından birinin diğerine kıyas daha zayıf kullanılması gibi ya da katmanlar arası iletişimin doğru yönlü olarak kullanılmaması gibi bazı yanlışlıkların yapılması mümkündür.

Bu mimariyi tecrübeli yazılımcıların kullanılmasında fayda vardır. Şart olmasa da MVVM ve MVP tecrübesine sahip yazılımcıların bu mimariyle çalışmaları, mimariyi doğru kullanmaları adına faydalı bir durum olarak görülebilir. Oluşan sınıf sayısı fazlalığı ve yapının büyüklüğü sebebiyle daha fazla zaman alabilecek bir mimaridir.

4.2.4.5. VIPER Pattern

VIPER mimarisi en yeni mimarilerden bir tanesidir ve mobil platformların gereksinimlerine en uygun mimarilerden biridir. Projeyi daha fazla parçaya bölerek kod yoğunluğunun azaltılmasını ve daha temiz bir mimari elde edilmesini sağlar. Bu mimari delegation driven bir mimaridir. Mimari içerisindeki katmanlar Protocoller aracılığıyla delegasyonla iletişim kurar.

Mimarinin açılımı ve katmanları şu şekildedir: View, Interactor, Presenter, Entity, Router. View katmanı ekranda gösterim yapılması için gerekli işleri yapar ve kullanıcı aksiyonlarını Presenter katmanına iletir. Interactor katmanında business logic işleri yapılır, servislerle iletişim sağlanır, veriler manipüle edilir. Presenter katmanı modülün ana beynidir, tüm diğer katmanlar Presenter ile bağlantılıdır ve Presenter köprü görevi yaparak diğer katmanların birbiriyle iletişimini sağlar. Entity katmanı Interactor katmanında kullanılan modelleri içerir. Router katmanında tüm katmanların Presenter'da bağlanması sağlanıp, sayfanın oluşturulması ve geçiş yapılması sağlanır.

Bu mimari modüllere ayrılarak her modülde farklı katmanlara ayrılır. Yapılacak işler MVP ve MVVM'e kıyas daha fazla farklı parçalara bölünmüş olur ve birbirinden ayrıştırılarak bağımsız çalışmaları sağlanır. Böylelikle test edilebilirlik oranı daha fazla artar. Bir problemle karşılaşıldığında o problemin kaynağını bulmak daha kolay olur çünkü yapı parçalara ayrılmış olduğundan bunun tespitini yapmak

daha kolaydır. Bir projede çalışan birden fazla yazılımcı projenin farklı parçalarına birbirinden bağımsız olarak müdahale edebilir ve aynı anda çalışabilirler. Kodlar ayrılmış olduğundan bir işe yönelik kod yoğunluğu daha azdır. Bu sayede dışardan projeye dahil olmuş başka bir yazılımcı kodları daha rahat okuyup, daha kolay müdahale edebilir. Bu durum bakım maliyetlerini ciddi oranda azaltıp kolaylaştırır.

Ancak bu kadar parçaya ayrılması birtakım dezavantajları da yanında getirir. Çok fazla parça demek çok fazla yaratılması gereken dosya ve yazılması gereken katman demek. Bu da ciddi zaman alıcı bir faktördür. Bir iş test edilmesi gerektiğinde o işi doğrulayabilmek için tüm katmanlardaki ilgili bölümlerinin de test edilmesi gerekmektedir. Bu durum her ne kadar testin daha doğru ve kolay yazılabilmesini sağlasa da harcanacak vakit açısından test maliyetini arttırmaktadır.

Bu mimaride yeni basit değişiklikler yapmak görece daha kolaydır. Modül katmanlara ayrıldığından görünümde değişiklik yapılacağı zaman View katmanında, gelen verinin modeli üzerinde değişiklik olacaksa Interactor katmanında değişiklik yapmak genellikle yeterli olacaktır. Ancak büyük veya köklü değişiklik yapmak çok ciddi zor olabilmektedir. Çünkü tüm katmanlarda değişiklik yapılmasını gerektirecek bir köklü değişiklik olması durumunda ilgili geliştirme bütün katmanlarda yapılacağından daha fazla efor harcatacaktır ve geliştirmesini yapmak daha zor olacaktır.

VIPER mimarisi ile çalışacak yazılımcıların tecrübeli olması gerekmektedir. Diğer mimarilere kıyas çok daha zor bir yapısı vardır. İlk defa bu mimaride çalışmış yazılımcıların mimariye ve projeye alışmaları için zamana ihtiyaçları olacaktır. Mimaride ucu açık bazı kısımlar olabildiğinden bu kısımların tecrübeli kişilerle birlikte tartışılarak netleştirilmesi faydalı olacaktır.

Bu mimariyi kullanan projenin başarıya ulaşması için başlarda çok iyi analizinin yapılması ve projedeki ihtiyaçların net olması gereklidir. VIPER ile geliştirilen proje çok daha fazla zaman harcanmasına sebep olacak ve geliştirilmesi de aynı şekilde daha zor olacaktır bu sebeple değişime adaptasyon süreçleri sancılı olacaktır. Bundan dolayı sonradan değişiklik yapılmasına gerek kalmayacak şekilde projenin planlanması gerekmektedir.

Bu yapı hızlı geliştirilmesi istenen projeler için uygun değildir. Tek bir yazılımcının üstesinden kalkabilmesi daha zordur bu sebeple daha fazla sayıda yazılımcıların birlikte çalıştığı büyük projelerde bu mimarinin tercih edilmesi tavsiye edilir. Mimari zorluğu sebebiyle daha yavaş sürede geliştirilecek büyük projelerde tercih edilmelidir. İlk geliştirme maliyeti her ne kadar daha fazla olsa da projenin ilerleyen süreçlerinde özellikle bakım dönemlerinde ve yeni özellik eklendiğinde toplam maliyeti daha fazla düşürecektir. Ancak küçük projelerde kullanılması durumunda hem daha uzun sürüp daha fazla maliyete sebep olacaktır hem de projenin geliştirmesi daha zorlu olacaktır.

VIPER mimarisi kullanılmaya karar verilirken tüm avantajlı dezavantajlı noktaları değerlendirilmeli ve projeye atanacak ekibin durumu, projedeki ihtiyaçlar, planlar da göz önünde bulundurulmalıdır.

Mimaride işleri kolaylaştırmak adına Sayed Mahmudul Alam'ın önerdiği [36] Generamba, VIPER Code, VIPER Gen gibi tools'lar da kullanılabilir. Böylelikle daha kolay bir şekilde dosyalar yaratılabildiği gibi, daha doğru ve aynı şekilde modüllerin ve katmanların oluşturulması sağlanabilir.

4.2.4.6. Coordinator Pattern

Coordinator örüntüsü isminden de anlaşılabileceği üzere bir şeyleri koordine etme amaçlı tasarlanmış bir örüntüdür. Bilindiği üzere mobil uygulamada pek çok sayfanın birbirleriyle bağlantısı vardır. Örneğin bir listeleme işleminden sonra detay sayfasına gidilmesi, bir ödeme işlemi yapılacağı zaman ilk listeden ödeme seçeneğini seçtikten sonra ödeme bilgileriyle detay ekranının açılması ve en son ödemenin sonuç ekranının açılması gibi bir akış ile çalışan sayfalar mevcuttur. Coordinator örüntüsü bize bunu yapmamızı kolaylaştıran ve işleri daha stabil daha derli toplu yapmamızı sağlayan bir olanak sağlamaktadır.

Coordinator örüntüsü özellikle MVC örüntüsündeki birtakım dezavantajları ortadan kaldırarak kodun daha rahat anlaşılması ihtimalini ve değişiklik yapılabilirliğini arttırmaktadır. Normal bir MVC projesinde bir sayfadan bir başka sayfaya geçiş yapılacağı zaman geçiş yapan sayfa geçiş yapılan sayfanın bilgilerini

biliyor olması gerekliydi. Geçiş yapılan sayfanın bilgileri tutulur ve geçiş yapan sayfa kendi tuttuğu bilgiyi geçiş yapılacak sayfaya ulaştırması gerekirdi. Bu sebepten dolayı kod esnekliğini ve farklı yerde kullanılabilirliğini yitirmekteydi. Coordinator örüntüsü sayesinde bu problem ortadan kaldırılmış oldu. Bu bilgi gönderme, farklı sayfaya geçme işlemlerini Coordinator örüntüsü kullanarak oluşturduğumuz Coordinator sınıfları yönetir. İstenirse bütün akış boyunca belli modeller de Coordinator sınıfları içerisinde tutulabilir böylelikle tek model üzerinden birden fazla sayfada akışın ortak model üzerinden kullanılabilmesi sağlanabilmektedir.

```
class CoordinatorPatternCoordinator
{
    let listPage = CoordinatorPatternListPageViewController()
    var selectedValue: String = ""

    func start() -> CoordinatorPatternListPageViewController
    {
        listPage.coordinator = self
        return listPage
    }

    func goDetail(value: String)
    {
        let detailPage = CoordinatorPatternDetailViewController()
        self.selectedValue = value
        detailPage.coordinator = self
        self.listPage.navigationController?.pushViewController(detailPage, animated: true)
    }
}
```

Şekil 37. Coordinator sınıfı kod örneği

Şekil 37'deki örnekte Coordinator örüntüsü kullanılmıştır. Coordinator sınıfında start ile akıştaki ilk sayfa olan ListPage sayfasının oluşması sağlanmıştır. goDetail metodu ile 2. Sayfa olan detay sayfasına geçiş yapılması sağlanmıştır. Akıştaki iki sayfada da aynı coordinator'ı kullanılmıştır.

```
class CoordinatorPatternListPageViewController: UIViewController {

    var coordinator: CoordinatorPatternCoordinator!

    override func viewDidLoad() {
        super.viewDidLoad()

    }

    @IBAction func value1Tapped(_ sender: Any) {
        self.coordinator.goDetail(value: "Value 1")
    }
    @IBAction func value2Tapped(_ sender: Any) {
          self.coordinator.goDetail(value: "Value 2")
    }
}
```

Şekil 38. Coordinator örüntüsü örneğinde listeleme yapan ilk sayfaya ait sınıfın kodu

Şekil 38'deki örnekte sayfada iki adet tuş bulunmakta ve basılan tuşa göre değer parametrik olarak coordinator'un goDetail isimli metoduna gönderilmekte ve coordinator sınıfında selectedValue isimli değerde tutulmaktadır. Metod çağrılınca detay sayfası oluşturulmakta ve aynı coordinator kullanılmak üzere detay sayfasının sınıfına da atanmaktadır.

```
class CoordinatorPatternDetailViewController: UIViewController {

    @IBOutlet weak var label: UILabel!
    var coordinator: CoordinatorPatternCoordinator!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.label.text = coordinator.selectedValue
    }
}
```

Şekil 39. Coordinator örüntüsü örneğinin detay sayfasına ait sınıfın kodu

Şekil 39’daki detay sayfası örneğinde de görüleceği üzere dışarıdan alınan bir değer yoktur. Sadece coordinator atanmıştır. View yüklendiği zaman label’daki metni değiştirmek için değer coordinator’dan alınmıştır.

Coordinator sınıfı sayesinde kod daha derli toplu hale gelmiş ve daha rahat okunabilir hale gelmiştir. Sayfaların birbirine olan bağımlılığı ortadan kalmıştır böylelikle daha esnek bir yapıya sahip olunmuştur. ViewController sınıfındaki kod kalabalığı daha da azalmıştır.

4.3. Tez Çalışmasıyla Ortaya Çıkan Yeni Tasarım Örüntüleri

Tez çalışması neticesinde bazı tasarım örüntülerindeki veya yazılım projesindeki eksikleri kapatan bazı yeni örüntüler keşfedilmiştir. Bu örüntüler şunlardır;

- Imitate Pattern
- Protected Template Pattern
- Util Based Pattern
- Flex Pattern

4.3.1. Imitate Pattern

Imitate örüntüsü bir oluşturucu örüntüdür. Prototype örüntüsü ile benzerlik taşır. Lakin Prototype örüntüsünden farkı burada bir modeli bütün olarak kopyalamak değil sadece belirlenen kısımların kopyalanmasına izin verir. Bir modelden başka bir model oluşması sağlanmış olur. Gerçek hayatta örnek verilmesi gerekirse bir arabadan esinlenerek orijinaline benzeyen başka bir araba yapılması örnek olarak verilebilir. Bu örüntü ile yaratılan nesnelerde aynı Prototype örüntüsündeki gibi yeni bir instance yaratılarak yeni bir nesne şeklinde oluşturulurlar. Bölümün devamında paylaşılan örneklerde akış detaylandırılmıştır.

```
class ImitatePatternViewController: UIViewController {

    @IBOutlet weak var label1: UILabel!
    @IBOutlet weak var label2: UILabel!
    override func viewDidLoad() {
        super.viewDidLoad()
        // Assume that web services return an object with name groundVehicle:
        let groundVehicle = GroundVehiclesModel(name: "Ford Focus", weight: 2230, wheelCount: 4)

        // Then you need to send kind, name and plate code

        if (groundVehicle.wheelCount == 4), let orderCar = groundVehicle.imitateAs(type:
            OrderCarModel.self) {
            orderCar.plateCode = "ABCD1234"
            orderCar.type = "Car"

            // Print results
            self.label1.text = "Imported Info\nName: \(groundVehicle.name)\nWeight:
                \(groundVehicle.weight)\nWheel Count: \(groundVehicle.wheelCount)"

            self.label2.text = "Exported Info\nName: \(orderCar.name)\nPlate Code:
                \(orderCar.plateCode!)\nType: \(orderCar.type!)"
        }
    }
}
```

Şekil 40. Imitate örüntüsü ViewController sınıfına ait kod

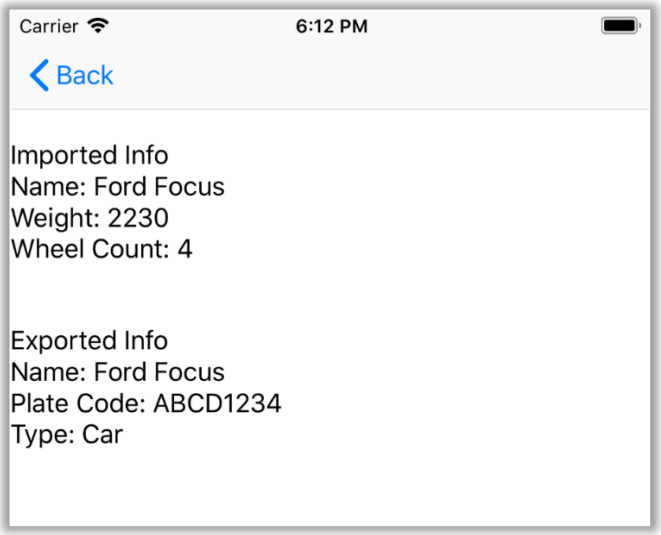
Şekil 40'daki örnekte servisten groundVehicle nesnesi şeklinde bir data alındığını ve servise orderCar nesnesi olarak yeni bir nesne gönderildiğini varsaymıştır. groundVehicle nesnesi GroundVehiclesModel modelini kullanmakta, orderCar ise OrderCarModel modelini kullanmaktadır. Örnek senaryoya göre servisten tipi bilinmeyen bir araç bilgisi gelmekte ve eğer teker sayısı 4 ise araç tipinin araba olduğu tespit edilmektedir. Örnek senaryoda bir müşterinin araç sipariş verdiği varsayılmıştır. Siparişi verebilmesi için araç model ismine, araç tipi bilgisine ve plaka koduna gerek vardır. Elde ettiğimiz groundVehicle nesnesi imitateAs çağrısı ile type parametresinde iletilen generic objeye dönüştürülerek yeni ve farklı tipte bir nesne elde edilmesi sağlanmıştır. Elde edilen nesnede kalan eksi bilgilerden olan plaka kodu ve tip bilgisi sonradan nesneye eklenmiştir. Şekil 41 ve Şekil 42'de uygulanmasına yönelik kodlar ve Şekil 43'te sonuç ekran görüntüsü paylaşılmıştır.

```
class GroundVehiclesModel
{
    let name: String
    let weight: Int
    let wheelCount: Int

    init(name: String, weight: Int, wheelCount: Int) {
        self.name = name
        self.weight = weight
        self.wheelCount = wheelCount
    }

    func imitateAs<T>(type: T.Type) -> T?
    {
        if(T.self == OrderCarModel.self) {
            return OrderCarModel(withGroundVehiclesModel: self) as? T
        } else {
            return nil
        }
    }
}
```

Şekil 41. Imitate örüntüsü örneği için oluşturulmuş GorundVehiclesModel'e ait kod

```
class OrderCarModel
{
    let name: String
    var plateCode: String! = "*
    var type: String! = "*

    init(withGroundVehiclesModel: GroundVehiclesModel) {
        self.name = withGroundVehiclesModel.name
    }
}
```

Şekil 42. Imitate örüntüsü örneği için oluşturulmuş OrderCarModel'e ait kod



Şekil 43. Imitate örüntüsü örneğinin sonuç ekran görüntüsü

4.3.2. Protected Template Pattern

Daha önce Template tasarım örüntüsünde anlatıldığı gibi, Template örüntüsünde süper metodun çağrılmasının unutulması probleminin giderilmesi için Protected Template örüntüsü oluşturuldu. Basitçe override edilen metot da üst sınıftan çağrım unutulması riskine karşın mutlaka çalışması gereken kısımların ayrı bir private metot ile kullanılması ile bu riski ortadan kaldırılması sağlanmaktadır.

Şekil 44'teki örnekte workSome metodu UUID elde etmekte bunu bir label'a basmaktadır. Burada Template örüntüsü örneği paylaşılmıştır, görüldüğü üzere super metot çağrılmamış ve başka bir yerden workSome metodu çağrılmamıştır.

```
class ProtectedTemplatePatternViewController: ProtectedTemplatePatternBaseViewController {

    @IBOutlet weak var label: UILabel!

    override func workSome(uuid: String) {
        //Assume that developer forgot to call super function of this function
        //Even he/she is forgot, function works correct
        self.label.text = "UUID is: \(uuid)"
    }
}
```

Şekil 44. Protected Template Pattern örneğinde asıl çalışan sınıfa ait kod

Şekil 45'teki örnekte görüleceği üzere base alınmış bu sınıfta Template tasarım örüntüsünün dezavantajını gidermek için Protocted örüntüsü kullanılmıştır. workSome metodu alt sınıftan erişilebilen bir metottur ve alt sınıfların bu metodu override ederek kullanabilmesi sağlanmıştır. Bu metottaki kod opsiyoneldir sadece güvenlik amaçlı açık bırakılmış ek metot görevini görür. Asıl işi yapan metot protectedWorkSome metodudur. Bu metoda alt sınıftan erişilemez, yanlışlıkla super çağrısının yapılmasının unutulması gibi hatalar sorun yaratmaz çünkü metot super çağrısı beklemeden kendi görevini yerine getirmektedir.

```
class ProtectedTemplatePatternBaseViewController: UIViewController {

    private var alluuid: String?

    override func viewDidLoad() {
        super.viewDidLoad()
        self.protectedWorkSome(uuid: nil)
    }

    private func protectedWorkSome(uuid: String?)
    {
        if(uuid == nil)
        {
             self.alluuid = NSUUID().uuidString
        }

        if(self.alluuid != nil)
        {
            self.workSome(uuid: self.alluuid!)
        }
    }

    func workSome(uuid: String)
    {
        print("Extra log - not neccassary to call this!")
    }
}
```

Şekil 45. Protected Template örüntüsü örneğinde kalıtım alınan üst sınıf

4.3.3. Utils Based Pattern

Util Based örüntüsü Türkçe ismi ile Hizmet Odaklı Örüntü yazılımda sıkça kullanılan bölümleri ortaklaştırmaya, kodu basitleştirmeye, kod tekrarını engellemeye, kodun esnekliğini arttırmaya, tek merkezden değişiklikle yönetilebilmesine odaklanır. Bu örüntü mimari tipi bir tasarım örüntüsüdür. Ayrı katmanlara sahip mimariler gibi

bu örüntüde de Util ayrıca bir katmandır. Bu katman projenin herhangi bir yerinden kolayca erişilebilir ve hizmet sağlanabilir şekilde ayarlanmalıdır.

Bir projedeki tüm modüller Utils kodlarına rahatça erişebilirler. Bu katman projenin ana omurgasında ayrı bir katmandır. İhtiyaca göre fonksiyonları static olarak tanımlanıp rahatça çağrılması sağlanabilir. Singleton nesne sayesinde erişim de sağlanabilir ya da normal sınıf instance yaratılarak da kullanılabilir. Buradaki önemli kural çağrım şeklinin o sınıf içerisinde aynı olmasıdır. Her sınıfın kendine göre belli bir düzeninin olması beklenir.

Bu katmanda business işlerine ait kodlar barındırılabileceği gibi iş kodundan ziyade yazılımın ihtiyaç duyduğu kendini tekrar eden yapıların, sıkça kullanılacak kodların veya tek merkezden yönetilmesi daha doğru olan önemli kodların burada kullanılması beklenmektedir.

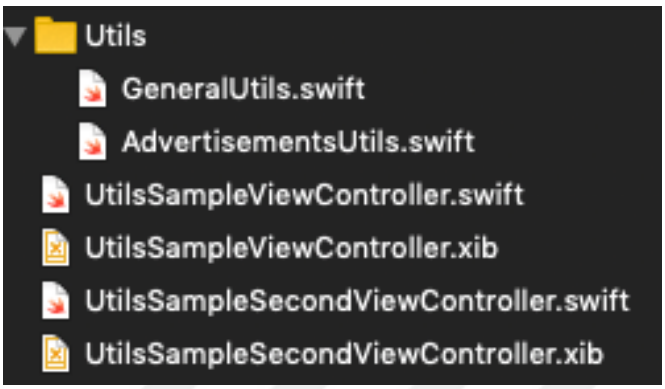
Oluşturulacak Utils sınıfının ismi ihtiyaca göre belirlenir. Örneğin reklam yönetimi için AdvertisementUtils kullanılırken, bir resim düzenleme işleri ile alakalı şeyler ImageUtils olarak tanımlanabilir, genel kodlar ise Utils olarak tek başına da tanımlanabilir.

Utils sınıfı istenirse Manager olarak da isimlendirilebilir. Ancak bu katmanda işimiz daha ziyade işlerimizi kolaylaştırmak ve bu katmanda yapılacak iş bir konudan bağımsız herhangi bir iş olabileceğinden Utils olarak tanımlamak daha sağlıklı olacaktır.

Bu katmanda yazılan kodların çok uzun olabileceği var sayılmaktadır. Dolayısıyla mümkün olduğunca basitleştirerek ve okunabilirliği arttırarak kod yazılmalıdır. Metot isimleri rahat anlaşılabilir olmalı ve kullanımları kolay olmalıdır. Belli işler kendi içlerinde gruplandırılmalıdır. Sınıf kodunda mümkün olduğunca mark yani işaret kullanılarak iş gruplarını ayırmak suretiyle istenildiğinde gerekli koda daha kolay ulaşılabilmesini sağlanabilmesi önerilir.

Bu örüntü, diğer bütün mimari tasarım örüntüleri ile birlikte rahatça kullanılabilir. Proje fark etmeksizin tüm projelerde ve platformlarda kullanılabilir. Temelinde basitçe ayrı bir klasörde Utils sınıflarını barındırıp önemli ve sıkça

kullanılacak veya tekrar edebilen kodları burada barındırıp kullanılması üzerine bir mantıkla kullanılmaktadır. Önemli veya tekrar eden kodların yanı sıra bütün modül, sayfa veya sınıfların ortak tek bir merkezden istenilen özelliği kullanabilmelerini sağlamayı hedefler.



Şekil 46. Utils Based Pattern proje dosya konumlandırması örneği

Şekil 46'daki ekran görüntüsünde görüleceği üzere örnekte Utils olarak ayrı bir klasör oluşturulmuştur. Burada genel işler GeneralUtils, uygulama içerisindeki reklamlarla ilgili işler ise AdvertisementsUtils üzerinden yönetilmektedir. Uygulamada ilk açılan ekranın UtilsSampleViewController olduğu varsayılmış, ikinci açılan ekran yani yönlendiren ekranın UtilsSampleSecondViewController olduğu varsayılmıştır.

```
class GeneralUtils
{
    static func log(_ string: String)
    {
        #if DEBUG
        print("GeneralUtilsLog-->> \(string)")
        #endif
    }

    static func getViewController<T:UIViewController> (_ type: T.Type, storyboard: Storyboards,
        customIdentifier: String? = nil) -> T
    {
        let storyboard = UIStoryboard(name: storyboard.rawValue, bundle: nil)
        let identifier = customIdentifier != nil ? customIdentifier! : String(describing: T.self)
        let vc = storyboard.instantiateViewController(withIdentifier: identifier) as! T
        return vc
    }

    static func goToViewController(from: UIViewController, to: UIViewController) {
        if(from.navigationController != nil) {
            from.navigationController?.pushViewController(to, animated: true)
        } else {
            from.present(to, animated: true, completion: nil)
        }
    }
}
```

Şekil 47. GeneralUtils isimli Utils Based Pattern örneği sınıfı kodu

Şekil 47'deki örnekteki genel hizmet sınıfında yani GeneralUtils sınıfı içerisinde uygulamanın temelde ihtiyaç duyduğu bazı kodlar yazılmıştır. Örnekte log yazmak için, bir Storyboard'dan sınıf elde etmek için ve bir ViewController sınıfından sayfaya geçmek için fonksiyonların yazılmış olduğu görülmektedir. Bu sınıftaki amacımız uygulamadaki temel operasyonların her sınıftan rahatça ortak bir şekilde erişilebilmesini sağlamaktır. Ayrıca bir değişiklik yapılması gerektiği zaman tek yerden yapılan değişiklikle hızlıca düzenlemeler yapılabilmektedir.

```swift
class AdvertisementsUtils {
    static let shared = AdvertisementsUtils()

    private init() {
        self.loadAllAds()
    }

    private func loadAllAds(){
        self.reloadInterstitialAds()
    }

    func reloadInterstitialAds() {
        // Code Here
    }

    func reloadBannerAds() {
         // Code Here
    }

    func showInterstitialAds() {
         // Code Here
    }

    func showBannerAds() {
         // Code Here
    }
}
```

Şekil 48. AdvertisementsUtils isimli Utils Based Pattern örneği sınıfı kodu

Şekil 48'de paylaşılan AdvertisementsUtils sınıfında uygulama içerisindeki reklamların yönetilebilmesi sağlanmıştır. Singleton sayesinde içerisindeki nesnelerin tek seferde yüklenmesi sağlanmıştır. Böylelikle önceki ekranda yüklenmiş bir reklam. Bu sınıf sayesinde ikinci ekrana geçildiğinde çok kolay bir şekilde reklamın gösterilebilmesi sağlanmıştır.

```swift
class UtilsSampleViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        AdvertisementsUtils.shared.showBannerAds()
    }

    @IBAction func goToSecondPage(_ sender: Any) {
        GeneralUtils.goToViewController(from: self, to: UtilsSampleSecondViewController())
    }
}
```

Şekil 49. Utils Based Pattern örneğinde ilk sayfayı çalıştıran ViewController sınıfı kodu

Şekil 49'daki UtilsSampleViewController sınıfındaki örnekte görülebileceği gibi ekran oluşturulduğunda banner reklam görüntülenmesi sağlanmış ve 2. sayfaya geç tuşunun bağlı olduğu action olan goToSecondPage aksion fonskiyonunda tuşa basıldığında 2. sayfaya kolayca geçilebilmesi sağlanmıştır. Buradaki en önemli unsur aynı kodların tamamen farklı sınıflarda da rahatça kullanılabilir olmasıdır.

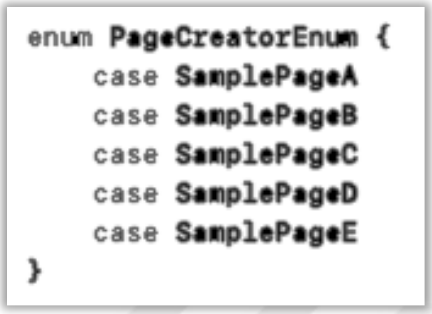
```
class UtilsSampleSecondViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        AdvertisementsUtils.shared.showBannerAds()
        AdvertisementsUtils.shared.showInterstitialAds()
    }
}
```

Şekil 50. Utils Based Pattern örneğinde ikinci sayfayı çalıştıran ViewController sınıfı kodu

Uygulamanın başka bir sayfadan geçiş yapılmak suretiyle açılan 2. Sayfasına ait UtilsSampleSecondViewController sınıfındaki örnekte görülebileceği üzere banner ve geçiş reklamlarının uygulamada açılması sağlanmıştır. 1. sayfadaki açılan banner'ın AdvertisementsUtils sayesinde 2. sayfada da aynı şekilde açılması sağlanmıştır. Bu sayfada diğer sayfaya göre bir farklılık yapılmış ayrıca geçiş reklamı da açılmıştır.

Örneklerden de anlaşılabileceği gibi Util Based Pattern işimizi çok önemli miktarda kolaylaştırmakta, tek merkezden yönetilmesi sayesinde herhangi bir değişiklik yapılması gerektiğinde veya düzeltme yapılması gerektiğinde bunun çok kolay ve hızlıca yapılabilmesini sağlamaktadır. Utils sınıfları başka sınıflardan bağımsız çalışırlar ve bu sınıfların fonksiyonlarını çağıran herhangi bir sınıf rahatça fonksiyonların çalışmasını sağlayabilir. Bu örüntü ile Utils sınıflarının projenin temelini oluşturması hedeflenmektedir. Bu örüntünün kullanılması ile proje geliştirme ve bakım maliyetlerinde çok ciddi düşüş olacaktır. Projeye yeni dahil olan kişilerin projeyi anlamasını ve çalışma yapmalarını çok ciddi oranda kolaylaştıracaktır.

Utils Based Pattern N defa tekrar edebilecek kodların sayısını sabitlemeye ve azaltmaya yardımcı olur. Bir ekran oluşturan PageCreatorUtils isimli bir Utils sınıf oluşturulduğu varsayılsın. Örnek için SamplePage ile başlayan A'dan E'ye kadar 5 sayfa oluşturabilecek PageCreatorEnum Şekil 51'deki gibi oluşturulmuştur.



Şekil 51. General Utils PageCreatorUtils için PageCreatorEnum kodu

Örneğe göre sayfalarda banner ve interstitial (geçiş) olmak üzere 2 tip reklam gösterebilme ve sayfa açıldığına dair istatistik gönderebilme imkânı olacaktır. Lakin her sayfada istenilen özellikler kullanılacaktır. Örneğe göre "AdvertisementsUtils.shared.showBannerAds()" komutuyla çağrılan metotta banner reklam gösterebilmek için 10 satır, "AdvertisementsUtils.shared.showInterstitialAds()" komutuyla çağrılan metotta interstital reklam gösterebilmek için 20 satır, "GeneralUtils.sendPageOpenStatistics(screenName: "Ekran İsmi", date: Date())" şeklinde çağırılan metotta ise istatistik bilgisi göndermek için 5 satır kod yazılmış olduğu var sayılsın. GeneralUtils ve Adverisements sınıfları ve içerisindeki metotlar örnekte Utils Based Pattern kullanılarak oluşturulmuş sınıf ve metotlardır. Klasik yöntemde yazılımcılar sayfaların sınıflarına bu kodları yazmaktadır. Şekil 52'de gene Utils Based Pattern kullanılarak bu kodların tekrar tekrar yazılması önlenmiş, tek yerden yönetilmesi sağlanmış ve sonuç olarak kod tekrarları önlenmiştir. Utils Based Pattern'in kod satırına katkısı N kadar olsa da örnekte 5 sayfa için klasik yöntemle ve Şekil 52'de Utils Based Pattern kullanılarak yazılan kod satır sayısı Tablo 4'te paylaşılmıştır. Tablodaki kod satır sayıları arasındaki farka bakılacak olursa, kod satır sayısı ciddi azaldığından bu örüntünün örnekteki yazılım projesine katkısı büyük

olmuştur. Çok tekrar eden kodların tek yerden yönetilmesi, ileride değişiklik yapılması durumunda harcanacak süreyi ve riski ciddi azaltacaktır.

```
class PageCreatorUtils {
    static func createPage(page: PageCreatorEnum) -> UIViewController {

        switch page {
        case .SamplePageA:
            let vc = GeneralUtils.getViewController(SamplePageAViewController.self,
                storyboard: .main)
            GeneralUtils.sendPageOpenStatistics(screenName: "Utils Based Sample A Page",
                date: Date())
            AdvertisementsUtils.shared.showBannerAds()
            AdvertisementsUtils.shared.showInterstitialAds()
            let presentedVC = self.createPage(page: .SamplePageB)
            GeneralUtils.goToViewController(from: vc, to: presentedVC, present: true)
            return vc
        case .SamplePageB:
            let vc = GeneralUtils.getViewController(SamplePageBViewController.self,
                storyboard: .main)
            GeneralUtils.sendPageOpenStatistics(screenName: "Utils Based Sample A's Popup B
                Page", date: Date())
            return vc
        case .SamplePageC:
            let vc = GeneralUtils.getViewController(SamplePageCViewController.self,
                storyboard: .main)
            AdvertisementsUtils.shared.showBannerAds()
            AdvertisementsUtils.shared.showInterstitialAds()
            return vc
        case .SamplePageD:
            let vc = GeneralUtils.getViewController(SamplePageDViewController.self,
                storyboard: .main)
            GeneralUtils.sendPageOpenStatistics(screenName: "Utils Based Sample D Page",
                date: Date())
            AdvertisementsUtils.shared.showBannerAds()
            return vc
        case .SamplePageE:
            let vc = GeneralUtils.getViewController(SamplePageEViewController.self,
                storyboard: .main)
            GeneralUtils.sendPageOpenStatistics(screenName: "Utils Based Sample E Page",
                date: Date())
            AdvertisementsUtils.shared.showInterstitialAds()
            return vc
        }
    }
}
```

Şekil 52. Utils Based Pattern örneğinde PageCreatorUtils sınıfında createPage metodu

Tablo 4. Örneğe göre klasik şekilde yazılan kodlar ile Utils Based Pattern kullanılarak yazılan kodlar arasında kod satır sayısı farkı

	Klasik					Utils Based Pattern				
	A	***B***	***C***	***D***	***E***	***A***	***B***	***C***	***D***	***E***
Banner Reklam Gösterme	10	0	10	10	0	1	0	1	1	0
Interstitial Reklam Gösterme	20	0	20	0	20	1	0	1	0	1
İstatistik Gönderme	5	5	0	5	5	1	1	0	1	1
İşle Alakalı Sayfa İçinde Kod Toplam	35	5	30	15	25	3	1	2	2	2

Yukardaki Tablo 4'te banner reklam gösterme için 10, interstitial reklam gösterme için 20, istatistik gönderme için 5 satır kod ile geliştirme yapılması gerektiği varsayılmıştır. Örneğe göre klasik yöntemde geliştiriciler kodları her satırda tekrarlı şekilde A, B, C, D ve E isimli sınıflara yazmışlardır. Utils Based Pattern kullanılan örnekte ise sadece işi yapan Utils sınıf ve metodlar çağrılmıştır. Utils Based Pattern kullanılarak sınıflarda tekrar kod yazılması engellenmiştir, yapılacak işlere ait kodlar A, B, C, D ve E sınıflarında değil Utils sınıflarına yazılacaktır dolayısıyla ilgili işlerin yapılabilmesi için yazılması gereken kod satır sayısı Utils sınıfı içerisinde tek seferde yazılır. Tablo 4'teki tüm sınıfların olduğu bir örnekte belirtilen işler için klasik yöntemle 110 satır kod yazılmışken, Utils Based Pattern kullanılarak sadece 10 satır ile Utils sınıf ve metodları çağrılmış toplamda 45 satır kod yazılmıştır.

4.3.4. Flex Pattern

Tez çalışması neticesinde günümüzde kullanılan örüntüler araştırılmıştır. Araştırma ile tasarım örüntülerinin olumlu ve olumsuz yönleri detaylandırmıştır. Mimari tasarım örüntüleri ise birbirleri ile kıyaslanmıştır. Örüntülerdeki eksikliklerin hangi yöntemlerle giderilebileceğinden bahsedilmiştir.

Flex Pattern tez araştırması neticesinde oluşmuş Türkçe ismi "Esnek Örüntü" olan tümüyle yeni bir mimari örüntüdür. MVP, MVVM, MVP-VM ve VIPER mimari

örüntülerinden ilham alınan kısımları vardır, tüm bu örüntülerin olumlu yönlerinden faydalanmış ve dezavantajlı kısımları bu yeni örüntü ile giderilmiştir.

Flex örüntüsü adından da anlaşılabileceği gibi projeye esneklik katan bir örüntüdür. View, Module ve Data Layer birbirinden bağımsız çalışırlar ve farklı yerlerde tekrar tekrar kullanılabilirler.

Ekranda oluşturulan bir sayfa istenirse başka bir sayfada tekrar kullanılabilir. Bunun için tekrar sayfayı oluşturmak gerekmemektedir. Sayfanın tekrar kullanılabileceği gibi ekrandaki görüntü bileşenleri (view components) tekrar tekrar aynı veya farklı yerlerde kullanılabilirler. Ekranda sayfayı oluşturan tüm view'lar (görüntüler) bağımsız parçalardır ve istenilen yerlerde kolayca kullanılabilirler. Bu örüntüde View izoledir, dışarıdan talimat alarak kullanılabilirler lakin doğrudan dış parçaya bağımlı olarak çalışmazlar böylelikle bağımsız ve esnek bir şekilde çalışmaları sağlanmıştır.

Mobil uygulamanın iletişimde olduğu Data Layer katmanında tümüyle izole bir katmandır. Bu katmandaki modellere doğrudan erişilemez ve modellerde değişiklik yapılamaz. Servisten data çekildiği anda veya servise bir şeyler iletileceği zaman modeller oluşur ve kullanılır. Data Layer sadece kendi bildiği modelleri kullandığından ve dışardan kendisine ait olmayan bir veri alıp kullanmadığından başka bir katmana bağımlılığı yoktur. Böylelikle aynı Data Layer farklı bir modülle de çalışabilir. Bu sayede aynı iş için tekrar kod yazılmamış olur, değişiklikler kolayca uygulanılır ve bir problem olması durumunda hata kolayca tespit edilir.

Projedeki modül kısmı projeyi yöneten ve iletişimi sağlayan asıl kısımdır. Modülde diğerleri gibi esnek çalışabilmektedir. Modül istediği Data Layer ile ve istediği View'ları kullanabilir. Ancak bu kısım diğerleri gibi izole bir kısım değildir, Data Layer ve View ile iletişim kurabilmek için kullanacağı katmana nasıl talimat vermesi gerektiğini ve yanıtı nasıl alacağını bilmesi, belirlemesi gerekir.

Bu örüntü projedeki her bir parçanın ve kodun esnekliğe sahip olmasını hedefler. Projenin herhangi bir yerinde kullanılan parça, kod veya sayfa başka bir yerde tekrar kullanılabilmelidir. Her ne kadar ilk planda bir parçanın tek bir yerde

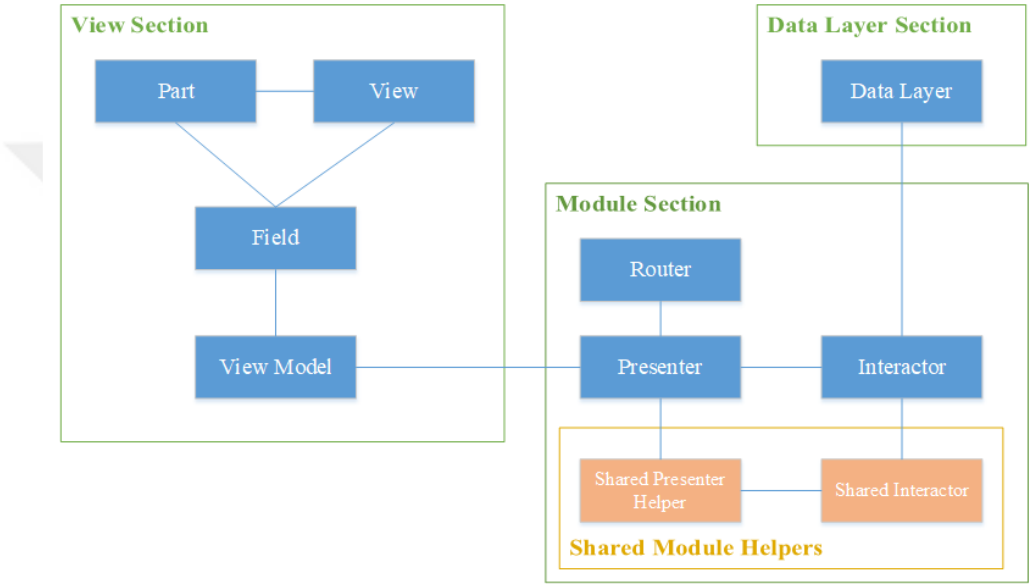
kullanılabileceği bariz olsa da ilerleyen sürümlerde tekrar kullanımı söz konusu olabilmesinden tekrar kullanılabilirlik çok önemlidir. View'lar parçalara bölünür, her bir parça başka bir parça ile birleştirilerek yeni bir view oluşturulabilmelidir. Aynı kodların tekrar tekrar kullanılmasını engellemeyi amaçlar. Bölümler arasında izolasyon sağlayarak başka bir bölüm ya da parçaya bağımlı kalınması engellenir böylece projedeki her parçanın başka yerlerde bağımsız bir şekilde kullanılması hedeflenir. Her bölümün kendine has sorumlulukları vardır, bu bölümler ve sorumluluklar sayesinde kod yoğunluğu belli başlı sınıflarda artmak yerine bölümlere dağılır. Bu sayede daha anlaşılır, bakımı kolay projelerin geliştirilmesi hedeflenir.

Flex Pattern esasları şunlardır;

- Parçalar izole çalışabilmelidir.
- Bölümler birbirinden bağımsız çalışabilmeli ve birbirlerine ait olmamalıdırlar.
- Diğer bölümler modül bölümüne ait değildir dolayısıyla sadece modüle göre yapılmamalıdır ve gruplandırılmamalıdır. Başka modüller ile de çalışabilmesi gerekmektedir.
- Bir parça başka bir yerde tekrar kullanılabilmelidir.
- Kod esnek olmalıdır. Değişim kolay uygulanmalı ve farklı yerlerde rahatça kullanılabilmelidir.
- Bölümlerin belirli sorumlulukları olmalıdır ve bu sorumluluklara göre çalışmalıdırlar.
- Kod tekrarları azaltılmalıdır. Tekrarları önlemek için kodun ortak çalışmasını sağlayan yapılar kullanılabilmelidir.

Bu örüntü View, Part, Field, View-Model, Presenter, Interactor, Data Layer ve Router parçalarından oluşur. Bu katmanlara ek Presenter ve Interactor'daki kod tekrarlarını engellemek adına SharedPresenterHelper ve SharedInteractor parçalarından ibaret 2 adet de destekleyici parça farklı destekleyici modüllerde bulunabilir. Görüntü Bölümü (View Section), Modül Bölümü (Module Section) ve Veri Katmanı Bölümü (Data Layer Section) olmak üzere 3 bölümden oluşur. View Section ile alakalı bölüm View, Part, Field ve ViewModel'den oluşur. Modül bölümü

ise Router, Presenter ve Interactor'dan oluşur. PresenterSharedHelper ve SharedInteractor modül bölümüne bağlıdır ve Presenter ile Interactor'ü destekler. View-Model; View ve Module ile alakalı işlerle alakalı bağlantıyı sağlayabilmek için Presenter ve Field'ı birbirine bağlar. Data Layer bölümü ise sadece Data Layer'dan oluşan 3. bölümdür. Module ve Data Layer işleriyle alakalı bağlantı Interactor ve Data Layer sınıfı iletişimi ile sağlanır. Router ile bir sayfanın çalışması için gerekli muhatapların atanması sağlanır.



Şekil 53. Flex Pattern şeması

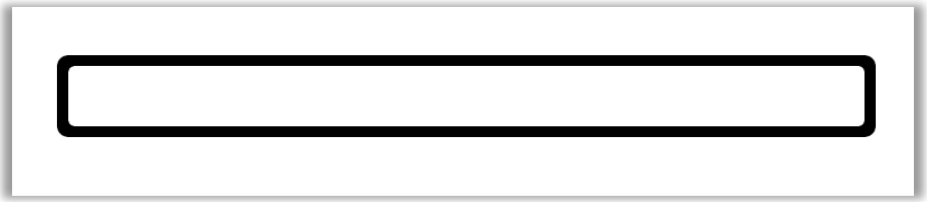
Görüntü bölümünün temel bileşeni Field'dır. Field aslen bir sayfayı veya sayfadaki bir bölümü temsil edebilen bir ViewController'dır. Field içersinde View ve Part'lardan oluşan nesneler tutulur. Sayfadaki bileşenlerden bağımsız olarak esnek bir şekilde eklenebilmektedir. Field'a benzetme yaparsak bir inşaata benzer. İnşaat alanında bir saha vardır, saha içine tuğla ve beton bloklarla inşaat yapılarak en sonunda bina meydana gelir. Aslına bakacak olursak, bir bina bir saha üzerinde birleştirilmiş pek çok parçanın bir araya gelmesinden oluşmuş bir yapıdır.

Gerçek örneğindeki gibi bu örüntü de de bir inşaattan esinlenilmiştir. Saha örüntüde Field olarak temsil edilmektedir. Bu sahanın üstüne konulan tuğlalar View, birden fazla tuğladan oluşmuş blok ufak parçalar ise Part olarak temsil edilmiştir.

Field'da eklenen View ya da Part parçaları Field'dan izole çalışmalıdır. Hiçbiri Field'a ViewModel'e ya da herhangi bir şeye bağımlı olmamalıdır. Hatta Field ve View parçaları da birbirinden bağımsız bir şekilde çalışmak zorundadır. Burada temel mantık herhangi bir parça herhangi bir yerde kolayca çalışabilmeli esasına dayanır.

Part birden fazla view'dan oluşan parçaları hazır bir yapı bloğu gibi bir arada kullanılabilmesini sağlayan parçadır. Part'ın içinde birden fazla farklı Part veya birden fazla View ile birlikte birden fazla Part'ın olduğu yapıyı da içerebilir. Part parçası ile asıl amaç tekrar tekrar aynı View'ları yaratmak veya kopyalayıp aynen kullanmak yerine farklılaşması durumunda aynı kalan kısımları tek yerden alıp ortak kullanılması esasına dayanır. Böylelikle aynı kalan kısımlar tek yerden müdahale ile her yerde aynı şekilde değiştirilebilmesi sağalabildiği gibi sorunlar yaşandığı zaman sorunu bulup düzeltme imkânı artmaktadır.

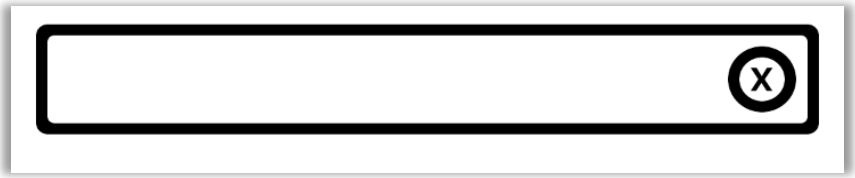
Şekil 54'deki örnekteki gibi bir TextFiledView olduğunu düşünülürse. Bu view basit bir TextField görünümüne sahip basit bir view'dır.



Şekil 54. Flex Pattern örneğindeki TextFieldView

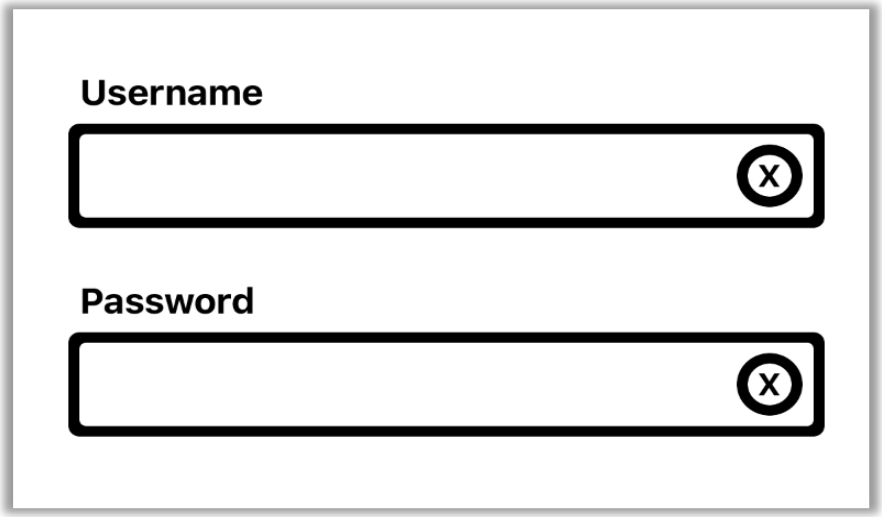
Örnekte bazı TextField'ların X tuşu ile TextField üzerinde yazılı olan tüm yazının silinebilmesine olanak sağlayan bir tuşunda olması istenildiği varsayılsın. Bu tarz yapılar geleneksel olarak yapılmaya çalışıldığında doğrudan mevcut TextField'ın kopyasından oluşturulan yeni bir View kullanılarak yapılmaktadır. Ancak Pattern'in gerekli olarak sunduğu yapıda ve tezde örneklenen yapıda Part parçası ile X tuşunu ve TextField'ı birleştiren yeni bir View oluşturabilmemiz sağlanmış ve Şekil 55'te ekran

görüntüsü paylaşılan bu yeni Part parçası olan yapıya TextFieldWithXButtonPartView ismi verilmiştir. Böylece bir şeyin üzerine yeni bir şey üretilmesi sağlanmıştır.



Şekil 55. Flex Pattern örneğindeki TextFieldWithXButtonPartView

Sonra birde üzerine Label olan ve X tuşuna sahip TextField'larında olması istenildiği varsayılsın. Bu bileşeni kullanan bir login (giriş) sayfası oluşturulması istenildiği varsayılsın. Bu isteği yerine getirebilmesi için yapılabilecekler basittir yeni bir TextFieldWithXButtonAndTopLabelPartView isimli Part parçası oluşturulur ve içerisine TextFieldWithXButtonPartView Part parçası eklenir. Böylelikle Part içerisinde Part eklenmesi yoluyla kopyalama yapmadan ortak tek yerden aynı parçaların farklı şekilde Şekil 56'daki gibi yeni bir Part oluşturularak kullanılması sağlanır.



Şekil 56. Flex Pattern örneğindeki TextFieldWithXButtonPartView

View, Field ve Part parçaları MVC örüntüsü baz alınarak çalışırlar. Flex örüntüsü sayesinde View katmanındaki kod yoğunluğu ve zorluğu daha az

olacağından bu bölümlerin MVC ile çalışması bir dezavantaj oluşturmaz. Ayrıyeten MVC ile çalışması birçok açıdan avantaj sağlayacaktır. Çünkü bu yapı sayesinde View'lara hızlıca kod yazılabilecektir. İzolasyon için Protocol kullanımı şart değildir, hızlı geliştirme istenirse Protocol kullanılmadan da etkili bir şekilde kullanılabilirler. View'lara ait sınıflar gerekli olması durumunda Model'e sahip olabilirler ancak gerekmediği durumlarda ayrıca Model dosyaları oluşturmak gereksizdir, basit nesneler View'ın sınıfı içerisinde tutulabilirler.

Field ve Part parçalarına bileşenler programattic veya 3.2.3. Etkin Arayüz Oluşturucu Araçlar Kullanımı ile alakalı bölümde detaylandırıldığı gibi Interface Builder üzerinden eklenerek kolayca kullanılabilirler.

```
@IBDesignable
class TextFieldWithXButtonAndTopLabelPartView: XibView {

    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var textField: TextFieldWithXButtonPartView!

    @IBInspectable var labelText: String = "" {
        didSet {
            self.titleLabel.text = self.labelText
        }
    }

    @IBInspectable var textFieldText: String? {
        get {
            return self.textField.textFieldText
        }
        set {
            self.textField.textFieldText = newValue
        }
    }

    @IBInspectable var isPasswordTextField: Bool = false {
        didSet {
            self.textField.isPasswordTextField = self.isPasswordTextField
        }
    }

    override func xibInitialized() {
        super.xibInitialized()
    }
}
```

Şekil 57. Flex Pattern örneğindeki TextFieldWithXButtonAndTopLabelPartView sınıfı kodu

Şekil 57'deki kod örneğinde TextFieldWithXButtonAndTopLabelPartView isimli Part parçamızın sınıfına dair kod paylaşılmıştır. Önceki örneklerde X butonuna sahip bir text field ile label'dan oluşan bir Part oluşturulmuştu. Şekil 55'teki kodda da görülebileceği üzere X tuşlu text field nesnesi textField isimli değişkende tutulmuş ve bir de titleLabel ismindeki değişkenle text field üstündeki label tutulmuştur. Text field'ın özelliklerinden olan yazısının değişimine ve text field'ın şifre tipinde olup olmayacağına textFieldText ve isPaswordTextField isimli getter ve setter'dan oluşan değişkenler ile müdahale edilebilmekte ve bilgi alınabilmektedir.

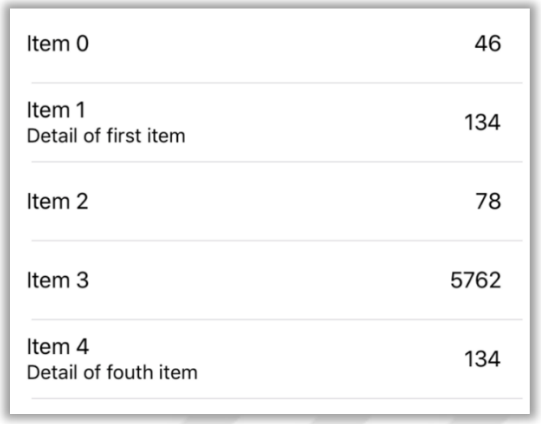
Örüntüde izolasyon hususu önemli bir husustur. Yukarıdaki örnekte de görülebileceği üzere eklenen view'lar eklenildikleri view'a bağımlılıkları yoktur. Bir view başka bir view'a eklendiği zaman ekleyen view'dan talimatları bağımsız bir şekilde alıp, bağımsız bir şekilde çalışırlar. Örüntünün eklenildikleri nesneyi bilip buna uygun dönüş yapmaları bir şey söz konusu bile değildir. Örüntünün katı kuralı izolasyonun mutlak suretle sağlanması yönündedir. Aşağıdaki TableView örneğinde konu daha da netleştirilmiş olacaktır.

```
@IBDesignable
class SampleTableView: XibView {

    @IBOutlet weak var tableView: UITableView!
    var model = [SampleTableViewModel]()

    override func xibInitialized() {
        super.xibInitialized()
        self.tableView.register(UINib(nibName: "SampleThreeValueTableViewCell", bundle: nil),
            forCellReuseIdentifier: "SampleThreeValueTableViewCell")
        self.tableView.register(UINib(nibName: "SampleTwoValueTableViewCell", bundle: nil),
            forCellReuseIdentifier: "SampleTwoValueTableViewCell")
    }

    func reloadTableView() {
        self.tableView.reloadData()
    }
}
```

Şekil 58. Flex Pattern örneğinde SampleTableView kodu

Şekil 58'de SampleTableView isimli bir TableView yaratılmıştır. Daha doğrusu bir View üzerine eklenmiş TableView'dan oluşan bir View yaratılmıştır. Bu view TableView'ı doldurabilmek için dışarıdan SampleTableViewModel isimli modele sahip dizin almakta ve model değişkeninde tutmaktadır. Sayfa oluştuğu anda

2 tipte TableViewCell register edilmektedir. Dışarıdan çağrılabilen reloadTableView metodu ile de TableView'ın reload edilmesi yani tekrar yüklenmesi sağlanmaktadır.



Şekil 59. Flex Pattern örneğindeki SampleTableView kullanılan bir sayfadan TableView sonuç ekran görüntüsü

Şekil 59'daki ekran görüntüsü çıktısını veren TableView örneğinde 2 farklı tipte cell ile ekrana görüntü basılması sağlanmıştır. İstenilen görüntünün oluşturulabilmesi için model kullanılmış ve gerekli yerlere bilgi iletimi bu yolla daha da kolaylaştırılmıştır. Örnekte model 2 farklı tipteki cell'i destekleyecek şekilde oluşturulmuştur.

```
class SampleTableViewModel {
    let twoValueItem: SampleTableViewTwoValueModel?
    let threeValueItem: SampleTableViewThreeValueModel?

    init (twoValueItem: SampleTableViewTwoValueModel) {
        self.twoValueItem = twoValueItem
        self.threeValueItem = nil
    }

    init(threeValueItem: SampleTableViewThreeValueModel) {
        self.twoValueItem = nil
        self.threeValueItem = threeValueItem
    }
}

class SampleTableViewTwoValueModel {
    let valueOne: String
    let valueTwo: Int
    let selectAction: (()->Void)?

    init(valueOne: String, valueTwo: Int, selectAction: (()->Void)? = nil) {
        self.valueOne = valueOne
        self.valueTwo = valueTwo
        self.selectAction = selectAction
    }
}

class SampleTableViewThreeValueModel {
    let valueOne: String
    let valueTwo: Int
    let valueThree: String
    let selectAction: (()->Void)?

    init(valueOne: String, valueTwo: Int, valueThree: String, selectAction: (()->Void)? = nil) {
        self.valueOne = valueOne
        self.valueTwo = valueTwo
        self.valueThree = valueThree
        self.selectAction = selectAction
    }
}
```

Şekil 60. Flex Pattern örneğindeki SampleTableView'de kullanılan modellere ait kod

Temel modelimiz SampleTableViewModel'dir. 2 farklı cell tipine göre içersinde 2 farklı değişken tutmuştur. Bunlardan ilki ekrana 2 tane label ile görüntü sağlayan twoValueItem, ikincisi ekrana 3 tane label ile görüntü sağlayan threeValueItem'dir. Görüleceği üzere selectedAction isimli değişkenlerde blok kod tutulması sağlanmıştır. selectedAction ile TableView'daki o modeli kullanan cell'e tıklandığınca dışarıdan iletilen kodun çalıştırılması sağlanacaktır.

```
extension SampleTableView: UITableViewDelegate, UITableViewDataSource {
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return model.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell
        {
        let item = model[indexPath.row]
        if let value = item.twoValueItem {
            let cell = self.tableView.dequeueReusableCell(withIdentifier:
                "SampleTwoValueTableViewCell", for: indexPath) as! SampleTwoValueTableViewCell
            print("Loading two value cell: \(value.valueOne)")
            cell.labelOne.text = value.valueOne
            cell.labelTwo.text = "\(value.valueTwo)"
            return cell
        } else if let value = item.threeValueItem {
            let cell = self.tableView.dequeueReusableCell(withIdentifier:
                "SampleThreeValueTableViewCell", for: indexPath) as! SampleThreeValueTableViewCell
            print("Loading three value cell: \(value.valueOne)")
            cell.labelOne.text = value.valueOne
            cell.labelTwo.text = "\(value.valueTwo)"
            cell.labelThree.text = value.valueThree
            return cell
        } else {
            return UITableViewCell(frame: .zero)
        }
    }

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let item = model[indexPath.row]
        if item.twoValueItem != nil, let action = item.twoValueItem?.selectAction {
            action()
        } else if item.threeValueItem != nil, let action = item.threeValueItem?.selectAction {
            action()
        }
    }

    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        return 60.0
    }
}
```

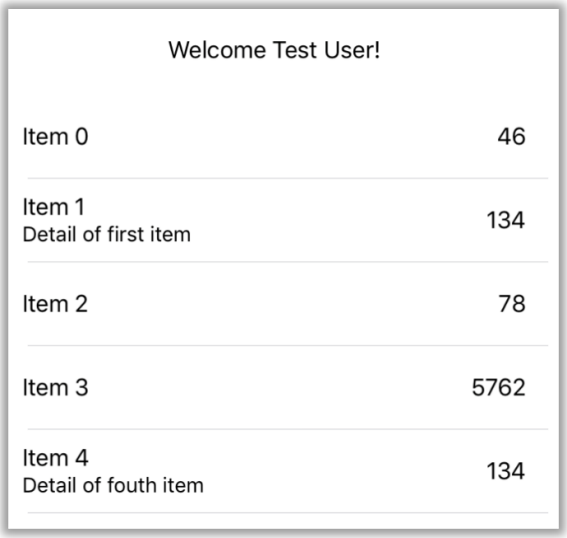
Şekil 61. Flex Pattern örneğindeki SampleTableView sınıfı extension kodu

Şekil 61'daki örnekte Extension ile ayrılan kısımda sadece TableView ile alakalı operasyonel işlemleri barındırılmasını sağlanmıştır. TableView'in cell'i ekrana görüntüleneceği zaman modelde ilk twoValueItem'e bakmakta ve içi dolu olması durumunda cell tipini iki labela sahip tip olarak kabul ederek ekrana bu şekilde basılması sağlanmaktadır. Eğer bu alan boş ise threeValueItem alanına bakılır ve aynı akışla bu sefer 3 label görünümlü cell tipini ile görüntü vermesi sağlanır. didSelectRowAt özelliğine bakılacak olursa burada önceden bahsedilen modeldeki selectAction'ların kullanıldığı görülür, cell'e tıklanması durumunda burada parametrik olarak iletilen kod bloğunun çalışması sağlanacaktır.

View bölümü ile alakalı anlatımlarda şimdiye kadar View, Field ve Part ile alakalı parçalara değinilmiştir. Örneklerle gözlemlenebileceği üzere tümüyle izole bir yapı söz konusu. Talimatı verenin bir metoduna geri dönülmediği ve talimat alanın ne yapacağını ona önceden nasıl yapılacağı bilgisi ile iletilerek tek yönlü bir akış sağlandığı gözlemlenebilir. Bu akış Field'a kadar bu şekildedir ancak Field ile ViewModel arasında bağımlılık vardır. Her Field'ın bir ViewModeli olması beklenir ve Field'ın ViewModel tarafından yönetilmesi gereklidir.

ViewModel bir Field'daki verileri ayarlar, yönetir, bir tuşa basıldığında nelerin yapılacağını ve bunun gibi Field üzerinde yönetim işlerini yöneten kısımdır. Bir diğer görevi de Presenter ve Field arasında köprü görevi görmektir. ViewModel emirleri Presenter'dan alır Field'a onun anlayacağı şekilde iletir. ViewModel emirleri doğrudan doğruya iletmez inisiyatif alarak iletir, kendi sorumluluğunda olan işleri kendisi yapar, Field'ın yapması gerekenleri Field'dan yapmasını ister. Gerekli görmedikçe Presenter'a dönüş yapmaz, gerekli olması durumunda Presenter'daki metotları çağırabilir.

Şekil 62'de MainPage sayfasına ait bir örnek ekran görüntüsü paylaşılmıştır. Ekrandaki yazılar dışarıdan alınan bilgilerle ekrana basılmaktadır. Ekran 1 tane Label ve 1 tane TableView'dan oluşmuştur.



Şekil 62. Flex Pattern örneğindeki MainPageFieldViewController sayfasının ekran görüntüsü

Şekil 63'deki kodda MainPage ekranının Field parçasına ait MainPageFieldViewController kod örneği paylaşılmıştır. Görüleceği üzere Flex örüntüsünün ve tavsiye edilen çalışma şeklinin sayesinde son derece basit ve az koda sahip bir ViewController oluşmuş olduğu gözlemlenebilir. Okunurluğu ve anlaşılırlığı son derece yüksek ve bakımı kolay bir sınıf oluşmuştur. View'da talimat verme gibi işler yapılmamaktadır, hiçbir parçaya neyi nasıl çalıştıracağını söylememektedir. Sadece kendisinin sorumluluğunda olması durumunda gerekli çalışmayı yapmaktadır. Alttaki örnekte ViewController'da çalıştırılması gereken ekranda alert dialog (uyarı iletişim kutusu) çıkarma işinin kodunun showAlert metodunda yazılmış olduğu görülecektir. Burada asla showAlerti çağıran bir kod yoktur, bu metodu çağıran ViewModel'dir. Bu ViewController'da view'ın yüklendiğini bildirmek için viewModel'deki viewDidLoad metodu çağrılmıştır.

```
@IBDesignable class MainPageFieldViewController: BaseViewController {

    @IBOutlet weak var welcomeMessage: UILabel!
    var viewModel: MainPageFieldViewControllerToViewModelProtocol!
    @IBOutlet weak var tableView: SampleTableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.viewModel.viewDidLoad()
    }

    func showAlert(title: String, message: String)
    {
        let alert = UIAlertController(title: title, message: message, preferredStyle:
            UIAlertController.Style.alert)
        alert.addAction(UIAlertAction(title: "OK", style: UIAlertAction.Style.default,
            handler: nil))
        self.present(alert, animated: true, completion: nil)
    }
}
```

Şekil 63. Flex Pattern örneğindeki MainPageFieldViewController sınıfı kodu

ViewModel'de 2 durum vardır. İlk duruma Presenter'dan talimatların gelmesi durumu ki bu MainPageFieldPresenterToViewModelProtocol sayesinde interface'deki ilgili metotların ve değişkenlerin eklenmesi ile sağlanır. İkinci durum ise ViewController'dan (Flex örüntüsünde Field olarak temsil edilir) gelen bilgilerin kullanılması ile alakalı durumdur ki bu da MainPageFieldViewControllerToViewModelProtocol sayesinde aynı şekilde

sağlanır. ViewModel'in kendi başına bağımsız çalışan bir parça olmadığı unutulmamalıdır. Ona ne yapması istendiği ve son durumda nelerin olduğu bilgileri iletir ve ViewModel sadece aldığı bilgilere göre gerektiği şekilde çalışır.

```
class MainPageFieldViewModel: MainPageFieldPresenterToViewModelProtocol {
    var view: MainPageFieldViewController!
    var presenter: MainPageFieldViewModelToPresenterProtocol!

    func setWelcomeMessage(username: String) {
        self.view.welcomeMessage.text = "Welcome \(username)!"
    }

    func setTableViewData(withItems: [SampleTableViewModel])
    {
        self.view.tableView.model = withItems
        self.view.tableView.reloadTableView()
    }

    func itemSelectedMessage(withDetail: String) {
        self.view.showAlert(title: "Item Selected", message: withDetail)
    }
}

extension MainPageFieldViewModel: MainPageFieldViewControllerToViewModelProtocol {
    func viewDidLoad() {
        print("viewDidLoad")
        self.presenter.viewDidLoad()
    }
}
```

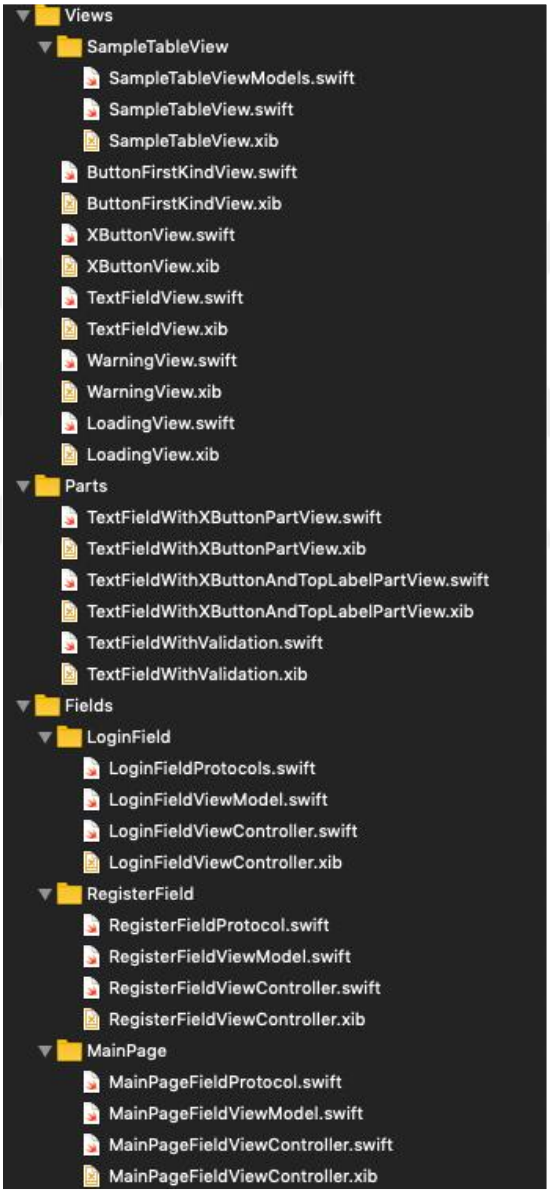
Şekil 64. Flex Pattern örneğindeki MainPageFieldViewModel isimli ViewModel kodu

Şekil 64'deki kod örneğinde ViewController'dan ulaşılan bölüme bakılacak olursa önceki örnekte viewDidLoad çağrımının buraya düştüğünü anlaşılır. Bu örnekte de bu çağrımla birlikte bir log yazılmış ve direkt olarak presenter'a iletildiği görülür.

Presenter'dan ulaşılan bölümde ise setWelcomeMessage metodu ile sayfanın üstündeki hoş geldin mesajında kullanıcının isminin atanması talimatının verildiği, ViewModel'inde bunu kendisi ayarlayarak yaptığı görülür. SetTableViewData ile tableView verileri ayarlanıp reload edilmesi sağlanmıştır. ItemSelectedMessage metodu ile de bir tableView cell seçildiğinde alert popup ile mesaj gösterilmesi istenmiş ve mesajın detayı iletilmiştir.

ViewModel köprü görevi görerek Presenter ve View bölümü arasındaki izolasyonu sağlamıştır. Ayrıca Protocol kullanımı sayesinde ViewModel'de Prestenter'dan izole çalışmaktadır. Dönüş yapacağı Presenter hakkında bilgi bilmesine gerek yoktur, nasıl dönüşler yapacağını Procotocol sayesinde bilmektedir.

View Section ile alakalı projeden örnek bir görünüm Şekil 65'teki ekran görüntüsündeki gibidir.



Şekil 65. Flex örüntüsüne ait bir örnek projede görüntü bölümünün proje dosyaları ekran görüntüsü

Modülü bir sayfa gibi düşünmekte mümkündür. Uygulamanın bir kısmını yani bir parçasını temsil eder. Modül işin yapılmasını talep eden, işin nasıl yapılacağını söyleyen, işin yapılması için gerekli bilgiyi toplayan yerdir. View Section ve Data Layer Section doğrudan Module tarafından yönetilir. Bu bölüm View ve Data Layer ile nasıl iletişim kuracağını bilir dolayısıyla bu bölümler hakkında yeterli bilgiye sahiptir.

Modül bölümü temelde Router, Presenter ve Interactor'dan oluşmaktadır. Shared Presenter Helper ve Shared Interactor destekleyici katmanlardır ve projede olmaları zorunlu değildir. Destekleyici katman modüldeki tekrar eden kodların veya işlerin tekrar tekrar yazılmasını engellemek için kullanılmaktadır. Destekleyici katmanlar paylaşımlıdır dolayısıyla ayrıca bir modül olamazlar veya bir modüle bağlı olamazlar. Bir projenin Presenter'ında olan aynı iş farklı farklı Presenter'larda yazılmışsa bu işi tekrar tekrar bu şekilde yazmak yerine bir tane paylaşımlı destekleyici Presenter yani SharedPresenterHelper oluşturarak aynı yerden tekrar tekrar kullanımını sağlamak mümkün olacaktır. Gene aynı şekilde bir modüldeki Interactor birden fazla kez kullanılması gerekebilir veya Interactor'da sadece belli bazı yerler tekrar tekrar farklı Interactor'da kullanılıyor olabilir. Bu duruma çözüm için Interactor SharedInteractor ile iletişim kurarak tekrarlı kullanımdan kaçınabilir. Interactor belli kısımları kullanılmasına imkân sağlayacak yapıda olabileceğinden gerekmeyen kısımları izole etmek için Protocol kullanılarak izolasyon sağlanabilir. Destekleyici katman kullanımı Presenter ve Interactor anlatımlarında detaylandırılacaktır.

Presenter proje parçasının beyni ve merkezidir. Bir sayfa Presenter aracılığı ile oluşturulur. Bilgilerin çekilmesini, bir işin neyi nasıl yapacağını Presenter yönetir. Gerekli parçalar Presenter'a bağlanarak parçanın yaşam bulması sağlanır. Presenter View bölümü ile iletişim kurmak ve View operasyonlarını yerine getirilmesini sağlamak için ViewModel ile iletişim kurar. Veri çekmek için ise Data Layer'a gidilmesi ve buradan verilerin çekilebilmesi gerekmektedir ancak Presenter doğrudan veriyi çekmez, veri alabilmek için Interactor ile bağlantı kurar. Presenter katmanında data filtreleme, data manipülasyon gibi veriyle alakalı işlemler yapılamaz bunun için Interactor görevlidir. View ile alakalı işlemler asla Presenter'da yazılmaz bunun için ViewModel veya View bölümündeki diğer parçalar sorumludur. Presenter'da amacnın

dışına çıkılamamalı burada business logic işlemleri yer almalı ve modül yönetimi yapılmalıdır.

```
class AuthLoginPresenter {
    var interactor: AuthLoginInteractor!
    var viewModel: LoginFieldPresenterToViewModelProtocol!

    let sharedPresenterHelperScreenOpenStatistic = ScreenOpenStatisticsSharedPresenterHelper()
}
```

Şekil 66. Flex Pattern kullanan bir projede AuthLoginPresenter ismiyle bir login sayfasının Presenter kodu

Şekil 66'daki örnekte görüleceği üzere Presenter'da bir interactor ve bir tane de viewModel tutulmuştur. Burada bir de SharedPresenterHelper kullanılmış olduğu görülmektedir. SharedPresenterHelper'lar ayırt edici şekilde isimlendirilmeli ve değişken olarak tutulmalıdırlar. Koda sonradan bakan başka kişi burada ortak destekleyici Presenter kullandığını kolayca anlayabilmelidir. Anlaşılırlığı ve ayırt ediciliği arttırmak için değişkenin isimlendirmesinde başına sharedPresenterHelper yazılmıştır.

```
extension AuthLoginPresenter: LoginFieldViewModelToPresenterProtocol {
    func viewDidLoad()
    {

            self.sharedPresenterHelperScreenOpenStatistic
            .sendScreenOpenStatistic(screenName: "Login")

        self.viewModel.setupVM(username: nil, password: nil, loginTapped:
            {username,password in
            print("Login with username: \(username), password: \(password)")
            self.viewModel.showLoading()
            self.interactor.login(username: username, password: password, completion: {
                (success, userModel) in
                self.viewModel.hideLoading()
                if(success && userModel != nil) {
                    self.viewModel.showAlertAndDoJob(title: "Success", message: "Login
                        Sucessful!\nWelcome \(String(describing:
                        userModel!.fullName!))!\nYour last login date is:
                        \(String(describing: userModel!.lastLoginDate!))", okTapAction: {
                        if(userModel != nil) {
                            self.viewModel.pushTo(viewController:
                                MainPageRouter.createMainPageHome(withUserModel:
                                userModel!))
                        }
                    })
                } else {
                    self.viewModel.showAlertAndDoJob(title: "Failed", message: "Login
                        Failed!", okTapAction: nil)
                }
            })
        }, registerTapped: {
            print("Registering")
            self.viewModel.pushTo(viewController: AuthRouter.createRegisterPage())
        })
    }
}
```

Şekil 67. Flex Pattern örneğindeki AuthLoginPresenter sınıfı extension kodu

Şekil 67'deki kod örneğinde ViewModel'den dönüşünde çalıştırılan kısım paylaşılmıştır. Sayfa viewDidLoad olduğunda ViewModel'ın username, password, login tuşuna basılınca yapılacak aksiyon ve register tuşuna basılınca yapılacak aksiyon atanmıştır. Login tuşuna basıldığında Interactor'a gidilip bilgi çekilmiş be Presenter sonuç geldiğinde ne yapılacağını ViewModel'e aktarmıştır. Register tuşuna basıldığında ise doğrudan Register sayfasına gidilmesi sağlanmıştır. Bütün aksiyonlar bir kod bloğu olarak ViewModel ve Interactor'lara iletilmiştir Presenter'da tekrar başka bir metodu çağrılmasına gerek kalmamıştır. ViewModel'in ve Interactor'ın bu sayede izole olabilmesi sağlanmıştır.

SharedPresenterHelperScreenOpenStatistic sayesinde SharedPresenterHelper kullanılması sağlanmış ve aynı işin farklı Presenterlarda tekrar tekrar yazılması önlenmiştir. Yukarıdaki kod örneğinde açılan ekranın bilgisini istatistik olarak gönderen bir özellik kullanılmıştır. Bu özellik bir de Interactor'a ihtiyaç duyar çünkü bilgiyi servise iletecektir. Tüm bu tekrarlar zaman kaybına sebebiyet verebilecek iken tek yerden kullanılması helper sayesinde büyük zaman kazandırılmıştır.

```
class ScreenOpenStatisticsSharedPresenterHelper {
    let interactor = ScreenOpenStatisticsSharedInteractor()

    func sendScreenOpenStatistic(screenName: String) {
        self.interactor.screenOpened(screenName: screenName) { (success) in
            print("Screen Open: \(screenName) Send Statistic Success is: \(success)")
        }
    }
}
```

Şekil 68. Flex Pattern örneğindeki ScreenOpenStatisticsSharedPresenterHelper sınıfı kodu

Ekran açma istatistiği göndermeye yönelik örnek Şekil 68'de paylaşılmıştır. Örnekte Interactor olduğu görülmektedir. Normalde Interactor olma zorunluluğu yoktur lakin yapılacak işte Interactor kullanılması gereken fonksiyon olması sebebiyle doğrudan bir Interactor bağlanmıştır. Bu örnekteki iş dışardan aldığı screenName parametresindeki ekran ismini istatistik tutulması için servise gönderilmesidir.

Interactor katmanı Data Layer ile iletişim kurup bilgiyi Presenter'a istediği şekilde dönen bir katmandır. Presenter'ın filtreleme, data manipülasyon, belli verileri tutma gibi işleri de bu katmanda yapılır.

Şekil 69'deki Interactor örneği yukarda paylaşılan Login sayfasının Presenter'daki Interactor bağlantısını kapsar. Bu örnekte SharedInteractor kullanılmamıştır. Basit haliyle Interactor kullanım örneği paylaşılmıştır.

```
class AuthLoginInteractor {
    func login(username: String, password: String, completion: @escaping
        (Bool,UserModel?) -> Void) {
        RemoteDataManager().login(parameters: LoginRequestModel(username: username,
            password: password)) { (responseModel) in
            if(responseModel.success == true) {
                let usermodel = UserModel(fullName: responseModel.fullname, token:
                    responseModel.token, lastLoginDate: responseModel.sessionDate)
                completion(true, usermodel)
            } else {
                completion(false, nil)
            }
        }
    }
}
```

Şekil 69. Flex Pattern örneğindeki AuthLoginInteractor sınıfı kodu

Login request (istek) atılması ile alakalı bir örnek Şekil 69'da paylaşılmıştır. Username ve password parametreleri iletilmektedir. Completion parametresi ile data layerdan elde edilen sonucun callback metod kullanılarak Presenter'a dönmesi sağlanmıştır. Interactor gelen datayı okumuştur, başarılı olup olmadığı durumunu kontrol etmiş ve Presenter'ın istediği ve anlayacağı tip modelde geri dönüş sağlamıştır.

Login örneğindeki Presenter'da SharedPresenterHelper kullanılmıştı. Bu ortak presenter ise SharedInteractor kullanmıştı. Lakin bazı durumlarda SharedPresenterHelper kullanılmak istenmeyebilir ama sadece SharedInteractor kullanılması gerekebilir. Bu durumda yapı buna da imkân vermektedir. Örnek için ikinci defa olarak oluşturulan AuthLoginSecondPresenter isimli farklı bir login safası parçası bahsedilen şekilde sadece SharedInteractor kullanmıştır.

```
extension AuthLoginPresenter: LoginFieldViewModelToPresenterProtocol {
    func viewDidLoad()
    {
        self.sharedPresenterHelperScreenOpenStatistic.sendScreenOpenStatistic(screenName: "Login")
```

Şekil 70. Flex Pattern örneğindeki AuthLoginPresenter extension sınıfı kodu

Şekil 70'deki kod örneği önceki Presenter anlatımında paylaşılan örnektir. Burada SharedPresenterHelper kod örneğindeki gibi kullanılmıştır.

```
extension AuthLoginSecondPresenter: LoginFieldViewModelToPresenterProtocol {
    func viewDidLoad()
    {
        self.interactor.screenOpened(screenName: "Login") { (success) in
            print("Screen Login Second Open Statistic Send Success is: \(success)")
        }
```

Şekil 71. Flex Pattern örneğindeki AuthLoginSecondPresenter extension sınıfı

Şekil 71'deki kod örneğinde görüleceği üzere bu örnekte modülün mevcut Interactor'una giderek işin yapılması sağlanmıştır.

```
class AuthLoginSecondInteractor {
    func login(username: String, password: String, completion: @escaping (Bool,UserModel?) ->
        Void) {
        RemoteDataManager().loginSecondForOtherUsers(parameters: LoginRequestModel(username:
            username, password: password)) { (responseModel) in
            if(responseModel.success == true) {
                let usermodel = UserModel(fullName: "\(responseModel.name!)
                    \(responseModel.surname!)", token: responseModel.token, lastLoginDate: nil)
                completion(true, usermodel)
            } else {
                completion(false, nil)
            }
        }
    }
}

extension AuthLoginSecondInteractor: ScreenOpenStatisticsSharedInteractorProtocol {
    func screenOpened(screenName: String, completion: @escaping (Bool) -> Void) {
        ScreenOpenStatisticsSharedInteractor.shared.screenOpened(screenName: screenName,
            completion: completion)
    }
}
```

Şekil 72. Flex Pattern örneğindeki AuthLoginSecondInteractor sınıfı

Şekil 72'deki AuthLoginSecondInteractor sınıfı kod örneğinde görüleceği üzere burada ScreenOpenStatisticSharedInteractor kullanılmıştır. Bu ortak yapı kullanılırken Interactor sınıfında SharedInteractor'un protocol'ü kullanılması göz ardı edilmemelidir. Bu Protocol'un kullanılması sayesinde aynı özelliği ortak Interactor'den kullanacak diğer Interactor'lerin çağrımının aynı olabilmesinin sağlanması hedeflenmiştir.

```
protocol ScreenOpenStatisticsSharedInteractorProtocol {
    func screenOpened(screenName: String, completion: @escaping (Bool) -> Void)
}

class ScreenOpenStatisticsSharedInteractor: ScreenOpenStatisticsSharedInteractorProtocol {
    static let shared = ScreenOpenStatisticsSharedInteractor()
    func screenOpened(screenName: String, completion: @escaping (Bool) -> Void) {
        RemoteDataManager().sendScreenOpenStatistic(parameters:
            ScreenOpenRequestModel(screenName: screenName, openDate: Date())) {
            (responseModel) in
            completion(responseModel.success)
        }
    }
}
```

Şekil 73. ScreenOpenStatisticsSharedInteractor sınıf ve protocol kodu

Pek çok Interactor'da tekrar tekrar yazılması gerekecek yapı Şekil 73'teki örnek sayesinde tek yerden tekrarlanmaksızın çalışabilmesi sağlanmıştır. Protocol sayesinde bu yapıyı kullanacak Interactor'lerin çağrımının her yerde aynı olması sağlanmıştır.

Sadece SharedInteractor kullanan, SharedPresenterHelper kullanmayan yerlerde neden Presenter'larında doğrudan SharedInteractor kullanılmadığı sorusu sorgulanabilir. Bunun sebebi gereksiz yere muhtemelen birden fazla Interactor ile Presenter'ı karmaşıklaştırmamak ve bu işi Interactor'de tutarak hangi kullanımların olduğunu daha rahat anlayabilmek için Interactor'dan kullanım söz konusudur.

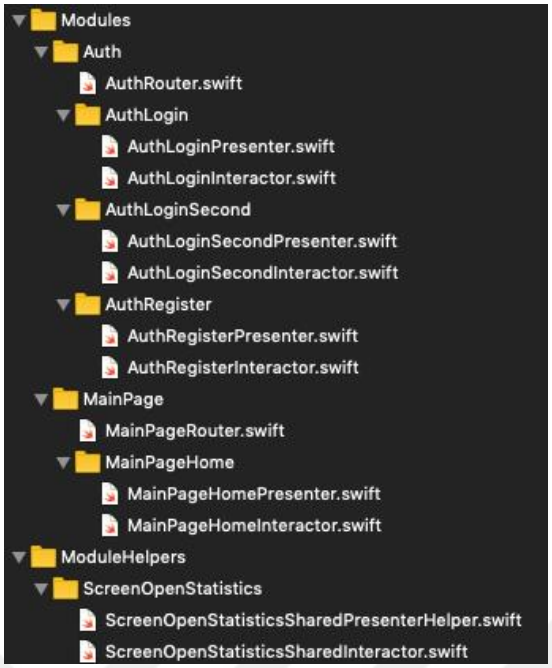
Router modülü oluşturan parçadır. Modülde hangi Presenter'ın kullanılacağını ve kullanılan Presenter'ın hangi View-Model ve Interactor ile iletişim kuracak bunu belirler. Router oluşturucu olarak çalışır, gerekli nesnelerin oluşturulmasını ve birbirlerine bağlanması ile görevlendirilmiştir.

Modülleri kendi içlerinde gruplandırmak mümkündür. Birden fazla modül aslında bir işin parçası olabilir. Örnek verilecek olursa; bir sayfadaki login ve register işleri aslında authentication'a ait bir iştir. Dolayısıyla login ve register modülleri Authentication grubu altında yerleştirilebilir. Bu durumda klasör hiyerarşisi şu şekilde gözükecektir Authentication/Login/ ve Authentication/Register/. Flex Pattern ile bu tip gruplandırılan işlemler için tek bir Router ile yönetilmesi ve Router sınıfının grup

klasöründe barındırılması önerilir. Örnekteki Router konumu ve isimlendirmesi ise şu şekildedir: Authentication/AuthenticationRouter.

```
class AuthRouter {

    class func createLoginPage() -> UIViewController {
        let presenter: AuthLoginPresenter = AuthLoginPresenter()
        let viewModel = LoginFieldViewModel()
        viewModel.view = LoginFieldViewController(nibName: "LoginFieldViewController",
            bundle: nil)
        viewModel.view.viewModel = viewModel
        viewModel.presenter = presenter
        presenter.viewModel = viewModel
        presenter.interactor = AuthLoginInteractor()

        return viewModel.view
    }

    class func createLoginSecondPage() -> UIViewController {
        let presenter: AuthLoginSecondPresenter = AuthLoginSecondPresenter()
        let viewModel = LoginFieldViewModel()
        viewModel.view = LoginFieldViewController(nibName: "LoginFieldViewController",
            bundle: nil)
        viewModel.view.viewModel = viewModel
        viewModel.presenter = presenter
        presenter.viewModel = viewModel
        presenter.interactor = AuthLoginSecondInteractor()

        return viewModel.view
    }

    class func createRegisterPage() -> UIViewController {
        let presenter: AuthRegisterPresenter = AuthRegisterPresenter()
        let viewModel = RegisterFieldViewModel()
        viewModel.view = RegisterFieldViewController(nibName: "RegisterFieldViewController",
            bundle: nil)
        viewModel.view.viewModel = viewModel
        viewModel.presenter = presenter
        presenter.viewModel = viewModel
        presenter.interactor = AuthRegisterInteractor()

        return viewModel.view
    }
}
```

Şekil 74. Flex Pattern örneğindeki AuthRouter kodu



Şekil 75. Flex Pattern örneğinde modül ve modül destekleyicilerin bulunduğu proje yapısının ekran görüntüsü

Data Layer servis ile veya yerel veritabanı ile iletişim kurup bilgi alan ve/veya bilgi gönderen katmandır. Bu katman tümüyle izole bir katmandır. Bu katmanda alınan veri veya gönderilecek veri değiştirilemez. Ne gönderilmesi gerekiyorsa veya gelen veri neyse doğrudan iletilmesi sağlanır. Business logic işleri bu katmanda yazılamaz.

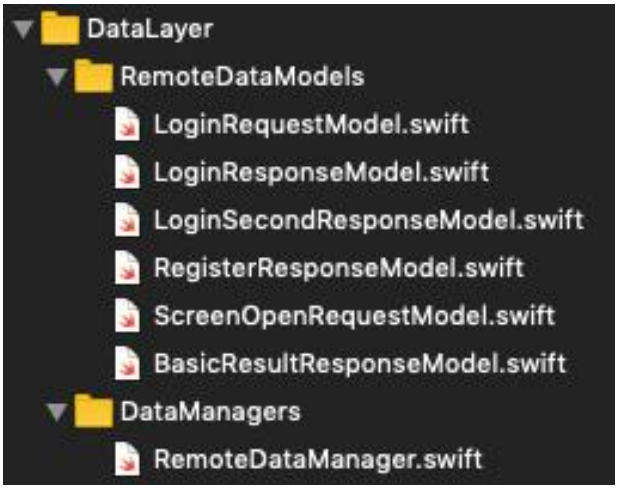
Bu katmanın modelleri tümüyle bu katmana özeldir ve Data Model olarak temsil edilirler. Değişkenleri immutable yani sabit tipte değişken olmaları beklenir. Sadece ilk yaratıldıkları zaman oluşturulurlar. Lakin mobilde gelen veriyi serialize ve deserialize edebilmek için yaygınca kullanılan bazı kütüphanelerin sadece mutable değişkenler ile çalışabilmesinden dolayı bu gibi durumlarda mutable tipte kullanılması mümkündür. Burada temel kural data modellerin içerisindeki bilgilerin sonradan değiştirilmemesine yöneliktir.

Data Layer katmanında 2 grup parçalar vardır biri Data Manger ki bu kısım bilgiyi almayı, bilgi göndermeyi ve ulaşılacak kaynak ile iletişimi yöneten kısımdır. Hangi bilginin gönderileceğini ve alınacağını söyler. Nasıl bilgi gitmesini ve hangi bilgilerin gösterileceğini de 2. kısım olarak Data Model sağlar.

```
class RemoteDataManager {
    func login(parameters: LoginRequestModel, completion: @escaping (LoginResponseModel) ->
        Void) {
        MockRemoteData().login(parameters: parameters, completion: completion)
    }

    func loginSecondForOtherUsers(parameters: LoginRequestModel, completion: @escaping
        (LoginSecondResponseModel) -> Void) {
        MockRemoteData().loginSecondForDifferentUsers(parameters: parameters, completion:
            completion)
    }

    func register(parameters: LoginRequestModel, completion: @escaping
        (RegisterResponseModel) -> Void) {
        MockRemoteData().register(parameters: parameters, completion: completion)
    }

    func sendScreenOpenStatistic(parameters: ScreenOpenRequestModel, completion: @escaping
        (BasicResultResponseModel) -> Void) {
        MockRemoteData().sendScreenOpenStatistic(parameters: parameters, completion:
            completion)
    }
}
```

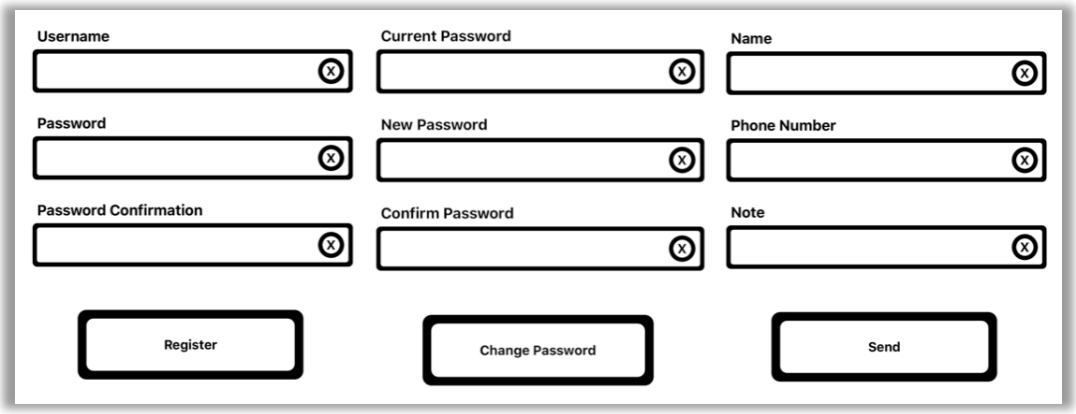
Şekil 76. Flex Pattern örneğindeki Data Layer katmanındaki Manager sınıfı kod örneği

Data Layer Section ile alakalı projeden örnek bir görünüm Şekil 77'teki ekran görüntüsündeki gibidir.



Şekil 77. Flex Pattern örneğindeki Data Layer bölümünün proje yapısının ekran görüntüsü

Bölümün devamında görünümü aynı olan ancak amacı farklı olan 3 ekranı Flex Pattern ve diğer örüntüler ile yapıldığı zaman ortaya çıkan farkı gösteren bir örnek yapılacaktır. Örneğe göre 3 labeldan oluşan ve 1 tuşa sahip aynı ekran üye kayıt, şifre değiştirme ve rezervasyon işi için kullanılacaktır.



Şekil 78. Aynı gönüme sahip üye kayıt, şifre değiş ve rezervasyon ekranları

Şekil 78'de birbirinin aynı ancak farklı yazılara ve aksiyonlara sahip üye kayıt, şifre değiştirme ve rezervasyon ekranlarına ait ekran görüntüleri paylaşılmıştır. Tablo 5'te Flex Pattern ve diğer örüntülerin 3 sayfalı bu örnek projede sahip oldukları sınıflara dair bilgi paylaşılmıştır. Tablo 5'te Flex örüntüsünün diğer örüntülere kıyasla bir ekran için daha az sayıda View ile çalışabildiği görülmüştür. Diğer örüntülerde iş ya da modül kadar View oluşturulurken Flex'te tek bir View kullanmak yeterli olmuş ve View, View-Model ile yönetilmiştir.

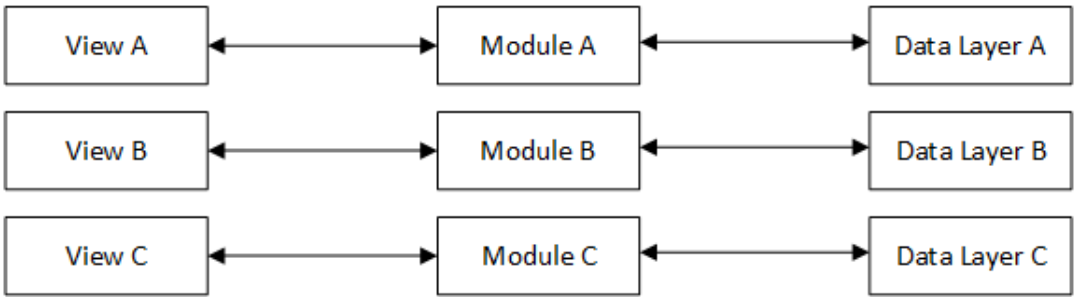
Tablo 5. Flex Pattern ve diğer örüntülerin örnek için sahip olduğu sınıflar

	View	ViewModel	Presenter	Interactor
MVC	RegisterVC PasswordVC ReservationVC			
MVP	RegisterVC PasswordVC ReservationVC		RegisterPresenter PasswordPresenter ReservationPresenter	
MVVM	RegisterVC PasswordVC ReservationVC	RegisterVM PasswordVM ReservationVM		
MVP-VM	RegisterVC PasswordVC ReservationVC	RegisterVM PasswordVM ReservationVM	RegisterPresenter PasswordPresenter ReservationPresenter	

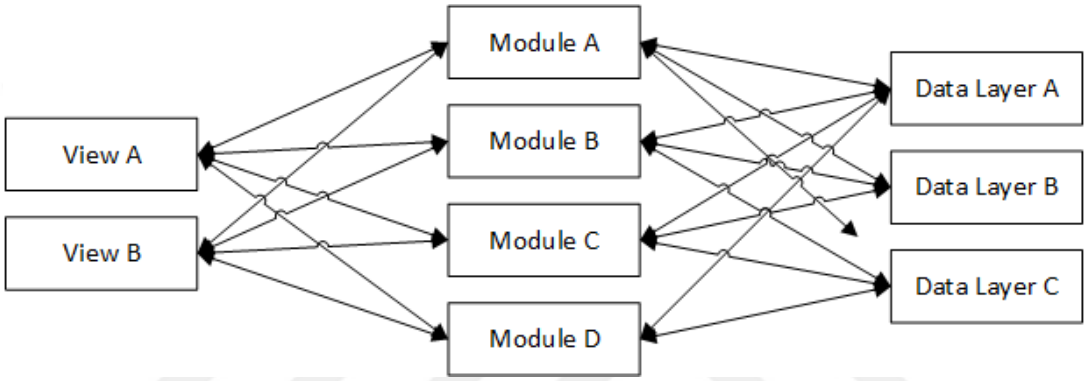
VIPER	RegisterVC PasswordVC ReservationVC		RegisterPresenter PasswordPresenter ReservationPresenter	RegisterInteractor PasswordInteractor ReservationInteractor
Flex	FormFieldVC	FormVM	RegisterPresenter PasswordPresenter ReservationPresenter	RegisterInteractor PasswordInteractor ReservationInteractor

Flex örüntüsü bölümleri arasında ayrı ayrı N kadar tekrarı önler. Bölümler projede farklı bölümlerle rahatça kullanılabildiğinden diğer örüntülerdeki gibi tekrar tekrar aynı şeylerin yazılma mecburiyeti ortadan kalkar. Örneğin bir bankanın bireysel müşterileri için yazılmış kayıt olma işlemi yapan veri katmanı bölümü farklı görüntü ve modül kullanan bireysel müşteriler için de rahatlıkla kullanılabilir. Rezervasyon işlemi için geliştirilmiş bir rezervasyon ekranı istenirse şifre değiştirme işlemi içinde hızlıca bir değişiklik ile kullanılabilir. İstenirse aynı şekilde kayıt olma için kullanılmış bir modül bölümü de belli başlı değişiklikler yapılarak farklı görüntü veya veri katmanı işleri ile birlikte kullanılabilirler. Şekil 78 ve Tablo 5'teki örneği baz alacak olursak bu örüntü view yani görüntü katmanında yazılmış kodların 3 defa tekrarlanmasını engellemiştir. Birbirinin aynı olsa da ekrana basılması için 10 satır kod yazılması gerekiyorsa klasik yöntemlerde bu kodlar her sınıfta tekrar tekrar yazıldığından toplamda 30 satır kod yazılmaktaydı. Ancak Flex Pattern sayesinde tekrarlama önlenerek orijinal kod satır sayısı ile yani 10 satır kod ile yazılabilmesi sağlanmış olmuştur. Tablo 5'teki örnek görüntü bölümünde N kadar kat katkı sağlanabileceğini gösterir.

Eğer Tablo 5 örneğine; görüntüsü aynı olmasına karşın RegisterCommercial isimli modül eklenseydi diğer örüntülerdeki gibi tekrar bir görüntü eklemeye veya eğer gerekli değilse tekrar farklı bir veri iletişimi için sınıf eklemeye gerek olmayacaktır. Dolayısıyla modül sayısı artsa bile diğer bölümlerin sayısı artmaz. Bu da N kadar kat katkı sağlayacaktır. Aynı durum bahsedilmiş olduğu gibi veri katmanı bölümü içinde geçerlidir. N kadar katkı ilişkisi Şekil 79 ve Şekil 80 diğer örüntüler ve Flex örüntüsü arasında kıyaslama sağlamaktadır.



Şekil 79. Flex harici diğer örüntülerdeki bölüm ilişkisi



Şekil 80. Flex Pattern'de bölüm ilişkisi

Şekil 79 diğer örüntüleri temsil etmektedir. Diğer örüntüler bir sayfayı tek bir modül olarak baz alırken kendi parçaları ile bağ kurarlar. Dolayısıyla modül kadar parçalara sahiptirler. Ancak Flex örüntüsünde her bölüm bağımsızdır. Şekil 80'deki gibi her bölüm kendi içinde bağımsız N kadar sayıda parçadan oluşabilirler.

5. SONUÇ

Kod kalitesini arttırma ve tasarım örüntüleri olarak iki bölüme ayrılmış olan tezde kaliteyi arttırmak için mevcutta olan çalışmalar ve tasarım örüntüleri incelenmiş, bazılarının üzerinde çalışılarak dezavantajları giderilmesi ve ek faydalar sağlayabilmek için çalışmalar paylaşılmış, kıyaslamalar yapılmış, farklı yöntemler paylaşılmış ve yeni örüntüler keşfedilmiştir.

Tez çalışması neticesinde elde edilen sonuçlara dayanarak kod kalitesini düşüren ve maliyeti arttıran faktörler şunlardır;

- Projenin koşullarına göre yanlış tasarım örüntüsü seçilmesi.
- Tasarım örüntülerinin yanlış kullanılması.
- Sınıflardaki kod miktarının aşırıya kaçması sonucu kodun anlaşılmasının ve değişiklik yapılabilmesinin zorlaşması.
- Değişime ayak uyduramayacak şekilde geliştirme yapılması.
- Kod bölümlerinde yeterince izolasyon yapılmaması neticesinde, bu bölümlere dışarıdan erişildiğinde kodda hata riskinin artması ve değişiklik yapılabilirliğin düşmesi.
- Kod tekrarları. Özellikle önemli kodların kopyalanarak başka yerlerde tekrarlı kullanımı neticesinde değişiklik yapmak gerektiğinde ortaya çıkan problemler.
- Tasarımsal işlemlerin çok fazla koddan yönetilmesi.
- Proje mimarisine, platform standartlarına ve OOP standartlarına uygun olmayan geliştirme yapılması.
- Servis ve mobil iletişiminde karmaşık data modellerin kullanılması.
- Servis ile iletişimde veri gönderirken veya alırken, veriyi kullanabilmek için çok karmaşık ve zor iş kodu yazmak zorunda olunması.
- Servis tarafında yapılan geliştirmenin mobile uygun olarak yapılmaması ve mobil ile servis geliştiren kaynakların iletişimsizliği.

Tez çalışması neticesinde elde edilen sonuçlara dayanarak kod kalitesini arttıran ve maliyeti düşüren faktörler şunlardır;

- Projenin koşullarına uygun olan bir tasarım örüntünün seçilmesi
- Tasarım örüntülerinin eksik veya sorunlu kısımlarını gidermek için gerekenlerin uygulanması.
- Sınıftaki kod miktarının mümkün olduğunca azaltılması ve rahat anlaşılır şekilde kod yazılması.

- Projedeki parçaların esnek şekilde tasarlanması. Oluşturulan bir parçanın başka bir yerde veya başka bir parçayla birlikte doğrudan kullanılabilmesi.
- Kod veya projedeki tasarımsal parçaların esneklik kazanabilmesi için parçalara bölünmesi.
- Projedeki parçalarda izolasyonun sağlanması. Böylelikle dışarıdan erişimle kod veya parça bütünlüğüne zarar verilmesinin engellenmesi.
- Kodların tekrarının engellenmesi. Tekrar edilmesi muhtemel kodların tek merkezden yönetilecek şekilde yazılması.
- Kod kalitesini arttırmak için yazılımı geliştirirken faydalı prensiplerden, tasarım örüntülerinden ve çalışmalardan faydalanılması.
- Proje mimarisine, platform standartlarına ve OOP standartlarına uygun geliştirme yapılması.
- Tasarımsal işlerin ağırlıklı olarak anlaşılır bir arayüz üzerinden yapılması.
- Kodu destekleyen yardımcı nesnelerin (örnek localization strings, enum ve constant nesneler) topluca rahat görülebilmesi ve nerelerde kullanıldığı rahat anlaşılabilmesi.
- Servis ile mobil arasında daha basit ve anlaşılır data modellerin kullanılması. Bu iletişimde gereksiz bilgilerin mümkün olduğunca olmaması.
- Servise veri göndermek üzereyken veya servisten bilgiyi alınca verilerin kullanılmasında iş kodu yazımının daha basit ve daha kolay anlaşılır olması.
- Servislerin mobile uygun şekilde tasarlanması ve servisi geliştiren kaynaklar ile mobil geliştiren kaynakların etkin bir şekilde iletişim halinde olmaları.

Tasarım örüntüsü tercih ederken projenin koşullarının iyi tespit edilmesi gerektiği sonucuna varılmıştır. Örneğin eğer projeyi bitirmek için yeterli zaman yoksa veya sık değişikliğe uğrayacak bir proje ise VIPER gibi karmaşık örüntülerden kaçınılması gerekir. Tasarım örüntüsü tercihi sadece boyutsal ve karmaşıklık olarak değil, projede gelecek planları, yapılabilecek değişimler, projenin özellikleri de göz önünde bulundurularak yapılmalıdır. Önemli olan bir diğer hususta katmanlardaki kod miktarının durumudur, örüntü tercihi yapılırken bu da göz önünde bulundurulmalıdır. Örneğin hızlı ve basit bir projede MVC kullanımı her ne kadar uygun olsa da eğer projede iş katmanı kodları karmaşıksa ve çok satır kod yazmayı gerektiriyorsa bu

durumda MVVM veya MVP tercih edilmesi daha doğru olacaktır. Özetle projenin durumu ve ihtiyaçları örüntü tercihini doğrudan etkilemeyecektir.

Tasarım desenlerinin bazı dezavantajlarını ortadan kaldırmak çeşitli örüntülerin kullanımı ile bu dezavantajları giderebileceği veya ekstra fayda sağlayabileceği sonucuna varılmıştır. Bu kapsamda mevcutta olan örüntüler tavsiye edilmiş ve tez çalışması ile birlikte Utils Based Pattern, Protected Template Pattern ve Imitate Pattern isimli 3 adet yeni örüntü keşfedilmiştir. MVC, MVP ve MVVM örüntülerdeki sayfa geçişlerini daha rahat yönetmek ve bilgi gönderebilmek için Coordinator Pattern kullanılarak daha güzel bir yapı kurgulanabileceği görülmüştür. Sıklıkla tekrar eden kodları tek yerde toplayıp kod tekrarını engellemek, birden fazla yerde yapılan bir işi tek yerden yönetip kullanmak için tez çalışması ile keşfedilen Utils Based Pattern kullanılabileceği görülmüştür. Tasarım desenleri her ne kadar büyük avantajlar sağlasa da bazı durumlarda yanlış kullanım sebebiyle işin doğru çalışabilmesine engel olabilecek yazılımcı bazlı bazı hatalar da yapılabilmektedir. Template Pattern kullanan bir yazılımcının üst sınıftaki metodu çağırmamasından kaynaklı yaşanan problemi çözmek için Protected Template Pattern keşfedilmiş ve paylaşılmıştır. Bir modeli Prototype Pattern gibi bütünüyle kopyalamak yerine, sadece belirli özellikleriyle istenilen şekilde bir taklidini oluşturmak için tez çalışması ile keşfedilmiş Imitate Pattern kullanılabileceği tespit edilmiştir. Prototype, Singleton ve Template örüntülerindeki bazı problemleri ortadan kaldıran yöntemlerden bahsedilmiştir.

Tez ile birlikte bir yazılım projesini daha da esnek hale getirebilen Flex Pattern ismiyle yeni bir mimari tasarım örüntüsü keşfedilmiştir. Örüntü ile bütün View parçalarının farklı yerlerde esnek şekilde tekrarlı kullanılabilmesi sağlanmıştır. Bu örüntüde temel hedef projedeki herhangi bir işin tekrar yapılmamasıdır. Uygulamadaki bir sayfanın, herhangi bir görüntünün, bir parça kodun, bir bütün sayfa kodun ve bunun gibi tekrar kullanılabilecek her türlü şeyin tek seferde geliştirilip kullanılması sağlanmıştır. Katmanlar arasında izole bir yapı kurularak, bir katmanın başka bir katmanı bozma ihtimali ortadan kaldırılmıştır. Katmanların başka katmanları bağımlılık şartı olmadan özgürce kullanabilmeleri sağlanmıştır.

Flex Pattern diğer mimari tasarım örüntülerinden daha farklı bir örüntüdür. Diğer mimari örüntüleri tümüyle izole olunması şartı koşmaz, bir görüntü veya sayfa için tüm parçalarla birlikte çalışırlar, parçaların tekrarlı kullanımları için herhangi bir kuralları yoktur. Oysa ki Flex Pattern, parçaların birbirlerinden bağımsız olması için izole olmaları şartını koşar, her parça birbirine bağımlı olmadan ayrı olarak çalışır, parçaların tekrar kullanılabilmesi örüntünün asıl amacıdır dolayısıyla tekrar kullanılabilirlik örüntünün temel kuralıdır. Bahsedilen temel kurallar ve yapısı dolayısıyla Flex Pattern esneklik, değişime açık olma ve tekrar kullanılabilirlik konusunda diğer mimari örüntülerden daha iyidir.

Flex Pattern büyük projeler için çok ideal bir örüntü olduğu ve değişikliğe açık olması sebebiyle diğer örüntülerden daha büyük avantajlara sahip olduğu sonucuna varılmıştır. Bu örüntüde her şey parçalara ayrılır ve tekrar kullanılabilir. Tekrar kod yazmaya veya tasarımını düzenlemeye gerek kalmaksızın parçalar farklı parçalarla birlikte kullanılabilir. İş katmanını destekleyen ve tekrarı önleyen kısımları mevcuttur. Görevler bellidir ve daha fazla ayrıştırılmıştır dolayısıyla kod yoğunluğu ve karmaşası bu örüntüde diğer örüntülere kıyas daha fazla önlenir. Flex örüntüsünde parçalar tek yerden yönetildiğinden, tekrar kullanımının çok kolay ve çoğunlukla ek geliştirmeye gerek duymamasından dolayı büyük projelerin bakım maliyeti ve değişiklik yapma maliyeti bu örüntüde daha az olduğu sonucuna varılmıştır.

Mobil platformlarda da yazılım kalitesinin arttırılması için SOLID, YAGNI, KISS, DRY, SOC gibi prensiplerin kullanılmasının çok önemli bir gereklilik olduğu sonucuna varılmıştır. Tekrarlı kullanımların olduğu sınıflarda, kod miktarını azaltmak ve basitleştirmek için Fluent Interface kullanımının büyük bir avantaj sağladığı görülmüştür. Test Driven Development sayesinde ileride yaşanabilecek sorunların çıkma ihtimalleri düşürülmekte ve uzun vadede maliyetin çok ciddi oranda düştüğü sonucuna varılmıştır.

Bir projeye sonradan dahil olan yazılımcıların projeyi daha iyi anlamaları veya zaman geçtikten sonra tekrar dahil olunan projeleri daha rahat anlama ve sorunların daha rahat tespiti ve çözümü için arayüz kullanımının çok önemli olduğu sonucuna varılmıştır. Arayüz kullanımını daha da etkinleştirmek, görüntü tekrarlarını engellemek ve iç içe kullanım imkanlarını sağlayabilmek için “3.2.3 Etkin Arayüz

Oluşturucu Araçlar Kullanımı" bölümünde önerilen yöntemler detaylandırılmıştır. Bu yöntemler sayesinde arayüz ile oluşturulan görüntüler parçalanarak tekrarlı kullanım imkanına sağlanmıştır.

Mobil yazılımın kalitesine ve maliyetine doğrudan etki eden bir diğer faktöründe web servis olduğu tespit edilmiştir. Servislerin mobile uygun hatta mümkünse mobile özel yazılması maliyete ve işin kalitesine önemli katkı sağlayacağı sonucuna varılmıştır. Servis tarafında karmaşık ve zor anlaşılan data modellerin geliştirilmiş olması projeyi negatif yönde etkileyecektir. Çünkü karmaşık bir yapının mobil tarafta geliştirmesi de bir o kadar zorlu ve zaman alıcı olacaktır. Bu maliyeti arttıracağı gibi bir işin hatalı çalışması riskini de aynı oranda arttıracaktır. Servis tarafında güzel yapı oluşturulması her ne kadar servis geliştirme eforunu artmasına sebep olabilse de bunun yerine mobil taraftaki eforun artmasından daha az maliyete sebep olacaktır. Günümüzde IOS ve Android gibi en az iki mobil platforma çalışıldığını dikkate alırsak mobil eforun artması demek bir işin iki kat daha fazla efora sebep olacaktır. Bu problemin çözümü için servis ile mobilin iletişiminin daha basit olması gerektiği sonucuna varılmış ve "3.2.4 Etkin Servis ve Mobil İletişimi" bölümünde iletişimi basitleştiren çözüm önerisi örneklerle detaylandırılmıştır.

Tez çalışması ile pek çok açıdan maliyeti etkileyen faktörler araştırılmış ve çözümler için çalışılmıştır. Kod kalitesini arttırarak maliyetinde düşürülebileceği tespit edilmiştir. Kaliteli kodun bakımı ve değişikliğe uygun olması sebebiyle maliyet avantajı büyük projelerde daha fazla olacağı sonucuna varılmıştır.

KAYNAKÇA

[1] [Çevrimiçi]. Mevcut: https://developer.apple.com/documentation/. [Erişildi: Mayıs 2019].

[2] «medium.com,» [Çevrimiçi]. Mevcut: https://medium.com/.

[3] «github.com,» [Çevrimiçi]. Mevcut: https://github.com/.

[4] «msdn.microsoft.com,» [Çevrimiçi]. Mevcut: https://msdn.microsoft.com/.

[5] G. E. Krasner ve S. T. Pope, *A Description of the Model-View-Controller UserInterface Paradigm in the Smalltalk-80 System*, 1998.

[6] Y. Zhang ve Y. Luo , «An architecture and implement model for Model-View-Presenter pattern,» %1 içinde *2010 3rd International Conference on Computer Science and Information Technology*, 2010.

[7] E. SØRENSEN ve M. I. MIHAILESCU, «Model-View-ViewModel (MVVM) Design Pattern using Windows».

[8] T. Lou, *A comparison of Android Native App Architecture,* Aalto University, 2016.

[9] M. R. J. Qureshi ve F. Sabir, *A COMPARISON OF MODEL VIEW CONTROLLER AND MODEL VIEW PRESENTER*, 2013.

[10] E. Kazan, *TASARIM DESENİ KULLANILARAK GELİŞTİRİLEN YAZILIM İLE KULLANILMADAN GELİŞTİRİLEN YAZILIMIN PERFORMANS ANALİZİ,* TURGUT ÖZAL ÜNİVERSİTESİ, 2015.

[11] C. Ünlü, *THE EFFECTS OF TEST DRIVEN DEVELOPMENT ON SOFTWARE PRODUCTIVITY AND SOFTWARE QUALITY,* Orta Doğu Teknik Üniversitesi, 2008.

[12] R. C. Martin, Agile software development: principles, patterns, and practices, Pearson Education, Inc., 2002.

[13] B. Aydınöz, *TASARIM ÖRÜNTÜLERİNİN NESNE TABANLI METRİKLER VE YAZILIM,* Orta Doğu Teknik Üniversitesi, 2006.

[14] M. İ. Akalın, *YAZILIMIN EVRİMLEŞME SÜRECİNDE TASARIM ÖRÜNTÜLERİNİN YAZILIM KALİTESİ ÜZERİNDEKİ ETKİLERİNİN İNCELENMESİ,* TRAKYA ÜNİVERSİTESİ, 2014.

[15] T. Tezel, Temmuz 2018. [Çevrimiçi]. Mevcut: https://medium.com/android-t%C3%BCrkiye/solid-prensipleri-5d2ef01f4eeb. [Erişildi: Mayıs 2019].

[16] G. Öztürk, Aralık 2016. [Çevrimiçi]. Mevcut: https://medium.com/@techmostal/solid-yaz%C4%B1l%C4%B1m-geli%C5%9Ftirme-prensipleri-86a236f6e961. [Erişildi: Mayıs 2019].

[17] H. Özel, Aralık 2018. [Çevrimiçi]. Mevcut: https://yazilimsanati.net/yagni-prensibi/. [Erişildi: Mayıs 2019].

[18] J. S. Miguel, Kasım 2017. [Çevrimiçi]. Mevcut: https://www.itexico.com/blog/software-development-kiss-yagni-dry-3-principles-to-simplify-your-life. [Erişildi: Mayıs 2019].

[19] E. Erol, Mayıs 2016. [Çevrimiçi]. Mevcut: https://erkanerol.github.io/post/kiss/. [Erişildi: Mayıs 2019].

[20] A. Hunt ve D. Thomas, The Pragmatic Programmer, Addison Wesley Longman, Inc., 1999.

[21] A. Çetinkaya, Ocak 2015. [Çevrimiçi]. Mevcut: http://devnot.com/2015/prensip-sahibi-yazilimlar-2-dry-soc/. [Erişildi: Mayıs 2019].

[22] E. Keskin, Şubat 2014. [Çevrimiçi]. Mevcut: https://kodcu.com/2014/02/fluent-interface/. [Erişildi: Mayıs 2019].

[23] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[24] E. Özdemir, Ocak 2018. [Çevrimiçi]. Mevcut: https://medium.com/@nerobianchi/tdd-nedir-965c1a26e68f. [Erişildi: Mayıs 2019].

[25] A. Çetinkaya, Mart 2015. [Çevrimiçi]. Mevcut: http://devnot.com/2015/tdd-test-driven-development-gercekten-zor-mu/. [Erişildi: Mayıs 2019].

[26] T. Örsbaş, 5 2010. [Çevrimiçi]. Mevcut: http://www.yazilimprojesi.com/index.php?q=yapi-struct-ve-sinif-class-arasindaki-farklar. [Erişildi: 5 2019].

[27] H. Özer, Ekim 2012. [Çevrimiçi]. Mevcut: http://harunozer.com/makale/prototype_tasarim_deseni__prototype_design_pattern.htm. [Erişildi: Mayıs 2019].

[28] H. Özer, Ekim 2012. [Çevrimiçi]. Mevcut: http://harunozer.com/makale/tasarim_desenleri__design_patterns.htm. [Erişildi: Mayıs 2019].

[29] R. C. ve G. B., Mayıs 2018. [Çevrimiçi]. Mevcut: https://rubygarage.org/blog/swift-design-patterns. [Erişildi: Mayıs 2019].

[30] «bogotobogo.com,» [Çevrimiçi]. Mevcut: https://www.bogotobogo.com/DesignPatterns/mvc_model_view_controller_pattern.php. [Erişildi: Mayıs 2019].

[31] «interserver.net,» [Çevrimiçi]. Mevcut: https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/. [Erişildi: Mayıs 2019].

[32] «docs.microsoft.com,» Ekim 2011. [Çevrimiçi]. Mevcut: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff709839(v=pandp.10). [Erişildi: Mayıs 2019].

[33] J. Gossman, Ekim 2005. [Çevrimiçi]. Mevcut: https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/. [Erişildi: Mayıs 2019].

[34] Ö. Akkoca, Temmuz 2018. [Çevrimiçi]. Mevcut: https://medium.com/t%C3%BCrkiye/mvvm-giri%C5%9F-ios-swift-4-6aac78d65d19. [Erişildi: Mayıs 2019].

[35] D. Hall , Temmuz 2015. [Çevrimiçi]. Mevcut: http://www.danielhall.io/the-problems-with-mvvm-on-ios.

[36] S. M. Alam, Mart 2017. [Çevrimiçi]. Mevcut: https://medium.com/@smalam119/viper-design-pattern-for-ios-application-development-7a9703902af6. [Erişildi: Mayıs 2019].

ÖZGEÇMİŞ

30 Temmuz 1991 tarihi, İstanbul İli Üsküdar İlçesi doğumluyum. İlk, Orta ve Liseyi yine aynı ilçede tamamladıktan sonra, Bahçeşehir Üniversitesi Meslek Yüksek Okulu'nda Bilgisayar Programcılığı programında Üniversite öğretimime başladım. İngilizce hazırlık eğitimimi bu üniversitenin ilk senesinde aldım. 2012 yılında bu Üniversite ve programdan mezun olduktan sonra Beykent Üniversitesi'nde Bilgisayar Mühendisliği bölümünde okuyup 2015 yılında mezun oldum. Lisans eğitimin ardından 2017 yılının bahar döneminde Beykent Üniversitesi Bilgisayar Mühendisliği Anabilim Dalında Tezli Yüksek Lisans eğitimime başladım.

Öğrencilik hayatımda içinde Microsoft'un da bulunduğu bazı şirketlerde staj yapmamla birlikte Lisans öğrenimim sırasında Microsoft'un Microsoft Student Program'ına seçilerek Microsoft ile birlikte gönüllü öğrenci çalışmaları yaptım. Profesyonel hayatımda çeşitli büyük kurumsal firmalarda IOS Uygulama Geliştiricisi pozisyonunda çalıştım ve halen kurumsal bir firmada aynı pozisyonda çalışmaktayım.

Nuri Gökhan ALP