**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**

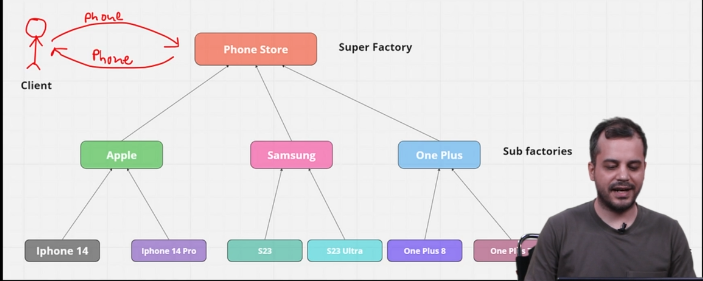
فرض کنید که یک Uiدارید که و در حال توسعه آن هستید و نیازمندی این است که برای ظاهر کار نیازمند تم های برای حالت های light dark , .. هستید و در ui خود نیز دارای ایتم های مانند button checkbox هستند و آن ها را در کنار هم ایجاد کرده اید اما نمی خواهید برای هر حالت یک instace از dark , light ایجاد کنید زیر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

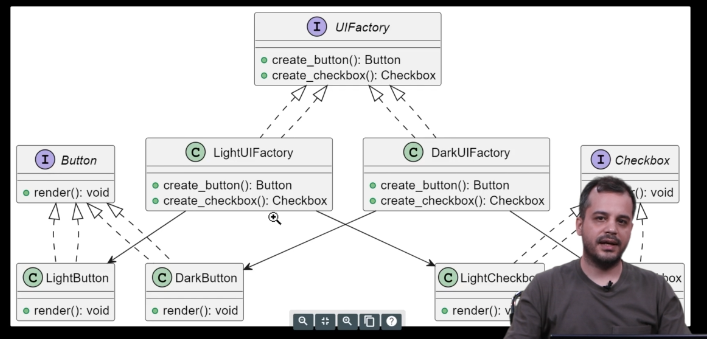


**When to use**

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**

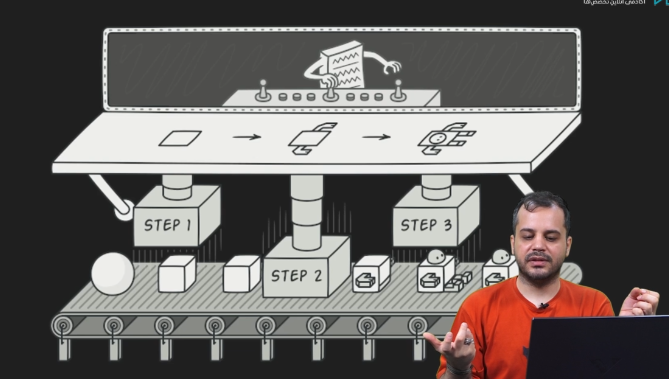
از نوع creational است و هدف مدیریت ایجاد یک complex object را مدیریت کنیم. هدف ایجاد object از یک کلاس است.

**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.