**Design patterns**

کمک می کند که ساختار کد بهتری داشته باشیم.

3 مبحث دارد

* Structural
* Behavioral
* Creational

**Creational**

هدف : نحوه ساختن یک instance از یک شی به چه صورت باشد و سرویس ما به چه شکلی instance را به ما بدهد.

**Structural**

آشنایی با ساختار سرویس کامپوننت و .. به شکل باشند. هدف چگونه یک ویژگی را به یک مفهوم اضافه کنیم و مفاهیم چگونه به هم ترجمه می شوند.

**Behavioral**

برای بررسی رفتار یک سرویس مثلا یا اطلاع رسانی کی رفتار کی انجام شده است. یا زمانی که بخواهیم زنجیره ای از کار ها انجام شود chain of responsibility. هدف این است که شی های یا object یا instance ها به هم دستور دهند یا اطلاع رسانی کنند

Abstract Factory

از نوع creational است.

هدف:

ارائه دهنده یک interface برای خانواده ای از object ها که به هم مرتبط یا وابسته هستند. برای ایجاد مجموعه ای از object ها که به هم مرتبط هستند و اطمینان از این که object ها با هم سازگار هستند.

بدون این که مشخص کند concrete class مشخص کند یعنی کلاس پیاده سازی آن معلوم نباشد.

**Problem**

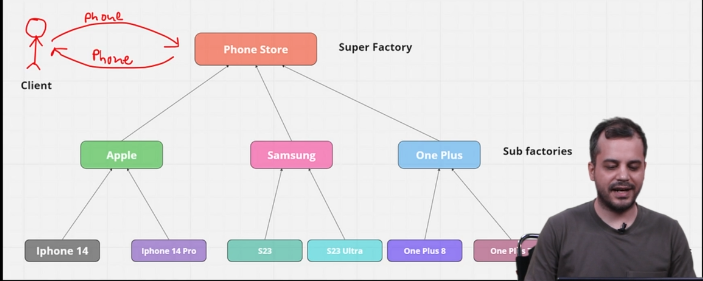
فرض کنید که یک Uiدارید که و در حال توسعه آن هستید و نیازمندی این است که برای ظاهر کار نیازمند تم های برای حالت های light dark , .. هستید و در ui خود نیز دارای ایتم های مانند button checkbox هستند و آن ها را در کنار هم ایجاد کرده اید اما نمی خواهید برای هر حالت یک instace از dark , light ایجاد کنید زیر را تعویض تم و افزودن تم را برای شما سخت می کند.

**Solution**

با ایجاد کردن interface برای object های که به هم ارتباط دارند و در یک خانواده هستند این کار را برای ما راحت می کند.

همانند کارخونه بسته به نیازمندی که دارید مبل رو بهت می دهد اما بسته به کارخونه ظاهر مبل فرق می کند. اما ماهیت همون مبله به شکل دیگری بسته به آن factory

در مثال زیر super factory ما ساخت موبایل است. تعاریف concret در لایه های پایین مثل اپل و سامسونگ که هر کدام موبایل می دهند به سبک خود مثلا S23 و iphone 14

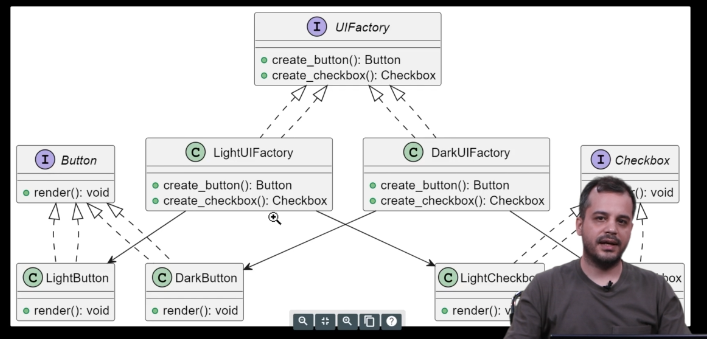


**When to use**

زمانی که سیستم می خواهد مستقل باشد از نحوه ایجاد product ها است در تعاریف بالایی.

می خواهیم یک خانواده مشترک و مرتبط از محصولات داشته باشیم در نهایت.

صرفا به interface نیاز داریم و نه implementation



**Builder**

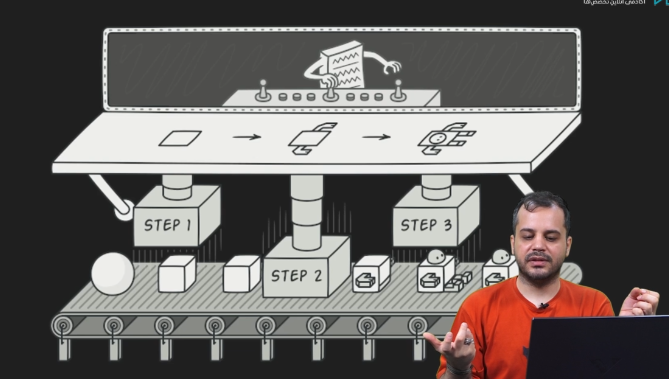
از نوع creational است و هدف مدیریت ایجاد یک complex object را مدیریت کنیم. هدف ایجاد object از یک کلاس است.

**Problem**

می خواهیم یک ساختمان بسازیم و در فرآیند ساخت قصد داریم که بخش های به آن اضافه کنیم مثل استخر حیاط و فضای و سبز و ... با استفاده از روش builder این امر میسر است یعنی متد های مخصوص اضافه کردن این feature ها داریم که کار را برای ما انجام می دهند.

نکته : مهم انجام مرحله به مرحله افزودن ویژگی ها است.

یک مفهوم داریم قصد ساخت object از آن داریم و مرحله به مرحله به آن می افزاییم.



می توانیم قسمت های را حذف کنیم از فرآیند و حتما لازم نیست همه مراحل را انجام بدهیم.

شکستن فرآیند و تقسیم به مراحل کوچک تر که با در کنار هم قرار گرفتن تکمیل می شود.

فرآیند ایجاد کردن باید اجازه بدهد که نمایده های مختلفی از object که ایجاد می شود ساخت.

دارای 2 حالت پیاده سازی است fluent , non fluent

nonfluent همان حالت عادی است

Fluent به گونه ی نوشته می شود که تمامی متد های ما مقدار بازگشتی از نوع همان کلاس را دارند که سبب می شود بتوانیم همانند یک دنباله تمامی متد هارو پشت سر هم صدا کنیم.

**Factory Method**

از نوع creational است برای ایجاد object ها کاربرد دارد.

متدی است که به ما اجازه می دهد که override شود در factory های مختلف از آن استفاده کرد و instance های مختلفی را دریافت میکند.

**Problem**

**Common = مشترک**

در یک رستوران غدا های مختلفی طبخ می شود غذای ایرانی ایتالیای و چینی اما موضوع مشترک بین همه این ها غدا بودن است پس ما یک interface خواهیم داشت که به ما غذا می دهد و سبک های مختلف می توانند instance از آن بسازند.

**UseCase**

زمانی که می خواهید برای کلاس های helper اطلاعاتی را localize کنید و آن ها delegate یا نماینده باشند.

رابط IProduct را داریم و از آن productA , productB را می سازیم.

**Prototype**

از نوع creational است و هدف ایجاد یک object جدید از یک instance است یعنی clone کردن یا copy کردن از instance موجود.

دارای روش های مختلف:

Manual

Constractor

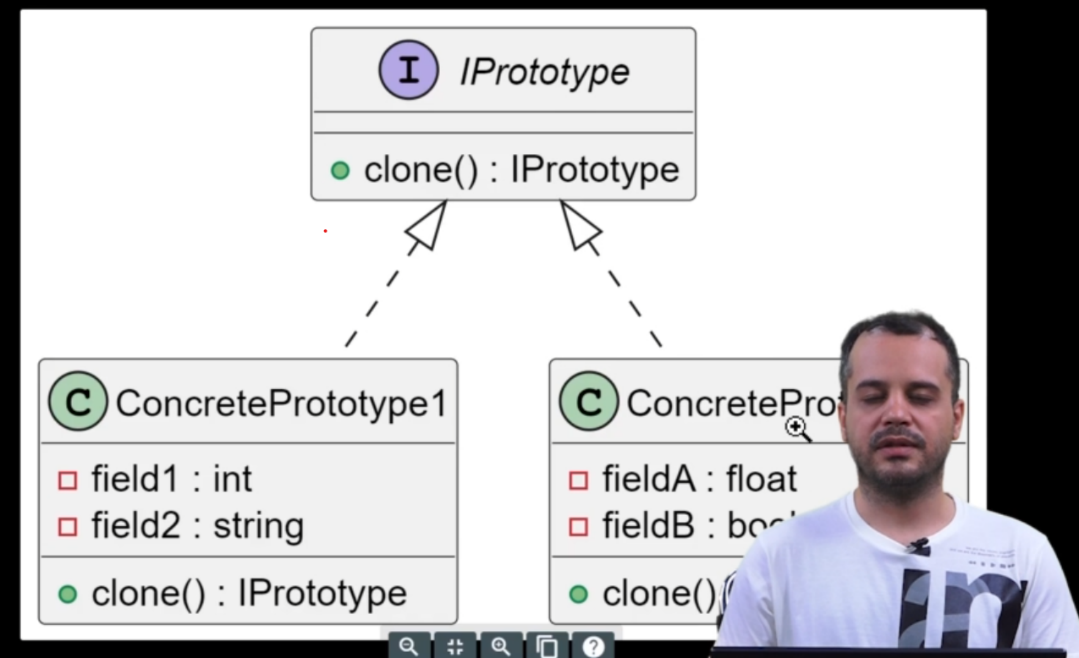
Shallow copy or dep copy دیپ حالتی است که instance ما تو در تو است داخلش کلاس دیگری داره و تو در تو است میخواهیم اون ها هم clone شود اما حالت shallow داخل نمی رود.

Sterilization با استفاده از json

امکانات زبان مثل IClone , Clone

نکته »

میتوانیم موقع clone کردن تصمیم بگیریم به چه صورت انجام شود دقیقا مشابه instance موجود باشد یا تفاوت داشته باشد.



Singleton

از نوع creational است و هدف آن اطمینان از این که یک object از کلاس فقط یکی داشته باشیم. به طور مثال فقط یک filesystem یا windowManager داشته باشیم.

باید نحوه دسترسی مشخص باشد.

Problem

برای جلوگیری از ساخت instance های مختلف کاربرد دارد فقط یک object داریم.

کاربرد در:

Dependency injection

Inversion of control

Ambient context

Service locators

در مواقعی که نیاز داریم فقط یک instance از کلاس باشد استفاده می کنیم باید life cycle و point of access آن مشخص باشد.

**نکته : باید constructor آن از نوع private باشد**

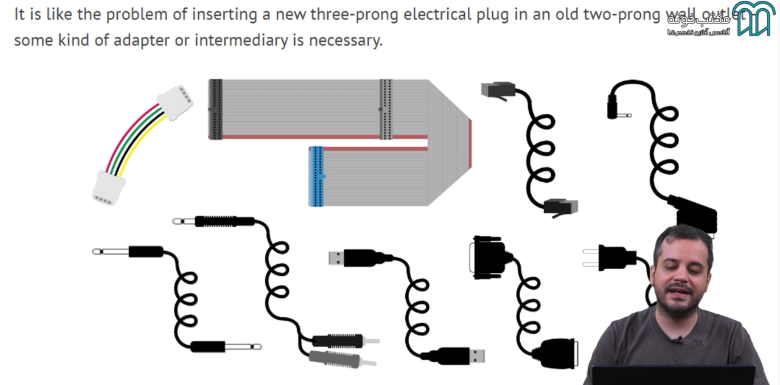
**نکته :**

**در حالتی که یک کلاس که از الگوی singleton استفاده می کند ایجاد می کنید باید مراقب thredSafe بودن آن باشید. اگر چند thread داخل شود دیگر حالت singleton ندارد. باید از lock استفاده کرد یک ترد وارد شود.**

**نوع Lazy<T> تا زمانی که دات نزنیم یک instance نمی سازد کاربرد در performance**

**Creational Design Pattern**

زمانی که یک سری interface یا incompatible ها که به هم دیگر نمی خورد کمک می کند که این ها در کنار هم دیگر قرار گیرند آن های که Incompatible هستند را Compatible می کند.

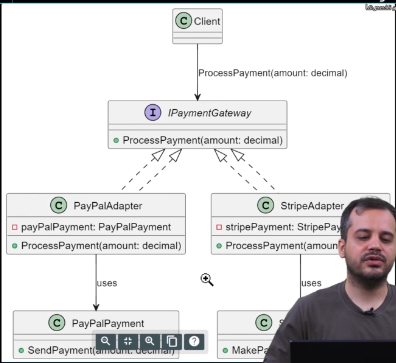


Problem

فرض کنید که یک فروشگاه دارید که با روش های مختلفی می توان اقدام به پرداخت کرد مثل بانک ملت و ملی و پاسارگارد هر کدام درگاه خود را دارند

با استفاده از adapter می توان Interface ایجاد کرد که با استفاده از آن integrate یا یکپارچه کرد.

یک روش برای ترجمه کردن است به متد دلخواه ما در واقع سازگار می کند compatible سازی انجام می دهد.



Bridge

از نوع stractural است و کاری که انجام می دهد این است که abstraction را از implementation جدا سازی می کند که بتوانند جداگانه از هم کارکنند.

در واقع پلی بین 2 سرویس بزرگ یا 2 ماژول است.

مثال:

یک بازی داریم که کلاس abstract است و در پلتفرم های مختلف قصد داریم بازی را اجرا کنیم با استفاده از bridge می توان برای هر پلتفرم برنامه را توسعه دهید.

Bridge vs adapter

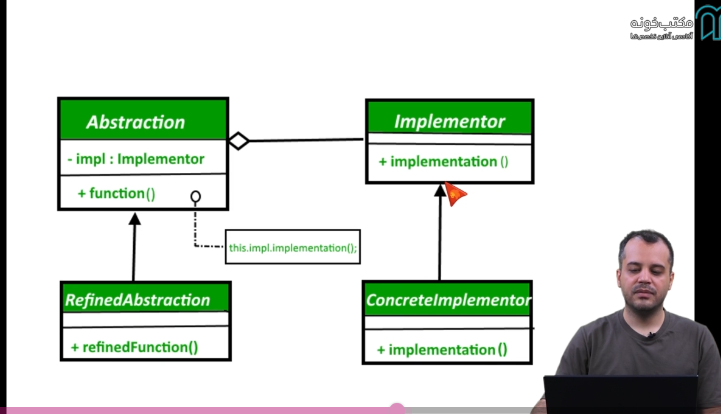
در bridge قسمت های از app که مستقل هستند این را develop می کنیم

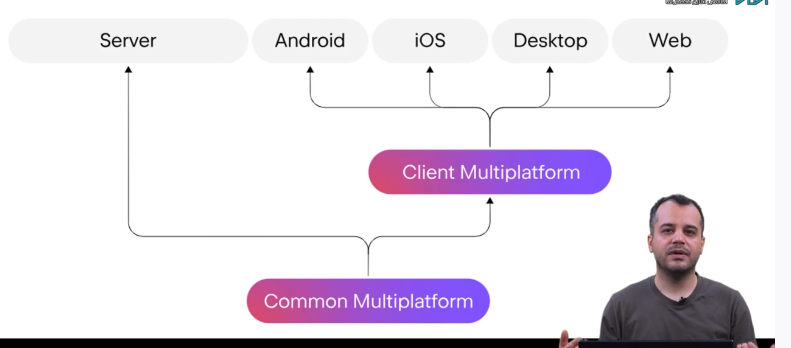
اما adapter در خود app هست و نه قسمت های مختلف داخل سرویس خودم یا App خودم.

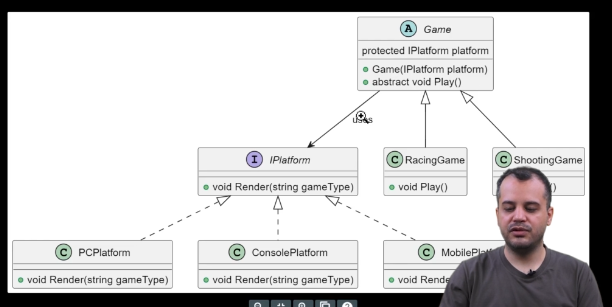
useCase

موقعی که بین پیاده سازی های مختلف در زمان runtime می خواهیم switch کنیم و می خواهیم پیاده سازی را از client خود مخفی کنیم.

همجنین برای share کردن کد خود در پلتفرم های مختلف.







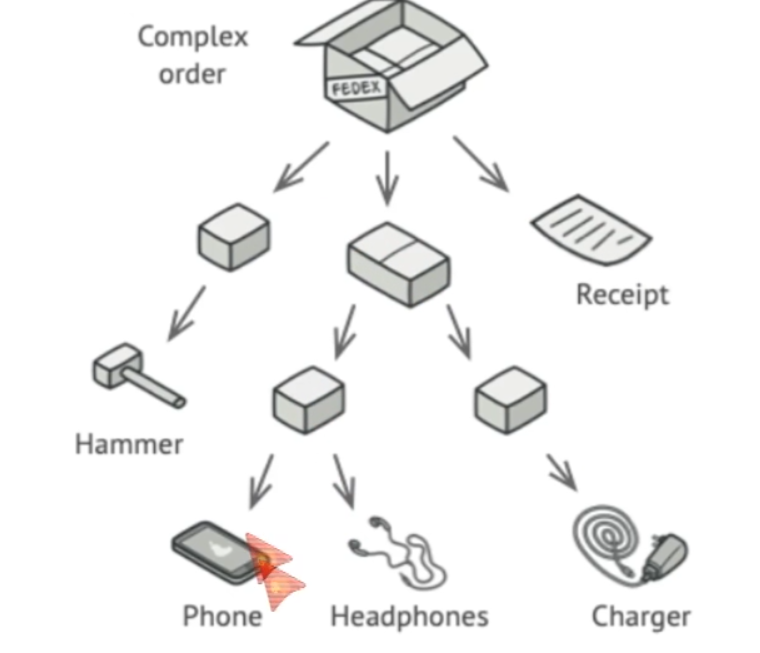
Composite

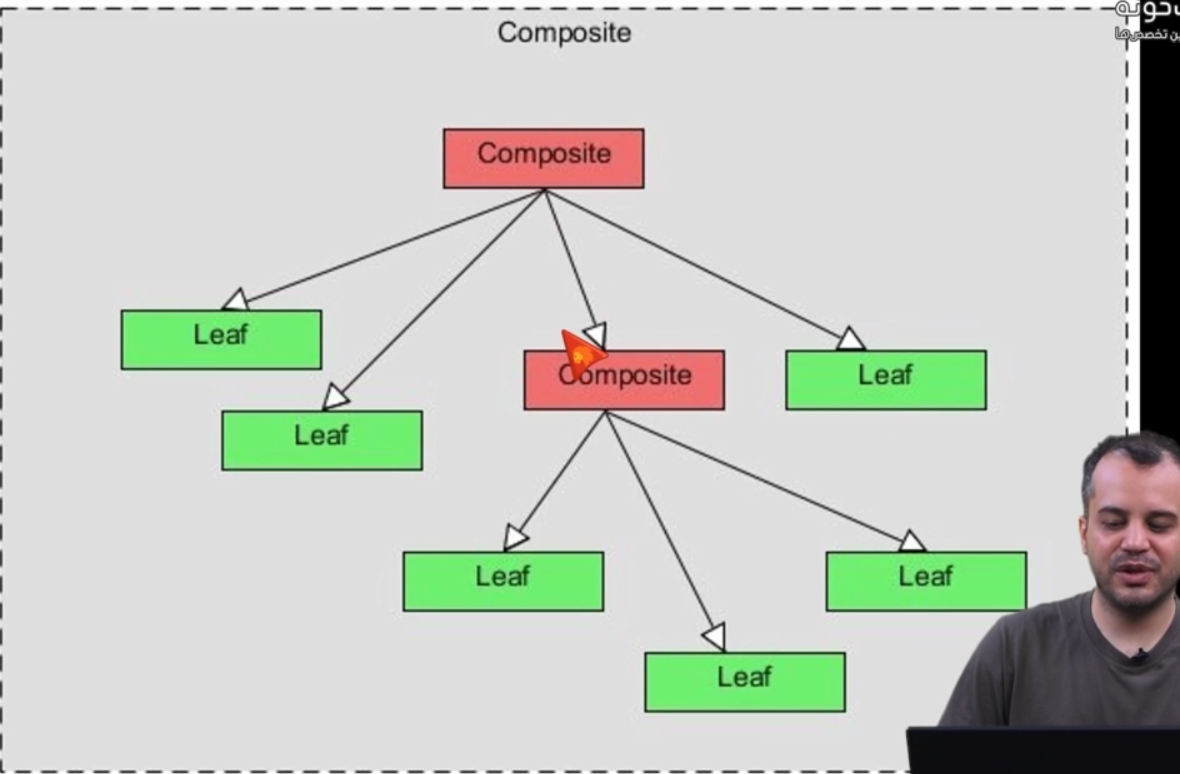
از نوع structural است و هدف آن این است که یک نمودار درختی را در نظر بگیرید این الگوی طراحی می تواند مشکلاتی که در این ساختار وجود دارد را حل کند.

Problem

به طور مثال یک فروشگاه اینترنتی داریم که موجودیتی به نام Order در آن وجود دارد و این موجودیت داخل خود می تواند OrderItem نیز دارشته باشد.

حل مسئله به کمک Compostie برای برای جدا کردن این 2 از یک دیگر کاربرد دارد.





اجزا:

Component

مشخص می کند که تعریف ما چیست؟ یک کلاس abstract یا interface است در واقع یک تعریف کلی داریم از مسئله خود از composite خود. تعریف واحد برای composite یک Component است.

Leaf

مقدار نهایی است که دیگر فرزند ندارد

Composite

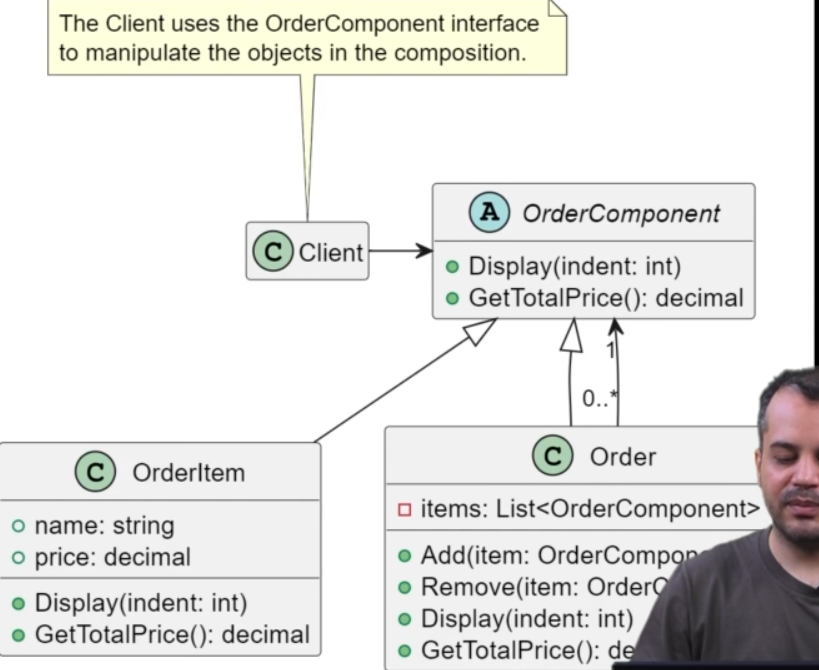
می تواند شامل یک composite دیگر یا leaf باشد.

Client

کسی است که در نهایت از Component ما استفاده می کند.

UseCase

زمانی که یک سلسله مراتبی از object ها را می خواهیم یکی در نظر بگیریم و می خواهیم که client ما فرق بین Leaf , Compositeرا متوجه نشود و در نظر نگیرد ignore کند



Decorator

از نوع Structural است. در framework ها و library ها بیشتر کاربرد دارد.

هدف : responsibility ها یا مسئولیت های مختلفی را به یک object به صورت Dynamicly اضافه می کند به یک object و functionality آن را extend می کند.

افزودن یک مسئولیت یا رفتار خاصی به یک object را انجام می دهد.

مثلا یک اتاق که به شکل های مختلف دکور می کنیم.

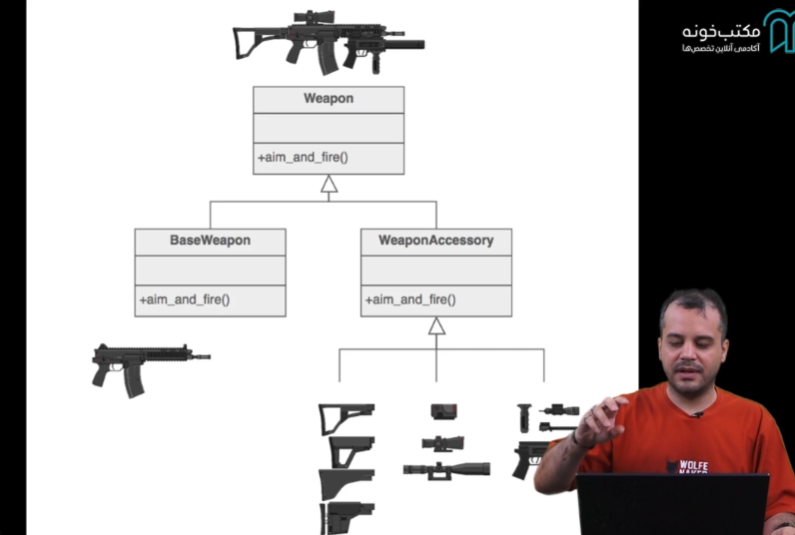
عملیات دکوریت کردن را انجام می دهیم.

Problem

نیاز داریم که یک سری feature را به یک object اضافه کنیم بدون تغییر در ساختار آن.

با استفاده از open/close می توانیم برای extension کار کنیم نه برای modify پس بنا براین می توانیم با استفاده از decorator رفتار جدید را به صورت dynamically اضافه کنیم بدون تغییر ساختار.

به آن wrapper نیز میگویند.



مفاهیم:

Component interface

یک interface است که مفهوم را بیان می کند.

Concrete component

پیاده سازی را در اینجا انجام می دهیم.

Decorator

مفهوم های جدید را اینجا اضافه می کنیم که یک interface است

Concrete decorator

پیاده سازی کلاس decorator را در این انجام می دهیم.

**نکته :**

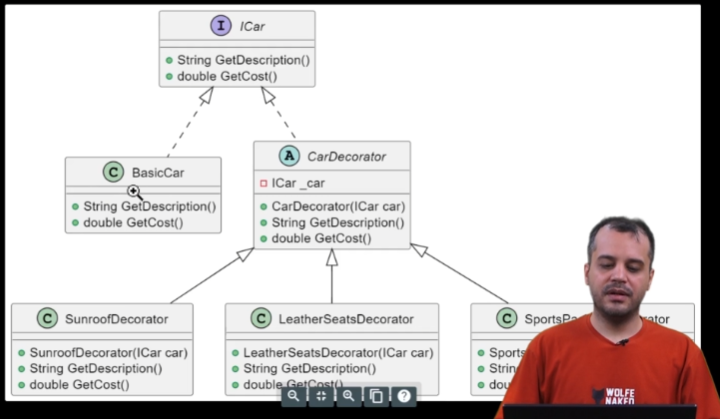
**مشابه chain of responsibility است که هر دور به صورت recursive پیاده سازی می شوند.**

**این امکان وجود دارد که چندین Decorator را به یکدیگر متصل کرد.**

**مثالا jango nodejs , asp هر کدام مجموعه ای از pipeline ها دارند که مراحل را به یک دیپر pass می دهند.**

**موارد استفاده :**

زمانی که نیاز است در runtime یک feature را به یک object اضافه کنیم بدون شکستن کد آن توانی گرفتن آن ویژگی وجود داشته باشد.



Facade

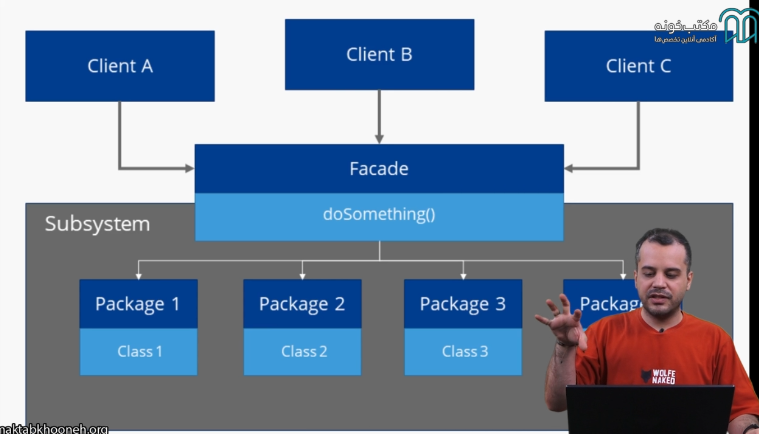
از نوع structural است. فساد.

هدف : به طور مثال سرویسی داریم که برای کنترل کل یک مجموعه کاربرد دارد مثلا ساختمان های مختلفی داریم که یک کارخانه را تشکیل می دهند و ارتباط بین ساختمان ها از طریق facade انجام می شود.

کنترل و مانیتور و صدا کردن را انجام می دهیم.

یک interface داریم در لایه بالا که زیرسیستم های من ار simplefie کرده است یا ساده سازی.

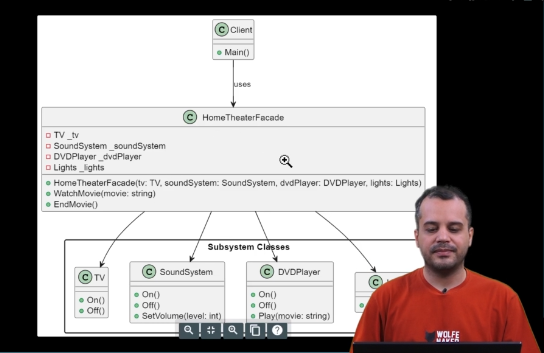
یک سرویس که روندی را برای انجام کاری تعریف می کند. مثلا سرویسی که فرآیند مربوط به خرید را از اول تا آخر کنترل می کند انبار سفارش پرداخت و تحویل.



مشکل : یک کتابخانه پیچیده داریم که می خواهیم کسی از آن استفاده کند در آن شی های ایجاد می شود وابتسگی های هست که برای کنترل این فرآیند از facade استفاده می کنیم.

کارکرد : simplify کردن Interface ساده کردن یک سیستم پیچیده زیر سیستم ها را یکپارچه می کند و مدیریت می کند زمانی که وابستگی ها زیاد است کنترل می کند.

می توان به صورت لایه بندی چندین facade نیز داشته باشیم برای کار های مختلف



Proxy

از نوع structural است. به معنی نیابت نیز هست.

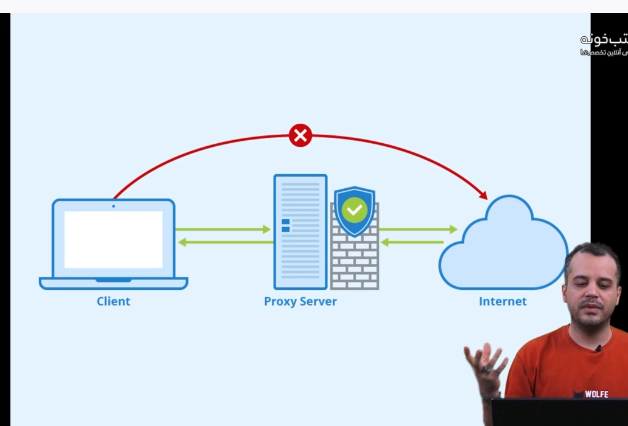
هدف : فرض کنید که یک سرویسی وجود دارد Proxy به ما کمک می کند که جایگزینی برای رسیدن به سرویس باشد.

فرض کنید دربی داریم که با استفاده از سنسور کارت خوان برای باز کردن آن اقدام می کنیم کاری که سنسور کارت خوان می کند به نیابت از در است برای رسیدن به آن.

کنترل دسترسی به در proxy کارت است.

یک proxy برای کنترل دسترسی به object اصلی کاربرد دارد و کنترل می کند.

دسترسی به internet از طریق proxy server ممکن است.



یک نوع بروکراسی است برای رسیدن به هدف است یعنی لازم به طی کردن مراحلی برای رسیدن به هدف.

UseCase

Lazy initialization (virtual proxy)

فرض کنید یک object بزرگ دارید که نیاز نیست هر موقع ایجاد شود با استفاده از این روش می توان تاخیر در ایجاد اضافه کرد.

AcessControl

محدود کردن دسترسی به یک object بنا به شرایط ما

Local execution of remote service(proxy remote)

مدیریت ارتباط با یک سرویس از طریق proxy مدیریت شود

Logging request(logging proxy)

پراکسی لاگ زدن سیتم درخواست های وارده را لاگ بزند و ثبت کند

Cashing request result(caching proxy)

پراکسی که دیتای که لود می شود را cash می کند و در درخواست های وارد شده بعدی دیتای کش را به ما می دهد.

Smart Reference

برای object های که حجیم هستند مدیریت resource ها و optimize کردن آن.

Proxy vs Façade

این دو بسیار شبیه به یک دیگر هستند و هر دو مدیریت یک compelex entity را مدیریت می کنند اما برعکس facade که خودش یک interface است proxy دارای یک interface که بسیار به object اصلی شبیه است.

Decorator , proxy نیز دارای یک ساختار بسیار شبیه به هم است اما هر کدام دارای هدف متفاوتی است.

Proxy کار مدیریت life cycle سرویس و object خود را انجام می دهد اما Decorator توسط کلاینت کنترل می شود.

When use

زمانی که یک جایگزین برای دسترسی به یک object داشته باشیم

زمانی که یک extra step می خواهیم برای یک دسترسی داشته باشیم – قدم اضافه برای رسیدن به یک object برای دسترسی به آن.

یک روش برای کاهش پیچیدگی و محافظت از کامپوننت اصلی است یا object ما.

