الف

- Cohesion : در معنی لغوی به معنای انسجام است. به این معنا است که از لحاظ مفهومی چقدر اعضای کد ما (method, class, package و ...) به هم مرتبط هستند و آیا هر کدام از این اعضا ایا در جای درستی قرار گرفته اند و هر کدام یک وظیفه ی مشخص دارند و فقط و فقط این وظیفه را انجام میدهند و در ان وظیفه بسیار خوب هستند و به این شکل با بقیه ی اعضا رابطه ی معناداری دارند؟ برای بهتر توضیح دادن این موضوع از یک مثال استفاده می کنم به این صورت که اگر قرار است ما یک وبسایت درست کنیم و یک package برای ان به اسم authentication در نظر گرفته ایم اگر در آن package مثلا class های مربوط به ui هم باشد که هیچ ربطی به آن ندارند در واقع انسجام کد پایین آمده است و این package یک وظیفه ی مشخص ندارد زیرا هم کار های مربوط به auth را انجام میدهد هم کارهای مربوط به ui را.

- Coupling : به این معناست که class ها یا package های ما تا چه حد از جزییات هم خبر دارند و به آن وابسته اند به این معنا که اگر مثلا در یکی از class ها ما از یک ArrayList برای ذخیره سازی یک نوع از داده استفاده می کنیم و یک class دیگر قرار است به عنوان client از آن استفاده کند آیا آن client مستقیم با خود ارایه در ارتباط است یا server class یک سری method به client میدهد که با صدا زدن آنها server خودش اطلاعات مربوط به ArrayList را عوض میکند؟

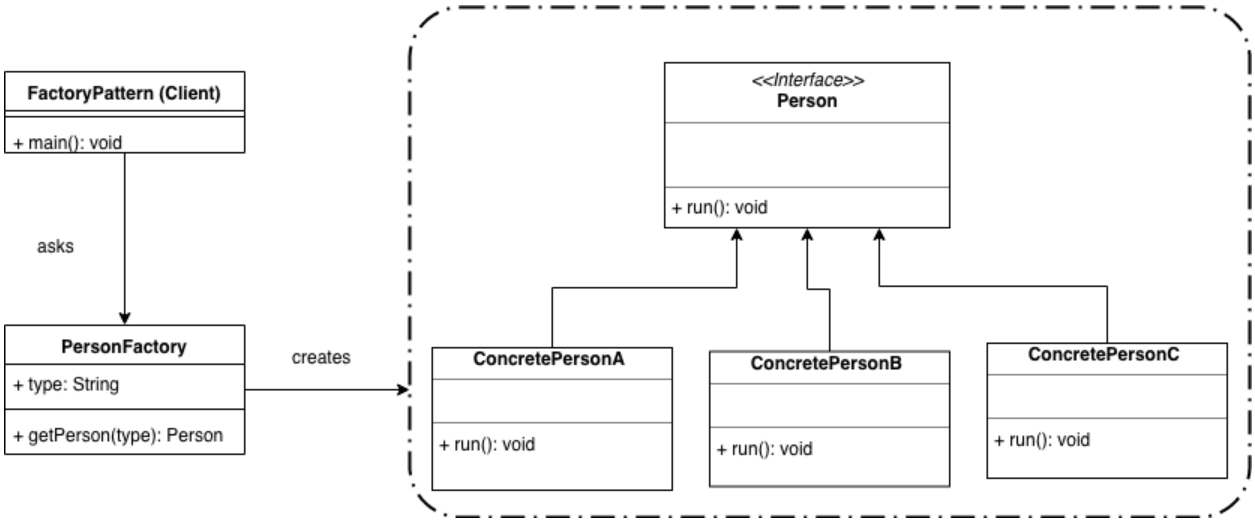
- Encapsulation : یکی از مفاهیم اساسی مهندسی نرم افزار است به این معنا که اگر یک class یک سری داده ای در خود به عنوان field ذخیره کرده است یا یک سری method به صورت دارد که وجود آنها به عنوان یک public behaviour معنایی ندارد ؛ داریم اگر آنها را از دید client code پنهان کنیم در واقع encapsulation را درست اجرا کرده ایم. پس در واقع encapsulation همان کپسوله کردن اجزای مختلف به نوعی معنادار است که دیگر client code به اجزای داخلی server code دسترسی مستقیم نداشته باشد و اگر می خواهد آن داده ها را عوض کند مجبور شود از طریق public behaviour هایی که server code اجزای آن را داده است ان تغییرها را اعمال کند.

- Enumerated type : در هنگامی که ما کد نویسی انجام میدهیم بسیاری از اوقات مجبوریم از primitive type ها (معمولا int ها هستند) برای نشان دادن یک مفهوم خاص استفاده کنیم (مثلا انواع admin یا paid user بودن یا guest بودن را با اعداد های ۱ و ۲ و ۳ نشان بدهیم) پس در درون کد ما عدد های 1 و 2 و 3 بسیار تکرار خواهند شد که به خودی خود معنای خاصی ندارند . حال اگر ما بیایم این عدد ها به یک اسم مشخص که معنادار تر از یک عدد خالی است map کنیم در واقع از enumerated type ها استفاده کرده ایم. حال در java برای enumerated type ها یک نوع syntax جدا وجود دارد که همه ی ان mapping هایی را که مربوط به یک موضوع واحد هستند را در یک جا نگه دارد که اسم ان syntax را enum گذاشته اند.

ب

به صورت کلی چون design pattern ها مربوط به یک سناریوی خاص و پرتکرار هستند پس بهتر است در توضیح هر design pattern ابتدا ان سناریو را بیان کنیم بعد خود design pattern که جوابی برای ان سناریو است را مطرح کنیم

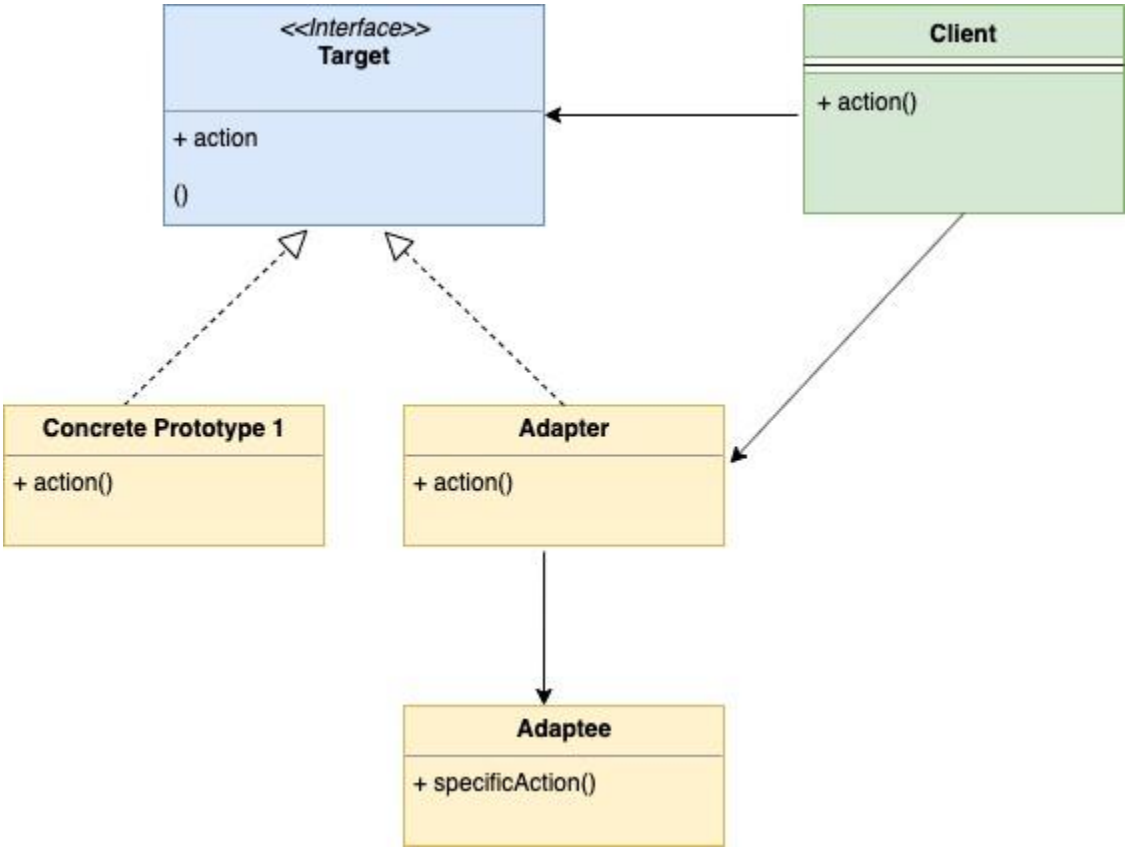
- Factory method : این سناریو را باید در نظر بگیریم که یک superclass داریم که چندین subclass دارد و قرار است که ما از یکی از این subclass ها یک نمونه بسازیم اما مشکل اینجاست که در زمان development نمی دانیم و در زمان runtime (مثلا بر اساس ورودی کاربر یا یک عدد random مشخص میشود یا شرایط خاصی که امکان دارد پیش بیاید و . . .) حال در این شرایط نیاز به یک logic داریم مربوط به چک کردن ان شرایط خاص و درست کردن نمونه ی درست از class درست است. خب یک ایده این است که client code خودش این کار را انجام دهد اما ما نمی توانیم هیچ اجباری به این بکنیم که client code ان بررسی های لازم را اول انجام دهد بعد نمونه را بسازد. پس راه حل بهتر این است که یک class درست کنیم که در هنگام اجرا شرایطی که باید را به صورت ورودی بگیرد و بعد ان نمونه ی مناسب را بسازد و return کند که اسم ان factory class است و به صورت کلی به این design pattern نیز factory method می گویند.



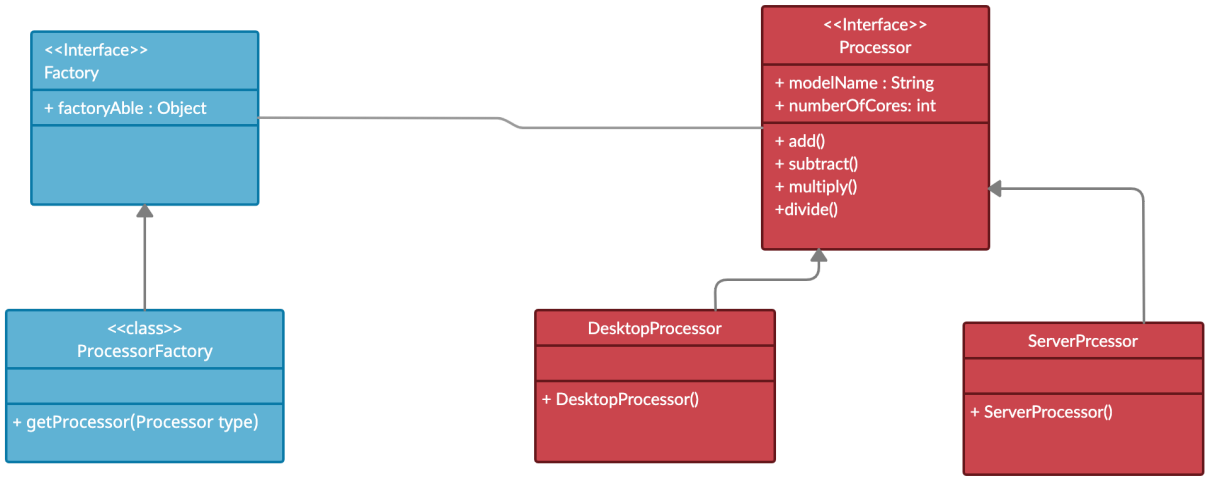
در این عکس همانگونه که به راحتی قابل تشخیص است ما یک class اصلی داریم که معمولا abstract هم هست به اسم PersonFactory که قرار است به عنوان یک wrapper به دور logic ساختن یک person قرار بگیرد و هنگامی که میخوایم یک person ایجاد کنیم یا type را به ان pass می دهیم یا ان متغیر هایی که مشخص کننده ی شرایط هستند و با استفاده از انها می توانیم تصمیم بگیریم (در اینجا خود پاس دادن type نشان داده شده است ولی به راحتی میتوان ان را overload کرد)

- Adapter : این شرایط را در نظر بگیریم که در نرم افزار مان یک package داریم (خواه خودمان آن را نوشته باشیم یا به صورت third party package باشد) که مثلا ورودی ای که میگیرد به صورت json است و حال در طرف دیگری از پروژه قسمت دیگری به هر دلیلی ورودی که میگیرد به صورت yml است و حال ما نیاز به استفاده از package در این قسمت دوم کد داریم پس نیاز داریم که دیتایی که به صورت yml داریم را به json در بیاوریم و نیازمند logic یی هستیم که این کار را برای ما انجام دهد. باز هم مانند مورد قبلی می توانیم

logic مربوط به jsonify کردن را به client code بسپاریم ولی دو مشکل اساسی داریم اولی اینکه در این حالت نمی توانیم client را مجبور کنیم که حتما تبدیل را انجام دهد و شاید ورودی اشتباه به آن package ما بدهد و مشکل بعدی این است که اگر قرض کنیم در جای دیگری از نرم افزار مثلا با xml هم که یک نوع سیستم برای انتقال داده ی بسیار قدیمی است نیز کار می کنیم حال دوباره client باید کار تبدیل را خودش انجام بدهد و این کار میتواند کیفیت کد را پایین می برد.پس از adapter استفاده می کنیم که به خوبی از public behaviour هایی که ان package دارد با خبر است و از طرفی می تواند ورودی را به صورت مثلا yml و یا xml بگیرد و آنها را تبدیل به json کند بعد تابع درست را در package با داده ی درست صدا بسازند و بعد نتیجه ی آن را return کند. به این روش adapter design pattern می گویند.



همونجور که در این عکس به راحتی قابل تشخیص است ابتدا client به adapter با data ی نامناسب درخواست میدهد (مرسوم است که adapter دقیقا همان public behaviour های target را داشته باشد) بعد factory آن داده ی نامناسب را به داده ی مناسب تبدیل می کند بعد آن را به target پاس میدهد بعد target کار خودش را انجام میدهد بعد نتیجه ی ان را به factory میدهد و بعد client هم return می کند.



فقط این نکته در اینجا حائز اهمیت است که من شخصا ترجیح میدهم که به جای type خود پارامتر های تعیین کننده را به getProcessor پاس بدم اما چون هیچ شرایطی برای اینکه در چه حالتی کدام را می سازیم بیان نشده بود دیگر صرفا خود type را پاس می دهم.