

## DevShimre-UnityNotes

### SOFTWARE PRINCIPLES

#### ➤ KISS Principle:

Açılımı; “Keep it simple, stupid” dir. Temelinde sorunların çözümlerinin basit tutulması presnibine dayanan bir mantık kuralıdır. Bir problemi çözerken olabilecek en basit ve yalın çözüm tercih edilmelidir. Zekice bir çözüm sunma kaygısından dolayı karşılaşılan sorunlara karmaşık yazılımsal/algoritmik çözümler üretmek yerine basit çözümlerin daha kullanışlı olduğunu ileri süren prensiptir.

#### ➤ DRY Principle:

Açılımı; “Don’t repeat your self” tir. Yazılım kalıplarının tekrarlanması eleştirir ve buna bağlı ortaya çıkan fazlakalıkların/karmaşıklığın önüne geçmeyi amaçlar. Her bir bilginin sistem içinde tek, anlaşılabilir ve yetkilendirilmiş bir gösterime sahip olmalıdır der. Tek tek benzer satırların veya farklı class’ larda aynı işi yapan algoritmaların zararlı olduğunu savunur. Eğer aynı amaca hizmet eden birden fazla yol varsa bu yollar en nihayetinde kesişecektir ve bu sebeple bunu tekil hale getirmek daha mantıklıdır. Okunabilirlik, bakım kolaylığı, sürdürülabilirlik, test edilebilirlik ve tekrar kullanılabilirlik en temel artılarıdır. Aynı şeyi birden fazla kez yapmanın, bunlardan birini kontrol etmek istediğinizde diğerini de hatırlama gerekliliği gibi durumları ortaya çıkarması da bir diğer kaygısıdır.

#### ➤ S.O.L.I.D Principles:

- **Single responsibility principle:** Her bir class’ ın veya modülün tek bir sorumluluğu olmalıdır. Unity özelinde component tek bir işlevsellikten sorumlu olmalıdır. Bu durum ilgili class’ ın sürdürülebilirliğini sağlar. Aksi durumda class tıkitış tıkitış bir hale gelir ve minik değişiklikler dahi içinden çıksamaz hale geleceği için yeni eklemeler yapılamaz. Classların birbirleri arasındaki iletişim ise action veya delegateler ile sağlanarak bağımlılık olmadan ilgili classlar, kendilerini tutan bir yönetici tarafından yönetilir. Böylece yönetici class da modülerlik kazanmış olur.
- **Open-Closed principle:** Bir class kendisini genel manada korurken aynı zamanda modifikasyona ve geliştirilmeye açık olmalıdır. Bu noktada interface ve inheritance kavramları önem kazanır.
- **Liskov substitution principle:** Kodlarda değişiklik yapmadan alt sınıflar türedikleri üst sınıf yerine kullanılabilmelidir. Böylece kodlarda değişiklik yapmadan alt sınıfta üst sınıf yerine kullanılabilir. Unity özelinde inheritance noktasında alt sınıflar aynı şeyi farklı şekillerde algılıyorlarsa, üst sınıf için değişiklik olmadan kendine özel değişikliklerini override ettikleri method içerisinde halletmeliler.
- **Interface segregation principle:** Sorumluluklar tek bir interface veya class’ta değil görev başına bir adet olacak şekilde yapılmalıdır. Aksi durumda classlar kullanmadıkları özelliklere sahip olmaya zorlanmış olurlar.
- **Dependency inversion principle:** Dependency en basit hali ile bir class’ ın diğer bir class’ ı kullanması yani ona işleyişinde bağımlı olmasıdır. DI çözümü için class’lar arası bağımlılılıkların az olması gereklidir üst seviye sınıfların çalışabilmek için alt seviye

sınıflara bağımlılığı olabildiğince az olmalıdır. Bunun için de üst seviye sınıflar alt seviye sınıfları somut ve katı halleri ile değil, olabildiğince soyut olarak tutarlar. Kısacası bir classın görevini yerine getirirken başka bir classa bağımlılığı olmamalıdır ya da olabildiğince soyut olmalıdır. Dependency çözümü constructor içerisinde doldurmaktır. Fakat unity özelinde monobehaviour içerisinde bir constructor yoktur ve bu da bu durumu engeller. Bu yüzden properety injection ile çözülebilir veya da initialize metodu constructor gibi davranışabilir. Fakat handler classlar monobehaviour olmadığı için constructor içerisinde readonly fieldlarını setleyebilirler ve böylece bağımlılık azaltılmış olur. İlgili class sahibi olduğu tek sorumluluğu, constructorunda, awake veya initialize methodu içerisinde oluşturacağı handlerlarını yöneterek yaparsa ve bu noktada kendi aralarındaki iletişimleride onları construct ederken sağlar ise, dependency minimize edilmiş olur yani onlara bağımlılıklarını enjekte etmiş olur. Interface kullanımı bu noktada soyut bağlantı için önemlidir. Kısacası class bağımlılığı kendisi almamalıdır, ona enjekte edilmelidir. Böylece class diğer classın nereden geldiğini umursamaz. DI sayesinde loosely coupled ilkesine de uygun kodlama yapılmış olur. Bu prensip, sürdürülebilirlik ve modülerlik kavramları ile yakından ilgilidir çünkü class katı bağlılara sahip olmadığı için içeriği zenginleştirilebilir veya da azaltılabilir bir hale gelir. Ortaya çıkıştı c++ taki memory leak kaygısındandır. Instance edilen her class tutulmalı ve gerektiğinde yok edilmelidir ki memory leak olusmasın. C# ta ise garbage collector vardır ve unity motoru c# ta yazılan kodda hiçbir referans kalmazsa garbage collectoru çalıştırır ve garbage temizlenmiş olur. DI da iki çeşit enjekte etme yöntemi vardır. İlkı contructorda yapmak, diğer ise resolve injection yani istenilen framede enjekte yapmak. Unity özelinde GameObject.FindObjectsOfType sahne contexti resolvelidir ve GameObject.GetComponent ise gameobject contexti resolvelidir. Bu durum ise ECS ye geçişti engelliyor ve performansı düşüren bir durum. Unityde Zenject kullanımının sebebi ise DI' yi en yukarıya yani constructor'a almaktır. Awake esnasında GameObject.FindObjectsOfType ile DI yapılsa dahi, projenin açılma süreleri çok uzun sürebilir çünkü bu static method, sahne contextindeki her objeyi tek tek kontrol ederek sonuç vermektedir. Keza GetComponentta ilgili gameobjectin ne kadar componenti varsa hepsini tek tek kontrol ederek sonuç vermektedir. Zenject gibi 3rd party frameworkleri ise kendi containerleri içinden bu aramayı awakeden de önce ateşlenen unity eventleri (OnSceneLoaded etc.) ile yaptığı için çok daha performanslı sonuçlar vermektedir. Unity özelinde iki objenin fiziksel temasında birinin diğerini tag ile bulup GetComponent ile class'ına erişmesindense karşı tarafın sahip olduğu interfaceye(IInteractable etc.) erişmesi daha doğrudur. Böylece karşı tarafla soyut bir bağlantı kurulmuş olur. Bu tarz bir iletişim yerine, observer pattern da tercih edilebilir çünkü soyut bağlantı kurulsa dahi GetComponent denileceği için optimizasyon karşıtı bir kontrol süreci başlamış olur. Kısa bir ifade ile DI, her seferinde yapılan tek tek kontrolleri "ki bu durum toplamda çok fazla işlem yapılması demektir", tek seferde yapmak ve ihtiyaçları enjekte etmektir. Bir class componentleri tutuyor ve ihtiyaç duyanlara ihtiyacı olan componenti verebiliyorsa orada bir DI vardır.

## DEPENDENCY MANAGEMENT SOLUTIONS

### ➤ Singleton And Encapsulation:

Singleton ile tekilik garanti edilir ve dependency çözümünde herkesin erişip bağımlılıklarını ordan alması için kullanılır. Fakat singleton kullanımı god class riski yarattığı gibi mimari noktasında dependency' i çözmek isterken daha fazla dependency' e sebep olabilmektedir. Encapsulation (Kapsülleme) ise, bir class' taki özellik veya işlevleri başka class' lardan saklamaya yarar. Private field' lar zaten başka class' lar tarafından erişilemez ve değiştirilemezler fakat encapsulation, bu private field' ı bilinçsiz kullanımından korumamızı sağlar. Encapsulation, başka class' ların erişimini ve bu erişimin düzeyini belirler yani sadece erişim değil güvenlik görevi de vardır. C# ta encapsulation property ile yapılır. Get ile veri okuma Set ile de yazma ve gerekli durumlarda belirli işlemlere göre yazma işlemleri uygulanır.

### ➤ Services Locator:

Singleton' un aksine singleton yapılcak class'ların hepsini merkezi kayıt defteri ile tekil ve global yaparak bütün singleton'ların tek bir yerde tutulmasını sağlar. Bu singleton' lara service ismi verilir. Merkezi kayıt defteri/ServicesLocator ise bu service'leri çektiğimiz singleton class' tır. İçinde get ve register metodları olur. Singleton' un aksine IService' den türeyen class'ları çektiğimiz için bu class'ların ne olduğu çağrıran class'ı ilgilendirmez. Bu sebeple Singleton' dan daha iyi bir pattern/yöntem olsa da bağımlılıklarımızı almak için yine bir aracıya yani service locator' e ihtiyaç duymaktayız. Daha iyi olan yöntem ise Dependency Injection yani bağımlılığın hiçbir aracı olmadan bir nevi büyü gibi isteyen class'a ulaştırılmasıdır ki bu noktada "Dependency Injection" framework'leri olan Zenject ve Strange IoC devreye girmektedir.

### ➤ Zenject:

Dependency injection için üretilmiş 3rd party bir framework' tür. Dependency classlarının container'lanması ve ihtiyaç duyulması halinde ihtiyaç duyana enjekte edilmesi prensibine dayanır. Container ve Installer olmak üzere iki temel kavram vardır. Container enjekte etme görevini yerine getirir. Installer ise bind işlemlerinin yapıldığı yerdir yani kimin enjekte edileceğinin yönetildiği yerdir. Project context, scene context ve gameobject context olmak üzere 3 context vardır. Project context bütün sistem açık olduğu sürece çalışır ve "Resources" klasörünün içinde olmalıdır. Scene context o sahne var olduğu sürece çalışır ve her sahnede bir tane olmalıdır. Benzer durum gameobject context için de geçerlidir. Context' lere installerlar eklenir. Installer'lar mono, scriptable veya prefab olabilir ve üçü de aynı işi yapar sadece mono, prefab veya scriptable olmalarından gelen kullanım ayırmaları vardır.

- **Injection:** Injection işlemi için [Inject] attribute'si kullanılır. Sadece constructor'a sahip class' larda constructor içinde doldurma yapıldığında herhangi bir attribute kullanmadan sadece o bağımlı olunan class'lar bind edilirse enjeksiyon gerçekleşecektir. Türleri;

a) Field:

```
public class Foo
{
    [Inject]
    IBar _bar;
}
```

b) Property:

```
public class Foo
{
    [Inject]
    public IBar Bar
    {
        get;
        private set;
    }
}
```

c) Constructor (Recomended):

```
public class Foo
{
    IBar _bar;

    public Foo(IBar bar)
    {
        _bar = bar;
    }
}
```

d) Method:

```
public class Foo
{
    IBar _bar;
    Qux _qux;

    [Inject]
    public void Init(IBar bar, Qux qux)
    {
        _bar = bar;
        _qux = qux;
    }
}
```

- **Bindings:** Bağımlılıklarımızı bağlama işlemidir. Inject attribute' si ile çağrılan dependency' lerin bind edilmesi gereklidir. Çok fazla bind çeşidi vardır. Transform kontrolüne bağlı, priority vererek, koşula bağlayarak, opsiyonellik vererek veya id vererek etc. Bu noktada bazı terimlerin de açıklanması gereklidir. Bu terimler;
  - a) *AsSingle*: İstenilen class sadece bir kez instance edilir. Cahced' en farklı her context için tekiliği sağlamasıdır.
  - b) *AsCached*: İstenen class daha önce istendiği ise eldeki verilir yoksa yeni instance edilir.
  - c) *AsTransient*: Her istemde yeni bir instance yaratılır.

- d) *NonLazy*: Default olarak ilk istenildiğinde instance yapılır ve buna "Lazy" denir fakat bu seçenekle sistem ilk kurulduğunda bir instance üretilir.
  - e) *Contract type*: Bağımlı olunan class.
  - f) *To*: Bağımlılık çağrııldığında çağrıranaya verilen dönüş.
  - g) *ID*: İsteğe bağlı olarak bind' lara id atanması mümkündür. Aynı contract type' a sahip bind' larda işlevlidir.
  - h) *Construction method*: Default olarak FromNew() yapılır fakat isteğe bağlı olarak farklı türleri de vardır.
- <https://github.com/modesttree/Zenject#construction-methods>
- i) *Scope*: AsSingle, AsCached, AsTransient' lerdır. Birden fazla Inject' te ne yapılacağını belirler.
  - j) *Arguments*: Yeni bir resulttype instance' si yaratılırken kullanılacak objeler listesi.
  - k) *InstantiatedCallback*: Bir bind sonucunda ilgili nesneye ihtiyaç varsa yararlı olan bir observe işlemi.

```
Container.Bind<Foo>().AsSingle();
Container.Bind<IBar>().To<Bar>().AsSingle();
```

```
Container.Bind<ContractType>()
    .WithId(Identifier)
    .To<ResultType>()
    .FromConstructionMethod()
    .AsScope()
    .WithArguments(Arguments)
    .OnInstantiated(InstantiatedCallback)
    .When(Condition)
    .(Copy|Move)Into(All|Direct)SubContainers()
    .NonLazy()
    .IfNotBound();
```

- **Installers**: Scriptableinstaller, monoinstaller ve prefabinstaller olmak üzere üç türü vardır fakat üçü de aynı işlev sahiptir. Bind işlemlerimizi bu class'ların içinde yaparız.
- **Context types**: Scene context, gameobject context ve project context' lerdır.
- **Non-Monobehaviour classes**: IInitializable, ITickable gibi interface'ler ile monobehaviour olmayan class'ları da game update' ye bağlamak veya genel game cycle' ye bağlamak açısından elverişlidir. BindInterfacesAndSelfTo<class> bind türü sayesinde hem class'ı hem de interfaceleri bind edebiliriz. BindInterfacesTo ile de class'ı gizli tutarak bind yapmak mümkündür.
- **Signals**: Zenject'in observer pattern'ıdır. A ile B class'larının direkt birbirlerini bilmeden "Signal" üzerinden haberleşmeleri prensibine dayanır. Container.DeclareSignal ile bağlanır ve signalBus.Fire ile de ateşlenir. RunSync default bir değerdir ve bütün subscribe' lere aynı anda haber gider. Async ise verilen tick priority sırasına göre hareket eder.
- **Complex bindings with sub-containers**:

- **Factory:** Runtime' de gerçekleşen instanceleri de bağımlılık kurallarımıza dahil etmek içindir. “FactoryInterface” class' i “PlaceHolderFactory” class'ından türetilir. “IFactory” interface'inden de “FactoryImplementation” class'ı türetilir. FactoryImplementation class'ından “Instantiate” yapılır ve bu işlemin farklı çok fazla şekli vardır. Container.BindFactory ile bu factory bind edilir. Buradaki önemli nokta bind factory dediğimizde içeri verilen parametrelerdir.

```

    @new
    public class FactoryInterface : PlaceholderFactory<Object, Enemy>
    {
    }

    @new
    public class FactoryImplementation : IFactory<Object, Enemy>
    {
        private DIContainer _diContainer;

        @new
        public FactoryImplementation(DIContainer diContainer)
        {
            _diContainer = diContainer;
        }

        @FrequentlyCalled(0+3 usages @new)
        public Enemy Create(@Object param)
        {
            var enemy = _diContainer.InstantiatePrefabForComponent<Enemy>(param);

            return enemy;
        }
    }

    Container.Bind<MissionSystem>().AsSingle().NonLazy();
    Container.Bind<LevelSystem>().AsSingle().NonLazy();
    Container.Bind<ChapterSystem>().AsSingle().NonLazy();
    Container.Bind<CurrencySystem>().AsSingle().NonLazy();

    Container.BindFactory<Object, Enemy, Enemy.FactoryInterface>()
        .FromFactory<Enemy.FactoryImplementation>();

```

### ➤ Strange IoC:

IoC' nin açılımı “Inversion of Control” yani, kontrolün tersine çevrilmesidir. Kısacası temelinde bir Dependency Injection frameworküdür. Fakat sadece DI kısmını içermez. Dependency injection dışında; shared pub/sub system, view mediation, battle-tested architecture, core binding framework ve multiple, modular context olmak üzere beş farklı ana özelliği daha vardır. OOP' nin kodları ayrıştırması ve bu sayede spaghettiyi önlemesi ile beraber, program parçalarının birbirlerini bilmeye ihtiyaç duymaları sonucunda tekrardan spagettinin oluşması ve bunun bir dependency olması sorununa çözüm olarak ortaya çıkmıştır. Bu sorunun çözümü için var olan yöntemlerden birisi Singleton'dur fakat StrangeloC' nin resmi dökümanında bu çözümü “Bad Solution” olarak tanımlamaktadır. Sebepleri için Singleton Pattern kısmındaki son cümleye bakılabilir. Yine aynı dökümda “Better Solution” ise, Factories & Interfaces olarak adlandırılmıştır. Bu yaklaşım ise test için daha uygundur ve karşılıklı bağımlılık daha azdır fakat, halen client' in factory' e erişime

ihtiyacı vardır ve code yapısı ile ilgili sürekli bir kaygı durumu olur. Son olarak “The Good Solution” ise Inversion Of Control’ dır. Yazımı kolay, esnek, kostümize edilebilir ve test edilebilirdir. Kısa bir tabir ile, DI kısmında da yer aldığı gibi bağımlılıkların enjeksiyonu temeldir. Strange aynı zamanda bir MVCS mimari pattern’ nine sahiptir. Bu pattern dışında; service locator, factory, command, singleton, observer ve mediator patternları da Strange’ de önemli yer kaplarlar. Bu patternler hakkında daha fazla bilgi patternlar kısmında bulunmaktadır. Service Locator pattern noktasında Strange’ de merkezi kayıt defteri “Injector” dır. Bir class, istenen enjeksiyona ilişkin interface’ yi, süper class’ i veya somut class’ i enjekte ettiğiinde Injector, runtime’ de erişim için hizmetin yerini belirler, böylece doğrudan erişimin de önüne geçilmiş olur. MVCS noktasında ise bu mimarideki “S” Service’ yi ifade eder ve uygulamanın kendisi dışındaki iletişimleri yöneten bölüm görevini üstlenir. Sunuculara, web’e, disk’ e yapılan çağrılar servicedir. Strangede açık bir Service class’ i yoktur. Aslında kavramsal olarak ifade etmek için yazılmıştır. S dışında MVC ise patternler kısmında ifade edildiği amaçlar için kullanılır. StrangeloC, Robotlegs micro framework’ ünden çok fazla etkilenmiştir. Bu sebeple “Robotlegs” hakkında, unity ve strange üzerinden de kıyaslama yapacak şekilde daha fazla bilgi vermek gereklidir.

**Robotlegs:** Adobe flash’ in temelini oluşturan ActionScript3 için yazılmış bir IoC micro framework’ üdür. ContextView, Context, Injection, Commands, Views, Mediation, Models ve Service kavramlarını içerir. RL’ de:

- **ContextView:** Üst düzey bir nesnedir. RL’ de üst düzey Sprite iken Unity’ de üst düzey bir GameObject olarak düşünülür. Uygulamanın çeşitli noktalarına enjekte edilebilir. Fakat Flash’ ta her şeyin top level bir nesneye bağlı olma ilkesi varken, unity bu mekanizmaya sahip değildir. Ayrıca unity, event bubbling denilen, view’ lerin contextleri, bağlam konusunda uyarması mekanizması ya da başka bir ifade ile en spesifik elemandan başlayıp en genel elemana kadar giden event zinciri durumu yoktur. Fakat bu durumların dışında Flash ile aynıdır. Bir gameobject/monobehaviour’ a manuel olarak eklenir. MVCView’ dan da türetilabilir.
- **Context:** Dependencylerin maplendiği yani haritalandığı yerdir. MVCContext’ ten de türetilenler ki resmi doküman tavsiyedir. Context’ in içerisinde Binder’ lar bulunur. Binder aslında bir bindings yani bağlama/bağlayıcı sözlüğüdür. Üç liste barındırır ve bu listeler key, value ve name’ dir. Override edilebilir methodlardan birisi mapBindings()’ tir ve amacı aslında bütün bind işlemlerini burada yaparak bind’ leri haritalandırmaktır. Diğer bir override method ise postBinding()’ tir ve amacı Launch()’ tan önce ama mapBindings()’ ten sonra yapmak istediğimiz bind’ ler içindir. Her IoC frameworkünde karşımıza belirli Binder’ ler çıkar. RL ve Strange için bu Binder’ lar:
  - a) *Injector binder:* Temel factory binder’ dir.

```
injectionBinder.Bind<IMyInterface>().To<MyImplementation>();
```

Map to Singleton:

```
injectionBinder.Bind<IMyInterface>()
    .To<MyImplementation>()
    .ToSingleton();
```

Map to value, note how the value does not use the generic brackets:

```
injectionBinder.Bind<IMyInterface>().ToValue(new MyImplementation());
```

Named injection 1. Name with a constant or Enum (non-generic):

```
injectionBinder.Bind<IMyInterface>()
    .To<MyImplementation>()
    .ToName(MyEnum.ONE);
```

Named injection 2. Name with a marker class (generic)... do you see how whenever we bind to a class we use generics?

```
injectionBinder.Bind<IMyInterface>()
    .To<MyImplementation>()
    .To<MyMarkerClass>ToName();
```

Polymorphous binding, multiple interfaces to a single implementation:

```
injectionBinder
    .Bind<IMyInterface>()
    .Bind<IOtherInterface>()
    .To<MyImplementation>();
```

## b) Command binder:

Command bindings

Command binding is a lot like the RL commandMap, but **sequencing** gives us an extra flavor to work with. Let's begin with binding an event to a Command.

```
commandBinder.Bind(EventMap.MY_EVENT).To<MyCommand>();
```

Now we'll bind an event to several commands that fire in parallel.

```
commandBinder.Bind(EventMap.MY_EVENT).To<MyFirstCommand>()
    .To<MySecondCommand>()
    .To<MyThirdCommand>()
    .To<MyFourthCommand>();
```

Note that I'm using the term "in parallel" a tad loosely. It's just a loop firing one after the other. But the Commands are assumed to be disconnected from one another and will all fire independently.

If you'd like to fire events serially, commandBinder has this capability too. This is useful for setting up guards to stop a Command from firing if certain conditions aren't met, and for handling complex chains where a process must complete a step before continuing.

To fire a sequence, just mark the binding with `InSequence()`.

```
commandBinder.Bind(EventMap.MY_EVENT).InSequence()
    .To<MyFirstCommand>()
    .To<MySecondCommand>()
    .To<MyThirdCommand>()
    .To<MyFourthCommand>();
```

RL's one-off method is also supported in the form:

```
commandBinder.Bind(EventMap.MY_EVENT)
    .To<MyCommand>().Once();
```

Which unmaps the binding as soon as the Command has fired. Note that for sequences the unmapping occurs only after the sequence has successfully completed.

- c) *Mediation binder: Mediator' lar, viewlar' a yani monobehaviour' lara doğrudan erişimi keser ve ara katman görevi görürler. Resimdeki gibi kolayca bind işlemi yani mediator' ün view' e bind'lanması yapılabilir.*

```
mediationBinder.Bind<MyView>().To<MyMediator>();
```

Just that simple. The only addition Strange supports is the option to bind multiple mediators. This allows the dynamic adding of Monobehaviours to your gameObjects.

```
mediationBinder.Bind<MyView>()
    .To<MyMediator>()
    .To<MyOtherMediator>();
```

- d) *EventDispatcher: MVCS kullanımında Context\_Dispatcher yani ilgili context boyunda haberleşme sağlayan ve Cross\_Context\_Dispatcher yani contextler arası iletişim sağlayan ya da context' i olmayan yani yerel iletişimde kullanılmak üzere üç çeşidi vardır.*

eventsDispatcher.unity.cs  
Just like in RL, we don't really do a lot of dispatcher mapping inside the context, but I'll discuss EventDispatcher here since it's one of the important pieces instantiated in the Context. First, note that, like most of Strange, use of the EventDispatcher is optional. You can subclass Context and set up a framework using a dispatcher you prefer (Unity-Signals anyone?). But the EventDispatcher works well here and will be pretty familiar. Also, it's probably worth noting that since our Events don't bubble, this dispatcher isn't quite as expensive as Flash's. Let's look at the mappings.

Add a listener: Anything can be used as the event type. I recommend an Enum (EventMap below might be an Enum or a list of constants):

```
dispatcher.AddListener(EventMap.HY_EVENT, onSomeEvent);
```

The handler must take either no arguments or a single argument like so:

```
private void onSomeEvent(Event evt)
{
    //do stuff.
}
```

A few important points here: IEvent is obviously the Event interface. For maximum decoupling, we use this instead of the concrete TmEvent. IEvent specifies three properties:

- type - The key of the event. Enum recommended, but can be anything.
- target - The dispatcher that dispatched the event.
- data - An arbitrary payload.

Also note that C# won't let you identify an event by the term 'event' since it's a keyword. So use my 'evt' convention or come up with your own.

Removing a listener is just what you'd expect:

```
dispatcher.RemoveListener(EventMap.HY_EVENT, onSomeEvent);
```

And we've added a boolean convenience function (true to add, false to remove):

```
dispatcher.UpdateListener(true, Events.HY_EVENT, onSomeEvent);
```

There are a number of options when dispatching:

Dispatch with null payload:

```
dispatcher.Dispatch(EventMap.HY_EVENT);
```

Dispatch with payload:

```
dispatcher.Dispatch(EventMap.HY_EVENT, 42);
```

Dispatch by providing the whole TmEvent explicitly:

```
TmEvent evt = new TmEvent();
(EventMap.HY_EVENT, dispatcher, someVar);
dispatcher.Dispatch(evt);
```

I've not implemented priorities in dispatching, and, as I rarely used it, probably won't unless someone tells me it's desperately important. And as I've noted already, bubbling doesn't really come into play.

A final word before I move on: there are three mappings for EventDispatcher in MVCSContext, each for a particular use:

1. The context-wide dispatcher, mapped to the name ContextKeys.CONTEXT\_DISPATCHER. Use this dispatcher as the event bus throughout your Context, just like RL.
2. The crossContext dispatcher, mapped to the name ContextKeys.CROSS\_CONTEXT\_DISPATCHER. Use this dispatcher for communicating between contexts.
3. The local dispatcher, which isn't mapped to any name. Injecting this is useful for creating a new instance of EventDispatcher for use in highly local communication (as between a View and its Mediator).

- **Injection:** [Inject] attributesi ile yapılır. RL deki yaklaşım Strange' de de aynıdır fakat RL Swiftsuspenders adlı bir enjektörü kullanırken, Strange' nin enjektörü kendine aittir. Strange' de setter injection, contructor injection destekler fakat field injection desteklemez. Ayrıca RL' deki gibi [PostConstruct] attributesi de Strange' de vardır. Son attribute ise [Construct] attributesidir çünkü c# ta birden fazla constructor olmaz bu nedenle bu attribute ile Strange' nin neyi baz alacağını emredir. Eğer bir adet constructor varsa o, iki adet varsa [Constructor] attributesi olan ve eğer birden fazla bu attributeden varsa en az parametresi olan baz alınır.

### Injection

Let's talk about those marvelously injected classes. Unlike RL, which used the `Swiftsuspenders` injector, Strange has an injector of its own. But if you're one of those people who feels they should roll their own, never fear; it's all abstracted so you could do just that.

Marking classes for injection is really straightforward if you're coming from Robotlegs. Here's a setter injection:

```
[Inject]
public IMyDependency myDependency{get;set;}
```

Notwithstanding a few C# syntactical differences, this is identical to the RL approach. You can inject things using the `[Inject]` metatag, but note that you must designate setters and getters, even if they're only stubs. Strange supports Setter injection and constructor injection, but not field injection.

You can also do a named injection. Here's an example with the name based on an Enum. I like this.

```
[Inject(SomeEnum.ONE)]
public IMyDependency myDependency{get;set;}
```

But you could also map a name to a marker class. The syntax ends up a little more convoluted.

```
[Inject(typeof(MyMarkerClass))]
public IMyDependency myDependency{get;set;}
```

Strange also supports the metatag `[PostConstruct]`.

```
[PostConstruct]
public void postConstruct()
{
    //do stuff immediately after injection
}
```

Just like in Swiftsuspenders, you can tag as many PostConstruct methods as you like. I haven't yet implemented an ordering parameter, but I'll attempt to add that soon.

Now Constructors are a bit more complicated, especially since in C# (unlike in Actionscript) you can have more than one. Because of this, I've created a set of rules to determine which Constructor to use. It works like so:

1. If there's only one constructor, use it.
2. If any constructor is marked with the `[Construct]` metatag, use that method for constructing.
3. If there are more than one, but none is marked with `[Construct]`, use the one with the fewest parameters.

- **Commands:** Execute methodunu override eder. Execute biter bitmez gc tarafından temizlenir fakat Retain ile bekletme ve Release ile gc ye devam et emrini verebilirisiniz. Burada dikkat edilmesi gereken Relase methodundan önce listenerin remove edilmesidir. Son olarak sequence commandlarda Fail methodu ile sequence' i bitirebilirisiniz.

```
using strange.extensions.command.impl;
using com.example.spacebattle.utils;
namespace com.example.spacebattle.controller
{
    class StartGameCommand : EventCommand
    {
        [Inject]
        public ITimer gameTimer{get;set;}

        override public void Execute()
        {
            gameTimer.start();
            dispatcher.dispatch(GameEvent.STARTED);
        }
    }
}
```

```

using strange.extensions.command.impl;
using com.example.spacebattle.utils;
namespace com.example.spacebattle.controller
{
    class StartGameCommand : EventCommand
    {
        [Inject] public IService server{get;set;}

        override public void Execute()
        {
            Retain();
            server.dispatcher.AddListener
                (ServerEvent.COMPLETE, onComplete);
            server.call();
        }

        private void onComplete(IEvent evt)
        {
            //remember to remove listeners!!!
            server.dispatcher.RemoveListener
                (ServerEvent.COMPLETE, onComplete);
            //Do something with the result
            Release();
        }
    }
}

```

```

override public void Execute()
{
    if (server.isAvailable)
    {
        Retain();
        server.dispatcher.AddListener
            (ServerEvent.COMPLETE, onComplete);
        server.call();
    }
    else
    {
        server.retries++;
        if(server.retries > server.maxRetries)
        {
            dispatcher.Dispatch(ServerEvent.NOT_RESPONDING);
        }
        Fail();
    }
}

```

- **Views and Mediation:** View' ler de Mediator' ler de Unity özelinde aslında monobehaviourdan türerler. Amaçları RL ile aynıdır. View, kullanıcının gördüğü, temas ettiği şey iken, mediator view' inin uygulamanın geri kalanı arasındaki iletişime köprü olur. Bu sayede Strange' nin en büyük amaçlarından biri olan unity' ye özgü kodları, sistemden ayırmış oluruz.
- **Models and Services:** Nothing.

## StrangeloC' nin Temel özellikleri:

- **DI:** Sınıfların bağımlılıklarını optimize edilmiş reflection/injection sistemi ile ayırrı. Bu sayede modüler ve temiz kod yazılır. Singleton, value veya factory olarak haritalama yapılabilir. Name injections, constructor veya setter injections, tercih edilen constructor' a inject, monoBehaviours' a inject, polimorfik bind (bütün interfaceleri veya istenenleri tek bir concrete class' a bind), reflection bind yapılarını destekler. [Inject] attribute' si ile enjekte işlemi yapılır.
- **Shared Pub/Sub System:** Her context bir adet EventDispatcher' a sahiptir ve buna koddaki herhangi bir nokta erişebilir.
- **View Mediation:** View' lerin models ve controllers lardan hiçbir yetenek kaybı olmadan temiz bir şekilde ayrılmasını sağlar.
- **Battle-Tested Architecture:** Robotlegs micro-framework' ü baz alır. Son derece yaygın ve sorumlu gelişimi hafifçe teşvik eden bir mimarıdır.
- **Core Binding Framework:** Strange' nin özüdür. Genişletilebilir bir Binder sınıfıdır. Her component Binder' in bir uzantısıdır.
- **Multiple, Modular Contexts:** Modüler contextler ile proje bölüm bölüm ayrılabilir ve böylece ekiplere dağıtılabılır. Bu sayede bu bölümler bir api üzerinden tak çalışır şeklinde kullanılır. Bu sayede development basitleştir ve tüm componentlar daha sonra yeniden kullanılabilir olur.

## Kullanımı:

- **Binding:** Strange' nin özüdür. Bir şeyin bir veya daha fazlasını başka bir şeyin bir veya daha fazlasına bağlamamızdır. Bir interfaceyi, bu interfaceyi uygulayan bir class' a veya bir event' i handlера bağlamak için veya sınıflardan biri oluşurken diğerini de oluşturulsın diye kullanırız. Strange kodlar arası daha bağımlılığı daha soyut daha dolaylı bir hale getirir. Aslında OOP nin temelini gerçekleştirir. Bind işleminde iki zorunlu parametre bir de opsionel parametre yer alır. Zorunlu olanlar bir key ve bir value' dir. Aslında key üzerinden değer triggerlenir. Böylece bir event, bir call back' i tetikleyen bir trigger olabilir. Bu durumun benzeri aslında bir class' ın oluşturulmasını bir callback gibi kullanıp başka bir class' ı oluşturmaktır. Opsiyonel parametre ise name' dir. Bind key' dir, To ise value. Aslında burada Binder key ile beslenir ve output olarakta value verir. Son fotoğraftaki Bind işlemlerinin hepsi aynı işlemi yapar.

```
Bind<Spaceship>().To<Liberator>();
```

```
Bind<IDrive>().To<WarpDrive>();
Bind(typeof(IDrive)).To(typeof(WarpDrive));
IBinding binding = Bind<IDrive>();
binding.To<WarpDrive>();
```

- **Extensions:** Strange bir DI framework' ü de olsa yapımcısı bundan daha fazlası olduğunu ifade ediyor ve bu noktada hem MVCS hem de bu mimariye entegre extensionslar sunuyor. Bunlar:

- a. *The injection extension:* IoC ile en yakın ilişkili extension' dur. Bu noktada hiçbir class, başka bir class'ın bağımlılıklarını açıkça yerine getirmemelidir. Bu duruma DI denir. DI' da bir class ihtiyaç duyduğu şeyleri mümkünse interface biçiminde ister ve Injector bu ihtiyacı karşılar. Factory' e benzer fakat her interface için bir factory kurmak gerekmekz. Factory' de olsa, tepeden aşağı zincir şeklinde de olsa bir bağımlılık olduğu için bu yöntem tercih edilmektedir. Strange' de C# in System.Reflection paketi ile istekte bulunan class incelenebilir. Strange' de bu paketi özelleştirdip kullanmaktadır. [Inject] attribute' si ile bağımlılığınızı istersiniz. Strange' de Context içerisinde soyut sınıf somut sınıfa bağlanarak bind işlemi yapılır. Bind türü olarak GetInstance ile yeni class construc edilebileceği gibi ToSingleton ile tekilik sağlanabilir veya bir interfaceden birden çok bind yapılacaksa ayrıştırma için adlandırılmış enjeksiyon yani ToName kullanılabilir. Birden fazla interface üzerinden de bind yapılabilir. Değer eşleme için de ToValue kullanılır. Her seferinde yeni bir örneğe ihtiyaç varsa da factory bind kullanılır. Strange de setter injection dışında bir de constructor injection vardır. Setter daha az kod ve daha esnek olmakla birlikte özel olması gereken bazı şeyleri halka açık yapar. Contructor da ise özel olması gerekenler özel kalırken, daha fazla kod ve daha az esneklik dez avantajlarına sahiptir. Strange [Construct] attributesine sahiptir ve bununla ilgili 2 tane varsa en az parametreli olan, 1 tane varsa o şeklinde bir kurula sahiptir. Bir method için [PostConstruct] attributesi kullanılır ise, bu durum setter injection tercih edildiğinde injectten hemen sonra bu methodun çalışması anlamına gelir yani bir constructor gibi davranışabilir. Dikkat edilmesi gereken konular; bağımlılık döngüsü oluşturmamak, reflection işleminin ağır bir işlem olması ve bindlamanın unutulmaması gereklilikleridir.
- b. *The reflector extension:* Dökümda fazla detay bilinmesi gerekmeyen, injection esnasında orda olduğunu bilmemizin yeterli olduğu extension. Normal reflectiondan ortalama 5 kat daha hızlı çalışan fakat yine de hız konusunda dikkatli olunması gereken bir yapı.
- c. *The dispatcher extension:* Strange' nin orijinal ve varsayılan iletişim sistemi. Güncel olarak type safe olan Signals de kullanılabilir ve önerilmektedir. Klasik Observer Pattern açısından dispatcher, subject' e karşılık gelmektedir. Genel interface IEvent' tir. ContextDispatcher veya CrossContextDispatcher ise MVCS sürümünde olan dispatcherlardır. Olayları dinlemek için AddListener, dinlemekten çıkmak için RemoveListener kullanılır. Dinleyiciyi bir boolean ile güncellemek için UpdateListener kullanılır. Subject ise Dispatch methodu ile sinyali verir. Dispatch yapıldığında aslında TmEvent oluşur ve dinleyiciler bunu haber alır.
- d. *The command extension:* MVCS' teki controllerların yerini alır. Commandlar ile eventleri veya signalleri bağlamak mümkündür. CommandBinder contexten gelen her dispatch' i dinler. Bir event veya signal tetiklendiğinde CommandBinder, bu event veya signalin bir veya daha fazla command' a bağlı olup olmadığı kontrol eder. Eğer bind varsa yeni bir command örneği oluşturup enjeksiyon yapılır yürütülür ve sonra imha edilir. Commandlar

*ICommand interfacesinden türer. ContextDispatcher' a erişebilir ve işleminin yanında dispatch yapabilir. Execute tamamlanınca da command temizlenir. Retain ve Release methodları ise RL' de anlatıldığı gibi hareket ederler. Sequence özelliği ile commandları sıralı şekilde çalıştırmak mümkündür. Sıra sonuna ulaşana kadar veya commandlardan biri fail olana kadar devam eder. InSequence metodu ile eklenir.*

```
using strange.extensions.command.impl;
using com.example.spacebattle.service;

namespace com.example.spacebattle.controller
{
    class PostScoreCommand : EventCommand
    {
        [Inject]
        IServer gameServer{get;set;}

        override public void Execute()
        {
            Retain();
            int score = (int)evt.data;
            gameServer.dispatcher.AddListener(ServerEvent.SUCCESS, onSuccess);
            gameServer.dispatcher.AddListener(ServerEvent.FAILURE, onFailure);
            gameServer.send(score);
        }

        private void onSuccess()
        {
            gameServer.dispatcher.RemoveListener(ServerEvent.SUCCESS, onSuccess);
            gameServer.dispatcher.RemoveListener(ServerEvent.FAILURE, onFailure);
            //...do something to report success...
            Release();
        }

        private void onFailure(object payload)
        {
            gameServer.dispatcher.RemoveListener(ServerEvent.SUCCESS, onSuccess);
            gameServer.dispatcher.RemoveListener(
                ServerEvent.FAILURE, onFailure);
            //...do something to report failure...
            Release();
        }
    }
}
```

#### Mapping commands

Although technically we can map Commands to events almost anywhere, we typically do so in the Context. Doing so makes it easy to locate when you (or anyone else) needs to find what's mapped to what. Command mapping looks a lot like injection mapping:

```
commandBinder.Bind(ServerEvent.POST_SCORE).To<PostScoreCommand>();
```

You can bind multiple Commands to a single event if you like:

```
commandBinder.Bind(GameEvent.HIT).To<DestroyEnemyCommand>().To<UpdateScoreCommand>();
```

And you can unbind at any time to remove a binding:

```
commandBinder.Unbind(ServerEvent.POST_SCORE);
```

There's also a nice "one-off" directive for those times where you only want a Command to fire just the next time an event occurs.

```
commandBinder.Bind(GameEvent.HIT).To<DestroyEnemyCommand>().Once();
```

By declaring `Once`, you ensure that the binding will be destroyed the next time the Command fires.

```
commandBinder.Bind(GameEvent.HIT).InSequence()
    .To<CheckLevelClearedCommand>()
    .To<EndLevelCommand>()
    .To<GameOverCommand>();
```

- e. *The signal extension: Strange v0.6.0 ile gelmiştir. EventDispatcher' a alternatifdir. EventDispatcher tek bir data property' si ile IEvent object' i üretir ve dispatch eder. Signals ise callback' lere bağlanır, signal gönderimi yeni bir nesne üretmez ve böylece gc iş yükü azalır ayrıca type safe özelliğine sahiptir. Signal ile callBackler eşleşmezse bağlantı bozulur. Diğer bir fark ise her context için tek bir EventDispatcher veya CrossContextDispatcher varken, Signalde her event, bir görevi atanın bireysel bir sinyalin sonucudur. Yani EventDispatcher tekken Signals istenilen sayıda olabilir. Burada önemli olan şey, dinleyici için signal compile time gereksinimidir. Yanlış ekleme durumunda proje compile olmaz. Signaller actiondan türelerler ve doğal olarak 4 parametreye kadar izin verirler. Signal' den türeyen sub classlar yazmak mümkündür. Varsayılan commandBinder' i To<SignalCommandBinder> ile değiştirerek signalleri commandlara bağlamak ve varsayılan olarak signal kullanmak mümkündür. Böylece commandBinder.Bind<SomeSignal>().To(SomeCommand) gibi bir bind işlemi ile event' i command' a bağlamak yerine signal command' a bağlanır. Halen commandBinder yazıyor fakat burada signalcommandbinder artık görevi yerine getiriyor. [Inject] ile enjeksiyonu mümkündür. Eksi olarak aynı türden iki parametreye sahip bir signal, herhangi bir command' a bind edilemez. Bir diğer eksik ise, EventDispatcher' i geçersiz kılmak, Start eventini geçersiz kılar. Bu sebeple custom bir start signal' i context' in Launch metoduna eklenmelidir. Herhangi bir command' a değil command' siz bir şekilde direkt injectionBinder ile de enjekte edilebilir.*

```

Signal<int> signalDispatchesInt = new Signal<int>();
Signal<string> signalDispatchesString = new Signal<string>();

signalDispatchesInt.AddListener(callbackInt);           //Add a callback with an int parameter
signalDispatchesString.AddListener(callbackString);     //Add a callback with a string parameter

signalDispatchesInt.Dispatch(42);                     //dispatch an int
signalDispatchesString.Dispatch("Ender Wiggin");      //dispatch a string

void callbackInt(int value)
{
    //Do something with this int
}

void callback(string value)
{
    //Do something with this string
}

```

```

Signal<int> signalDispatchesInt = new Signal<int>();
Signal<string> signalDispatchesString = new Signal<string>();

signalDispatchesInt.AddListener(callbackString); //Oops! I attached the wrong callback to my Signal!
signalDispatchesString.AddListener(callbackInt); //Oops! I did it again! (Am I klutzy or what?!)
```

```

//You can do this...
Signal<SuperClass> signal = new Signal<SuperClass>();
signal.Dispatch(instanceOfASubClass);

//...but never this
Signal<SubClass> signal = new Signal<SubClass>();
signal.Dispatch(instanceOfASuperclass);

```

```

//This works
Signal<int, int> twoIntSignal = new Signal<int, int>();
twoIntSignal.AddListener(twoIntCallback);

//This fails
Signal<int, int> twoIntSignal = new Signal<int, int>();
commandBinder.Bind(twoIntSignal).To<SomeCommand>();
```

- f. *The mediation extension:* Bu extension Unity3D ile kullanılmak üzere yazılmış tek extension' dur. Bunun sebebi de view ile sistemin geri kalanının nasıl iletişim kuracağıının belirlenmesidir. Viewlar enjekte edilebilirler fakat modellere veya hizmetlere view enjekte etmek doğru bir uygulama değildir. Çünkü bu sistemdeki ana mimarının amacı zaten view' ları sistemden mediator aracılığı ile uzak tutmak. View' in amaçları; görselliği sunmak, kullanıcının temaların gönderilmesi( signal veya event) ve sahip olduğu görselliğin değiştirilmesini sağlamaktır. Bu noktada bu değişimi sağlayacak olan ise mediator' dır. Mediator de bir monobehaviour' dur. Lite versiyonu ise non monobehaviour olan versiyonudur. Yapısı gereği ince bir katman olarak tanımlanır. OnRegister, injectiondan hemen sonra ateşlenir. Her view için bir adet mediator olması gereklidir.

```

using Strange.extensions.mediation.impl;
using com.example.spacebattle.events;
using com.example.spacebattle.model;
namespace com.example.spacebattle.view
{
    class DashboardMediator : EventMediator
    {
        [Inject]
        public DashboardView view{get;set;}

        override public void OnRegister()
        {
            view.init();
            dispatcher.AddListener
                (ServiceEvent.FULFILL_ONLINE_PLAYERS, onPlayers);
            dispatcher.Dispatch
                (ServiceEvent.REQUEST_ONLINE_PLAYERS);
        }

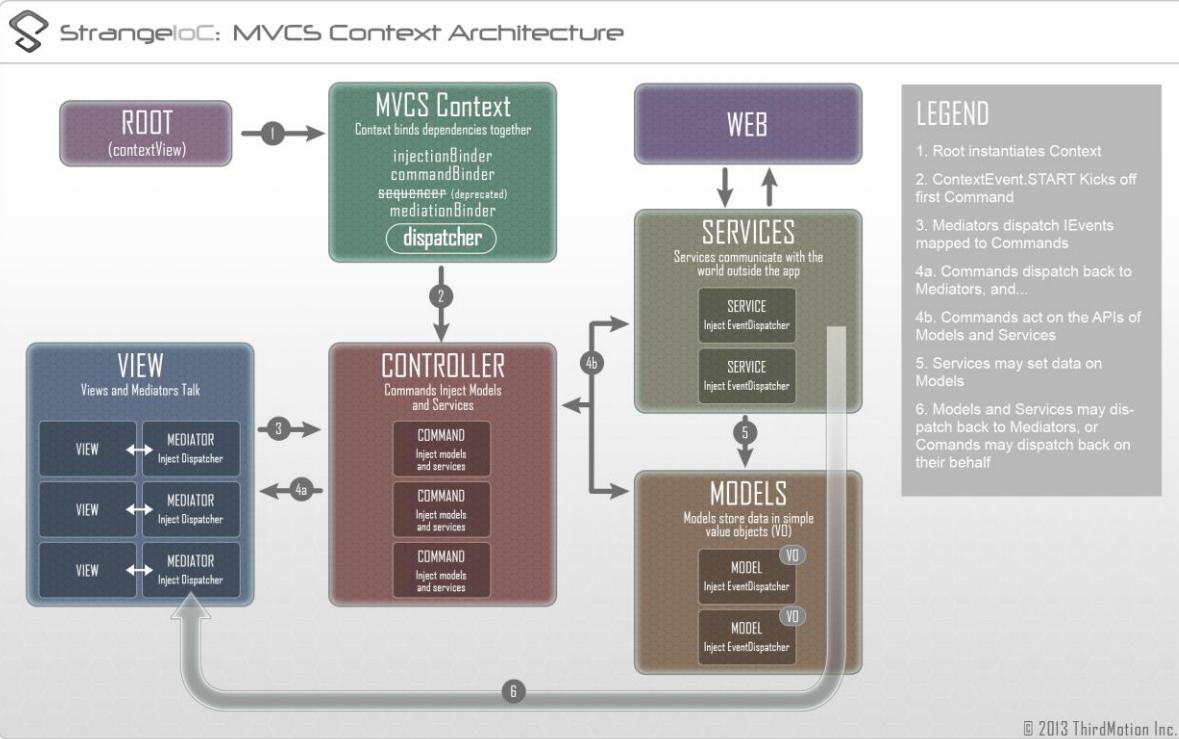
        override public void OnRemove()
        {
            dispatcher.RemoveListener
                (ServiceEvent.FULFILL_ONLINE_PLAYERS, onPlayers);
        }

        private void onPlayers(IEvent evt)
        {
            IPlayers[] playerList = evt.data as IPlayers[];
            view.updatePlayerCount(playerList.Length);
        }
    }
}

```

g. *The context extensions: Context bind işlemlerinin yapıldığı yerdir. Birer adet; injectionBinder, mediationBinder ve commandBinder içerir. Keza MVCSContext' te de durum böyledir. Bu yapıyı oyunun modülleri gibi düşünürsek birden fazla olması genellikle tercih edilmektedir.*

- **MVCS:** Strange' nin tüm sistemini kolay bir şekilde kullanmamızı sağlayan, temelinde bir structal patterndir. Oyunun giriş noktası Root yani bir monobehaviour olan ContextView' dır. MVCS Context tüm bindinglerin yapıldığı yerdir. Dispatcher context içi haberleşme aracıdır ve standart dispatcher TMEvents adlı bir class gönderir fakat değişiklikler için Signal kısmına bakılabilir. Modeller data container görevi üstlenir. Services ise dış dünya ile ilgili bağlantıyı sağlar. View' lar kullanıcının gördükleri ve temas ettikleridir ve monobehaviour' dur. Mediator' ler ise view' lar gibi monobehaviour olup view ile sistemin geri kalanı arasındaki bağ ve view'ları sistemden ayıran parça görevi görürler. Atılan kod tabiri yapılır çünkü çok spesifik özelliklere sahip olacaktır. View'ların mümkün olan en üst seviyede sistemin geri kalanından ayrıştırılması önemlidir. Bu bağlamda View, herhangi bir context' i, command' i, service' yi veya root' u bilmemelidir. Yalnızca dökümanda dispatcher sahibi olabilir olarak belirtilmiştir. Mediator' deki metodlar OnRegister ve OnRemovedir. OnRegister injection' dan hemen sonra ateşlenir. OnRemove ise monobehaviour' daki OnDestroy' un bir sonucudur. Mediator' dan listenerları temizlemekte gc için önemlidir. Models ve Services commandlar tarafından kullanılırlar. Bu sebeple event listen veya signal tarzı işlemlerde bulunmamaları gereklidir. Mükündür fakat doküman tarafından MVCS mimarisine uygunluk ve proje yönetimi açısından kesinlikle tavsiye edilmemektedir. Son olarak bind işleminde CrossContext ataması yaparak her contexten erişilebilir bir bind yapılabilir.

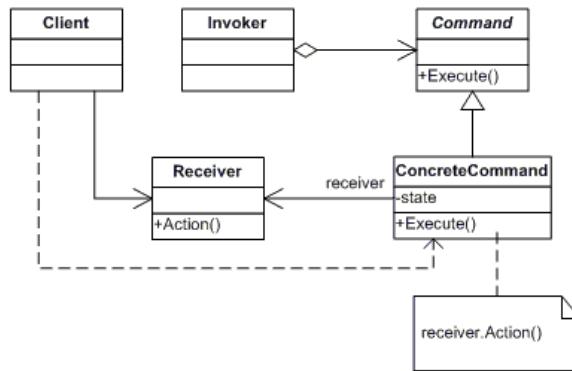


© 2013 ThirdMotion Inc.

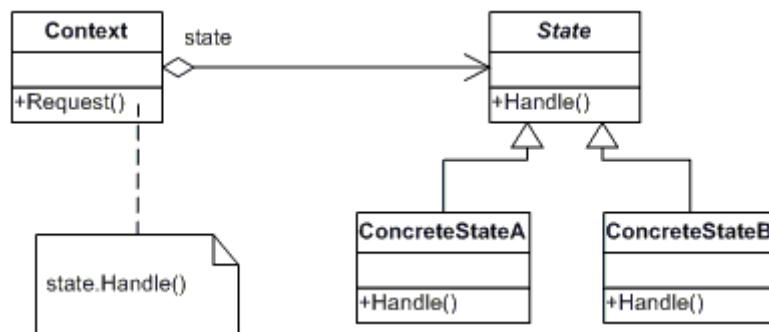
## DESIGN PATTERNS

### ➤ Behavioral Patterns:

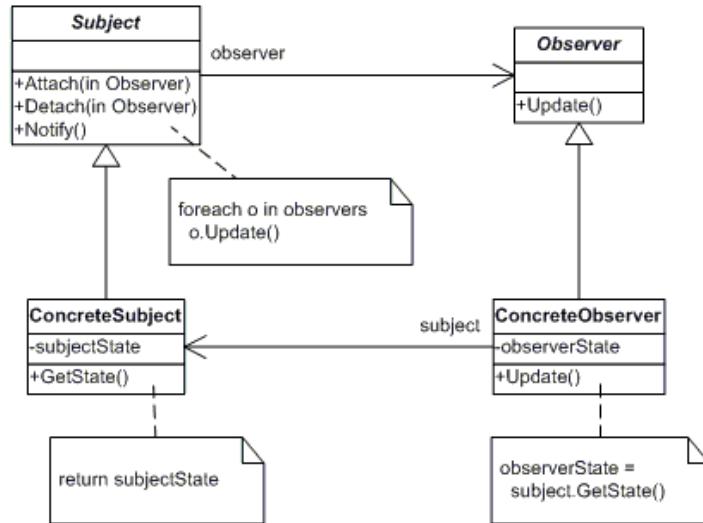
- **Command Pattern:** Temeli isteklerin nesneleştirilmesidir. Bu sayede istekleri parametreleştirme ve geri alma işlemleri yapılabilmektedir. Command, ConcreteCommand, Client, Invoker ve Receiver class ve nesneleri ile gerçekleştirilir. Command: bir işlemi execute eder ve bu execute bir interfaceden veya abstract classtan gelir. ConcreteCommand: Receiver nesne ile command arasındaki bağlantıyı gerçekleştirir. Alıcı üzerindeki action'ı uygulayarak command'daki executenin uygulanmasını sağlar. Client: ConcreteCommand'ı üretir ve alıcısını set eder. Invoker: Command'ın isteği yerine getirmesini söyler. Oyunlarda genellikle; rebind keys, replay sistemi yapmak için veya undo, redo sistemi için kullanılır.



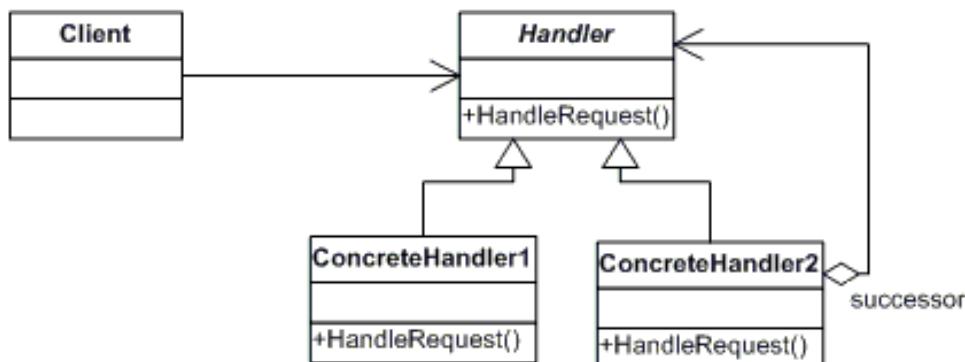
- **State Pattern:** Runtime geçiş yapılmasını kolaylaştırır. Bir nesne runtime' da davranışını tamamen değiştirecekse, state design kullanmak hem kontrollü hem de kolaydır. AI konusunda kullanımı yaygındır. İşçi bir karakterin yakınına düşman geldiğinde saldırması ve düşman ile işi bir şekilde bittiğinde yeniden işine geri dönmesi gibi durumlarda kullanımı yaygındır. İlgili nesnenin durumu değiştiğinde davranışı da değişecekse, state pattern kullanmak mantıklıdır çünkü development tarafı için de kontrolü elimizde tutmamızı sağlar. Menu sistemlerinde, turn based combat oyunlarda, ai tarafında ve unity özelinde animasyon tarafında kullanılmaktadır.



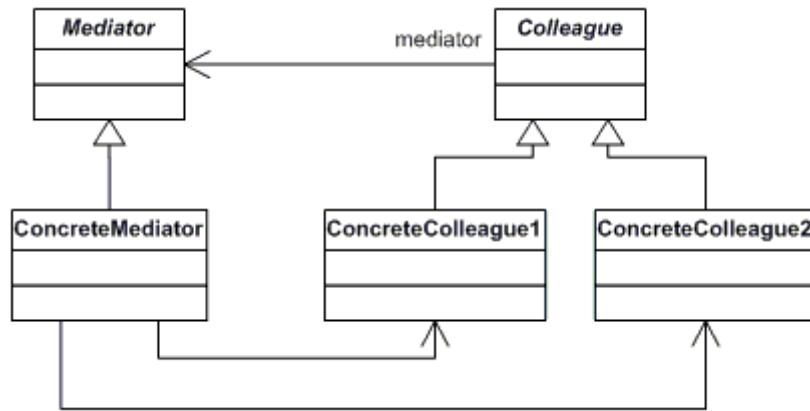
- **Observer Pattern:** Son derece popüler bir patterndir. One To Many prensibine dayanır. Bu prensip bir nesnedeki değişikliğin, birden fazla nesneyi etkilemesidir. Oyun içerisinde bor cuk kez farklı şeyler olur ve buradaki olmaktan kasit olacak olan değil olmuş ve bitmiş şeylerdir. Bu bağlamda event, messages tan farklıdır. Bu pattern, olmuş ve bitmiş şeyler ile ilgilidir. Bir düşmanın ölümü, bir collectablenin toplanması gibi durumların sonucunda, bu olaylar ile ilgili olanların bu eventlere subscribe olması ve olayın gerçekleşmesi sonucu tetiklenmeleri ile observing yani gözleme işlemi yapılmış olur. Action, delegate, Unity Event, EventHandler' lar observe için kullanılır. Observer pattern'daki en büyük sorun olayın gerçekleşmesi ile bütün abonelerin tetiklenecek olmasıdır. Bu durum optimizasyon kaygısı oluşturabilir ve bunun önüne geçmek için Event-Queue yöntemi kullanılır.



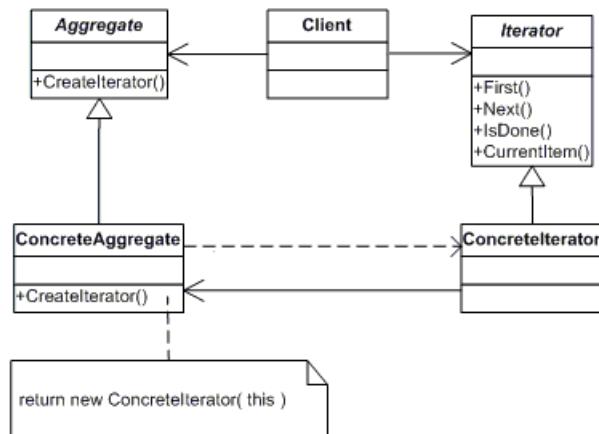
- **Chain Of Responsibility Pattern:** İstekleri bir zincir boyunca iletmemizi sağlar. İlgili isteği işlemek için birden fazla nesneye şans verilen ve istek işlenene kadar devam eden bir zincirdir. Zayıf bağlı olma ilkesine uygundur. Clientin istediği bir concrete handlerdan requesti ile zincir başlar ve handlerdan türemiş olan concrete handlerlar sırası ile isteği işlemeyi denerler. İşlemeyi yapamayan concrete handler ardılına görevi devreder.



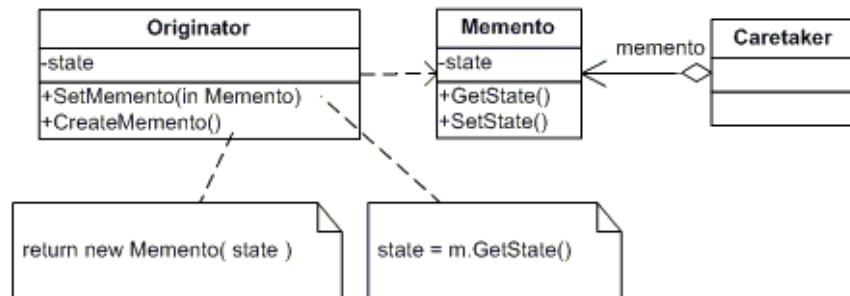
- **Mediator Pattern:** Nesneler arası iletişimde bir aracı kullanılması prensibine dayanır. Arabulucu, nesnelerin açıkça birbirlerine atıfta bulunmasını engelleyerek loose couple ilkesini sağlamış olur ve bu durum nesnelerin etkileşimlerini nesnelerden bağımsız şekilde değiştirmemize olanak sağlar.



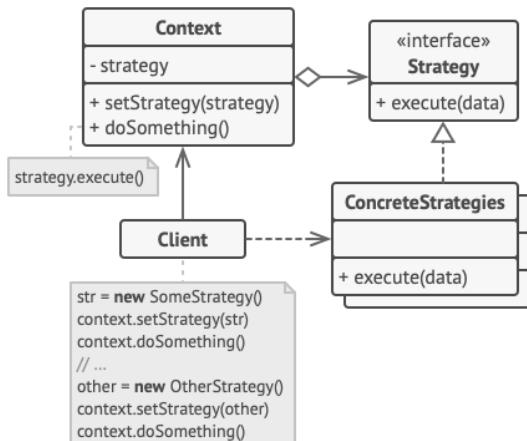
- **Iterator Pattern:** Ana fikir, bir koleksiyonun geçiş davranışını, iterator adı verilen ayrı bir nesneye çıkarmaktır. Koleksiyonlarda aynı öğeye tekrar tekrar erişmeden geçiş yapılabilmesi içindir.



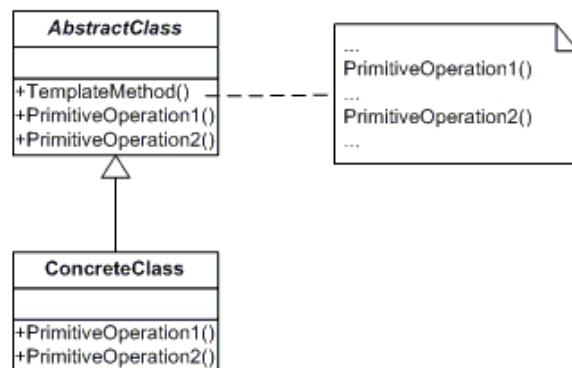
- **Memento Pattern:** Bir işlemin save edilmesi ve ileride geri alma işleminin yapılmasıındaki, güvenlik veya gizlilikten kaynaklı anlık üretim yapılamaması problemlerinden doğmuştur. Bu sorunların temelinde bozuk kapsülleme yatar. Bu noktada çözüm ise anı oluşturma işlemini originator'un yapması ve caretaker'in ise geri alma yani anıyı işleme görevini yapmasıdır. Caretaker asla anının içeriği ile ilgilenmez.



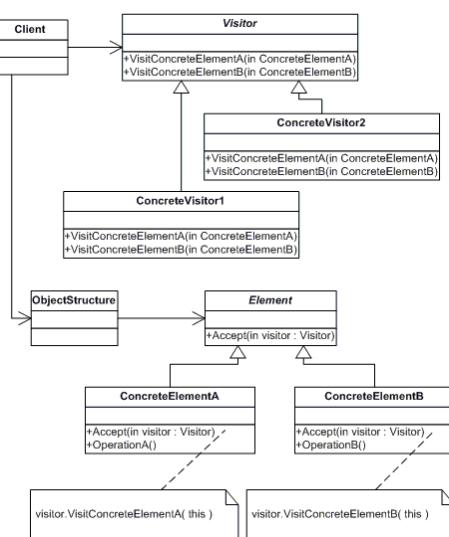
- **Strategy Pattern:** Bir algoritma ailesi tanımlanmanıza, her birini ayrı bir sınıfı koymانıza ve nesnelerini birbiriyle değiştirilebilir hale getirmenize izin verir.



- **Template Method Pattern:** Aynı genel algoritmayı kullanan classlar var fakat bu sub classlar, algoritmadaki bazı adımları farklı bir şekilde uygulamak istiyorlarsa kullanılır. Sub classlar algoritma iskeletine zarar vermeden kendi isteklerini yapabileceklerdir.

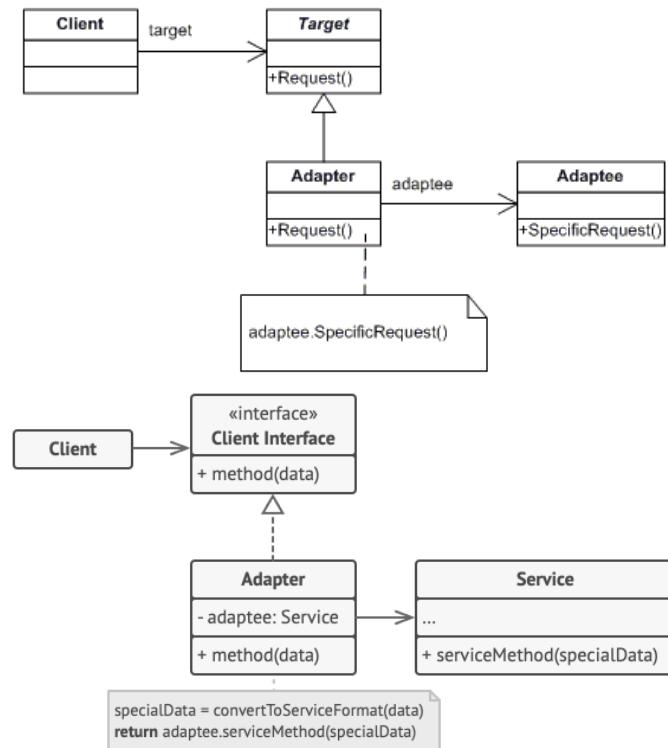


- **Visitor Pattern:** Algoritmaları, üzerinde çalışıkları nesnelerden ayırmamıza yarar.

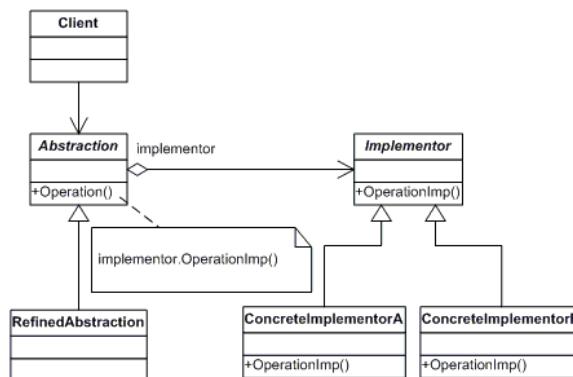


## ➤ Structural Patterns:

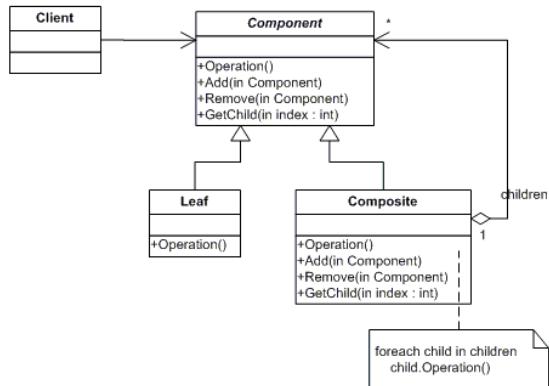
- **Adapter Pattern:** Adapter pattern, uyumsuz interfacelere sahip nesnelerin iş birliğini yapmasını sağlar.



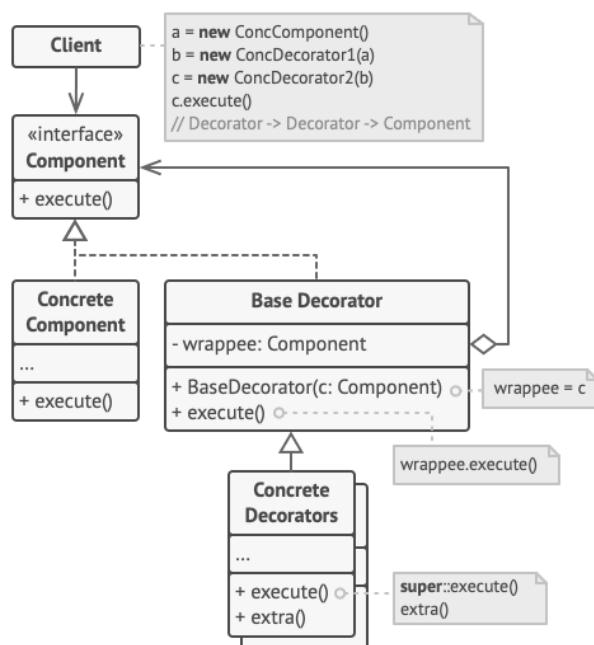
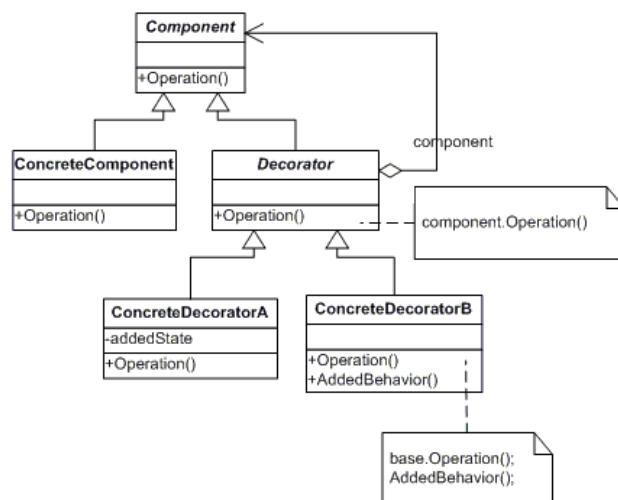
- **Bridge Pattern:** Büyük bir class'ı veya yakından ilişkili sınıflar kümesini birbirinden bağımsız olarak geliştirebilen iki ayrı hiyerarşide ayırmamıza yarar. Soyutlama ve uygulama prensibine dayanır.



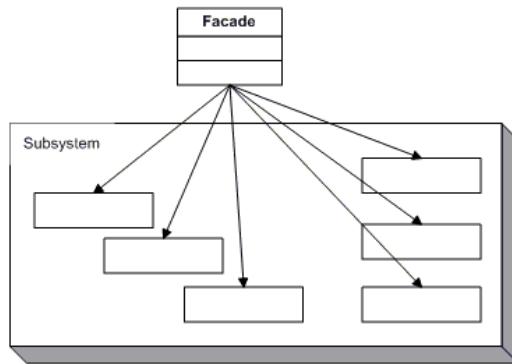
- **Composite Pattern:** Ağaç mimarisidir. Ordulardaki emir komuta zincirine benzer.



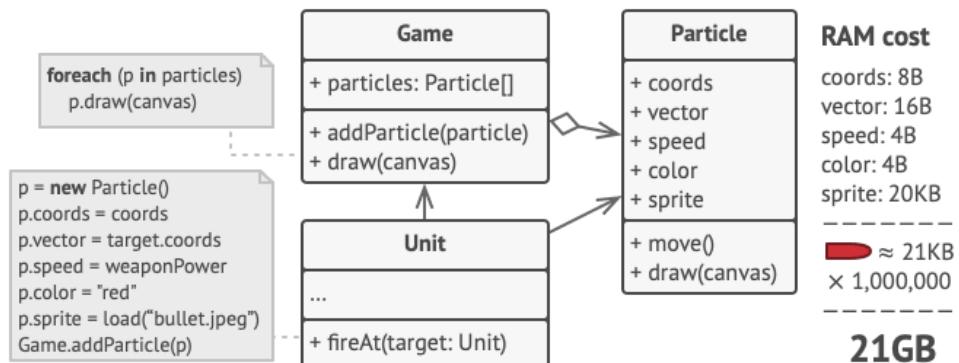
- **Decorator Pattern:** Sub classların işlevlerini dinamik olarak değiştirmemize yarar. Kalıtım statiktir ve sub clas süper classtan kalıtım yoluyla bazı özellikler devralır. Decarot patternda ise sub classlara bir alternatif yaratılarak, işlevsellik gerçek zamanlı bir şekilde değiştirilir.

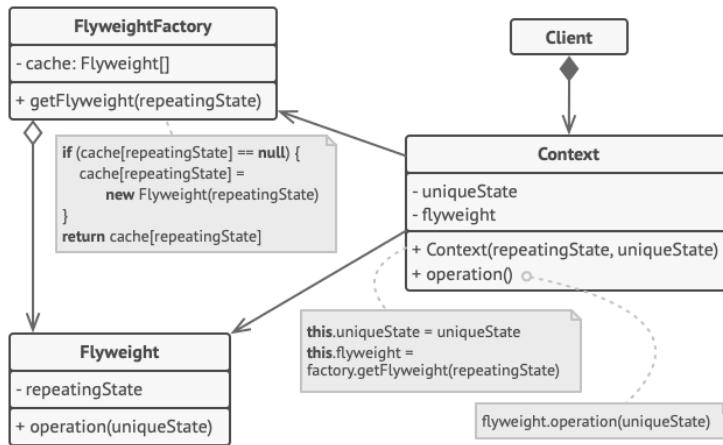
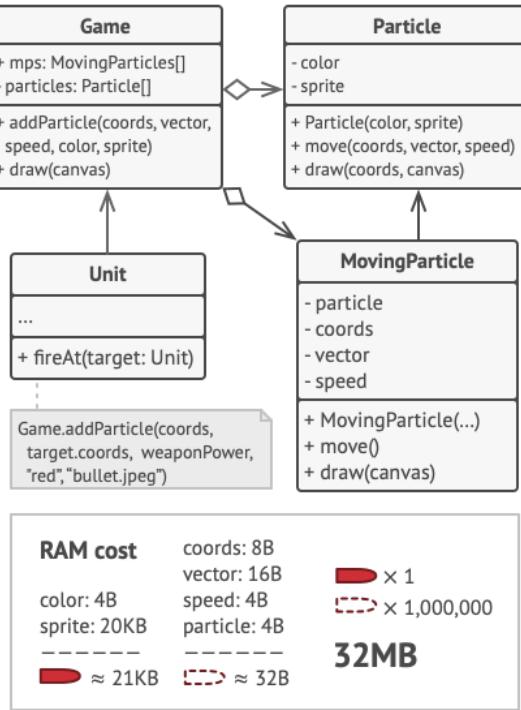


- **Facade Pattern:** Bir library, framework veya karmaşık bir class yapısın basitleştirilmiş bir arabirim sağlar.

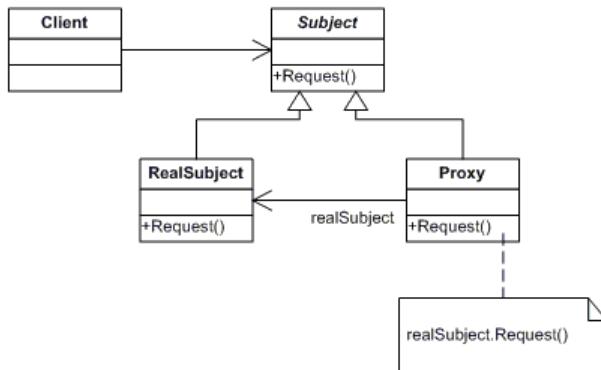


- **Flyweight Pattern:** Bir nesne çok az bellek kullanabilir fakat bu nesnenin çokça üretilmesi durumda bellek kullanımı şieseektir. Bu noktada flyweight pattern ile kodun paylaşılması yöntemi ile bellek kullanımı azaltılır. Unity'de shared mesh ve shared material noktasında vardır. Shared meshte bir değişiklik yapılrsa o meshi kullanan bütün objelerde mesh değişir. Flyweight konusunda dikkat edilmesi gereken noktalardan birisi, flyweight bir kez init edilmelidir yani değişmez olmalıdır. Diğer nokta ise bir fabrika tarafından üretilip, ortak bir havuzda tutulmalarıdır. Bu sayede istek durumunda eşleşen bir flyweight varsa isteyene verilir yoksa yenisi üretilip verilir ve bu üretilen de pool'a eklenir.



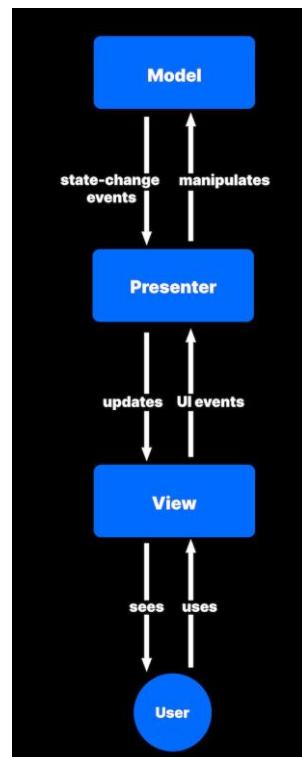


- **Proxy Pattern:** Orijinal nesne için bir yer tutucu sağlar. Nesneye erişim istediğiinden önce veya sonra bir şeyle yapmamızı mümkün kılar.



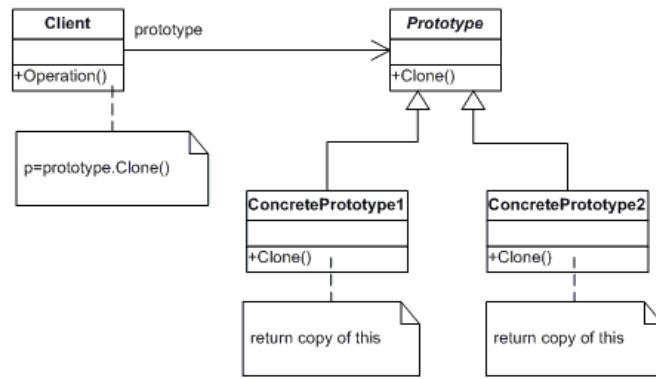
- **MVC Pattern:** Web geliştirmede backend tarafında sıkılıkla kullanılan bir patterndir. Unity oyun motorunun yapımında da kullanılmıştır. Temeli görüntü, data ve logic' i ayırtırarak bakımı ve kontrolü kolay bir mimari oluşturmaktır. Model, View ve

Controller olmak üzere üç temel bileşeni bulunur. Model, data container görevi görür ve herhangi bir logic veya calculation yapmaz. View, kullanıcının sahnede gördüğü görselliği ifade eder. Controller, sistemin beyni olarak çalışır ve runtimededataları calculate etmek ve logicleri barındırmak onun görevidir. Unity özelinde mvc den türetilmiş bir mimari olan mvp mimarisi unity' nin kendi dökümanlarında yer almaktadır. MVP mimarisinin sebebi, runtime' de view' in modeldeki data değişikliklerini izlemesi zorunluluğundan doğmuştur. Model View Presenter mimarisi, view' in modeli sürekli takip etmesi yerine presenter' in model ve view arasındaki aracı rolünü üstlenmesini baz alır. Presenter, controller' in yerine geçer.



### ➤ **Creational Patterns:**

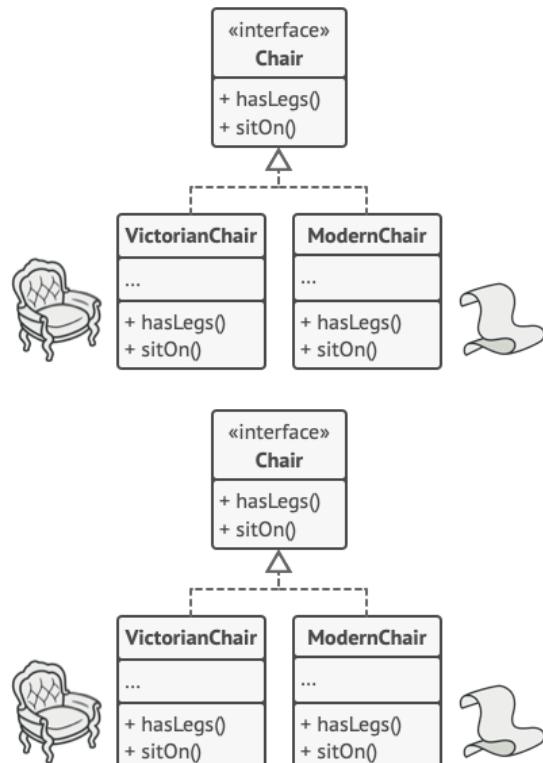
- **Prototype Pattern:** Bir objeyi dışarıdan kolanlamak her zaman mümkün olmaz. Bu sebeple, Prototype pattern, clone işlemini, clone olacak nesneye devreder. Colne olacak nesneye prototype denir. Subclass'a alternatif bir yöntemdir.

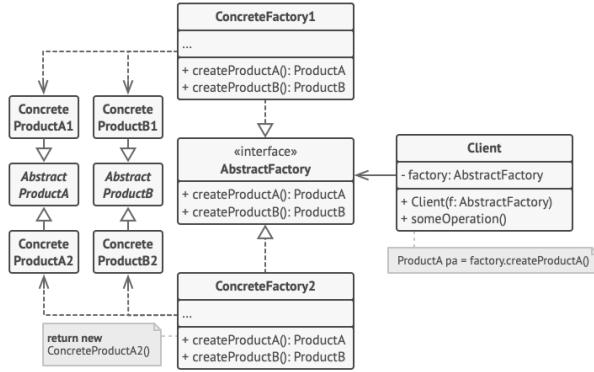


- **Singleton Pattern:** Bir nesnenin sadece bir adet örneği olduğunu garanti eden ve ona global bir erişilebilirlik veren pattern'dır. Mimari açıdan doğru bulunmaz. Core veya manager gibi classlar da kabul edilebilirdir fakat gerekmedikçe kullanılmaması gereklidir. Bağımlılık merkezi yaratması, bütün class'ların işlerini doğru yaptığı vaarsayımlına fazlası ile odaklanmış olması, testinin zor olması, god class riski barındırması ve refactorünün zor olması sebepleri ile mimari açıdan tavsiye edilmemektedir. Anti pattern olarak nitelendirilen pattern'lardandır.

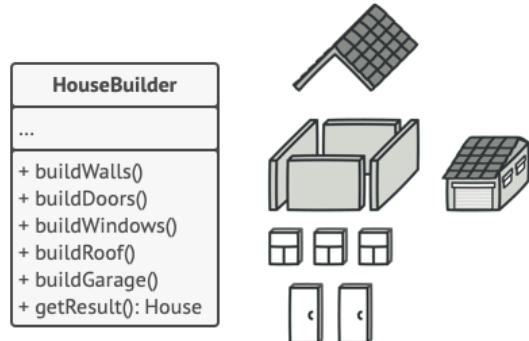
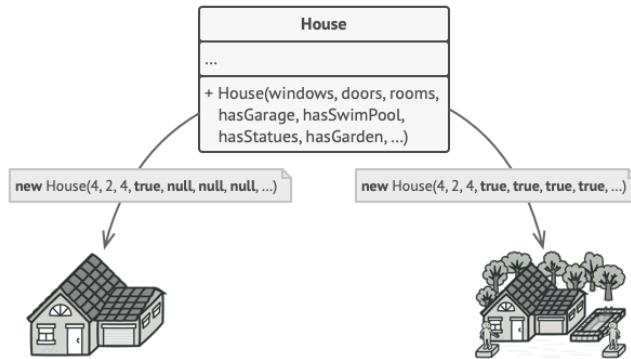


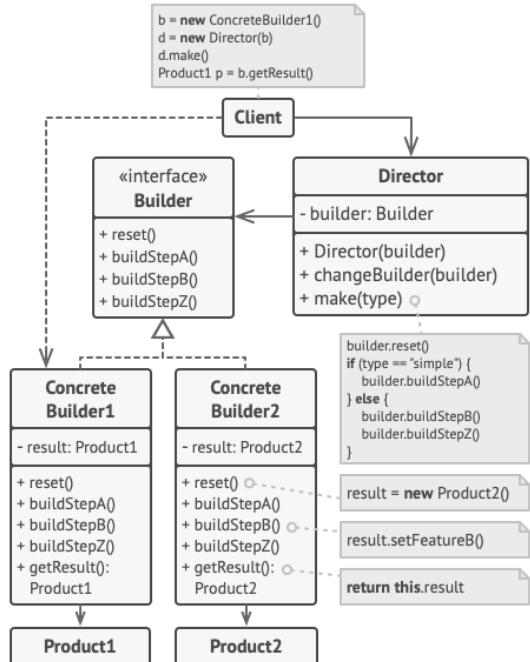
- **Abstract Factory Pattern:** İlgili nesnelerin somut hallerini içermeden üretimelerini sağlar. Aynı aileden olan fakat farklı amaçlara sahip nesneler için yeni bir class oluşturmadan her bir aile için bir factory olacak şekilde çalışılır.



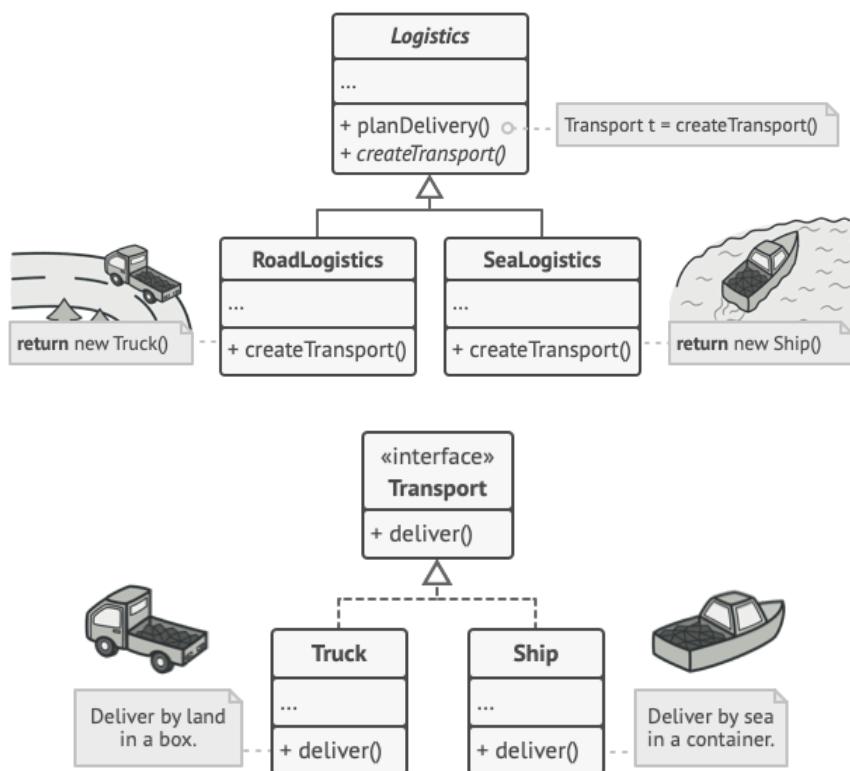


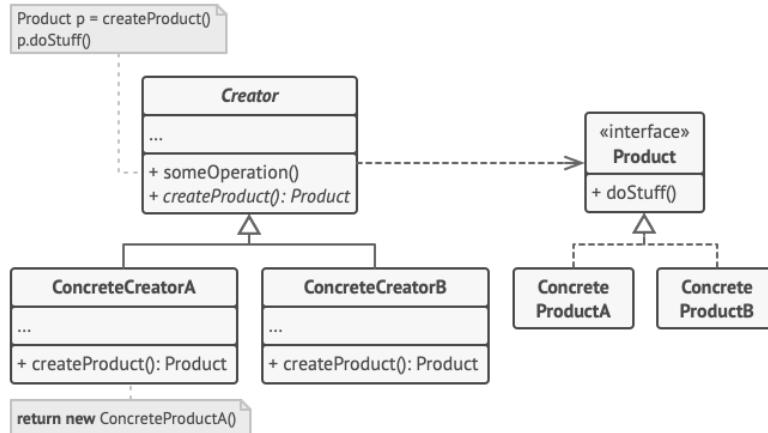
- **Builder Pattern:** Ana prensibi karmaşık nesnelerin aşama aşama oluşturulmasıdır. Aksi durumda constructor veya setup methodu çok fazla parametre alır. Bu karmaşa yerine aşamalı bir çözüm önerir.





- **Factory Method Pattern:** Alt sınıfların oluşturulacak nesnelerin türünü değiştirmesine izin veren patterndir.





## ➤ Game Programming Patterns:

- **Subclass Sandbox Pattern:** Benzer sub classların farklı davranışları olması durumunda, süper classta tanımlanan soyut methodun alt sınıfı uygulanmasıdır.

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  namespace SubclassSandbox.Superpowers
6  {
7      //Superpower parent class which defines all superpowers we can combine in the child classes
8      public abstract class Superpower
9      {
10         //This is the sandbox method in which the children combine superpowers
11         public abstract void Activate();
12
13         //Different superpowers
14         protected void Move(string where)
15         {
16             Debug.Log($"Moving towards {where}");
17         }
18
19         protected void PlaySound(string sound)
20         {
21             Debug.Log($"Playing sound {sound}");
22         }
23
24         protected void SpawnParticles(string particles)
25         {
26             Debug.Log($"Firing {particles}");
27         }
28     }
29 }

```

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  namespace SubclassSandbox.Superpowers
6  {
7      //Child class that defines a single superpower behavior
8      public class SkyLaunch : Superpower
9      {
10         public override void Activate()
11         {
12             PlaySound("Launch sound");
13             SpawnParticles("Dust");
14             Move("The sky");
15         }
16     }
17 }

```

```

1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   namespace SubclassSandbox.Superpowers
6   {
7       //Implementation of the Subclass Sandbox pattern from the book Game Programming Patterns
8       public class GameController : MonoBehaviour
9       {
10           private SkyLaunch skyLaunch;
11
12           void Start()
13           {
14               skyLaunch = new SkyLaunch();
15           }
16
17           void Update()
18           {
19               if (Input.GetKeyDown(KeyCode.Space))
20               {
21                   skyLaunch.Activate();
22               }
23           }
24       }
25   }
26
27 }
```

---

- **Type Object Pattern:** Bir nesnenin türünü, türü tanımlanmayan bir nesneye yani Type Object'e değiştirmek için kullanılır. Kalıtımın kullanılmadığı veya kullanılamayacağı anlarda etkilidir. Var olan bir nesneye bir davranış eklemek ile ilgilidir.
- **Componenet Pattern:** Componentler pc lerdeki mouselere benzerler. USB protot ile istediğiniz cihaza bağlar ve kullanırsınız. Unity'de collider, mesh etc. Componentlar zaten vardır. Yeniden kullanılabilirliği sağlarlar. Type Objectten farkı type object kendi kendine bir anlam ifade ederken, component bir gameobject ile anlam ifade eder.
- **Event Queue Pattern:** Bir event ateşlendiğinde, subscribers aynı anda bu durumdan etkilenirler. Event queue'de ise bu etkilenme bir sıra ile yapılır.
- **Game Loop Pattern:** Oyunların çekirdeğinde yer alır. Temel olarak durdurulana kadar devam eden bir while döngüsüdür. Unity'de update ve fixedupdate zaten bir game loop pattern örneğidir.
- **Service Locator Pattern:** Oyun yapmaya başlandığında, çok fazla standartize edilmiş metodlar kullanırız. Bunlara servis denir ve her yerden erişilebilir olurlar. Fakat halen erişildikleri koddan bağımsızdır. Temelinde bir class'ın başka bir class'ı merkezi bir kayıt defteri aracılığı ile bulmasıdır. Singleton' dan farkı aslında her bir önemli class'ı singleton yapmak yerine istek durumunu Service class'ının kontrol etmesi ve daha önce instance ettiği bir istek varsa instance edilen class'ı yoksa da yeni instance edip yarattığı class'ı isteyene vermesidir. Singleton' un bir listeli veya array' li hali gibi düşünülebilir. Unity'de Mathf, random etc. Anti pattern olarak nitelendirilen pattern'lardandır.

```

@4 usages @1 exposing API
public class ServiceLocator
{
    private Dictionary<Type, IService> _services;

    public static ServiceLocator instance = new ServiceLocator();

    @usage
    public ServiceLocator()
    {
        _services = new Dictionary<Type, IService>();
    }

    @usage
    public static void Register<T>(IService service) where T : IService
    {
        _instance._services.Add(typeof(T), service);
    }

    @usage
    public static T Get<T>() where T : IService
    {
        return (T)_instance._services[typeof(T)];
    }
}

```

The screenshot shows the Unity Editor's Project window with the 'Assets' tab selected. The project structure includes various Unity assets like Scenes, Materials, and Models. In the code editor, the `AppStarter.cs` file is open, which contains the following code:

```

1 UnityityEngine;
2
3     ace Game.Scripts
4
5     blic class AppStarter
6
7         [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
8         public void Init()
9         {
10             var testGroup = "A";
11
12             if (testGroup == "A")
13             {
14                 ServiceLocator.Register<IAvertisementService>(new ApplovinAdvertisementService());
15             }
16             else // Test Group B
17             {
18                 ServiceLocator.Register<IAvertisementService>(new IronSourceAdvertisementService());
19             }
20         }
21
22     }

```

The screenshot shows the Unity Editor's Project window with the 'Assets' tab selected. The code editor has switched to the `NullAdvertisementService.cs` file, which contains the following code:

```

1 namespace Game.Scripts
2
3     public class NullAdvertisementService : IAvertisementService
4     {
5         @4 usages
6         public void showRewardedVideo(Action onSuccess, Action onFailure)
7         {
8             onSuccess?.Invoke();
9         }
10
11         @4 usages
12         public void showInterstitial(Action onSuccess, Action onFailure)
13         {
14             onSuccess?.Invoke();
15         }
16
17         public bool isRewardedVideoAvailable()
18         {
19             return true;
20         }
21
22         public bool isInterstitialReady()
23         {
24             return true;
25         }
26     }
27

```

- **Data Locality Pattern:** DOTS'da kullanıcılar sistemdir. Verilerin yan yana dizilmesi prensibine dayanır. Mantiği CPU' nun verimli kullanılmasıdır.
- **Dirty Flag Pattern:** Aslında tek bir bool'dur. Bir değişiklik olup olmadığını kontrol eder. Unity'de rigidbody'nin bir kuvvet uygulanmadığında sleepin olması bu durumdur.
- **Object Pool Pattern:** Objeleri ihtiyaç duyulduğunda sürekli üretmek yerine belirli bir sasyıda üretip, ihtiyaç dahilinde çağırmak ve obje ile işimiz bittiğinde geri objeyi poola gönderme prensibine dayanır. Memory kullanımını azaltarak performans artışı sağlar. Sık sık oluşturulması gereken nesnelerin varlığı durumunda kullanılır.

## ATTRIBUTES

### ➤ Unity Engine Attributes:

- **SelectionBase:** Çok child'a sahip bir objeye scenede tıklandığında selectionbase attributesine sahip olan class'ı barındıran objeye tıklanmış sayılacaktır. Kola tıkladığımızda da bacaga tıkladığımızda da ilk olarak mesh barındırmayan, parent obje olan, player objesini tıklamayı sağlayabiliriz.
- **Header:** Başlık ekler. Altında bulunan fieldların en üstünde [Header("Başlık")] yer alır.
- **Tooltip:** Değişkenin üzerine editörde Mouse getirdiğimizde [Tooltip("Not")] ilgili yazının çıkışmasını sağlar.
- **ContextMenuItem:** İlgili değişkene sağ tıklandığında parametre olarak verilen methodu çağırabilmeyi sağlar. [ContextMenuItem("item name", "functionname")].
- **ContextMenu:** İlgili methodun, classın sağ üstündeki settings iconuna tıklandığında orada gözükmesini ve tıklandığında çalıştırılmasını sağlar.
- **Serializable:** MonoBehavior veya scriptable olmayan classları editörde görünür ve üzerinde oyananabilir kılar.
- **SerializeField:** Public olmayan değişkenleri editörde görünür ve üzerinde oynanabilir kılar.
- **HideInInspector:** İlgili değişkeni inspectorde görünmez ve değiştirilemez yapar.
- **MinAttribute:** İlgili değişkenin alabileceği minimum değerini belirlememizi sağlar.
- **Range:** İlgili değişkenin alabileceği değer aralığını belirlememizi sağlar.
- **Multiline:** String değişkenlere inspectorda belirlediğimiz kadar line ekler.
- **Space:** [Space(x)] x piksel boşluk ekler.

### ➤ Unity Editor Attributes:

- **AddComponentMenu:** İlgili class' i üst layouttaki component bölümünün içine ekler. [AddComponentMenu("A/B/C")] şeklinde olabilir. Aynın zamanda gameobject'e addcomponent dediğimizde de orada bizim belirlediğimiz şekilde kodu ekleyebiliriz.
- **MenuItem:** Üst layout'a static bir methodu eklememizi sağlar. Özellikle ClearAllData gibi methodlarda sıkılıkla kullanılır.

## ALGORITHMS AND LIBRARIES

### ➤ ALGORITHMS:

- **Fisher-Yates Shuffle:** Bir array veya list' i rastgele dizmeye yarar.

```
public static void Shuffle<T>(this IList<T> ts) {
    var count = ts.Count;
    var last = count - 1;
    for (var i = 0; i < last; ++i) {
        var r = UnityEngine.Random.Range(i, count);
        var tmp = ts[i];
        ts[i] = ts[r];
        ts[r] = tmp;
    }
}
```

## ➤ Mathf:

- **Deg2Rad:**  $(\pi * 2) / 360'$  tır. Dereceden radyana dönüştürme işlemini yapar.
- **Epsilon:** En düşük floattır.  $5 + e = 5$ ,  $5 - e = 5$  iken  $0 + e = e$ ,  $0 - e = -e$  dir.
- **Infinity:** Sonsuz bir değer verir. Genellikle raycastlarde görürüz.
- **P:**  $3.14159265358979\dots$  😊
- **Rad2Deg:**  $360 / (\pi * 2)$ ' dir. 1 radian = 57.29578 degree.
- **Abs:**  $-10.5'$  i  $10.5$  olarak ve  $-10'$  u  $10$  olarak dönürtür. Otomatik olarak negatif atar.
- **Acos:** İlgili değerin ark-kosinüsünü dönürtür.
- **Asin:** İlgili değerin ark-sinüsünü dönürtür.
- **Atan:** İlgili değerin ark-tanjantını dönürtür.
- **Atan2:** İki parametresi vardır. Tan değeri  $y/x$  olan açıyı radyan cinsinden dönürtür. (7,8,9 ve 10. Maddelerdeki ilgili değerler radyan cinsinden dönerler.)

```
public class ExampleClass : MonoBehaviour
{
    public Transform target;

    void Update()
    {
        Vector3 relative = transform.InverseTransformPoint(target.position);
        float angle = Mathf.Atan2(relative.x, relative.z) * Mathf.Rad2Deg;
        transform.Rotate(0, angle, 0);
    }
}
```

- **Ceil:** İlgili float değeri eşit veya daha büyük en küçük integer değere dönüştürür.

```
public class ExampleClass : MonoBehaviour
{
    void Example()
    {
        // Prints 10
        Debug.Log(Mathf.Ceil(10.0F));
        // Prints 11
        Debug.Log(Mathf.Ceil(10.2F));
        // Prints 11
        Debug.Log(Mathf.Ceil(10.7F));
        // Prints -10
        Debug.Log(Mathf.Ceil(-10.0F));
        // Prints -10
        Debug.Log(Mathf.Ceil(-10.2F));
        // Prints -10
        Debug.Log(Mathf.Ceil(-10.7F));
    }
}
```

- **Clamp:** 3 parametre alır. İlk parametre value, ikinci parametre minValue ve üçüncü parametre maxValue'dır. Value değerini min ve max arasına kilitler.
- **Clamp01:** Clamp ile aynı işi  $0$  ile  $1$  arasında yapar. Değer  $0$  dan küçükse  $0$ ,  $1$  den büyükse  $1$  dönürtür.
- **ClosestPowerOfTwo:** İkinin katlarını baz alır. Girilen değere en yakın ikinin katını dönürtür.

- **Cos, Sin, Tan:** Kosinüs, sinüs ve tanjant değerlerini dönderirler.
- **FloatToHalf:** Float değeri, 16 bitlik hale getirir.
- **HalfToFloat:** Float değeri, 32 bitlik hale getirir.
- **Floor:** Girilen değerden küçük ya da eşit en yakın değeri dönderir.
- **InverseLerp:** a ve b değerleri arasını yüzde 0 ile 100 arasına alır ve 0 ile 1 arası değer dönderir.
- **Lerp:** Minden maxa t referansındaki sürede değer dönderir.
- **LerpAngle:** Lerp ile aynı mantıktadır fakat 360 beyond rotatelerde daha doğru sonuç verecek şekildedi. Transform.Rotate içerisinde kullanılabilir.
- **Log:** İlgili değerin belirtilen tabanda logaritmasını dönderir.
- **Min ve Max:** İlgili değerin min ve max sınırlarını belirlememizi sağlar.
- **MoveTowards:** Current değeri, target değere, maxDelta üzerinden taşır. Lerp ile benzerdir fakat hız asla maxDelta yi aşamaz.
- **MoveTowardAngle:** 360 beyond temelli move towardstır.
- **PingPong:** İlgili değeri arttırıp azaltan bir değer dönderir. Fakat ilgili değerin kendiliğinden artan bir değer olması gereklidir.
- **Round:** 0.5f ve altı ise altına aksi durumda üstüne yuvarlar.  $-10.7 = -11$  ve  $10.2 = 10$ 'dur.
- **SmoothDamp:** İlgili değeri, target değere, referans olarak verilen bir velocity ile, smoothTime baz alınarak ve opsiyonel olarak max speed de verilecek şekilde arttırır.
- **SmoothStep:** Lerp gibi ama başta hızlanıp sona doğru yavaşlayarak bu işlemi yapar. Animatifdir.
- **Sqrt:** İlgili değerin kökünü alır.

### ➤ **Time:**

- **CaptureDeltaTime:** Oyun içerisinde ekran görüntüleri almamıza yardımcı olmasına oynatma zamanını yavaşlatıp hızlandırmamıza imkan veren özellik. Bu sayede karelerin arasına istediğimiz kadar zaman koyabiliyoruz. Unity'nin belirttiği üzere özellikle görsellerden bir gif veya video oluşturmak istiyorsanız işinize yarayacak bir yöntemdir.
- **CaptureFramerate:** CaptureDeltaTime özelliği gibi çalışır, ancak zaman aralığını vermek yerine saniyede elde etmek istediğiniz kare sayısını bu değere atamamız gerekiyor. Yukarıdaki örneğin aynısının captureFramerate halini görebilirsiniz.
- **DeltaTime:** Bir önceki kare ile şu anki kare arasındaki zamanı elde etmemizi sağlayan özellikle. Bu değer aynı zamanda Update fonksiyonlarımızın da hangi zaman aralıklarıyla çalıştığını gösteren bir değerdir. Saniye cinsinden değer döndürür ve sadece değer okuma yapabiliriz. Yani bu değeri değiştiremeyiz. İki frame arasında geçen zamanı verdiği için değişkendir. Bu durum onu fixeddeltaTime'den ayıran temel özellikle.
- **FixedDeltaTime:** FixedUpdate gibi fonksiyonların kullandığı sabit zaman aralığının değeri. MonoBehaviour içerisinde kullandığımız FixedUpdate dışında, Rigidbody vb fizik hesaplamalarında da bu değer kullanılmaktadır. Saniye cinsinden değer döndüren bu özellik, aynı zamanda değerini değiştirmemize de imkan tanır. Son fixed

update framesinden itibaren geçen süredir ve sabittir. Bu sebeple fizik motorlarında kullanılır.

- **FixedTime:** FixedUpdate fonksiyonlarının tümünün çalışmaya başlamasından itibaren geçen süreyi elde etmemizi sağlar. Saniye cinsinden oyun başlangıcından itibaren geçen zamandır.
- **FixedUnscaledDeltaTime:** FixedDeltaTime'nin Time.TimeScaleden etkilenmeyen halidir.
- **FixedUnscaledTime:** FixedTime'nin Time.TimeScaleden etkilenmeyen halidir.
- **FrameCount:** Toplarda geçmiş kare sayısını elde etmemizi sağlar. Kare sayısının hesaplanması başlaması ancak bütün Awake fonksiyonları bittikten sonra gerçekleşecektir.
- **InFixedTimeStep:** Bu kod çağrıldığında FixedUpdate gibi sabit aralıklı bir fonksiyondan çağrılmış çağrılmadığını elde etmemizi sağlar. Eğer bahsedildiği gibi FixedUpdate vb sabit aralıklı çalışan bir yapı içerisindeysek, true değeri döndürecektr.
- **MaximumDeltaTime:** Bir karenin, update işleminin işlem süresinin alabileceği maksimum değeri belirlememizi veya elde etmemizi sağlar. Fizik veya benzeri sabit aralık kullanan sistemler (örneğin FixedUpdate) bu değeri kullanarak işlem sürelerini ayarlayacaklardır. Özellikle Garbage Collector ve fazla işlem gerektiren fizik işlemlerinden kaynaklanan kare sayısı azalmalarını engellemek için kullanılabilir. Unity değer aralığı saniyenin 10'da biri (1/10) ve 3'te biri (1/3) olacak şekilde ayarlanması tavsiye etmektedir.
- **MaximumParticleDeltaTime:** Particle objelerinin update fonksiyonlarının sürebileceği maksimum değeri elde etmemizi sağlar. Eğer update işlemi bu sürenin üstüne çıkarsa, update işlemleri çoğalarak daha ufak update işlemleri haline dönüşür. Ufak bir değer verildiğinde, particle için daha doğru sonuçlar elde edilse de, iş yükü artacaktır. Büyük değerler verildiğinde ise particle daha az doğruluğa sahip sonuçlar verecektir ama performans artacaktır.
- **RealtimeSinceStartup:** Uygulamanın başlatılmasından itibaren geçen geçen gerçek zamanı elde etmemizi sağlar. timeScale değerinden etkilenmez.
- **SmoothDeltaTime:** DeltaTime ile olduğu gibi, kareler arasında zamanı elde etmemizi sağlar. Ancak, önceki kare arasındaki zaman değerleriyle karşılaştırılması ve buna göre hesaplanması ile daha yumuşak bir değer elde etmemizi sağlar.
- **Time:** Classın çalışmaya başladığı andan itibaren geçen zamanı saniye cinsinden elde etmemizi sağlar.
- **TimeScale:** Zaman çarpanı olarak da belirtebileceğimiz bu özellik, oyun içerisinde işlemler gerçekleştirken zamanı nasıl ele alacağını belirlememizi sağlar. En basit örnekle, 0 değerini verirsek oyunda fizik işlemleri, kullanıcıdan veri alma gibi bütün işlemler duracaktır, 2 değerini verirsekte, 2 katı hızında işlemler gerçekleşecektir. Standart haliyle 1 değerine sahiptir.
- **TimeSinceLevelLoad:** En son sahnenin yüklenme anından itibaren geçen zamanı saniye cinsinden elde etmemizi sağlar.

- **UnscaledDeltaTime:** TimeScale değerinden etkilenmeyecek şekilde, kareler arasında geçen zamanı elde etmemizi sağlar.
- **UnscaledTime:** Classın başlangıcından beri geçen zamanı elde etmemizi sağlar. Yine aynı şekilde, timeScale değerinden etkilenmeyen bir değişkendir.

## MULTITHREADING

### ➤ JOB SYSTEM AND BURST COMPILER:

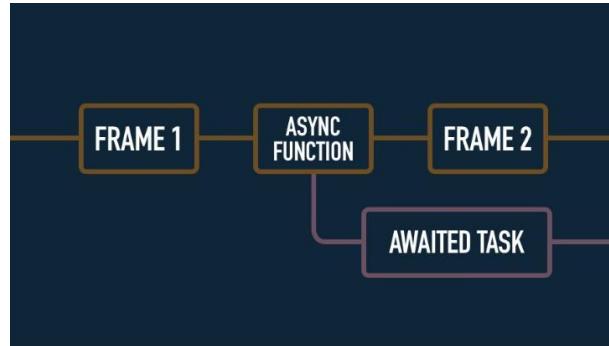
Bilgisayarlar, çıktıyı çalıştırırken aslında c# hakkında bir bilgiye sahip değillerdir. En alt katmanda assembly dili ile çalışırlar. Günümüzde assembly yazmak anlamsız olsa da bilgisayarlar yazdığımız kodları assembly' e dönüştürüp çalıştırır. Kisaca program başladığında bilgisayarın anlayacağı dil olan assembly' e dönüştürme yapılır ve bu dönüştürme işlemine compiler denir. C# ve java gibi dillerde bu dönüştürme doğrudan yapılmaz. C# ta .net framework, intermeidary language adı verilen dil ile çalışır ve bu dil aslında bir arabulucu dildir. IL bütün .net dillerinde vardır. IL ise assembly' e çevirme işlemini yapar. Unity' de ise bu durum biraz daha farklıdır. Unity birden fazla platforma çıktı verme ihtimali olduğu için direkt assembly' e çevirme işlemi yapılmaz çünkü mac ve windows assembly' leri farklı iken android' de assembly diye bir durum bulunmaz. Bu yüzden Unity IL çıktısını IL2CPP den geçirerek c++ diline çevirir. C++ genel bir dil olduğu için bütün platformlara çıktımı mümkün değildir. Bu sebeple son hal çıktısı c++ olarak alınır.

- **Job system:** Unity normalde tek thread üzerinde çalışırken bu durumu multithread' e çevirmemizi sağlar. Multithread ise tek bir çekirdek (Main thread) üzerinde kod emirlerinin sıra sıra işlenmesi yerine bazı kod işlemlerinin farklı çekirdeklerde alınarak paralel şekilde işletilmesidir. Avantajı optimizasyonu arttıracı ve birden fazla thread üzerinden paralel programlama yapmamızdır. Fakat render thread' e bir etkisi bulunmamaktadır yani grafik sebepli optimizasyon kayıplarını çözmez. Aynı zamanda native array kullanımı zorunludur yani normal array kullanmana izin vermez ve native array garbage collector tarafından silinen bir şey olmadığı için manuel dispose yapılmalıdır aksi halde memory leak durumu yaşanır. C++ taki gibi new edilenleri kendimiz yok etmeliyiz.
- **Burst compiler:** Job sistemiyle verimli bir şekilde çalışması için yapılan bir derleyicidir. IL' yi LLVM kullanarak derler. [BurstCompile] attribute' sini job struct'ının üzerine koyarak kullanılır. Job gibi primitive türleri destekler. Enum ve string desteklemez.

### ➤ ASYNC Programming:

Multithreading konusu ile yakından ilgilidir. Tek bir thread üzerinde işlemlerimiz sırası ile yapılacaktır ve bir sonraki işlem önceki işlemin bitişini beklemek zorundadır. Multithreadingte ise, tek bir thread yerine yeni işlemimize de bir thread atayarak kendisinden önceki thread' in bitişini beklemek zorunda kalmamasıdır. Optimizasyonu arttırsa da çok fazla thread kullanılması da optimizasyonu negatif yönlü etkilemektedir. Async programlama ise, tek thread üzerinde önceki işlemin bitmesini beklemeden yeni işlemi devreye alabilmektir. Tek thread' in birden fazla işlem için sıra bazlı yani sync/ eş zamanlı kullanılması yerine async/ eş zamansız ve sırayı ignore ederek kullanılmasıdır. Unity özelinde cloud data noktasında sıkılıkla kullanılır çünkü ana işlemleri engellemeden, cloud işlemleri gibi

ağır işlemleri de yapmamıza olanak sağlar. Monobehaviour classlarda IEnumarator ile işlemi tek frame yerine birden fazla framede yapabiliyoruz ve benzeri bir yapı monobehaviour olmayan classlarda async programming ile mümkündür. Async programlamada methodun belirli bir kısmı ya geciktirilir ya da belirli bir noktadan itibaren devam etmeden önce bir görevin/Task tamamlanması beklenir. Methodun tür belirteçinden önce async yazılması ile method async method olur fakat bu yeterli değildir. Await anahtar kelimesi ile bir görevin tamamlanmasını beklemek gereklidir. Bunun için await Task.Delay(milisecond) yazarak zaman bazlı bekleme yapılabilir. Async programlamada async methodun elle silinmesi gereklidir ve bu işlem için o methodda construct edilecek olan cancellationtoken kullanılabilir. Task bu noktada async görevimizi tanımlamaktadır. Task yaratmak için, System.Threading.Tasks kütüphanesini kullanmalıyız.



```
void CountToTen()
{
    int number = 0;
    while (number < 10)
    {
        number++;
        Debug.Log(number);
    }
    Debug.Log("I've finished counting!");
}
```

Copy

```
async void CountToTenAsync()
{
    int number = 0;
    while (number < 10)
    {
        number++;
        Debug.Log(number);
        await Task.Yield();
    }
    Debug.Log("I've finished counting!");
}
```

Copy

```
async void Wait()
{
    Debug.Log("Give me a second...");
    await Task.Delay(1000);
    Debug.Log("Ok, I'm done.");
}
```

```
async void Start()
{
    await MyFunction();
    Debug.Log("All Done!");
}
async Task MyFunction()
{
    // Waits 5 seconds
    await Task.Delay(5000);
}
```

```
async void Start()
{
    await Task.Run(() => Count());
    Debug.Log("All Done!");
}
void Count()
{
    for (int i = 0; i < 10000; i++)
    {
        Debug.Log(i);
    }
}
```

## DATA ORIENTED PROGRAMMING

### ➤ ECS:

Açılımı entity component system' dir. Temelinde DOTS' ı kullanan bir mimaridir. Klasik Unity mimarisi, OOP ilkelerine uygun olarak tasarlanmış GameObject - Mono Behaviour - Data And Game Loop System mimarisini kullanırken, ECS ise DOP ilkesini baz alarak tasarlanmış olan DOTS' ı kullanır. ECS, Entity - Component/Data - Game Loop System şeklinde ifade edilebilir. Entity, components ve system kelimeleri ayrı ayrı anlam ifade etmektedir.

- **Entity:** GameObject' in yerini alan en basit tabiri ile bir şeydir. Datadan gelen verilere göre işlem yapar. İçeriği sadece yapacağı işlemin logic kısmıdır.
- **Component:** Saf data görevi görür ve üzerinde değişiklik yapılamaz. Buradaki datalar ile entityler beslenmektedir.
- **System:** Game Loop' tur yani update yapılarını barındırır. Updateler içerisinde componentlar işlenir ve tekrar entity' lere gönderilir böylece DOP ilkesine uygun bir şekilde data ve logic ayırtırılmış şekilde bir game loop oluşturulmuş olur.

### ➤ DOTS:

- **Data oriented programming (DOP):** Programlamada alışla gelmiş OOP yaklaşımına alternatif olan bir yaklaşımdır. OOP yaklaşımında kalıtım gereği tek tek üretme ve

üretilen nesnelerin kendi görevlerini yerine getirmesi esasken, DOP yaklaşımında nesnelerin toplu şekilde yönetilmesi ilkesi esastır. DOP' ta yönetme sadece dataları değiştirerek yapılır ve bu yönü ile flyweight patternla yakınlaşır. Unity özelinde OOP yaklaşımına göre üretilen nesneler sahnede yüzlerce olmaları durumunda yüzlerce update methodu oluşacakken, DOP yaklaşımında ise sadece 1 update methodu üzerinden bütün nesneler sadece dataları değiştirilerek yönetilirler ve görevlerini yerine getirirler. Bu durum DOP' u OOP den çok daha performanslı bir yaklaşım haline getirir. Günümüzde büyük projelerde OOP yerine genelde DOP ilkesi kullanılır fakat OOP deki inheritance ve polymorphism' in sağladığı modülerliğin olmaması da eksiyi yanlarından birisidir. Developer friendly olmaması da eksiyi yanlarından birisidir.

- **Data oriented technology stack:** Data oriented design (DOD) anlayışından türeyen data oriented programming (DOP) ilkesine uygun sistemdir. Job ve Burst sistemleri ile çalışabilir. ECS mimarisi DOTS' ı kullanmaktadır.

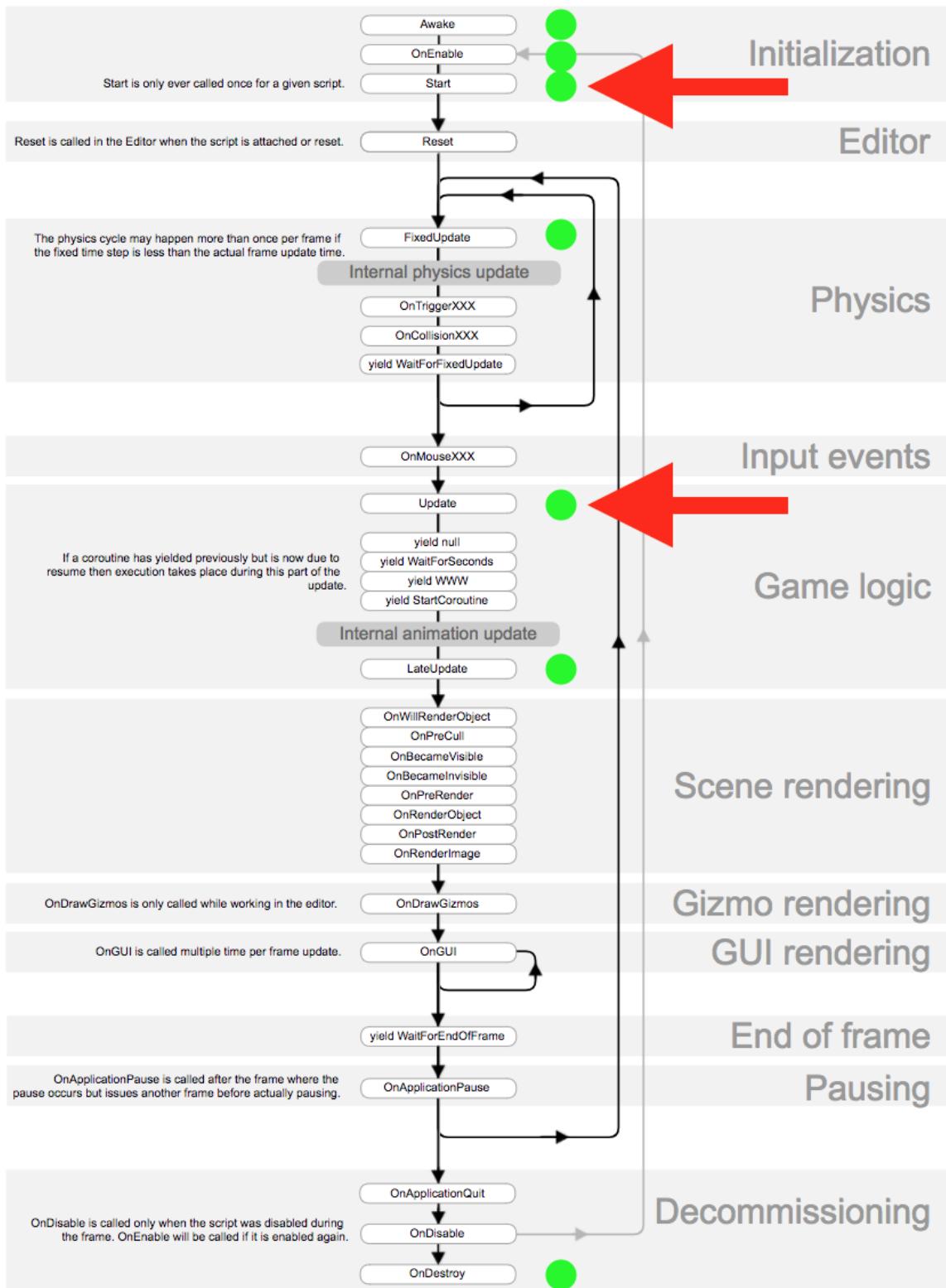
## KEYWORDS

### ➤ **Partial, Sealed, Internal Keywords:**

- **Partial:** Partial anahtar kelimesi, derleme/compiling işleminde bu anahtar kelimeye sahip class, method, interface veya structların tek bir nesne gibi compile edilmesini sağlar. Class tarafında iki farklı class' ı sanki tek classmış gibi kullanmamızı sağlar. Method tarafında ise, methodun partial classta yer olması gereklidir ve erişim belirteçleri kullanamaz, bunun yerine methodu kapsülleyerek başka bir classtan erişebiliriz. Partialların isimleri aynı olması, aynı namespacede yer almaları ve aynı erişilebilirliğe sahip olmaları zorunludur. Partial anahtar kelimesine sahip olanlardan bir tanesi bile static veya abstract yapılrsa veya mühürlenirse hepsi bu özelliğe sahip olmuş olur.
- **Sealed:** Türkçe karşılığı mühürlemektir. Kullanıldığı class veya methodun kalıtımını engeller. Sealed bir classtan başka bir class türetilemeyeceği gibi sealed belirteçi konmuş bir metoddan da override işlemi yapılamaz.
- **Internal:** Class, struct, method ve propertylere bir erişim belirteci olarak verilebilir. Publicten farkı, sadece aynı proje veya namespaceden erişmek mümkündür.

## EXECUTION ORDER/LIFE CYCLE

## ➤ Unity Life Cycle / Order Of Execution:



## OPTIMIZATION TECHNIQUES

### ➤ CPU & Memory Optimization:

- **Fundamentals of CPU & Memory:** CPU, basit aritmetik işlemleri yapar. Çalıştığımız programların işlemlerini/komutlarını input olarak alır ve output verir yani bu komutları işler. Aynı anda birden fazla iş yapamaz ama saniyede binlerce farklı komutu işleyebilir. CPU bir komutu 4 ana döngüde çalıştırır. Bunlar;
  - Fetching: Çalıştığımız programın t anında yapmak istediği komutu ram' den control unit' e aktarılır.*
  - Decoding: Contrul unit, bu komutu aritmetic logic unit' enin anlayacağı şekilde makine diline çevirir.*
  - Executing: Makine diline çevrilmiş komut ise aritmetic logic unit' e tarafından işlenip çıktısı tekrar ram' e verilir.*
  - Storing: Ram depolama işini yapar.*

Ram ve CPU bu doğu sebebi ile çok yakın temasta çalışır. En yaygın komutlar MOV(Move), ADD(Addition), SUB(Subtraction) ve RET(Return) komutlarıdır. Memory, sistemin açılması için gerekli hafızadır. Çok küçük parçalara ayrılmıştır ve bu parçalara hafıza gözleri/hücreleri denilmektedir. Her bir hücre 1 bayttır. Her bir hücre de unique address ile tutulur. Bu adres üzerinden hücreye erişilip verileri alınır veya yazılır. Memory tipleri temelde 2 ye ayrılır. İlkı primary memory' dir. İkincisi ise secondary memory' dir. HDD, SSD, Compact Disk etc. Secondary memory klasmanına girer. Ram ve Rom ise primary memorydir. Ram, SRAM(Stack) ve DRAM(Heap) olmak üzere ikiye ayrılırken, Rom ise PROM/EPROM ve EEPROM olmak üzere ikiye ayrılır. Rom' un açılımı read only memory' dir ve silinemez veriler产生. Silinmemesi gereken pc ilk açıldığında kullanılan işletim sistemi verileri ve bios etc. Ram' in açılımı random access memory' dir. DRAM heap memory kısmını kullanırken, SRAM ise stack kısmını kullanır. RAM, runtime memory ile ilgili iken, ROM başlangıç anı ile ilgilidir. Runtime de işletim sistemi verileri tutulacaksa SRAM(Stack)' de tutulur. SRAM memory' nin bir array' idir. Last in first out prensibi ile çalışır ve value type' lar tarafından kullanılır. DRAM(Heap) ise, reference type' ları depolayacak parçaların bulunduğu, onlar için ayrılmış bir alandır. Buradaki parça kelimesinin karşılığı CHUNK' tır ve CHUNK' lar verileri tutma/depolama görevini yerine getirir. DRAM' de istenilen şekilde veri ekleme ve çıkarma yapılabilir. İki farklı memory alanına ihtiyaç duyulmasının sebebi ise, value type' ların boyutları daha önceden belli iken, reference type' ların boyutları önceden belli değildir. Value typler; bool, byte, char, decimal, double, enum, uint, ushort, float, int, long, sbyte, short, struct ve ulong type' larıdır. Reference type' ler ise; class, interface, delegate, object, string type' larıdır. Primitive değişkenler Stack' te tutulurken aynı zamanda Heap' e gidecek olan reference type' ların da bir referansı/adresi bulunur bu primitive type' lar gibi direkt veriye değil adrese ulaşmaktadır. Bu sebeple reference type kelimesi kullanılır. Bu noktada GC kavramı da ortaya çıkmaktadır. Heap bellek bölgesinde olmasına rağmen Stack bölgesinde herhangi bir adresi olmayan verilerin kontrolü ve temizlenmesi GC' nin işidir. Stack ve Heap alanlarının farkları;

- Stack' te memory allocation/tahsis statikken, Heap' te dinamiktir.*
- Stack' te doğrudan depolama varken, Heap' te depolama doğrudan değildir.*
- Stack' te yeniden boyutlandırılamazken, Heap' te değişken bir boyuta sahiptir.*

- d) Stack'ı hızlı yapar.
  - e) Stack'ı LIFO mantığı varken Heap'ı yerleştirme her zaman yapılabilir ve serbesttir.
  - f) Stack'ı sadece sahibi olan thread erişebilir ve görüntüleme yapabilir fakat Heap'ı her thread bu izne sahiptir.
  - g) Yinelemeli çağrılarda memory Stack'ı hızlıca dolarken bu konuda Heap daha yavaş dolar.
  - h) Stack'ı sadece bir thread tarafından kullanılabılırken, Heap uygulamanın bütün parçaları tarafından kullanılabilir.
  - i) Stack'ı yer almazsa, .net "StackOverflowException" hatası verir.
  - j) Stack, integral type'ları, primitive type'ları ve object'ların referanslarını tutar.
  - k) Stack'ı yoktur fakat Heap'ı vardır.
- **Introduction to Optimization:** İki kelimeyle bir temeli vardır. Budget ve profilling. Budget kaynağı yani sistemin gücünü, profilling ise bu kaynağın nasıl kullanıldığını içeren süreçtir.
  - **Garbage Collection:** Memory' de işi bitenlerin temizlenmesi işlemidir. Memory' de heap kısmında çalışır. Referans'ı kaybedilen yani işlemi bitmiş olanların temizlenmesidir. Teknik olarak maliyetli bir işlemidir ve oyunlarda anlık fps droplar yaşanmasının ana sebebidir. Unity' de c# dili kullandığımız için manuel yapmamız gerekmek fakat C++ ta manuel olur ve yapılmaması memory leak durumuna sebep olur. Unity' de Project Settings içerisinde Use Incremental GC seçilerek tek frame yerine birden fazla frame' de GC yapılabilir ve default değer olarak açıktır. Oyunun reklam izlenmesi, loading screen gibi geçiş anlarında manuel olarak GC çağrımak faydalıdır. Sistemi bir frame'de aşırı zorlayan işlemlere "Spike" denmektedir ve GC de genel bir spike sebebidir. "Every frame cost", uygulamanın bir frame' si için harcadığı maliyettir ve optimizasyonda önemlidir. Telefonların ısınması ve fazla şarj tüketmesi bu veri ile yakından ilişkilidir. "Loading time" ise genel yükleme süresidir. "Fragmentation" kavramı ise, memory ve GC konusunda çok önemlidir çünkü; heap kısmında var olan bir nesne daha sonra yok olduğunda heaptan ondan arta kalan bir delik oluşturmuş olur ve bu delikte boş bellek olarak sayılır fakat bu noktada memory ardışıklığını kaybetmiş olacağı için silinenden büyük veya silinenden küçük boyutta bir nesne memory' e gelidiğinde yiğin parçalanması yani fragmentation durumu ortaya çıkaracaktır. Bavulun verimli kullanılmadığı için 5 nesne alırken aslında boşluklar ardışık şekilde düzenlense 10 nesne alacak olması durumudur. GC, fragmentation'ı çözmek için, memory' de önce allocate edilmiş ama sonra free hale gelmiş irili ufaklı ama çok sayıda olan ve ardışık olmayan boşlukları kaydırır böylece sistemden heap büyültme talep etmeden zaten elde olan alan verimli kullanılmış olur. Aksi durumda aslında 500mb kullanan sistem için 1, 2 gb alan ayrılmış olur. DOTS bu noktada önemlidir çünkü, verileri boşluk oluşmayacak şekilde saklar. Array ve list' te bu konuda farklıdır. Array ardışık saklama yaparken, list ayrı yerlerde sakladığı için array kullanmak daha optimizedir. Unity özelinde ek bilgi olarak, Unity 5 sonrasında öncesinin aksine, hierarchy deki nesneler, hierarchy deki gibi ram' e sıralanmaktadır ve bu sebeple çok fazla SetParent kullanmak sürekli ramde' de sıra kaymalarına sebep olacağı için

optimize değildir ve büyük boşluklara sebep olabilir. SetParent yerine, parent objenin transform.childCount ile child sayısı setlenmelidir.

- **Unity Profiling Tools:**

- a) *Unity Profiler: Frame, frame uygulamayı incelememizi sağlar. Sadece CPU değil, GPU, Audio etc. konularında da bilgi verir. "Editor loop" kısmı önemsenmez çünkü editorun harcadığı kaynaktır. Çıktı alındığında editor loop zaten kalmaz. İncelemede "Player Loop" kısmı önemlidir.*
- b) *Memory profiler: Programın ne kadar memory kullandığını gösterir. Snapshot alınarak real time dan veri alınabilir. Memory profiler tree map üzerinden ise çok detaylı bir izleme yapılabilir. Memory profiler default olarak yoktur bu sebeple package managerden import edilmelidir. Snapshotlar kıyaslanabilir ve bu durum çok faydalıdır.*
- c) *Code coverage: Kullanılmayan class ve metodları gösterir.*

- **Unity Optimization Techniques:**

- a) *Flyweight pattern: Onlarca objenin ortak verilerini tek bir bellek bölgesinden okumasıdır. Kullanımı patternlar kısmında anlatıldığı gibidir ama kısaca çokça olan ve aynı işi yapan nesnelerin her birinin ortak olan datalarını instance etmesindense bu datanın bir kere yaratılıp hepsinde verilmesidir.*
- b) *Scriptable objects: Aşağı yukarı flyweight pattern'a yakındır. Varlık amacı ortak data görevi görmektir ve isteyen nesnenin kullanabilmesidir. Farkı scriptable object unity' e özel olduğu için daha hızlı compile edilmektedir.*
- c) *Object pooling: Bu da genel olarak patternlar kısmında anlatıldığı gibidir. Ek olarak fragmentation' u önleme etkisi ile çok etkilidir.*

- **Script Optimization:** Mükün oldukça array kullanmak daha optimizedir.

Kullanılmayan unity eventleri (update etc.) kaldırılmalıdır ve unity eventlerinde tek bir yerden ateleme yapmak yani dar boğaz yapmak son derece optimizedir. Animation.StringToHash kullanımı faydalıdır çünkü string de versen unity bunu yapacak. GameObject.AddComponent yapılmamalıdır. Update içinde string new leme ve değer atama yapılmamalıdır. Referanslar cahce edilmelidir. Coroutine içinde yield return new WaitForSeconds yerine değişken olarak WaitForSeconds oluşturulmalıdır. Unity tarafında bazı objeler external call ile yani c++/native taraftan. Çağrılmaktadır. Transform etc. olan objeler cache edilerek kullanılmalıdır. LinQ kullanımını azaltılmalıdır çünkü Garbage bırakırlar. String manipülasyonlarıçoska string builder kullanılmalıdır.

- a) *Array: Rastgele erişim hızı yüksektir. Resize işlemi maliyetlidir. Memory tarafı için nesneler belli ise kullanmak gereklidir. List' e göre daha optimizedir.*
- b) *List: Dinamik size' a sahiptir. Esnekdir. Yüksek memory kullanımına sahiptir ve rastgele erişim hızı düşüktür.*
- c) *Dictionary: Key temelli hızlı bir erişim sistemi vardır. Bellek tüketimi yüksek olabilir. Hashleme işleminin kalitesi performansı etkiler.*
- d) *Linked list: Listedeki her bir elemanın/node birbirlerine link şeklinde bağlı olduğu bir veri yapısıdır. Her eleman bir değer ve bir referans içerir. List gibi dinamiktir ve runtime' de ekleme çıkarmalar kolaydır fakat her eleman kendisinden bir önceki ve bir sonraki elemanın bilgisini tutar. Array' in aksine*

*ram' de yan yana dizilmesi gerekmez çünkü her node öncekinin ve sonrakinin yerini bildiği için ramde istedikleri yerde olabilirler.*

- e) **Queues:** *FIFO* prensibine dayanan yani ilk giren ilk çıkar prensibine sahip dinamik bir veri yapısıdır. Bir eleman çıkarılmak istendiğinde en öndeği eleman çıkar, yeni eleman eklendiğinde ise o sıranın en sonuna geçer.
- f) **Stacks:** *LIFO* prensibine yani ilk giren son çıkar prensibine dayanan dinamik bir veri yapısıdır.
- g) **Hash tables:** *Dictionary gibi key ve value' den oluşur. Farkları ise; dictionary aksine oluştururken veri tipini belirtmek gerekmez ve dictionary generic bir yapıdır fakat hashtable non genericdir. Bu durum gereği generic yapılar gibi runtime' de gereksiz cast kullanılmasını önleme ile compile time' de type safe değişken kullanımını zorlama yani bu sayede run time' de muhtemel tip dönüşümü hatalarını önleme özelliğine sahip değildir.*
- h) **Binary search tree:** *Full veya Strict olmak üzere iki türü vardır. Temelinde ode' lardan oluşan ve her bir node' un en fazla 2 child node'a sahip olduğu veri yapılarından bir tanesidir. Node bir veri yapısının en temel birimidir ve veri içerebileceği gibi aynı zamanda diğer node' lar ile arasında bir bağlantı da bulundurabilir. En tepede binary root yer alır ve root' tan küçük değere sahip olanlar sol, büyük değere sahip olanlar sağ kola geçerler. Bu kural sol ve sağ tarafta yer alan ilk node' lerin root sayılması ile onlar için de uygulanır.*  
*Böylece ağaç dalları şekli alınmış olur. Binary search tree kullanarak oluşturan bir yapıda, bir elemanı silmek, eklemek veya bulmak gibi işlemler hızlı gerçekleştirilebilir. Burada bir elemanı bulabilmek için tek tek tüm elemanları dolaşmak yerine her seferinde veri setini ikiye bölgerek ilerleme sağlanır.*
- i) **AVL tree:** *Herhangi bir root' un iki alt dalının yükseklik farklarının birden fazla farklılık gösteremeyeceği, kendi kendini dengeleyen bir ikili arama ağacıdır (BST). Herhangi bir zamanda birden fazla farklılık gösterirlerse, bu özelliği geri yüklemek için yeniden dengeleme yapılır. Classic BST' ye göre daha karmaşıktır.*
- j) **Red-black tree:** *Veriyi ağaçta tutarken ağacın dengeli olmasını sağlayan bir algoritmadır. Her node kırmızı veya siyahdır. Root her zaman siyadır. Bütün yaprak düğümler yani altı olmayan yalnız olanlar siyadır ve son kuralı bütün kırmızı düğümlerin çocukları siyadır.*

- **Unity Optimization Tips:** Sahne hierarchy' si çok karışık olmamalıdır çünkü sistemin sahne içi searchlerinin maliyeti artmaktadır. Resources folder kullanımında dikkatli olunmalıdır çünkü build ile beraber çıktıda bu folder içindekiler de gelir.

## ➤ **GPU Optimization:**

- **Fundamentals of GPU:** Grafik renderlarından sorumlu component. CPU' nun aksine sıralı değil paralel işlem yapar.
- **Difference between CPU and GPU:** Mimarileri farklıdır. CPU sıralı şekilde tek zamanlı iken GPU paralel işlemler yapabilir çünkü aralarındaki çekirdek farkı çok yüksektir. CPU tek thread için optimize edilmiştir. GPU grafik, ai, kripto gibi alanlarda kullanılırken, CPU daha sistemsel ihtiyaçlar içindir. Aralarındaki farkın temeli ise CPU' nun kullanıldığı alanlarda yapacağı işlerde sıralı işlemin temel alınması ve genelde

önemli olanın sıralı işlem yapması olmasıdır. GPU' da ise bir pikselin basılması için diğer pikselin bitmesini beklemesine gerek olmaması durumu ve bu tarz durumların mühendislikte çokça bulunmasıdır. CPU ve GPU ların çekirdeklerinin tükettiği enerji miktarı da farklıdır. CPU çekirdekleri GPU çekirdeklerinden daha hızlı işlem yapar ama GPU çokça işlemi aynı anda yapar ve GPU çekirdekleri CPU çekirdeklerine göre daha yavaştır.

- **Rendering Optimization Terms:**

- Draw call: Bir şeyi ekran'a çizmek içindir. İçerisinde transform, scale, shader ve texture, mesh, vertices, diğer render parametreleri vardır. GPU bu call ile gelen parametrelere göre çizimi yapar. Maliyetlidir ve CPU üzerinde de işlem yükü ağırdır çünkü görüntünün CPU' ya da iletilmesi gereklidir. Optimizasyon için genel olarak draw call azaltılmaya çalışılır Mesh X material count ile kaç draw call gönderileceği bulunur.*
- Set pass call: GPU bir objeyi çizdiğinde, çizim için gelen parametreleri tutar. Buna GPU' nun state' i denebilir. Yeni bir call geldiğinde bu state değişir ve bu değişimi set pass call yapar ve maliyetlidir. Unity özelinde çokça material sebep olur. Kisacası GPU tarafından rendering state' i değiştirme işlemidir. Materyal sayısının az olması bu sayıyı azaltır çünkü draw call sayısı düşecektir.*
- Batching: Bir draw call paketidir. Her nesne için draw call göndermek yerine çokça nesneyi tek draw call ile göndermektir. Unity' de mesh rendererde ne kadar material varsa o mesh için o kadar draw call üretilir. Unity' de stats içerisinde draw call yazmaz onun yerine batches yazar.*
- Others: Tris (Triangles): Modelin kaç üçgeni vardırı gösterir.*
- Verts (Vertices): Bir 3d model üç boyutlu uzaydaki tanımıdır yani kaç noktadan oluştuğudur.*
- Shadow casters: Gölgedir ve bunlar da draw call oluşturur.*

- **Unity Profiling Tools:**

- Frame debugger: Projenin rendering sırasını gösterir ve bu sayede frame frame renderi görebiliriz.*
- Unity Profiler Rendering: batch count, set pass count, tris ve vertices sayıları görülebilir. Alt kısımda yer alan bilgilerden; used textures o framede kullanılan texture sayısını ve toplam boyutunu içerir, render textures normal textureden farkı runtime' de değiştirilebiliyor olmasıdır ve bu kısımda sahnedeki kullanılan render texture sayısını ve toplam boyutunu verir, used buffer gpu' nun çizdiği nesnelerin datalarını tutar, vertex ve index buffer da vardır.*
- Unity Profiler UI: Layoutları ve render bilgisini alıyoruz. Render bilgisi saniye cinsindendir. Layout bilgisi ise Unity ne kadar performans harcıyor bilgisini gösterir. Unity özelinde Canvas içerisindeki bir nesnenin kapatıp açılması veya image değişikliği veya da scale değişiklikleri bütün UI' in tekrar renderlenmasına sebep olur. Bu sebeple canvas group alpha kullanımı ve canvaslara bölme optimizasyon açısından önemlidir.*
- Unity Profiler UI Details: Batches, vertices ve markers bilgileri yer alır. Markers kısmı buton interactionlarını gösterir.*

- **GPU Optimization Techniques:**

- a) *Static batching: Sahnedeki gameobjectlerin transform değişikliği yapmayacak olması, mesh rendereri olması ve aynı material ve shader' a sahip olmaları gereklidir. Bu objeler static batch edilebilir. Hem GPU hem de CPU açısından optimizedir. CPU için objenin statik olması artık transform' unun da hesaplanması demektir. Unity' de nesne static olarak işaretlenerek kullanılır. Static nesneler tek bir draw call çağrıları gönderirler ve böylece tek framede çizilirler, sonuç olarak GPU için optimize olurlar.*
- b) *Dynamic batching: Statik olamayan objelerde kullanılır. Aynı material' e sahip, aynı scaleye sahip ve aynı sırada render edilen objeleri runtime' de batchlemeye çalışır. Shader bazında vertices sayısı önemlidir. 180 ve altı vertices' te vertex position, normal, UV0, UV1, tangents kullanılırken, 300 vertices ve altı shaderlarda vertex position, normal ve tek UV parametre olarak kullanılır. Bu sebeple CPU üzerinde olumsuz etki yapabilir. CPU üzerinde, object management (track same object), transformation updates (during runtime), batching, GPU ile senkronizasyon ve render ile senkron olma işlemlerine sebep olur. Küçük vertices değerlerine sahip, hareket etmesi gereken nesnelerde olumlu sonuç verir.*
- c) *Enable gpu instancing: Aynı material ve aynı mesh' e sahip nesneler için kullanılır. Semi-static nesnelerde son derece etkilidir. Kısmi hareketleri olan nesneleri GPU' ya tek bir draw call ile gönderebilir.*
- d) *SRP batcher: URP projelerde oto açık gelir ve gpu instancing' i ezer. Unity scriptable render pipeline' lerde kullanılır. CPU yükünü ve draw call sayısını azaltmak için kullanılır. Diğer batch tekniklerinin aksine aynı material kullanma zorunluluğu yoktur. Aynı shader variant' ini kullanmaları yeterlidir. Unity özelinde standart material' i kullanan bütün nesneler URP projelerde SRP batcher tarafından aynı shader variant' ini kullandıkları için beraber batch edilirler ve tek bir draw call gönderilir. Diğer tekniklerin aksine sadece mesh renderer' i değil skinned mesh renderer' i da kapsar.*
- e) *Reducing shadow usage: Herbir gölge minimum bir draw call üretir. Gölge konusunda olabildiğince mesafeye bağlı kalite düşürme, çözünürlük düşürme ve cascade sayısını azaltma yoluna gidilmelidir. Shadow resolution' i düşürmek draw call sayısını değiştirmez fakat render edilecek görüntünün boyutunu azaltacağı için faydalıdır. Static batching kullanmak gölge konusunda da pozitif sonuç verir.*
- f) *Material shader optimization: Unity' nin standart shader' i özellikle mobil tarafta gereksiz miktarda parametreye sahip olmasından dolayı maliyetlidir. Bunun yerine legacy shader' lar kullanılabilir. Daha az parametreye sahip shader ile standart bir shader' dan elde edilen görüntü elde edilebilir ve bunun mümkün olduğu durumlarda da az parametrelili shader' i kullanmak iş yükünü azaltır.*
- g) *Texture optimization: Texturelerin size' ları küçültülebilir. Atlas mapping kullanarak 20 nesnenin tek bir material' i kullanması sağlanabilir böylece bu yirmi nesne static batching yapıldığın da tek bir draw call oluşacaktır. Sprite*

*atlas kullanılması ile bütün texturelerin tek bir texture olarak okunması gibi yöntemler kullanılabilir.*

- h) **Combine meshes:** Tek bir objenin çok fazla child meshlerden oluşması durumunda bir nesne için 10 20 belki daha fazla draw call olabilir. 3D artistler tarafından nesne tek bir mesh olarak gelmelidir fakat o taraftan halledilemiyorsa da unity içerisinde farklı assetler ile mesh combine mümkündür.
  - i) **For UI:** Layout group yerine farklı canvaslar kullanmak, sprite atlas kullanmak, word space veya screen space camera canvaslarda camera boş kalmamalı çünkü unity her framede arıyor, ui runtime de en ufak bir değişiklikte baştan sonra render edilir bu yüzden ui içi değişiklikten olabildiğince kaçınılmalıdır, ui elementlerinde animator kullanmakta her framede UI'ın baştan çizilmesine sebep olur, UI özelinde olabildiğince UI'ı sabit tutmak temel optimizasyondur. Best fit ve rich text kapatılmalıdır yine aynı sebeplerle, raycast almayacak elementlerin raycast targete kapatılmalıdır, outline ve shadow gerekmedikçe kullanılmamalıdır.
  - j) **For mobile:** Animasyonlar 30 frame olmalı, target fps 30 olmalı, 3D modellerde olabildiğince düşük vertices değerleri olmalı, write/read enabled eğer runtime de mesh veya texture ile işimiz yoksa kapalı olmalı çünkü hep GPU hem de CPU da birer örneğinin tutulmasına sebep olur. Camera clipping/render distance'si olabildiğince düşük olmalıdır.

**Unity Optimization Priority:** Bütün optimizasyon teknikleri bir arada kullanılabilir at unity ilk olarak static batching ve SRP'yi, ikinci olarak GPU instancing, ve son amada dynamic batching sırasına göre yapmamızı önermektedir.

## ➤ Adaptive Performance:

Uyarlanabilir Performans, mobil cihazınızın termal ve güç durumu hakkında geri bildirim almanızı ve uygun şekilde tepki vermenize olanak tanır. Örneğin, daha uzun bir süre boyunca sabit kare hızları sağlamak ve termal daralmayı önlemek için cihazdaki sıcaklık eğilimlerine ve olaylara tepki veren uygulamalar oluşturabilirsiniz.

# ASSETS MANAGEMENT

## ➤ **Addressable Assets System:**

Unity 2019 LTS sürümü ile gelmiştir. Adreslenebilir varlıklar ile içerik yönetimini daha verimli hale getirmeyi amaçlar. Geleneksel olarak assetsleri, içeriği verimli bir şekilde yükleyecek şekilde yapılandırmak zor bir işlemidir. Bu sistem ile assetlerin tamamının otomatik bir şekilde yüklenmesi yerine, geliştiricilerin assetleri fiziksel lokasyonlarından ayırmasını sağlar ve runtime'de dinamik bir şekilde asset load ve update işlemlerinin yapılmasını kolaylaştırır. Dinamik yükleme özelliği ile assetlerin isteğe bağlı bir şekilde yüklenmesini sağlar, bu sayede ilk yükleme sürelerini ve bellek kullanımını azaltır. Assetler bağımsız olarak güncellenebilir ve

yalnızca gerekli değişiklikler indirilerek update boyutu küçültülür. Assetler remote olarak barınabilirler böylece kullanıcının tüm oyun paketini indirmesine gerek kalmadan güncelleme alması mümkün olur. Aynı zamanda bu sistem asset bağımlılıklarının yönetimini yönetmeyi kolaylaştırdığı gibi verimli bir yükleme sağlar. Addressable Assets ile yineleme döngüleri kısaltılır çünkü içeriğe adresiyle atıfta bulunulur. Böylece bir address referansı ile içerik hemen alınır. Sistem yalnızca adressteki içeriği değil, aynı zamanda içeriğin bağımlılıklarını da döndürür. Sistem asset' in tamamı hazır olduğunda bilgilendirme yapar. Böylece içerik döndürülmeden önce bütün meshler, shaderlar, animasyonlar ve daha fazlası yüklenmiş olur. Address yalnızca assetsleri yüklemekle kalmaz, aynı zamanda bunları kaldırma görevini de üstlenir. Referanslar otomatik olarak sayılır ve böylece potansiyel memory problemleri bulmak güçlü bir profiler ile mümkün hale gelir. Sistem karmaşık bağımlılık zincirlerini eşler ve anlar bu sayede assetler taşındığında veya yeniden adlandırıldığında bile paketlerin verimli bir şekilde paketlenmesine olanak tanır. DLC üretmek ve azaltılmış uygulama boyutunu desteklemek için assetsler, hem local hem de remote dağıtım için kolayca hazırlanabilir.

- **Varlık/Asset:** Aslında oyunu yaratmak için kullanılan içeriklerdir. Bu noktada prefablar, audio clipler, textureler, animationlar ve daha fazlası varlıktır.
- **Adres/Address:** Bir şeyin bulunduğu konumu tanımlar. Bir cep telefonu aradığınızda telefon numarası onun adresi görevini görür. Kişinin nerede olduğundan bağımsız bir şekilde telefon numarası yani address üzerinden ona ulaşılabilir. Unity' de bir asset, addressable olarak işaretlendiğinde bu asset artık addressable asset olur ve her yerden çağrılabılır. İster localde isterse content delivery network' te olsun, sistem onu bulacak ve geri gönderecektir. Adresi üzerinden tek bir addressable yüklenebileceği gibi, birden fazla addressable de tanımlayabileceğimiz özelleştirilmiş bir grubu kullanarak da yüklenebilir.

### ➤ **Asset Bundle:**

Assetleri (Textures, models, audio clips etc.) verimli yükleme ve depolama için ayrı dosyalar halinde paketlemesine olanak tanıyan bir özellikle. Unity' nin kendine özel zipi gibi düşünülebilir. Path olarak Application.persistentDataPath kullanılır. Bu bundleler development sürecinde oluşturulur ve genelde runtimede yüklenir. Belirli assetlerin veya asset gruplarının ihtiyaç durumunda yüklenmesini sağlayarak memory kullanımını ve loading süresini azaltır. Asset bundellar belirli platformlar için compresslene bilir ve optimize edilebilir. Böylece dosya boyutu ve yükleme performansı iyileştirilir. Bundleler ana oyun paketinden ayrı olarak dağıtılabılır ve güncellenebilir. Böylece oyuncunun oyunun tamamını indirmesine gerek kalmadan, yeni assetler veya güncellemer olması sağlanır. Hem localden hem de clouddan bundle çekilebilir. Bu noktada version numaralndırma da önem kazanır çünkü, bazı indirilebilir içeriklerin bazı gereksinimleri olabilir. Bu sebeple bundlelerde de manifest olması gereklidir. Bu noktada oyunun güncel sürümü ve minimum sürümü de manifestte bulunur ve dlc yüklemesi esnasında sürüm kontrolü yaparak bağımlılıklar kontrol edilmiş olur. Temel olarak bundle sayesinde dlc yapmak kolaydır. Oyuna yeni eklenecek featureler için de kullanılabilir. Bu noktada bundle ile yeni görevler, yeni içerikler eklenebilir. İndirme işlemi için Unity Web Request kullanılır.

- **Unity Web Request:** WWW protokolünü unityde kullanmamızı sağlar. WWW protokülü internetin temel dilidir ve küresel ölçekte verilerin değişim tokus kurallarını belirler. Get, Post, Put, Delete, Patch ve Head olmak üzere altı temel işlemi vardır. Unity' de web request ile internetten dosya indirme veya karşılık dosya yüklemeye işe yarar. Uzaktan asset paketleri, kullanıcı bilgisi etc. Özellikleri yüklememizi sağlar.
- **Kısıtlamalar:** Scriptler bundle hale getirilmez.
- **Addressable Asset System' den farkı:** Addressable, assetlerin yönetilmesine ve loadlanmasına odaklanarak, dinamik yükleme, sürüm oluşturma ve uzaktan barındırma gibi özellikler sağlar. Asset management için daha esnek ve ayrıntılı bir yaklaşım sunar. Asset Bundle ise, öncelikle assetlerin verimli bir şekilde paketlenmesine ve yüklenmesine odaklanır. Development sürecinde oluşturulurlar ve assetlerin modüler yüklenmesine olanak tanırlar. Bazı durumlarda beraber kullanılabilirler. Assetleri ayrı dosyalar halinde paketlemek için Asset Bundle kullanılır ve bu bundleleri addressleyerek yüklenmeleri ve güncellenmeleri yönetilebilir.

## CLOUD SERVICES

### ➤ **Firebase:**

- **Kullanım yerleri:** Firebase; özünde bir cloud service' dir ve kullanıcı datalarını tutmamızı sağlar. Teknik olarak bir backend service' dir. Node js veya php kullanmadan yani kendi sunucumuzu yazmadan, firebase serverleri üzerinden backend işlemlerimizi yapmamızı sağlar. Unity firebase' e rakip olarak "Unity Gaming Service" hizmetini sunmaktadır. Firebase' nin real time data base yerine, fire store data base hizmeti de vardır fakat ikinci yazılım ve okunan data başına price alırken, ilki connection başına alır. İlkini tek bir connection gibi gösterip çok ucuz miktara çok fazla kullanıcı işlemi yapılabilir yani optimize edilebilir. Bunun sebebi ise real time data base' nin kullanıcı ile veri tabanı arasında bir socket açması ve bu socket üzerinden anlık olarak datanın senkron olmasını sağlamasıdır. Aslında bu manipülasyonda araya bir katman yazılarak tek bir socket gösterilir ve tek socket ücreti ödenir.
- **Remote config:** Projeler için sunucu üzerinde data tutulmasını sağlayan ve uygulamayı güncellemeden belirli dataları güncellememizi sağlayan bulut hizmetidir.
- **Authentication:** Kullanıcıların login işlemleri içindir. Google, guest facebook etc. desteği vardır.
- **Cloud function:** Firebase CLI kullanarak yazılan metodları cloud' a senkron ettiğimiz işlevdir. Tetikleme clientte yapılır ve bununla senkron edilen cloud function çalışır. Hacklenmesini istemediğimiz core işlemleri (coin earn atc.) burada yapmak güvenliği arttırmıştır. Java script (node.js), type script veya python ile yazılabılır.
- **Source:** <https://github.com/Orhonbey/UnityFirebaseSample>
- **Documantion:** <https://nickel-zircon-67b.notion.site/Unity-le-Firebase-Kullan-m-cb2e90960b94476281076920c991f336>

## MULTIPLAYER

## ➤ Unity Networking:

Networking iki farklı yaklaşım modeline ayrıılır;

- **Master server model:** Bu modelde bütün client' ler tek bir server' a bağlanır server üzerinde oyun mekanikleri döner ve client' lara gerekli cevapları server verir. Bu modelde network performansı server' a bağlıdır. Server maliyeti stüdyoya aittir ve yüksek maliyetlidir.
- **P2P model:** Client' lerden birisi "Host" olarak server görevini de üstlenir. Network performansı client' lerden host olanın network kalitesine bağlıdır. Maliyet düşüktür. Hileye daha açıktır.

Önemli kavramlar;

- **Client:** Oyunu oynayan kullanıcının cihazı.
- **Server:** Oyun işlemlerini yürüttüğüm cihaz.
- **Proxy:** Lobilerin adreslerini tutar. Kullanıcının hangi lobiye bağlanacağını belirler. İstenilen kurallara göre lobiye bağlar.
- **Traffic manager:** DNS kullanan ve client isteklerini sistemin geri kalanına iletten birim.
- **Authentication api:** Client' lerin bağlanması sağlar.
- **Lobby:** Kullanıcıların toplandığı ve oyuna yönlendirildiği yer.
- **Database server:** Kullanıcı bilgilerini içerir, gönderir ve günceller.
- **Game server:** Bütün oyunun döndüğü yerdir. Instance' in her birini oyun gibi düşünebiliriz çünkü kullanıcılar bu instance' lere bağlanırlar. Instance şeklinde her oyunu ayırmak daha optimizedir. Mirror, photon etc.

Genel iletişim protokelleri;

- **HTTP:** Request/Response ile bir istekte bulunulur. Long pool ile de sürekli server' a istekte bulunup server' daki son değişiklikleri isteyen bir yöntemdir fakat long pool yöntemi eskide kalmıştır ve yavaş çalışmaktadır.
- **Socket:** Server ile client arasında bir kanal açılır ve bu kanal 7/24 açıktır. İstediği an iki tarafta istekte bulunup cevap alabilir ve bu socket' in genel mantığıdır.
  - a) **TCP:** *Bu protokol, client veya server' a bir paket gönderdiğinde mesajın ulaşıp ulaşmadığını kontrol eder ve mesajları sıralı gönderir. Paketler sıralı gitmeli ve kaybolmamalı ise bu protokol tercih edilir. Native TCP (Mobilde ve masaüstünde) ve Websocket(Browser üzerinde) olmak üzere ikiye ayrılır. Kart oyunu gibi sıralı oyunlarda daha uygundur.*
  - b) **UDP:** *TCP' nin aksine mesajın ulaşıp ulaşmadığına bakmaz ve mesajı sıralı değil karışık gönderir. UDP için önemli olan mesajı elden çıkartmasıdır. Reliable ve Unreliable olmak üzere ikiye ayrılır. FPS gibi oyunlarda daha yaygındır. Online oyunlarda karşı oyuncunun işinlanır gibi konumunun tam bize ulaşmaması durumları aslında UDP protokolünden kaynaklanır çünkü paket bize doğru şekilde ulaşmadığı için bu görüntü oluşur. UDP, TCP' den daha hızlıdır çünkü mesaj boyutu küçüktür.*

Mirror Kütüphanesi;

- **Mirror server:** Online oyun kütüphanesidir.
- **Network behaviour/Server:**
  - Client RPC:* Server verdiği kararı client' lere client rpc ile gönderir. Bir player' in yaptığı işlemin diğer kullanıcıların ekranında bu işlemin gözükmesi etc.
  - Target RPC:* Server yine bir karar verir fakat spesifik bir client için çalışır.
  - SYNCVARS:* Bir değişkenin server üzerinden bütün client' lere senkron edilmesidir.
- **Network behaviour/Client:** Bütün prefablarımız sahnede olan objelerimiz network behaviour' dan türer.
  - Commands:* Client' in server' a oyun içi konular ile ilgili mesaj göndermesine yarar.
  - Network messages:* Client' in server' a oyuna ilk girişi, çıkış gibi temel mesajları göndermesini sağlar.

