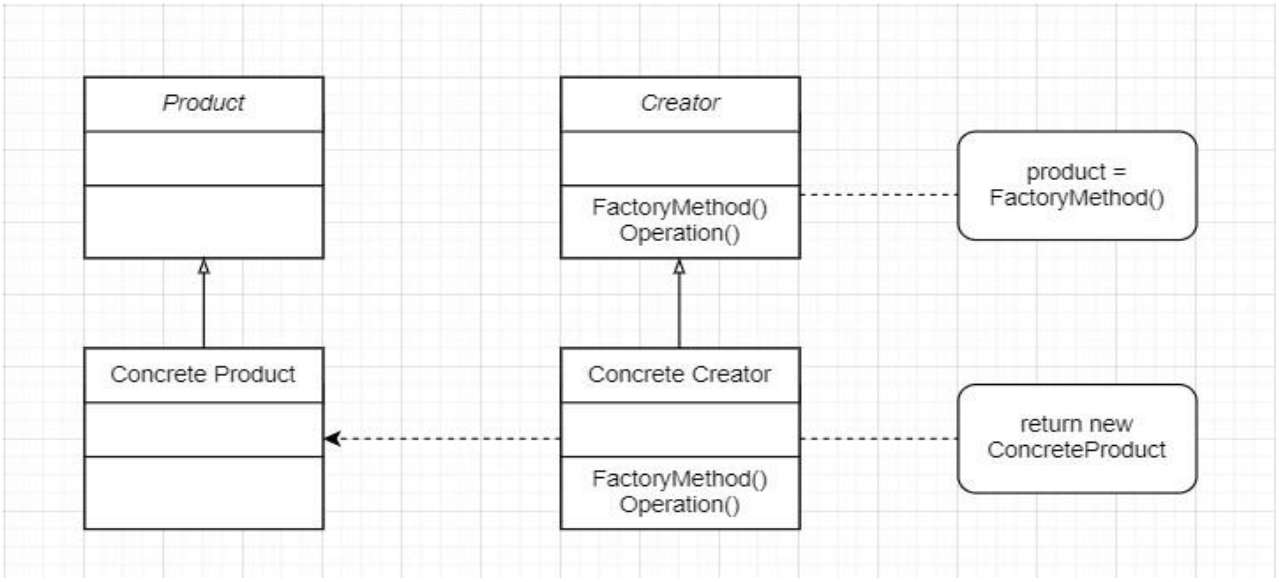
سوال اول:

یک) دیزاین‌پترن‌ها در واقع راه‌حل‌ها و طراحی‌هایی از قبل پیش‌بینی شده برای مشکلات احتمالی در طراحی و حرفه‌ی مهندسی نرم‌افزار هستند. باید توجه داشت که دیزاین‌پترن‌ها مانند تکه کدی نیستند که قابل کپی و استفاده در کد خودمان باشند بلکه یک blueprint یا الگو برای استفاده در طراحی‌ها هستند تا کار طراحی نرم‌افزار را برایمان راحت‌تر کنند. دیزاین‌پترن‌ها به سه دسته‌ی اصلی تقسیم می‌شوند که هر یک نیز دارای روش‌ها و زیرشاخه‌هایی معروف و کاربردی هستند. سه دسته‌ی اصلی آن‌ها عبارتند از: creational pattern, structural pattern, behavioral pattern.

دو) دیزاین‌پترن‌ها راه‌حل‌ها یا به بیان دیگر blueprintهایی هستند که در گذشته بارها توسط افراد و گاه شرکت‌های مختلف به کار گرفته شده‌اند و در اکثر موارد کاربردی بودن و مفید بودن آن‌ها ثابت شده است. همچنین داشتن یک پیش‌زمینه و الگوی کلی برای انجام طراحی باعث می‌شود که سرعت کد زدن ما بیش‌تر شود و در واقع زمان کمتری برای فکر کردن برای حل مشکل و ارائه‌ی یک طراحی مناسب برای برنامه‌ی خود بگذاریم. همچنین می‌توان به این مورد نیز اشاره کرد که آن‌ها را می‌توان یک زبان مشترک برای برنامه‌نویس‌ها دانست که باعث می‌شود فهم کدها برای دیگران آسان‌تر شود.

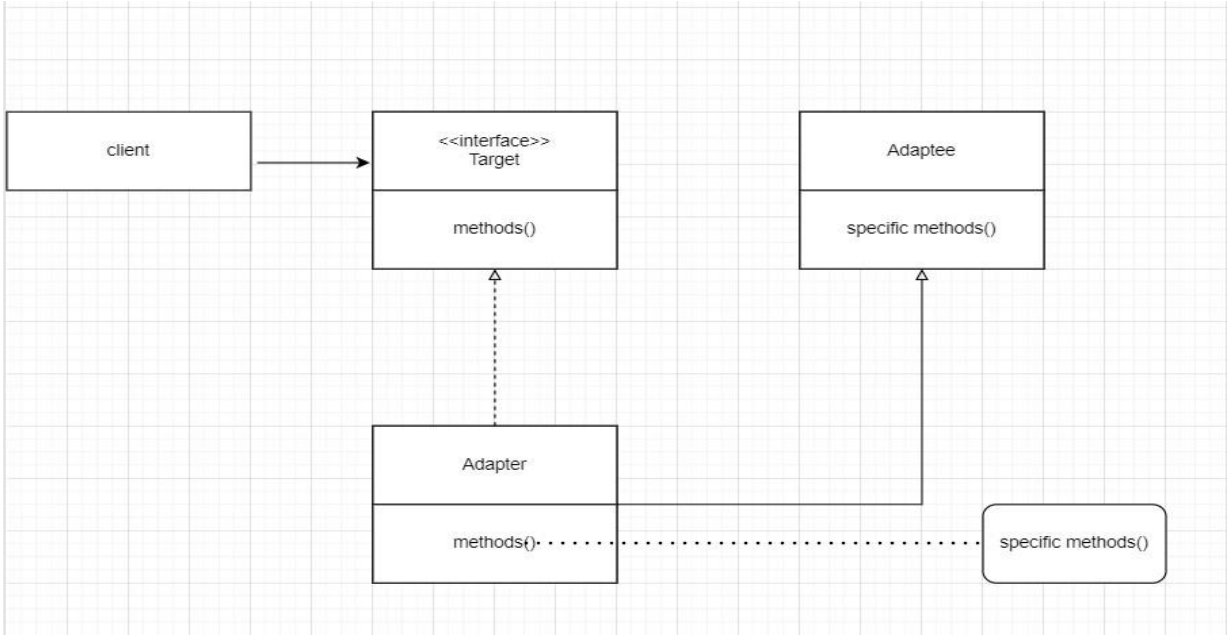
سه) از دسته‌ی اصلی factory method می‌توان به استفاده از creational pattern که در کلاس‌هایی مانند collections استفاده شده است یا استفاده از Builder در کلاسی مانند StringBuilder اشاره کرد. از دسته‌ی اصلی structural pattern می‌توان به استفاده از adapter در متود asList در کلاس Arrays یا استفاده از Proxy در کلاسی با همین نام اشاره کرد. از دسته‌ی اصلی behavioral pattern ها نیز می‌توان به استفاده از observer در کلاسی مانند PropertyChangeSupport و یا استفاده از template در کلاس‌هایی مانند InputStream و OutputStream اشاره کرد.

چهار) Factory method یک creational dessign pattern است که هدف اصلی از استفاده از یک کلاس برای ساخت آبجکت‌هایی ازکلاس‌های مختلف است که همه‌ی آن‌ها یک interface مشترک را implement کرده اند. هدف از این کار این است که در صورتی که خواستیم در آینده به کد خود قابلیت‌ها و کلاس‌های بیش‌تری اضافه کنیم، به جای اینکه لازم باشد تغییرات زیادی را در بسیاری از بخش‌های کد انجام بدهیم و در هر جایی که یک شی از یکی از آن کلاس‌ها ساخته‌ایم تغییراتی رو اعمال کنیم، تغییر را فقط در کلاسی که مسئول ساختن اشیای مختلف است انجام بدهیم و در آینده با استفاده از رفرنس‌هایی از نوع interface که آن‌ها از آن ارث می‌برند به متودهای آن کلاس‌ها که در واقع متودهای آن اینترفیس را implement کرده‌اند به آن متودها دسترسی داشته باشیم. این کار باعث رعایت encapsulation نیز می‌شود و کار ساخت اشیا را فقط درر داخل همان کلاس نگه می‌دارد و همچنین باعث می‌شود که ساخت اشیا در زمان runtime و به صورت داینامیک به شکل بسیار بهتری انجام شود.



Adapter method: هدف و استفاده از Adapter همانطور که می‌توان از اسم آن نیز حدس زد، تبدیل و استفاده از دو interface غیرقابل‌تبدیل به یکدیگر است. در این روش کاری که به صورت معمول انجام می‌دهیم، این است: اینترفیس هدف (target) داریم که می‌خواهیم از متودهای آن استفاده کنیم ولی از طرفی کلاسی داریم که متودهایی دارد که به برنامه و هدف آن اینترفیس نزدیک هستند ولی نمی‌توان با استفاده از implement کردن آن اینترفیس آن کلاس را تشکیل داد زیرا آن متودها پیاده‌سازی و حتی نام کاملا نامربوطی به متودهای آن اینترفیس دارند. ولی از طرفی می‌خواهیم در هنگام استفاده از آن شی یک متود و یک اسم را برای آن متودها صدا بزنیم و آبجکت‌های کلاس‌های مختلف با همان method calling وظایف منحصر به خود را انجام دهند. در اینجا راه ما استفاده از یک کلاس adapter است که اینترفیس هدف ما را پیاده‌سازی می‌کند، یک

فیلدِ آبجکت از کلاسی که قابل تبدیل نیست نیز دارد و در هنگام ساخت آبجکت از کلاس adapter این شی را می‌گیرد و به آن فیلد اساین می‌کند. در پیاده‌سازی متودهای اینترفیس اما ما به جای پیاده‌سازی آن‌ها، متودهای مربوطه و مرتبطی را که در کلاس آن شی وجود دارد با استفاده از آن شی کال می‌کنیم. حال اگر در هنگام ساخت شی یک رفرنس از interface هدف داشته باشیم و به آن یک شی از کلاسمان اساین کنیم، با استفاده از متودهای inteفaceمان می‌توانیم به صورت غیرمستقیم به متودهای آن کلاس دسترسی داشته باشیم و آن‌ها را صدا بزنیم.



پنج)