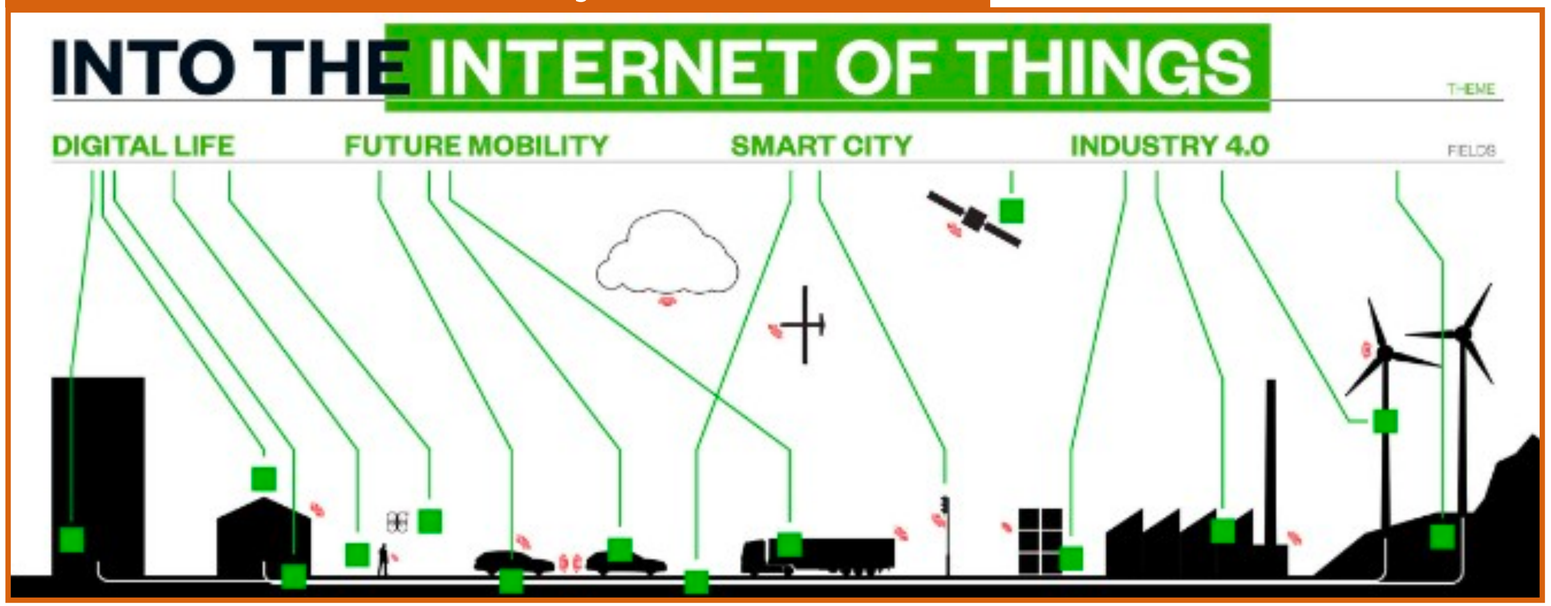
CODE QUALITY

https://www.onurdayibasi.com/

Yazılım Mimarisi Özet

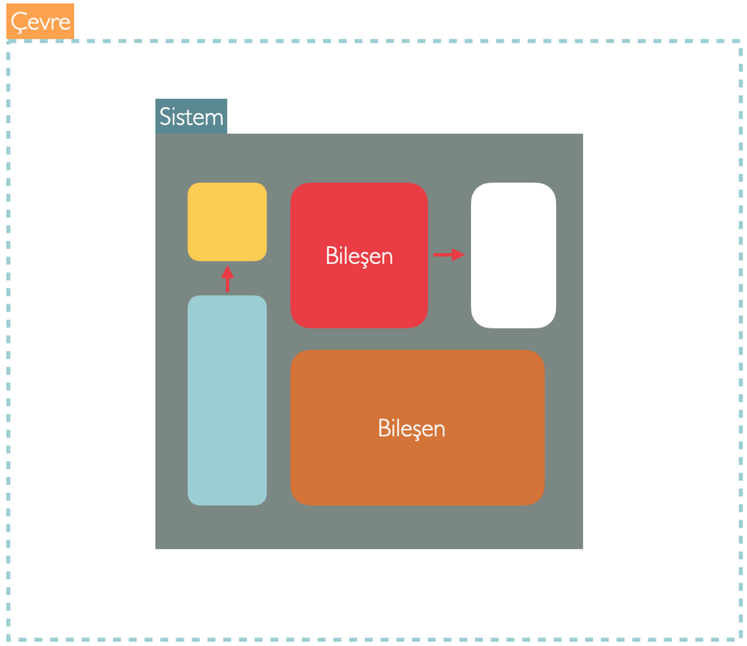
Yazılım Sistemlerinin Yaşamımızdaki Yeri

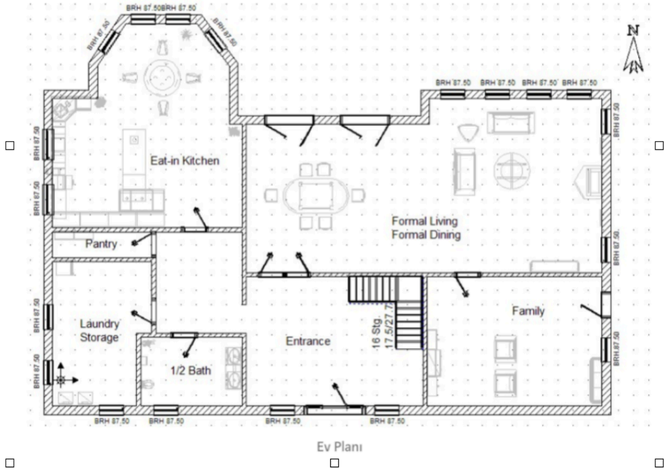


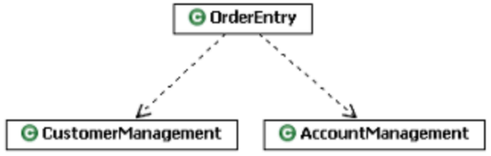
Yazılım Sistemleri Canlıdır

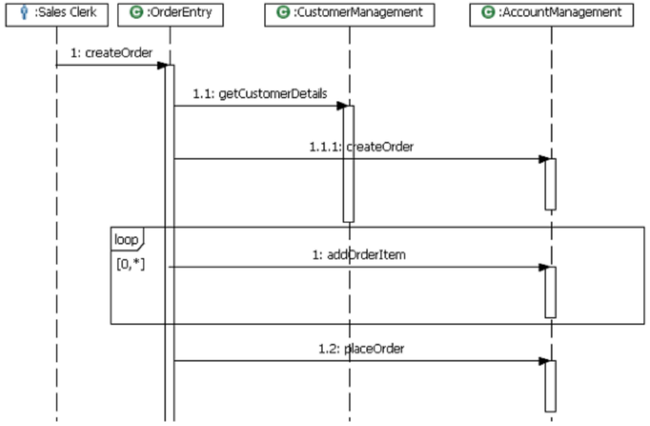
Doğar → Gelişir → Büyür → Yaşlanır → Ölür

Güncelleme →


Çevre
Sistem
Bileşen
Bileşen


N
Eat-in Kitchen
Pantry
Formal Living
Formal Dining
Laundry
Storage
1/2 Bath
Entrance
16 Stg
17.5/27.7
Family
Ev Planı


OrderEntry
CustomerManagement
AccountManagement


:Sales Clerk
:OrderEntry
:CustomerManagement
:AccountManagement
1: createOrder
1.1: getCustomerDetails
1.1.1: createOrder
loop
[0,*]
1: addOrderItem
1.2: placeOrder

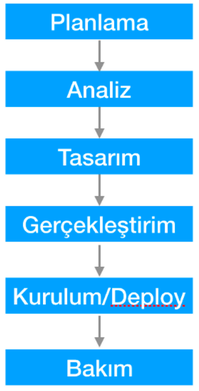
PAYDAŞLAR

Son Kullanıcı
Sistem Yöneticisi
Pazarlama Market
Müşteri
Yazılım Geliştiriciler
Proje Yöneticisi
Bakım Ekibi
Mimar

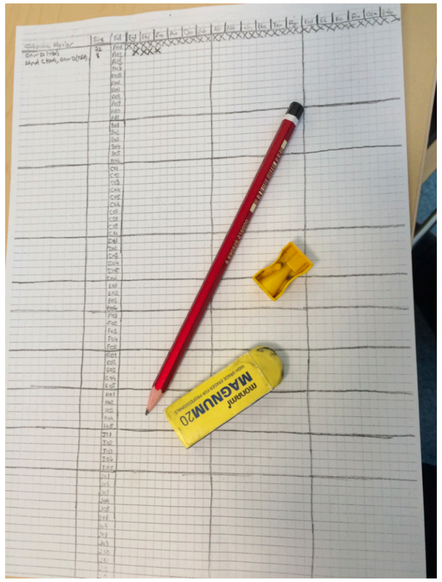
Microservices
Monalitic
Layered
Component-Based
SOA
MVC Pattern
Flux Pattern
Saga Pattern

Kurum/İşletme
Sistem
Yazılım
Donanım
Organizasyon
Bilgi

Fazlar


Planlama
Analiz
Tasarım
Gerçekleştirim
Kurulum/Deploy
Bakım

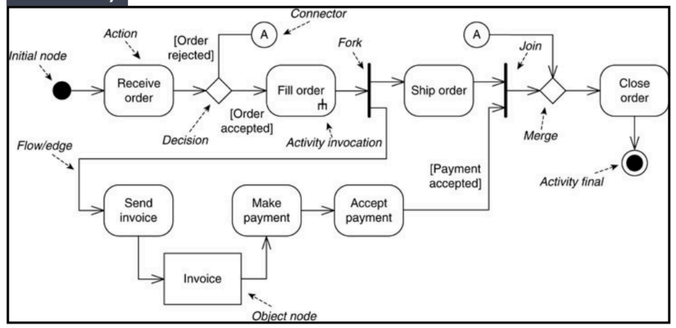
CONCLUDING THE FELLOWSHIP
FOLLOW-UP AFTER THE FELLOWSHIP IS OVER



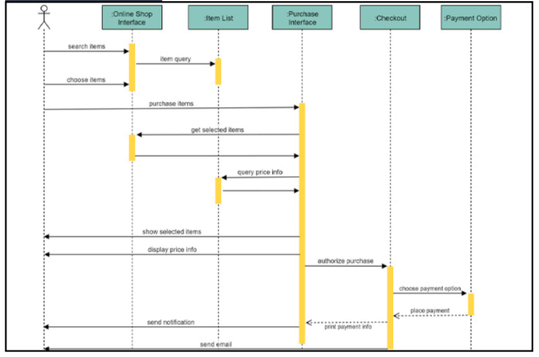
Behavioral Diagram ile Davranışı Modellemesi Yapılır.

- Activity,
- State Machine,
- Use Case,
- Timing, Sequence,
- Communication,
- Interaction Overview Diagrams

Activity

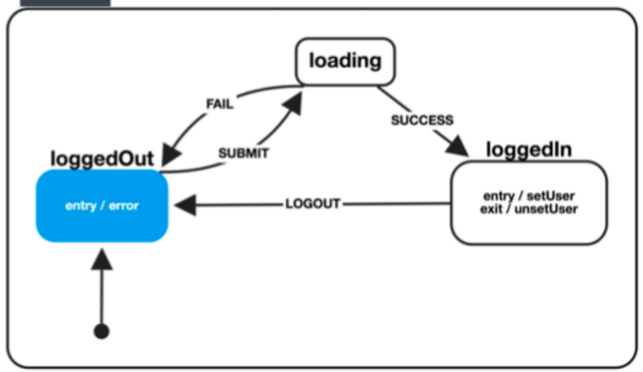


Sequence

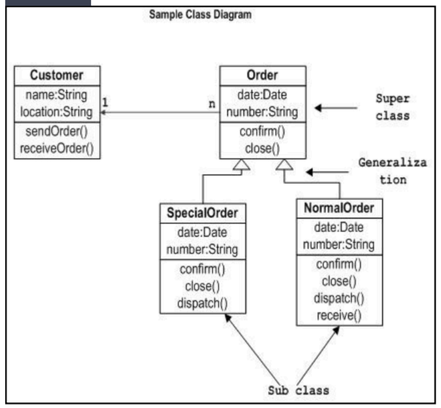


UseCase

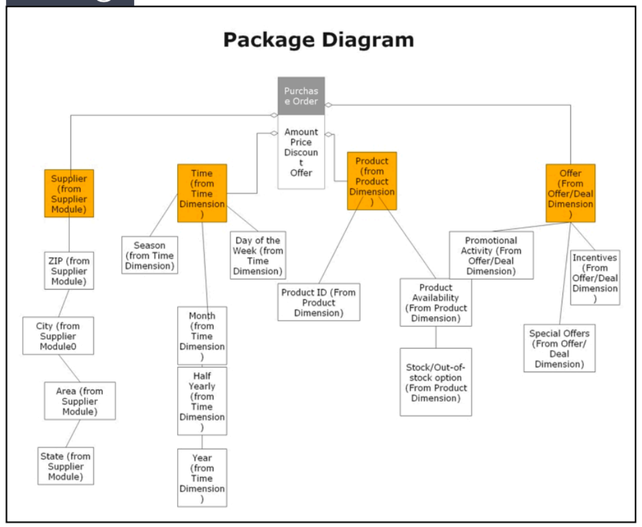
State



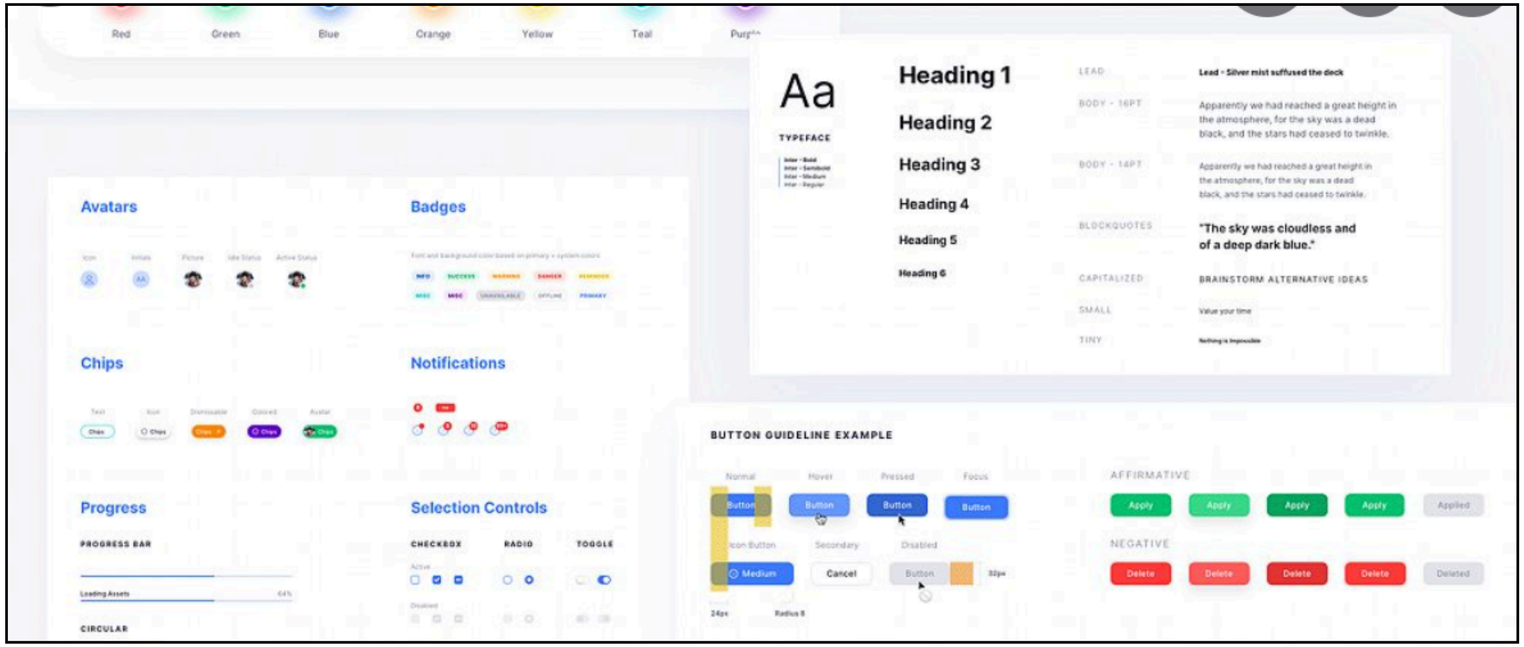
Class

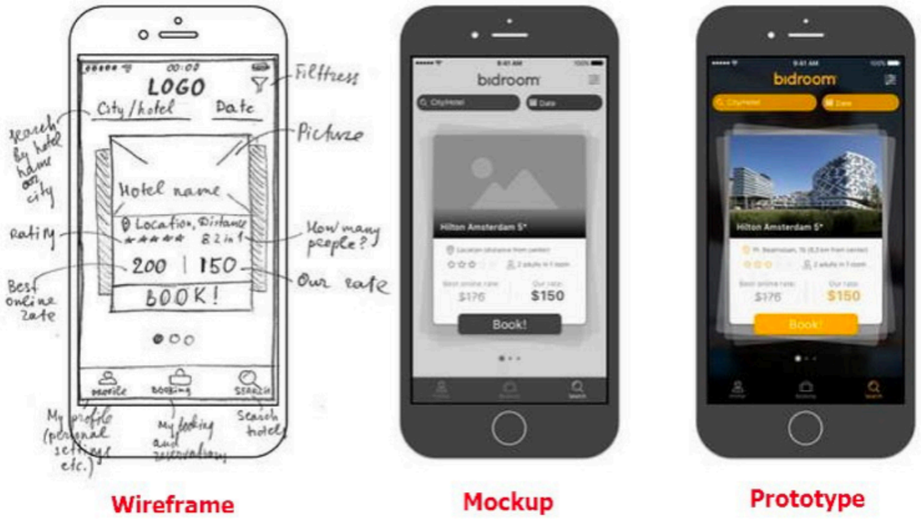


Package



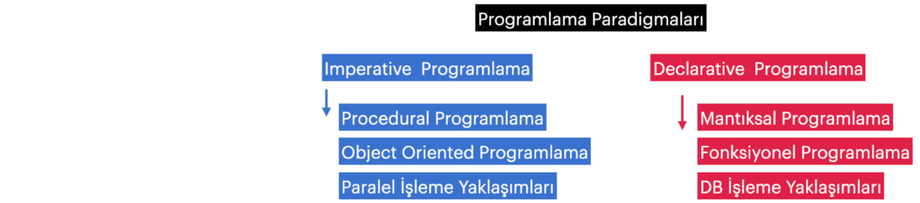
- Design System , Pattern Library, Style Guiding
- Sketch, Wireframe, Mockup, Prototype

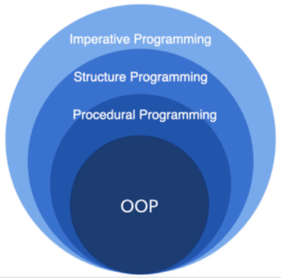




Wireframe Mockup Prototype

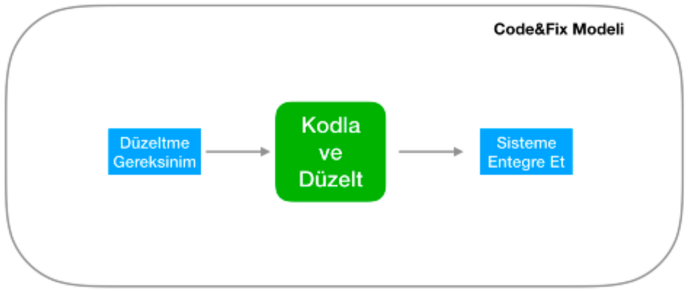
Gerçekleştirme Fazı

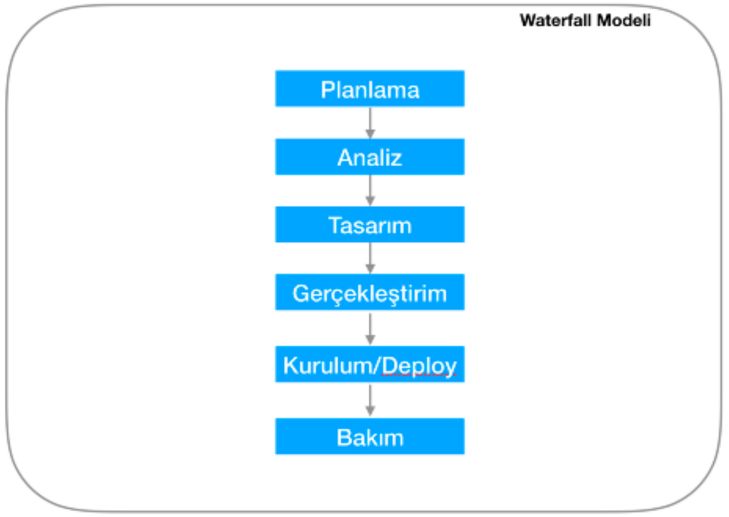


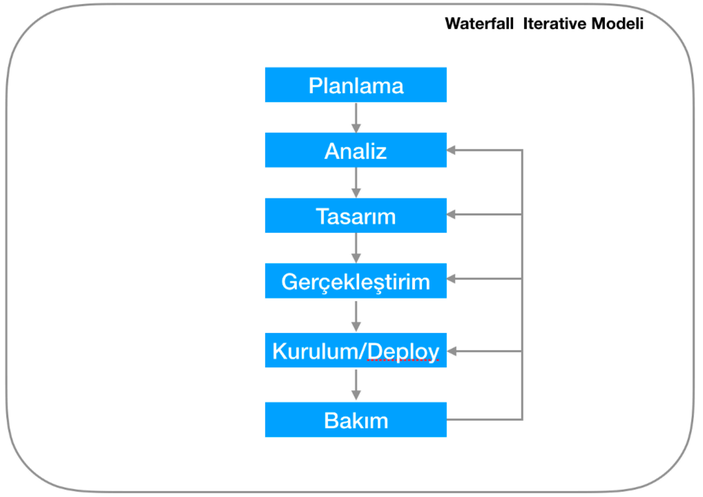


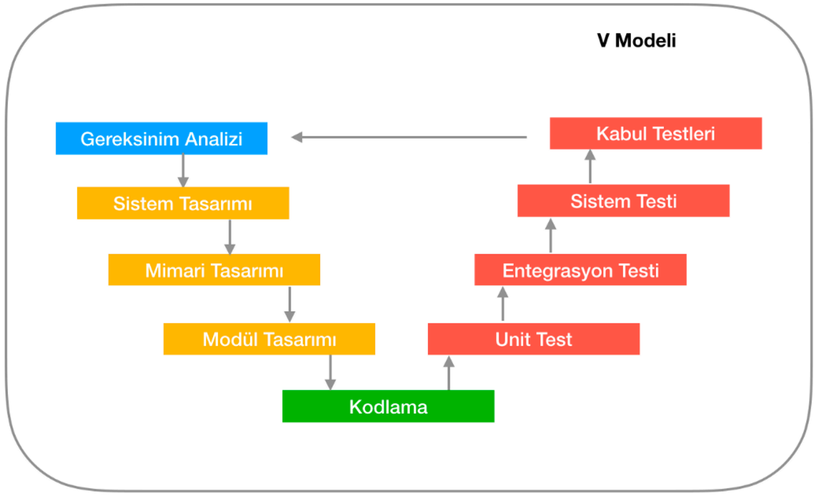
Yazılım Geliştirme Modellerinden Örnekler

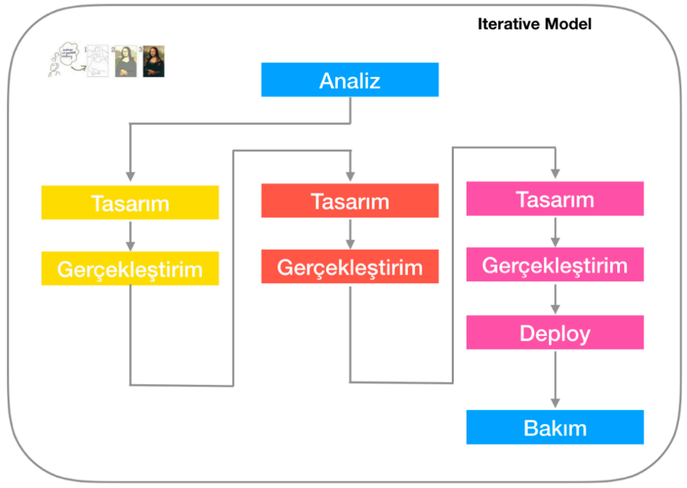
Code & Fix
Waterfall
Waterfall Iterative
V Model
Iterative
Incremental
Evolutionary
Spiral
Rational Unified Process
Agile (XP, Kanban, Scrum)

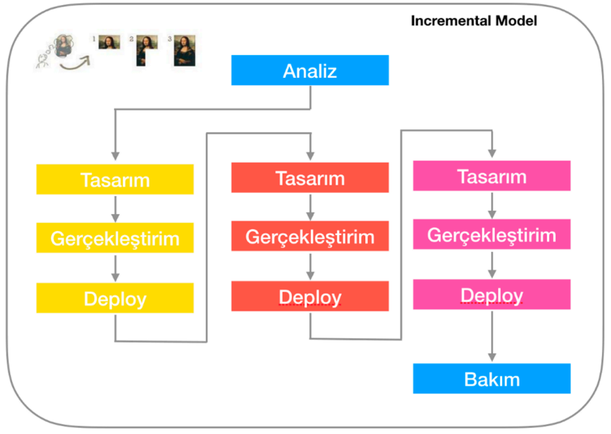




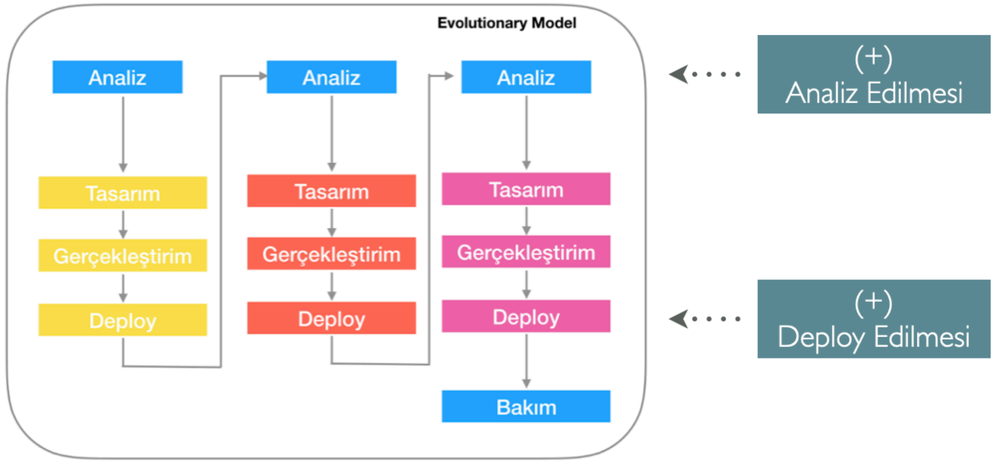


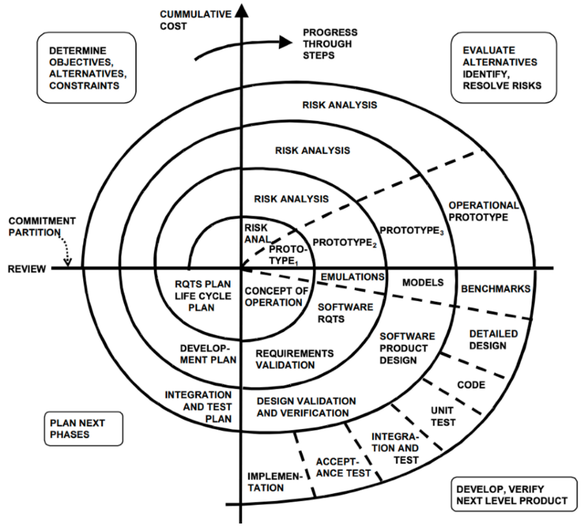


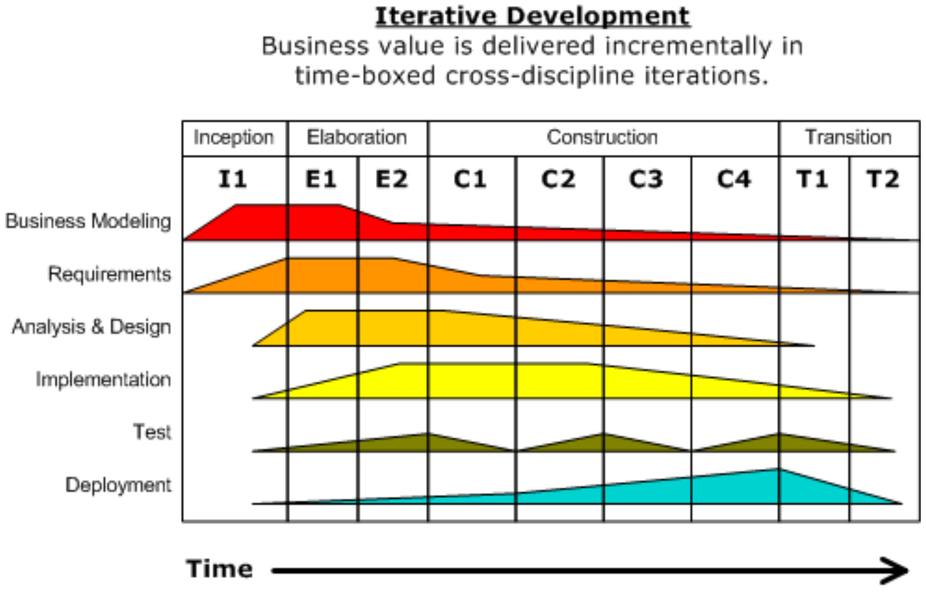


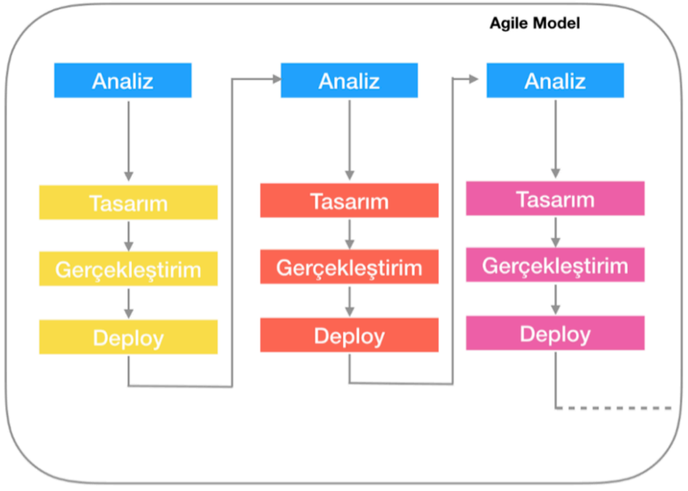


(+)
Deploy Edilmesi


Evolutionary Model
Analiz
Tasarım
Gerçekleştirim
Deploy
Analiz
Tasarım
Gerçekleştirim
Deploy
Analiz
Tasarım
Gerçekleştirim
Deploy
Bakım
(+)
Analiz Edilmesi
(+)
Deploy Edilmesi

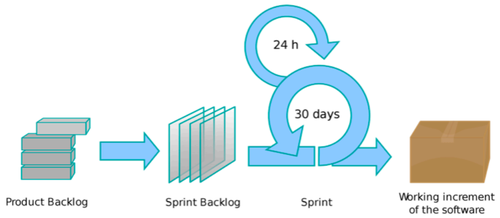

CUMMULATIVE COST
PROGRESS THROUGH STEPS
DETERMINE OBJECTIVES, ALTERNATIVES, CONSTRAINTS
EVALUATE ALTERNATIVES IDENTIFY, RESOLVE RISKS
RISK ANALYSIS
RISK ANALYSIS
RISK ANALYSIS
COMMITMENT PARTITION
REVIEW
RISK ANAL
PROTO-TYPE1
PROTOTYPE2
PROTOTYPE3
OPERATIONAL PROTOTYPE
RQTS PLAN LIFE CYCLE PLAN
CONCEPT OF OPERATION
EMULATIONS
SOFTWARE RQTS
MODELS
BENCHMARKS
DEVELOP-MENT PLAN
REQUIREMENTS VALIDATION
SOFTWARE PRODUCT DESIGN
DETAILED DESIGN
CODE
INTEGRATION AND TEST PLAN
DESIGN VALIDATION AND VERIFICATION
UNIT TEST
INTEGRA-TION AND TEST
ACCEPT-ANCE TEST
IMPLEMEN-TATION
PLAN NEXT PHASES
DEVELOP, VERIFY NEXT LEVEL PRODUCT


Iterative Development
Business value is delivered incrementally in time-boxed cross-discipline iterations.
Inception
Elaboration
Construction
Transition
I1
E1
E2
C1
C2
C3
C4
T1
T2
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Time

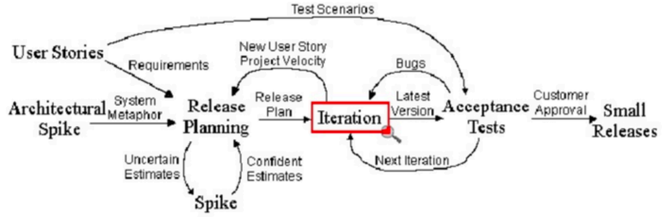

Agile Model
Analiz
Tasarım
Gerçekleştirim
Deploy
Analiz
Tasarım
Gerçekleştirim
Deploy
Analiz
Tasarım
Gerçekleştirim
Deploy

KANBAN

Scrum



XP



Çevik Yazılım Geliştirme Manifestosu

Bizler daha iyi yazılım geliştirme yollarını
uygulayarak ve başkalarının da uygulamasına yardım ederek ortaya çıkartıyoruz.
Bu çalışmaların sonucunda:

Süreçler ve araçlardan ziyade **bireyler ve etkileşimlere**
Kapsamlı dökümantasyondan ziyade **çalışan yazılıma**
Sözleşme pazarlıklarından ziyade **müşteri ile işbirliğine**
Bir plana bağlı kalmaktan ziyade **değişime karşılık vermeye**
değer vermeye kanaat getirdik.

Özetle, sol taraftaki maddelerin değerini kabul etmekle birlikte,
sağ taraftaki maddeleri daha değerli bulmaktayız.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Çevik Yazılımın Prensipleri

1. En önemli önceliğimiz değerli yazılımın erken ve devamlı teslimini sağlayarak müşterileri memnun etmektir.

2. Değişen gereksinimler yazılım sürecinin son aşamalarında bile kabul edilmelidir.Çevik süreçler değişimi müşterinin rekabet avantajı için kullanır.

3. Çalışan yazılım, tercihen kısa zaman aralıkları belirlenerek birkaç haftada ya da birkaç ayda bir düzenli olarak müşteriye sunulmalıdır.

4. İş süreçlerinin sahipleri ve yazılımcılar proje boyunca her gün birlikte çalışmalıdırlar.

5. Projelerin temelinde motive olmuş bireyler yer almalıdır. Onlara ihtiyaçları olan ortam ve destek sağlanmalı, işi başaracakları konusunda güven duyulmalıdır.

6. Bir yazılım takımında bilgi alışverişinin en verimli ve etkin yöntemi yüz yüze iletişimdir.

Çevik Yazılımın Prensipleri

7. Çalışan yazılım ilerlemenin birincil ölçüsüdür.

8. Çevik süreçler sürdürülebilir geliştirmeyi teşvik etmektedir. Sponsorlar, yazılımcılar ve kullanıcılar sabit tempoyu sürekli devam ettirebilmelidir.

9. Teknik mükemmeliyet ve iyi tasarım konusundaki sürekli özen çevikliği artırır.

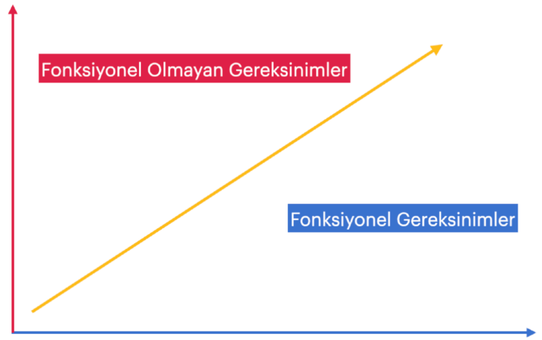
10. Sadelik, yapılmasına gerek olmayan işlerin mümkün olduğunca arttırılması sanatı, olmazsa olmazlardandır.

11. En iyi mimariler, gereksinimler ve tasarımlar kendi kendini örgütleyen takımlardan ortaya çıkar.

12. Takım, düzenli aralıklarla nasıl daha etkili ve verimli olabileceğinin üzerinde düşünür ve davranışlarını buna göre ayarlar ve düzenler.

1. Fonksiyonel ve Fonksiyonel Olmayan Gereksinimleri Doğru Şekilde **İlerletmemek**

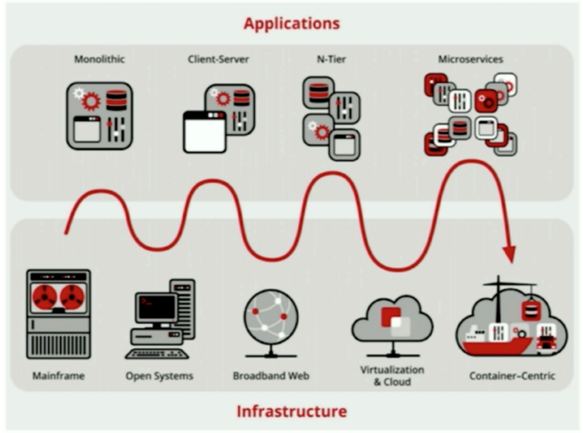
Fonksiyonel gereksinimler bir sistemin nasıl davranacağı, iş kurallarını, akışı için gereksinimleri tanımlarken,



2. Gereksimimleri ve Sorunları Doğru Alanlarda **Çözmemek**

Yazılım mimari sorunları deneyiminin fazla olduğu alanlarda çözmeye çalışır. Çünkü bu yöntem ile kendisini daha güvende hisseder. Sorunu hangi alanda, hangi araçlara göre çözeceğiz ?

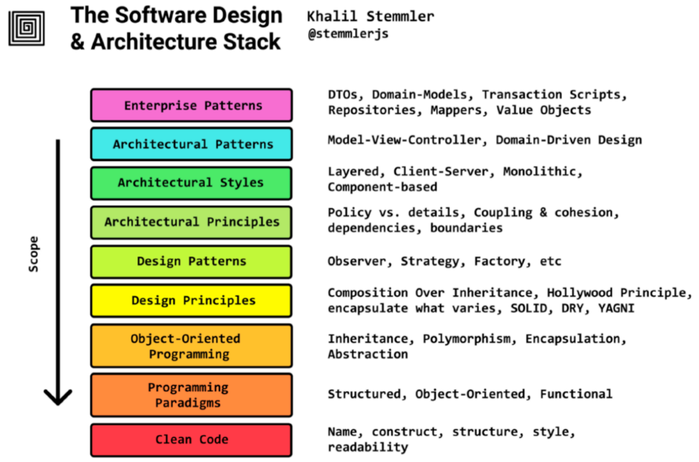
3.Değişime ve Yeni İsteklere Karşı Mimariniz, Ufak Yamalar ile Projenizi Ne Kadar Ayakta Tutabilirsiniz ?

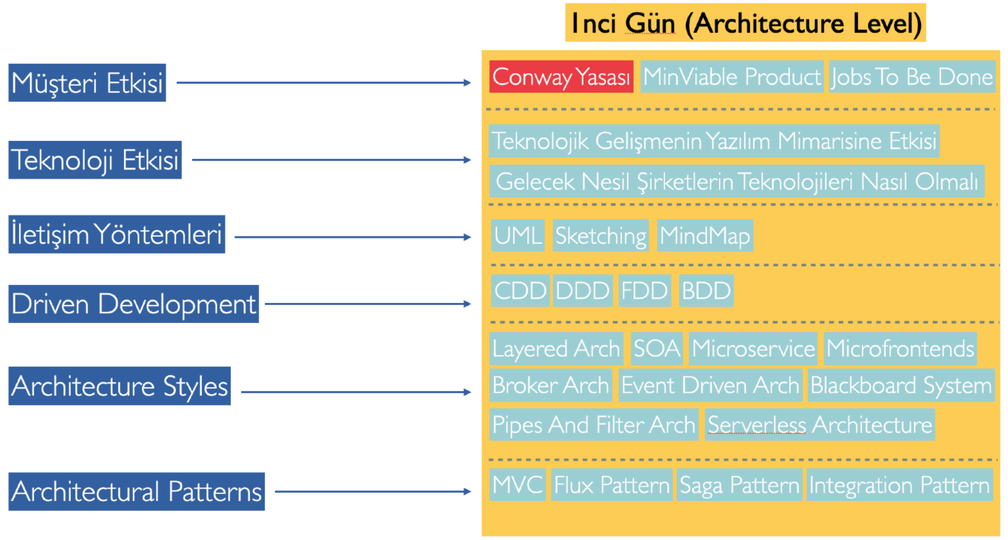


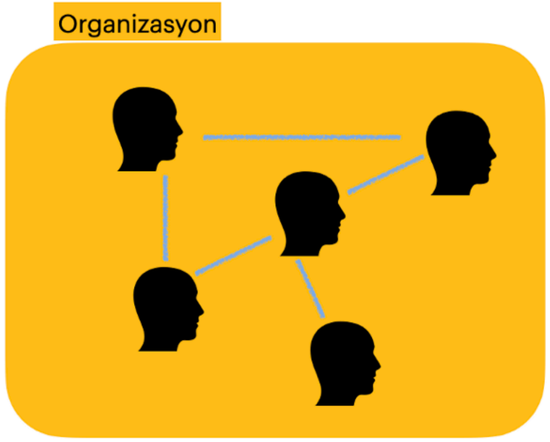
Dinamik Yapılar (Oyun Motorları, AdHoc Report NoCode LowCode Aracları)

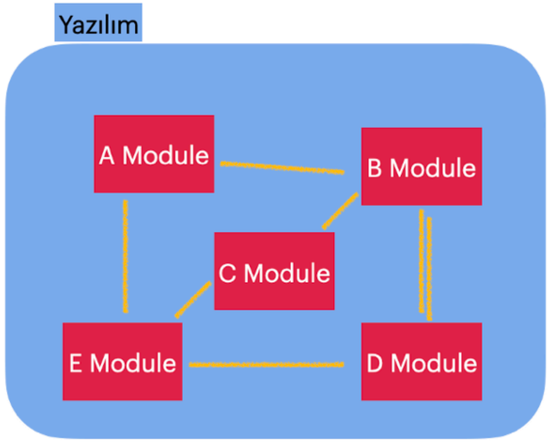
Mimari Bilgiyi Şirket İçerisinde Yayma

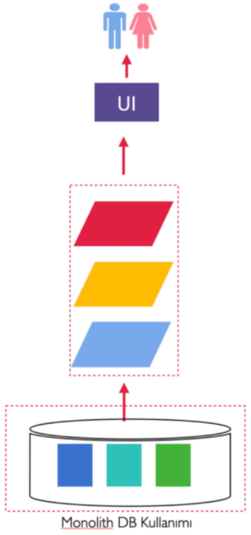
Yazılım Mimarisi Stack



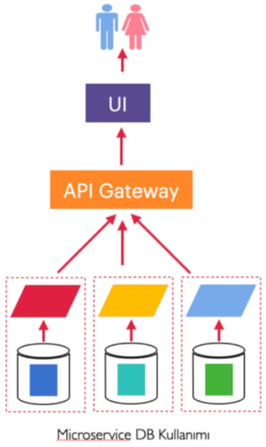




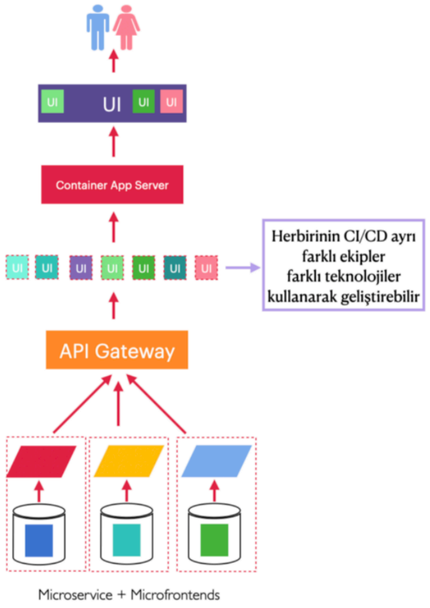




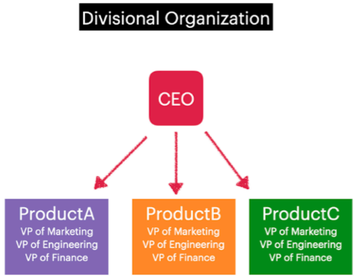
Monolith DB Kullanımı

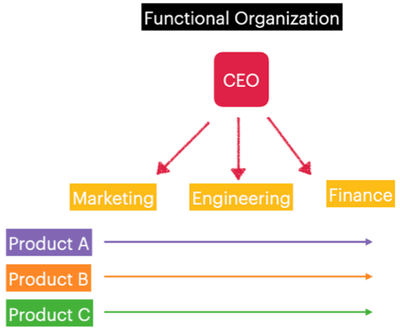


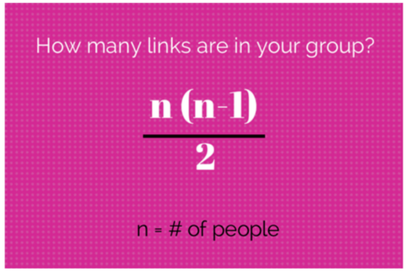
Microservice DB Kullanımı



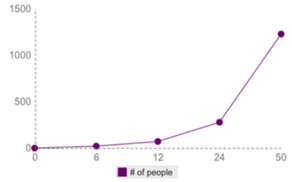
Microservice + Microfrontends



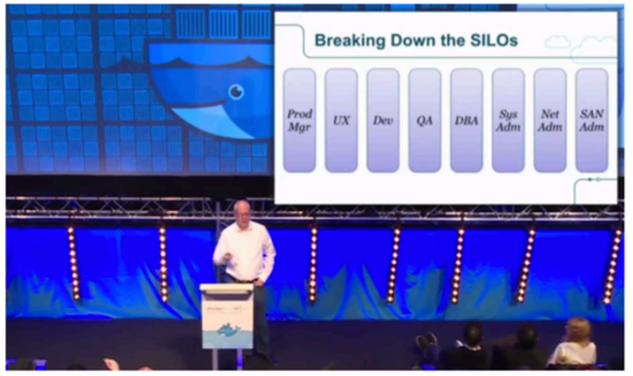




Bu karmaşıklığı aşağıdaki grafikte daha net olarak görebiliriz.

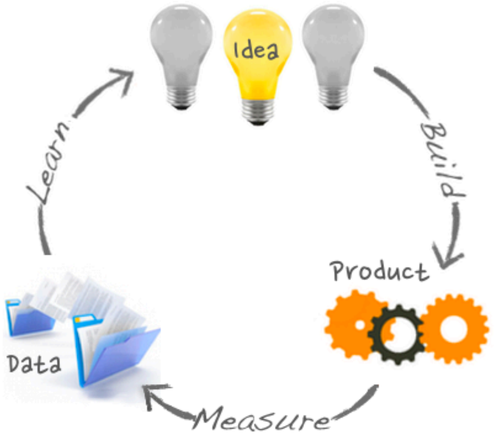


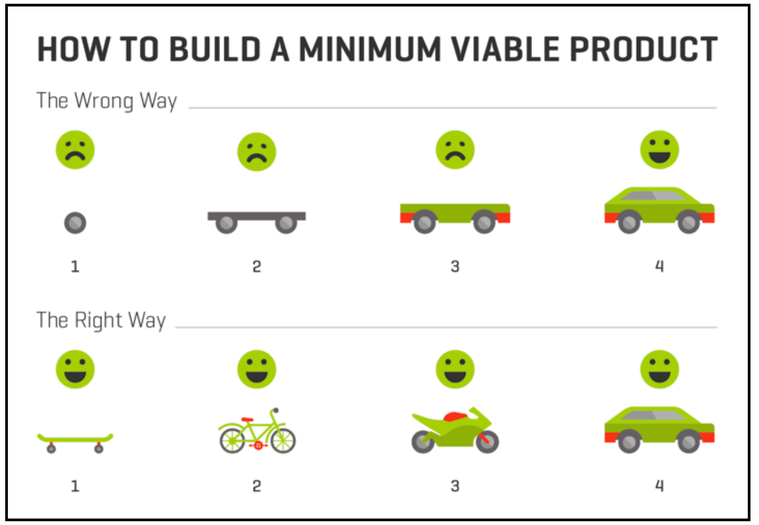
Yukarıdaki bölümleme bizim eskiden beri gelen klasik IT bölümlememizin dışında olan bir yönelim. Ama bu yönelimler DevOps'un giderek gelişmesi, ekip elemanlarının farklı sorumlulukları alabilmesi zamanla kırılacak ve aşağıdaki resimden yavaş yavaş 2PT 'ye evrilecektir.

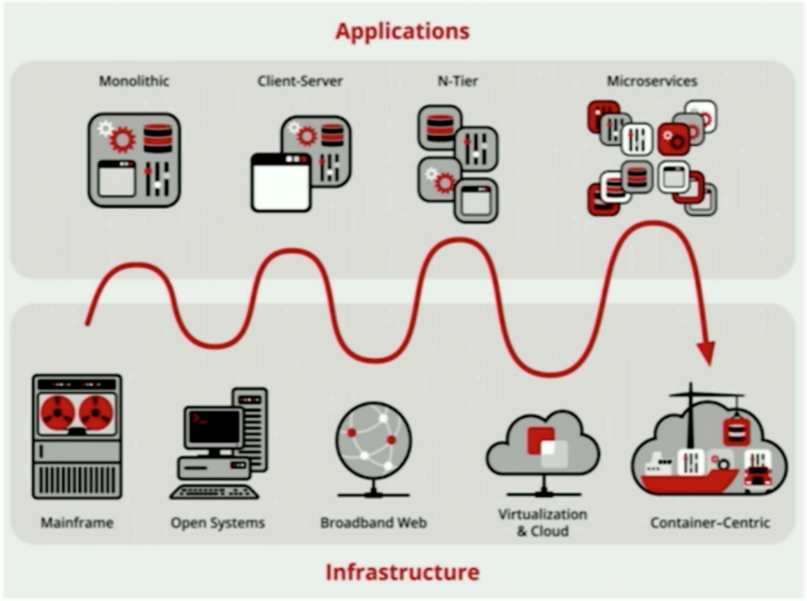


Eski Yönelim

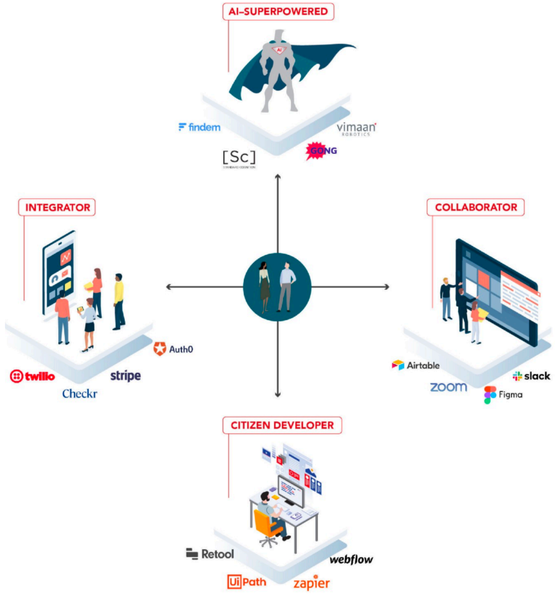
Minimum Viable Product



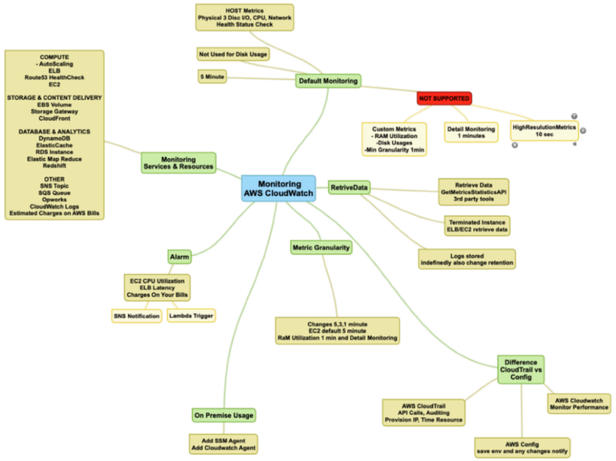




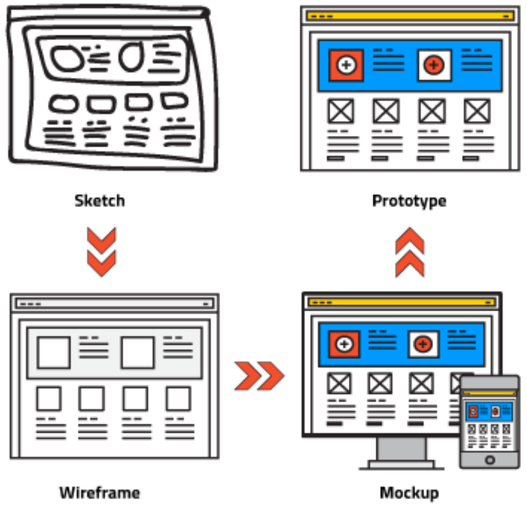
THE AGILE WORKPLACE OFFERS SMARTER MODELS OF WORK

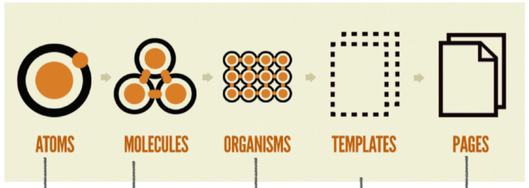


Monitoring (AWS CloudWatch)



AWS CloudWatch





Atoms: Web sayfasını oluşturan en küçük elemanlardır. Daha fazla parçalanamazlar. HTML tags ; ***label, input, button, div, canvas …***

Organisms: Birden fazla molekülün, bazen aynı türdeki molekül gruplarının tekrar ederek oluşturdukları yapılardır. Örneğin; **NavigationBar, Table, ListItems vb..**

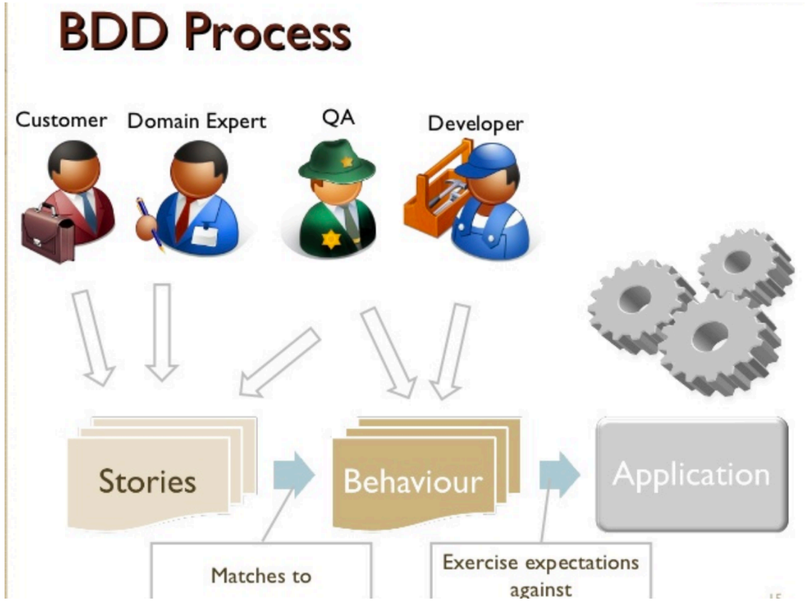
Molecules: Atomların bir araya gelerek oluşturdukları küçük bileşenlerdir. Örneğin bir arama molekül oluşturmak istiyorum;

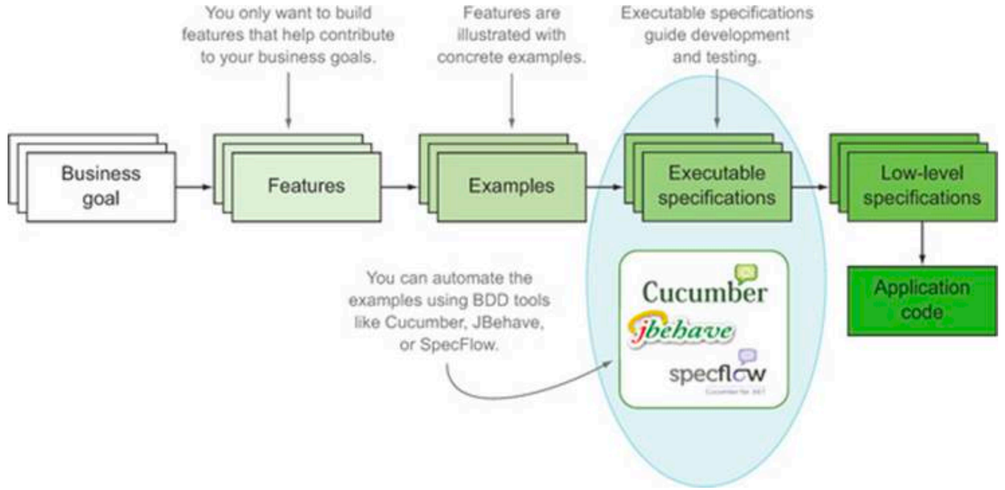
- Label, Input, Button birleştirerek bu molekülü oluşturabilirim
- ***Label*** kullanıcıya bu bileşenin nerede aradığını, ***Input*** arama yapmak istediği kelimeyi gireceği ortamı, ***Button*** ise arama işlemini tetiklemesini sağlar

Burada moleküllerin ***sadece bir iş yapması ve onuda iyi yapması*** üzerine tasarlanmalıdır.

Templates: Bir veya birden organizma/molekülün layout üzerindeki yerleşim yapısıdır.

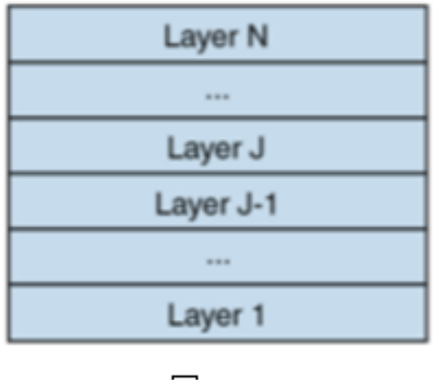
Pages: Template belli veri ile doldurularak gösterildiği gerçek sayfadır.

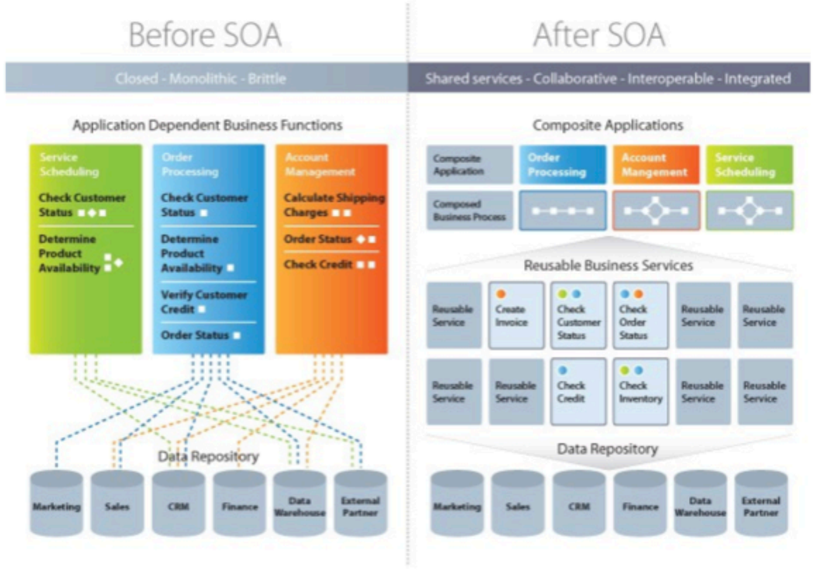


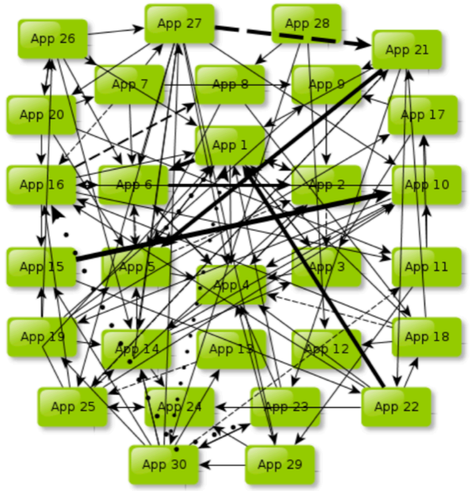


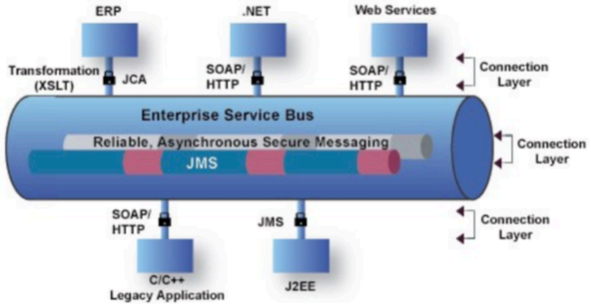
Inci Gün (Architecture Level)

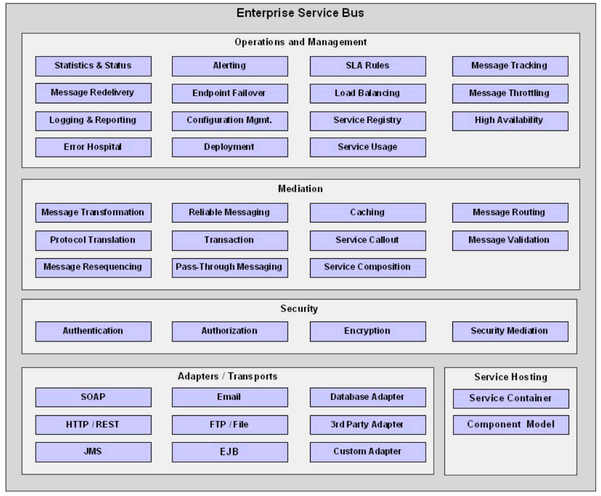
- Müşteri Etkisi
  - ~~Conway Yasası~~ ~~MinViable Product~~ ~~Jobs To Be Done~~
- Teknoloji Etkisi
  - ~~Teknolojik Gelişmenin Yazılım Mimarisine Etkisi~~
  - ~~Gelecek Nesil Şirketlerin Teknolojileri Nasıl Olmalı~~
- İletişim Yöntemleri
  - ~~UML~~ ~~Sketching~~ ~~MindMap~~
- Driven Development
  - ~~CDD~~ ~~DDD~~ ~~FDD~~ ~~BDD~~
- Architecture Styles
  - Layered Arch, SOA, Microservice, Microfrontends
  - Broker Arch, Event Driven Arch, Blackboard System
  - Pipes And Filter Arch, Serverless Architecture
- Architectural Patterns
  - MVC, Flux Pattern, Saga Pattern, Integration Pattern

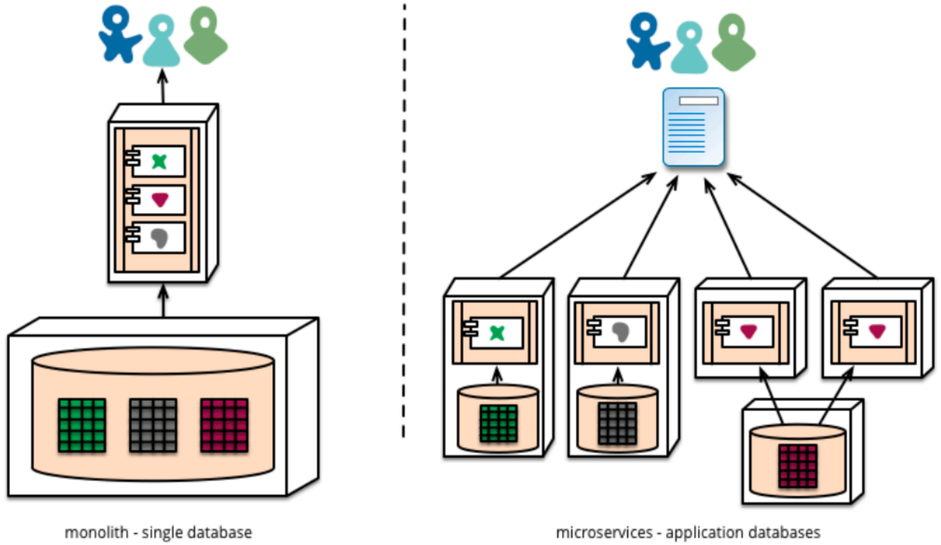

Layer N
...
Layer J
Layer J-1
...
Layer 1

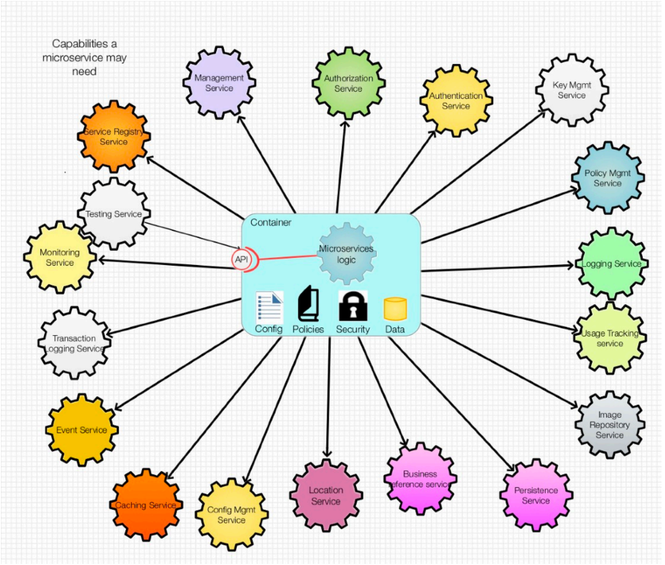

Before SOA
Closed - Monolithic - Brittle
Application Dependent Business Functions
Service Scheduling
Check Customer Status
Determine Product Availability
Order Processing
Check Customer Status
Determine Product Availability
Verify Customer Credit
Order Status
Account Management
Calculate Shipping Charges
Order Status
Check Credit
Data Repository
Marketing
Sales
CRM
Finance
Data Warehouse
External Partner
After SOA
Shared services - Collaborative - Interoperable - Integrated
Composite Applications
Composite Application
Order Processing
Account Mangement
Service Scheduling
Composed Business Process
Reusable Business Services
Reusable Service
Create Invoice
Check Customer Status
Check Order Status
Check Credit
Check Inventory
Data Repository
Marketing
Sales
CRM
Finance
Data Warehouse
External Partner


App 26
App 27
App 28
App 21
App 7
App 8
App 9
App 20
App 17
App 1
App 16
App 6
App 2
App 10
App 15
App 5
App 4
App 3
App 11
App 19
App 14
App 13
App 12
App 18
App 25
App 24
App 23
App 22
App 30
App 29

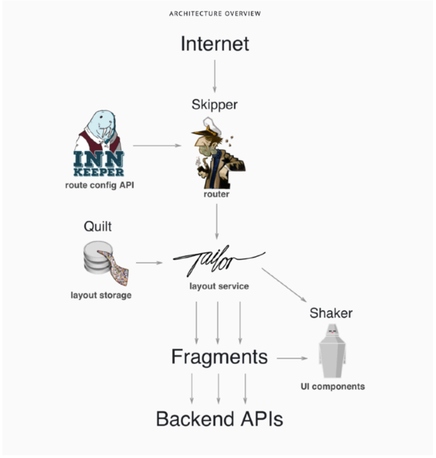

ERP
.NET
Web Services
Transformation (XSLT)
JCA
SOAP/ HTTP
SOAP/ HTTP
Connection Layer
Enterprise Service Bus
Reliable, Asynchronous Secure Messaging
JMS
Connection Layer
SOAP/ HTTP
JMS
Connection Layer
C/C++ Legacy Application
J2EE

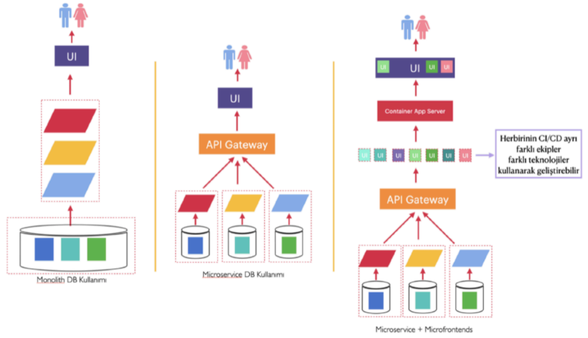

Enterprise Service Bus
Operations and Management
Statistics & Status
Alerting
SLA Rules
Message Tracking
Message Redelivery
Endpoint Failover
Load Balancing
Message Throttling
Logging & Reporting
Configuration Mgmt.
Service Registry
High Availability
Error Hospital
Deployment
Service Usage
Mediation
Message Transformation
Reliable Messaging
Caching
Message Routing
Protocol Translation
Transaction
Service Callout
Message Validation
Message Resequencing
Pass-Through Messaging
Service Composition
Security
Authentication
Authorization
Encryption
Security Mediation
Adapters / Transports
SOAP
Email
Database Adapter
HTTP / REST
FTP / File
3rd Party Adapter
JMS
EJB
Custom Adapter
Service Hosting
Service Container
Component Model



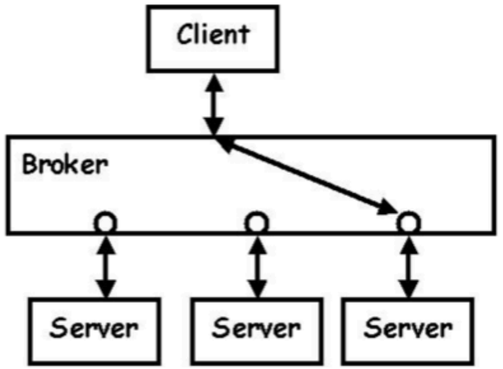


Microfrontends

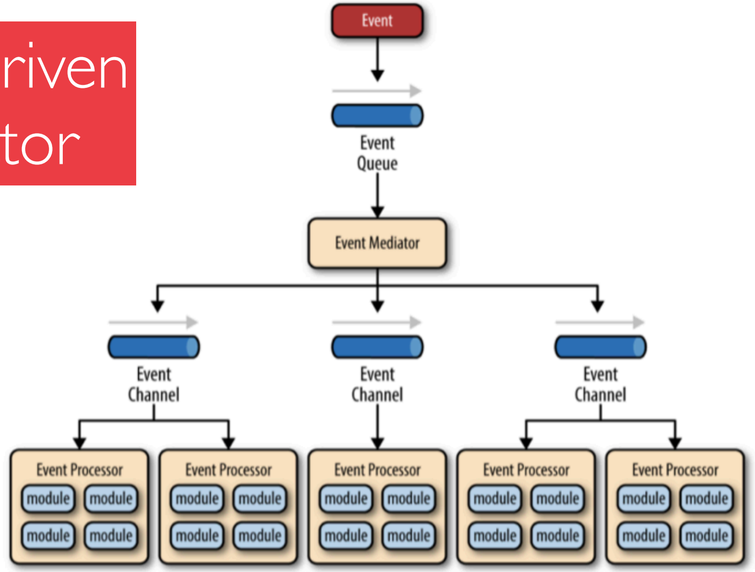


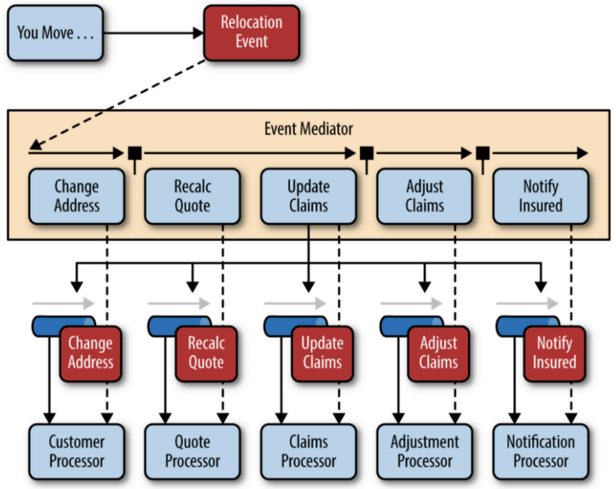


Broker

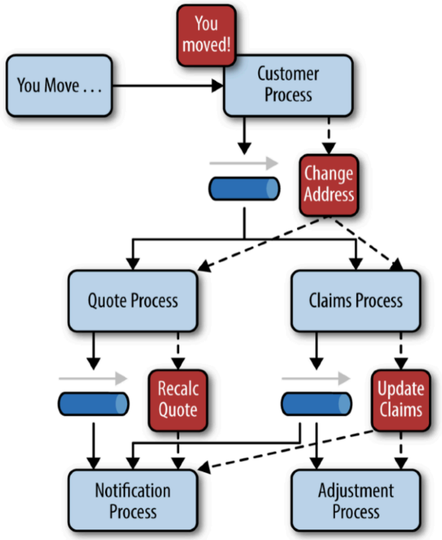


Event Driven Mediator

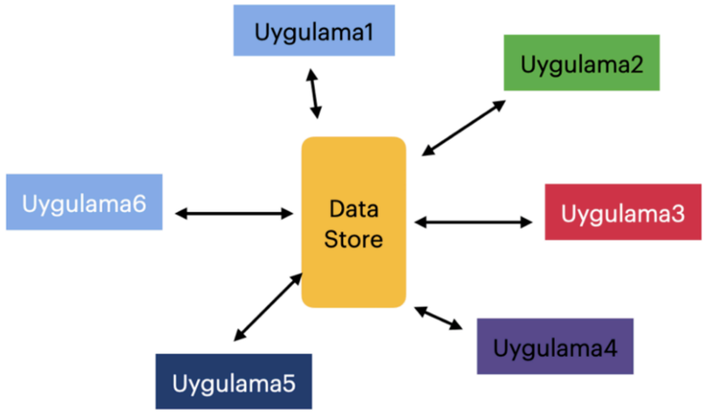




https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html (Mediator)



https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html (Broker)



Pipe And Filters

COMPUTE LAYER (HESAPLAMA KATMANI)

Lambda

APIGateway

Step Functions

DATA LAYER (VERİ KATMANI)

DynamoDB
DAX
Streams

S3

ElasticSearch

AppSync

MESSAGING AND STREAMING LAYER

SQS

SNS

Kinesis
Data Analytics

Firehose

USER MNG AND IDENTITY LAYER

Cognito

EDGE LAYER

Cloudfront

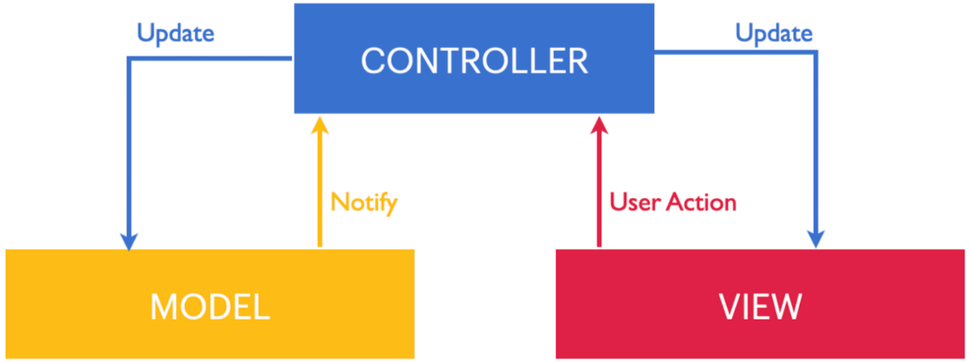
SYSTEM MONITORING AND DEPLOYMENT

CloudWatch

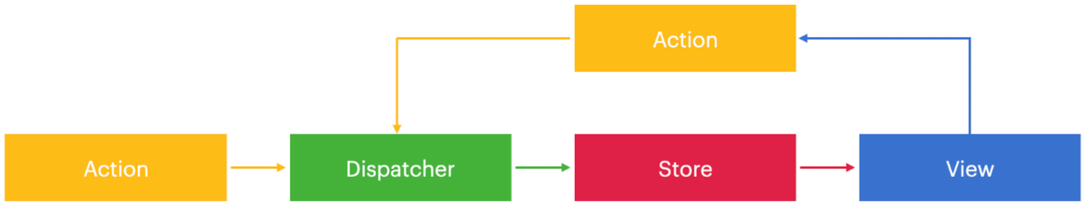
X-Ray

SAM

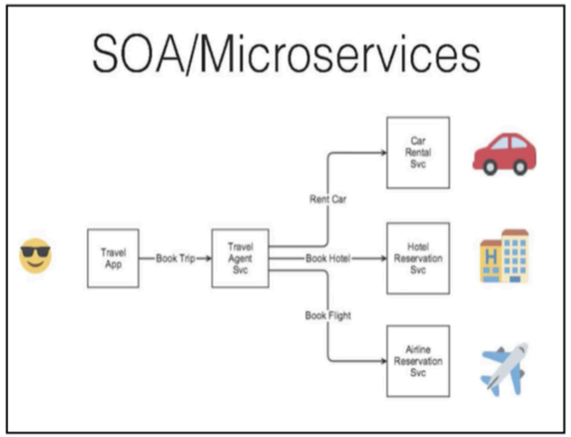
MVC


Update
CONTROLLER
Update
Notify
User Action
MODEL
VIEW

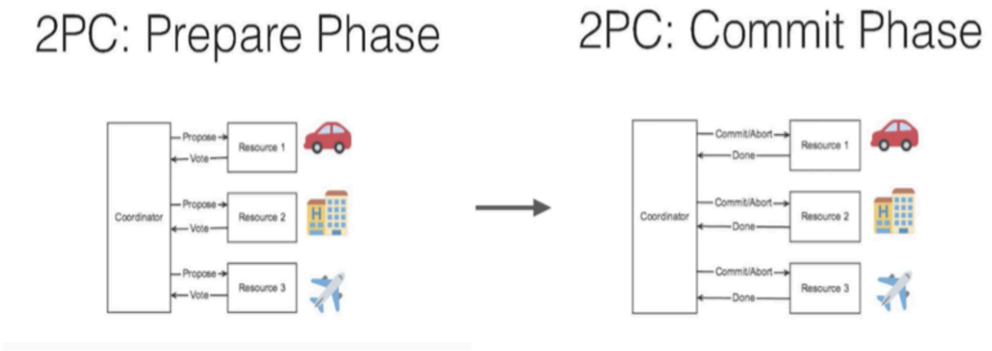
Flux


Action
Action
Dispatcher
Store
View

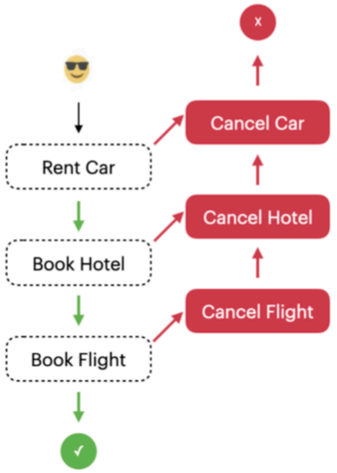
Distributed System



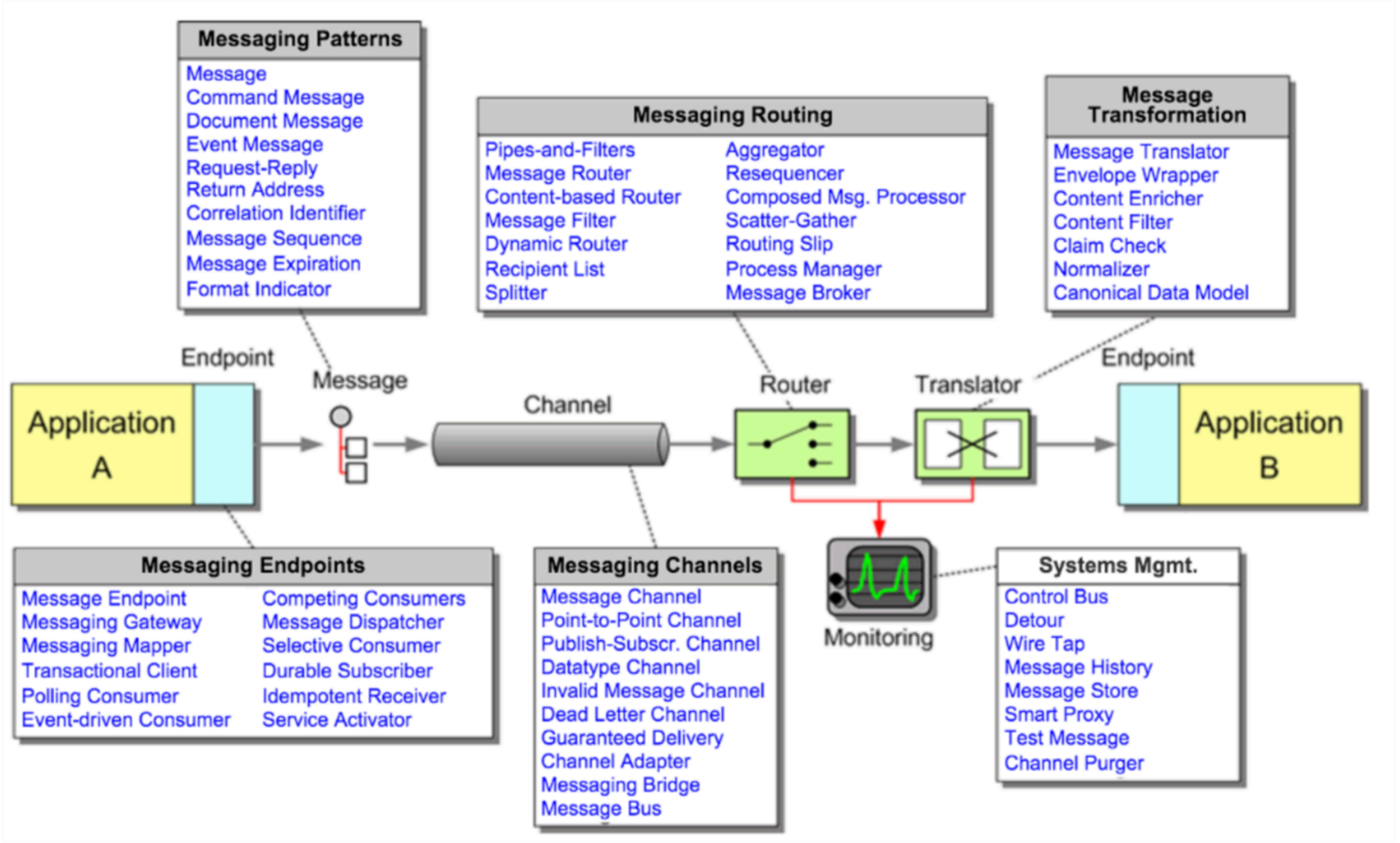
Dağıtık DB



Two Phase Commit



Saga Pattern


Messaging Patterns
Message
Command Message
Document Message
Event Message
Request-Reply
Return Address
Correlation Identifier
Message Sequence
Message Expiration
Format Indicator
Messaging Routing
Pipes-and-Filters
Message Router
Content-based Router
Message Filter
Dynamic Router
Recipient List
Splitter
Aggregator
Resequencer
Composed Msg. Processor
Scatter-Gather
Routing Slip
Process Manager
Message Broker
Message Transformation
Message Translator
Envelope Wrapper
Content Enricher
Content Filter
Claim Check
Normalizer
Canonical Data Model
Endpoint
Message
Channel
Router
Translator
Endpoint
Application A
Application B
Messaging Endpoints
Message Endpoint
Messaging Gateway
Messaging Mapper
Transactional Client
Polling Consumer
Event-driven Consumer
Competing Consumers
Message Dispatcher
Selective Consumer
Durable Subscriber
Idempotent Receiver
Service Activator
Messaging Channels
Message Channel
Point-to-Point Channel
Publish-Subscr. Channel
Datatype Channel
Invalid Message Channel
Dead Letter Channel
Guaranteed Delivery
Channel Adapter
Messaging Bridge
Message Bus
Monitoring
Systems Mgmt.
Control Bus
Detour
Wire Tap
Message History
Message Store
Smart Proxy
Test Message
Channel Purger

Bügünün Konuları

İÇERİK → 2nci Gün (Code Quality)

- Temiz ve Kaliteli Kod
  - Kod Kalitesini Neler Kötü Etkiler
  - Teknik Borç
- Kodu İyileştirme
  - Clean Code
  - Code Smells
  - Refactoring
- Kod Geliştirme Yöntemleri
  - AOP
  - TDD
  - SOLID
  - KISS
  - DRY
  - YAGNI
- Kodu Dayanıklı Geliştirmek
  - 12 Factor App
  - Reactive Programming
  - Chaos Eng
- Design Patterns
  - Design Patterns
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Anti Patterns
  - Anti Patterns
  - Spaghetti Code
  - Lava Flow Programming
  - Golden Hammer
  - Reinvent The Wheel

Temiz ve Kaliteli Kod

İÇERİK → 2nci Gün (Code Quality)

- Temiz ve Kaliteli Kod → Kod Kalitesini Neler Kötü Etkiler, Clean Code
- Kodu İyileştirme → Teknik Borç, Refactoring, Architecture Clearance
- Kod Geliştirme Yöntemleri → AOP, TDD, SOLID, KISS, DRY, YAGNI
- Kodu Dayanıklı Geliştirmek → 12 Factor App, Reactive Programming, Chaos Eng
- Design Patterns → Design Patterns, Creational Patterns, Structural Patterns, Behavioral Patterns
- Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

Kötü Kod

Bana göre kodun kalitesi sadece **yazılımcının elinde belirlenen bir kavram değildir**. Bu kodun kalitesi bakıldığı perspektif ve zaman aralığına göre değişiklik gösterir. Örneğin;

Çalışan bir uygulamada son kullanıcının karşısına **kritik bir hata** çıktığında kodu çok kötü yazmışlar denir. (Test Eksik)

Geliştirme aşamasında veya bakım aşamasında kodu başka arkadaşlara teslim ettiğimizde **kod anlaşılmıyorsa**, kodu devralan arkadaşlardan veryansın ve küfürler geliyorsa. (Temiz Kod yazılmamış)

Geliştirme aşamasında Müşteri bazı mantıklı ek istekler ile geldi ve bunların yapılma **maliyetleri çok yüksek** çıktığı ve kodun değişemeyeceğini söylüyorsak (Genişleyebilir/Değiştirilebilir Bir Kod Değil)

Kod geliştirme süresi proje takvimine ve planına **uymuyorsa** (Algorithma, 3rd Parti vb..)

Kod Kalitesini Neler **Kötü** Etkiler (Yönetimsel Başarısızlıklar)

Projenin **Gereksinimlerini** veya **Teknik Risk Alanlarının** Doğru Belirlenememesi

Proje **Ekibinin** doğru belirlenmemesi (Çok genç yeni mezun - bilgi birikimi az, Herkes çok deneyimli — Çatışma ve Ego, Hem deneyime, hem geri bildirime , hemde daha hızlı kod yazabilecek gençlere ihtiyaç var)

Yanlış **Zaman Planlaması** ve **Doğru Kaynakların** Kullanılmaması, İletişim Eksikliği (Analistler, Yazılım Mimari ve Geliştiriciler, İnsan Kaynakları, Muhasebe, Müşteri, Alan Uzmanı, Sistem Mühendisleri ile iletişimi doğru gerçekleştirememesi)

İşin maliyetine göre bütçe **Azlığı** ve kaynakların **Azlığı** (Çalışanlar burnout olur, donanım veya 3rdParty kütüphaneler veya SaaS ile daha basit ve daha kaliteli yapılacak işler yapılamaz

Kod Kalitesini Neler **Kötü** Etkiler (Süreçsel)

Proje'de doğru **Yazılım Geliştirme Modelinin** kullanılmaması (Spiral yerine Agile yapmaya çalışmak vb…)

Yazılım geliştirme modelinin **Fazlarının** neler içerdiğini ve nasıl işletilmesi gerektiğini bilmeyip, bu fazların içerisini doğru şekilde dolduramamak. (Planlama, Analiz, Tasarım ve Gerçekleştirim vb… Geliştirme Modeli içerisinde nasıl olgunlaştırılacağını Bilmemek)

Fazların **kalitesini** arttıran yazılım geliştirme **Prensiplerinden** bir haber olmak (Conway, OOP, TDD, BDD, Min-Value Product, Jobs To Be Done, Reactive Programing, 12 Factor Apps, SOLID, KISS, YAGNI vb..)

Kod Kalitesini Neler **Kötü** Etkiler (Mimarinin Sorunlu Olması)

Yazılım sisteminizin hangi **sistem bileşenlerinden** hangi **donanım altyapılarından** oluşacak, iyi şekilde bölümleyemezseniz.

Bu sistem bileşenlerinin, kendi **iç bileşenleri ve dış sistemler ile etkileşimlerinin** nasıl olacağını belirleyemezseniz. (Mimari ve Entegrasyon **Örüntüler**)

Düzgün bir **CI/CD Pipeline** sahip olmamanız durumunda. **Github** (branch yapıları) araçlarına hakimiyet, **Code Review**, **otomatik build alma**, bu süreçte **testlerin otomatik** çalıştırılması, **Test Covarage** çıkarılması, Sonar gibi araçların kod kalitesindeki eksiklikleri çıkarması eksikse.

Hangi deneyimdeki kişiler yazılım sistemini kullanacak ve destek sağlayacak belirleyemez (eğitim, Ulda User Guideing, Oyunlardaki Öğretici Senaryolar, Help)

Kod Kalitesini Neler Kötü Etkiler (Geliştirici Ekip)

Test Test Test ... yazılmaması, yazıyormuş gibi gözükmek (Nasıl etkiler?)

Pair Programming yapmamak (Nasıl etkiler)

Code Review yapmamak (Nasıl ?)

Sonar vb Code Kontrol Araçlarını kullanmamak (Nasıl ?)

OO Geliştirme Prensiplerini BilMemek (Abstraction, Encapsulation, High Cohesion, Low Coupling, Dependency Injection, SOLID, KISS, YAGNI, DRY)

★**Tasarım Örüntülerini BilMemek** (Singleton, Command, vb…)

Teknolojik Gelişmelere Ayak Uydurmak veya Baskılara Dayanmak

Trendler/Hype veya yeni teknolojiler her zaman sizin sorununuzu çözen **Silver Bullet** değildir. (Hype: Machine Learning, BigData …)

Yönetimden, Müşterilerden, Çevrenizdeki yeni teknolojileri **kullan baskısı** olacaktır

Teknolojileri herşeyin **çözümü** olarak gören yöneticilerin gazı ile ilerlemek

Teknolojik gelişmelerin **artısı eksisi**, projeye etkisini iyi şekilde değerlendirip ona göre kullanmayı bilmek Bulut Teknolojileri (Lambda), Frontend Teknolojileri (React), İletişim Teknolojileri (GraphQL, WebHook),

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod →

Kodu İyileştirme →

Kod Geliştirme Yöntemleri →

Kodu Dayanıklı Geliştirmek →

Design Patterns →

Anti Patterns →

~~Kod Kalitesini Neler Kötü Etkiler~~ Teknik Borç

Clean Code Code Smells Refactoring

AOP TDD SOLID KISS DRY YAGNI

12 Factor App Reactive Programming Chaos Eng

Design Patterns Creational Patterns

Structural Patterns Behavioral Patterns

Anti Patterns Spaghetti Code

Lava Flow Programming Golden Hammer

Reinvent The Wheel

Teknik Borç Nedir ?

Yazılım Geliştirme sırasında **bilerek veya bilmeyerek** yazılımın istenilen kalitede geliştirilmemesi ve kalite eksikliklerinin gerçek hayattaki finansal borçlar gibi birikerek ekibin hareket kabiliyetini, hızını düşürmesidir.

Yazılımın istenilen kalitede geliştirilmemesi NE DEMEK ?

Yazılımın kalitesini incelerken Internal(İç) ve External(Dış) kaliteler olarak 2ye ayırabiliriz.

Internal Quality (İç Kalite)

İç kalite dediğimiz kısım kodun kaliteli yazılıp yazılmadığı kısmıdır.

Kodunuz kaliteli bir şekilde yazılmış ve yapılandırılmış mı ?
Kod anlaşılabilir ve bakımı kolay yapılabilir mi ?
Kod ne seviyede test yazılmış, ve % kaç oranında kod covarage bulunuyor.

External Quality (Dış Kalite)

Yazılım Müşterinin ihtiyaçlarını karşılıyor mu ?
Yazılım çöküyor mu ? Yazılım etkileşimi ve tepkime süreleri nasıl
İyi bir UI tasarımı ve kullanabilirlik sunabiliyor mu ?

Yazılımın kaliteli geliştirilmemesinin NEDENLERİ ?

Bunun bir çok nedeni olabilir,

- zamanın azlığı,
- takımın deneyim eksikliği,
- yönetimsel hatalar,
- yanlış 3rd party ürün ve kütüphane seçimleri,
- müşteriyi iyi analiz edememe,
- yanlış bir geliştirme modelini kullanma,
- yanlış önceliklerde kod geliştirme,
- design system kullanmamak,

vb.. bir çok konu buna neden olmuş olabilir. Bu konuda Kod kalitesi için **Kod Kalitesini Neler Kötü Etkiler** ve Yazılım Mimarisi açısından **Yazılım Mimarının 5 Yanlışından** bahsetmiştik

Yazılım Entropisi

Bilim insanları düzensizliği entropi adı verilen nicelik ile ölçerler. Sistemlerdeki **düzensizlik arttıkça, entropi de artar.** Bu durum da faydalı (iş yapabilir) enerji miktarı azalır, faydasız enerji artar [Vikipedia]

As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it" by Meir Manny Lehman

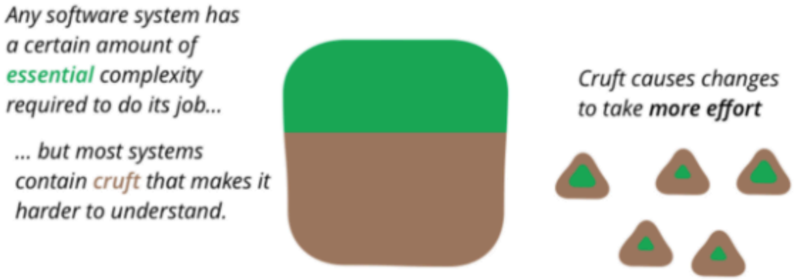
Gelişmekte olan yazılım içinde benzeri bir durum söz konusudur. Geliştikçe , kompleks seviyesi artacak, bu artış **yapının bozulmasına** neden olacak bir çok iş gücü önceki bu kodları bakım yapmaya veya bu tarz sorunları azaltmaya gidecektir.

Yazılım Entropisi

Burada kastedilen kod geliştirme ilerledikçe kodun geliştirmenin daha zorlaştığı, yeni yeteneklerin sisteme rahat bir şekilde eklenemediği ve ekibin , yazılım geliştirmenin giderek yavaşladığını farketmekle başlayacaktır. Bu aşamada en iyi çözümlerden biriside kod için Teknik Borçlardan Refactoring yöntemi ile kurtulmaktır. Refactoring düzgün ve sorunsuz olması **Test ve Test Code Coverage** istenilen düzeylerde olmasıyla münkündür.

Yazılımınız değişmese bile **ortam ve teknoloji geliştiği** için yazılımınız çevresiyle olan etkileşimi değişecek veya kullanıcıların yazılımdan beklentileri farklılaşacak yazılımınız yine bozulmuş olacaktır.

Teknik Borç Ödenmez ise Ne Olur ?



The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

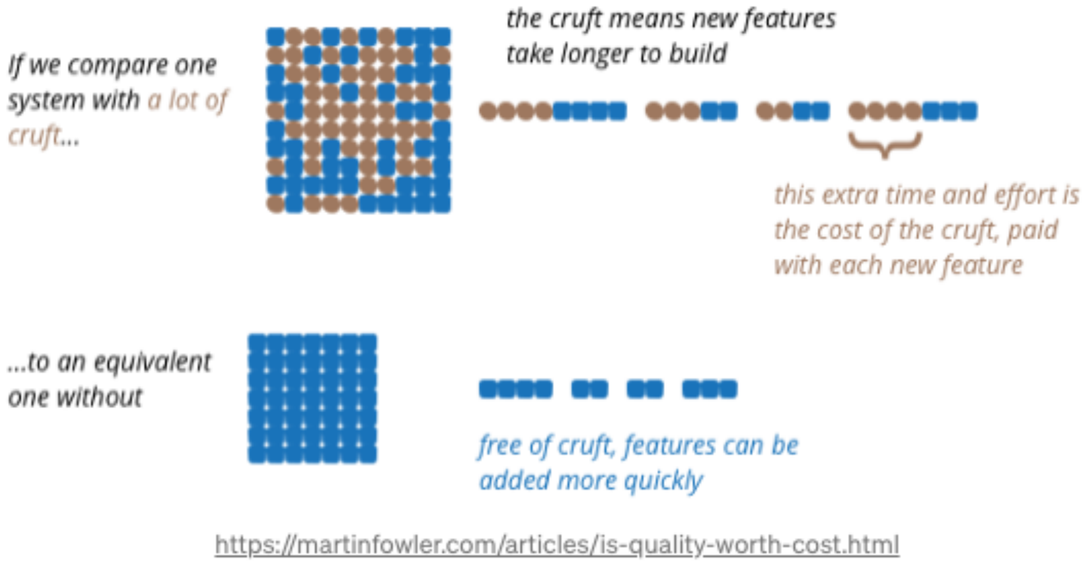
https://martinfowler.com/bliki/TechnicalDebt.html

Teknik borç zamanında ödenmez ise ekibin üretim hızı giderek azalarak işin içerisinden çıkılamayan bir yazılıma sahip olursunuz. Martin Fowler örneklerinde bu konuyu çok güzel özetlemiş

Yazılım = (Group of Function)
Herhangi bir yazılım sistemi esas işini gerçekleştirmek için bir miktar kompleski yapısında içerir, fakat çoğu sistem ekstradan bir tortu içerir, bu sistemin daha zor anlaşılması ve değiştirilmesine neden olur.

Örneğin aşağıdaki resimde yeşil kısım bizim yazılımdan beklediğimiz kısım, fakat kahverengi , yazılımda istemediğimiz kısımlar, bunun yanında ise bu yazılımı kırılım halinde gösterilmiş üçgenlerin çevresinde bu tortudan farklı kalınlıklarda olduğunu ve bunların değişimi engellediği ve güçleştirdiğidir.

Teknik Borç Maliyet

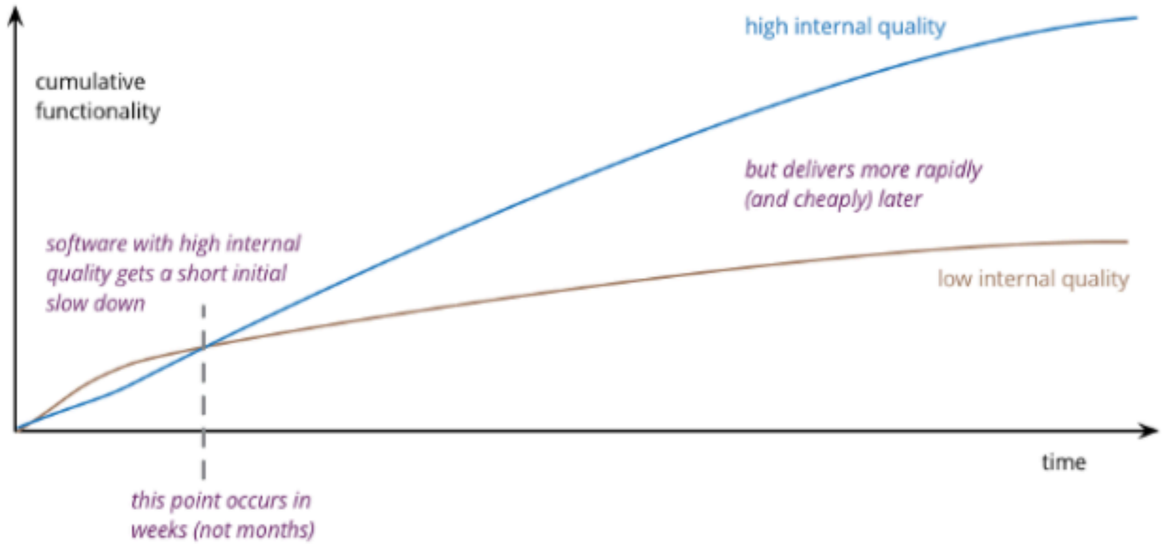


https://martinfowler.com/articles/is-quality-worth-cost.html

Yazılım geliştirmek için bu tortulu yapılara uygun kod geliştirmek ekstradan efor anlamına gelir. Aşağıda resimdeki **her kahverengi daire** yazılım geliştirmek için teknik borçtan doğan ekstra maliyettir.

Yazılım geliştirme hızınızı ne kadar eksi yönde etkilediğini görebilirsiniz.

Teknik Borç Maliyet



https://martinfowler.com/articles/is-quality-worth-cost.html

Aşağıdaki resimde ilk başta **low internal quality** ile daha hızlı kod geliştirip, daha fazla yetenek eklediğini düşünebilirsiniz ama zaman içerisinde tam tersine **high internal quality** göre yeni kabiliyet eklemenizin ne kadar zor olduğunun farkına varacaksınız.

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Teknik Borç~~

Kodu İyileştirme → Clean Code, Code Smells, Refactoring

Kod Geliştirme Yöntemleri → AOP, TDD, SOLID, KISS, DRY, YAGNI

Kodu Dayanıklı Geliştirmek → 12 Factor App, Reactive Programming, Chaos Eng

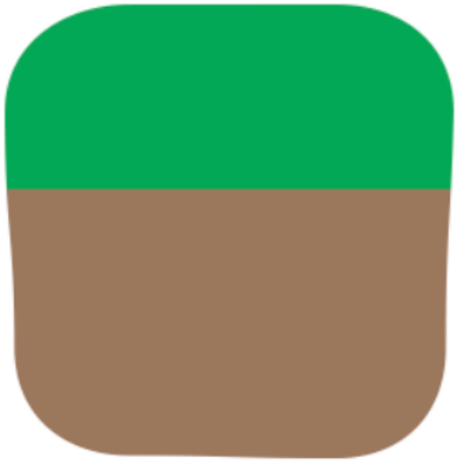
Design Patterns → Design Patterns, Creational Patterns, Structural Patterns, Behavioral Patterns

Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

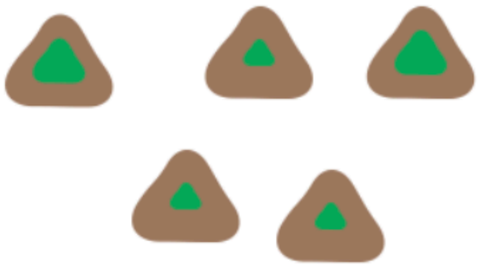
Clean Code

*Any software system has a certain amount of **essential** complexity required to do its job...*

*... but most systems contain **cruft** that makes it harder to understand.*



*Cruft causes changes to take **more effort***



The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

https://martinfowler.com/bliki/TechnicalDebt.html

Clean Code

Teknik Borç'tan (Tecnical Debt) bahsetmiştik. Gerçek yazılmak istenen kodun çevresindeki tortunun ekibi nasıl yavaş yavaş harektet edemez hale geldiğinden bahsetmiştik.

Peki bu tortuları en baştan oluşturmamak için kodumuzu nasıl geliştirmeliyiz ? → (Temiz Kod)

Veya kodumuzu geliştirdik ama istediğimiz kalite yazamadık bu durumda bu kodu nasıl Temiz Kod'a dönüştürebiliriz ? → (Refactoring) üzerinde duracağız.

Clean Code (Temiz Kod) yöntemi ile en baştan bu tortuların oluşmasını engelleyebilir veya azaltabiliriz.

Clean Code

Kodun temiz olması, kodu yazan geliştirici dışında ekiptekilerin kodu kolay şekilde anlayabilmesi ve geliştirme yapabilmesidir. Burada kolay kelimesini vurgulamak istiyorum. Kolay için olması gerekenler

- **readability**, (Basitçe okunup anlaşılıyor ise)
- **changeability**, (Basitçe değiştirilebiliyor ise)
- **extensibility** (Basitçe genişletilebiliyor ise)
- **maintainability**. (Basitçe bakım yapılabiliyor ise)

Genel Kurallar

- Standart yaklaşımları uygula
- Kodu basit tutmaya çalış, Olabildiğince kompleks yapılardan uzak dur
- Kodu bulduğunuzdan daha temiz halde bırakın.
- Her zaman sorunun kaynağına odaklanın. Problemin ana kaynağını bulmaya çalışın

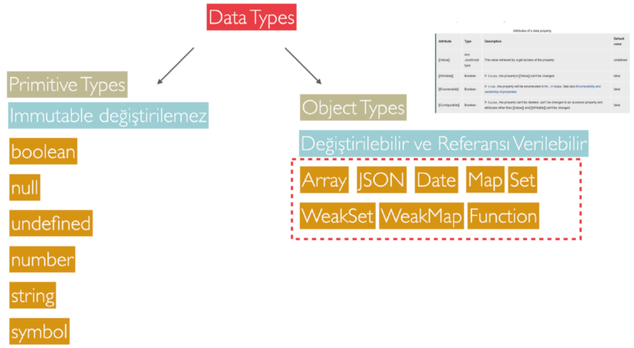
Tasarım Kuralları

- Konfigüre edilen veriyi kodun içerisinde derinlerinde değilde rahat erişilebilen değiştirilebilen kısımda bulundur.
- if/else veya switch/case koşulları yazmak yerine polymorphism tercih et.
- multi-threading kodları ayrıştır.
- Her kod yapısını konfigürasyonlu ve dinamik hale getirmekten kaçının.
- Dependency Injection (Bağımlılık Enjekte Etmeyi) kullanın.
- Bir sınıf doğrudan sadece bağımlılıklarını bilmelidir yasasını takip et.

Clean Code

Anlaşılabilirlik İpuçları

- Tutarlı ol. Bir işi bir yöntemle yapıyorsan, her yerde aynı yöntemi kullan.
- Açıklayıcı değişken isimleri kullan.
- Kod içerisindeki değişkenlerin tuttuğu (primitif ve object) türemiş verilerin veya kod akışının sınıf koşullarını kapsayacak şekilde encapsulate edin. Sınır koşullarını takip etmek zor olduğundan bu tarz işlemleri tek bir yerden gerçekleştirin.
- Object Türler yerine Primitif Türleri tercih edin. (Immutable)
- Mantıksal bağımlılıklardan kaçının. Aynı sınıf içerisinde başka bir takım şeylere bağımlılığı olan metodlar yazmayın.
- Negative koşullardan sakının.



Clean Code

İsimlendirme Kuralları

- Açıklayıcı ve kafa karışıklığına neden olmayacak isimlendirmeler kullanın
- İsimlendirmeler ile anlamlı ayrımlar oluşturun (Soyutlama için önemli)
- Telefazu kolay isimlendirmeler bulun
- Aradığınızda kolay şekilde bulunabilecek ve erişebilecek isimlendirmeler seçin
- Kodun içerisinde gizli rakamlar ve sabitlerden kaçının
- Encoding ve değişkenlerin önüne bir takım başlangıç tag takmayın.
- Not: Eskiden Code Editorleri zayıfken C gibi dillerde i, s …. Javada Interface I gibi ön takılar olurdu. Şimdi editörlerin görsel gösterimleri ve arama kabiliyetleri sayesinde bu tarz isimlendirmelere gerek yok

A.5 Fonksiyon Kuralları

- Küçük olmalı.
- Bir tek iş yapmalı
- İsmi açıklayıcı olmalı
- Olabildiğince az argüman almalı
- Side Effect içermemeli
- Flag (true/false) argümanlarını parametre olarak koda geçirilip bu method içerieisinde farklı metod çağrımları yapılmamalı.

Yorum Satırı Kuralları

- Olabildiğince kendinizi kod içerisinde anlatmaya çalışın,
- Gereksiz yorum ekleme
- Başlangıç bitiş kodu ayırma amaçlı yorum satırları eklemeyin.
- Kodu yorum satırı haline getirip bekletmeyin. Gereksiz kodları silin.
- Yorumu yazma nedeninizi açıklayın.
- Kodun açıklaması olarak kullanın.
- Sonuçların uyarısı olarak kullanın.

Source Kod Yapısı

- Kavramları dikey olarak ayır.
- Birbiri ile ilişkili kodlar dikey yoğunlukta görüntülenmeli
- Değişkenleri kullanım alanlarına yakın tanımla
- Birbirine yakın fonksiyonları birbirine yakın tanımla
- Benzer işleri yapan fonksiyonlar birbirine yakın olmalı
- Fonksiyonları aşağı yönlü akacak şekilde yerleştirin.
- Kod satırlarını kısa tutun.
- Yatay hizalama yapmayın. (üstteki, alttaki satır ile hizalama)
- Boşluk kullanarak ilişkili şeyleri birbirine yakın , ilişkisizleri uzaklaştır.
- Kodun oluşturduğu girintileri bozmayın

Objeler ve Veri Yapıları

- Kodun iç yapısını gizleyin.
- Veri yapılarını, dillerin hazır Collection yapılarını tercih edin
- Hibrid yapılardan kaçının.
- Olabildiğince küçük olun.
- Bir sey yapın. (Bir iş yapın)
- Küçük sayıdaki değişkenlerle çalışın.
- Temel sınıf, kendisinden türeyenler hakkında hiçbir şey bilmemelidir.
- Fonksiyona bir takım parametreler geçirerek bunların istenen davranışa göre şekillenmesindense, birçok sade fonksiyona sahip olmak daha iyidir.
- Static metodları tercih etme.

Clean Code

Tests

- Her test için bir assert
- Test okunabilir olmalı
- Test hızlı çalışabilir olmalı
- Test bağımsız olmalı
- Test tekrar edebilir olmalı

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Teknik Borç~~

Kodu İyileştirme → ~~Clean Code~~ Code Smells Refactoring

Kod Geliştirme Yöntemleri → AOP TDD SOLID KISS DRY YAGNI

Kodu Dayanıklı Geliştirmek → 12 Factor App Reactive Programming Chaos Eng

Design Patterns → Design Patterns Creational Patterns Structural Patterns Behavioral Patterns

Anti Patterns → Anti Patterns Spaghetti Code Lava Flow Programming Golden Hammer Reinvent The Wheel

Code Smells

Temiz kodun tanımını ve özelliklerini verdik. Bazen bir takım nedenlerden dolayı.

- zamanın azlığı,
- takımın deneyim eksikliği,
- yönetimsel hatalar,
- yanlış 3rd party ürün ve kütüphane seçimleri,
- müşteriyi iyi analiz edememe,
- yanlış bir geliştirme modelini kullanma,
- yanlış önceliklerde kod geliştirme,
- design system kullanmamak,
- vb.. bir çok konu buna neden olmuş olabilir.

Kodumuzu istediğimiz Clean Code (Temiz Kod) kalitesinde yazamayız. Temiz kod yerine kirli kod ortaya çıkar ve **Code Smells (Kod kokmaya)** başlar.

Code Smells

Temiz kodun tanımını ve özelliklerini verdik. Bazen bir takım nedenlerden dolayı.

- zamanın azlığı,
- takımın deneyim eksikliği,
- yönetimsel hatalar,
- yanlış 3rd party ürün ve kütüphane seçimleri,
- müşteriyi iyi analiz edememe,
- yanlış bir geliştirme modelini kullanma,
- yanlış önceliklerde kod geliştirme,
- design system kullanmamak,
- vb.. bir çok konu buna neden olmuş olabilir.

Kodumuzu istediğimiz Clean Code (Temiz Kod) kalitesinde yazamayız. Temiz kod yerine kirli kod ortaya çıkar ve **Code Smells (Kod kokmaya)** başlar.

Code Smells

Kod kötü kötü kokmaya başlar 😀 Tabi bu bir metafor. Aslında Kod ile ilgili aşağıda listelediğmize benzer sinyaller alıyorsanız

- **Rigidity (Sertlik):** Yazılımı değiştirmeniz çok zor hale gelmiş, bir değişiklik bir çok yeri etkiliyor ise.
- **Fragility (Kırılganlık):** Yazılımda ufak bir değişiklik uygulamanın bir çok yerini bozuyor ise.
- **Immobility (Hareketsizlik):** Kodunuzun bir takım tekrar eden kısımlarını diğer projelerde kullanamıyorsanız.
- **Needless Complexity:** Gereksiz kompleks ise
- **Needless Repetition:** Gereksiz tekrar var ise
- **Opacity (Saydamlık):** Okunabilirliği zor ise.

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Teknik Borç~~

Kodu İyileştirme → ~~Clean Code~~ ~~Code Smells~~ Refactoring

Kod Geliştirme Yöntemleri → AOP TDD SOLID KISS DRY YAGNI

Kodu Dayanıklı Geliştirmek → 12 Factor App Reactive Programming Chaos Eng

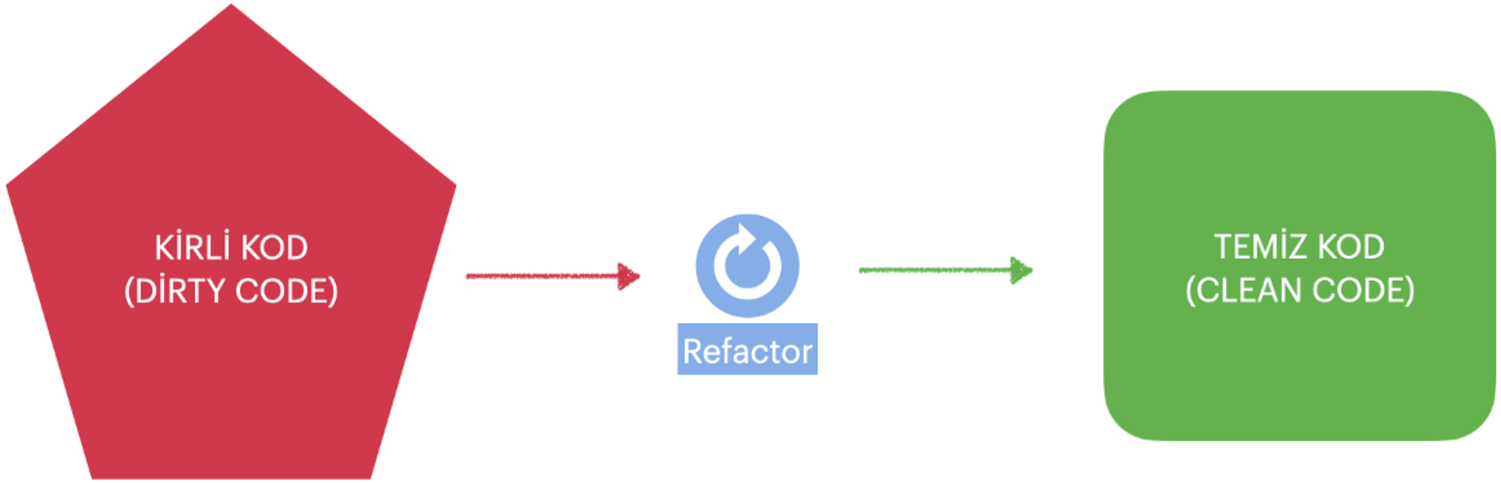
Design Patterns → Design Patterns Creational Patterns Structural Patterns Behavioral Patterns

Anti Patterns → Anti Patterns Spaghetti Code Lava Flow Programming Golden Hammer Reinvent The Wheel

Refactoring

Özetle kodunuza her yeni özelliği eklerken bir takım kalite kayıplarına neden olacaksınız ve teknik borçlar zaman içerisinde artacak, kodunuz yavaş yavaş kokmaya başlayacak.

Bu durumda mevcut kabiliyetleri hiç değiştirmeden (yenisini eklemeden, eskisini bozmadan) adım adım kodun kaliteli hale getirilme
işlemine **REFACTORING** denir.



Refactoring

SORU 1: Uygulama yeteneklerini hiç değiştirmeden **koruyarak** kodun üzerinde **Nasıl Refactoring** Yapacaksınız ?

Cevap,

Test. Kodunuz code coverage ne kadar iyiyse, kodunuz davranışları otomatik testler ile ne kadar çevrelenmiş ise, kodunuzda refactor yaptığınızda uygulamanızın değişmediğinden emin olursunuz. (**TDD ve CI** kavramlarınının çok iyi oturmuş olması gerekiyor.)

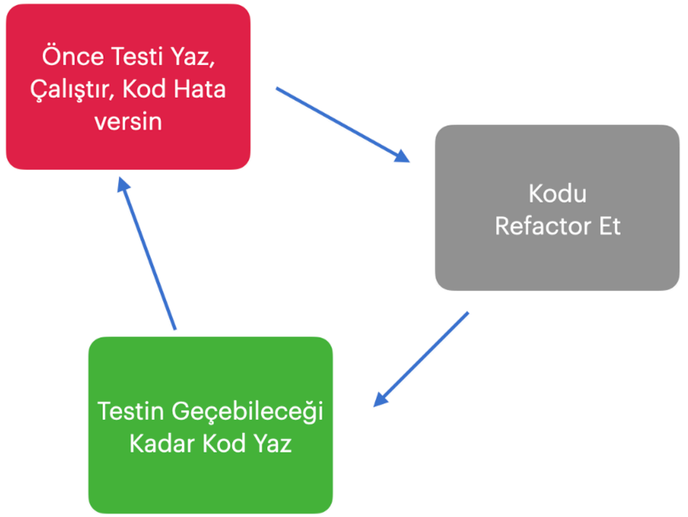
Küçük, küçük adımlarla ilerleyin. Kod üzerindeki değişikliği planlayın, ufak parçalara bölün, her seferinde bu ufak adımlardan bir tanesini yapıp, testlerinizi çalıştırın, refactoring başarılı olduğundan emin olduktan sonra diğer ufak adıma geçin.

Refactoring

SORU 2: Refactoring ne zaman yapmalısınız ?

Cevap, Fırsat bulduğunuz her zaman. Bir takım problemleri kod değişikliklerini ufakken çözmek çok daha basittir. Koddaki problem çok yayılmadığı için hem etkisi, hem değişiklik yapılan kod satır sayısı az olacaktır. Bu yüzden her fırsat bulduğunuzda kod kalitenizin kontrol ederek gereken yerlerde refactoring yapın.

Refactoring



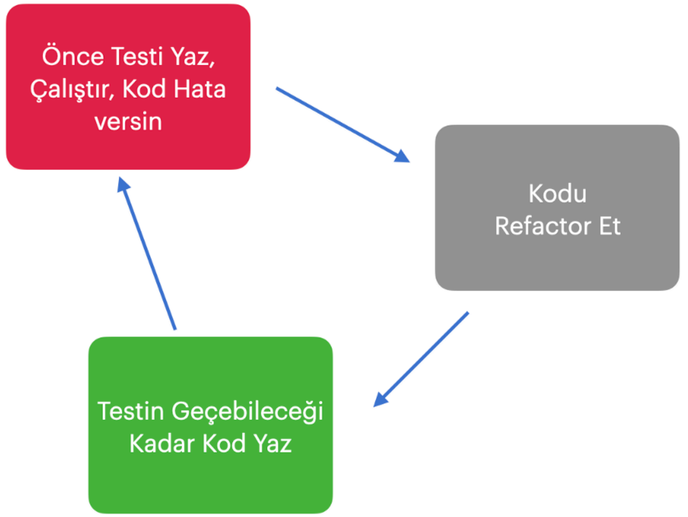
SORU 3: Refactoring nasıl yapacağız. Bir takım teknikleri bulunuyor mu ?

1. TDD (Test Driven Development):

Agile (Çevik) yazılım geliştirme metodunda sıkça kullanılan bir yöntemdir.

- *Test-First* yönteminde önce kod geliştirilmei yapmadan önce test yazılır.
- Testin çalıştırılması ve hata vermesi üzerine kod refactor edilerek testing geçirilmesi sağlanır.
- Böylece her bir geliştirme aşamasında öncelik test'in yazılması, testin hata almasınınn sağlanması ve sonrada kodun refactor edilmesidir.

Refactoring



Not: Testin güzel bir yanıda kodun test yazılmak için istenilen soyutlama seviyesini zorunlu sağlatmasıdır. Geliştiriciyi dışarıdan bağımlılık enjekte edilecek şekilde kodu geliştirtir. Bunun nedenide gerçek uygulama verileri yerine test verileri ,gerçek servisler yerine mock servisleri koda geçip test yapabilme zorunluluğudur.

Refactoring

2. Basitleştirme ve Sadeleştirme Yöntemi

Kodumuzu oluşturan parçalara baktığımızda ***paketler, modüller, sınıflar ve içerisinde yer alan metodların*** olduğunu görürüsünüz.

Zaman içerisinde yeni yetenekler eklendikçe bu parçaların büyüdüğünü ve şiştiğini görürsünüz. Kod fazlalıkları, tekrarları, kodun kontrolünü giderek zorlaştırır.

Bu durumda geliştiriciler kodu parçalara yeni sınıflar veya metodlar oluşturarak delegasyon yöntemi ile iş mantıklarını azaltır veya tekrar eden kodları ortak soyutlamaların içerisine taşırlar.

Bu kısımda soyutlama aşamasında ne kadar çok ***Kalıtım yerine*** Composition(Kapsamayı) tercih ederseniz kod gelişiminiz o kadar sağlıklı olur. Bu konuyu ilerleyen zamanda Fonksiyonel Programalama yazılarımda Object Compositon içerisinde anlatıyor olacağım.

Ama temel adım olabildiğince ortak kod mantıklarını bir alanda toplayıp, kod tekrarını önlemek olmalıdır.

Refactoring Kullandığım Yöntemlerden Bazıları

Extract Method

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOuts
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstan
}
```

https://refactoring.guru/extract-method

Inline Method

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ?
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2
  }
}
```

https://refactoring.guru/inline-method

Extract Variable

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC"
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

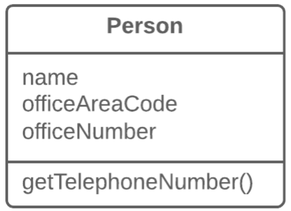
https://refactoring.guru/extract-variable

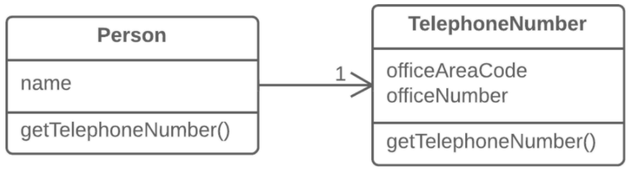
Inline Temp Yapmak Herzaman Mantıklı Değil ?

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
```

```
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

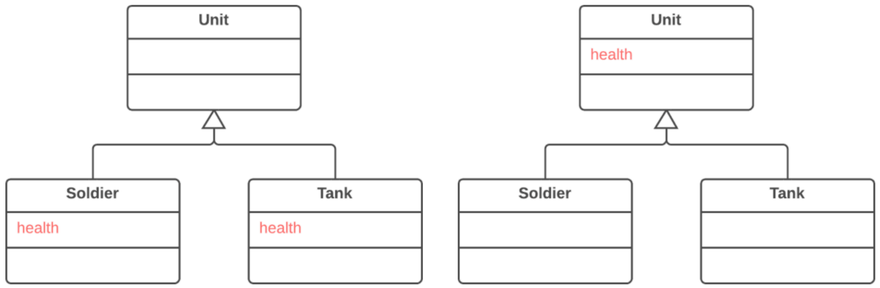
Extract Class



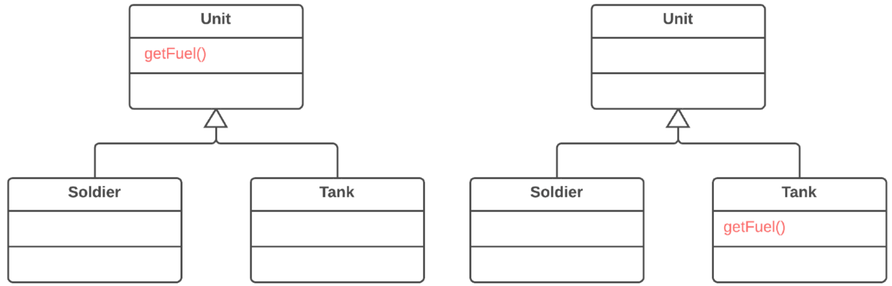


https://refactoring.guru/extract-class

Pull Up



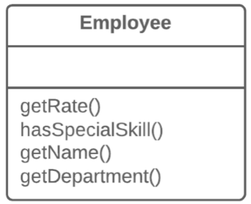
Pull Down

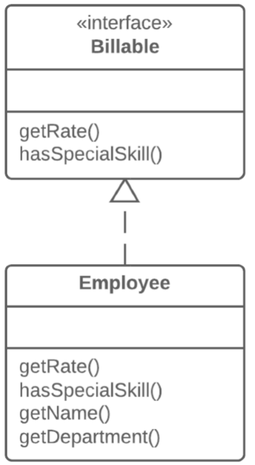


Renaming

Method Parameter Optimization

Extract Interface





https://refactoring.guru/extract-interface

Kodu Geliştirme Yöntemleri

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~

Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~

Kod Geliştirme Yöntemleri → AOP TDD SOLID KISS DRY YAGNI

Kodu Dayanıklı Geliştirmek → 12 Factor App, Reactive Programming, Chaos Eng

Design Patterns → Design Patterns, Creational Patterns, Structural Patterns, Behavioral Patterns

Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

AOP

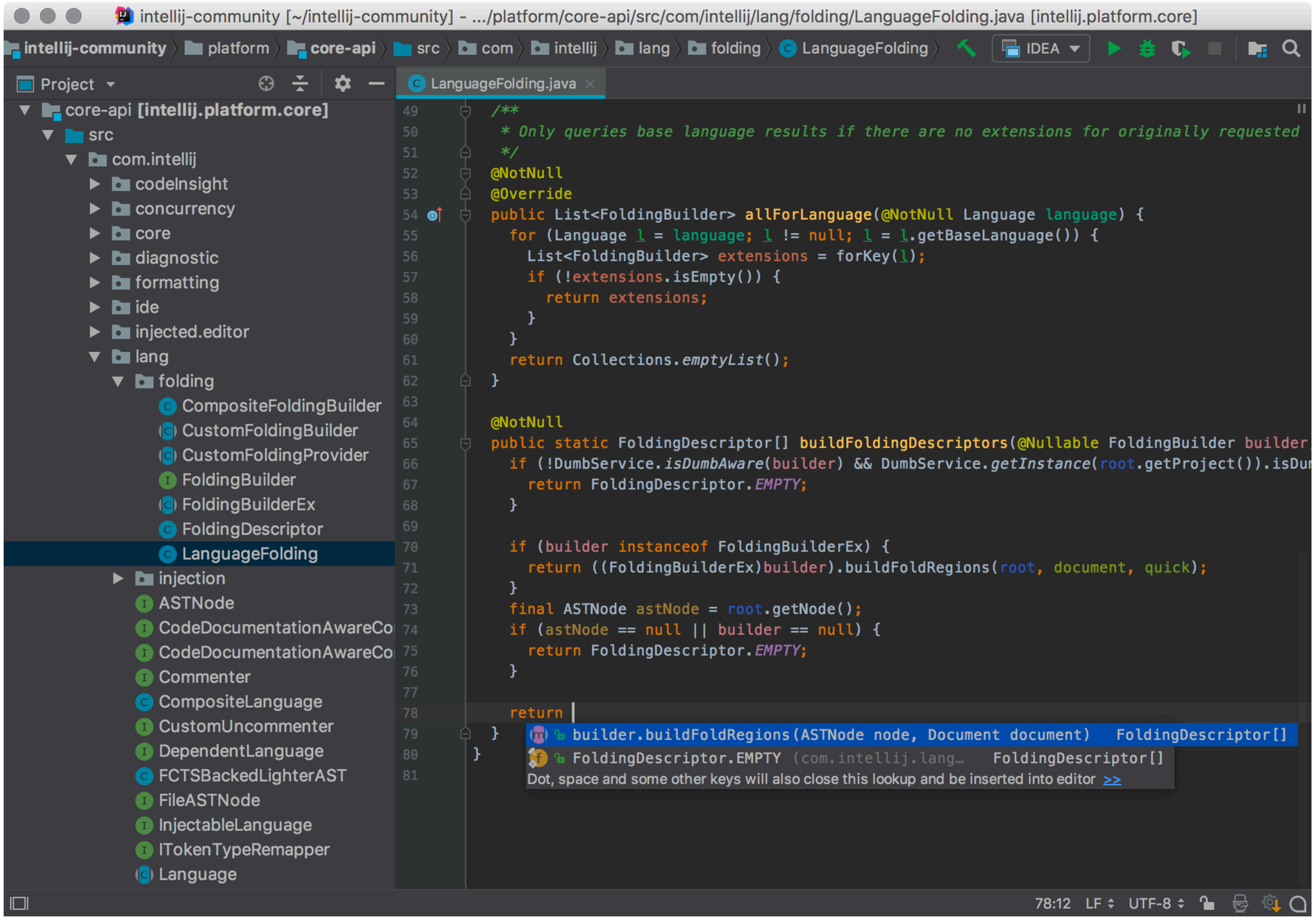
Geliştirdiğimiz sistemlerde tüm sistemler tarafından ortak olarak kullanılması gereken fonksiyonaliteler vardır. Bu tip yapılar Nesne Yönelik Programalayı zorlayan konulardır çünkü **High Cohesion — Low Coupling** kavramını bozan **Cross-Cut** fonksiyonaliteleri içerir,

Örneğin

- Sistemin Durumuna Göre Tüm Bileşenlerin farklı şekilde davranması
- Dark Mode / Light Mode
- Kullanıcı rolü veya yetkisinin olup olmaması
- Loglama Mekanizmaları
- Hata yakalama mekanizması (Exception Handling Mekanizması)
- Cache Mekanizmaları

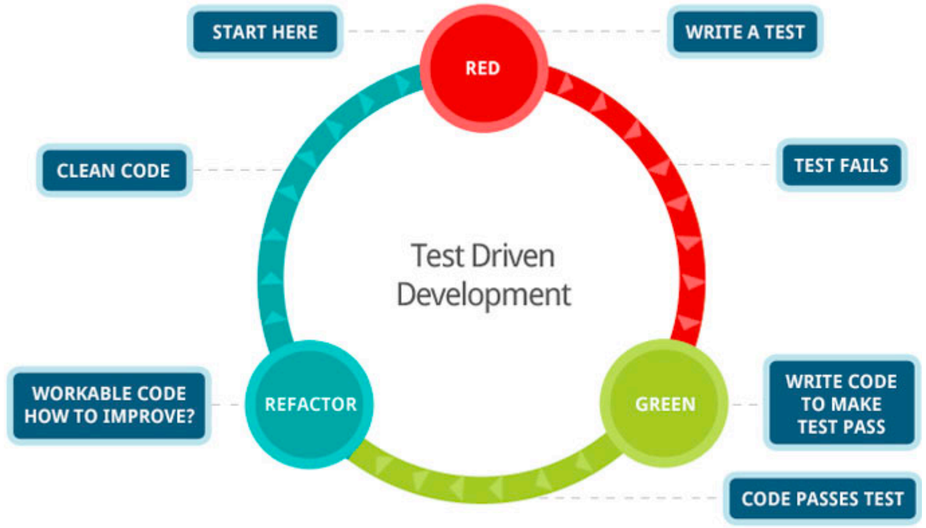
Bunları geliştirebilmeleri için kodun içerisine bunlar ile ilgili kodları yazmak yerine fonksiyonların üstüne konulan **Annotation/Aspect** tagları ile bu kodların Nesne tabanlı programlamanın içerisine dağılması engellenmiştir. Encapsulation sağlanmıştır.

AOP


intellij-community [~/intellij-community] - .../platform/core-api/src/com/intellij/lang/folding/LanguageFolding.java [intellij.platform.core]
intellij-community
platform
core-api
src
com
intellij
lang
folding
LanguageFolding
IDEA
Project
core-api [intellij.platform.core]
src
com.intellij
codeInsight
concurrency
core
diagnostic
formatting
ide
injected.editor
lang
folding
CompositeFoldingBuilder
CustomFoldingBuilder
CustomFoldingProvider
FoldingBuilder
FoldingBuilderEx
FoldingDescriptor
LanguageFolding
injection
ASTNode
CodeDocumentationAwareCo
CodeDocumentationAwareCo
Commenter
CompositeLanguage
CustomUncommenter
DependentLanguage
FCTSBackedLighterAST
FileASTNode
InjectableLanguage
ITokenTypeRemapper
Language
LanguageFolding.java
/**
* Only queries base language results if there are no extensions for originally requested
*/
@NotNull
@Override
public List<FoldingBuilder> allForLanguage(@NotNull Language language) {
for (Language l = language; l != null; l = l.getBaseLanguage()) {
List<FoldingBuilder> extensions = forKey(l);
if (!extensions.isEmpty()) {
return extensions;
}
}
return Collections.emptyList();
}
@NotNull
public static FoldingDescriptor[] buildFoldingDescriptors(@Nullable FoldingBuilder builder
if (!DumbService.isDumbAware(builder) && DumbService.getInstance(root.getProject()).isDu
return FoldingDescriptor.EMPTY;
}
if (builder instanceof FoldingBuilderEx) {
return ((FoldingBuilderEx)builder).buildFoldRegions(root, document, quick);
}
final ASTNode astNode = root.getNode();
if (astNode == null || builder == null) {
return FoldingDescriptor.EMPTY;
}
return
}
builder.buildFoldRegions(ASTNode node, Document document) FoldingDescriptor[]
FoldingDescriptor.EMPTY (com.intellij.lang… FoldingDescriptor[]
Dot, space and some other keys will also close this lookup and be inserted into editor >>
78:12
LF
UTF-8

2nci Gün (Code Quality)

TDD (Test Driven Development)



Kod öncesinde Unit Test yazarak yaptığınız yazılım geliştirme yöntemi. Kodu geliştirmeden önce testi yazıyoruz. Test çalışmıyor hata veriyor.

Kodu geliştiriyoruz. Test geçiyor. Yeni ekliyeceğimiz kodlar ve geliştireceğimiz kısımlar için yine Testi güncelliyoruz.

Test yine hata veriyor. Sonrasında kodu refactor ederek testin tekrar geçmesini sağlatıyoruz.

Utility, Function, Library, Framework gibi unsurlarda bence TDD oldukça başarılı işler.

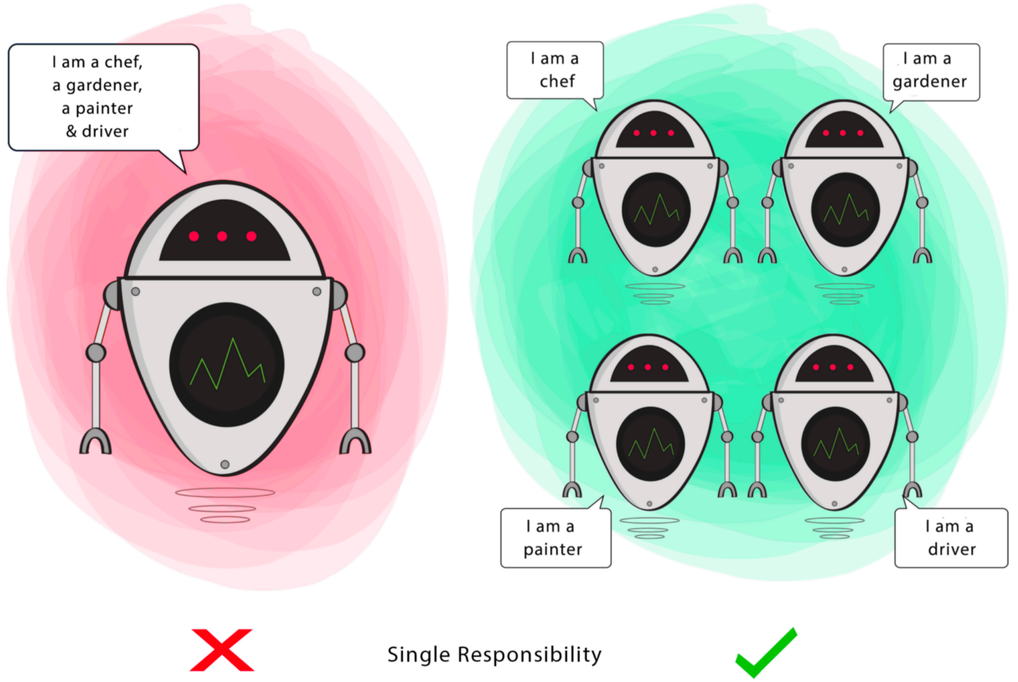
Input/Output sınırlı. Kodunuz daha çok logic/mantık odaklı. Ve siz burada TDD ile bu kodun doğru çalışıp çalışmadığını Fonksiyon bazında , Sınıf bazında, Modül bazında, Kütüphane bazında gerçekleştiyorsunuz.

2nci Gün (Code Quality)

S. Single Responsibility Principle

Bir sınıfın sadece bir amacı/sorumluluğu olmalı ve **tüm fonksiyonları bu amaç doğrultusunda** yazılmalıdır.

Birden fazla amacı tek bir sınıf içerisine koymaya başladığınızda sınıfınızın kodunun karmaşıklaşmaya başladığını , **her değişiklik ihtiyacını bu sınıf içerisine** koymaya başladığınızı görebilirsiniz.



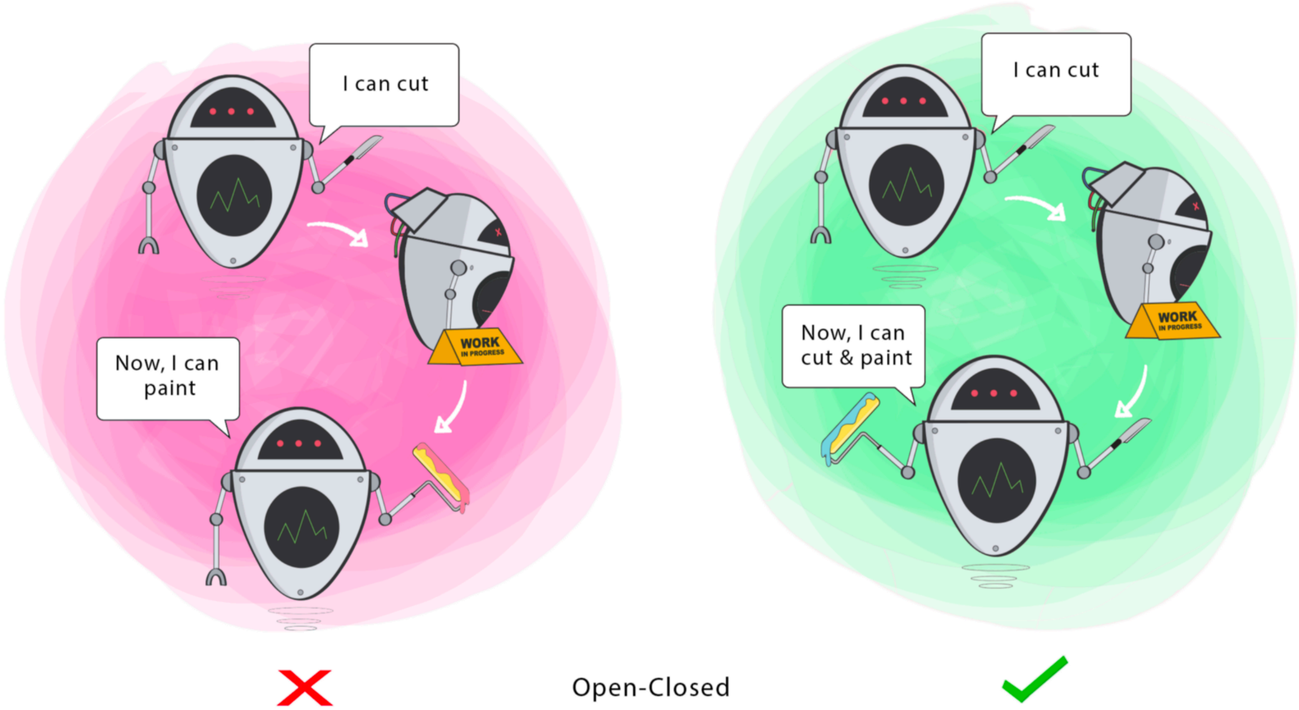
O. Open-Closed Principle

Sınıflarınızın **genişlemeye açık , değişime kapalı** olmasıdır**.**

Özetle amacı sisteme yeni bir fonksiyon eklemek istediğimiz zaman bunu mevcut sınıflara dokunmandan

AbstractClass/Interface extend ederek gerçekleştirebilmektir.

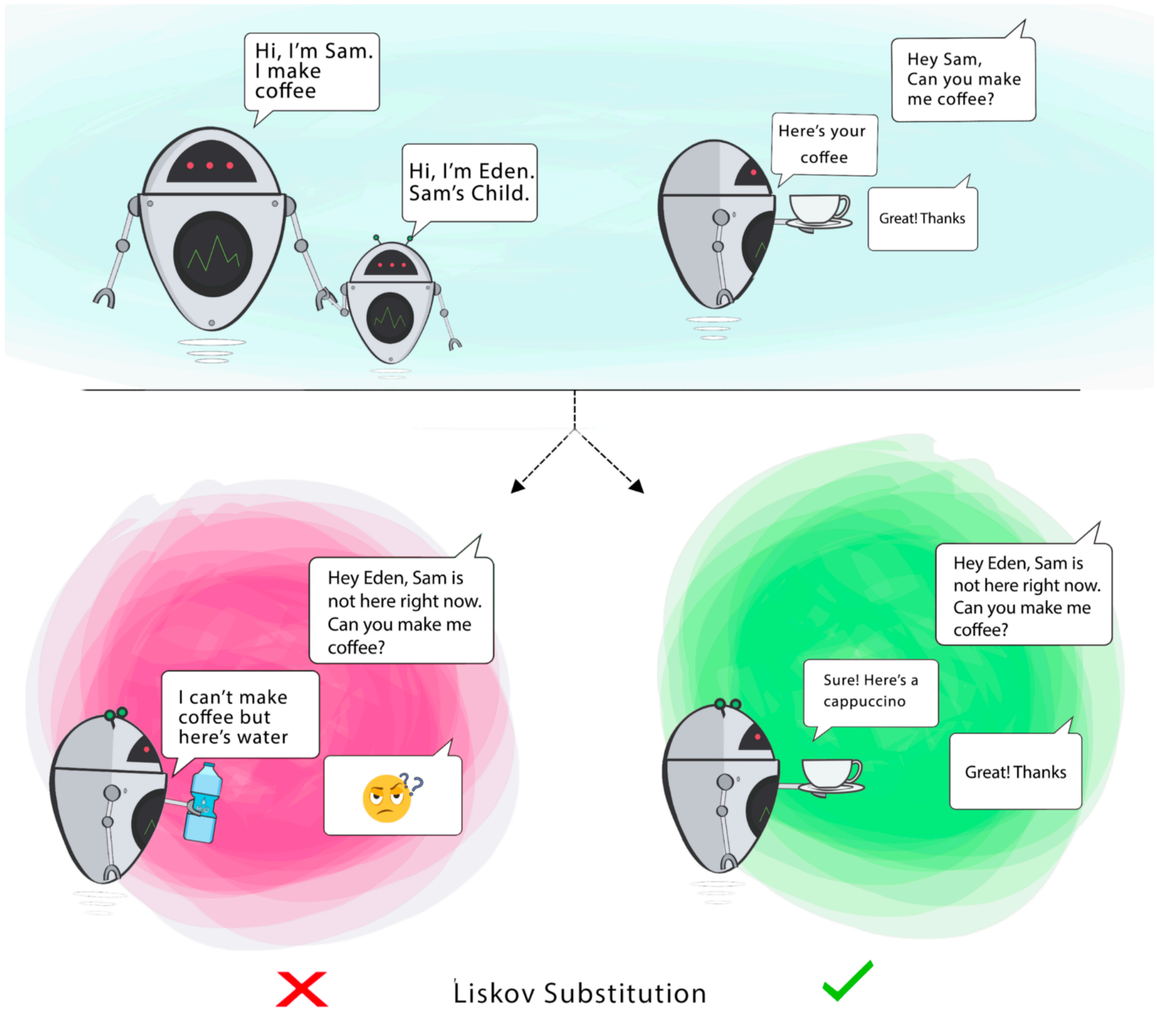
Bu sayede yeni özellik eklemek için mevcut sınıfların güncellenmesine gerek kalmaz.



L. Liskov Substitution Principle:

Tasarım örüntülerindeki Template Method olduğu gibi Child sınıfların türediği Parent sınıfın metodlarının ve tanımının alt sınıflar içinde bir anlamı olması/ gerçekleştirimi olmasıdır.

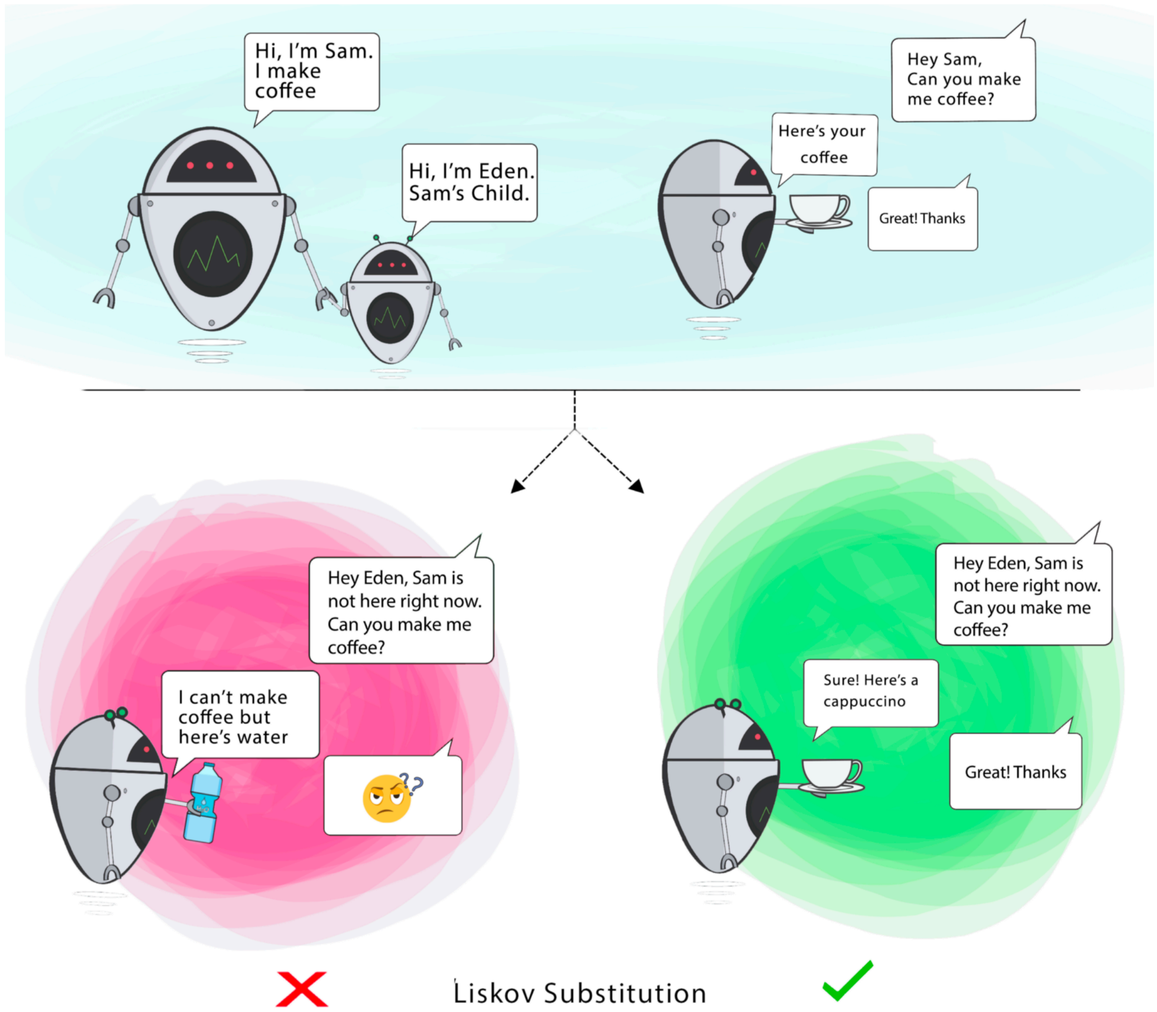
Bu sayede Parent için oluşturduğunuz bir metodu bu Parent'dan türeyen tüm Child sınıflarda kullanabilirsiniz.



L. Liskov Substitution Principle:

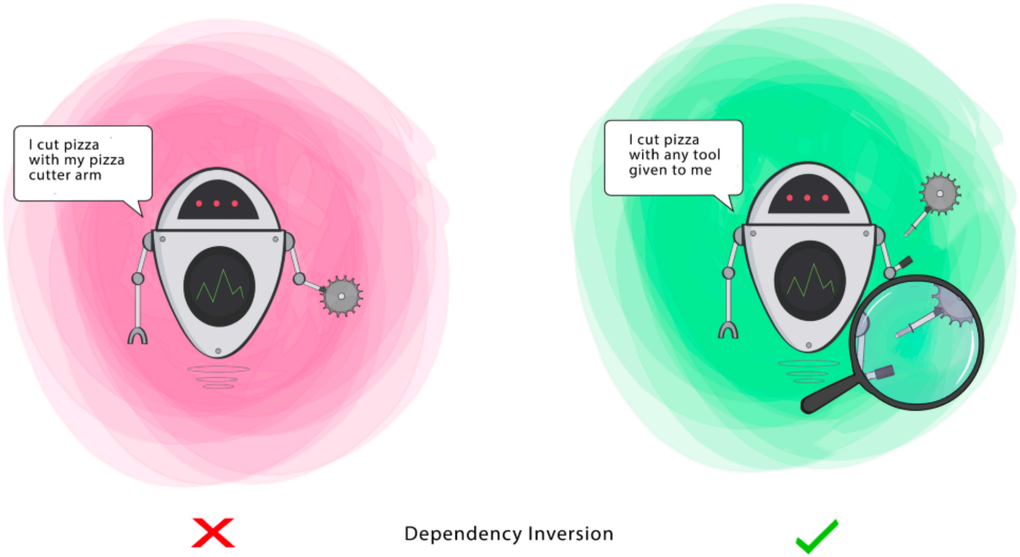
Tasarım örüntülerindeki Template Method olduğu gibi Child sınıfların türediği Parent sınıfın metodlarının ve tanımının alt sınıflar içinde bir anlamı olması/ gerçekleştirimi olmasıdır.

Bu sayede Parent için oluşturduğunuz bir metodu bu Parent'dan türeyen tüm Child sınıflarda kullanabilirsiniz.



D. Dependency Inversion principle:

Bir bileşen/sınıf başka bir bileşeni/sınıfı kullanarak diğer sınıfa işlemler yaptırıyor ise diğer sınıf ile kendisinin belirlediği bir arayüz üzerinden konuşması ve sınıfın kendisine dışardan enjekte edilmesi büyük fayda sağlar. **(Dependency Injection).** Bu sayede çağıran sınıfın mantığını çok çok sınırlandırarak sadece işine odaklanmış Bileşenler yapabilirsiniz.



Örneğin: Bir sınıfınız var ve işi kendi tuttuğu değişkenlerin state yazmak. **yaz()** Siz bu yaz fonksiyonunu gerçekleştiren dosyaya yazma sınıfını direk kullanırsanız. Yarın buyara **DB/Log/Dosya/Console/Newtork** vb.. farklı kanallara yazma imkanını taşıyan gerçekleştirimleri bu sınıfta kullanamazsınız. Halbuki bu işlemi dışardan başka bir mantık karar verirse , sizin yazdığınız sınıfın yapması gereken sadece kendine enjekte edilen bilmediği bir sınıfın yaz() fonksiyonunu çağırmaktan ibaret olacak. Gerisini düşünmeyecek :)

2nci Gün (Code Quality)

- Temiz ve Kaliteli Kod
  - ~~Kod Kalitesini Neler Kötü Etkiler~~
  - ~~Clean Code~~
- Kodu İyileştirme
  - ~~Teknik Borç~~
  - ~~Refactoring~~
  - ~~Architecture Clearance~~
- Kod Geliştirme Yöntemleri
  - ~~AOP~~
  - ~~TDD~~
  - ~~SOLID~~
  - KISS
  - DRY
  - YAGNI
- Kodu Dayanıklı Geliştirmek
  - 12 Factor App
  - Reactive Programming
  - Chaos Eng
- Design Patterns
  - Design Patterns
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Anti Patterns
  - Anti Patterns
  - Spaghetti Code
  - Lava Flow Programming
  - Golden Hammer
  - Reinvent The Wheel

KISS, YAGNI, DRY

KISS (Keep It Simple, Stupid): Kodu olabildiğince basit ve sade bir şekilde sadece işi yapacak şekilde geliştirmek. **YAGNI(You aren't gonna need it)** Extreme Programlamanın temellerinden olan bu yöntemde gerekmiyorsa fazladan bir fonksiyon ekleme. Bu 2 yöntemde de geleceğe yönelik over-engineering yaparak karmaşık sistemler tasarlamamak üzerine

DRY (Don't repeat yourself): Kodunuzda farklı modülün içerisinde benzer fonksiyonalitede yapılar bulunur, bunları her modülde tekrar yazmanız kod tekrarı(code dublication) neden olacaktır. Olabildiğince belli işlerin sorumluluğunu ilgili modüllere vererek abstraction iyi sağlamak gerekiyor.

Kodu Dayanıklı Geliştirmek (Bulut ve SaaS uygun)

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~

Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~

Kod Geliştirme Yöntemleri → ~~AOP~~ ~~TDD~~ ~~SOLID~~ ~~KISS~~ ~~DRY~~ ~~YAGNI~~

Kodu Dayanıklı Geliştirmek → 12 Factor App, Reactive Programming, Chaos Eng

Design Patterns → Design Patterns, Creational Patterns, Structural Patterns, Behavioral Patterns

Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

12 Factor Apps

THE FACTORS

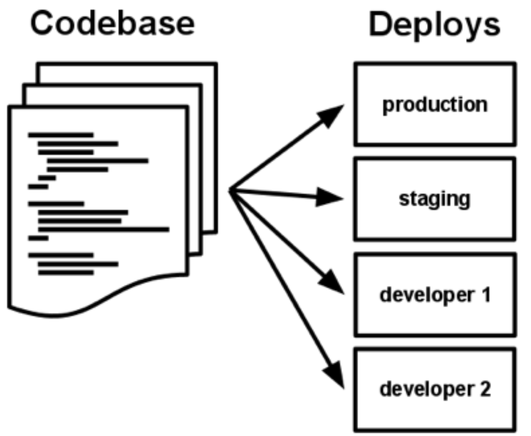
1. Codebase
2. Dependencies
3. Config
4. Backing Services
5. Build, Release Run
6. Processes
7. Port Binding
8. Concurrency
9. Disposability
10. Dev/Prod Parity
11. Logs
12. Admin Processes

I- Codebase

Codebase kaynak kod ve dosyaların yönetimini sağlayan araçtır. Örneğin: Git, Subversion, Mercurial vb..

Uygulamanızın dev, staging, production olsun **bir tane Codebase** içerisinde olması. Environment ile ilgili config bilgilerinin ilgili environment ortamında bulunması.

Staging ortamına commit edilmemiş dev geliştirmeleri veya production ortamına commit edilmemiş staging geliştirmeleri olabilir. Tüm bu versiyonlara ait kodun tek bir codebase üzerinden kontrol edilebilmesi gerekmektedir.



2- Dependencies

Projenizin bağımlılıklarını, bağımlı olduğu diğer kütüphaneleri projenizin içerisine statik olarak eklememek, açık bir şekilde bir dosyada belirterek ve **bir paket yöneticisinin** ilgili bağımlılıkları getirmesini sağlatarak gerçekleştirmelisiniz. Bunun için farklı dillere yönelik farklı paket yöneticileri bulunuyor.

- RubyGems / Bundler (Ruby)
- PIP / PyPI (Python)
- Composer (PHP)
- **NPM (Node.JS)**
- Bower (JS, CSS, HTML)
- CocoaPods (Objective-C)
- Maven (Java)
- Lein (Clojure)

Bu sayede Codebase içerisindeki bir dosyada bağımlılıklarınızın ismini, versiyonunu tutmanız yeterli.

3- Config

Projenize ait konfigürasyon bilgilerini **kodunuzun içerisinde statik olarak tutmayın**. Çünkü test, staging, prod veya başka bir durum için ortamı klonlandığında Bu konfigürasyon bilgilerinin güncellenmesi gerekecektir.

Konfigürasyon bilgileri;

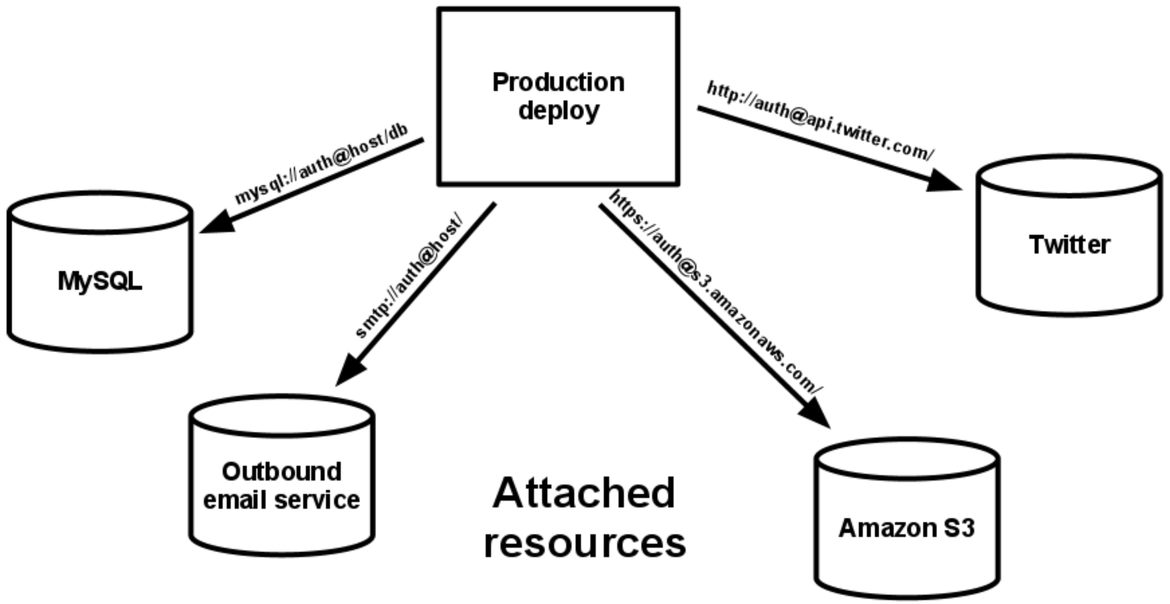
- Veritabanı, MemCache ve diğer Backend servislerine erişim bilgilerini içerebilir.
- Dışarıdaki başka servislere erişim için Credentials(AWS, Twitter, Facebook vb)
- Deploy edilen hostname ile ilgili konfigürasyon bilgilerini içerebilir.

Bu tip konfigürasyon bilgilerinin env variable olarak saklanması. Kodun ortamdan bağımsız olmasını sağlayacaktır.

4- Backing Services

Arka planda çağrılan servisleri mikro-servis olarak kullanması. Bir kaynak olarak kullanılması. İlgili kaynağa erişebilmek için gerekli DNS adresinin ve Credentials bilgisinin yeterli olması. En önemli terimlerden birisi **Loose Coupling olması**

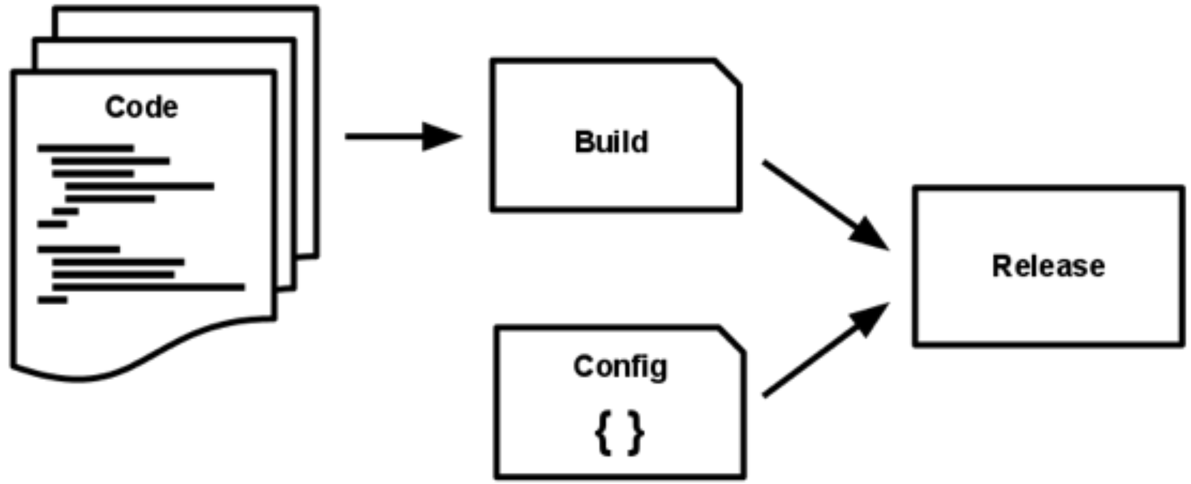
Bu sayede sistemin arkaplanında kullanılan servisleri kod değişikliği yapılmadan sadece konfigürasyon güncellemesi ile değiştirilebilir.



5- Build Relase, Run

Build, Relase ve Run işlemlerinin birbirinden ayrılması. Deployment araçları, otomasyon araçlarının bu işleri birbirinden ayırarak tüm süreçleri oluşturması beklenir.

- Code Güncellemesi
- Build
- Release (Config bilgilerinin eklenmesi)
- Run Time Deploy'u



6- Processes

Stateless (State tutmayan) process geliştirin. State veya veri tutma ihtiyacınız var veritabanı gibi stateful backend servislerinde bu bilgileri tutun.

Bu sayede 1..n Process uygulamanızı çalıştırabilirsiniz. Burada bellekte tutulan sticky session yerine bunları **Redis/Memcached** saklayarak servis olarak kullanmak Process sayınızı kolay bir şekilde arttırabilmenizi sağlar.

7- Port Binding

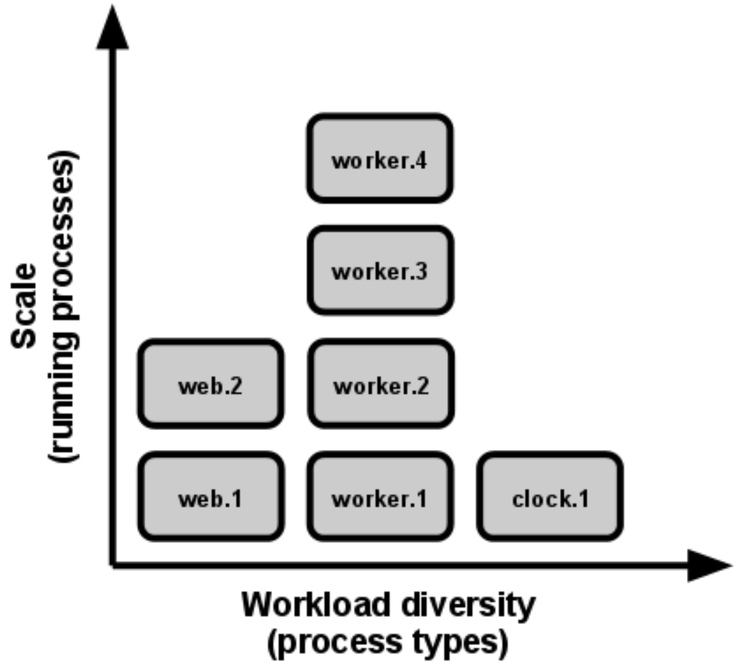
Servislerin dışarıya port binding sayesinde belli URL ve portlar üzerinden erişilebilir hale gelmesi. Bu sayede istediğiniz servise bu IP ve Portlar üzerinden erişebilirsiniz.

Müşteri Servisi https://192.162.12.16:5555
Ürün Katalog Servisi https://23.123.15.17:5000

8- Concurrency

Çalışan process ölçeklendirilme ihtiyaçlarına göre eş zamanlı şekilde tasarlanması ve duruma göre klonlanması ve restart edilebilmelidir.

Bu sayede Req/Resp yük çoğaldı ise Web modülü işleme kısmında bir gecikme var is Worker modülü ölçeklendirilebilecektir (Yatay ölçeklendirme).



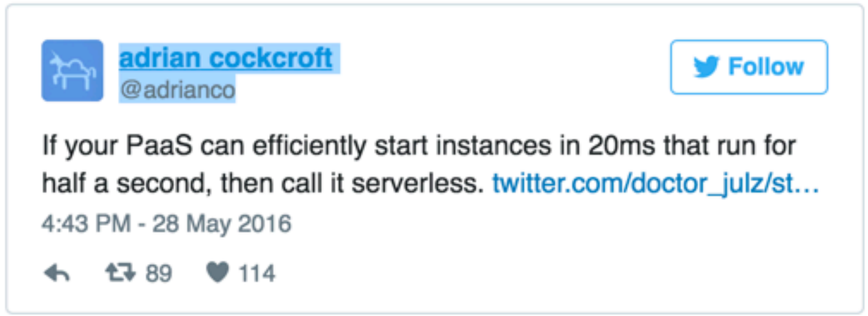
9- Disposibility

Servisler hızlı şekilde başlatılabilmeli ve kapatılabilmelidir. Servislerin tek kullanımlık oluşturulup yok edilebilmelidir. Bunun maliyeti oldukça düşük olmalı. Docker/ Container yapıları bunlara uygun öneklerdir.

Aynısı AWS Lambda ortamı içinde söyleyebiliriz.

Comparison with PaaS

Given that Serverless FaaS functions are very similar to 12-Factor applications, are they in fact just another form of 'Platform as a Service' (PaaS) like Heroku? For a brief answer I refer to Adrian Cockcroft



In other words most PaaS applications are not geared towards bringing entire applications up and down for every request, whereas FaaS platforms do exactly this.

10- Dev/Prod Paritiy

Summarizing the above into a table:

	Traditional app	Twelve-factor app
Time between deploys	Weeks	Hours
Code authors vs code deployers	Different people	Same people
Dev vs production environments	Divergent	As similar as possible

Geliştirme ve Canlı ortam arasında oluşacak boşluğu(gap) kapatmayı amaçlar. Bu boşluklar/farklılaşmalar neden ortaya çıkar ?

- **Zaman Farkı:** Geliştirme ortamına Push saatte bir gerçekleşirken, production ortamına Push işlemi haftalar aylar alabilir.
- **Personel Farkı:** Geliştirme(Dev) ve Deploy(Ops) işlemlerinin farklı kişiler tarafından yapılması, geliştirme işleminin bitmesine rağmen deploy süresini daha ileriye atabilir.
- **Tools(Araçlar) :** Geliştirme ve Canlı ortamda kullanılan araçların farklı olması.

Örneğin geliştirme ortamında Tomcat, Canlı ortamda Weblogic kullanılması vb..
Geleneksel Uygulama ile Twelve-factor App arasındaki olması istenen farkları aşağıda listelemiş.

11- Logs

Logları dosyaya statik yazılan bir log cümleleri olarak tutmak yerine. Dağıtık sistemde event-stream oluşturularak başka servislerin bunları yakalayıp, görüntülemesi, debugging, üzerinde arama yapabilmesine izin veren Hadoop, ElasticSearch sistemlerinin kurulu olmasını bekler.

Çünkü sistemin gerçek zamanlı/geçmişe yönelik görüntülemenin en iyi yöntemi log tur.

12- Admin Processes

Tek seferlik yapılan admin ve yönetim görevlerinin scriptler ile gerçekleştirilmesi. Örneğin.

- Veritabanı migration işlemlerinin çalıştırılması
- Tek seferlik scriptlerin çalıştırılması
- Console üzerinden bazı kodların çalıştırılması
- vb..

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~

Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~

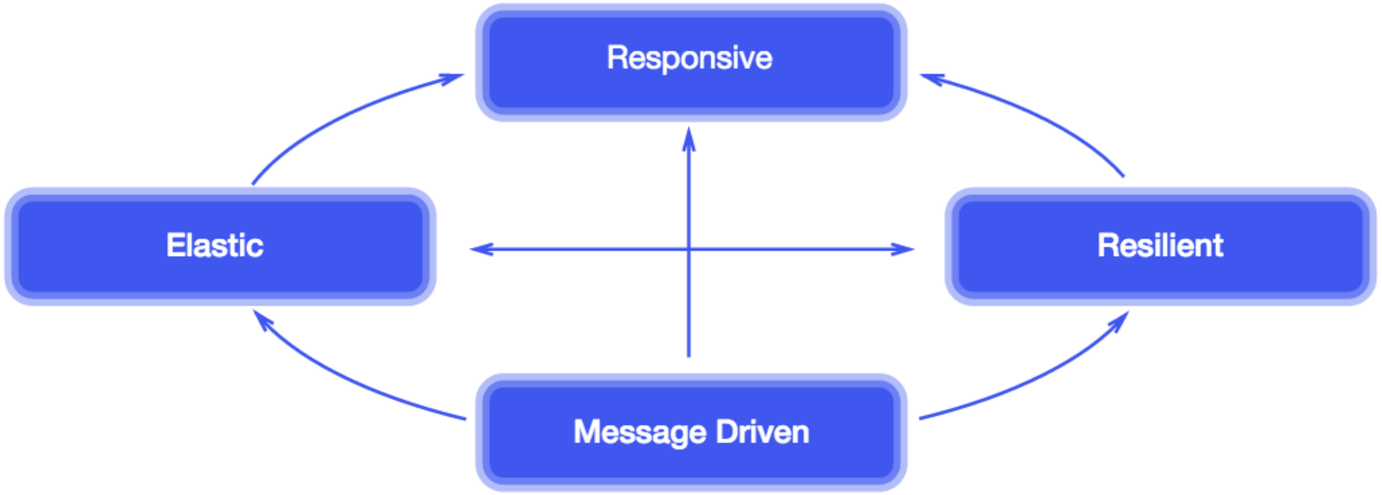
Kod Geliştirme Yöntemleri → ~~AOP~~ ~~TDD~~ ~~SOLID~~ ~~KISS~~ ~~DRY~~ ~~YAGNI~~

Kodu Dayanıklı Geliştirmek → ~~12 Factor App~~ Reactive Programming Chaos Eng

Design Patterns → Design Patterns, Creational Patterns, Structural Patterns, Behavioral Patterns

Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

Reactive Manifesto



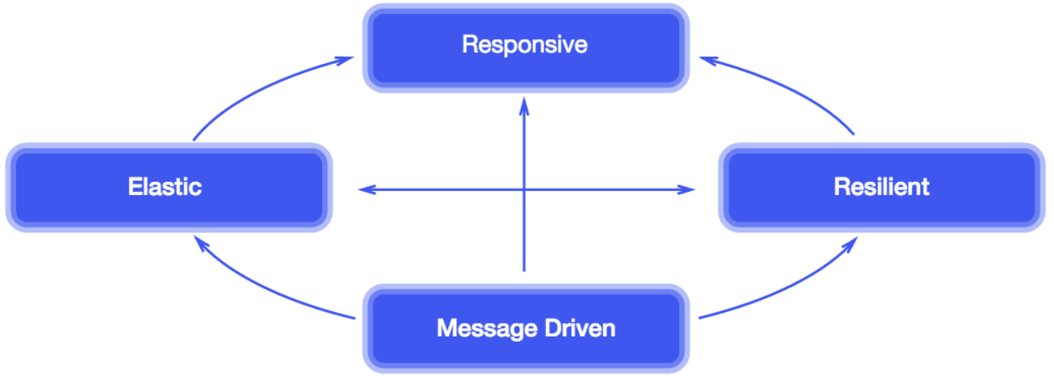
Reactive Yapılara Neden İhtiyaç Var ?

Günümüzün uygulama geliştirme ihtiyaçları değişmiştir. Eskiden kullanıcılar uygulamalarını belli bir kullanıcı kitlesine hitap ederken, bu uygulamaların sunucular ile etkileşimi belli sınırlar içerisindeyken, günümüzde mobil cihazların ve bunların üzerinde kullanılan uygulamaların çok hızlı şekilde artması.

Mobil cihazların sadece telefon ile sınırlı kalmadan Gözlük, Saat, Araba, Ev, Ev Aletleri, Akıllı şehirler(Internet Of Things) hepsinin internet üzerinden sunucular ile sürekli irtibat halinde durumunda olma ihtiyacı.

Uygulamaların kullanılma oranlarını **7/24 olması**, uygulamaların cevap verme durumunun **%100 istenmesi**. Elde edilen verilerin petabytes seviyelerinde olması. Mevcut yazılım mimari stillerin günümüz ile örtüşmediğini göstermektedir.

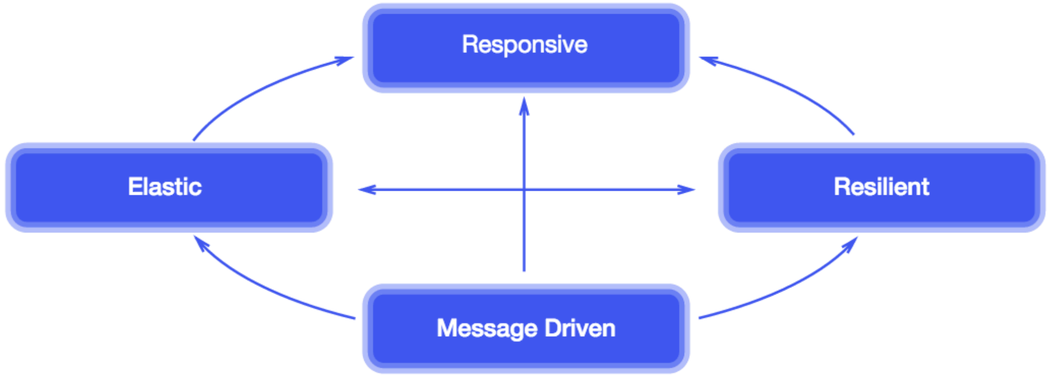
Elastic



Sunucularınıza gelen isteklerin çok artması ve azalması durumunda , sunucularınız **Dikey ve Yatay'da ölçeklenebilir** olması gerekmektedir. Mevcut bulut yöntemleri bu ölçeklenmeyi otomatik olarak yapabilmektedir. Ama sizin sisteminiz bu ölçeklenmeye uygun yazılıp yazılmadığı önemlidir.

Yada uygulamanızın A ,B, C servislerinden sadece B'ye gelen yük artıyor ise bu ölçeklenmeyi sadece B seviyesinde nasıl yapacağız(**Microservice**). Veritabanını nasıl böleceğiz, nasıl scale edeceğiz ? Uygulamanız bölündükten sonra ortak verileri nerden sağlayacak ?

Message Driven

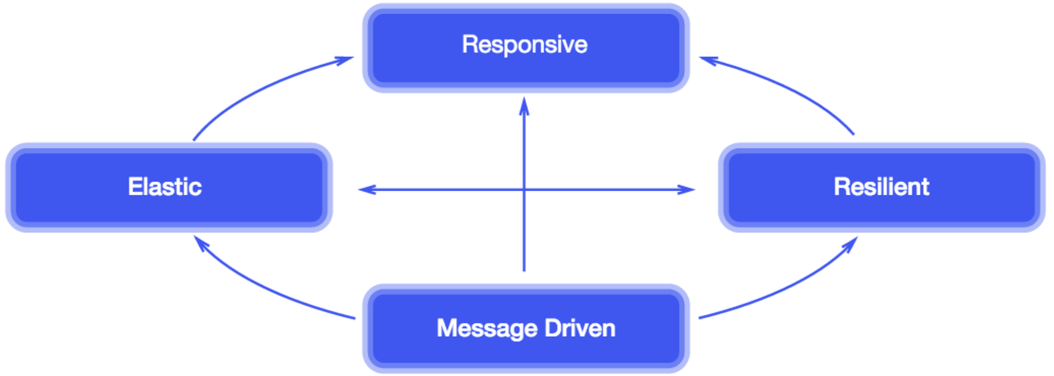


Ne kadar ölçeklenebilir bir sistem yapıyor olursanız olun. Mevcut sisteminiz içerisinde Disk Erişimi, Network Erişimi, CPU erişimi gibi durumlarda gelen istekleri **Bloklayacaktır.**

Bu durumu engellemek için ya **JVM seviyesinde Non-Blocking Queue, NIO(Netty)** veya Event Driven (Node.js) platformlar ile uygulamanızı geliştirmeniz gerekir.

Bunun haricinde JVM'ler arası sistemlerde **ActiveMQ, RabbitMQ veya Simple Message Queue** gibi çözümler ile sisteminizin bloklanmadan çalışmasını sağlayabilirsiniz. Sonuçta sisteminizde çok uzun sürebilecek ve arkadan gelen isteği almanızı engelleyecek durumlar olabilir. Bu durumları engellemek için sisteminizi Message-Driven geliştirmeniz gerekir.

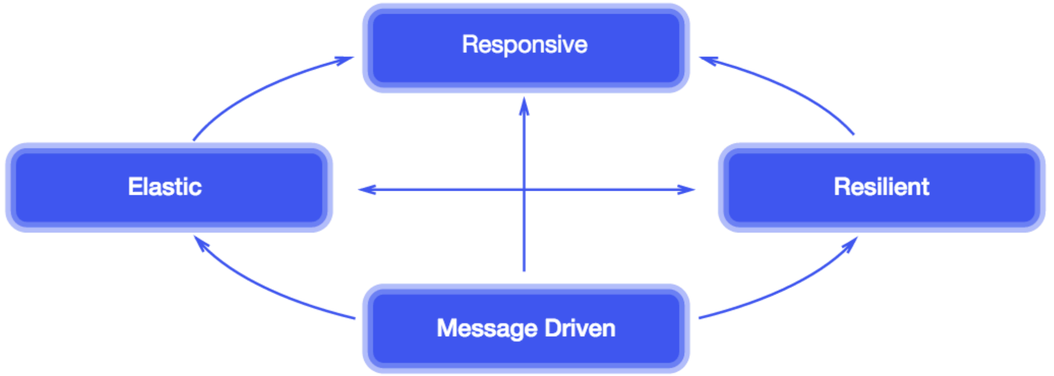
Resilient



Artık sadece Mission-Critical uygulamaların değil her türlü uygulamanın **hata durumlarına karşı toleranslı** geliştirilmesi gerekiyor.

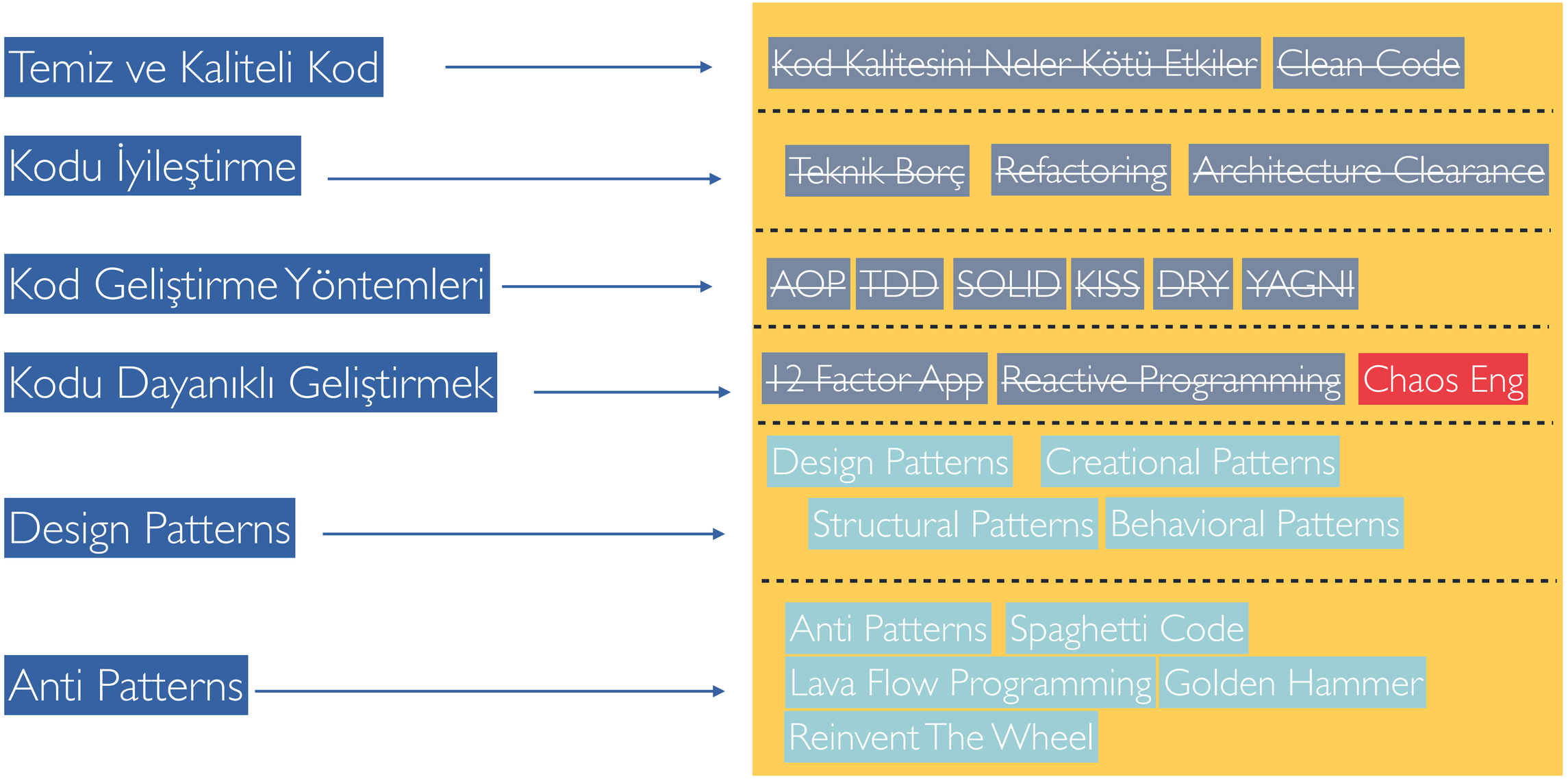
Örneğin sunuculardan birisinin kapanması durumunda **Load Balancer** bağlı diğer sunucuların işlerine devam etmesi. Veritabanının çökmesi durumunda Replikasyon veritabanının devreye girmesi, Mobil cihazın offline olması durumunda da Local veritabanı ile durumuna devam etmesi, internet bağlantısından sonra sunucu ile senkron olabilmesi vb..

Responsive



Sistemin kullanıcıya ve diğer sistemlere **yüksek kalitede hizmet** verebilmesidir. Cevap süresinin kullanıcıyı ve diğer sistemleri tatmin eder düzeyde yer almasıdır.

2nci Gün (Code Quality)



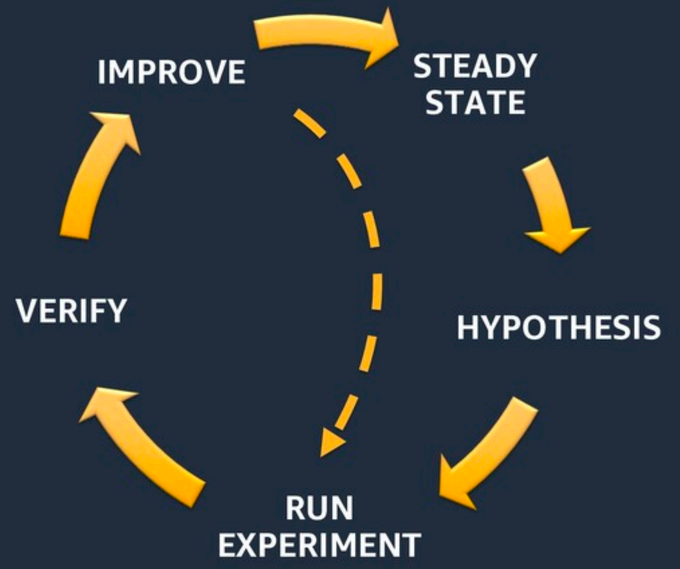
@ana_m_medina

Like a vaccine, we inject harm

to build ***immunity***.

Chaos Engineering

A scientific method

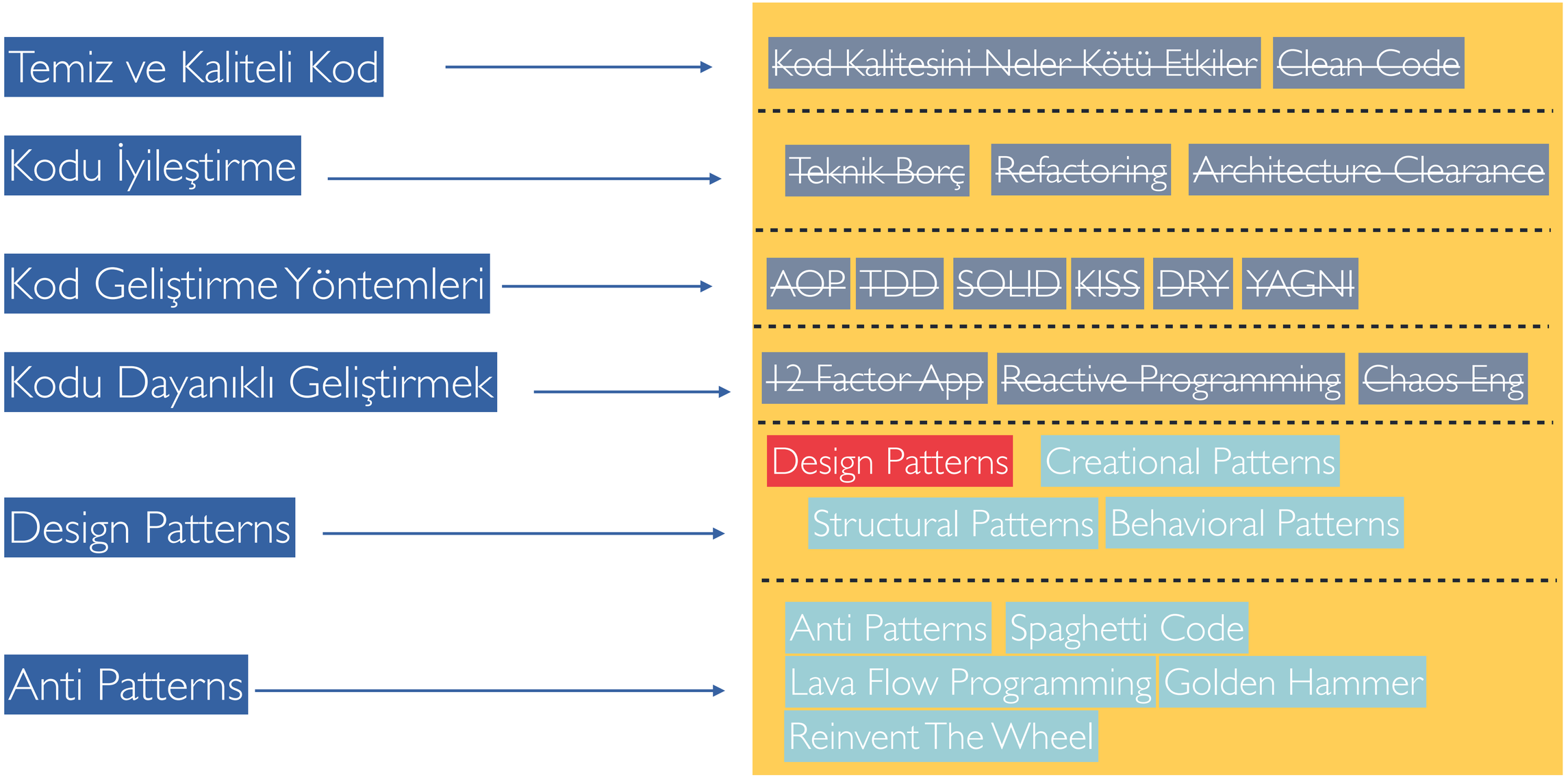


© 2020, Amazon Web Services, Inc. or its Affiliates.

aws

Design Patterns

2nci Gün (Code Quality)



Tasarım Örüntüsü Nedir ?

Tasarım örüntüleri insanların projelerinde sık sık karşılaştıkları problemler çözüm olarak getirdikleri **Problem-Çözüm** ilişkileridir.

Tasarım Örüntüsü Ne Sağlar ?

- Kodun daha esnek olabilmesini sağlamak (Extendable)
- Kodun tekrar tekrar kullanılabilmesi (Reusable)
- Değişime dayanıklı kodlar geliştirebilmek. (Min Effect)

Bazı Temel Kavramlar

- Abstraction (Soyutlama)
- High Cohesion, Low Coupling
- Dependency Injection

- Interface, Abstract, Class
- private, protected, public
- static, …

Tasarım Örüntüleri

1. Creational Patterns

- Abstact Factory
- Builder
- Factory Method
- Prototype
- Singleton

2. Structural Patterns

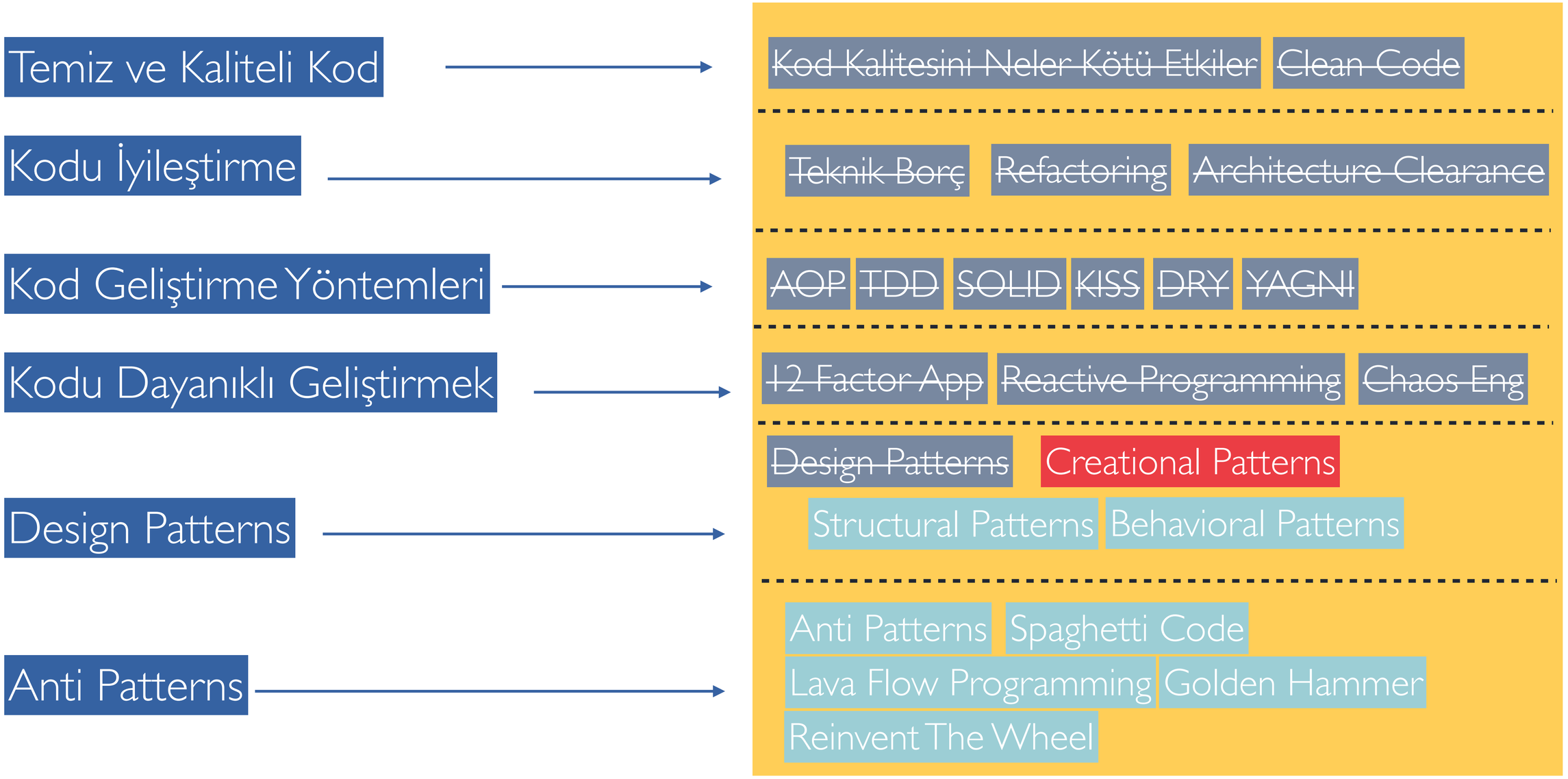
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

3. Behavioral Patterns

- Chain Of Responsibility
- Command
- Interpreter
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Creational Patterns

2nci Gün (Code Quality)



Creational Patterns

new keyword kullanımını yönetir !

Abstract Factory

Abstract Factory Hangi Durumda Kullanmalıyız ?

Uygulamalarda birbirine bağımlı bir grup sınıfı A durumu için oluşturup, B durumu için bir başka bir grup sınıfı oluşturmanız gerekir. Bu durumda mevcut çözüm iki durumu **if/else** veya **switch** yazmak şeklinde olacaktır.

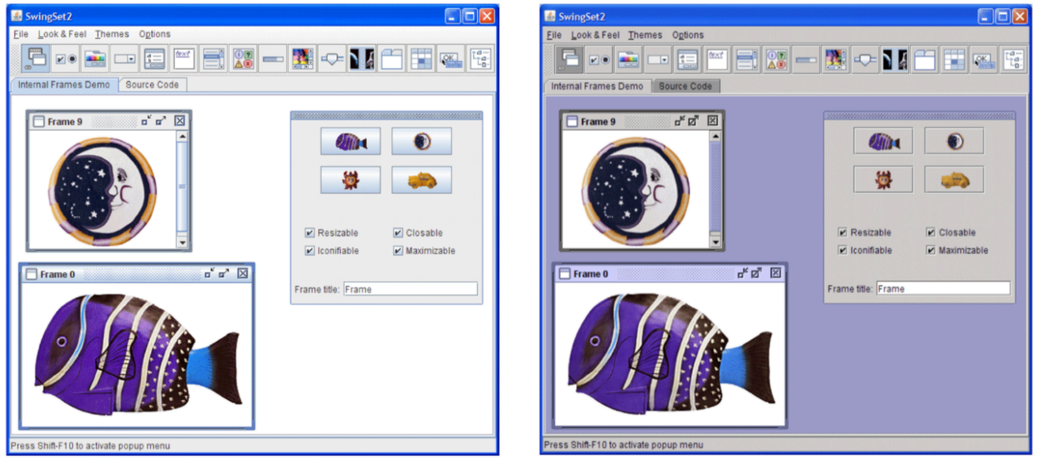
```
class Factory

if(durumA)
   new X_AClass, new Y_AClass, new Z_AClass

else if(durumB)
   new X_BClass, new Y_BClass, new Z_BClass
```

Abstract Factory

Abstract Factory bize **benzer işlevi gerçekleştiren** farklı **grup bileşenleri/sınıfları** üretme sırasında karşımıza çıkacak **karmaşayı nasıl önleyebileceğini** anlatmaktadır.

Abstract Factory için Örnek

Örneğin İşletim sistemine göre Swing bileşenlerinin farklı görüntülenmesi gerecektir. Windows, Linux, MacOS vb işletim sistemlerinde tüm UI bileşenlerinin farklı farklı rendering ihtiyaçları olacak. Bu durumda Look And Feel değiştirme ihtiyacımızın olduğu durumda



Ocean Theme / Steel Theme

Bu bileşenleri üretme kodunun mantığını Factory içerisinde yeni **if/else** ekleyerek gerçekleştirebiliriz. Ama bu her bir durum için oluşturulacak bileşen sayısı çok fazla olduğu durumlarda bu sınıfın kodları oldukça karmaşık hale gelecektir. Yeni bir işletim sistemi geldiğinde bu ana sınıfta oldukça büyük değişiklikler yapmak gerecektir.

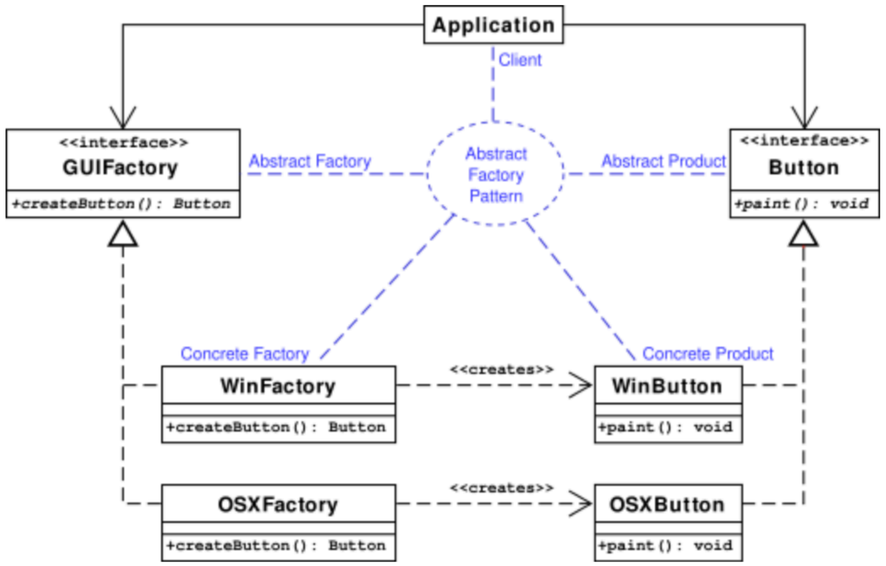
Bu karmaşıklığı önlemenin en iyi yöntemide Sınıf Grubu üretme mantıklarını belli sınıflarda toplamaktır.

Abstract Factory Kullanmanın Avantajları

Aşağıdaki örnekten üzerinden anlatırsak

WinFactory ve OSXFactory gibi 2 tane üretim sınıfı oluşturduğumuzda

- Kendi bileşenlerini üretmekten sorumlu hale gedi. Sorumluluk alanı daraldı ve daha kontrol edilebilir hale gedi.
- Bu tek bir sınıfın çok büyümesini engelledi. Sadece kendi sorumluluğundaki bileşenleri bilmesini sağlandı.



Abstract Factory Class Diagram

Abstract Factory Değişimden Nasıl Etkilenir

- Yeni bir işletim sistemi eklendi: Diğer Factory sınıfları bu durumdan etkilenmeyecek , yeni bir tane daha eklenecek. GUIFactory sadece bir satırlık if/else eklenecek.
- Yeni UI Bileşeni eklenmek isteniyor. Bu durumda AbstractFactory (GUIFactory) bu bileşen oluşturma mantığı eklenmeli ve bundan türeyen Concrete sınıflar (WinFactory ve OSXFactory) bu geliştirmeye göre kendi yapılarını bunu eklemeliler ve bunun ile ilgili bileşenler oluşturulmalı.

Kullanım Alanları

Örneğin: Aynı bileşenler aşağıdaki modlara göre farklı davranması gerekebilir.

- UI Gündüz Modu / Gece Modu
- UI Player/ Trainer
- Editing Mode / Playing Mode / Replay Mode

Bu gibi durumlarda tüm UI bileşenlerinin bu modlara göre keskin çizgiler ile ayrılması gerekiyor ise aşağıdaki kod örneğinde olduğu gibi Abstract Factory örüntüsünü kullanırız.

Örneğin: Bir bileşen yazdınız Query mekanizmasının işletilmesini sağlıyor. ConsoleUI, WindowUI, YardımEkranı, Parser, Mapper vb.. ortak sınıfları bulunuyor. bu durumda. Bir bağlamda Query mekanizması için tüm bu nesnelerin oluşturulması , başka bir bağlamda aynı yapıların başka sınıfları oluşturulması gerekeceği durumlarda bu Abstract Factory Tasarım örüntüsünden faydalanabiliriz.

Abstract Factory Kod - I

```
//GuiFactory example

//Abstract Product
interface Button {
        void paint();
}
//Abstract Product
interface Label {
        void paint();
}

//Abstract Factory
interface GUIFactory {
        Button createButton();
        Label createLabel();
}

//Concrete Factory
class WinFactory implements GUIFactory {
        public Button createButton() {
                return new WinButton();
        }

        public Label createLabel() {
                return new WinLabel();
        }
}

//Concrete Factory
class OSXFactory implements GUIFactory {
        public Button createButton() {
                return new OSXButton();
        }

        public Label createLabel() {
                return new OSXLabel();
        }

}
```

Örneğimizi incelersek Interface oluşturulacak UIBileşenleri ***(Button, Label)*** ve bunları oluşturacak AbstractFactory sınıfı olan ***(GUIFactory)*** GUIFactory hangi bileşen arayüzlerini oluşturacağını biliyor sadece

```
Button createButton();
Label createLabel();
```

GUIFactory'den türeyen bu gruplamayı sağlayacak ConcreteFactory sınıflar bulunuyor. Bunlar ***WinFactory ve OSXFactory*** bunlar gerçekleştirim bileşenleri oluşturmakten sorumlular.

Abstract Factory Kod -II

```
//Concrete Product
class OSXButton implements Button {
        public void paint() {
                System.out.println("I'm an OSXButton");
        }
}

//Concrete Product
class WinButton implements Button {
        public void paint() {
                System.out.println("I'm a WinButton");
        }
}

//Concrete Product
class OSXLabel implements Label {
        public void paint() {
                System.out.println("I'm an OSXLabel");
        }
}

//Concrete Product
class WinLabel implements Label {
        public void paint() {
                System.out.println("I'm a WinLabel");
        }
}

//Client application is not aware about the how the product is created. Its only responsible to give a name of
//concrete factory
class Application {
        public Application(GUIFactory factory) {
                Button button = factory.createButton();
                Label label = factory.createLabel();
                button.paint();
                label.paint();
        }
}

public class ApplicationRunner {
        public static void main(String[] args) {
                new Application(createOsSpecificFactory());
        }

        public static GUIFactory createOsSpecificFactory() {
                String osname = System.getProperty("os.name").toLowerCase();
                if(osname != null && osname.contains("windows"))
                        return new WinFactory();
                else
                        return new OSXFactory();
        }
}
```

Application ekrana Label ve Button render etmek istiyor ama hangi Win/OSX Factory ile çalışıp çalışmadığını bilmek zorunda değil. Bu sayede bu servisi kullanan kişi kendisini işletim ***sisteminin detaylarından kendisini soyutlamış*** oluyor.

Burda işletim sistemi detayına göre hangi AbstractFactory döneceğini bilen ApplicationRunner ama oda GUIFactory ne dönüyor , içerlerinde ne logicler var bilmek zorunda değil .

Bu tasarımda herkes kendi işinden sorumlu 😃

- **Application** bana Button, Label ver paint et diyor sadece
- **ApplicationRunner** → Hangi işletim sistemi olduğunu bilip ona göre doğru Concrete GUIFactory dönmekten.
- **GUIFactory** → Sadece üreteceği bileşenlere ait metodlara sahip metodlarda hangi Interface döneceğini biliyor
- **WinFactory, OSXFactory** → kendi Concrete UI bileşenleri nasıl üreteceklerini biliyorlar
- **Button, Label** → bileşen interface bileşen mekanizmalarının hangi özelliklerinin ve metodlarının olması gerektiğini tanımlıyorlar.
- **WinButton,WinLabel** → Windows işletim sisteminde düğme ve etiketin nasıl paint edileceğini biliyorlar

Builder Pattern

Builder Hangi Durumda Kullanmalıyız ?

Aşağıdaki sınıfı incelediğimizde 4 tane özelliği bulunuyor. Bunlar *x,y,width, height.* 4 tane özelliği bu nesnenin anlamlı olması için gerekli. Bu durumda bu nesne için 1 tane constructor yeterli olacaktır.

```
class Rect
   public Rect(int x, int y, int width,int height) //Constructor
```

Bu tarz Rect gibi oluşturulması basit yani 1,2 constructor oluşturarak nesneyi oluşturabiliyorsanız bunlara oluşturulması basit nesneler diyebiliriz.

Ama bazı nesneler vardır ki bunların çok fazla parametresi bulunur. Ve bunlardan x1,x3 ile başka nesne görünümü oluşturulurken, x4,x5 ile başka bir nesne görüntüsü oluşturulabilir durumların sayısı çok arttığında tüm kombinasyonları karşılayacak bir constructor oluşturmak oldukça zor olur.

```
class Complex
   int x1,x2,x3,x4 ...................
   constructor1(x1, x2)
   constructor2(x3, x4)
   constructor3(x5, x1)
   .....
```

Builder

Builder Pattern bize **kompleks nesneleri oluşturmakta** zorlandığımız durumlarda nasıl bu işlemi **kolaylaştırabileceğimiz** konusunda bir yaklaşım sunar.

Builder Hangi Durumda Kullanmalıyız ?

Örneğin bir ***SQL cümlesi oluşturacaksınız*** Select her zaman olacak ama Where olmak durumunda değil, GroupBy olmak durumunda değil vb…

Örneğin bir ***yemek sipariş nesneniz*** var. İstenilen yemek parçalarına göre Hamburger, Patates, Kola olsun veya yanında İçecek olmasın gibi nesnenizin farklı farkı görüntüleri olabilir

Bu durumda bir diğer yöntem boş/default constructor ile sınıfı oluşturup bu nesneye ***set veya add*** gibi metodlar ile oluşturabilirsiniz.

```
class Complex
   setX1()
   setX2()
   setX3()
   setX4()
```

```
class Complex
   setX1()
   setX2()
   setX3()
   setX4()
```

Bu durumda sınıfınız bir çok set metodu oluşacaktır. Örneğin bir Sipariş bilgisini aşağıdaki şekilde yazalım.

```
//Sipariş Oluşturma işlemi
Order order=new Order();
order.addHamburger(type, count)
order.addDrink(type, count)
order.addPizza(type, count)
......
```

Burada kodu yazarken sipariş oluşturma işi bir bütün olarak gözükmez. Çünkü geliştirici arada başka başka işlemlerde yapabilir

```
//Sipariş Oluşturma işlemi
Order order=new Order();
order.addHamburger(type, count)
order.addDrink(type, count)
sum = 3+2;
callFunction()
order.addPizza(type, count)
......
```

Builder Avantajları

Ama bunu Builder Pattern ile yazdığımızda her kendi basit DSL(Domain Specific Language) oluşturacak ***this*** döneceği için daha odaklı bir nesne oluşturma yapısı oluşturabiliriz.

Nested Builder

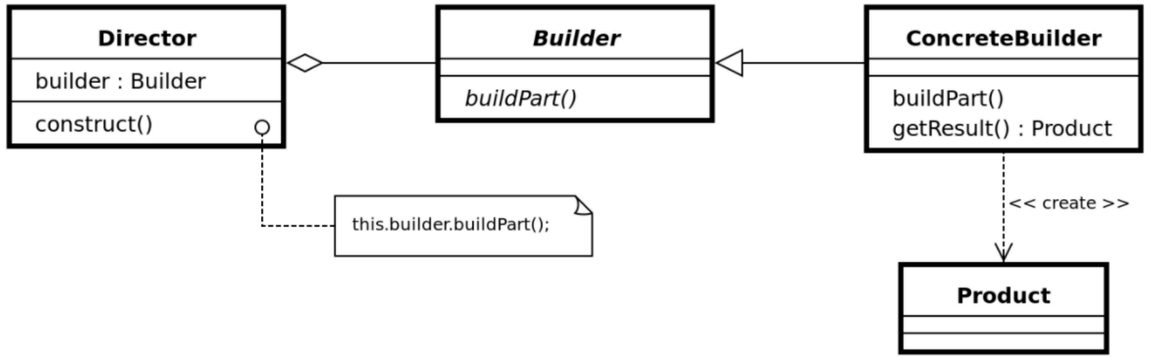
```
new Order.Builder().hamburger(ABurger,2).drink(BDrink,1).build()
```

Peki bunu nasıl gerçekleştireceğiz. Kompleks nesnemizin içerisine bunun üretilmesinden sorumlu bir Builder objesi olacak bu bizim Burada kompleks nesne oluşturma mantığını nesnenin kendi kendi içerisindeki bulunan Builder

```
public class Order
   ArrayList<Enum,int> hamburgers
   ArrayList<Enum,int> drinks
   ArrayList<Enum,int> pizza

        public static class OrderBuilder
             //return this
              public OrderBuilder hamburger(type, count){}
              public OrderBuilder drink(type, count){}
              public OrderBuilder pizza(type, count){}

            //return Order
             public buid(){ return new Order()..}
```

Not: Burada Order default constructor private hale getirmek , set metodlarını dışarıya kapatmak bu kompleks nesnenin olabildiğince kontrollü oluşmasını sağlayacaktır.

Builder Yapısı



Builder Pattern Class Diagram

Diğer bir yöntemde Product türünü oluşturabilecek ara Builder interface gerçekleştirmiş sınıflar farklı türdeki nesneleri oluşturma adımlarını kendi içlerinde saklar ve buna göre oluştururlar.

Örneğin aşağıda Araba nesneleri oluşturmak isteyelim. Ferrari, Opel, Mercedes, BMW, WV vb.. türlerde oluşturmak.

Builder Pattern Örneği

```
/// <summary>
/// The director
/// </summary>
public class SportsCarBuildDirector
{
    private ICarBuilder _builder;
    public SportsCarBuildDirector(ICarBuilder builder)
    {
        _builder = builder;
    }

    public void Construct()
    {
        _builder.Colour = "Red";
        _builder.NumDoors = 2;
    }
}
```

```
/// <summary>
/// The builder abstraction
/// </summary>
public interface ICarBuilder
{
    string Colour { get; set; }
    int NumDoors { get; set; }

    Car GetResult();
}
```

```
/// <summary>
/// Concrete builder implementation
/// </summary>
public class FerrariBuilder : ICarBuilder
{
    public string Colour { get; set; }
    public int NumDoors { get; set; }

    public Car GetResult()
    {
        return NumDoors == 2 ? new Car("Ferrari", "488
Spider", Colour, NumDoors) : null;
    }
}
```

```
/// <summary>
/// Represents a product created by the builder
/// </summary>
public class Car
{
    public string Make { get; set; }
    public string Model { get; set; }
    public int NumDoors { get; set; }
    public string Colour { get; set; }

    public Car(string make, string model, string colour, int
numDoors)
    {
        Make = make;
        Model = model;
        Colour = colour;
        NumDoors = numDoors;
    }
}
```

```
public class Client
{
    public void DoSomethingWithCars()
    {

        var builder = new FerrariBuilder();
        var director = new SportsCarBuildDirector(builder);

        director.Construct();
        Car myRaceCar = builder.GetResult();
    }
}
```

Factory Method

Factory Method Hangi Durumda Kullanmalıyız ?

Bir arayüzden türemiş farklı farklı nesnelerimiz/bileşenlerimiz olduğunu düşünelim. Bu sınıflara ihtiyacımız olduğu yerlerde **new** ile oluşturduğumuz durumda tüm Concrete sınıfların kullanımı tüm sınıflara dağılmış olması istenilen bir yöntem değildir. Yeni nesne oluşturma ve bunun nesnenin yönetimi olabildiğince kontrollü bir şekilde yapılması geliştirilen uygulamalar için faydalı olacaktır.

Factory Method

Factory Method bize **tek bir arayüzden(interface) türetilmiş sınıfların** üretilmesi işlemini soyutlayan bir örüntüdür.

Factory Method Örnek

Shape: Şekil interface arayüzü sadece ***draw*** metodu bulunuyor. Bundan türeyen

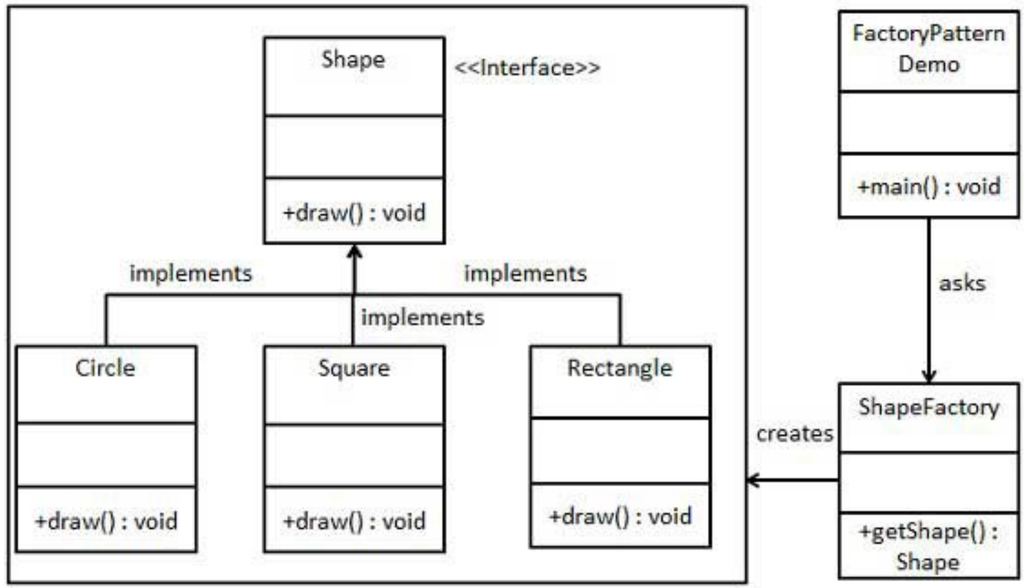
- Circle, Square Rectangle ….. şeklinde uzunca bir liste olsun.

Burda bu Shape türlerini kullanacak Client sınıflar. Arka planda Concrete nesneleri bilip bunları oluşturmak ile uğraşmak yerine ***type vb*** bilgileri verip bu sınıf oluşturma sorumluluğunu başka bir sınıfa bırakması oldukça faydalı olacaktır. Bu durumda

ShapeFactory : Shape oluşturulmasından sorumlu sınıf type hangi concrete sınıfları üreteceğini bilmesini. Bu üretim mantığını tek bir sınıf içerisinde tutup, bir geliştirme ve eklenme yapılacağı zaman sadece bu sınıfta değişiklik olması sağlanır.

Type bilgisinin String olması durumunda bir array içerisindeki [circle, square, …..] bilgileri dinamik oluşturmasını

İstemci sınıfta Concrete sınıflara ***import*** bağlantının olmamasını sağlar.



Factory Method Kod

Shape.java

```
public interface Shape {
   void draw();
}
```

Shape Interface

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Concrete Classes

ShapeFactory sınıfının görevi şekil oluşturma işlemini soyutlamak. Aldığı parametreye göre bir şekil oluşturuyor. Yarın sisteme yeni bir şekil eklemek istediğimizde değiştireceğimiz kısım sadece ShapeFactory içerisindeki bu method olmuş olacak. ***getShape*** metodu **new** metodunu soyutlamış olduk.

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

ShapeFactory Sınıfı

Şimdi bu Methodu dışarıdan bir istemcinin nasıl kullandığına bakalım. Öncelikle bir ShapeFactory oluşturuyor. Burdan istediği tipteki şekli ShapeFactory istiyor ve sadece draw metodunu çağırıyor.

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();
```

Prototype

Prototype Hangi Durumda Kullanmalıyız ?

Normal zamanda nesnenin oluşması için geçen süre kısadır. Fakat oluşturduğumuz nesnenin bir takım bilgileri bir dosyadan veya veritabanından veya bir servisten doluyorsa bu nesneyi ***"cache'lememiz"*** avantajlı hale gelecektir.

Veya bir nesne oluşturacağız sınıftaki öğrenciler örneğin. Bu nesnede bir sürü ortak özellik olduğu, sadece isim , soyad ve no değişecek ise ***new*** *ile* nesne oluşturmak yerine mevcut bir nesnenin ***clone*** oluşturup bu yeni nesnede daha az değişiklik yaparak kullanabiliriz.

Bazen orjinal nesne üzerinde (state) üzerinde işlem yapmak yerine onun clone üzerinde işlem yapıp sonrasında bu nesneyi atmakta isteyebiliriz.

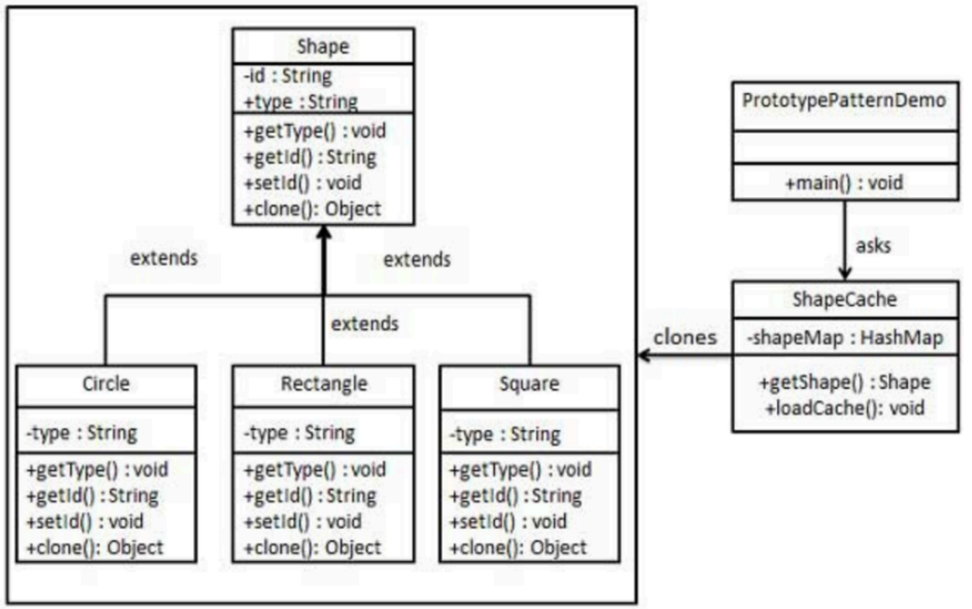
Tüm bu işlemler yapabilmek için ***Shape*** arayüzünde **clone** metodu olmalı. Bundan türeyen sınıflar Circle, Rectangle, Square kendisini nasıl clone oluşturacağını bilmelidir.

Prototype

Prototype örüntü, nesnenin bir **kopyasını oluşturmanın zahmetli** olduğu, zaman aldığı durumlar için **bellekteki** bir kopyasının oluşturularak kullandırır.

Prototype Yapısı

Burda immutable primitif değerler için ***Shallow Copy*** işe yarayabilir ama referans nesneler için ***Deep Copy*** yapılması gerekmektedir.



Prototype Pattern Class Diagram

Aşağıdaki örnekte görüldüğü gibi Shape sınıfı ***"Cloneable"*** sınıfından türemektedir. Bundan oluşturulan Rect, Square, Circle gibi sınıflar klonu oluşturulabilen nesnelerdir.

Prototype Kod

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Shallow vs Deep Copy

Shallow(Sığ) kopyalama esnasında sadece first-level (birincil) hiyerarşideki değerleri kopyalar. Spread operatörü' de bu kopyalamaya göre çalışır.

Deep(Derinlemesine) kopyalama ise bütün alt hiyerarşileri gezip her bir değeri kopyalar bu zahmetli ve uğraştırıcıdır ama referansların birbirini etkilememesi için deep copy zorunludur.

Shallow Copy

```
let obj = { a: [1, 2, {users: [{ name: 'onur', age: 12 }] }], b:2 };
const cloneObj={…obj}; //same => Object.assign({}, obj);
obj.a[2].users[0].name='deniz';
obj.b=4;
console.log(cloneObj);//Result interesting..
//cloneObj.b is 2 not 4 because it clones first level and cloneObj
holds a new memory location for b
//but cloneObj.a[2].users[0].name is 'deniz' why because it hold
reference cloning is not deeply.
```

Deep Copy

1. Serialize/Deserialize : *Hızlı kopyalama için — JSON.parse/stringify* kullanabilirsiniz.

2. lodash.clonedeep

```
const clonedeep = require('lodash.clonedeep')
```

3. Recursive olarak nesneyi dolaşarak birbirine atama yaparak kopyalama yapabilirsiniz.

```
function clone(obj) {
     if (obj === null || typeof (obj) !== 'object' ||
'isActiveClone' in obj)
        return obj;    if (obj instanceof Date)
        var temp = new obj.constructor(); //or new Date(obj);
    else
        var temp = obj.constructor();    for (var key in obj) {
        if (Object.prototype.hasOwnProperty.call(obj, key)) {
            obj['isActiveClone'] = null;
            temp[key] = clone(obj[key]);
            delete obj['isActiveClone'];
        }
    }
    return temp;
}
```

Prototype Kod - I

```java
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

Rectangle.java

```java
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```java
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```java
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Prototype Kod - 2

```
import java.util.Hashtable;

public class ShapeCache {

   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>

   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }

   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes

   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);

      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

Prototype Avantajları

Eğer mevcut durumda elimizde bir A arayüzünden türeyen AConcrete olsun bunun bir çok özelliği var ise ve bu clone nesnelerin içerisine almadığımız durumda kodun farklı yerlerinde kendi ***new*** nesnemizi oluşturup buna değerleri setleme işlemlerini dışarı taşımamız gerekir.

```
aClone=new AConcrete2()
aClone.x=instance.x
....
```

OO Geliştirmede en önemli konulardan birisi **new** işlemini olabildiğince kontrollü yapmaktır. Bundan dolayı new yapılan kısımlardaki nedenselliklere göre tasarım örüntüleri oluşturulmuştur.

Singleton

Singleton Hangi Durumda Kullanmalıyız ?

Oluşturulan sınıfın proğram içerisinde tek instance olmasını istediğimiz durumlarda bu örüntüyü kullanırız. Yani sisteminizi cross cut eden tek bir çıkış noktası olan nesne türleri için;

- Logging
- Communication
- Database
- Access
- OpenGL (Rendering Context)
- OpenAL, FMod (Audio Context)
- vb …

Singleton

Singleton üretilen nesnenin sistemde tek **bir tane** olmasını **garantileyen** bir üretim örüntüsüdür.

Singleton Hangi Durumda Kullanmalıyız ?

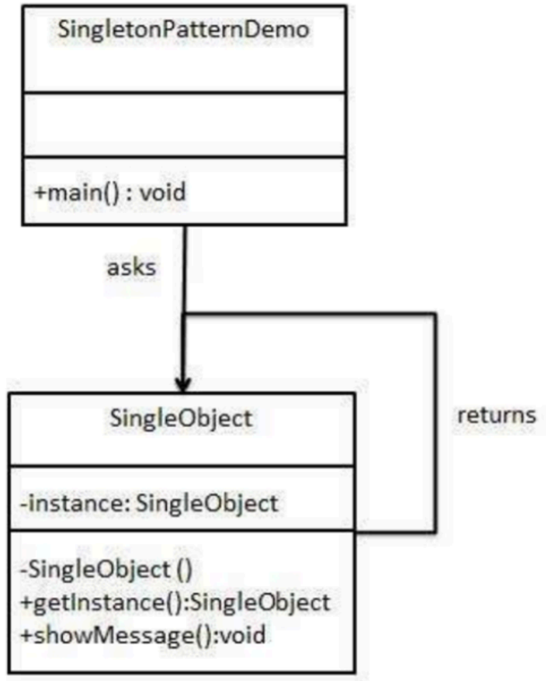
Burada amacımız N tane sınıf üretmek değildir. Çünkü bu sınıflar belli işlemleri yapan sınıflardır. Processor sınıflardır.

- *Loglama'da* dosyaya/console log yazar.
- *Communication* ağ üzerinden paket gönderir.
- *Database'de* sorgu veya update yapar.
- OpenGL grafik kartına çizim komutları gönderir.

Tüm bu işlemler sırasında tek bir Logger, tek bir DB, tek bir Grafik kartınız var ise N tane nesne üretmeniz anlamsızdır. Tek bir nesne üzerinden işlemleri yapmanız Context(Bağlamı) kaybetmemeniz gerekir.

Singleton Yapısı

Tüm bu işlemler sırasında tek bir Logger, tek bir DB, tek bir Grafik kartınız var ise N tane nesne üretmeniz anlamsızdır. Tek bir nesne üzerinden işlemleri yapmanız Context(Bağlamı) kaybetmemeniz gerekir.



Singleton Pattern Class Diagram

Aşağıdaki SingleObject oluşturulan instance bir tanesi olmasını garantiler. Bu işlemi lazy olacak şekilde de yapabilir yani getInstance çağrıldığında da bu işlemi yapabilir

```
getInstance()
   if(!instance) intance=new SingleObject()
   return intance;
```

Veya daha ilk aşamada **new** bu Objeyi oluşturup, tüm süreç boyunca bu objenin kullanılmasını sağlar. Burada SingletonObject oluşturabilmek için default constructor private hale getirmeyi unutmamak gerekiyor.

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

SingleObject (Singleton)

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Creational Patterns Özet

Abstract Factory: Nesne grupları oluşturabilecek soyut fabrikadır

Builder: Kompleks nesneleri oluşturmanızı sağlar

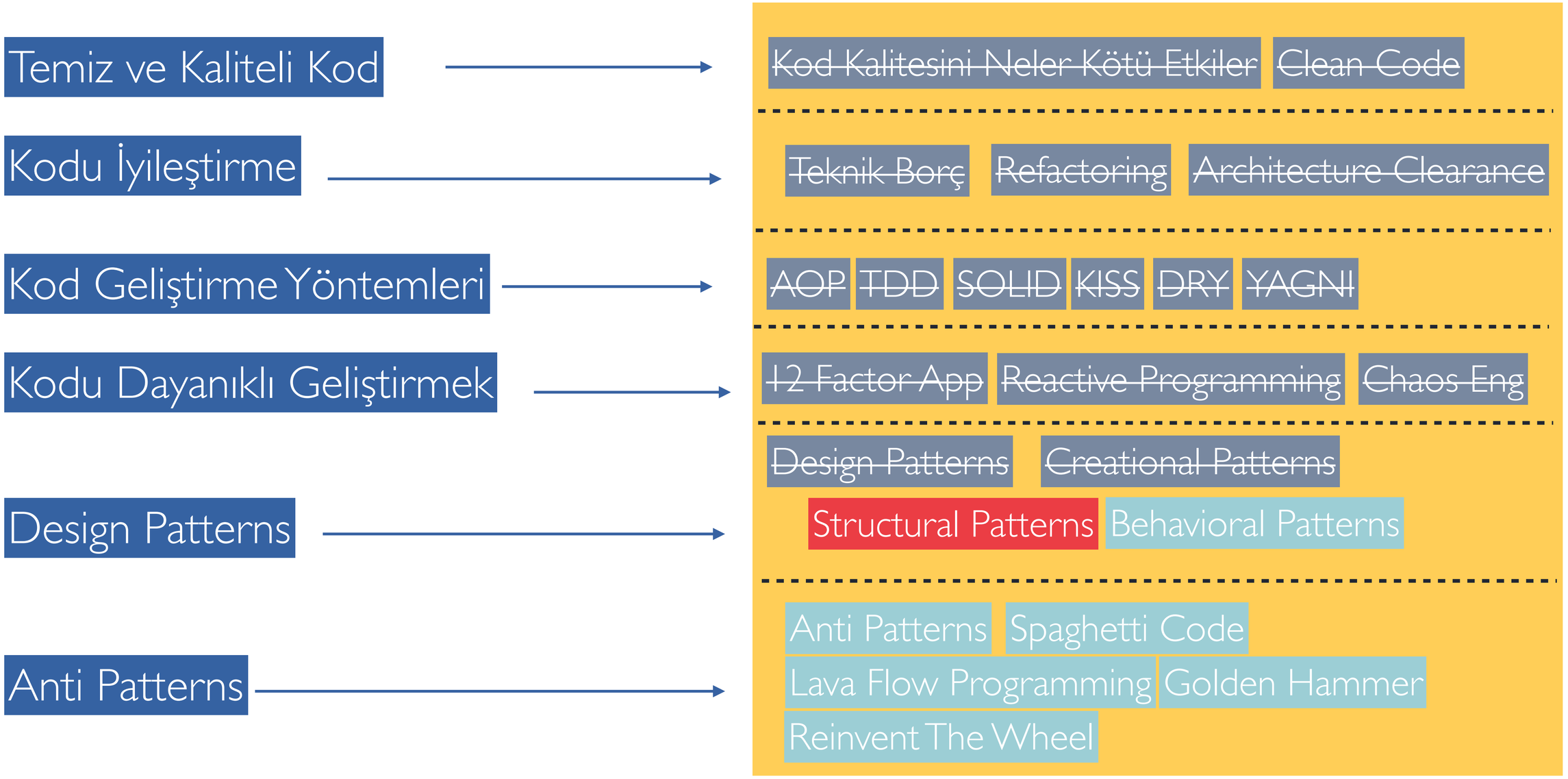
Factory Method: Asıl sınıfı bilmeden nesneler oluşturmanızı sağlar.

Prototype: Mevcutta bulunan bir sınıftan yeni bir sınıf oluşturur.

Singleton: Nesnenin sistemde bir referansı olmasını sağlar

Structural Patterns

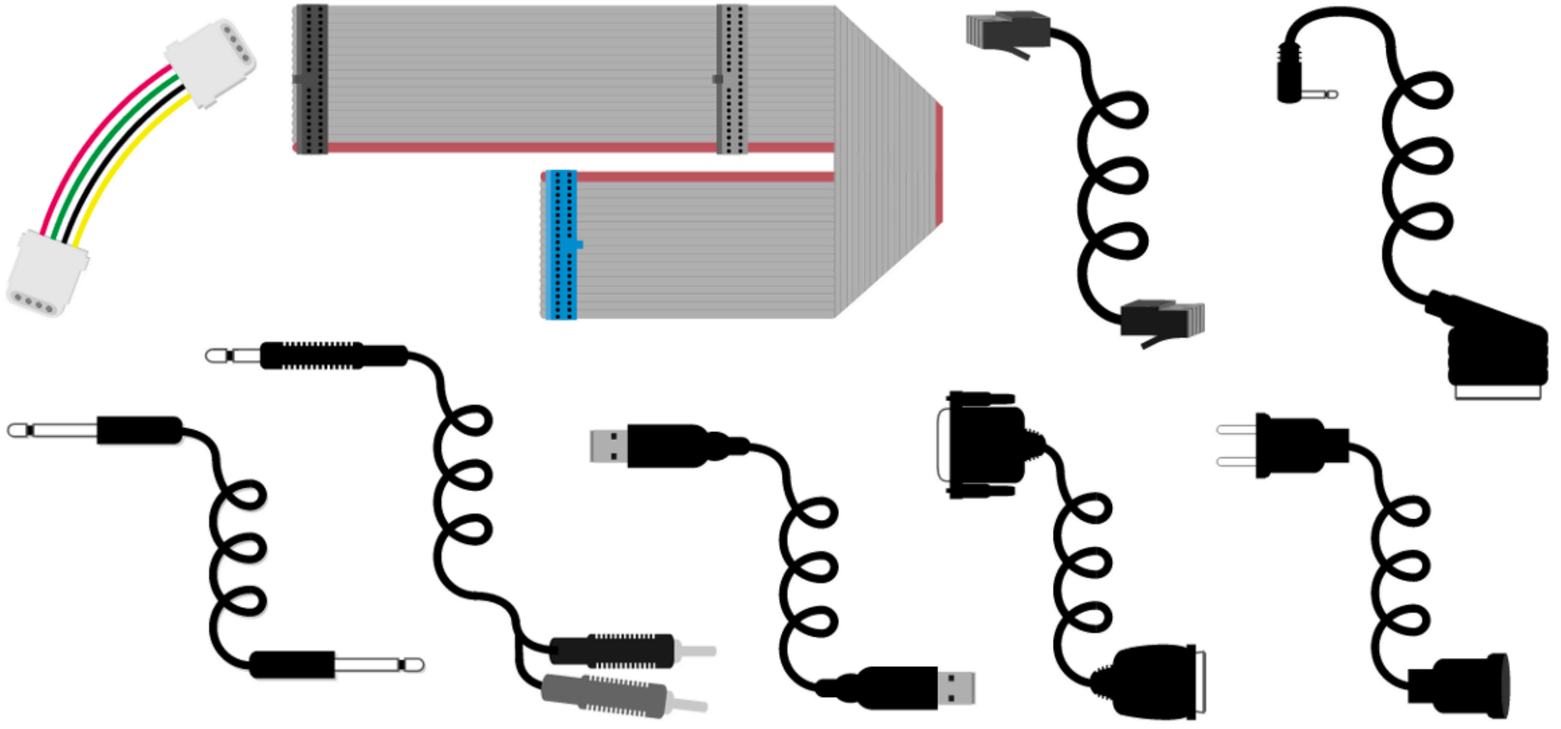
2nci Gün (Code Quality)



Structural Patterns

Sınıflar arası kurulan doğru **ilişki/bağlantılar** üzerinden kurduğu **yapılar** ile ilgilenir bunları düzenlemeye ve anlamlandırmaya çalışır.

Adapter



Adapter Hangi Durumda Kullanmalıyız ?

Bu durum yazılımda sizin veri yapınızın ihtiyaçlarının veya formatının dışarıdan kullandığınız bir kütüphane, servis vb uyuşmaması durumunda ortaya çıkar.

Siz kodunuzu geliştirirken sizin kullandığınız iş bağlamında o özellikleri farklı bir isimde veya formatta tuttuyor olabilirsiniz. Ama bunu bir ***Rendering*** kütüphanesi ile kullanıyorken, ***I/O kütüphanesi*** ile vb başka bir kütüphanede kullanıyorsanız kütüphane veya servisin kendi Interface olabilir.

Sizin veri yapınız bu Interface(Arayüze) uymadığında mevcut sınıflarınızı değiştirmek yerine bu sınıfların çevresini kaplayacak(wrapping) adapter sınıflar oluşturulur. Bu sayede mevcut çalışan yapıları bozmadan diğer kütüphaneler ile çalışabilecek hale gelmiş olursunuz.

Adapter

Adapter örüntü gerçek hayatta karşılığı olan bir tasarım örüntüsüdür.
Bilgisayar ve TV için bir çok adaptör bulunur çünkü üretilirken
belli standartlara göre üretilen bu 2 uç aslında karşı tarafın ihtiyacı olan
veri yapısını farklı bir formatta tutmaktadır. Adaptör bunun dönüşümünü
gerçekleştirerek **2 yapının bir arada çalışabilmesini** sağlar.

Adapter Örnek Kullanımlar

JDK Adapter Pattern Örnekleri

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Android UI Örneği

Örneğin Android'de **ListView'lerin** kullandığı ArrayAdapter sınıfıda Adapter örüntüsünü kullanır. Amacı Veri Modeli ile Arayüz arasında bağlantıyı sağlayabilmektir.

Adapter Örnek Kullanımlar

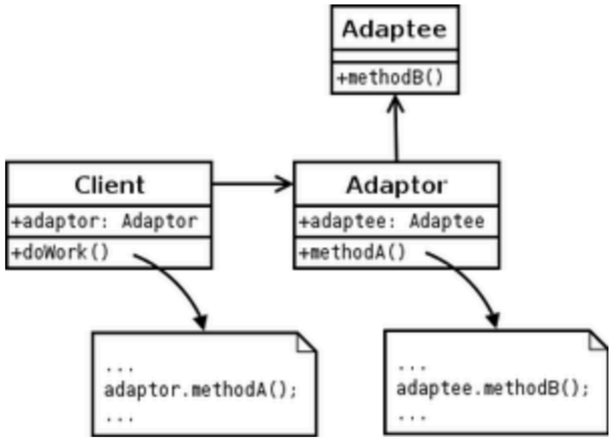
JDK Adapter Pattern Örnekleri

- java.util.Arrays#asList()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Android UI Örneği

Örneğin Android'de **ListView'lerin** kullandığı ArrayAdapter sınıfıda Adapter örüntüsünü kullanır. Amacı Veri Modeli ile Arayüz arasında bağlantıyı sağlayabilmektir.

Adapter Yapısı



Object Adaptor

Object adaptör Client(İstemcinin) esas çağrı yapacağı nesnenin **methodA** fonksiyonu olmaması nedeniyle araya koyduğumuz Adaptor bu **methodA** isteğini karşılar ve bunu içerisinde referansını tuttuğu nesneden çağrımlar yaparak çözer. Bu ***runtime*** referans nesnenin Adaptor geçirilerek yapılır.

Adapter Kod

SceneGraph Kütüphanesi

```
interface Shape {
    void draw(int x, int y, int z, int j);
}
```

```
public class AdapterDemo {
    public static void main(String[] args) {
        Shape[] shapes = {new RectangleAdapter(new Rectangle()),
                          new LineAdapter(new Line())};
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (Shape shape : shapes) {
            shape.draw(x1, y1, x2, y2);
        }
    }
}
```

```
class LineAdapter implements Shape {
```

```
class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
}
```

Mevcut Kodunuz

```
class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "), to p
    }
}
```

```
class Rectangle {
    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" + x + ";"
                + ", height: " + height);
    }
}
```

Değişim İhtiyacını Nasıl Kontrol Altında Tutacağız ?

Örnekte görüleceği gibi mevcut yapılarımızda Line ve Rectangle çizdirme kodumuz çalışıyor olsun. (Şu anda Sysout ile console yazdırıyoruz ama siz ekrana çizdirdiğinizi düşünün) Bir SceneGraph kütüphanesi iterate ederek veritabanından okuduğu şekilleri çizdirecek ve bizim sistem ile entegrasyon sağlamak istiyoruz.

Bu durumda mevcut kodumuzu değiştirmek istemiyoruz çünkü başka yapılar bizim kodumuzu önceden kullanıyordu.

- Method değişikliği yaparsak daha önceden Rectangle sınıfını kullanan diğer sınıflarımızıda değiştirmemiz gerekir
- Sınıfları **Shape** extends edersek bu sefer her böyle yeni Interface gelen metodlardan dolayı mevcut sınıfımız mantığı büyüyecek.
- Bu durumda en iyi yöntemlerden birisi mevcut sınıfların çevresini kaplayacak wrapper eklemek. ***(Yeni sınıf ekliyoruz. Mevcut yapıya dokunmuyoruz)***

Adapter Yöntemi Nerede Yetersiz Kalabilir

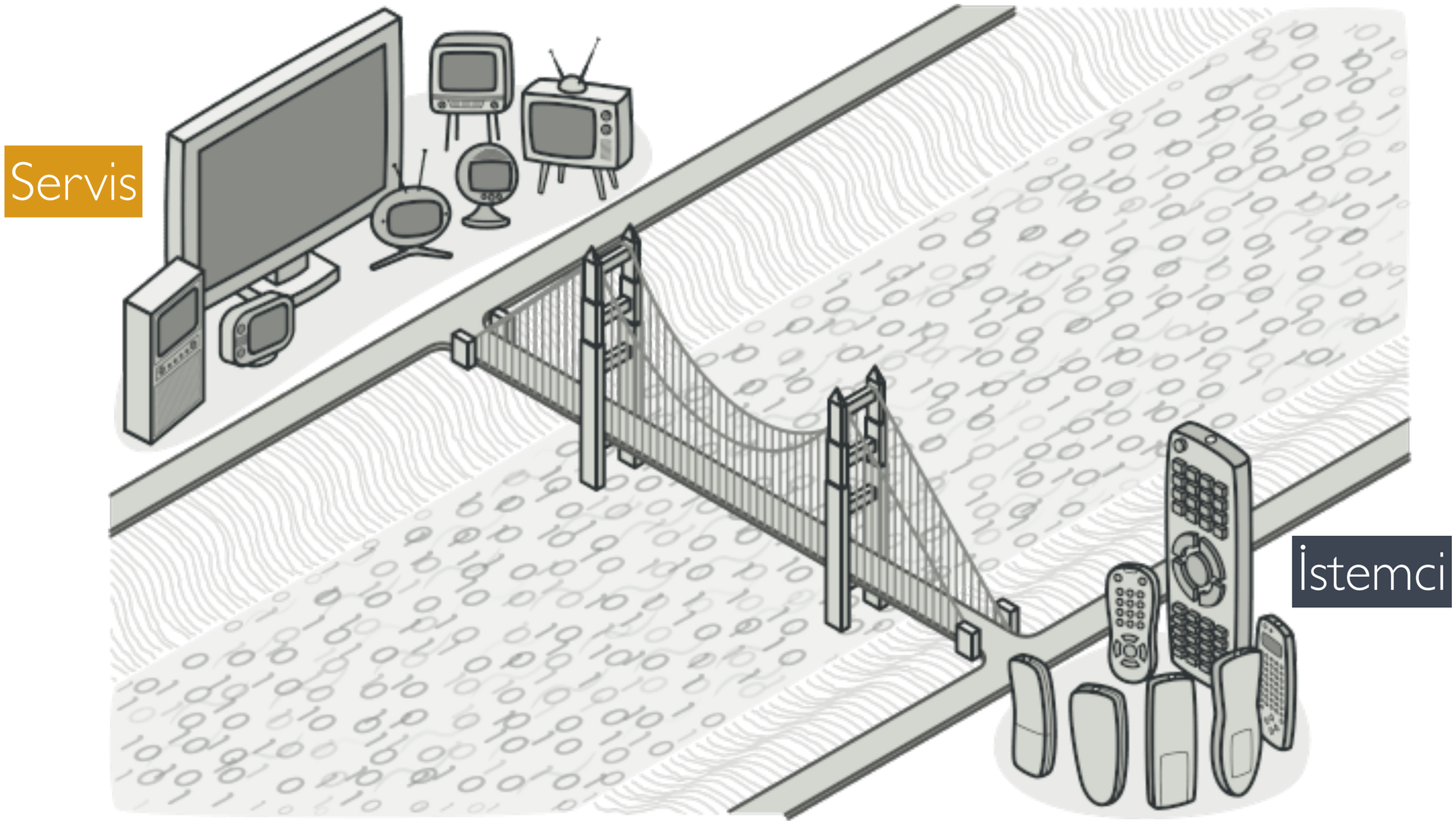
Önceden çalıştığım bir projede 2 uygulama Senaryo Editörü ve Senaryo Oynatıcısı ortak nesneleri kullanıyordu ama farklı ekipler ve farklı ihtiyaçlardan dolayı ayrı zamanlarda geliştirilmiş ve veri yapılarında farklılıklar ortaya çıkmıştı.

Bu 2 uygulamanın birlikte çalışması ve birlikte aynı yapılar içerisinde kullanılması gerekmişti. bu durumda 40–50 farklı entity nesnesinin hepsi biri diğerine benzetilmek için Adaptor sınıflar yazıldı ve beraber çalışılması sağlandı.

Sonradan analizlerimiz sonrasında mevcut entity yapılarının farklı olmadığını sistemde 50 ‘tane entity için fazladan adaptor nesnesinin olduğuna ve yeni nesneler geldikçe ekleme yaptığımız sınıf sayısının çok fazlalaştığını farketmiştik.

Ve bu işin basit bir ***Refactoring*** değil ***Architecture Clearance*** *(Mimari düzeltmeler)* yapmamız gerektiği ve bu entity farklılaşmasını engellememizin gelecekte işleri yönetmek açısından çok önemli olduğunu farkettik. Ve Adaptör yöntemini terk ederek işi daha temelde çözmeye yöneldik.

Bridge



https://refactoring.guru/

Bridge Hangi Durumda Kullanmalıyız ?

Sisteminizde 2 farklı yapı grubu var. Bu grupların farklı farklı gerçekleştirimleri bulunuyor. Bu yapıların birbirlerini gerçekleştirim seviyesi bilmesi yerine belli interface(arayüzler) üzerinden referanslarını tutarak kullanması, karmaşık hiyerarşik yapıların önüne geçecektir.

Bridge

Bridge farklı katmanlarda bulunan **gerçekleştirimleri** **soyutlayarak** birbirlerini iletişim kurabilecekleri bir köprü yapısı kurarak **hiyararşik karmaşıklığın** önüne geçmeye çalışır.

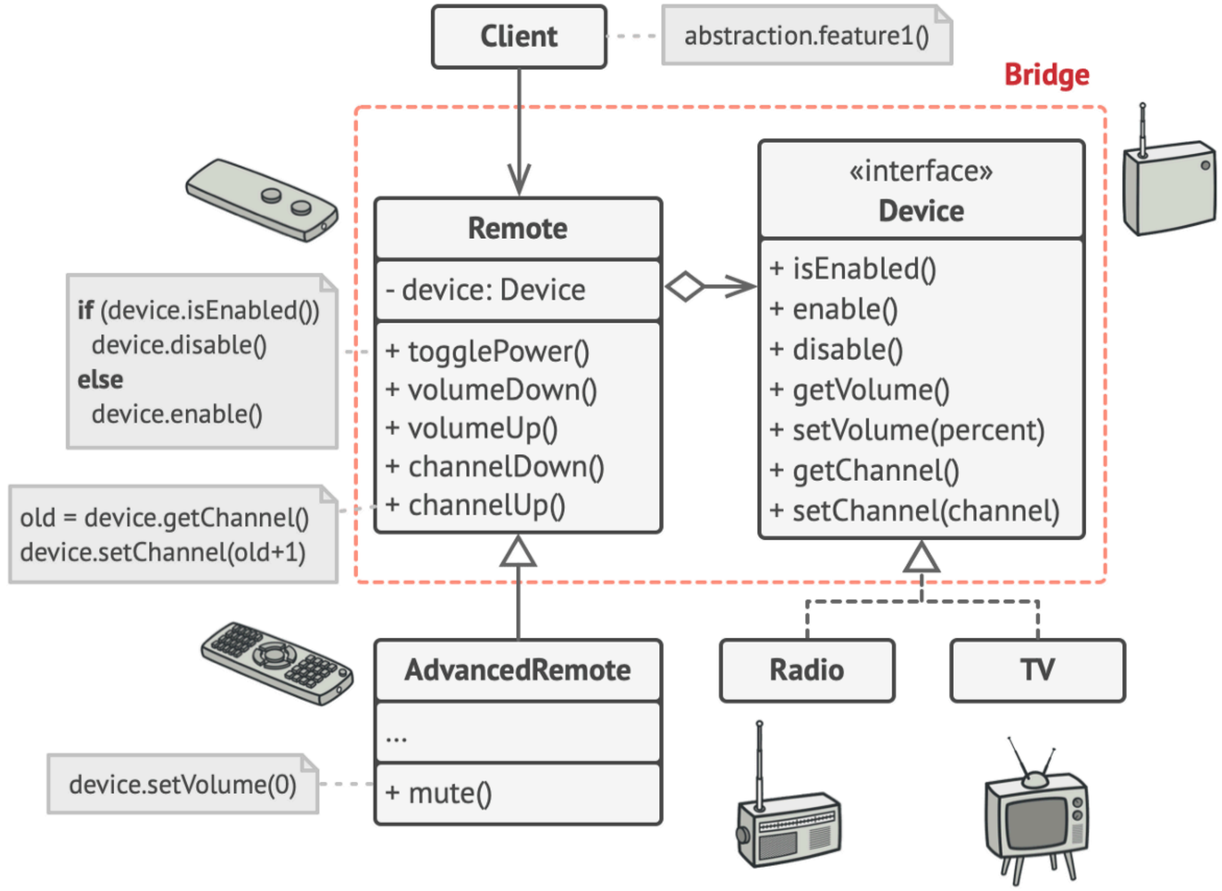
Bridge Örnek Kullanım - I

Uzaktan Kumandalar ve Akıllı Ev Cihazları

Uzaktan Kumandalar (Eski nesil 2 düğmeli, Çok düğmeli ve Touchpad vb kumanda çeşitleri olsun) Cihaz Grubu (TV, Radyo, Müzik Seti) olun.

Kumandalar bu cihazları kontrol etmek istediklerini düşünelim. Bu durumda En son Teknoloji TV ve en eski TV veya az özellikli Radyo vs için 1000 lerce kumanda üretilmesi gerekir.

Bu gerçekleştirim seviyesi karmaşadan kaçmak için kullanılacak sistemler kısmında yani cihazlar kısmında bir soyutlama yapılması



https://refactoring.guru/

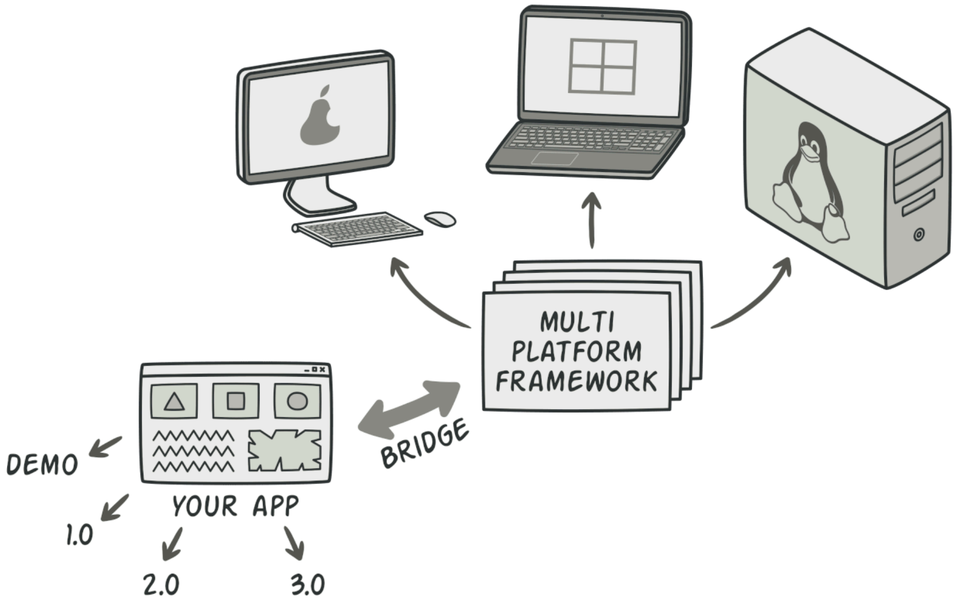
Bridge Örnek Kullanım - 1

UI Bileşenleri ve İşletim Sistemleri

UI Bileşen Grubu(Düğme, TextField vb) ve İşletim Sistemi Grubu (MacOS, Linux, Windows) bunların I/O mekanizmaları ve Rendering mekanizmalarında farklılıklar bulunur.

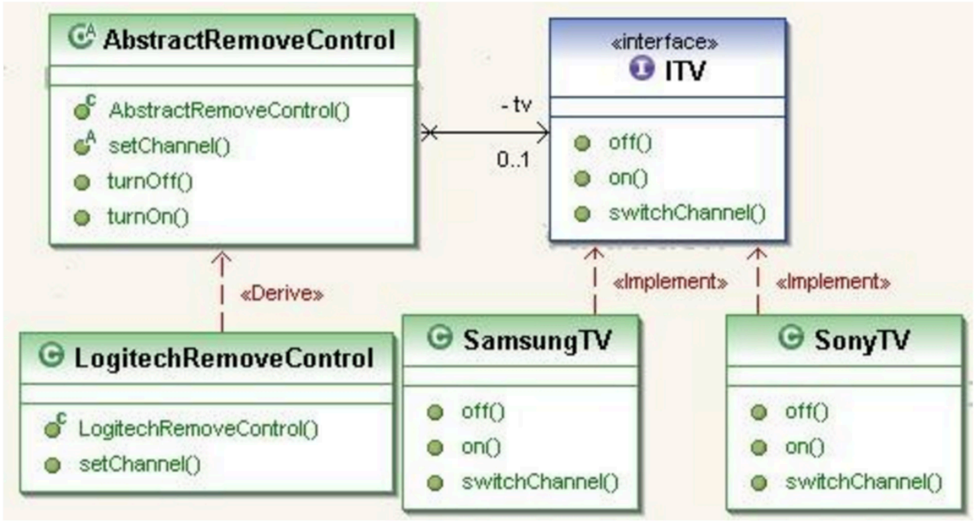
Eğer biz gerçekleştirim olarak Implementation seviyesinde bu nesneleri yazarsak WindowsTextField, MacOsTextField … gibi sayılarda sınıflar oluşturup bu logicleri bunların içerisinde gerçekleştirmemiz gerekecek.

Bu her bir bileşen ve işletim sistemi eklenmesi durumunda geometrik katlanarak artacaktır.



https://refactoring.guru/

Bridge Kod - I



Yukarıdaki örneğe baktığımızda 2 farklı soyutlama katmanı bulunmaktadır. Uzaktan kumanda ve Televizyon.

- Logitech Uzaktan Kumanda → AbstractRemoveControl türüyor
- Televizyon → ITV türüyor

Elimizde 2 farklı yapı bulunmakta. TV'nin açılıp kapanma ve televizyon kanallarının seçiminin yer aldığı sınıflar. **TV**

```
public interface ITV {
        public void on();
        public void off();
        public void switchChannel(int channel);
}
```

```
public class SamsungTV implements ITV {
        @Override
        public void on() {
                System.out.println("Samsung is turned on.");
        }

        @Override
        public void off() {
                System.out.println("Samsung is turned off.");
        }

        @Override
        public void switchChannel(int channel) {
                System.out.println("Samsung: channel - " + channel);
        }
}
```

```
public class SonyTV implements ITV {

        @Override
        public void on() {
                System.out.println("Sony is turned on.");
        }

        @Override
        public void off() {
                System.out.println("Sony is turned off.");
        }

        @Override
        public void switchChannel(int channel) {
                System.out.println("Sony: channel - " + channel);
        }
}
```

Bridge Kod -II

```
public class Main {
        public static void main(String[] args){
                ITV tv = new SonyTV();
                LogitechRemoteControl lrc = new LogitechRemoteControl(tv);
                lrc.setChannelKeyboard(100);
        }
}
```

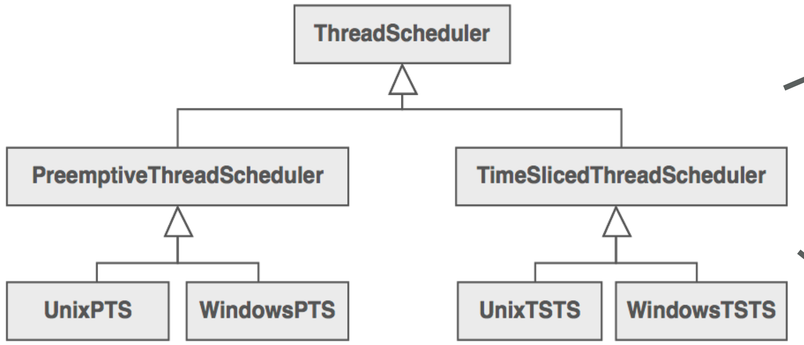
Burda LogitechRemoteControl hangi TV ünitesini çalıştırdığını arkaplanda delegation yapılırken ITV hangi metodlarının çalıştığı ile ilgilenmez. Tek bilmesi gereken AbstractRemoteControl sınıfı ve buradan çağırabileceği metodlar.

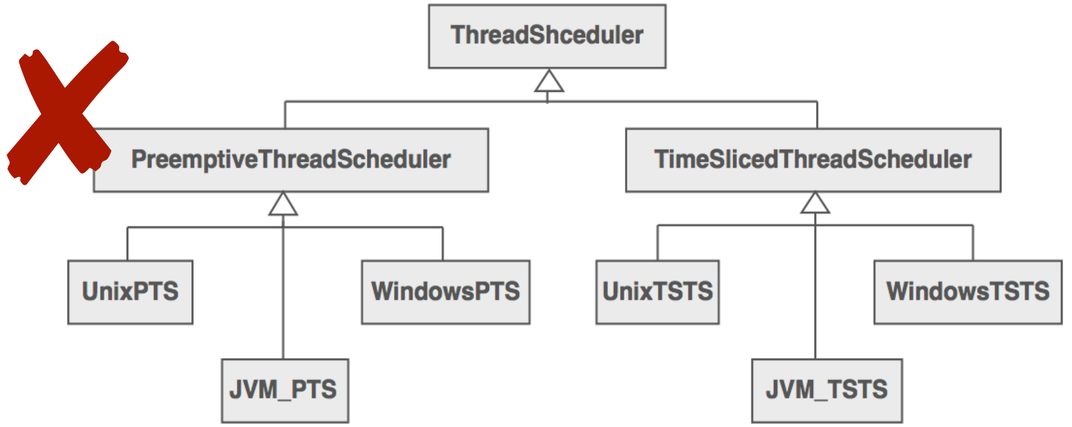
AbstractRemoveControl ile TV üzerinde erişim sağlayarak Bridge kurarak gerçekleştirimleri çalıştırır.

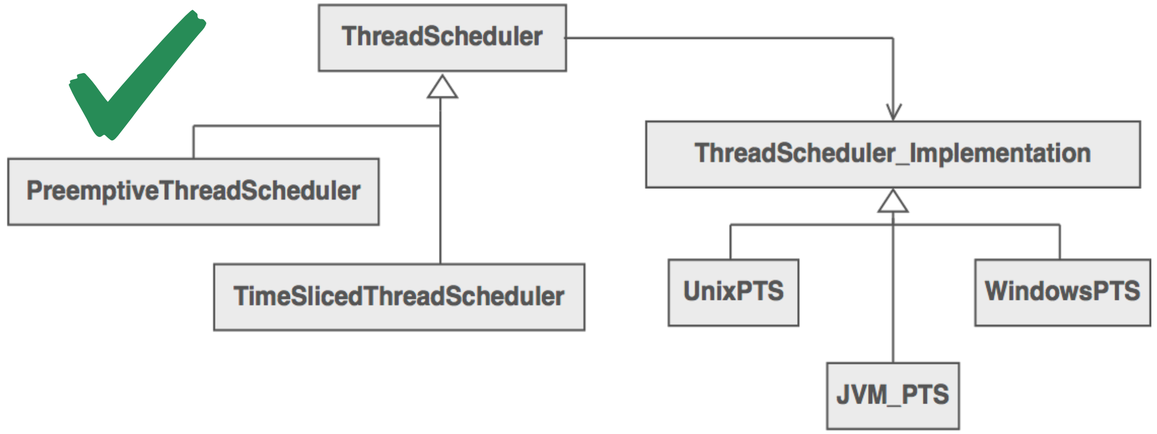
```
public abstract class AbstractRemoteControl {
        /**
         * @uml.property  name="tv"
         * @uml.associationEnd
         */
        private ITV tv;

        public AbstractRemoteControl(ITV tv){
                this.tv = tv;
        }

        public void turnOn(){
                tv.on();
        }

        public void turnOff(){
                tv.off();
        }

        public void setChannel(int channel){
                tv.switchChannel(channel);
        }
}
```

```
public class LogitechRemoteControl extends AbstractRemoteControl {

        public LogitechRemoteControl(ITV tv) {
                super(tv);
        }

        public void setChannelKeyboard(int channel){
                setChannel(channel);
                System.out.println("Logitech use keyword to set channel.");
        }
}
```

Bridge Örnek Kullanım -2







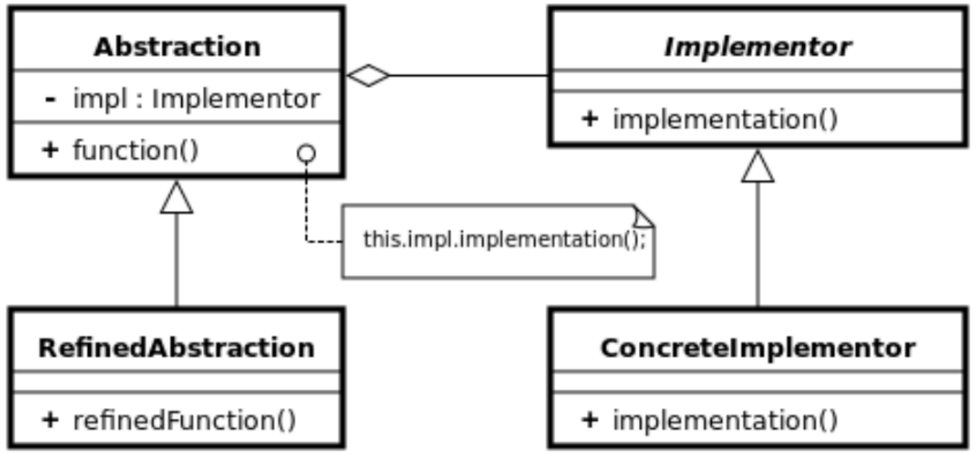
Thread ve İşletim Sistemi

Örnekte Bridge örüntüsünün avantajını daha net görebilirsiniz. Elimizde Thread türleri olsun 2 tip(Preemptive,TimeSliced) ve bu **thread'lerin** çalıştığı 2 tip(Unix, Windows) ortamı olsun.

Şimdi yeni bir ortam eklemek istediğimizde örneğin JVM platformunu bu sisteme eklemek istediğimiz zaman mevcut sistemin 2x2 → 2x3 → 6 olacak ve bu mekanizmada sistem çarparak büyüyecek. (Geometrik Büyüme)

Bridge Yapısı

Servis olarak kullanılacak ***ConcreteImplementor*** bir arayüz ile soyutlanır. Bu ***Implementor*** Interface, bu servisi kullanıcı grubu sınırlandıracak Abstract bir sınıftan türetilir. Ve bundan türetilen sınıflardan yapılan çağrılar delegasyon yöntemi ile Implementor sınıfına yönlendirilir.



Composite

Composite Hangi Durumda Kullanmalıyız ?

Ağaç veri yapısı türünde olup örneğin ortak bir method içeriyorsa

- ***hoverInput***→ Menu yapıları (PopupMenu, Menus, TreeMenu vb..),
- **draw** → UI Component yapıları (JPanel, JTextfield vb..)
- **calculateSize** → Klasör ve Dosya yapıları

Bu durumda Composite(iç içe bir yapı) oluşturup iç içe recursive fonksiyonların çağrımların yapılması sağlanır

Composite

Composite örüntü **ağaç veri yapısında** olan
kendi içerisinde kendi türünden çocuk sınıf listesi tutan
ve **belli ortak işlevleri** olan yapılar için kullanılan bir örüntüdür.

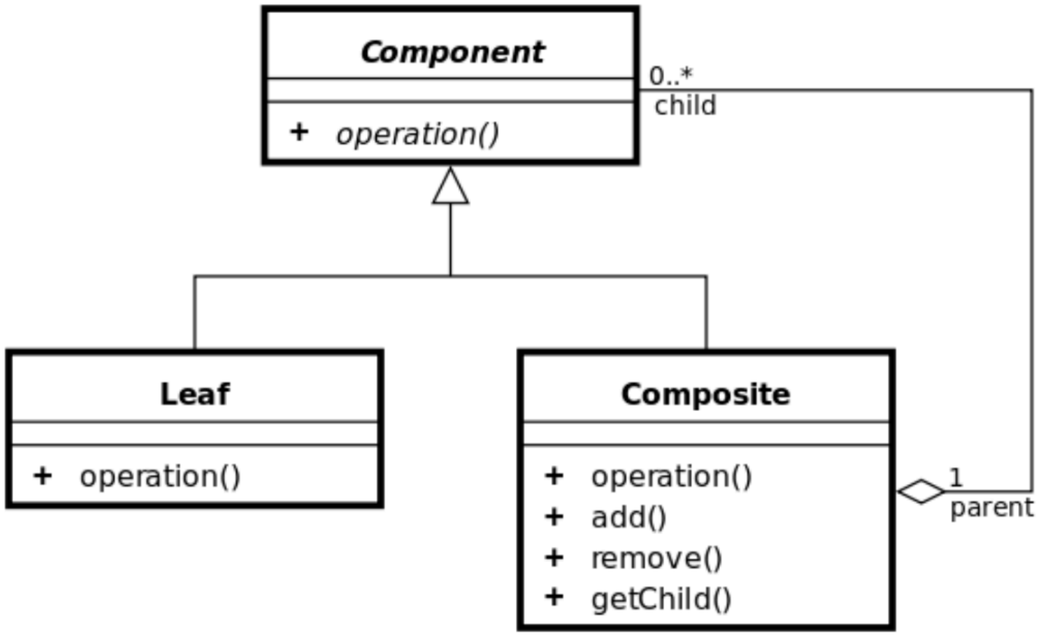
Composite Hangi Durumda Kullanmalıyız ?

Ağaç veri yapısı türünde olup örneğin ortak bir method içeriyorsa

- ***hoverInput***→ Menu yapıları (PopupMenu, Menus, TreeMenu vb..),
- **draw** → UI Component yapıları (JPanel, JTextfield vb..)
- **calculateSize** → Klasör ve Dosya yapıları

Bu durumda Composite(iç içe bir yapı) oluşturup iç içe recursive fonksiyonların çağrımların yapılması sağlanır

Composite Yapısı



Composite Kod

```
public class CompositeDemo {
    public static StringBuffer compositeBuilder = new StringBuffer();

    public static void main(String[] args) {
        Directory music = new Directory("MUSIC");
        Directory scorpions = new Directory("SCORPIONS");
        Directory dio = new Directory("DIO");
        File track1 = new File("Don't wary, be happy.mp3");
        File track2 = new File("track2.m3u");
        File track3 = new File("Wind of change.mp3");
        File track4 = new File("Big city night.mp3");
        File track5 = new File("Rainbow in the dark.mp3");
        music.add(track1);
        music.add(scorpions);
        music.add(track2);
        scorpions.add(track3);
        scorpions.add(track4);
        scorpions.add(dio);
        dio.add(track5);
        music.ls();
    }
}
```

```
// Define a "lowest common denominator"
interface AbstractFile {
    void ls();
}
```

```
// File implements the "lowest common denominator"
class File implements AbstractFile {
    private String name;

    public File(String name) {
        this.name = name;
    }

    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name);
    }
}
```

```
// Directory implements the "lowest common denominator"
class Directory implements AbstractFile {
    private String name;
    private ArrayList includedFiles = new ArrayList();

    public Directory(String name) {
        this.name = name;
    }

    public void add(Object obj) {
        includedFiles.add(obj);
    }

    public void ls() {
        System.out.println(CompositeDemo.compositeBuilder + name);
        CompositeDemo.compositeBuilder.append("   ");
        for (Object includedFile : includedFiles) {
            // Leverage the "lowest common denominator"
            AbstractFile obj = (AbstractFile) includedFile;
            obj.ls();
        }
        CompositeDemo.compositeBuilder.setLength(CompositeDemo.compositeBuilder.leng
    }
}
```

Decorator

Decorator Hangi Durumda Kullanmalıyız ?

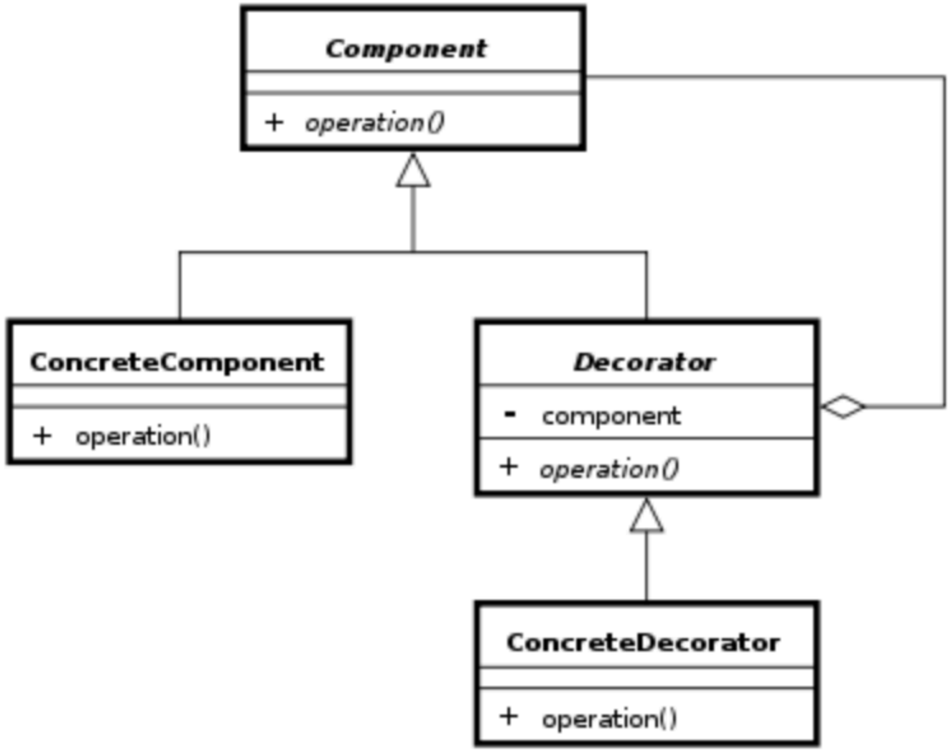
Not* Decorator pattern uygulamalarda sıkça kullanmak zorunda kaldığım örüntülerden birisidir.

Yapınızı oluşturursunuz çalışan bir yapınız var ve bir arayüzden türemiş bir çok nesne bulunuyor. Bu arayüz mekaniklerini kullanan bir çok istemci var. Siz hiç bir arayüz metodu değiştirmeden çağrılan metodun içerisinde ekstra işlemler yapmak durumunda iseniz bu örüntüyü kullanabilirsiniz.

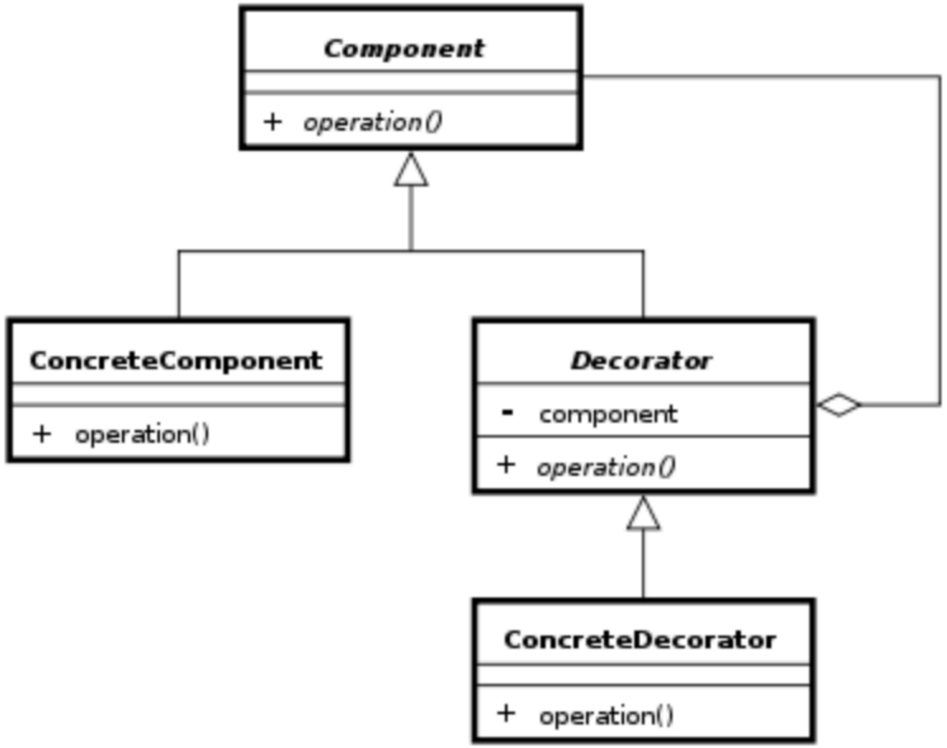
Decorator

Decorator örüntü mevcut tasarımınızda herhangi bir sınıfınızın
yaptığı **iş yeterli olmadığı** ve sınıfınıza
yeni fonksiyonaliteler eklemeniz gerektiği durumda
aynı arayüzü koruyarak ve
mevcut sınıfları ve
algoritmaları değiştirmeden bu eklemeyi nasıl yapabilirsiniz.

Decorator Yapısı



Decorator Yapısı

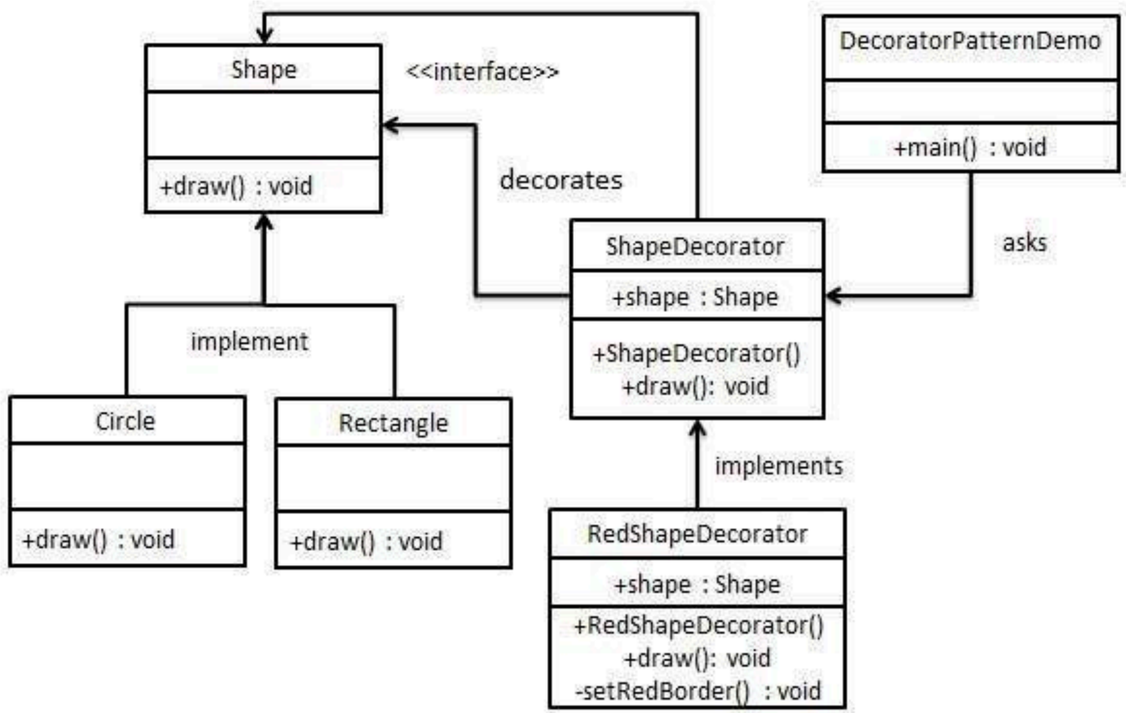


Decorator Örnek

Aşağıda örnekte görüldüğü gibi Shape arayüzünün görevi ilgili *draw()* fonksiyonunu çağırarak çiziminin sağlanmasıdır. Biz buna RedBorder ekstre özelliği kazandırmak ister isek. Önümüzde 2 seçenek bulunmaktadır.

- Shape sınıfını güncellemek
- Shape interface türeyen Decorator oluşturup bu fonksiyonu Decorator içerisinde gerçekleştirmek.

2nci seçeneğin avantajı mevcut fonksiyon ve algoritmalara dokunmadan fonksiyon eklemelerini dışarıdan yapabilmektir. Aşağıdaki kodlarda fonksiyon eklemesini *RedShapeDecorator.draw()* fonksiyonunda yaptığını görmekteyiz. Bu sayede mevcut Circle, Rectangle vb.. Shape türeyen sınıflarımızda bir değişiklik yapmadan kodumuzu değiştirmiş olduk.



Decorator Kod

Demo

```java
public interface Shape {
   void draw();
}
```

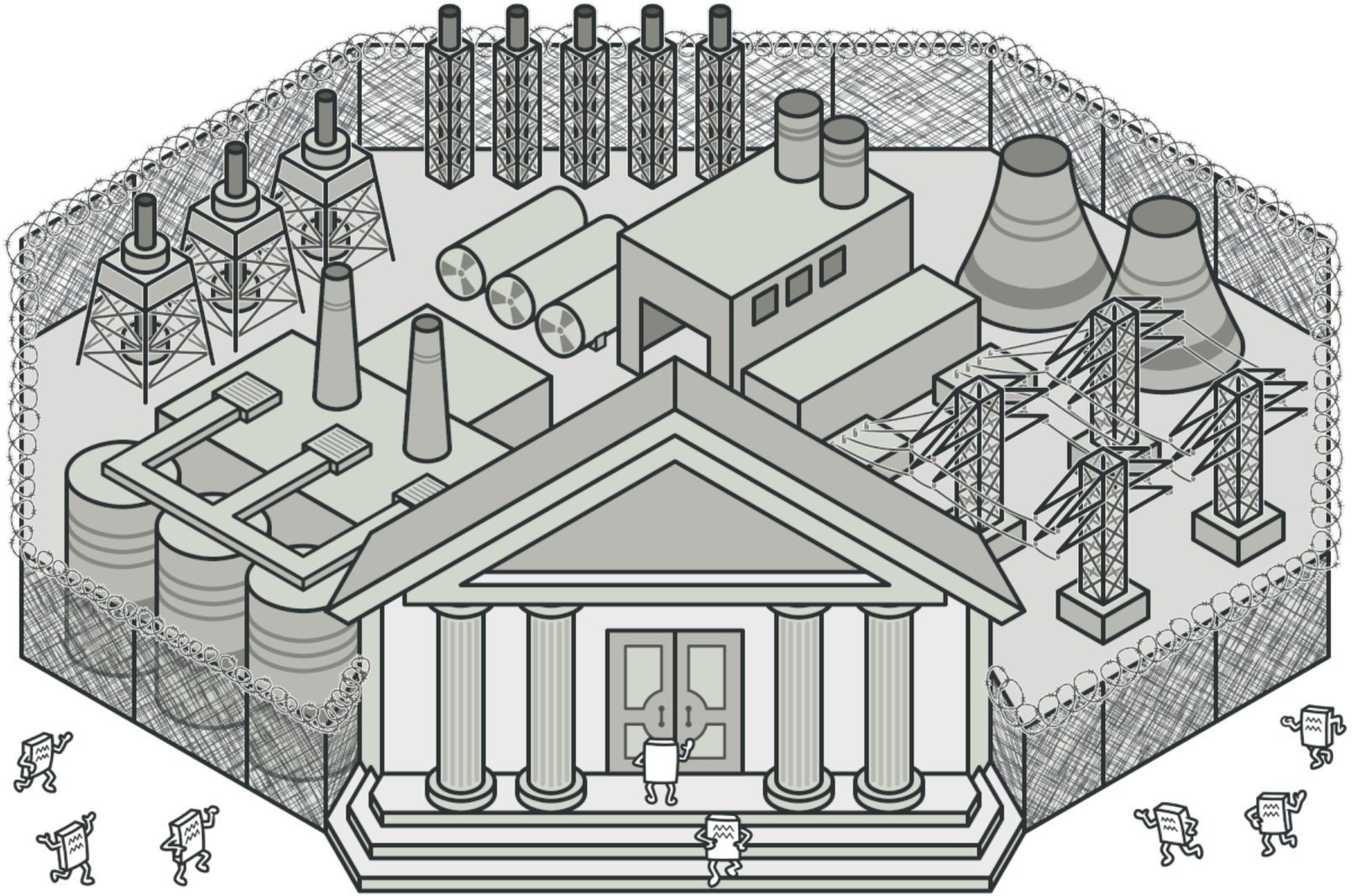
```java
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

```java
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

```java
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

Facade



https://refactoring.guru/

Facade Hangi Durumda Kullanmalıyız ?

Sisteminizde bir çok bileşen bulunuyor. İstemci olarak sizin bu bileşenlerin birçoğunu bilip bir işlem yapabilmek için farklı farklı bileşenlerden farklı sıralarda metodlar çağırmanız gerektiği durumda İstemci sınıfı bu karmaşadan ve bu iş mantıklarından soyutlamak için istemcilerin önüne yazılan bir soyutlama katmanıdır (sınıfıdır)

Facade

Facade örüntü **bir çok bileşenden** oluşup dışarıya bir kütüphane şeklinde hizmet verdiğinde istemcinin **arkaplanda olan karmaşıklığı** bilmesine gerek yoktur. Burda yapılan soyutlamaya Facade örüntüsü denir.

Facade Örnek Kullanımı

Mesela GIS(Geographic Information System). içerisinde bir çok alt sistem mevcuttur, her birisinin birbirleri ile çalışması için bir sürü arayüz ve listener mekanizması mevcuttur.

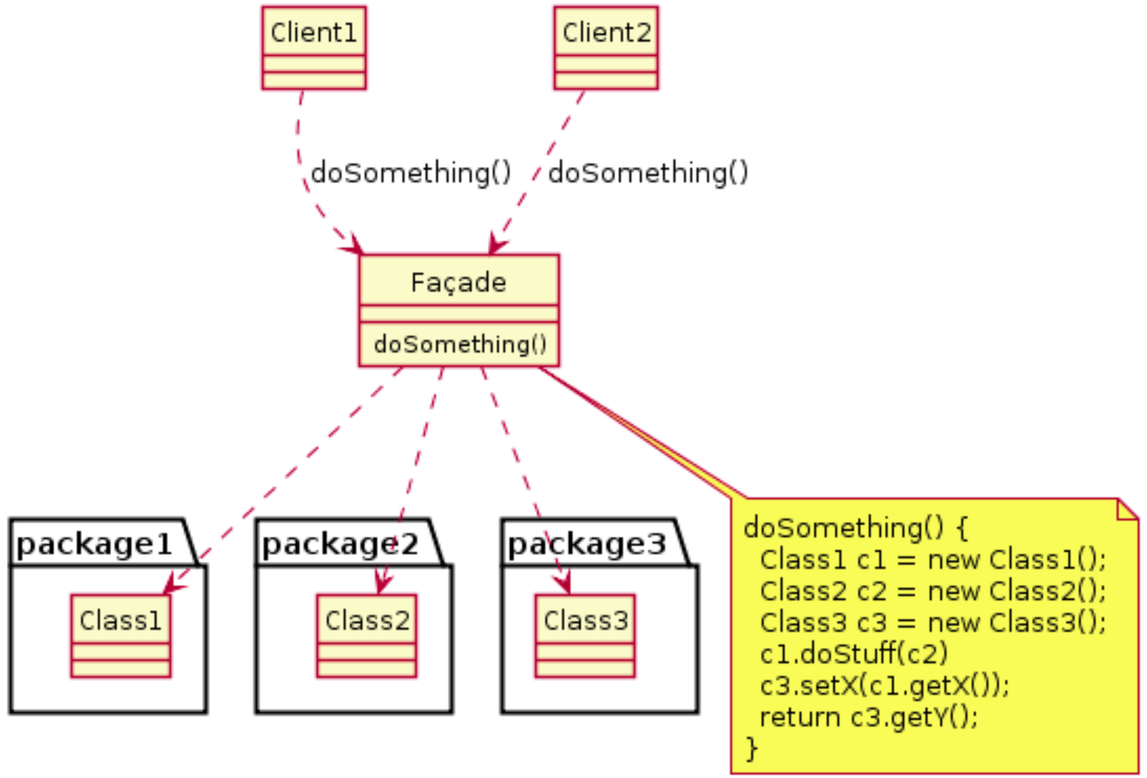
- Harita Altlığı,
- Projeksiyon Mekaniği
- VeriModeli,
- Drag-Drop mekanizması,
- Layer sistemi,
- Icon sistemi,
- vb… alt sistemlerinin olduğunu düşünelim.

Facade Kullanmanın Avantajı

Mevcut durumda istemci bütün bu alt sistemleri, bunların arayüzlerini ve "*listenerları*" öğrenmesi gerekir, Facade Pattern'i sayesinde istemci bundan soyutlanır. GIS uzmanı olan kişi tüm alt sistemleri içerecek bir Facede sınıfı içerisinden dışarıya daha temiz ve basit bir arayüzle fonksiyon erişim imkanı vererek, sanki bir API gibi kullanıcının GIS bilmeden bu sistemi kullanmasını daha rahat bir şekilde sağlamış olur.

Aynı zamanda ilerde bu alt sistemler üzerinde yapacağı değişiklikler, refactoring ve yeni fonksiyon güncellemelerinden son kullanıcıyı etkilemeden yapabilmesini sağlar. Sonuçta Facade ile alt sistemler soyutlanmış olur.

Facade Yapısı



Facade Kod

```
int main() {
  ComputerFacade computer;
  computer.Start();
}
```

```
class ComputerFacade {
 public:
  void Start() {
    cpu_.Freeze();
    memory_.Load(kBootAddress, hard_drive_.Read(kBootSector, kSectorSize));
    cpu_.Jump(kBootAddress);
    cpu_.Execute();
  }

 private:
  CPU cpu_;
  Memory memory_;
  HardDrive hard_drive_;
};
```

```
struct CPU {
  void Freeze();
  void Jump(long position);
  void Execute();
};
```



```
struct HardDrive {
  char* Read(long lba, int size);
};
```

```
struct Memory {
  void Load(long position, char* data);
};
```

Flyweight

Flyweight Hangi Durumda Kullanmalıyız ?

Sınıfın içerisinde hep aynı değerleri alan değişkenleri bir veri yapısı haline getirip ortak(shared) olarak cache'den kullanmasını sağlatan bir örüntüdür.

Üretilecek sınıf sayısı bir veya iki tane oldugunda bu örüntüyü kullanmak anlamlı degildir ancak milyonlarca aynı sınıftan oluşturuyorsak bellek önemli olmaya başlar ve bu durumu performanslı çalıştırabilmek için bu örüntü kullanılmalıdır.

Flyweight

Flyweight örüntü nesneden **binlerce, milyonlarca** olduğu durumda **bellekte kapladığı alanı azaltmak** kullanılan bir örüntüdür.

Flyweight Örnek Kullanım

Oyunlardaki parçacık sistemleri veya ASD'nin Metamorphosis Scene Demosunu izlediğimizde bir sürü karınca ve böceğin aynı anda hareket ettiğini görebiliriz. Bu durumda tüm nesnenin tüm alanlarını bellekte tutmak ciddi bir performans kaybına neden olacaktır.

Aşağıdaki alanlardan ortak alanları ***shared bir yapıda*** tutarak bellek kullanımını azaltabiliriz.

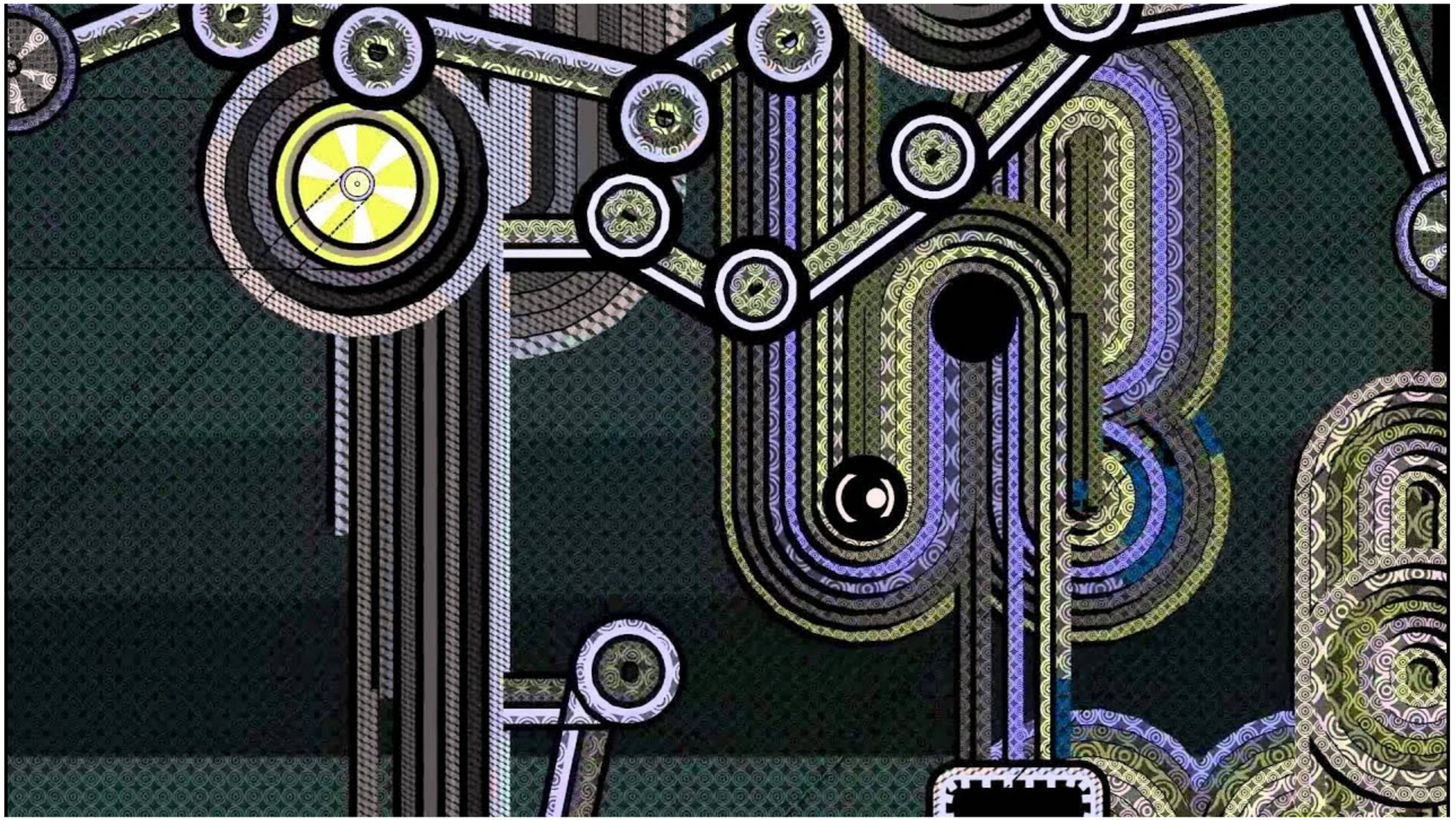
Ortak Alanlar:

- boyut
- renk
- hız

Ortak Olmayan Alanlar

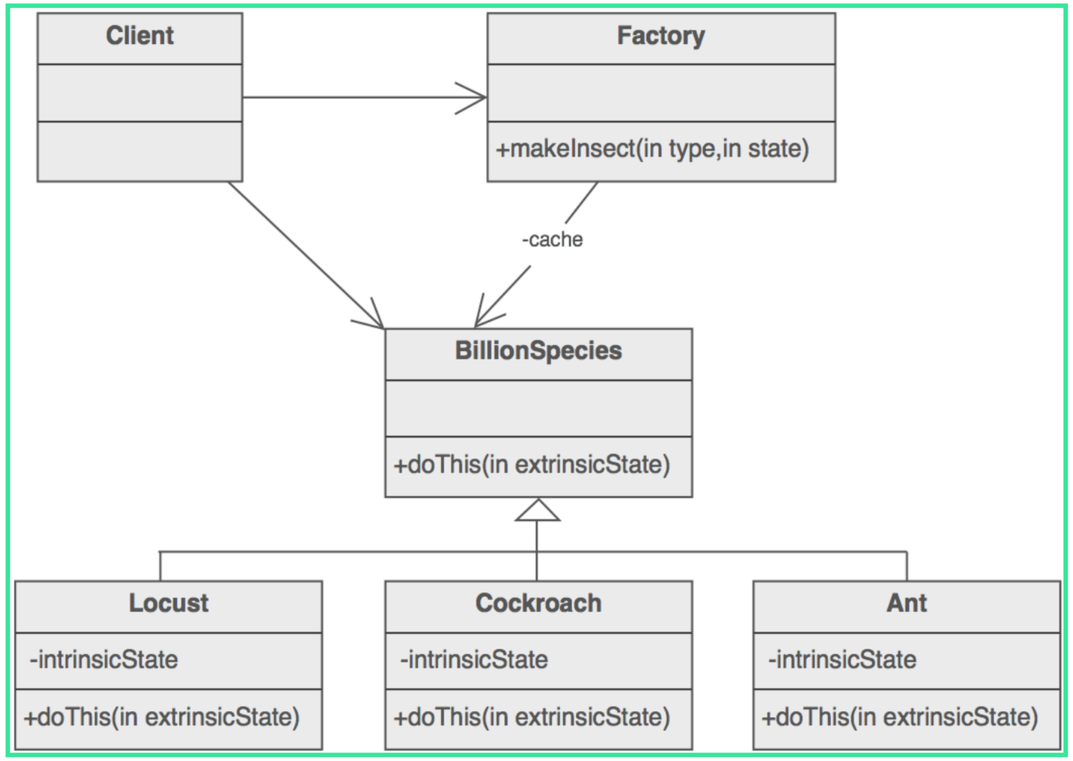
- konum

Flyweight Örnek Kullanım

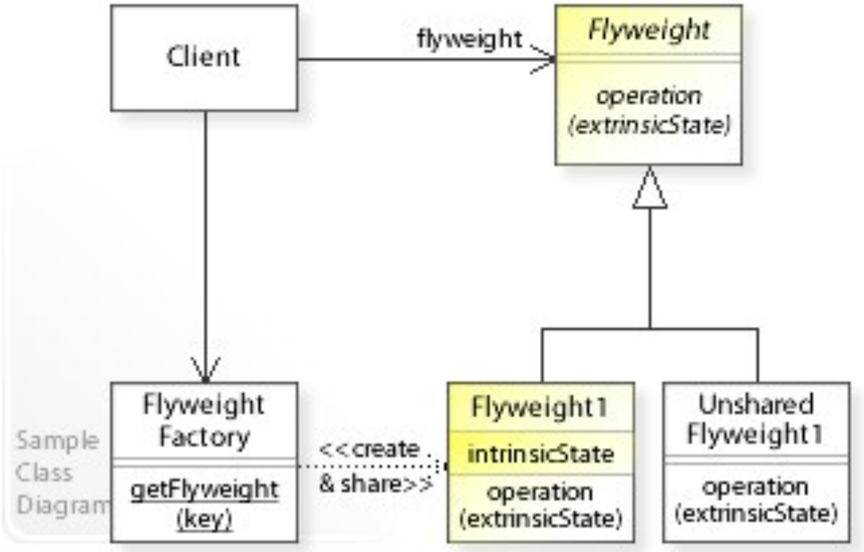


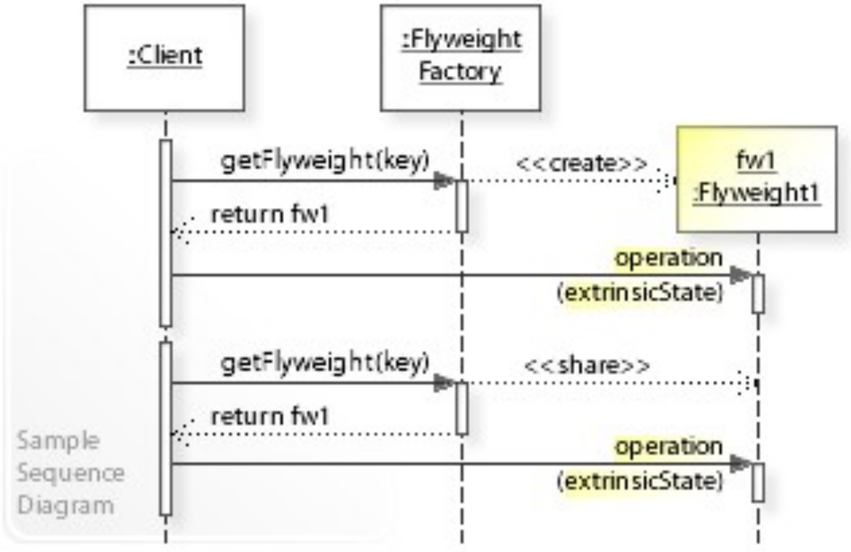
Flyweight Örnek Kullanım

Buradaki bir sürü karınca ve böceği nasıl oluşturulduğunu görebilirsiniz. Factory makeInsect dediğimiz zaman örneğin karınca oluşturulurken konum alanı dışındaki tüm özellikler ***HashMap<String, Insect>*** shared alanından çekilerek kullanılabilir. Bu durumda sizin bellekten kullanacağınız alan sadece konum için olan float x ve y alanı olacaktır.

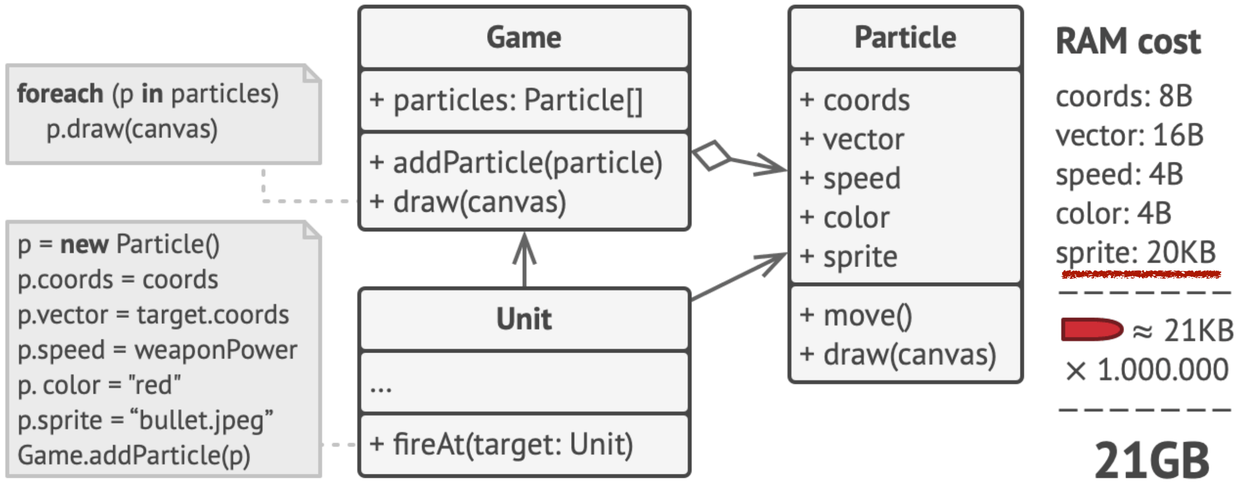


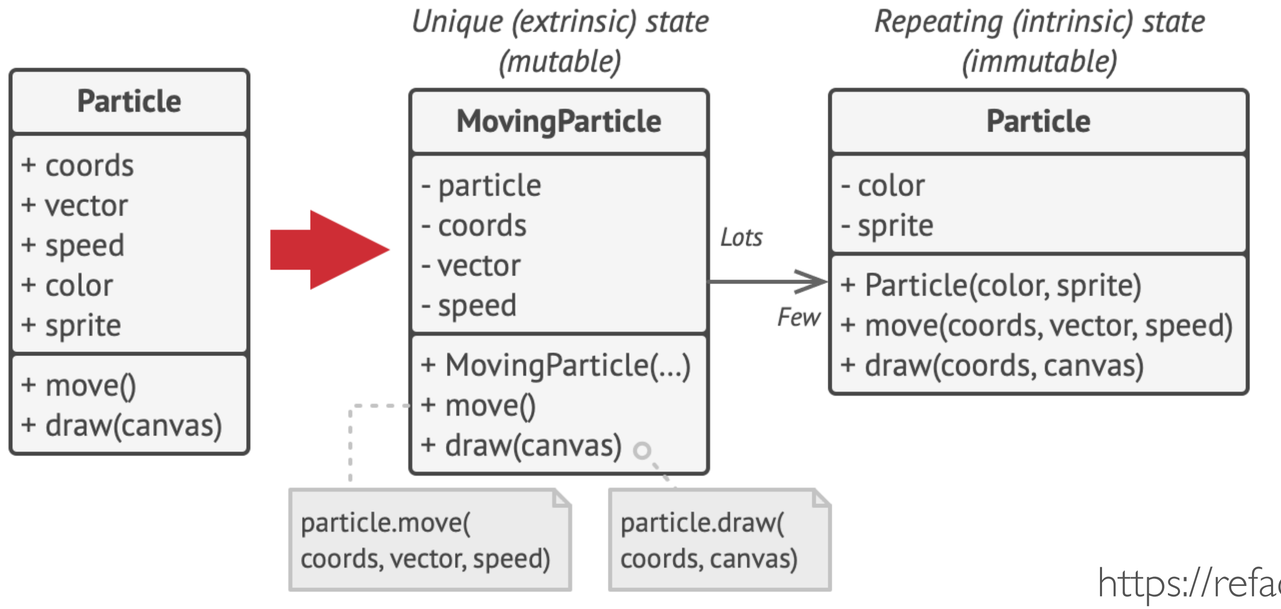
Flyweight Yapısı





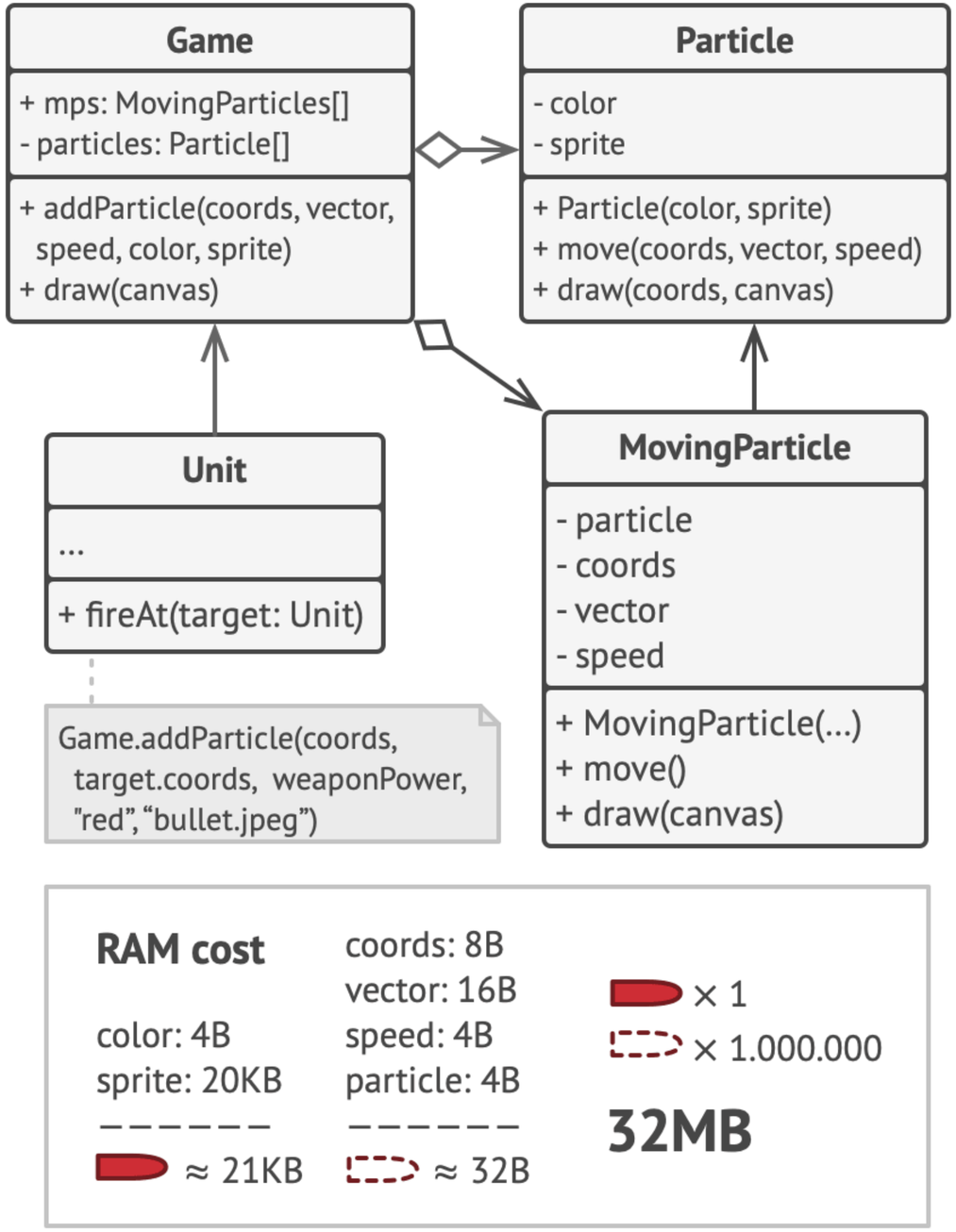
Flyweight Yapısı





https://refactoring.guru/

Flyweight Yapısı



Proxy

Proxy Hangi Durumda Kullanmalıyız ?

Gerçek nesneye erişimin zor olduğu durumlarda Örneğin ;

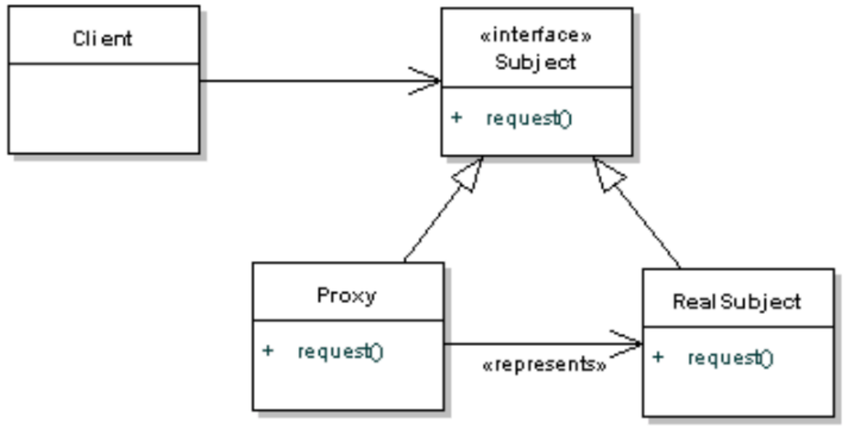
- İlgili nesnenin/servisin uygulamanın ilk başından oluşturulması ve tüm uygulama boyunca ayakta tutulması sisteme büyük yük getirebilecek durumlar olabilir. Sistem olarak bu büyük servise sadece ihtiyacınız olduğunda bağlanmanız gerekiyorsa bu tip durumlarda **Virtual Proxy (Lazy Initialization)** kullanılabilir
- İlgili nesne önemli kritik noktalara erişiyor ise işletim sistemi ve yetklili servisler, ve istemci sınıfın bu yetkileri yok ise arada bulunan vekil sınıfta bu tip kontroller yapılabilir **Protection proxy**
- Kullanacağımız gerçek nesne aynı makinede çalışmıyor Local makineden Remote makineye bir network call yapılarak nesneye ulaşıyorsanız. Arada oluşan veya öncesinde oluşacak problemli durumları istemci nesneye yansıtmamak için **Remote proxy** kullanılabilir. (Java RMI, Jini, JAX-RPC)
- İstemcinin gerçek nesne üzerinden çağırdığı isteklerin log tarihçesi için ***Log Proxy*** olarak kullanabilirsiniz bu sırada aynı parametreler için aynı sonuçları dönüyorsa bunlar için ***Cache Proxy*** olarak kullanabilirsin
- Yüklü gerçek nesne üzerinde nesnelerin referansını tutup, bu gerçek nesneleri cache'leyip. İlgili proxy kullanan istemci kalmadığı durumda gerçek nesneleri otomatik olarak kaldırılmasını sağlayan ***Smart Reference Proxy*** oluşturabilirsiniz.

Proxy

Proxy örüntüsü **gerçek nesneye erişimin zor** olduğu
veya **zaman aldığı durumlarda** aynı interface implement eden
Proxy'sine (Vekil'ine) erişerek işlemlerinizi proxy nesnesi üzerinde yapmanızı sağlar.

Proxy Yapısı

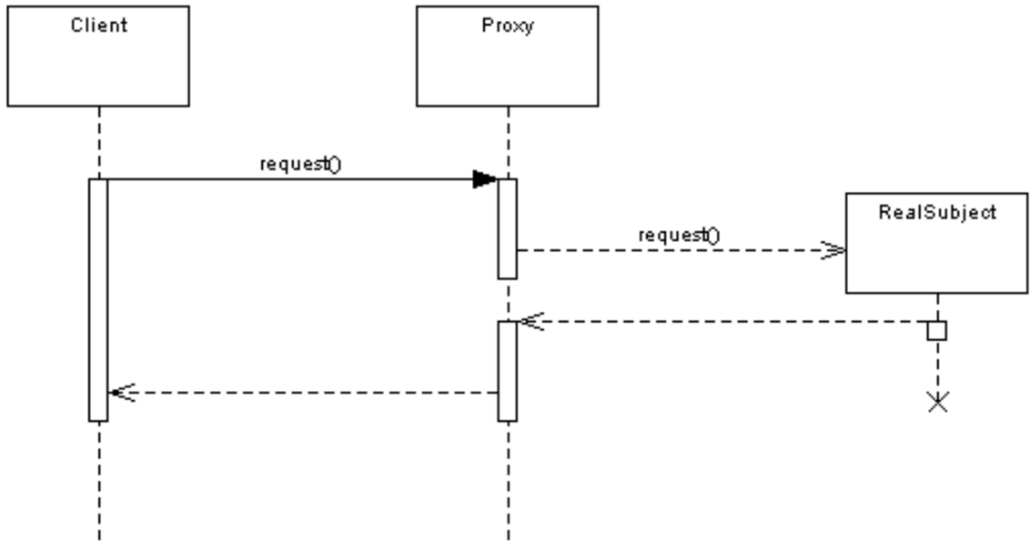
Burda gerçek nesnemizin aynı servislerini verebilmek için Proxy ve Gerçek nesnemiz aynı arayüz'den Subject türemesi gerekiyor.



Proxy Akışı

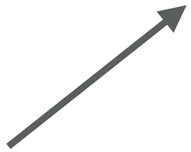
İstemci arayüz üzerinden Proxy sınıf ile iletişime geçiyor. Proxy arkaplanda gerçek obje ile iletişime geçerek bir takım gerçek objede olan sorunları, problemleri istemciye yansıtmadan çözmeye çalışıyor.

Örneğin İstemci bir resim dosyası istiyor, resim yükleninceye kadar geçici bir resim yükleniyor resmi dönebilir vb..



Proxy Örnek Kod

```java
// 5. To support plug-compatibility between
// the wrapper and the target, create an interface
interface SocketInterface {
    String readLine();
    void  writeLine(String str);
    void  dispose();
}
```

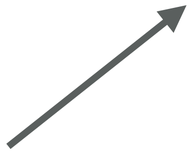


```java
public class ProxyDemo {
    public static void main( String[] args ) {
        // 3. The client deals with the wrapper
        SocketInterface socket = new SocketProxy( "127.0.0.1", 8080, args[0].eq
        String  str;
        boolean skip = true;
        while (true) {
            if (args[0].equals("second") && skip) {
                skip = !skip;
            } else {
                str = socket.readLine();
                System.out.println("Receive - " + str);
                if (str.equals(null)) {
                    break;
                }
            }
            System.out.print( "Send ---- " );
            str = new Scanner(System.in).nextLine();
            socket.writeLine( str );
            if (str.equals("quit")) {
                break;
            }
        }
        socket.dispose();
    }
}
```

```java
class SocketProxy implements SocketInterface {
    // 1. Create a "wrapper" for a remote,
    // or expensive, or sensitive target
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public SocketProxy(String host, int port, boolean wait) {
        try {
            if (wait) {
                // 2. Encapsulate the complexity/overhead of the target in the wrapper
                ServerSocket server = new ServerSocket(port);
                socket = server.accept();
            } else {
                socket = new Socket(host, port);
            }
            in  = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public String readLine() {
        String str = null;
        try {
            str = in.readLine();
        } catch( IOException e ) {
            e.printStackTrace();
        }
        return str;
    }

    public void writeLine(String str) {
        // 4. The wrapper delegates to the target
        out.println(str);
    }

    public void dispose() {
        try {
            socket.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Proxy Örnek Kod

```java
// 5. To support plug-compatibility between
// the wrapper and the target, create an interface
interface SocketInterface {
    String readLine();
    void  writeLine(String str);
    void  dispose();
}
```



```java
public class ProxyDemo {
    public static void main( String[] args ) {
        // 3. The client deals with the wrapper
        SocketInterface socket = new SocketProxy( "127.0.0.1", 8080, args[0].eq
        String  str;
        boolean skip = true;
        while (true) {
            if (args[0].equals("second") && skip) {
                skip = !skip;
            } else {
                str = socket.readLine();
                System.out.println("Receive - " + str);
                if (str.equals(null)) {
                    break;
                }
            }
            System.out.print( "Send ---- " );
            str = new Scanner(System.in).nextLine();
            socket.writeLine( str );
            if (str.equals("quit")) {
                break;
            }
        }
        socket.dispose();
    }
}
```

```java
class SocketProxy implements SocketInterface {
    // 1. Create a "wrapper" for a remote,
    // or expensive, or sensitive target
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;

    public SocketProxy(String host, int port, boolean wait) {
        try {
            if (wait) {
                // 2. Encapsulate the complexity/overhead of the target in the wrapper
                ServerSocket server = new ServerSocket(port);
                socket = server.accept();
            } else {
                socket = new Socket(host, port);
            }
            in  = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public String readLine() {
        String str = null;
        try {
            str = in.readLine();
        } catch( IOException e ) {
            e.printStackTrace();
        }
        return str;
    }

    public void writeLine(String str) {
        // 4. The wrapper delegates to the target
        out.println(str);
    }

    public void dispose() {
        try {
            socket.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Proxy vs Facade vs Decorator

Geliştiricilerin kafasında şu şekilde sorular takılabilir ?

- **Aynı mantıkla çalışan Proxy ve Facade ne gibi farklılıkları olduğu ?**
  Proxy bir tane gerçek nesneye veya servise erişirken oluşacak problemleri giderirken, Facade daha çok servisin olduğu ve bu servislerden farklı farklı method çağırımlarını soyutlayarak istemcinin işlerini kolaylaştırdığını görebilirsiniz
- **Aynı yapıda olan Proxy ve Decorator arasında ne gibi farklılıklar olduğu ?**
  Proxy gerçek nesnenin arayüzündeki servisler/methodlar ile sınırlıdır. Ama decorator mevcut nesneye yeni özellikler/fonksiyonlar kazandırmayı amaçlar.

Structural Patterns Özet

Adapter: Arayüz uyumsuz sınıfı ilgili yapıyı destekleyecek halde adapte eder.

Bridge: 2 farklı sınıf grubunu(Servis & İstemci) gerçekleştirimlerini soyutlamayı sağlar.

Composite: Ağaç yapısında bir veriyapısını sahip ve hepsinde ortak fonksiyon çalıştırmak istiyorsanız

Decorator: Mevcutta kullandığınız nesnenin davranışını runtime dinamik olarak genişletebilmek için

Facade: Kompleks bileşenlerin basit bir arayüz ile istemciye servis sunar hale getirilmesi

Flyweight: Kompleks nesne modelleri (fazla bellek alanı) çok nesne ihtiyacı olduğu durumlarda performans sağlar

Proxy: Vekil (temsilci) oluşturarak , istemcinin bu temsilci üzerinden haberleşmesi ve yetki kontrolü, maliyet …

Behavioral Patterns

2nci Gün (Code Quality)

- Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~
- Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~
- Kod Geliştirme Yöntemleri → ~~AOP~~ ~~TDD~~ ~~SOLID~~ ~~KISS~~ ~~DRY~~ ~~YAGNI~~
- Kodu Dayanıklı Geliştirmek → ~~12 Factor App~~ ~~Reactive Programming~~ ~~Chaos Eng~~
- Design Patterns → ~~Design Patterns~~ ~~Creational Patterns~~ ~~Structural Patterns~~ Behavioral Patterns
- Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

Behavioral Patterns

Davranışsal(Behavioral) örüntüler daha çok **algoritmaların davranışların soyutlanması** üzerine olan örüntülerdir.
Algoritmalar arası **iletişim** şekli**,** sorumlukların **paylaşımı** veya **aktarımının** nasıl yapılacağını tanımlar.

Chain Of Responsibility

COR Hangi Durumda Kullanmalıyız ?

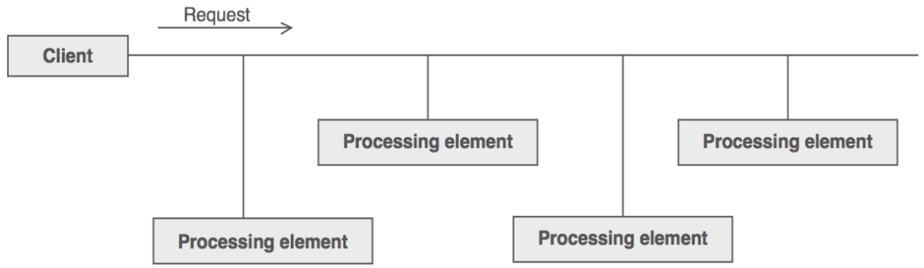
İstemcinin isteği var ve bu isteğin hedefe ulaşıncaya kadar üzerinde bir takım işlemler yapılması gerekiyor , bir takım processing sınıflar tarafından bu isteğin işlenmesi ve bir sonraki işlemciye geçirilmesi gerektiği durumlarda kullanılmalıdır.

Servlet Örneği

Servlet çağrımında Filtreler bu duruma güzel bir örnek

- Authentication Filter (Kimlik Doğrulama)
- Logging Filter (Loglama)
- Data Compression Filter (Veri Sıkıştırma)
- Encryption Filter

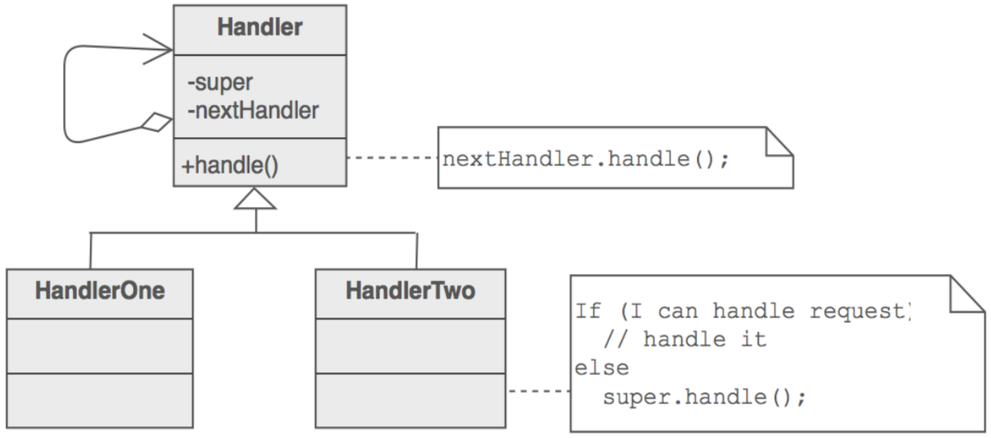
Bu örüntüde processing elementler sadece kendilerinden bir sonraki processing element bilirler ve kendileri ile ilgili işlemi gerçekleştirdikten sonra bir sonraki processing element ilgili request nesnesini geçirirler.



Servlet gelen Request ilerleyişi

try/catch Örneği

java'daki Exception Handling mekanizmasıda bir Chain Of Responsibility örneğidir. try/catch bloğunda işlenemeyen bir exception kendisinden bir önceki çağrım yapan fonksiyona o da kendisinden bir önceki çağrım yapan fonksiyona Exception throw edebilir. Bu sayede Exception ele alma sorumluluğu olan fonksiyonda işlenmiş olur. Aşağıdaki gösterimde ***Handler*** kendisinden sonra gelen Handler tutuyor, kendisi Exception handle edemez ise bir sonraki bir üst fonksiyona hatayı geçiriyor.

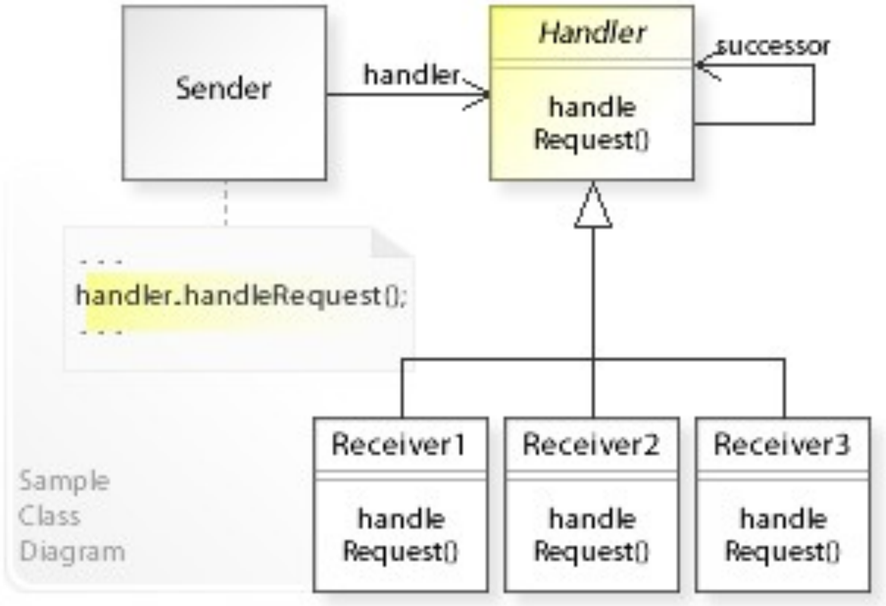


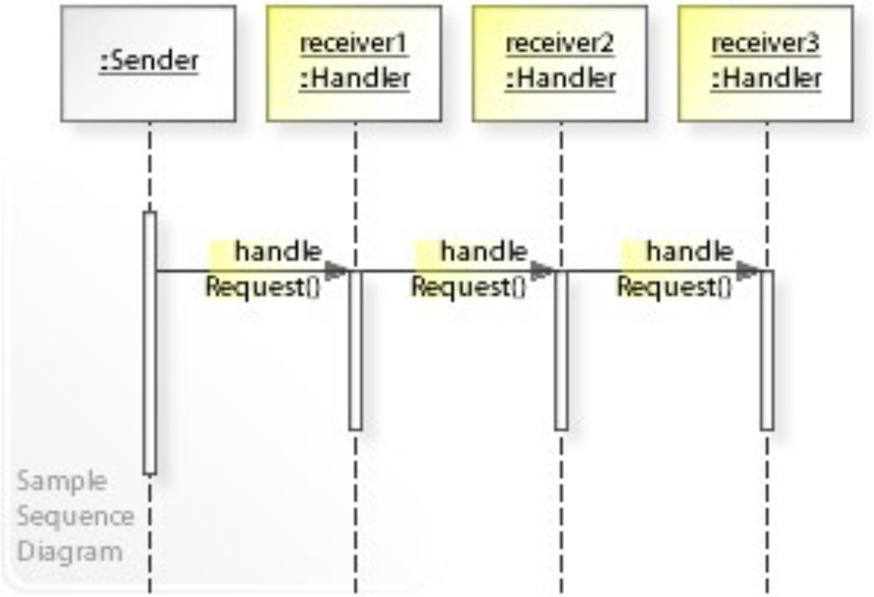
COR Yapısı

Chain Of Responsibility

Chain of Responsibility, istemcinin bir **isteği** var
ve bu isteğin sıra ile **işlemci sınıflar** tarafından **işlenmesi** ve
bir **sonraki** işlemciye geçirilmesi gerekiyor.

Chain of Responsibility Yapısı





Command

Command Hangi Durumda Kullanmalıyız ?

Command örüntüsü denilince UI Uygulamaları aklıma geliyor. Örneğin Word, Photoshop, Maya. Çünkü tüm bu araçlarda yaptığınız işleri geri alma **(Undo)** veya yaptığınız işleri gruplayarak tekrar çalıştırma **(Macro)** ihtiyacı vardır. Bu tarz bir şey yapabilmek

Aynı şekilde elimizdeki bu komutları bir kuyruğa yazıp işletebilir, işletme sırasında durdurup , devam ettirebiliriz. Bu sayede remote çağrılarıda gerçekleştirebiliriz.

Command

Command örüntü istemcinin hedef nesne üzerinde yaptığı işlemleri
bir komut objesi haline gelip işleterek bu komutların saklanması**(macro)**,
komutların geri alınması**(undo)** stateful halde yönetilmesini sağlar.

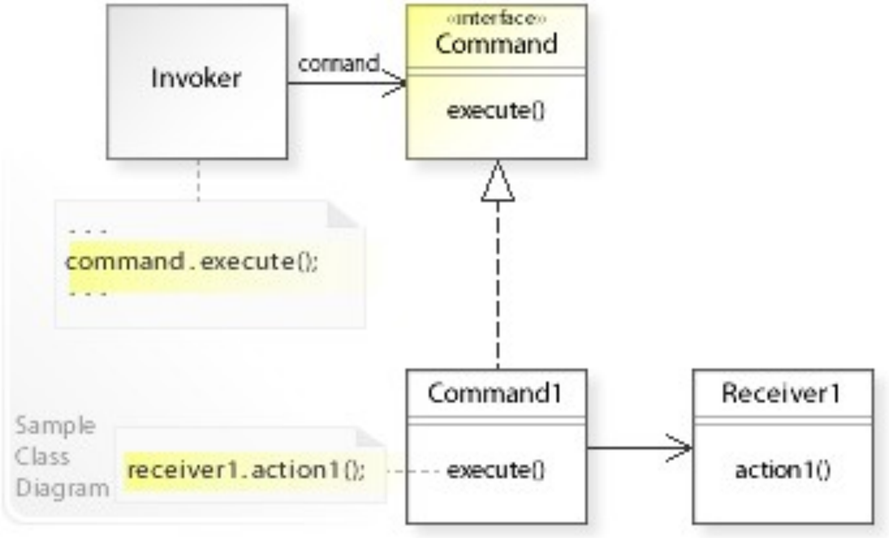
Command Yapısı

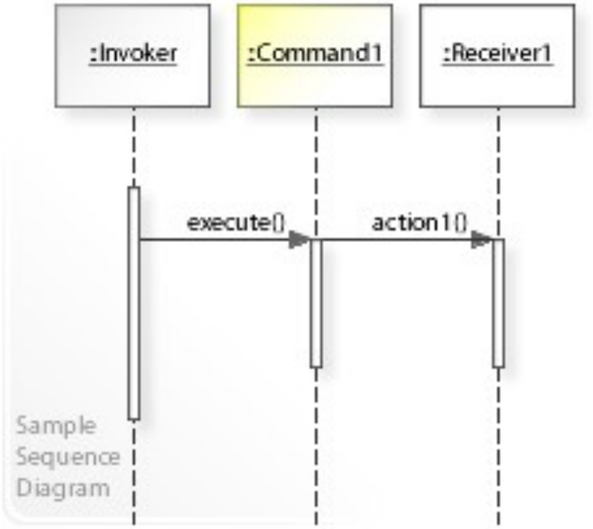
Invoker: Çalıştıracağı Command arayüzünden komutları execute eder.

Command:içerisinde execute vb komut olan arayüz (interface)

Command1, Command2 .. : Receiver üzerinde işlem yapabilecek istemci için anlamlı komutlardır.

Receiver: Servis sağlayan asıl sınıftır.





Command Kod Örneği

```
class CommandManager
{
    private Stack commandStack = new Stack();

    public void ExecuteCommand(Command cmd)
    {
        cmd.Execute();
        if (cmd is UndoableCommand)
        {
            commandStack.Push(cmd);
        }
    }


    public void Undo()
    {
        if (commandStack.Count > 0)
        {
            UndoableCommand cmd = (UndoableCommand)commandStack.Pop();
            cmd.Undo();
        }
    }
}
```

Command Manager sınıfı çalıştırılmak istenen komutları Stack veriyapısında tutuyorki , **undo** işlemi sırasında en son yapılandan eskiye doğru yapılan işlemleri geriye alabilsin.

Command yapımızı tanımlıyoruz. Command örüntüsü için en önemli konu **execute()** fonksiyonu, yani işlemi gerçekleştirildiği fonksiyon. Birde bu işlemi geriye alabilmek için **undo()** fonksiyonunu tanımlıyoruz.

```
public abstract class Command
{
    public abstract void Execute();
}

public abstract class UndoableCommand : Command
{
    public abstract void Undo();
}
```

```
class BoldCommand : UndoableCommand
{
    private Document document;
    private string previousText;
    public BoldCommand(Document doc)
    {
        this.document = doc;
        previousText = this.document.Text;
    }

    public override void Execute()
    {
        document.BoldSelection();
    }

    public override void Undo()
    {
        document.Text = previousText;
    }
}
```

Birçok **UndoableCommand** sınıfımız olabilir.

- BoldCommand,
- ItalicCommand,
- UnderscoreCommand,
- CopyCommand gibi.

Bu Command sınıfı dokümanın değiştirilmeden önceki state tutuyorki **undo()** işlemi sonrasında yaptığı işlemi geri alabilsin. **execute()** fonksiyonunda Dokuman nesnesi üzerinden ilgili değişikliği gerçekleştiriyor.

Document: TextBox içerisindeki metni değiştirerek ***Bold, Italic, Underline*** haline getiriyor.

```
class Document
{
    private TextBox textbox;
    public Document(TextBox textbox)
    {
        this.textbox = textbox;
    }

    public void BoldSelection()
    {
        Text = String.Format("<b>{0}</b>", Text);
    }

    public void UnderlineSelection()
    {
        Text = String.Format("<u>{0}</u>", Text);
    }

    public void ItalicizeSelection()
    {
        Text = String.Format("<i>{0}</i>", Text);
    }

    public void Cut()
    {
        textbox.Cut();
    }

    public void Copy()
    {
        textbox.Copy();
    }

    public void Paste()
    {
        textbox.Paste();
    }

    public string Text
    {
        get { return textbox.Text; }
        set { textbox.Text = value; }
    }
}
```

Receiver Class

Interpreter

Interpreter Hangi Durumda Kullanmalıyız ?

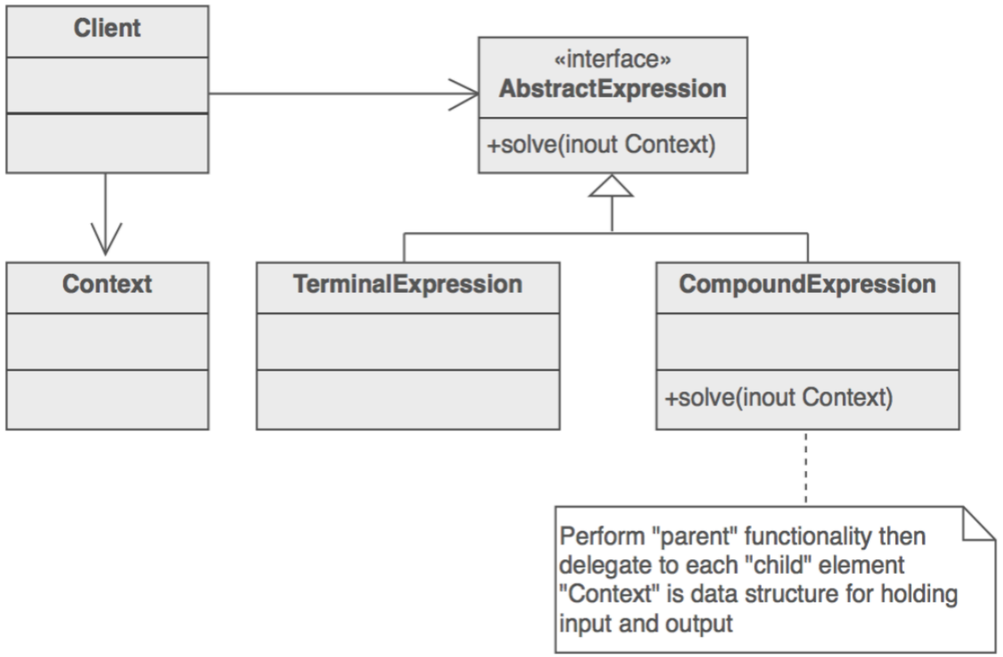
Eğer elinizde belli gramer uygun içerik var ise. Örneğin türkçe bir metin, bir şarkının notaları, herhangi bir programlama dili ile kodlanmış bir uygulama, SQL veya kendi oluşturduğunu siz bu kurallara göre bu içeriği analiz ederek hiyerarşik ilişkili nesneler(Domain Modelini) oluşturmak istiyorsanız bu örüntü sizin için uygun bir örüntüdür.

Interpreter

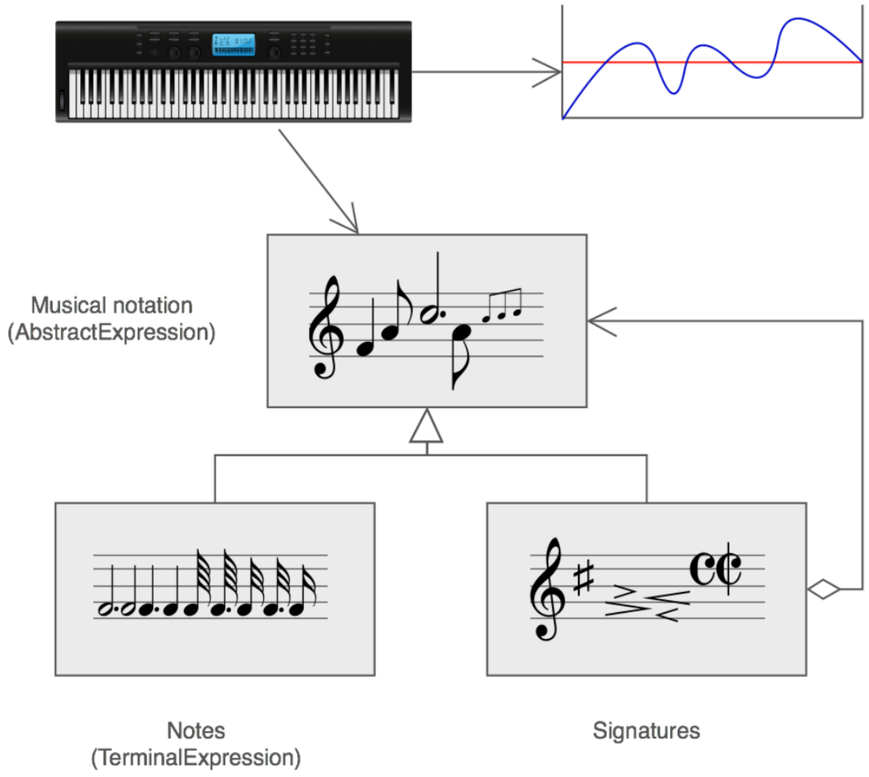
Interpreter örüntü istemcinin String formatında ama belli bir **grammer** göre yazılmış komutları **parçalara ayırarak** işletilmesini sağlayan örüntüdür.

Interpreter Örnek ?

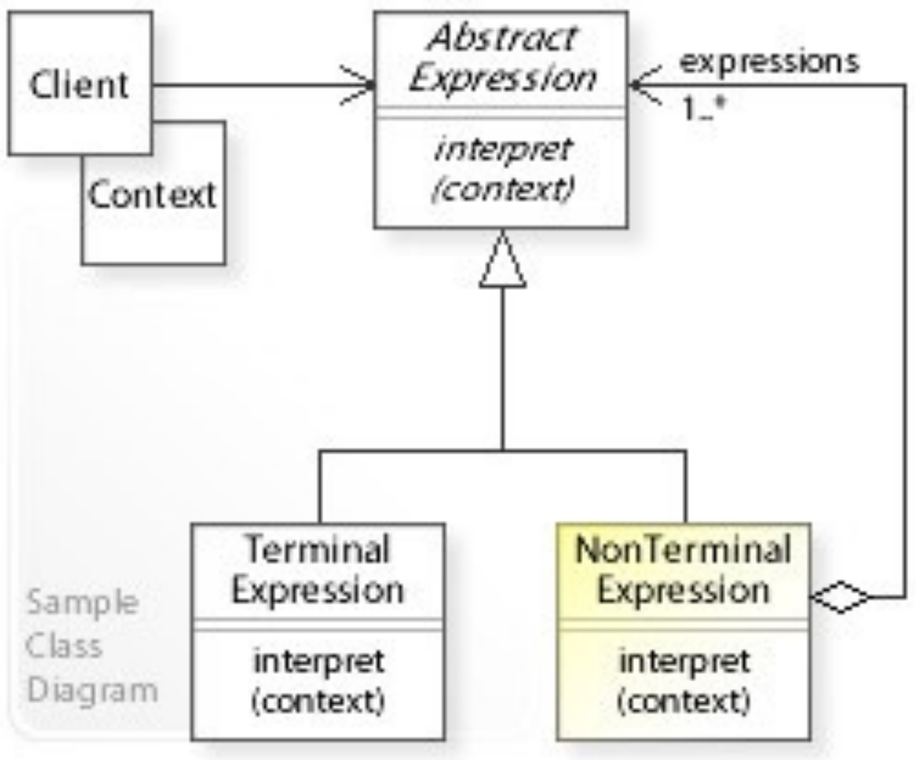
Aşağıdaki UML şemasında belirli bir Context var. Client bu Context'i işleyebilmek için bir sürü Expression(İfade yorumlayıcı) tutuyor. Context'i sırası ile bu yorumlayıcılarda işler ve domain nesnelerine dönüştürür.

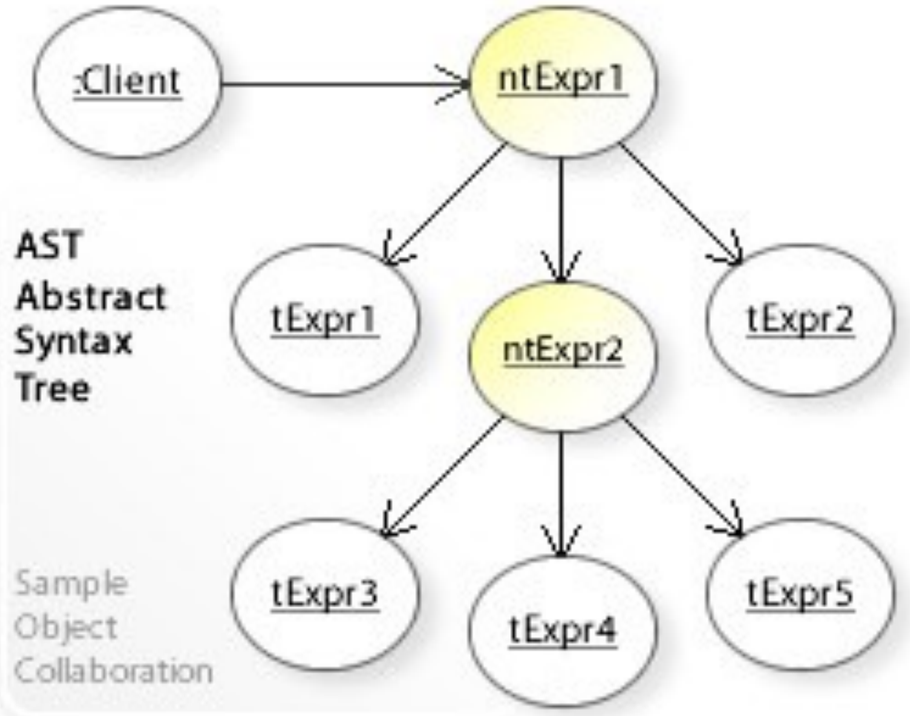


Aşağıda çalınan bir parçanın müziksel gösterimlerinin analiz edilerek ayrıştırıldığına. Notalar ve Müzik işaretlerine dönüştürüldüğünü görebilirsiniz.



Interpreter Yapısı





Interpreter Kod

```java
public class ExpressionParserTest {

    @Test
    public void testParse() throws Exception {
      String input="2 1 5 + *";
       ExpressionParser expressionParser=new ExpressionParser();
       int result=expressionParser.parse(input);
       System.out.println("Final result: "+result);

    }
}
```

```java
package guru.springframework.gof.interpreter;


public interface Expression {
    int interpret();
}
```

```java
public class MultiplicationExpression implements Expression{
```

```java
public class SubtractionExpression implements Expression{
```

```java
public class AdditionExpression implements Expression {
    private Expression firstExpression,secondExpression;
    public AdditionExpression(Expression firstExpression, Expression secondExpression){
        this.firstExpression=firstExpression;
        this.secondExpression=secondExpression;
    }
    @Override
    public int interpret(){
        return this.firstExpression.interpret()+this.secondExpression.interpret();
    }
    @Override
    public String toString(){
        return "+";
    }
}
```

```java
public class NumberExpression implements Expression{
    private int number;
    public NumberExpression(int number){
        this.number=number;
    }
    public NumberExpression(String number){
        this.number=Integer.parseInt(number);
    }
    @Override
    public int interpret(){
        return this.number;
    }
}
```

```java
public class ExpressionParser {
    Stack stack=new Stack<>();
    public int parse(String str){
        String[] tokenList = str.split(" ");
        for (String symbol : tokenList) {
            if (!ParserUtil.isOperator(symbol)) {
                Expression numberExpression = new NumberExpression(symbol);
                stack.push(numberExpression);
                System.out.println(String.format("Pushed to stack: %d", numberExpression.interpret()));

            } else  if (ParserUtil.isOperator(symbol)) {
                Expression firstExpression = stack.pop();
                Expression secondExpression = stack.pop();
                System.out.println(String.format("Popped operands %d and %d",
                        firstExpression.interpret(), secondExpression.interpret()));
                Expression operator = ParserUtil.getExpressionObject(firstExpression, secondExpression,
symbol);
                System.out.println(String.format("Applying Operator: %s", operator));
                int result = operator.interpret();
                NumberExpression resultExpression = new NumberExpression(result);
                stack.push(resultExpression);
                System.out.println(String.format("Pushed result to stack: %d"
```

```java
public class ParserUtil {
    public static boolean isOperator(String symbol) {
        return (symbol.equals("+") || symbol.equals("-") || symbol.equals("*"));

    }
    public static Expression getExpressionObject(Expression firstExpression,Expression
secondExpression,String symbol){
        if(symbol.equals("+"))
            return new AdditionExpression(firstExpression,secondExpression);
        else if(symbol.equals("-"))
            return new SubtractionExpression(firstExpression,secondExpression);
        else
            return new MultiplicationExpression(firstExpression,secondExpression);
    }
}
```

Mediator

Mediator Hangi Durumda Kullanmalıyız ?

Mediator örüntü birçok bileşenin birbirinden servis kullandığı ve bu servis kullanımlarından dolayı bağımlılıkların çok arttığı durumlarda bu bağımlılıkları merkeze taşıyarak bağımlılık kaosunu çözen bir yaklaşımdır.

Mediator örüntüsü gerçek hayattan alınmış bir örüntüdür. Havadaki uçakların rotaları ve birbirleri ile konuşması yerine, yer istasyonları ile iletişim kurarak belli bir merkezlerden yönetilmeleri sağlanır.

Bunun sayesinde havadaki binlerce uçağın birbirleri ile konuşmaları sonucu oluşturacakları karmaşıklığın önüne geçilmiş olur. Problem merkezileştirilerek çözülür.

Mediator

Mediator örüntü **birçok bileşenin** birbirinden servis kullandığı
ve bu servis kullanımlarından dolayı **bağımlılıkların çok arttığı** durumlarda
bu bağımlılık kaosunu çözecek merkezi bir sistemdir.

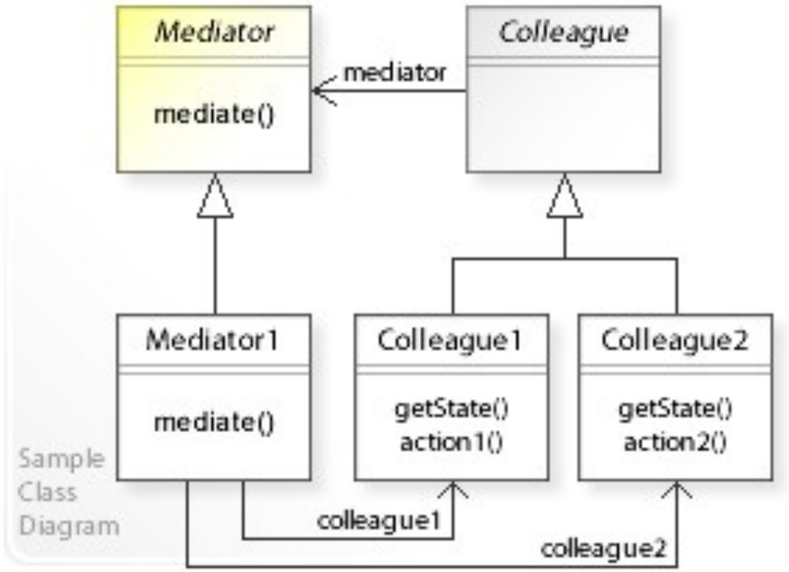
Mediator Örnek Kullanımı ?

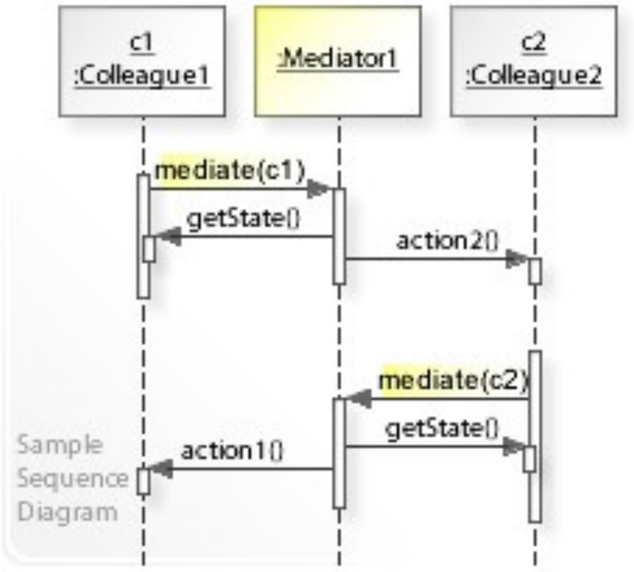
Yazılımda da benzer durumlar ile zaman zaman karşılaşırız. Örneğin önceden çalıştığım bir projede benzer bir problem ile karşı karşıya gelmiştik. Arayüz uygulamamızın bir çok bileşeni ve bu bileşenlerin birbirleri ile konuşması gerekiyordu.

- Menuler,
- Toolbar
- EH nesnelerini tutan modeller
- Harita Katmanı
- 2 Boyutlu Radar Scope'lar
- 3D Visualization Ekranları
- Model parametrelerinin gösterildiği ekranlar
- vb.. bir sürü bileşen olduğunu düşünün.

Uygulamanın merkezine bir ScreenMediator nesnesi oluşturarak tüm nesnelerin birbirinden habersiz hayatlarına devam etmesini sağlamıştık. Bu bir yaklaşım da bileşenlerin oldukça sade ve basit geliştirilmesini sağlarken tüm iletişim ve koordinasyon yoğunluğu Merkezi nesnenin üzerine bindiğini görebiliriz. Merkezi nesne büyür, karmaşıklık artar. Diğer bir yöntem de **Observer** örüntüsü bunu daha sonraki yazılarımda anlatacağım.

Mediator Yapısı





Mediator Örnek Kod

```
class Program
{
    static void Main(string[] args)
    {
        // The client code.
        Component1 component1 = new Component1();
        Component2 component2 = new Component2();
        new ConcreteMediator(component1, component2);

        Console.WriteLine("Client triggets operation A.");
        component1.DoA();

        Console.WriteLine();

        Console.WriteLine("Client triggers operation D.");
        component2.DoD();
    }
}
```

```
// The Base Component provides the basic functionality of storing a
// mediator's instance inside component objects.
class BaseComponent
{
    protected IMediator _mediator;

    public BaseComponent(IMediator mediator = null)
    {
        this._mediator = mediator;
    }

    public void SetMediator(IMediator mediator)
    {
        this._mediator = mediator;
    }
}
```

```
class Component2 : BaseComponent
```

```
class Component1 : BaseComponent
{
    public void DoA()
    {
        Console.WriteLine("Component 1 does A.");

        this._mediator.Notify(this, "A");
    }

    public void DoB()
    {
        Console.WriteLine("Component 1 does B.");

        this._mediator.Notify(this, "B");
    }
}
```

```
// The Mediator interface declares a method used by components to notify the
// mediator about various events. The Mediator may react to these events and
// pass the execution to other components.
public interface IMediator
{
    void Notify(object sender, string ev);
}
```

```
// Concrete Mediators implement cooperative behavior by coordinating several
// components.
class ConcreteMediator : IMediator
{
    private Component1 _component1;

    private Component2 _component2;

    public ConcreteMediator(Component1 component1, Component2 component2)
    {
        this._component1 = component1;
        this._component1.SetMediator(this);
        this._component2 = component2;
        this._component2.SetMediator(this);
    }

    public void Notify(object sender, string ev)
    {
        if (ev == "A")
        {
            Console.WriteLine("Mediator reacts on A and triggers folowing operation
            this._component2.DoC();
        }
        if (ev == "D")
        {
            Console.WriteLine("Mediator reacts on D and triggers following operatio
            this._component1.DoB();
            this._component2.DoC();
        }
    }
}
```

Observer

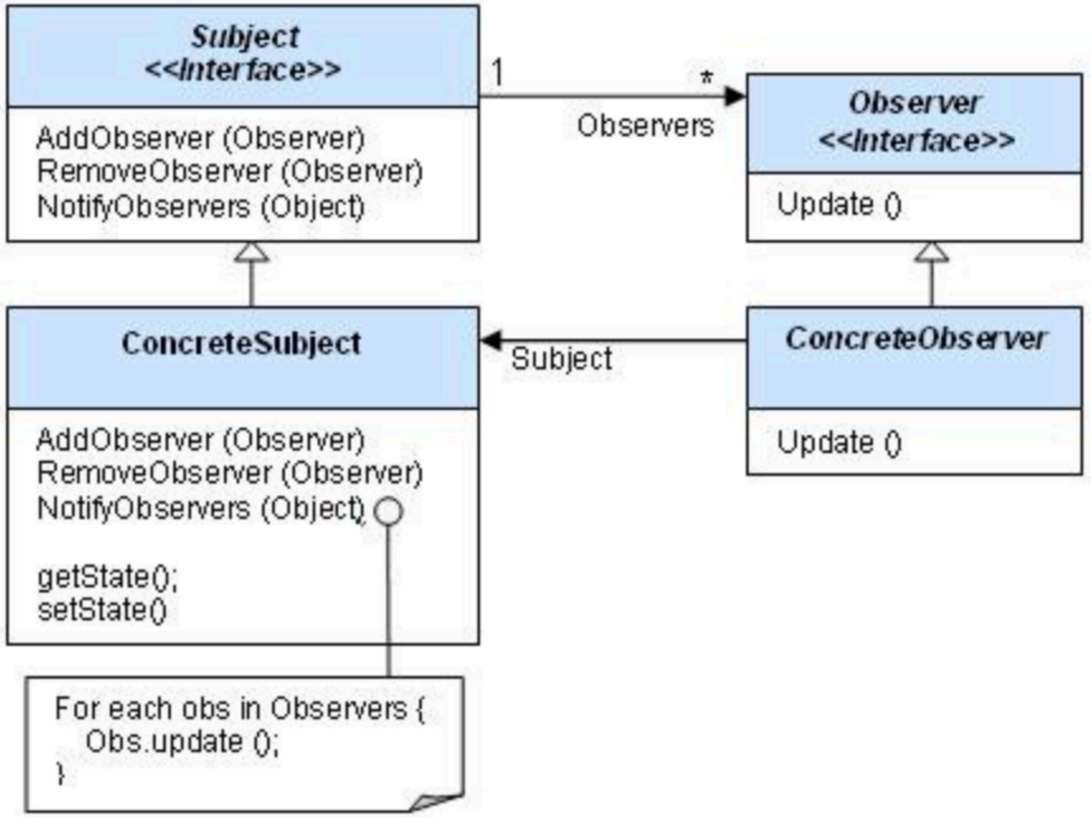
Observer Hangi Durumda Kullanmalıyız ?

Aynı bileşeni birden fazla kaynağın dinleme ihtiyacı fazla ve bileşen sayısının fazla olduğu durumda Observer örüntüsü pub/sub mekanizması ile bileşenlerin birbirine bağımlılıklarını azaltır. Ve tüm sistemin bu pub/sub mekanizmasındaki Eventler üzerinden haberleşmesini sağlar. Örnek UI bileşenleri

Observer

Observer örüntüsü tüm UI bileşenlerinin haberleşme ve iletişim temelini oluşturur. Bileşenlerin üzerinedeki Mouse, Keyboard, Gesture **eventlerini dinleyip** nesnede oluşan **değişikliklerini** kendilerini dinleyen Observer sınıflarına iletirler.

Observer Yapısı



Memento

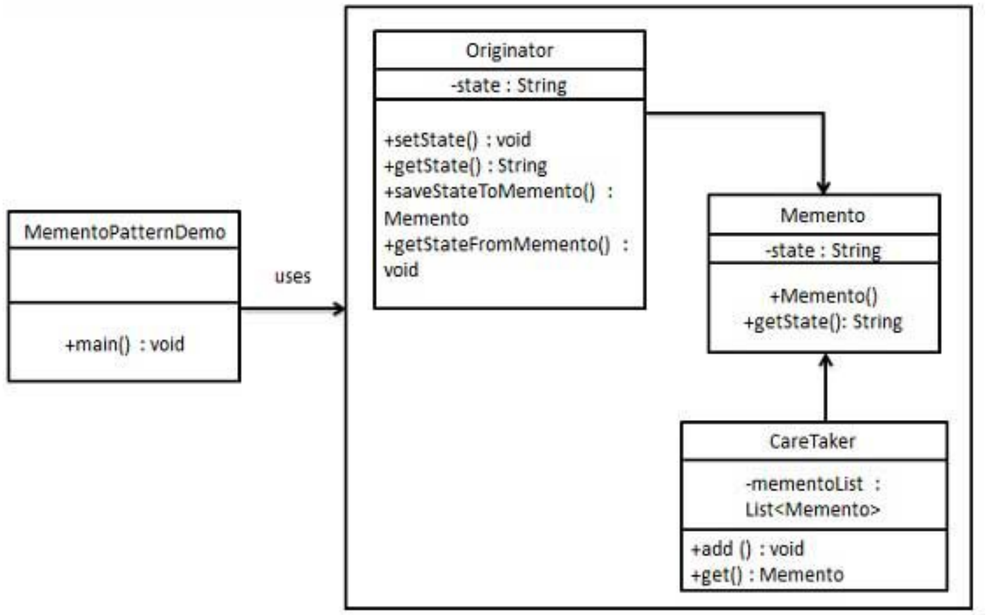
Memento Örnek

Örneğin bir bilgisayar oyunu oynuyorsunuz ve ara ara oyununuzu kaydederek Snapshot'lar oluşturuyorsunuz. Zamanı gelince istediğiniz Snapshot dönüp oyununuzu geri alarak devam edebilirsiniz.

Bunun için yazılımcı olarak oyun içi karakterlerinizin ve nesnelerin o anki parametrelerini kaydedip, yeri gelince bu parametrelere ulaşabilmemiz gerekir.

Memento örüntüsünü bu tip durumlarda kullanabilirsiniz. 3 tip sınıftan oluşur.

- **Originator:** Nesnenin nasıl kaydedilip nasıl çekileceğini bilen sınıftır.
- **Memento:** Kaydedilecek nesnenin state tutar.
- **CareTaker:** Memento nesnesinin versiyonlarını tutan yapıdır.



Memento

Memento örüntüsü nesnenizin **statelerini tutmak** ve
bu nesnelerde değişen state'lerden **geriye dönmek** istediğiniz zaman
kullanabileceğiniz bir örüntüdür.

Memento Kod

```
public class MementoPatternDemo {
   public static void main(String[] args) {

      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());

      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

```
public class Originator {
   private String state;

   public void setState(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento Memento){
      state = Memento.getState();
   }
}
```

```
public class Memento {
   private String state;

   public Memento(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }
}
```

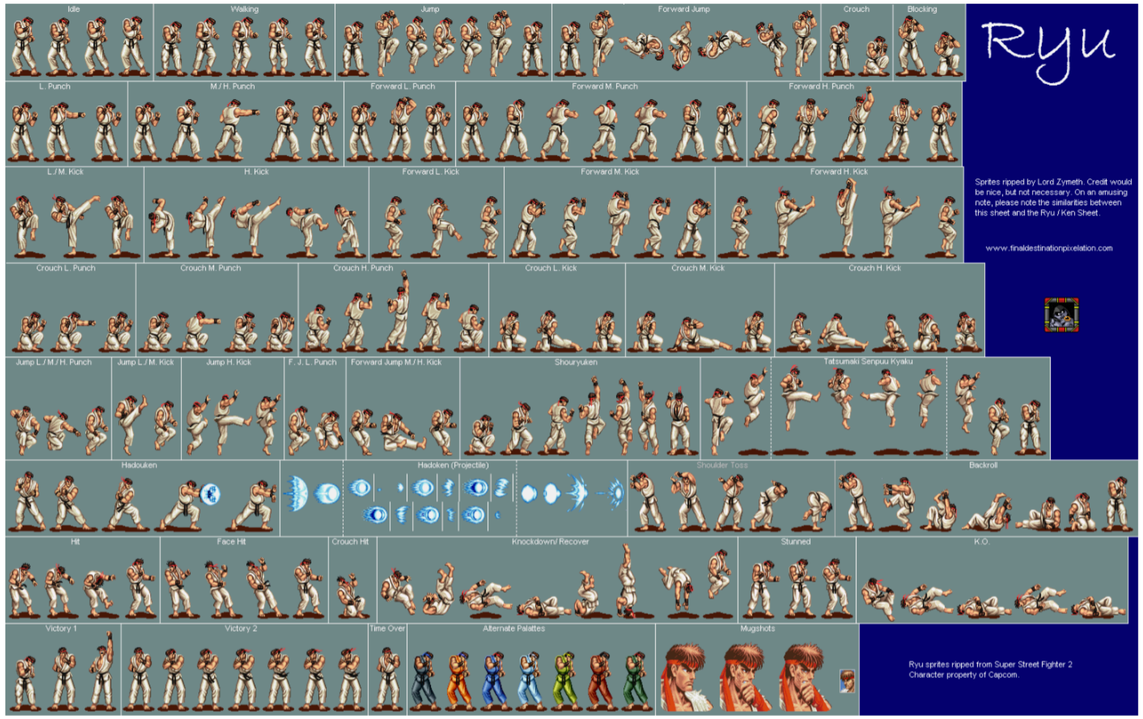
State

State Örnek Kullanım

Oyun geliştirmede çok kullanılan bir örüntüdür. Oyundaki nesnelerin hareketlerini belli durumlara göre gerçekleştirmeniz gerekir. Nesnenin ilgili durumda yapacağı hareketleri ve davranışları bu State sınıfları belirler

Örneğin Street Fighter oyununa bakalım. Oyunu kodlarken Ryu'nun haraket durumlarını kodlamamız gerekir.

- Idle (Sallanma)
- Walking
- Jump
- Forward Jump
- Crouch
- Blocking
- Left Punch
- M/H Punch
- Forward Left Punch



Elimizde State arayüzünden türeyen nesneler olması gerekir.

StateIdle, StateWaiting, StateJump gibi.. her bisinin *handle()* metodu içerisinde ilgili sprite run edilerek nesnenin bir durumdan → diğer duruma geçmesi sağlanır.

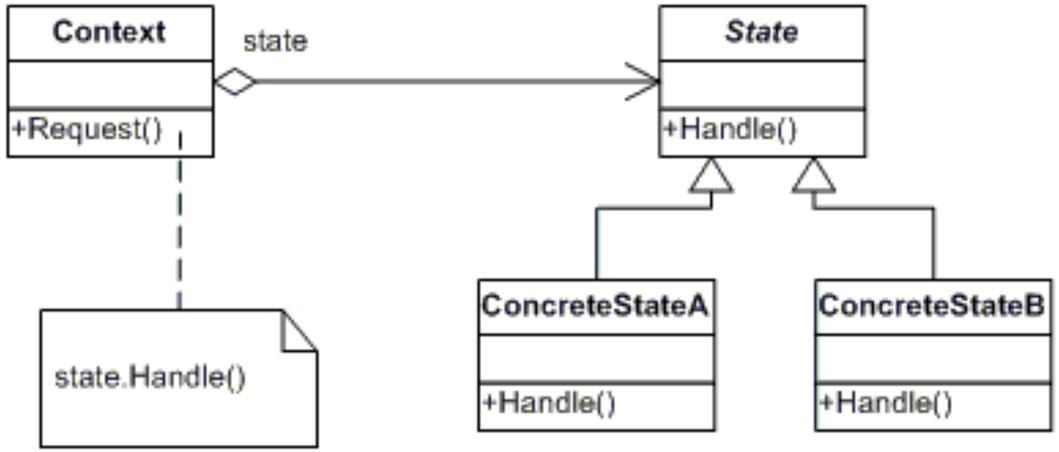
State

State örüntüsü, nesnenizin **iç durumunu** geçişlerinin belirlenerek bu state geçişleri üzerinden **davranış değişikliklerini** hedefler

State Yapısı

Context bu durumlar arasındaki geçişten sorumludur. Mesela Jump durumundan sonra blocking durumu olmaz, arasında illa Idle durumuna geçmek gerekir şeklinde kurallar var ise bu geçişi Context sınıfı gerçekleştirir.

Context yapılan isteklerin arkasında ya Oyuncu ya da Yapay zeka vardır. Buna göre Context kendisinden istenen davranışı yerine getirir.



Strategy

Örnek Kullanım

Örneğin Bir savaş alanında(arenasında) savaşan 3 tane robotunuz olsun. Bunların 3 tip davranış ile hareket etmesini istiyorsunuz.

- **Normal Davranış:** Normal hızla gezinir. Çevresinde gezen robotları aldırmaz. Rastgele hareket eder.
- **Defansif Davranış:** Yavaş hareket eder. Çevresinde gezen robotlar belli bir yakınlığa geldiğinde uzaklaşır.
- **Agrasif Davranış:** Hızlı hareket eder. Çevresinde gezen robot farkettiğinde üzerine doğru hareket eder.

Bunun için birinci yöntem Robot sınıfının içerisinde switch/case yazıp davranış mantıkları bu sınıf içerisinde yazdığınızda aşağıda ki 2 problem ile karşılaşırsınız.

- Robot'a yeni bir davranış eklemek için Robot nesnesini güncellemeniz gerekir.
- Bu davranışı Robot dışında bir nesnede kullanmak isterseniz kodu kopyalayıp/yapıştırmanız gerekir.

Aşağıdaki kodda **IBehaviour** interface bizim soyutlamak istediğimiz davranışı yani Robot'un *move* fonksiyonunu soyutlar.

Aşağıdaki kodda **IBehaviour** interface bizim soyutlamak istediğimiz davranışı yani Robot'un **move** fonksiyonunu soyutlar.

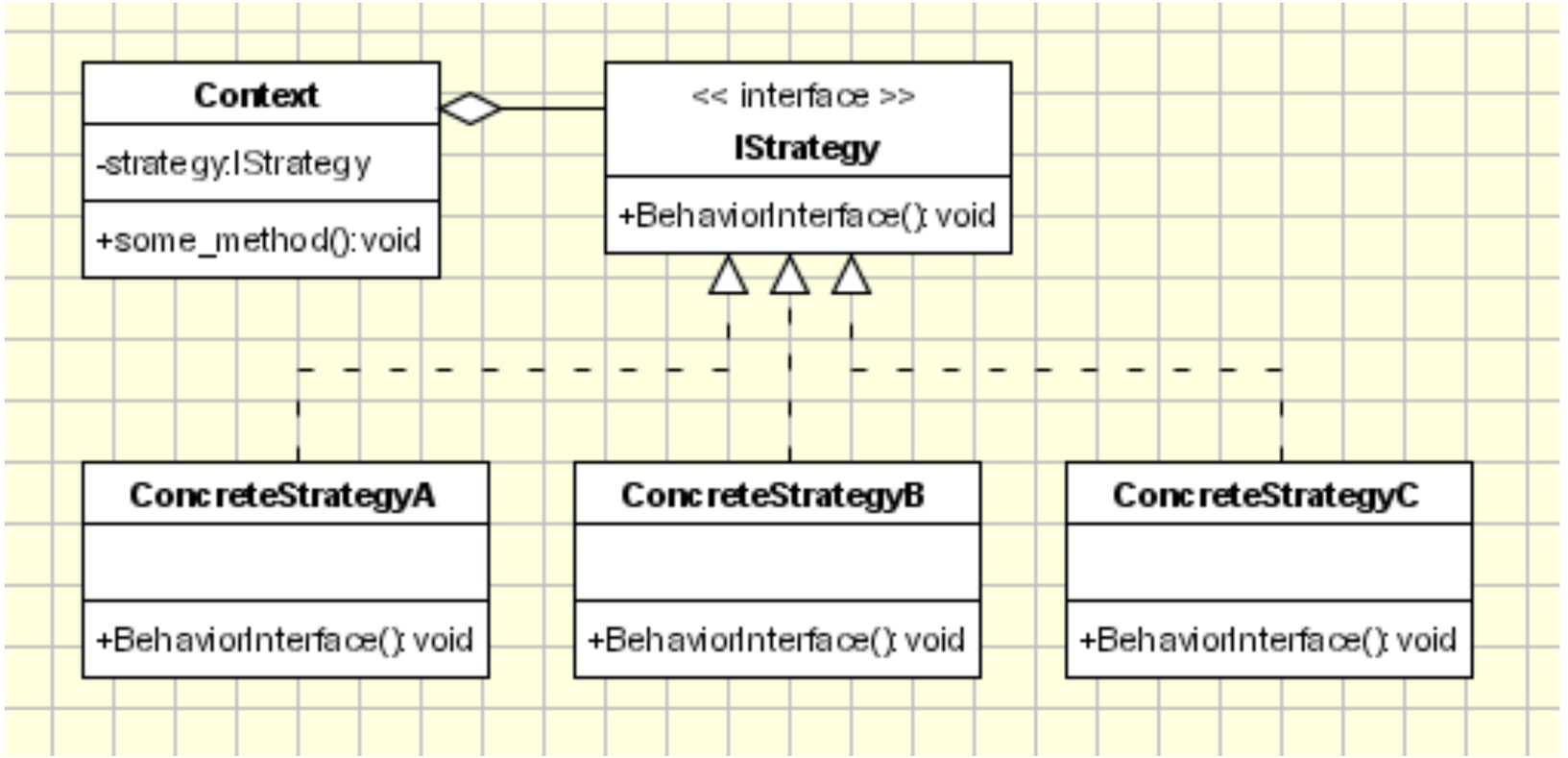
Bundan türeyen 3 tip davranış var. Robot nesnesi kendisine atanan davranışı Stratejisini çalıştırarak hareketini gerçekleştirir. Bu sayede Robot hangi hareketi yapacağı konusunda bir kodu/mantığı kendi içerisinde barındırmaz.

- "AggressiveBehaviour"
- "DefensiveBehaviour"
- "NormalBehaviour"

Strategy

Strategy örüntüsü sizin nesnenizin içindeki **davranışı soyutlayıp** dışarıdan bu **davranışı/stratejiyi** nesneye ekleyerek çalıştırmanızı sağlar.

Yapısı


Context
-strategy:IStrategy
+some_method():void
<< interface >>
IStrategy
+BehaviorInterface(): void
ConcreteStrategyA
+BehaviorInterface(): void
ConcreteStrategyB
+BehaviorInterface(): void
ConcreteStrategyC
+BehaviorInterface(): void

Örnek Kod

```
public interface IBehaviour {
	public int moveCommand();
}

public class AgressiveBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tAgressive Behaviour: if find another robot attack it");
		return 1;
	}
}

public class DefensiveBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tDefensive Behaviour: if find another robot run from it");
		return -1;
	}
}

public class NormalBehaviour implements IBehaviour{
	public int moveCommand()
	{
		System.out.println("\tNormal Behaviour: if find another robot ignore it");
		return 0;
	}
}

public class Robot {
	IBehaviour behaviour;
	String name;

	public Robot(String name)
	{
		this.name = name;
	}

	public void setBehaviour(IBehaviour behaviour)
	{
		this.behaviour = behaviour;
	}

	public IBehaviour getBehaviour()
	{
		return behaviour;
	}

	public void move()
	{
		System.out.println(this.name + ": Based on current position" +
					"the behaviour object decide the next move:");
		int command = behaviour.moveCommand();
		// ... send the command to mechanisms
		System.out.println("\tThe result returned by behaviour object " +
				"is sent to the movement mechanisms " +
				" for the robot '"  + this.name + "'");
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

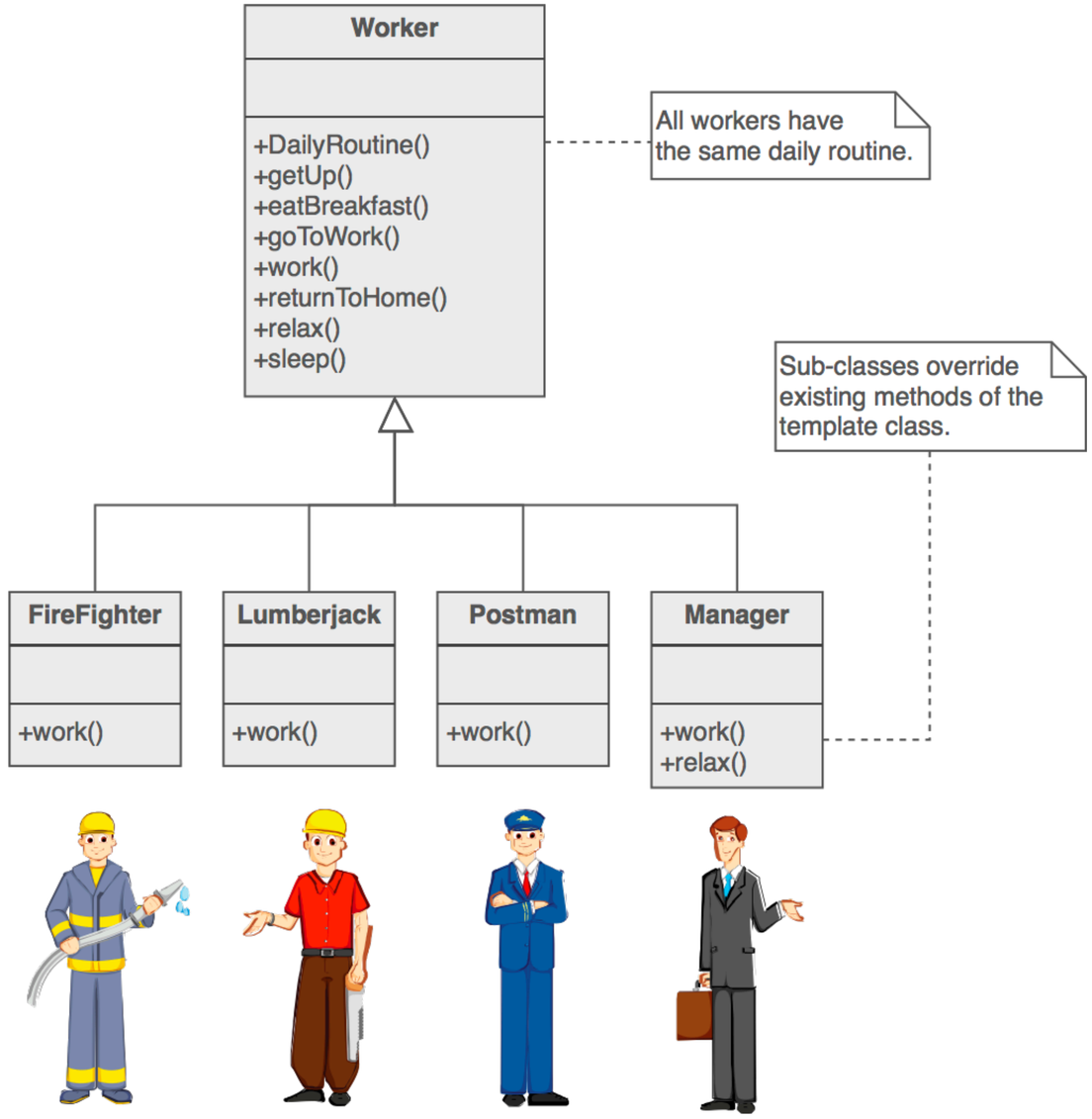
```
public class Main {

	public static void main(String[] args) {

		Robot r1 = new Robot("Big Robot");
		Robot r2 = new Robot("George v.2.1");
		Robot r3 = new Robot("R2");

		r1.setBehaviour(new AgressiveBehaviour());
		r2.setBehaviour(new DefensiveBehaviour());
		r3.setBehaviour(new NormalBehaviour());

		r1.move();
		r2.move();
		r3.move();

		System.out.println("\r\nNew behaviours: " +
				"\r\n\t'Big Robot' gets really scared" +
				"\r\n\t, 'George v.2.1' becomes really mad because" +
				"it's always attacked by other robots" +
				"\r\n\t and R2 keeps its calm\r\n");

		r1.setBehaviour(new DefensiveBehaviour());
		r2.setBehaviour(new AgressiveBehaviour());

		r1.move();
		r2.move();
		r3.move();
	}
}
```

Template Method

Örnek Kullanım

Örneğin Worker Abstract sınıfındaki *DailyRoutine()* template methoddur. Bu method içerisinde sıra ile aşağıdaki fonksiyonlar çağrılır ve bu fonksiyon değiştirilemez.

- getUp
- eatBreakfast
- goToWork
- work
- returnToHome
- relex
- sleep



Template Method

Template Method örüntüsü Abstract sınıf içerisinde diğer fonksiyonların
belli bir sıra ile çağrılma zorunluluğu var ise
bu sınıf **final** hale getirilerek Abstract sınıftan türeyen alt sınıflar
bu Template Method **değiştirilmesi engellediği** örüntüdür.

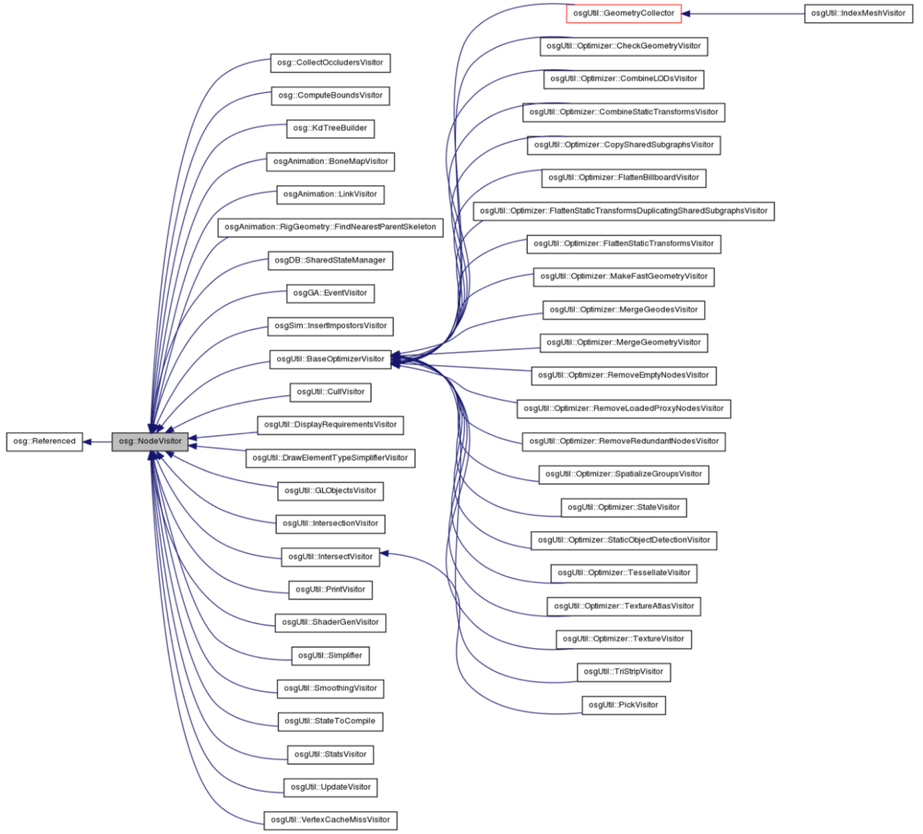
Visitor

Örnek Kullanım

Visitor örüntüsü denilince aklımıza hemen SceneGraph gelir. Yani Sahne veya UI nesnelerinin Composite yapıda iç içe tutulduğu yapılardır. Visitor bütün bu nesnelerin nasıl gezilmesi gerektiğini soyutlamayı sağlayan bir örüntüdür.

- **Rendering Visitor:** Kamera'nın bakış açısından ZBuffer olarak en uzaktaki nesneden en yakındaki nesneyi gezecek şekilde *render()* metodu çağrılır.
- **Intersection:** ZBuffer olarak Mouse'un en öndeki nesneden en arkadaki nesneyi gezecek şekilde *intersect()* metodu çağrılırki en öndeki nesne seçilsin.

İhtiyaca göre visitor çeşitlilikleri oluşturulabilir. Örneğin aşağıdaki resimde OpenSceneGraph'ta yer alan visitor sınıflarını görebilirsiniz. Aynı iterator örüntüsünde olduğu gibi ağaç yapısında olan Node'ları gezmek için kullanılır.

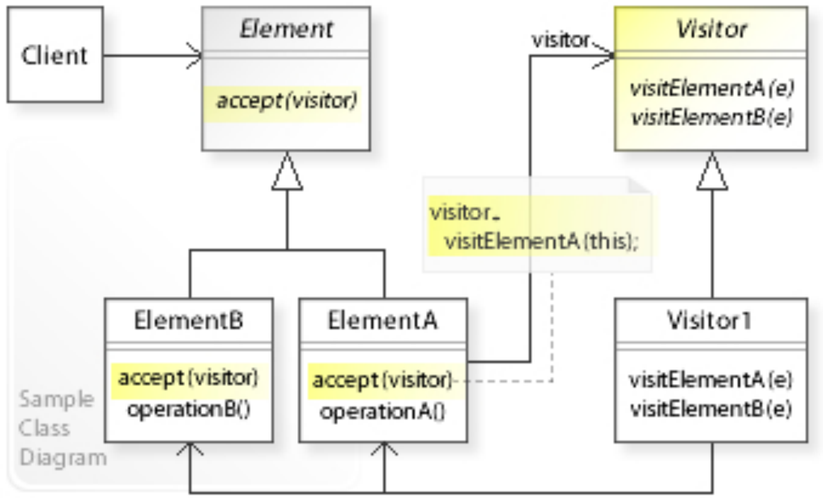


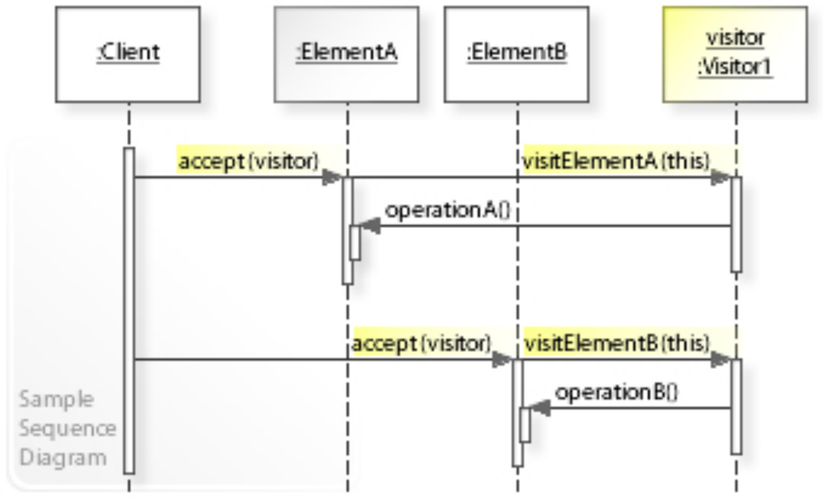
Bu visitor sınıfları genelde **travarse()/visit()** fonksiyonunda **recursive** olarak Node'lar içinde dönerek gerçekleştirilir.

Visitor

Veri yapıları üzerinde **spesifik bir işlemi** gerçekleştirmek için oluşturulan **gezinme** örüntüsüdür.

Visitor Yapısı

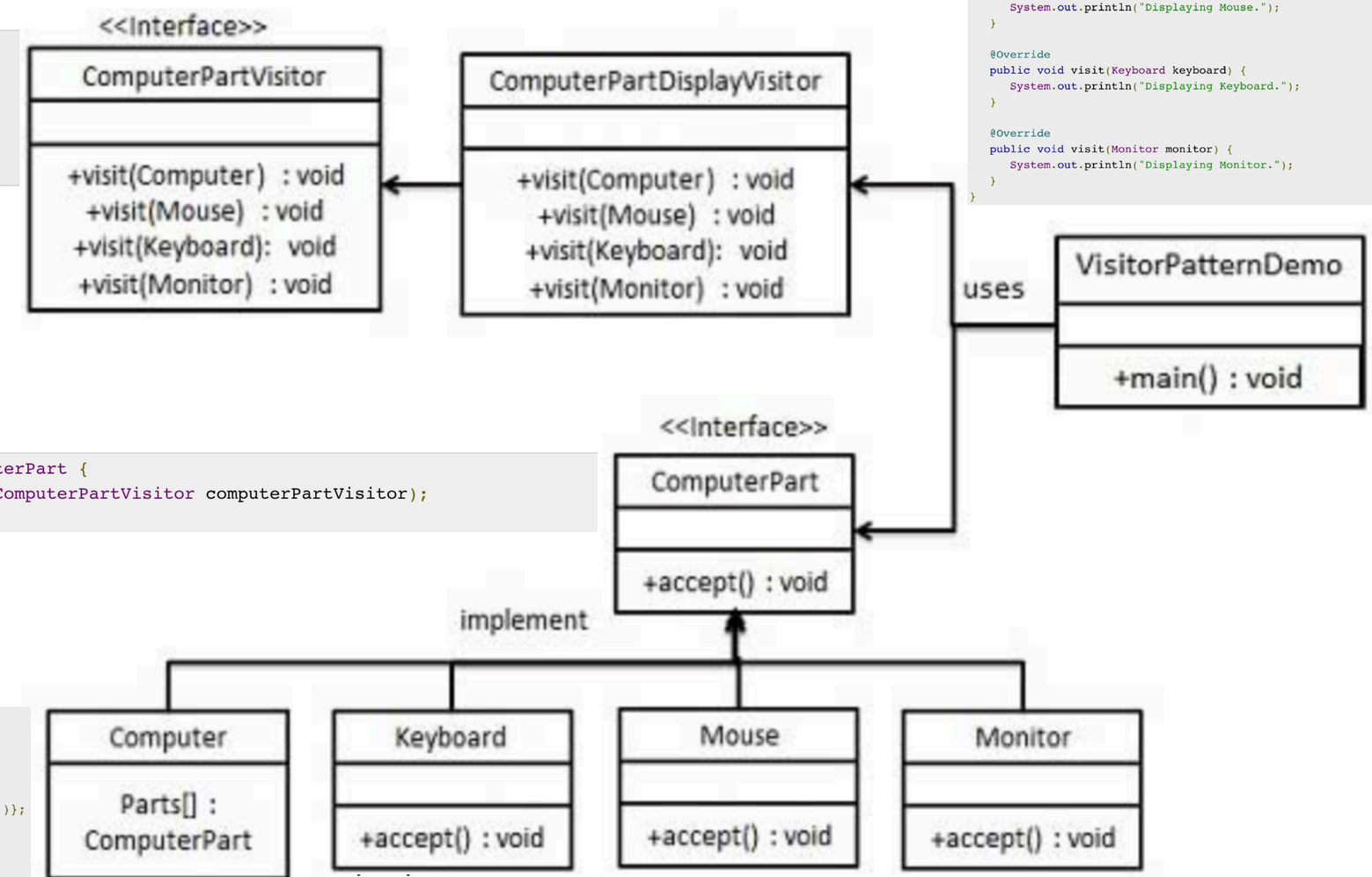




Visitor Yapısı

```java
public interface ComputerPartVisitor {
        public void visit(Computer computer);
        public void visit(Mouse mouse);
        public void visit(Keyboard keyboard);
        public void visit(Monitor monitor);
}
```

```java
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

   @Override
   public void visit(Computer computer) {
      System.out.println("Displaying Computer.");
   }

   @Override
   public void visit(Mouse mouse) {
      System.out.println("Displaying Mouse.");
   }

   @Override
   public void visit(Keyboard keyboard) {
      System.out.println("Displaying Keyboard.");
   }

   @Override
   public void visit(Monitor monitor) {
      System.out.println("Displaying Monitor.");
   }
}
```




```java
public interface ComputerPart {
   public void accept(ComputerPartVisitor computerPartVisitor);
}
```

```java
public class Computer implements ComputerPart {

   ComputerPart[] parts;

   public Computer(){
      parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
   }


   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      for (int i = 0; i < parts.length; i++) {
         parts[i].accept(computerPartVisitor);
      }
      computerPartVisitor.visit(this);
   }
}
```

```java
public class Keyboard implements ComputerPart {

   @Override
   public void accept(ComputerPartVisitor computerPartVisitor) {
      computerPartVisitor.visit(this);
   }
}
```

Behavioral Patterns Özet

Chain of Responsibility: İsteği zincir halindeki işlemci sınıflar üzerinde işleterek sorumluluğun aktarıldığı örüntüdür.

Command: İsteği Komut Objesi haline getirip (komut ve parametreleri) sistemi bu komutlar üzerinden işletildiği örüntüdür.

Interpreter: SQL, Regex vb.. gramer içeren bir String parçalarına ayırarak AST oluşturmayı sağlayan örüntüdür.
(Girdinin çok dinamik olabilmesini sağlar)

Iterator: Alttaki veri yapısını ve gösterimi ile ilgilenmeden sıra ile bu veri yapılarındaki elemanlar üzerinde gezinmeyi sağlar.

Mediator: Bir çok sınıfın birbirini bilip metodlarını çağırması yerine bir sınıf diğer tüm sınıfları bilerek yönetmeyi sağlamasıdır.

Memento: Nesnenin bir önceki state dönebilmesini sağlayan örüntüdür.

Observer: publish/subscribe örüntüsüdür. Observer Eventlere bağlanıp dağıtık bir sistemde çalışmasını sağlar

State: İç state değişimleri yöneterek nesnenin davranış değişikliğini yöneten örüntüdür.

Strategy: Bir algoritma grubunun run-time on the fly verilen strategy gore işletilmesini sağlayan örüntüdür.

Template Method: Bir algoritmanın adım adım oluşturulabiliyorsa , sub-class bunlardan kendilerine özelleşmişlerini override eder kullanılmasını sağlar.

Visitor: Veri yapısının hiyerarşik yapısı ile bu yapı üzerinde gezerken çalışacak algoritmaların ayrılmasını sağlayan örüntüdür.

Antipatterns

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~

Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~

Kod Geliştirme Yöntemleri → ~~AOP~~ ~~TDD~~ ~~SOLID~~ ~~KISS~~ ~~DRY~~ ~~YAGNI~~

Kodu Dayanıklı Geliştirmek → ~~12 Factor App~~ ~~Reactive Programming~~ ~~Chaos Eng~~

Design Patterns → ~~Design Patterns~~ ~~Creational Patterns~~ ~~Structural Patterns~~ ~~Behavioral Patterns~~

Anti Patterns → Anti Patterns, Spaghetti Code, Lava Flow Programming, Golden Hammer, Reinvent The Wheel

AntiPattern

AntiPattern, Yazılım Geliştirme sırasında **kaçınılması gereken** şeylerdir. Bundan önceki konularda Mimari Örüntüler ve Tasarım Örüntülerinden bahsetmiştim.

Mimari Örüntürler önceki projelerden elde ettiğimiz **benzer stilleri**, Tasarım Örüntüleri ise **Problem-Çözüm** ilişkilerini yeni geliştireceğimiz projelerde kullanmamızı önerir.

Antipattern ise kısa vadede kendisi çözümmüş gibi göstersede aslında projenize zarar veren **sakınmanız ve uygulamamanız** gereken tekrarlayan davranış şekilleridir.

AntiPattern tasarım, geliştirme ve süreçlerin işletilmesi sırasında karşımıza çıkarlar ve temelde **3 gruba ayrılırlar.**

- Development AntiPatterns (Geliştirme)
- Architecture AntiPatterns (Mimari)
- Management AntiPatterns (Yönetimsel)

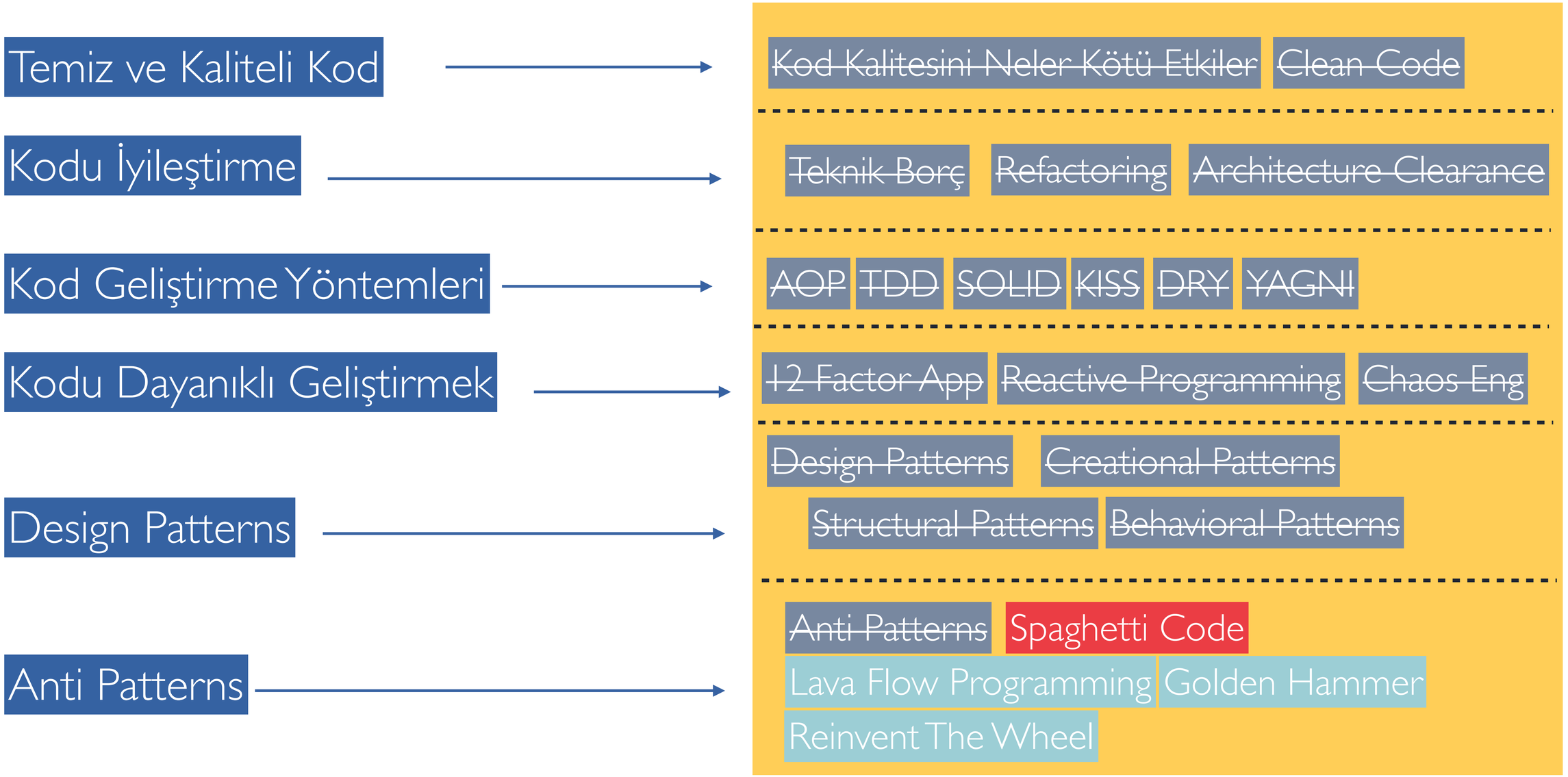
AntiPattern

AntiPattern isimleri genelde akılda kalacak şekilde gerçek hayattan benzerlikler içerir. Örneğin Spagetti Kod Makarna'nın birbirine karmaşması ile Kodun karmaşıklığı birbirine benzetilmek istenmiştir.

Bir çok antipattern bulunuyor. Ben ilerleyen zamanda bu antipattern bazılarından detaylıca bahsedeceğim. (Wikipedia)

Analysis paralysis, Cash cow, Design by committee, Escalation of commitment, Management by perkele, Matrix Management, Moral hazard, Mushroom management, Silos, Vendor lock-in, Death march, Groupthink Smoke and mirrors, Software bloat, Waterfall model, Bystander apathy, Abstraction inversion, Ambiguous viewpoint, Big ball of mud, Database-as-IPC, Gold plating, Inner-platform effect, Input kludge, Interface bloat, Accidental complexity, Action at a distance, Blind faith, Boat anchor, Busy spin, Caching failure, Cargo cult programming, Coding by exception, Error hiding, Hard code, **Lava flow**, Loop-switch sequence, Magic numbers, Magic strings, Soft code, **Spaghetti code**, Copy and paste programming, **Golden hammer**, Improbability factor, Not Invented Here (NIH) syndrome, Premature optimization, Programming by permutation, **Reinventing the wheel**, Reinventing the square wheel, Extension conflict, JAR hell, BaseBean, Call super, Circle-ellipse problem, Circular dependency, Constant interface, **God object**, Object cesspool, Object orgy, Poltergeists, Sequential coupling, Yo-yo problem, Hurry up and wait, Magic pushbutton, Race hazard Stovepipe system Anemic Domain Model Silver bullet Tester Driven Development Dependency hell ,DLL hell …

2nci Gün (Code Quality)



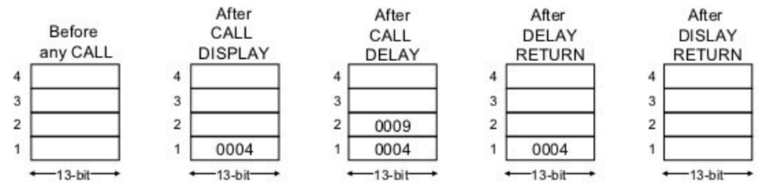
Spaghetti kod

Kodun akışının **takip edilemez** şekilde bir yerlerden bir yerlere zıplaması, atlaması demek. Eskiden Low Level Programlama dillleri ile kod geliştiriliyordu. Komutlar içerisinde bir yerden bir yere zıplayarak kod geliştiriliyordu.

GOTO

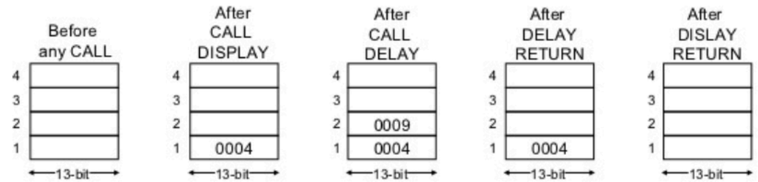
İlk Assembly yani makine dilinde Register(yazmaç) veriyi taşı, toplama işlemi yap sonra bunu diske yaz emirlerinin olduğu kod blokları arka arka yazıldığını görebilirsiniz.

```
00020 ;delay subroutine
                00021          ORG     30H     ;put delay at address 30H
0030   30FF     00022 DELAY    MOVLW   0xFF    ;WREG = 255
0031   0091     00023          MOVWF   MYREG
0032   0000     00024 AGAIN    NOP             ;no operation wastes clock cycles
0033   0000     00025          NOP
0034   0000     00026          NOP
0035   0B91     00027          DECFSZ          MYREG,F  ;decrease until MYREG
becomes 0
0036   2???     00028          GOTO    AGAIN   ;repeat decrement process
0037   0008     00029          RETURN          ;return to caller
                00030          END
```



Assembly Code

```
00020 ;delay subroutine
                00021          ORG     30H        ;put delay at address 30H
0030  30FF      00022 DELAY    MOVLW   0xFF       ;WREG = 255
0031  0091      00023          MOVWF   MYREG
0032  0000      00024 AGAIN    NOP                ;no operation wastes clock cycles
0033  0000      00025          NOP
0034  0000      00026          NOP
0035  0B91      00027          DECFSZ             MYREG,F  ;decrease until MYREG
becomes 0
0036  2???      00028          GOTO    AGAIN      ;repeat decrement process
0037  0008      00029          RETURN             ;return to caller
                00030          END
```



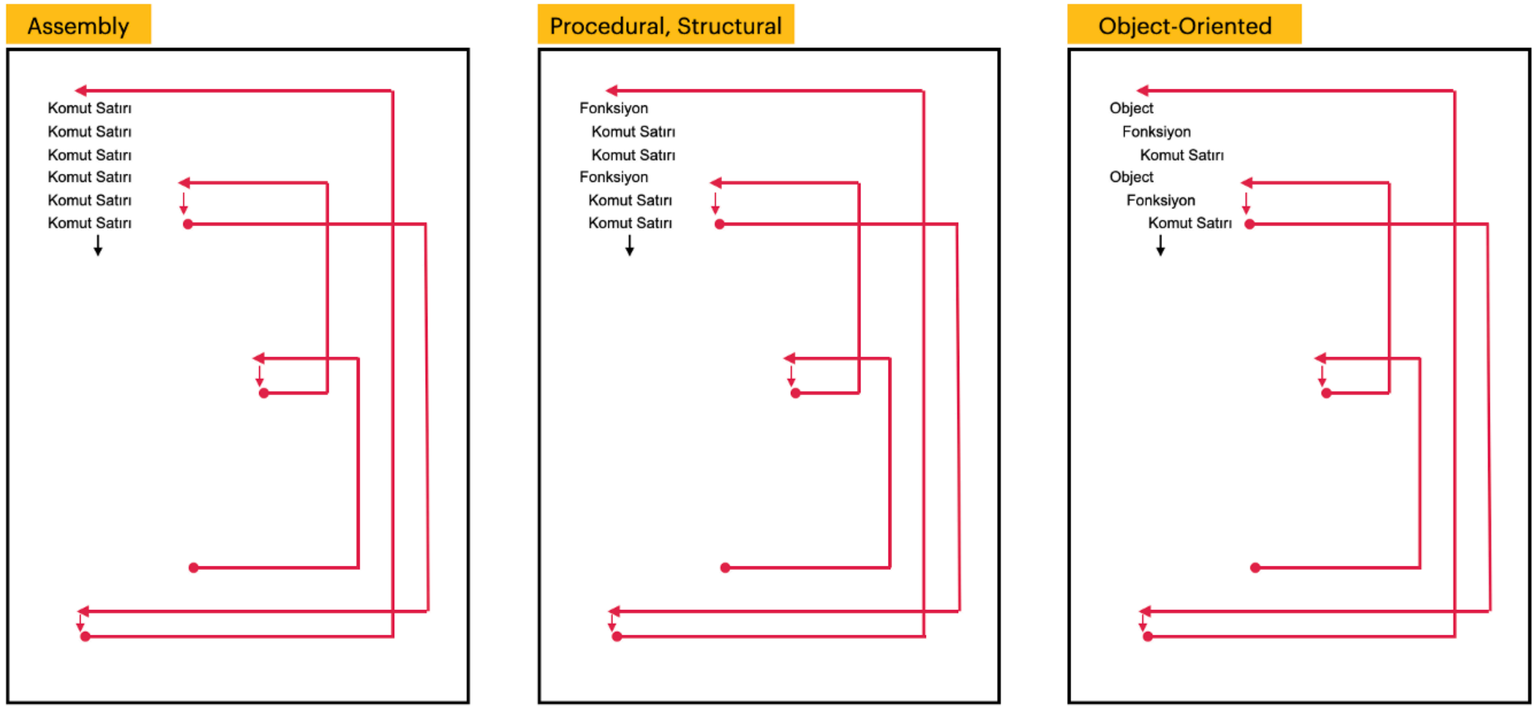
Zaman içerisinde **High Level Language** (Yüksek seviyeli diller ile birlikte) makine dilinden uzaklaştıkça GOTO kullanımının yazılım geliştirmeye zarar verdiği ve kodun spagetti gibi karıştırdığını belirtmişlerdir. Go to Statement Considered Harmfull . (Dijkstra 1968)

Bu konunun detayı için Programlama Paradigmaları içerisinde bulunan GOTO, Structural, Procedural, OOP ve Functional Programming kavramlarını detaylıca inceleyebilirsiniz.

Asıl Sorun Kod içerisinde Kontrolsüz Zıplama

Peki artık **GoTo** ile kod geliştirmiyoruz. Spaghetti Code hale biz geliştiriciler için bir sorun mu ?

Evet hangi seviyede kod yazarsanız yazın, hatta microservis geliştirme yapın kodun akışı aşağıdaki şekilde gerçekleşecektir.

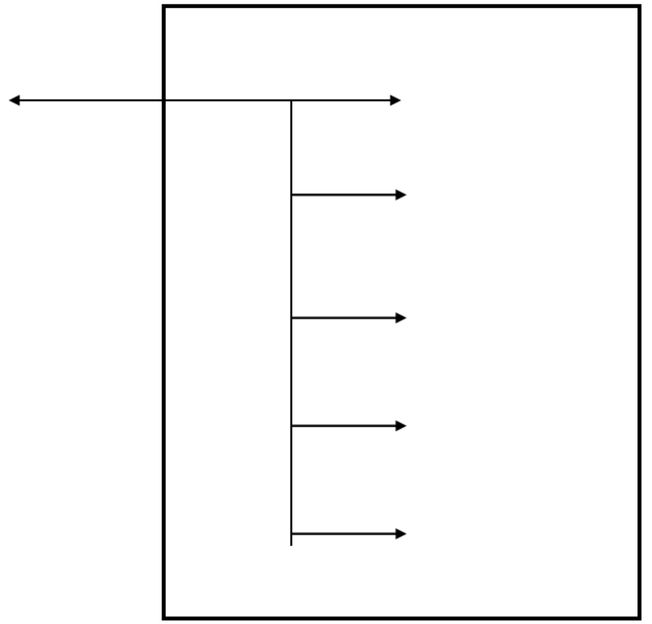


Kodun işleyişi bu şekilde neden oluyor peki ?

- Koşullar
- Döngüler
- I/O Erişim (Disk, Grafik Kartı, Network)
- Durumlar Arası Geçişler
- Hata Durumları
- Async İşlemler

Özetle **business flow (iş akışını gerçekleştirmek)** için kodunuz sürekli olarak dallanması bir yerden bir yere atlaması gerekiyor. Asıl mesela geliştiricilerin bu atlamayı nasıl yönettikleri ile ilgili.

Akışı Yönetmek

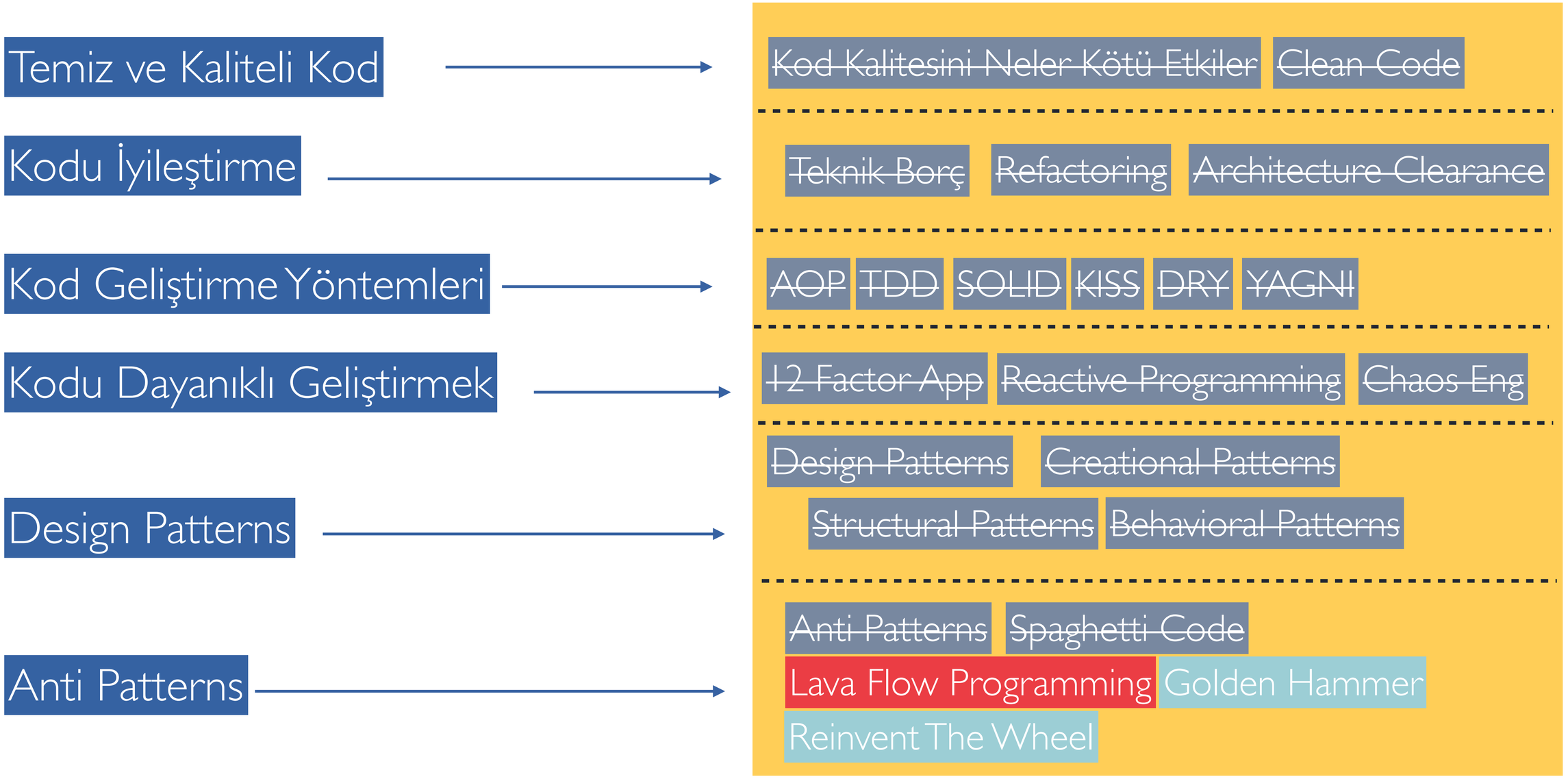


Kod içerisinde biraz öncede bahsettiğim gibi farklı seviyelerde farklı özellikleri kullanarak bu zıplamayı daha kontrol edilebilir, daha izlenebilir ve daha okunaklı hale getirebilirsiniz.

Örneğin çok uç bir örnek olacak ama Mimari seviyede transactional bir işlemi bir kuyruk mekaniğine SQS, ActiveMQ, RabbitMQ farklı processler bağlayarak kod işletimini State Mekaniğini bu yapılar üzerinde yönetebilirsiniz.

Bu işlemler için **Saga Patterns, Step Functions** yapıları kullanabilirsiniz.

2nci Gün (Code Quality)



Lava Flow Programming

Lav Akışı ile programlama, **canlı için hazır olmayan kodun** canlıya gerekli düzeltmeleri yapmadan çıkması, yeterli dokümantasyonu olmaması.

Kodun ilgili kısımlarının nasıl çalıştığı **know-how (bilgi birikimi)** zaman içerisinde ekip tecrübesinden silinmesi, ekibin koda dokunmaktan korkar hale gelmesidir.

Lava Flow Programming Nedenleri

Hangi ürün olursa olsun, geliştirilen ürünün analiz, tasarım, geliştirme veya test süreçleri **hiçbir zaman tam olmaz**. Hatta ekip yapacakları ürün hakkında yeterli deneyime sahip değilse ürünün içerisine gereğinden fazla özellik, gereğinden fazla kod eklemiş olabilir. Bazı kodları sırf denemek için eklemiş ama yeterli temizliği yapmadan canlıya çıkmış olabilir.

Bu süreç zaman içerisinde **kod geri dönülemez sorun yumağına sanki bir kartopu gibi dönüşerek** işin içinden çıkılamaz hale gelebilir. Çünkü canlıda ki her kodun ekstra bakım yükü olacaktır

Lava Flow Programming Nedenleri

Kodunuzun düzgün ve neden bu şekilde çalıştığını belirten

- bir **tasarım, analiz dokümanı** yoksa,
- kod bir takım örüntüler, **yazılım geliştirme prensipleri (SOLID, DRY, YAGNI)** vb… uzak geliştiriyor ise.
- Doğru **modüler yapılar** kurulmamış ise,
- Mantıklı ve doğru **Soyutlama ve Kapsama** yapılmıyor ise,
- **Kaliteli testler** kodun çalışma akışlarını korumuyor ise,
- Çalışan kişiler çok sık ve **kısa zaman aralıkları ile değişmiş** ise,
- Kod müşteri tarafından kullanılmıyor ve sürekli bir doğrulamadan geçmiyor ise kod bir zaman sonra ekibin nasıl çalıştığını bilmediği, dokunmaktan korktukları bir hal alır.

Özellikle ekip içerisinde geliştiricilerin değişmesi sırasında hep bir **know-how ve bilgi eksikliği** olacaktır.

Bu Durum Nasıl Sorunlara Yol Açar ?

Programlama seviyesinde problemler;

Bir API/Model geliştirdiniz. Bu API/Model kullanan başka servisler geliştirildi.

Yeni sürümlerle birlikte eskiden olan bazı parametre ve değerlerin gereksiz olduğuna karar verdiniz bunları silmek istediniz çünkü ek yük getiriyor. Öncelikle silmek istediğiniz alanları kullanan başka projeler olmuş mu? Bunları kullanarak geliştirme yapmışlar mı ?

Burda devreye eski sürümlere uyumluluk devreye girer. Bu yüzden geçmişte yapılan geliştirmeler ve koddaki özellikleri ekip üyelerinin çok iyi bilmesi ve neden oraya eklendiğinin, nasıl değiştirecekleri konusunda bir planlarının olması önemlidir.

Bu Durum Nasıl Sorunlara Yol Açar ?

Veri seviyesinde problemler;

Geliştirdiğiniz üründe fazladan veri oluşturdunuz veya fazladan veri girişi yaptırdınız. Bu durumda sisteminizde 1000 lerce kullanıcı için ekstradan

- bellekte ve diskte alan tutacak
- network daha fazla veri gönderecek
- ve bu verinin işlenmesi, güvenliği
- vb.. konularda fazladan enerji harcayacaksınız.

Birde bu hatalı geliştirmelerin üzerine daha fazla programlama ve veri inşaa edildi.

Bu durum dan nasıl kurtulacaksınız ? Sileceğiniz verinin etkisi olan yazılımlarda bunu yönetmeniz gerekmektedir. Bu verinin etkisini ve oluşturabileceği müşteri rahatsızlıklarını , konuşarak, bilgilendirme yaparak çözmeye çalışmalısınız.

Bu Durum Nasıl Sorunlara Yol Açar ?

Teknolojik gelişmeler legacy(eski) yapı ile çelişebilir.

Gerçekte (PoC) Proof Of Concept çalışmaları, veya başka bir alanda deney projeler ile altyapılar yapmak hep savunduğum bir yöntemdir. Burada önemli olan bu deneysel çalışmaları nasıl gerçek ürünün içerisine taşıyacağımızdır.

Yöntem nasıl olmalıdır ?

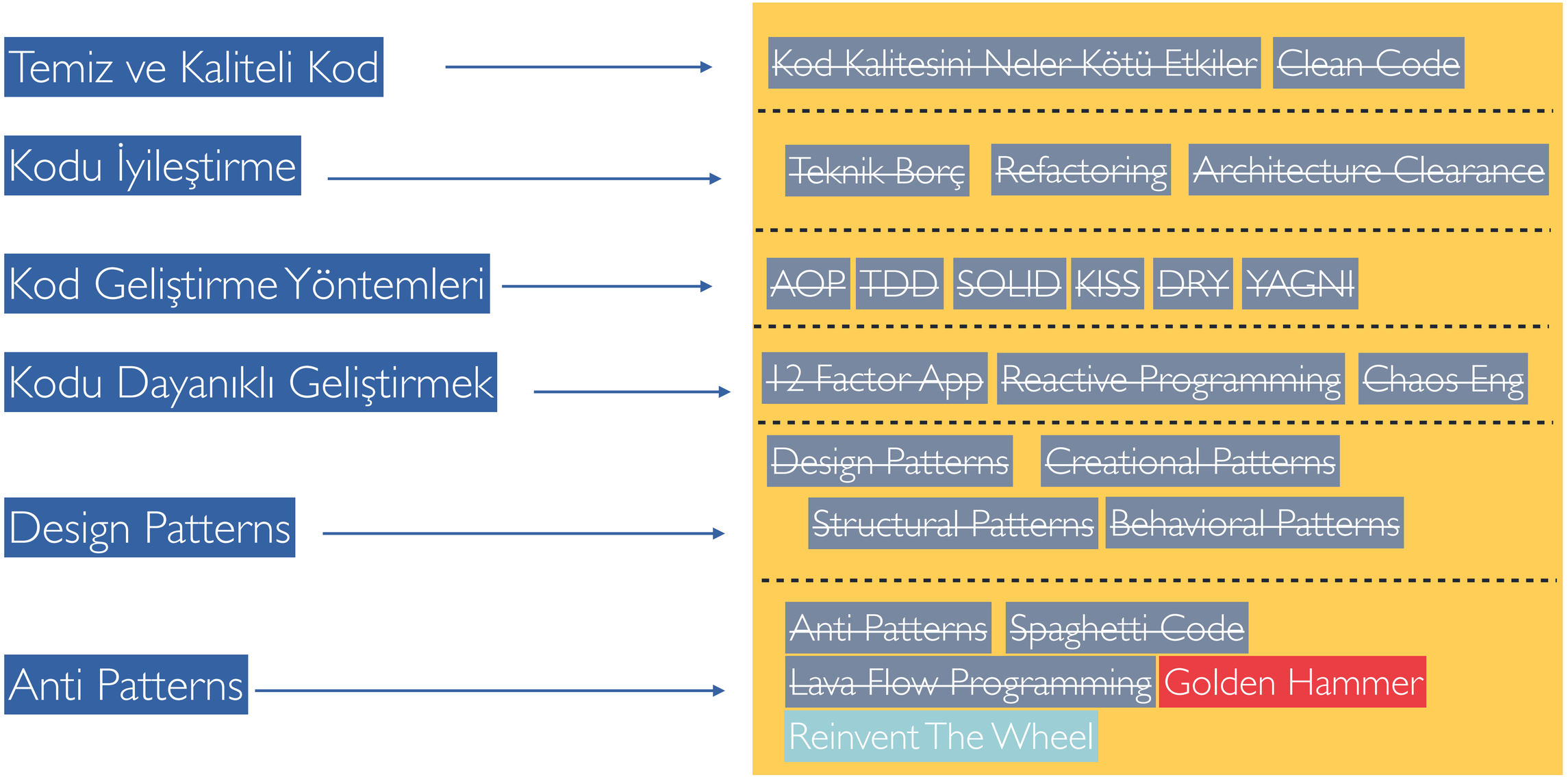
- Öncelikle analizin iyi yapılmış, ürün kapsamının ve beklentilerin iyi anlaşılmış olması gerekmektedir.
- Kütüphane seçimi yapılacaksa, bu kütüphane geliştirilmeye devam ediyor mu ? Açık kaynaklı mı? Kütüphane ile ilgili çıkan hatalar fixleniyor mu, ne kadar büyük bir komünite tarafından destekleniyor. İhtiyaçlarınızı karşılamadığı durumda değiştirebilir misiniz?
- Ürün geliştirmede kullandığını teknoloji/kütüphaneler ne kadar hakimsiniz, yeri geldiğinide üzerinde kendi projenize özel geliştirmeler yapabiliyor musunuz ?
- Yeri geldiğinde bu kütüphaneleri alternatifleri ile değiştirebilecek planlamalar yaptınız mı ?

Lava Flow Programming Özet

Eğer geliştirme ekibi **teknoloji/kütüphane** konusunda bu analiz çalışmalarını iyi dokümante etmeyip, sonraki geliştiricilere iyi bir şekilde anlatmadıysa, yeni gelenler sürekli olarak bu teknolojiler niye seçildi. Yenilerine geçelim. Yenileri daha başarılı diyerek hemen üst yönetimi ikna çalışmalarına başlayacaklardır. 😀

Lav Akışı ile programlama, canlı için hazır olmayan kodun canlıya gerekli düzeltmeleri yapmadan çıkması, yeterli dokümantasyonu olmaması. Kodun ilgili kısımlarının nasıl çalıştığı know-how (bilgi birikimi) zaman içerisinde ekip tecrübesinden silinmesi, **ekibin koda dokunmaktan korkar hale gelmesi**. Kodun içerisindeki bir takım yanlışlıkları, hataları görüp bunlara müdahale edemez duruma gelmek.

2nci Gün (Code Quality)



Golden Hammer AntiPattern

Bu anti-pattern benzer bir konuya Yazılım Mimarının 5 Yanlışı yazısı içerisindeki Mimarın kullandığı yöntemlerde **kendisine en çok güvendiği alanda sorunları çözmeye çalışır** sözünü hatırlatıyor. Ama bu yöntem her zaman bizi doğru yola götürmez.

Genelde ekibinize verilen süre kısa, bütçe kaynağı az olur. Kısa zamanda çok iş başarmanız istenir. Bu durumda **Maslow law of the instrument** söylediği gibi, ufak bir çocuğa çekici verirseniz, tüm gördüğü şeylere çekiçle vuracaktır.
Her şeyi(problemi) bir çivi olarak görmeye başlayacaktır.

“I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.
— Abraham Maslow said in 1966
Bütün sorunlarını, problemlerini aynı yöntem ile çözmeye çalışacaktır. (Maslow burda Metafor yapıyor aslında)

Çivi : problem, sorun, geliştirme, vb.
Çekiç : problemi çözme yöntemi.

Golden Hammer AntiPattern Karşı

Bu anti-pattern özellikle alternatif yöntemleri üzerine düşünmeyi önerir. Önceki projelerinizde çok büyük başarılar elde etmişseniz.

Aynı;

- modelin
- yöntemin
- yönetim modelinin
- teknolojinin
- kütüphanelerin
- market stratejilerinin

yeni karşılaştığınız sorunlara aynı şekilde merhem olacağı, çözeceğinin garantisi yoktur. Bu ancak problem ne kadar önceki probleme benziyor ise o kadar geçerlidir. Hatta zaman faktörü ve kullanıcıların değiştiğini düşününce aynı yöntemin birebir çalışmayacağı bir gerçektir.

Golden Hammer AntiPattern Çözüm

Burada mühendis olarak alternatifleri değerlendirme metodumuzun olup, bir problem geldiğinde direk bildiğimiz çözümü uygulamak yanında alternatiflerinide denemek, PoC (Proof Of Concept) çalışmaları yapmak çok önemlidir. Alternatifler arasında maliyet karşılaştırması yapıp, buna göre seçim yaparak teknoloji, kütüphane, yöntem seçmemiz daha doğru olur.

Önemli Not: Bazı yazılarda Golden Hammer Anti-pattern çözmek için biraz araştırma yapmanız yeterli şeklinde yazılar yazmışlar. Örneğin;

blog okuyun,
karşılaştırmalara bakın,
stackoverflow bakın
vb

yöntemler ile alternatifler oluşturun diyorlar. Gerçek dünya yukarıda bahsettiğimiz kadar basit değil.

Golden Hammer AntiPattern Çözüm

Örneğin siz Spring Framework kullanıyorsunuz. Bu konuda yılları verip bir birikim yapmışsınız. Bu framework bir çok problemi çözüp veya nelerin çözülemeyeceğini öğrenmişsiniz.

Yeni bir problem var ve bu problem Spring ve B ürünü arasında karar veremiyorsunuz. B ürünü hatta daha uygun gibi gözüküyor. Ama geliştirme sırasında ihtiyaçlarınız daha genişledi ve B ürünü o kısımda yeterli desteği veremiyor ise işte ozaman sorunlar başlar. Bu sefer geçmişte Spring kısmında çözümüne ulaştığınız bir çok konuyu B ürünü içinde yapmaya çalışacak ve çok zaman harcayacaksınız.

İşte Gerçek Dünya;

- Proje için zaman baskısı
- Proje için bütçe ve kaynak baskısı
- Proje için her an değişebilirlik ve yeni yetenek gereksinimleri
- Proje içerisinde turnover rate yüksek olması
- Müşteri ve Pazarın çok hızlı değişmesi.
- Teknolojinin çok hızlı değişmesi
- Regulasyon ve Pandemi gibi etkenler
- vb..

Böyle çevik bir ortamda mühendislerin çözümü genelde hep altın çekiç olacaktır. Yani bildikleri araçları kullanmak. Ama bu paralel bir araştırma, okuma, denemelere engel değil. Belki yeterli bir araştırmadan sonra bu araçların bazılarını değiştirebiliriz. 😃

2nci Gün (Code Quality)

Temiz ve Kaliteli Kod → ~~Kod Kalitesini Neler Kötü Etkiler~~ ~~Clean Code~~

Kodu İyileştirme → ~~Teknik Borç~~ ~~Refactoring~~ ~~Architecture Clearance~~

Kod Geliştirme Yöntemleri → ~~AOP~~ ~~TDD~~ ~~SOLID~~ ~~KISS~~ ~~DRY~~ ~~YAGNI~~

Kodu Dayanıklı Geliştirmek → ~~12 Factor App~~ ~~Reactive Programming~~ ~~Chaos Eng~~

Design Patterns → ~~Design Patterns~~ ~~Creational Patterns~~ ~~Structural Patterns~~ ~~Behavioral Patterns~~

Anti Patterns → ~~Anti Patterns~~ ~~Spaghetti Code~~ ~~Lava Flow Programming~~ ~~Golden Hammer~~ Reinvent The Wheel

Reinvent The Wheel AntiPattern

Tekerleği yeniden icat etmeye gerek yok. Bir problem veya bir sorun olunca onu kendi yöntemlerimiz ile çözmeyi severiz. Ama bu konuyu iyi tartmak lazım. Çünkü kendi çözümlerinizi geliştirdiğinizde, ekiptekilere bir takım aktarımlar yapmanız lazım;

- Dokümantasyon
- Üzerinde çalıştırılan örnekler
- Güncellemeri, çıkan hataları gidermesi, bakımı
- Farklı farklı ortamlarda testlerinin çalıştırılması
- ve bir çok ekstra işi sizin çözmeniz gerekir.

Bu yüzden Tekerleği yeniden icat etmek yerine, çevrenizde birileri bu problemi çözen bir şeyler geliştirmiş mi, bu işin *best-practice* neymiş diye bakmak en mantıklı çözümdür.

Yazılımlar, hatta bulut sistemi ile donanım altyapıları günümüzde çok kolay şekilde **soyutlanabilen ve tekrar kullanılabilen** birimlere dönüştürülmüştür. Aşağıda bu konuya örnekler ile biraz değinmek istiyorum.

Reinvent The Wheel AntiPattern - Çözümü

Soyutlama ve Tekrar Kullanılabilirlik

Kod: OpenSource geldiği noktalar inanılmaz. GitHub bireylerden tutun , en büyük şirketlerin kütüphanelerini açık kaynak olarak sundukları ortamlar. NPM, MVN gibi kod kütüphaneleri bizim bu yapıları alıp ürünlerimize entegrasyonunu inanılmaz hızlandırmış durumda.

Oyun Geliştirme Ortamları: Unity, Unreal Engine vb…Asset Store hazır birçok tekrar kullanabilir kaynak ve altyapılar ile çok hızlı bir şekilde oyun geliştirmenizi sağlıyor. Eskiden oyun motoru geliştirmek için çok uğraşırdık (**Not:** Zamanında uğraşmış olduğum SWEngine ,SWTutorials)

NLP: Yapay zeka konusunda AWS, Google, Microsoft vb büyük firmaların;

Word2Vec, GloVe gibi veri gösterim algoritmaları ve bunların hazır olarak sunulduğu FastText gibi kütüphaneler ile sunuluyor. İlerleyen süreçte büyük şirketler Google(Google AI), Elon Musk kurguğu (OpenAI) ve Salesforce (Einstein AI) ve diğer büyükler arasında Dil Modellemesi ve sonuçta insansı cevap verebilen, insansı dil çıktı oluşturabilen eğitilmiş modeller ortaya çıkardılar. Bu modellerden en son duyrulanı GPT-3 çok büyük verinin(tüm internetin metinlerinin kullanılarak) , milyarlarca parametreyle insan beynindeki bilgi ağına benzer bir yapı oluşturuldu ve Geldiğimiz noktada bu yapılar bize Metin alanında inanılmaz insansı yetenekler sunuyor.

Örnek Data Ortamı: Kaggle içerisinde hazır bir çok ***veri seti*** bulunuyor.

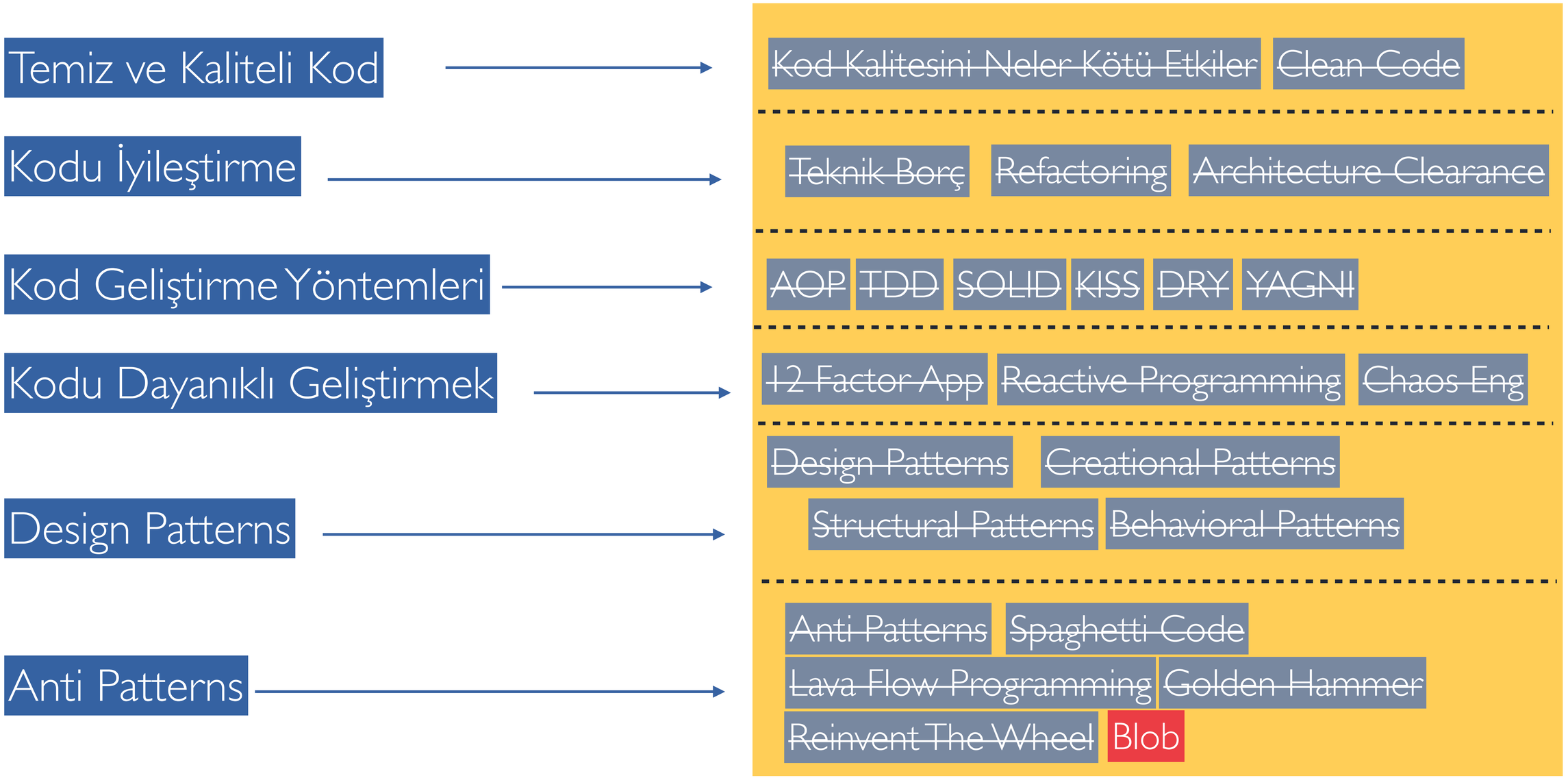
Bulut Hizmeleri: AWS Serverless Servisleri ile Uygulama Geliştirmenin Avantajları yazısında **Managed Servislerin** biz geliştiricilere kazandırdığı avantaj olarak uygulama ve altyapı geliştirme sırasında;

- scalability,
- reliability,
- elasticty,
- resillence,
- security,
- vb..

Kalite yeteneklerini soyutladığını, bize bunları para ile kiralayabildiği bir altyapı sunabiliyor. İşte AWS burada size. *Tekerleği Yeniden İcat Etmek Yerine* API'leri kullanarak kendi işinize odaklanın diyor. Aynısı Microsoft, Google ve Diğer Bulut Servisleri içinde aynı.

SaaS hizmeleride benzer imkanlar sunuyor. Kütüphaneleri direk sisteminize dahil edip sizin yönetmeniz yerine bu işlemleri 3rd Party hizmet olarak sunuyor. Örneğin Auth0, Atlas, Stripe, Mailgun, SendGrid vb sisteminize çok rahatlıkla takıp bir anda production(canlı) seviyesinde hizmetleri sisteminize entegre edebilirsiniz.

2nci Gün (Code Quality)

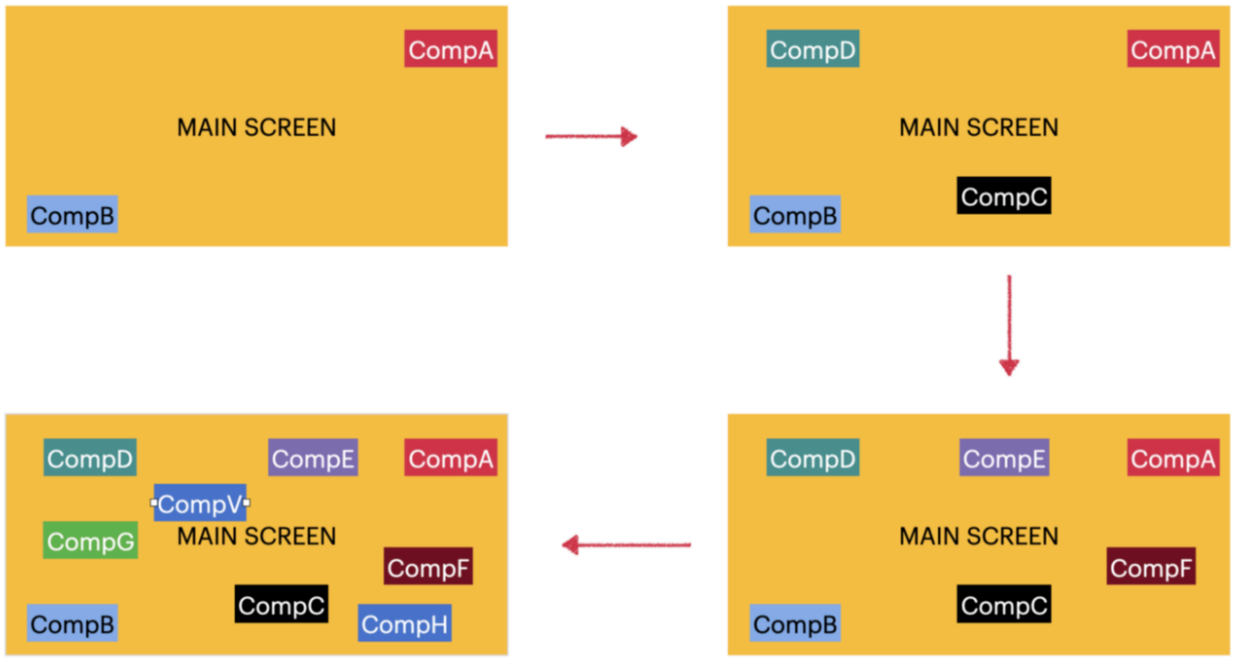


Blob AntiPattern

Blob nasıl oluşur. Öncelikle en çok uğraştığım UI ve Desktop, Web uygulaması geliştirme kısmından örnek vererek konuyu anlatayım.

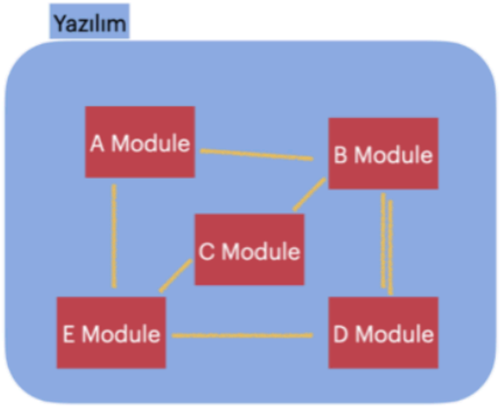
Uygulamalar genelde iterative olarak gelişir. İlk önce 2 bileşeni olan uygulama zaman içerisinde 4, 8, 16, 32 … olarak ilerlemeye başlar.

Not : Yazılım Geliştirme Modelleri yazısı içerisinde Iterative Geliştirme ile ilgili detaylara ulaşabilirsiniz.



Blob AntiPattern

Bizim ilk yazdığımız kodda CompA ve CompB birbiri ile direk konuşmasını istemeyiz çünkü ilerleyen süreçte bu sayı artacağı için bileşenlerin birbirleri ile konuşma arayüzlerini ve bağlantılarını kontrol edemez hale geliriz.



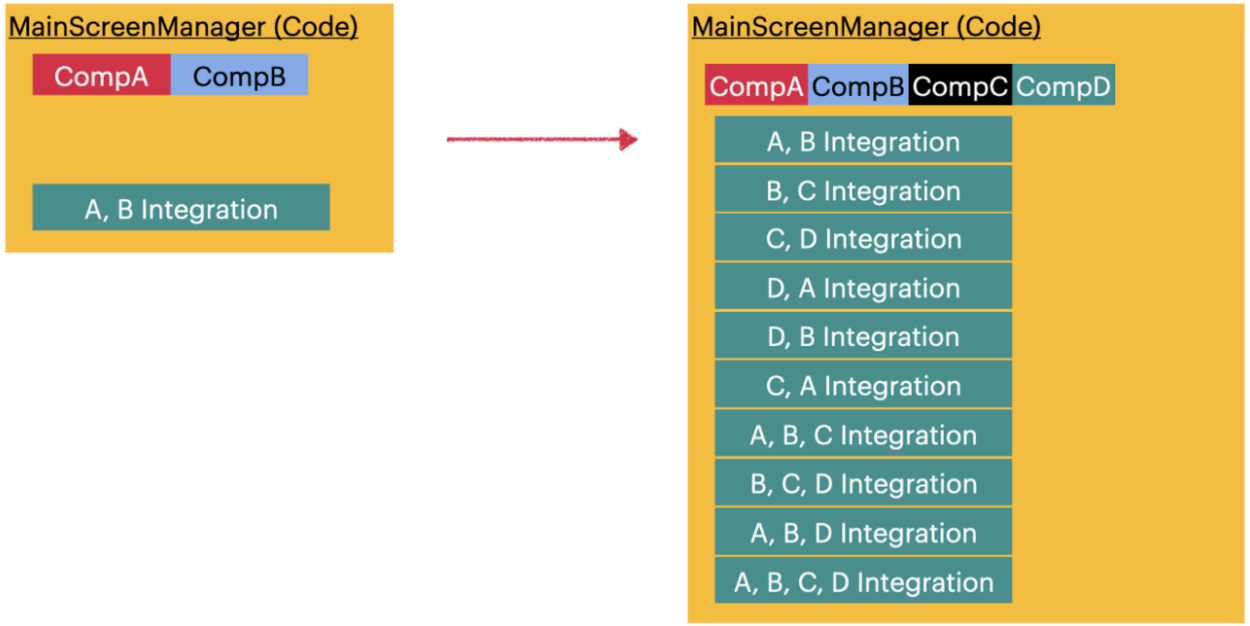
Modüller Direk Birbiri ile Konuşması Durum

Bileşenlerin birbirleri ile olan iletişimi sırasında bu karmaşıklıktan kurtulmak için 2 örüntüden birisini tercih ederiz genelde.

- Observer Pattern
- Mediator Pattern

Blob AntiPattern

Mediator Pattern , Observer Pattern göre daha kolay takip edilebilme ve kod okuması yapılsa bile Mediator Pattern tek bir sınıf üzerinde zaman içerisinde tüm bileşenlerin mantığını tutma ve yönetmeye dönüşürse yukarıda bahsettiğimiz Blob Pattern dönüşür. Aşağıdaki resimde uygulamamız 2 bileşenden 4 bileşene çıktığında, bu bileşenler arasındaki iş kurallarını ve akışlarını MainScreenManager belirliyor ise bu kodun ne kadar büyüdüğünü sizde göreceksiniz. Bileşen sayınız 50, 100 çıktığında artık kodunuz yönetilemeyecek hale gelir.



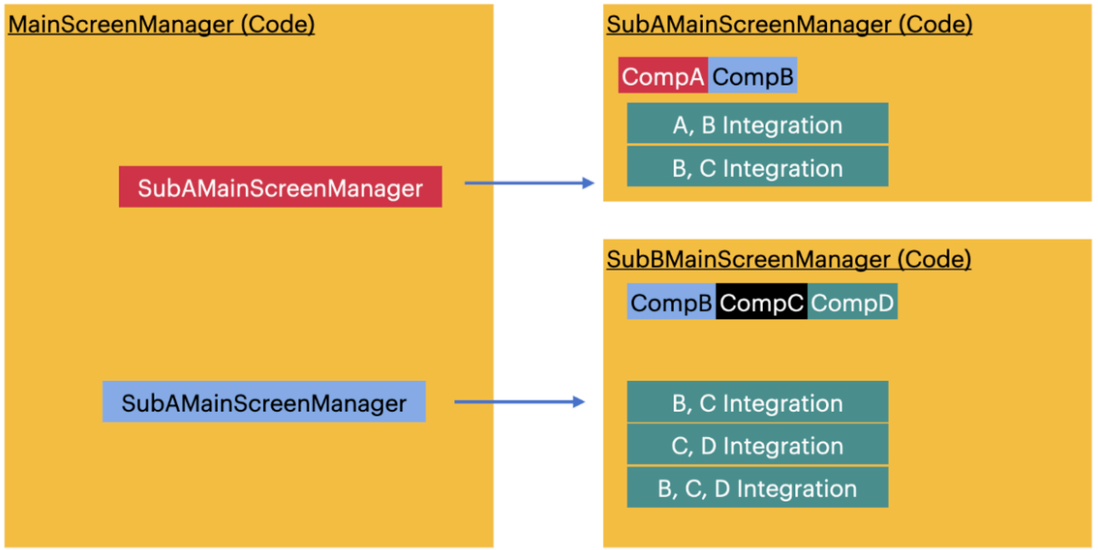
MainScreenManager Kodu Nasıl Karmaşıklaşır

Blob AntiPattern

Peki bu yöntemi neden tercih ediyoruz. Çünkü geliştirmesi çok basit. Tek bir sınıf veya modül içerisinde tüm bağımlılıkları görmek ve birbirine bağlamak çok 2,3 satır kod ile yapabileceğiniz işler. Ama bu yapı 500, 1000+ üzerinde satıra ulaştığında artık kodun bir yumağa dönüştüğünü sizde farkedeceksiniz.

Bu durum da Blob Anti-Pattern gerçekleştirmiş olursunuz. Bu aşamadan sonra tamamen yapısal değişiklikler Observer , Pub/Sub, Event Based yapılara geçmek oldukça zordur.

Yapılacak en mantıklı davranış MainScreenManager sınıfı mantıklı parçalara bölerek, delegation yöntemi ile SubManager yönetmeye çalışmaktır.



SON