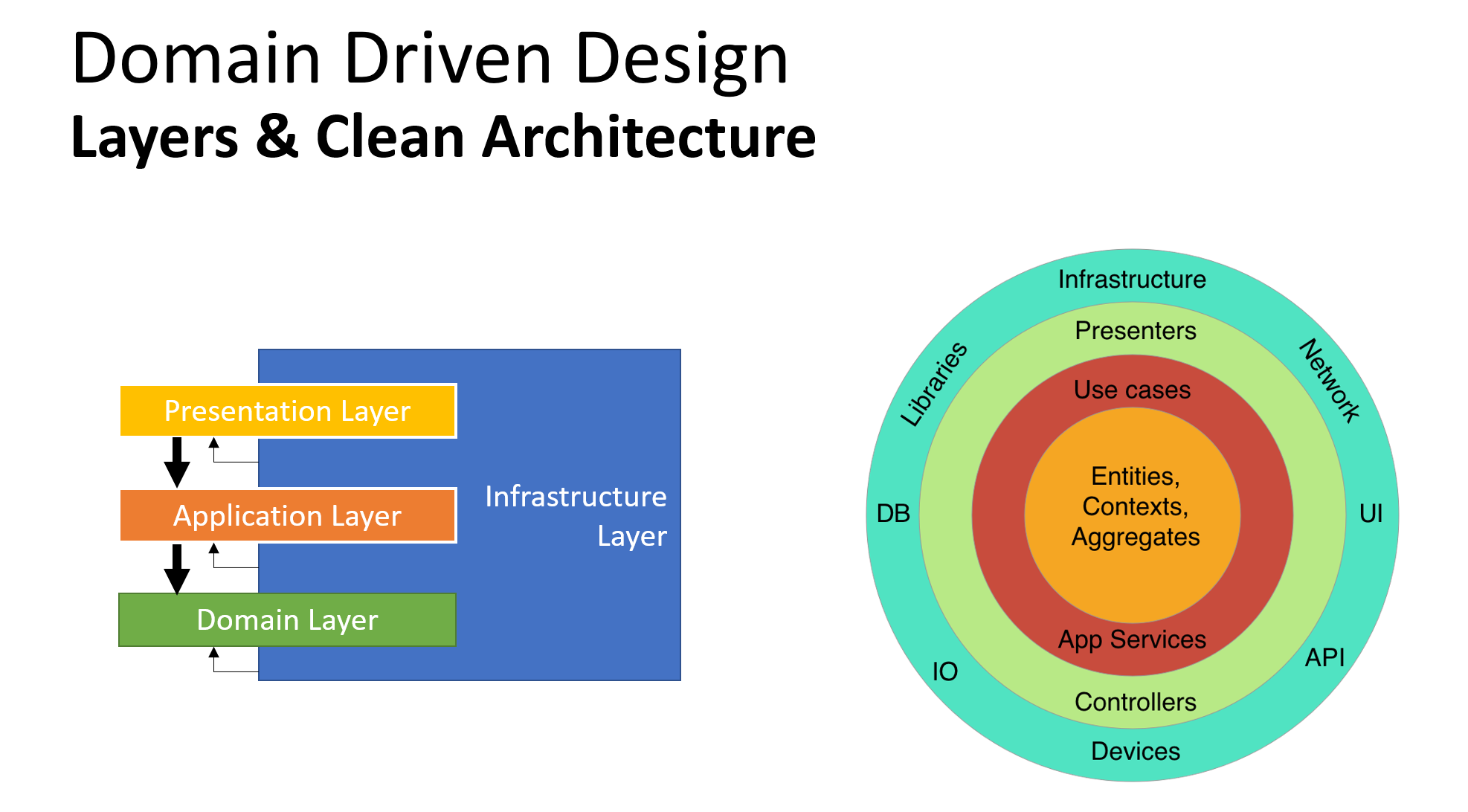
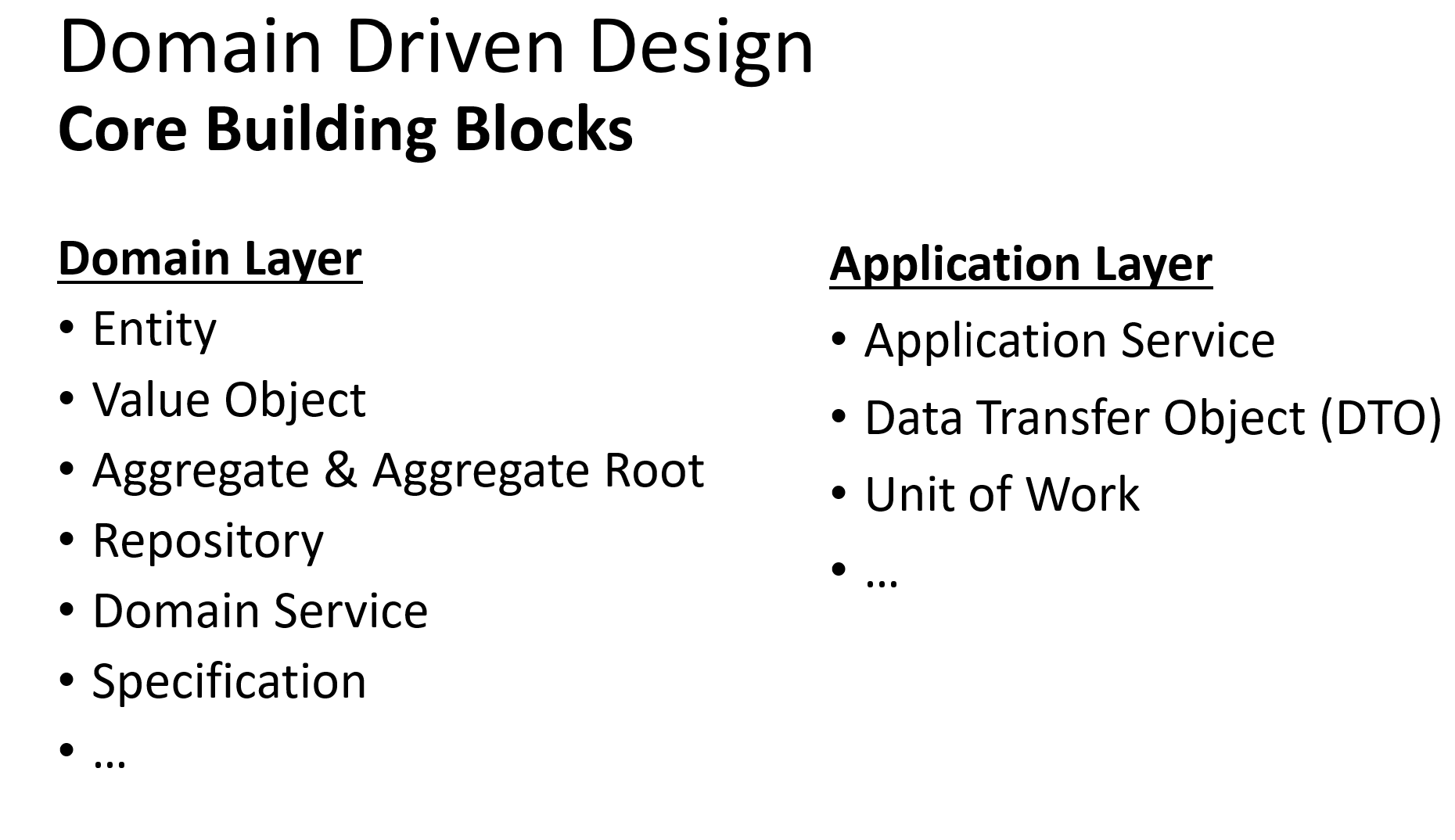
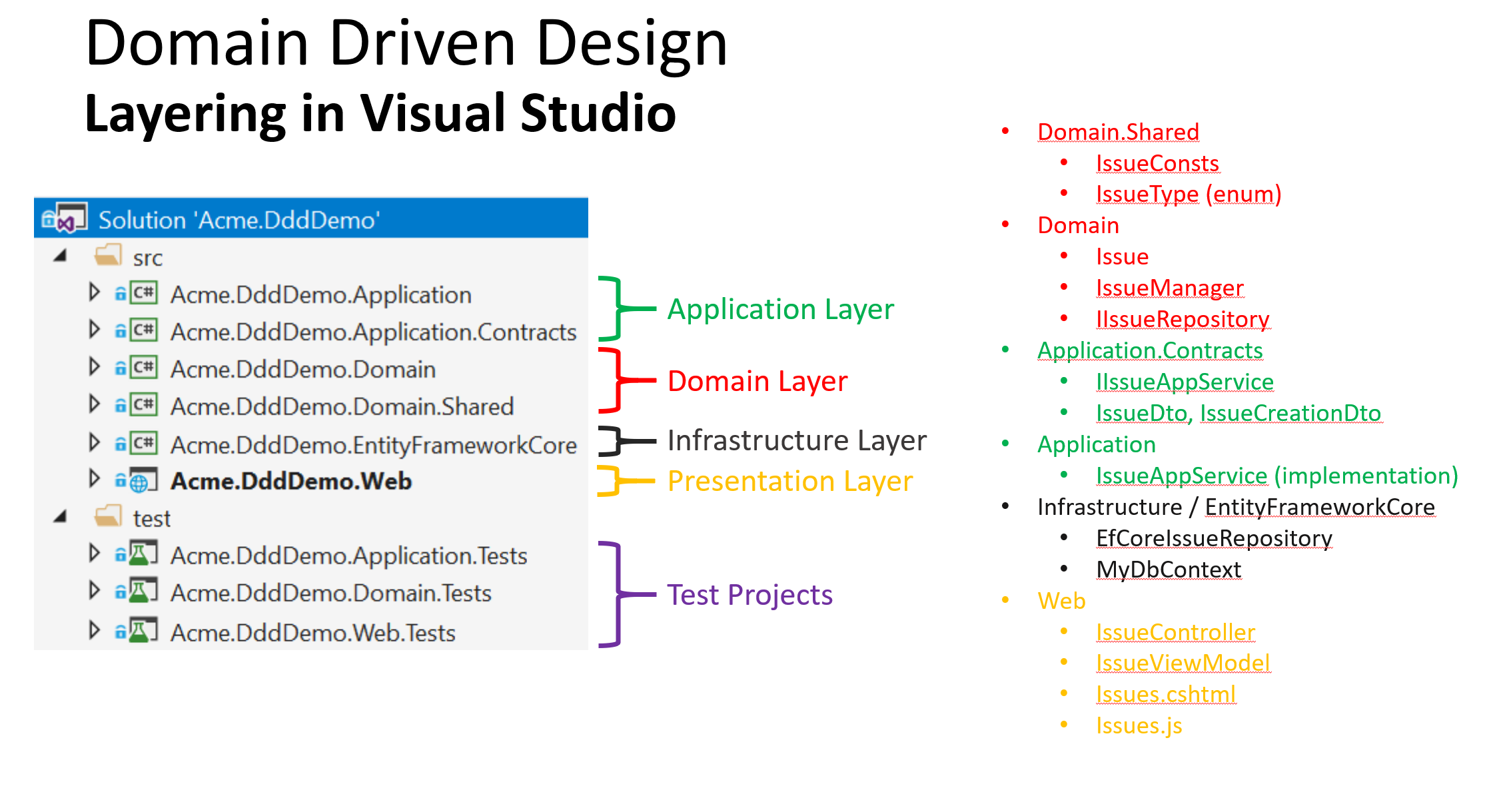
**DDD VE CLEAN ARCHITECTURE ÇALIŞMA NOTLARI**

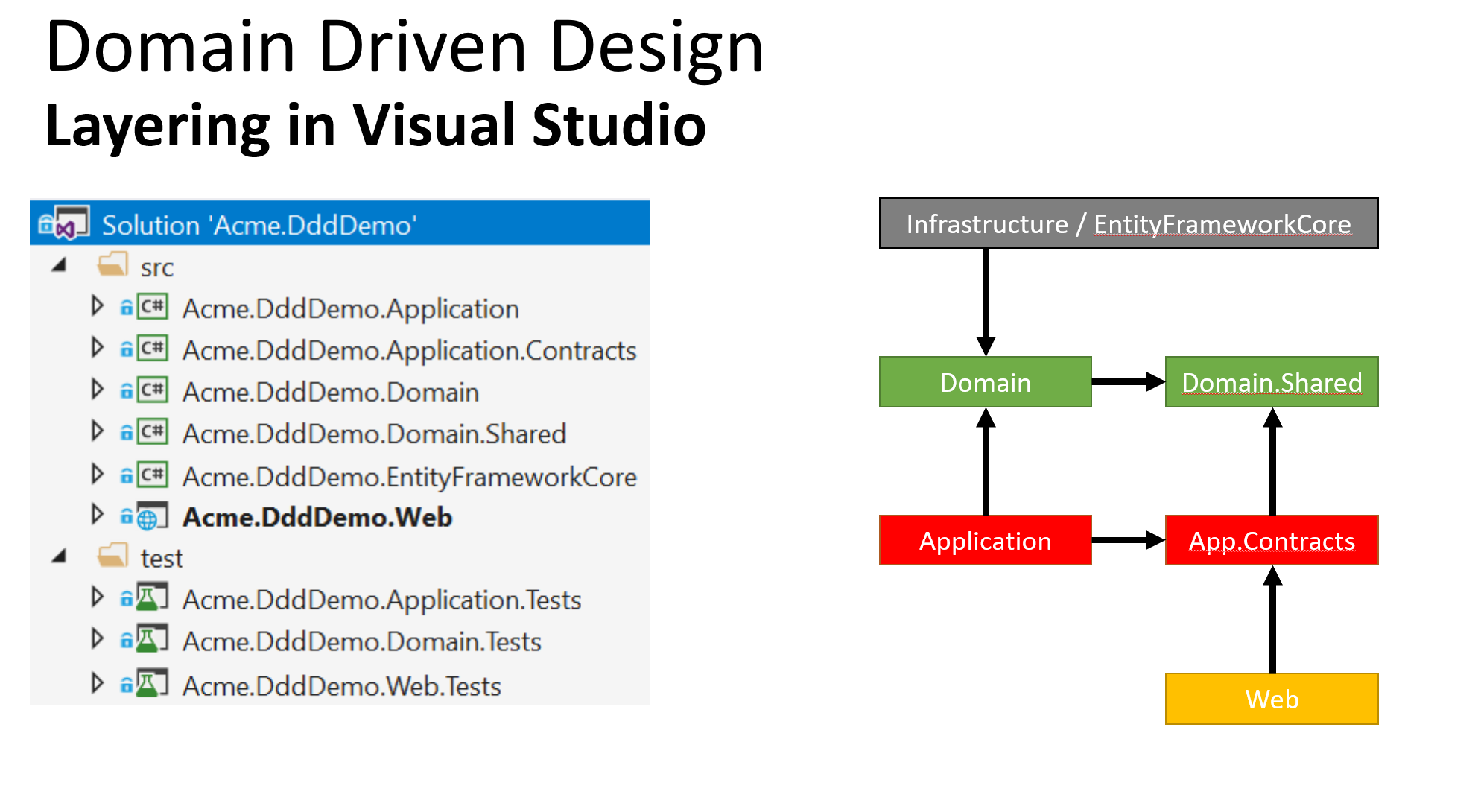
**(Arif EROL - 21.05.2025)**

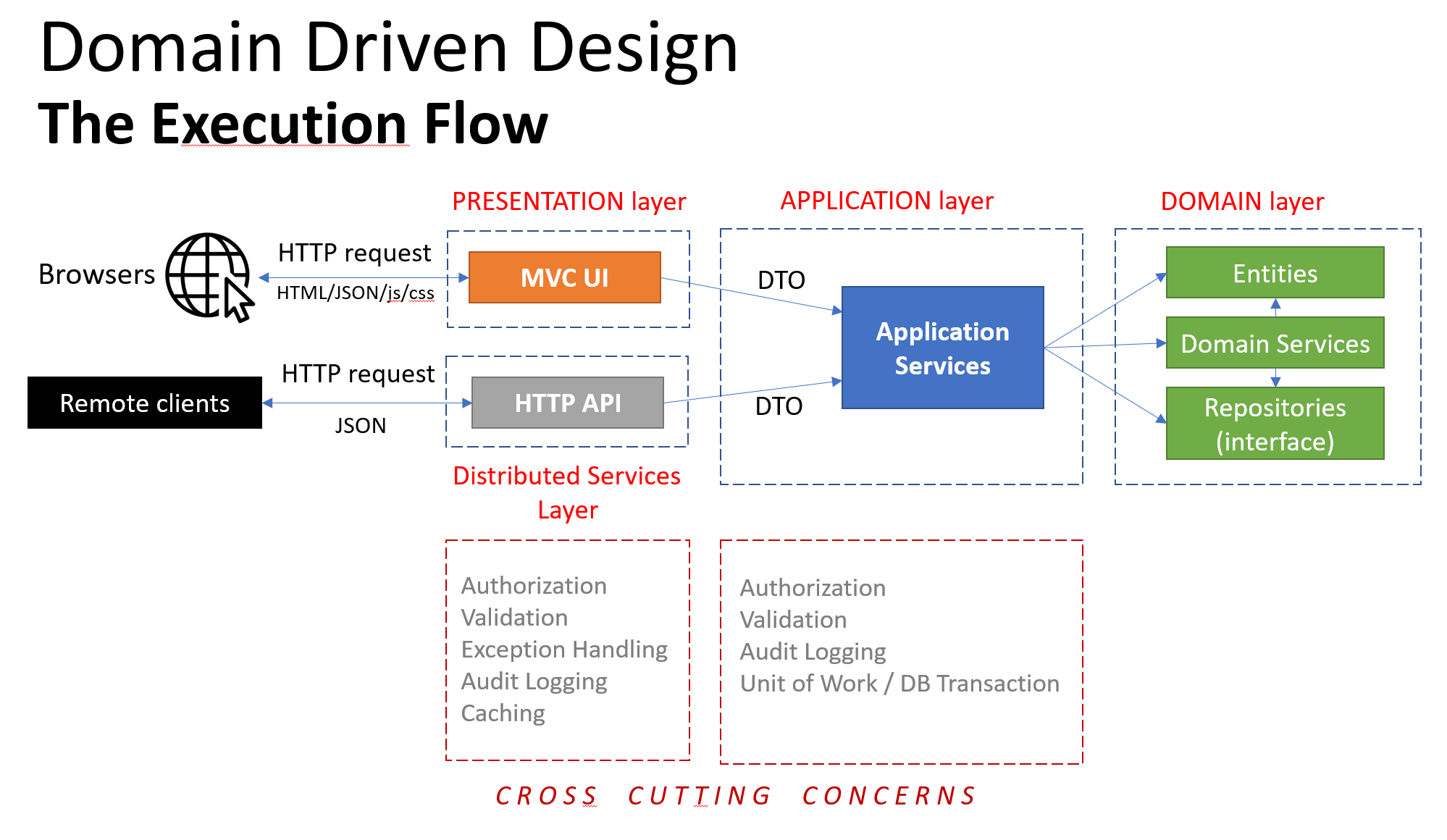


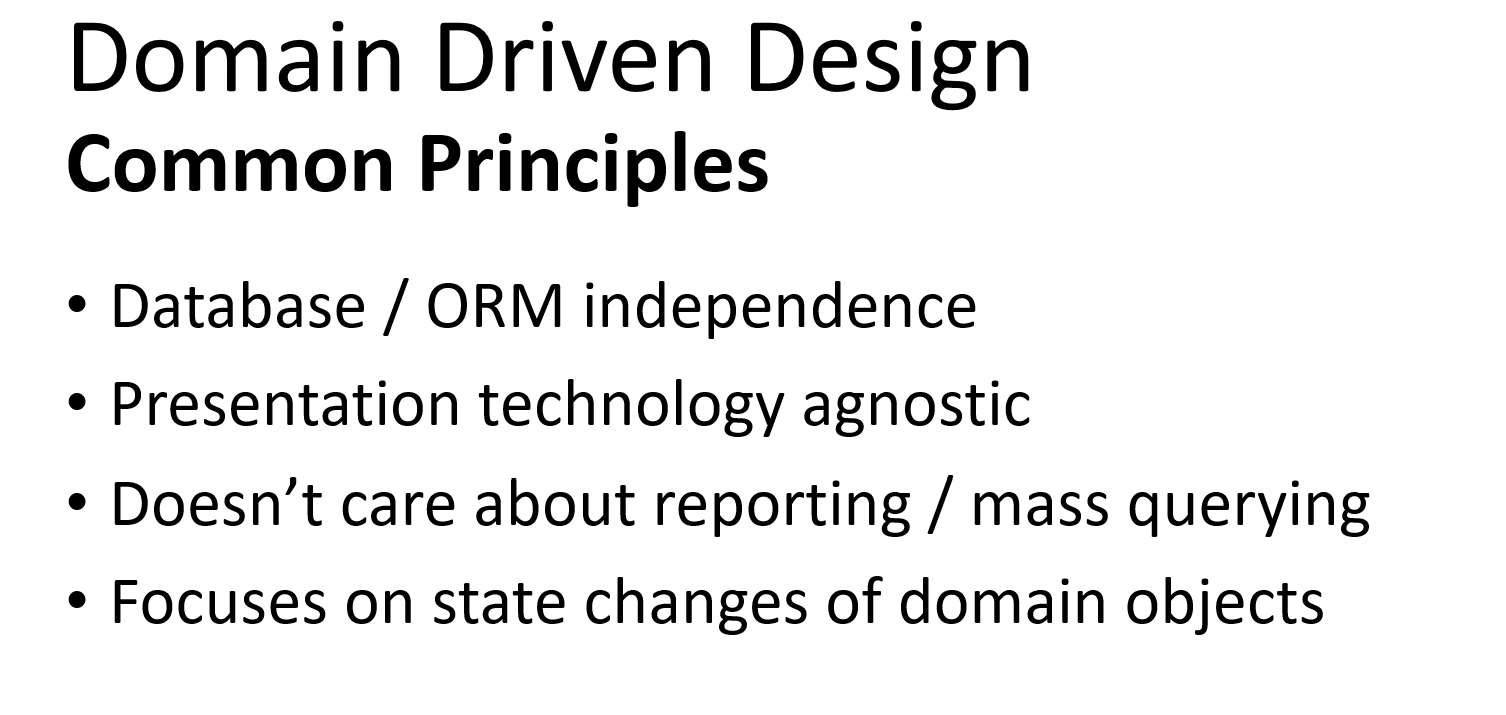
Dört katman ile az bağımlı, kohesif, az değişen, clean bir mimaride yazılım geliştirme yapılmalıdır.

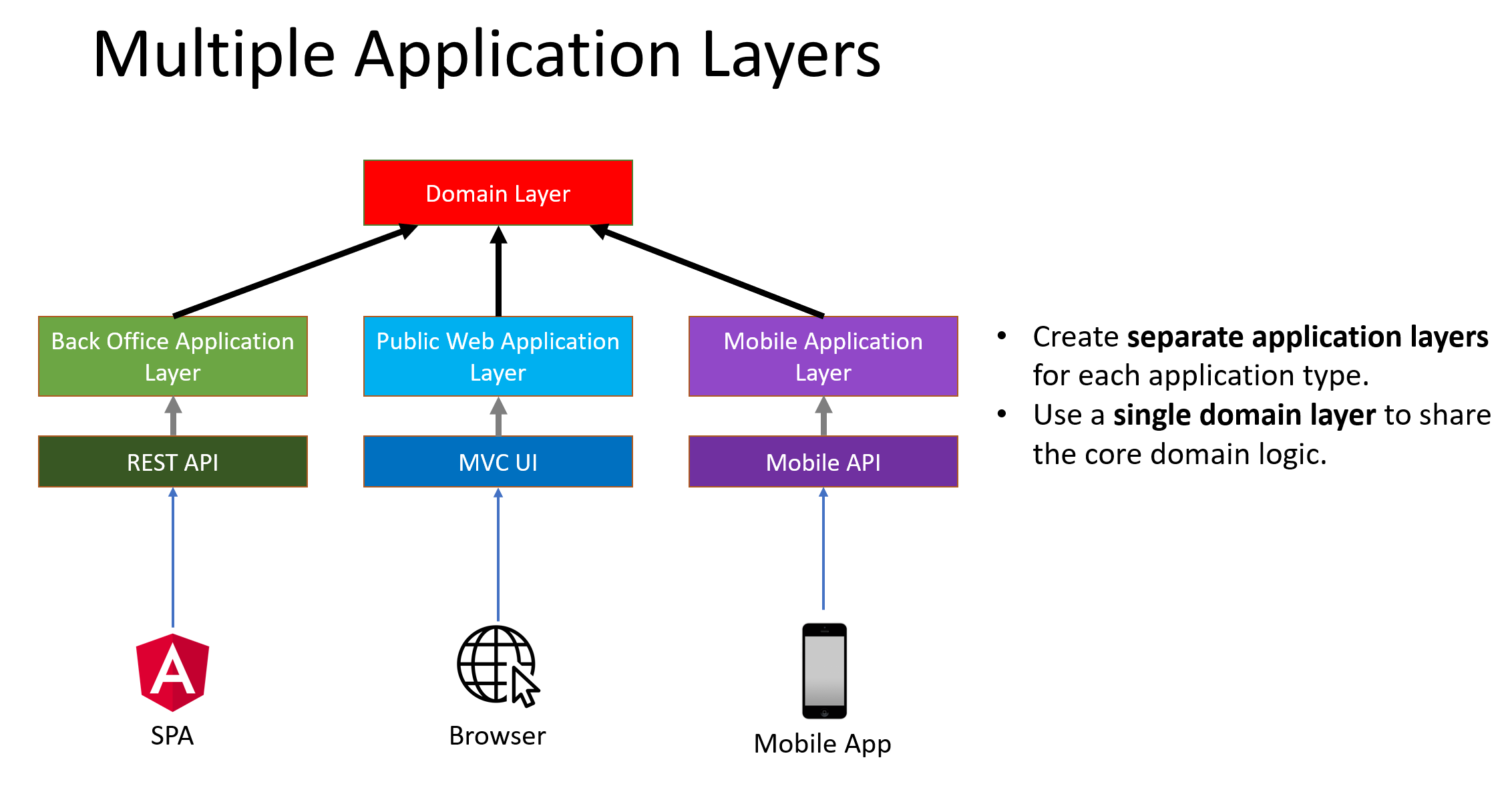












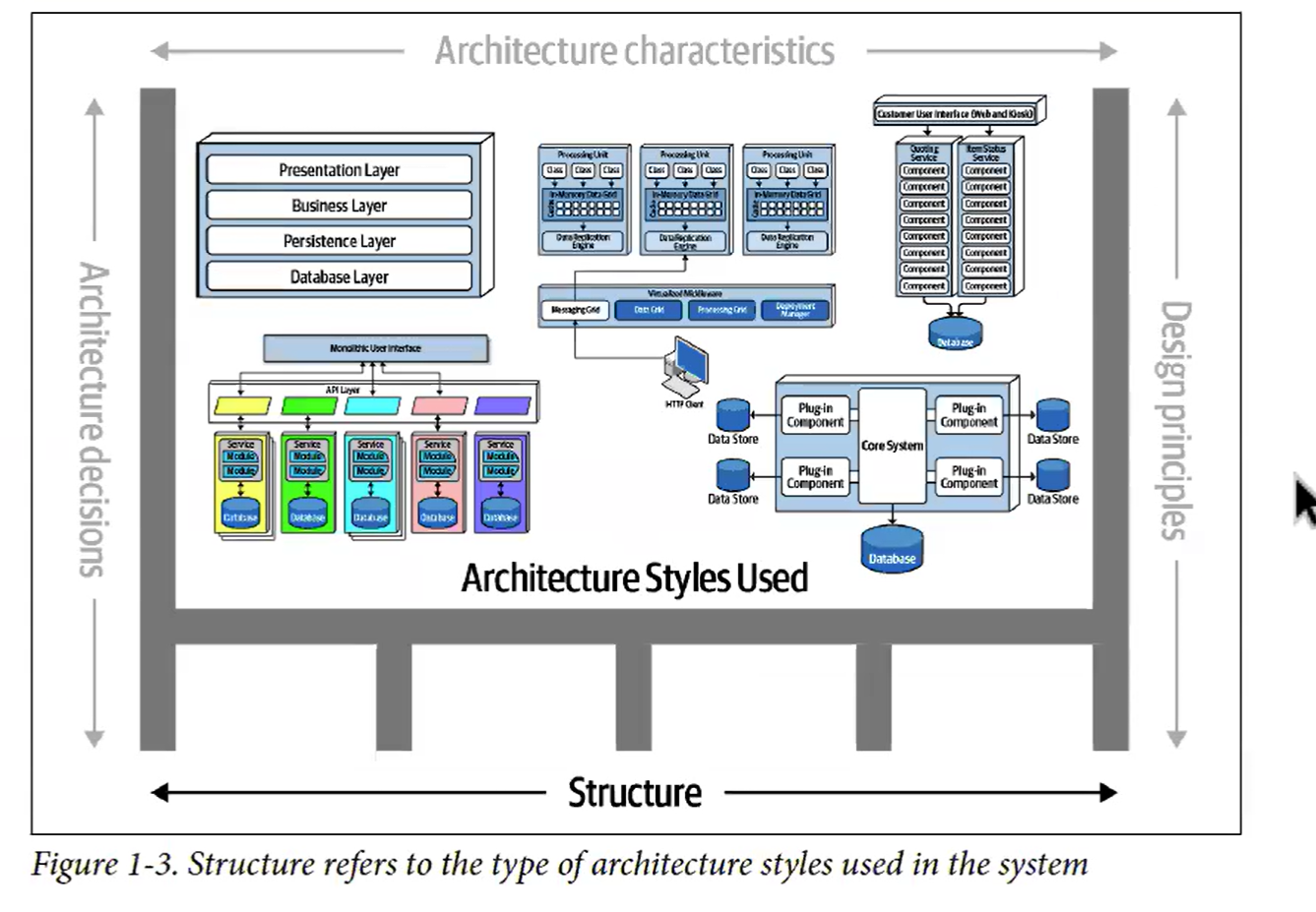
**Kaynak** : <https://github.com/hikalkan/presentations/blob/master/2025-04-25%20Domain%20Driven%20Design/DDD%20Presentation.pptx>

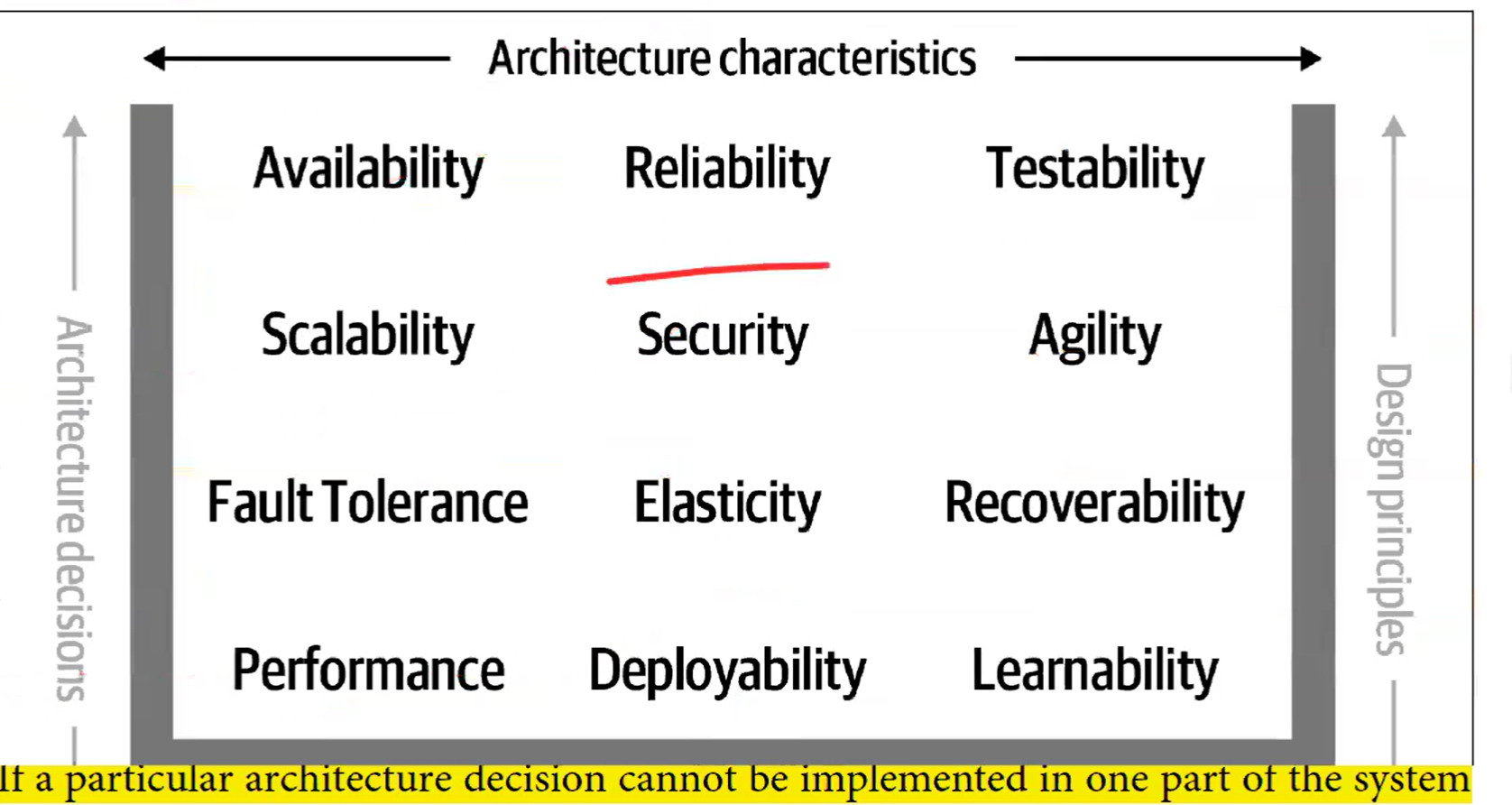
**Software architecture :**

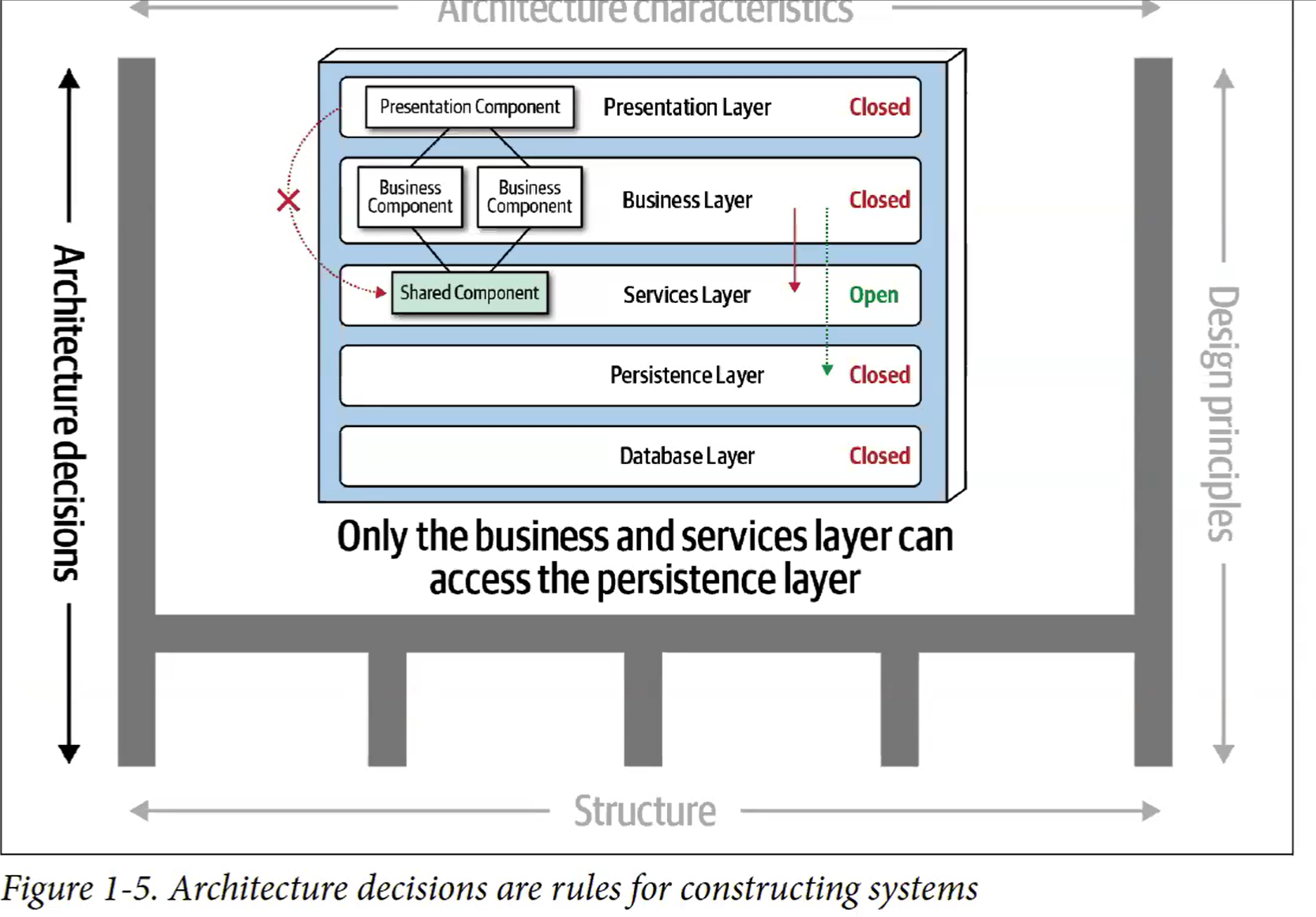
**Functionan ve non-functional requirementlar vardır. Software architecture değiştirilmesi zor, none-functional , mimari requirementlardır. Mimari kararları vermek önemli dir. Zorunlu uyulması gerekir.**

**Structure, Architecture characteristics, Design principles, Architecture decisions**

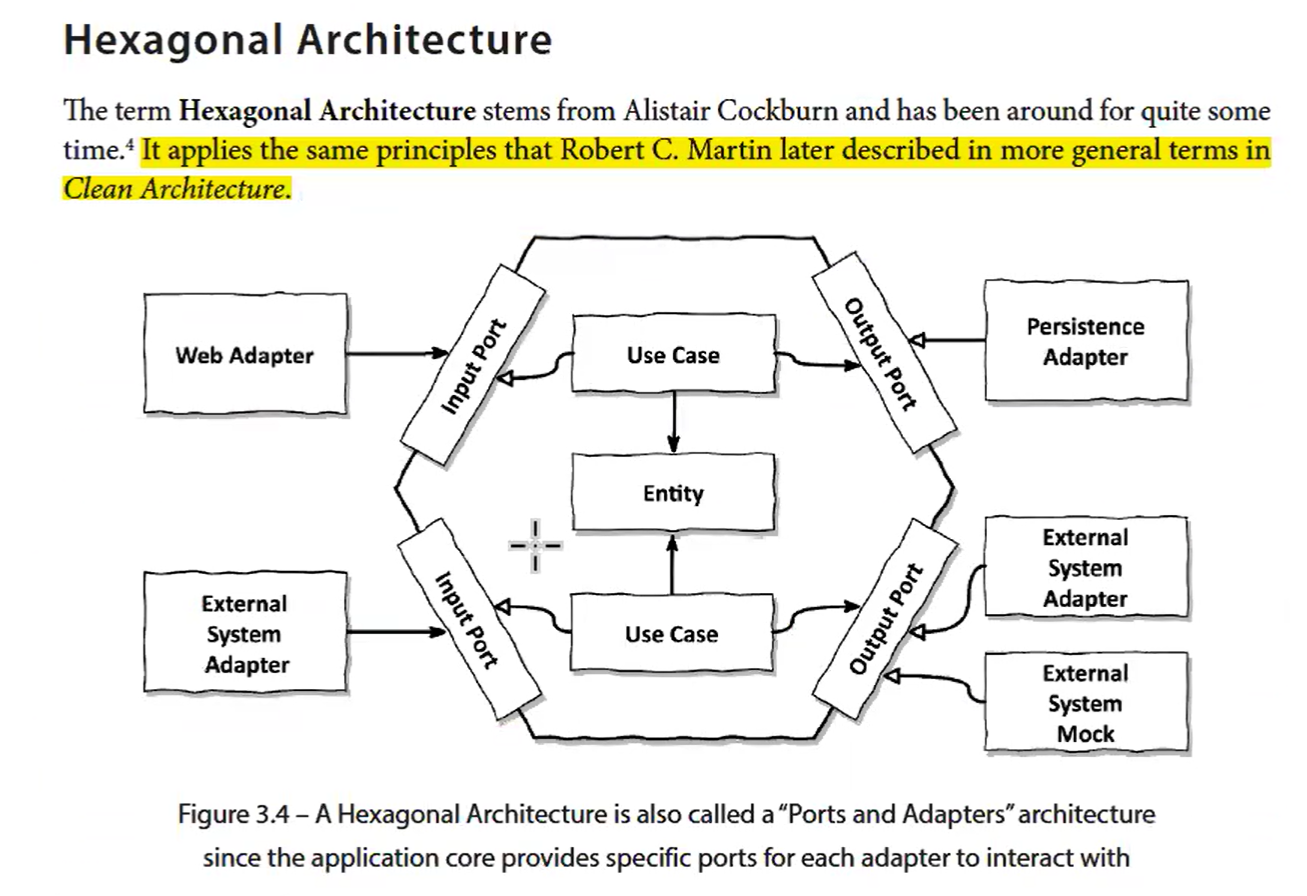
**Architecture characteristics;** \*ability lerdir. Bu \*ability ler none-functional requirementlar, quality requirementları yada architecturel requirementlar olarak adlandırılırlar.

****

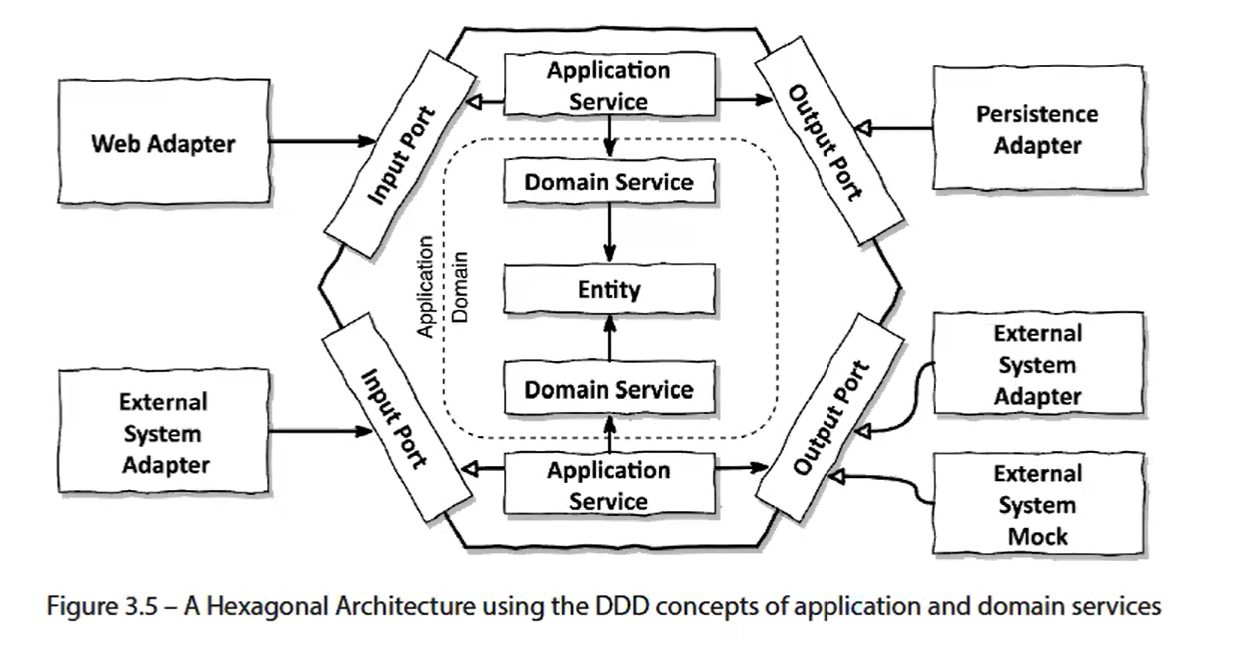
****

****

Hexagonal mimaride sol taraf “in” sağ taraf “out” içerideki “application” dan olabildiğince bağımsız dır. Onion architecture içerideki “application” ile ilgilenir.



Soldaki adaptörler drive(süren) ediyor, sağdaki adaptörler driven(sürülen) ediyor.



Application services ve domain services ler, Domain Driven Design konsept lerini kullanan Hexagonal Architecture yazılım mimarisini kullanan bir çözümdür.

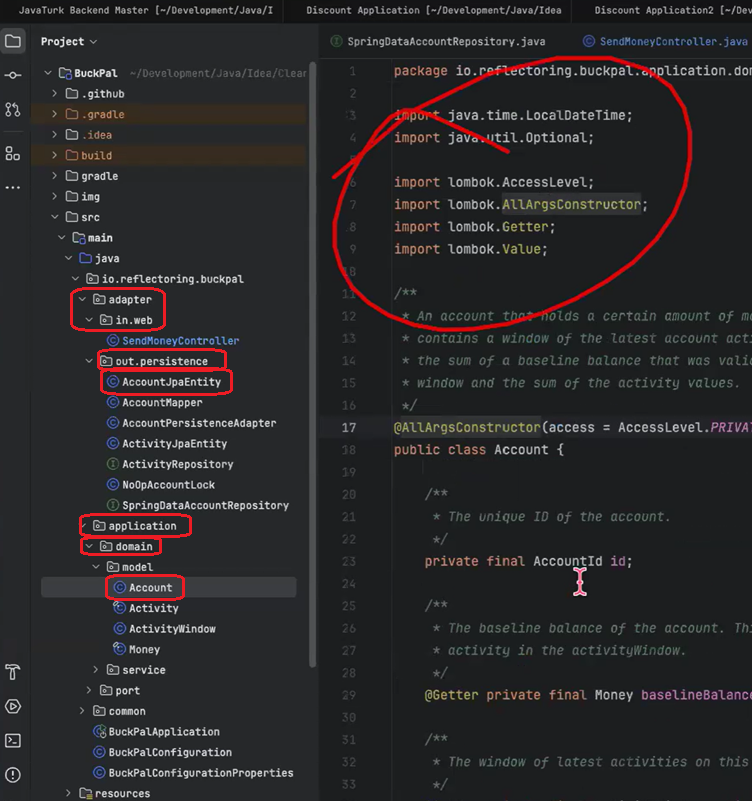
**UseCase(Business Process, User Stories):** User ın belirli bir hedefi gerçekleştirmesi için olabilecek olan bütün durumları , bütün akışları içerisinde barındıran yapıdır. Usecase leri domain layer tarafında implemente ediyoruz.

**Not:** Veri tabanında birden fazla crud işlem yapan businesslogic varsa Transactional scope kullanılır. Anemic kod olmaması için; **Transactional scope Application Service layer da başlatılır. Security işlemleri de Appliation service de yapılır. Domain layer da transational yada security işlemleri yapılmaz!**

**Not:** DDD tasarımına göre kodlarken; önce entity lerde businesslogic(bs) yazılır , sonra ihtiyaç varsa domain servis, sonra ise application servis ler de bs ler yazılır.

**Not:** Domain service; entity ler arasında businesslogic in implemente etmesinden sorumludur. Application service; business logic i çağırır, business logic in öncesi ve sonrasında, security gibi audit loglama gibi, cache gibi işleri yapar.

**Not:** DDD de Entity ile JPAEntity lerini aşağıdaki örnek proje structure ında olduğu gibi ayırmak gerekiyor. JPAEntityMapper larını da “out.adapter.out.persistence” içine koy(Mapper ları çok fazla değiştirilmeyen “infra” layer içine koyma!).



**Not:** Service: Entity olmayıp business logic in parçası olan süreç içeren şey service tir. State i yoktur. Entityler state tutarlar.

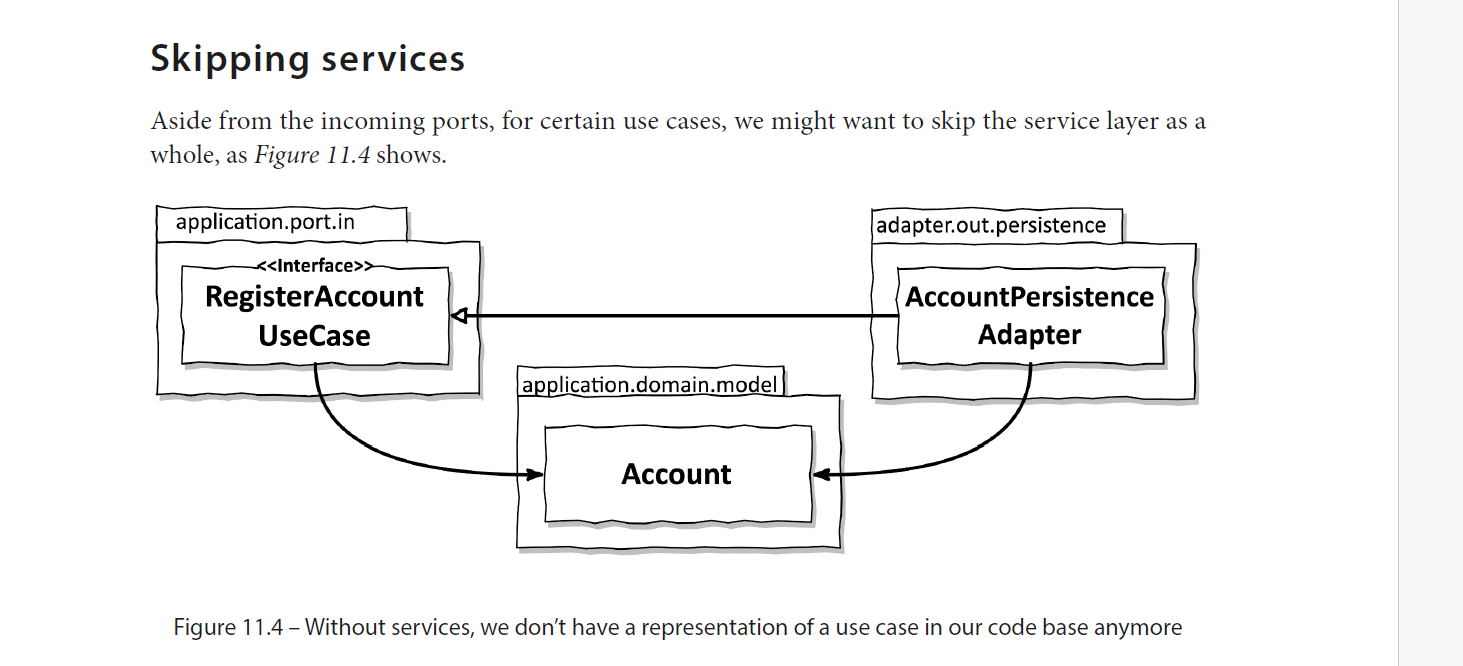
**Not:** Web adapter de(security validations) , app layerda(bs validations) validation ları yap.

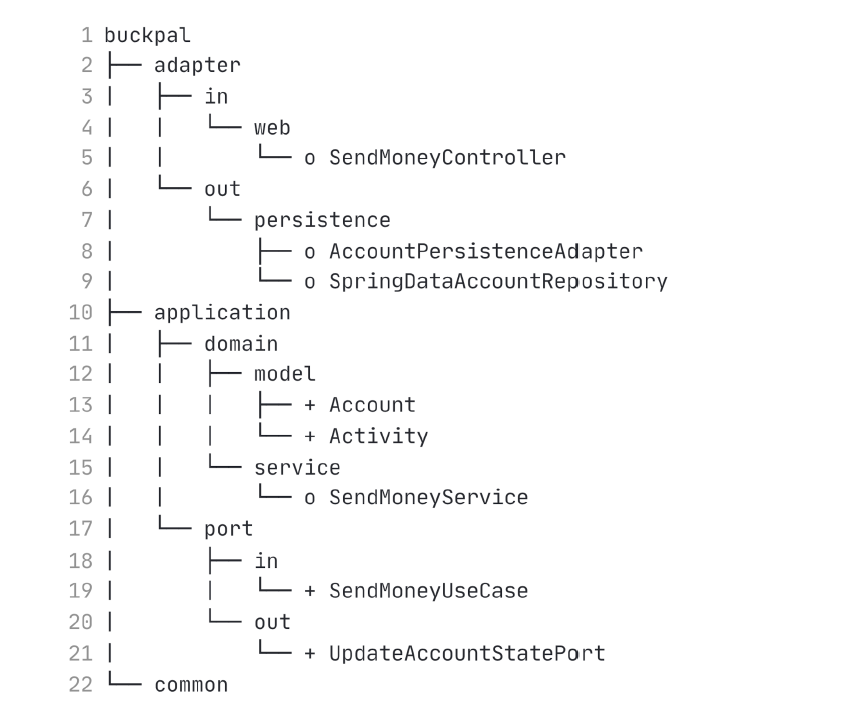
**Not:** Web adapter ında; http request leri objelere maplenir, authorization checks, validate inputs(sanitizations,…),usercase e gönderilecek objeleri oluşturulur, http response u oluşturup dönüş yapılır. Http ile alakalı hiç bir şey diğer katmanlara gitmememeli. Web adapter ında controller ları use case lere göre ayırabiliriz. Bir controller a 15-20 usecase yazıp şişirmiyoruz. Controller ların methodları 5-6 yı geçmemeli. Controller method isimleri verirken CRUD isimlerin yerine BS isimleri veriyoruz örneğin “createAccount” yerine “registerAccount” ismini verin.

**Not:** DDD de hiç bir nesnenin data değişkeni i primitive olmamalı , nesne olmalı diyenler var(Data oriented programming)

**Not :** İf , else kodlarını kaldırmak için strateji pattern ını kullanmayı unutma.(Refactoring)

**Not:** AppLayerda , nesnelerin, domain objelerinin durumunu değiştiren bir business logic yoksa, sadece CRUD işlem yada getData gibi bir usecase varsa, (dostlar domain layerda görsün diyerek-Akin hoca)application layer dan domain layer a gitmeye gerek yok. Bu durumda doğrudan Persistence adapter dan doğrudan verileri çekebiliriz.





**Boiler-plate project structure:**

src/main/java/tr/org/ecommerce/

├── EcommerceApplication.java

├── adapter/

│ ├── in/

│ │ └── web/ # Incoming requests – REST Controllers (driving adapters)

│ └── out/

│ └── persistence/ # Outgoing interactions – Data access layer (driven adapters)

│ ├── adapter/ # Bridges between domain repositories and database implementations

│ ├── entity/ # JPA entities representing database tables

│ ├── mapper/ # Converters between entity and domain models

│ └── repository/ # Spring Data JPA repositories

├── application/

│ └── service/

│ ├── impl/ # Use case implementations (Application Services)

│ └── spec/ # Application service interfaces (Input Ports)

├── common/

│ ├── exception/ # Global/common exception definitions

│ └── util/ # Utility classes (e.g., ID generator)

├── domain/

│ ├── exception/ # Domain-specific exception types

│ ├── model/

│ │ ├── category/ # Category aggregate root

│ │ │ ├── rootentity/ # Category domain entities

│ │ │ ├── policy/ # Category-related business rules (policy interfaces and impls)

│ │ │ └── service/ # Domain services for category logic

│ │ ├── common/ # Shared domain models (e.g., ID value object)

│ │ └── product/ # Product aggregate root

│ │ ├── rootentity/ # Product domain entities

│ │ ├── policy/ # Product policies (business constraints)

│ │ └── service/ # Product domain services

│ ├── port/

│ │ ├── in/

│ │ │ └── rest/ # Port interfaces exposed to external REST adapters (input ports)

│ │ └── out/ # Port interfaces for persistence or external systems (output ports)

│ └── usecase/ # Domain-level use case interfaces

├── infrastructure/

│ ├── config/ # Spring framework configuration (e.g., MapperConfig, JacksonConfig)

│ └── exception/ # Global exception handlers (e.g., ControllerAdvice)

**TESTING ARCHITECTURE**

* Unit test ve Integration testleri whitebox testleridir, developerlar yazarlar, bu testler önleyici testlerdirler. System Testleri blackbox testidir, bulmaya yönelik testlerdir.
* Deploy öncesi test yazılırsa Test Driven Development yapılıyor demektir.
* Arrange Act Assert(AAA pattern i) XUnit test yaklaşımı. Arrange (DB den very getirme) Act(Domain layerdaki BS çalışsın) Assert(Output Port da doğrula)
* BDD (Behavior Driven Development)