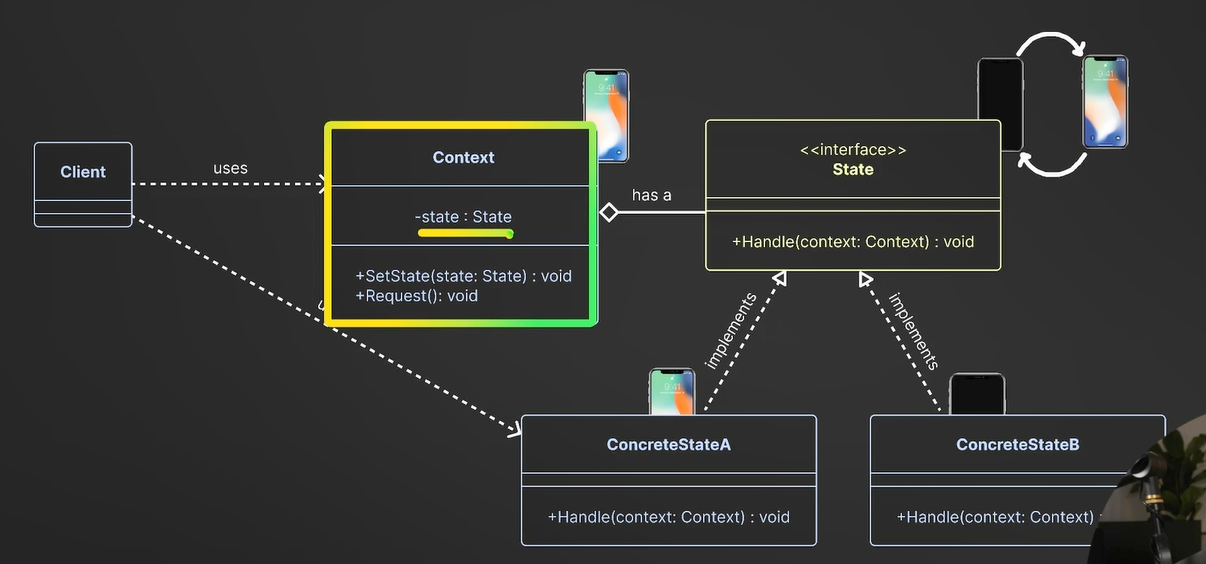
1. Singleton؟
   1. یه creational pattern هست که برای ساخت instance از کلاس هایی استفاده میشه که میخوایم در طول اجرای برنامه فقط یه instance از اون کلاس رو داشته باشیم. برای نمونه، هر برنامه ای نیاز به یه logger داره که برای قسمت های مختلف برنامه log بندازه. زمانیکه میخوایم logger رو new کنیم، چیزایی مثل زمان شروع اجرای برنامه (که خب معمولا بعد از اجرای برنامه با یه Datetime.Now مقداردهی میشه) رو داریم براش تعیین می کنیم؛ اگه این کلاس قرار باشه با هر بار new کردن اش زمان شروع اجرای برنامه رو برای هر instance اش تغییر بده، دیگه اون زمان valid نیست. پس لازمه که از این کلاس، همیشه یه instance در کل برنامه در زمان اجراش داشته باشیم.
   2. یه سری مشخصه ها این پترن داره که باید در زمان پیاده سازیش رعایتشون کنیم:
      1. کلاس مورد نظر باید private constructor داشته باشه که کسی نتونه خارج از کلاس، ازش new کنه.
      2. کلاس مورد نظر باید یه فیلد private static از نوع خود کلاس داشته باشه که instance ساخته شده از کلاس رو داخل اون قرار می دیم. این مورد و مورد قبلی در واقع دارن دسترسی از بیرون برای new کردن کلاس رو از بین می برن و در آن واحد کنترل instantiation کلاس رو به خود کلاس میسپرن.
      3. instance مورد نظر از طریق یه متد قابل دسترسی هست که اول چک می کنه که اون فیلد از نوع کلاس خالی هست یا نه. اگه خالی باشه، new رو انجام میده و تو فیلد میریزه. بعدش فیلد رو return میکنه.
   3. این پترن thread-safe نیست به دلایل زیر:
      1. Race condition: ممکنه چندتا thread به صورت همزمان درخواست ایجاد شئ بدن که هنوز برای اولین بار ایجاد نشده.
      2. Partial initialization: ممکنه در حین پروسه ایجاد instance برای یه thread، یه thread دیگه ای بیاد و درخواست این شئ رو بکنه که در این صورت هر دو thread دارن عمل new رو انجام میدن.  
         برای رفع این مشکل، از یه lock و lock object قبل از چک کردن null بودن instance، استفاده میشه.
      3. حالا یه حرکتی این وسط هست به نام double check lock که بحث پرفورمنسی ایجاد می کنه؛ در حالت بالا، زمانی که برای اولین بار میخوایم new اش کنیم، یهو تعداد زیادی thread میان و یکیشون وارد lock میشه. باقی thread ها پشت سرش می مونن تا این lock رو release کنه. در نتیجه به race condition   
         می خوریم. پس قبل از lock هم یه بار دیگه شرط null بودن رو میذاریم. این باعث میشه که نهایتا به تعداد core های cpu مون thread بیاد پشت lock منتظر بشه. بعدش یکی وارد lock میشه و new می کنه. از اون thread های پشت lock که موندن، حالا هر کدوم که وارد بشه، دیگه نمیتونه new کنه (به خاطر شرط null بودن دوم). در این وضعیت، هم thread safety رو رعایت کردیم هم به کمترین تعداد thread ممکن اجازه دادیم که برای اولین بار بیان پشت lock. (این وضعیت فقط مال دفعه اولیه که instantiation میخواد انجام بشه چون دیگه بعدش شئ ساخته شده و نهایتا null بودن اولیه چک میشه که دیگه false ئه و در نتیجه اصلا به lock نمیرسه)
   4. نکته مصاحبه ای: اگه بخوایم پراپرتی Instance رو به شکل زیر داشته باشیم، چطوری باید باقی کد کلاس رو بنویسیم؟  
      public static Singleton Instance { get; } = new();  
      جواب: ازstatic constructor به علاوه nested class داخل کلاس Singleton مون استفاده می کنیم. توی IL، یه flag هست به اسم beforeFieldInit که داخل سوالات csharp دربارش توضیح دادیم. اگه این flag غیرفعال بشه (با static constructor)، اونوقت به محض اینکه اولین پراپرتی static این کلاس بهش reference داده بشه، کل پراپرتی های static کلاس instantiate میشن.   
      خب تا اینجا خیلی کار خاصی نکردیم. اصل ماجرا در ادامه اس. هنوزم الان اگه کسی بخواد میتونه با صدا زدن Singleton.Instance، هر بار شئ رو از اول بسازه. میایم یه کلاس nested داخل همین کلاسمون میسازیم و این خط کد رو می بریم داخل اون و بعدش به کلاس nested مون هم یه static ctor میدیم. بعدش یه پراپرتی Instance دیگه از نوع Singleton توی خود کلاس Singleton درست می کنیم که Instance داخل Nested رو بر می گردونه.  
      حالا چه اتفاقی می افته؟ وقتی برای اولین بار یه پراپرتی static از کلاس Singleton صدا زده میشه، همه static ها از جمله Instance، instantiate میشن. اما نکته اینجاس که خود Instance داره static ctor کلاس Nested رو صدا میزنه که اونم یه بار موقع instantiation داره اجرا میشه. در واقع همه بحث های thread safe بودن و اولین بار ایجاد کردن شئ رو با این روش هندل می کنیم.
2. Factory method؟
3. Abstract factory؟
4. Builder؟
5. Strategy؟
6. Observer؟
7. State؟
   1. یه behavioural pattern هست و تعریف کلی این پترن میگه که یه آبجکت زمانی که وضعیت درونی اش تغییر میکنه میتونه رفتارشو تغییر بده. مثلا دکمه پاور یه گوشی رو در نظر بگیریم؛ اگه گوشی در وضعیت روشن باشه این دکمه کار خاموش/لاک کردن رو انجام میده و اگه خاموش باشه، کار روشن کردن رو.  
      دقت کنیم که در هر حال، اینترفیس ما که همون دکمه هه باشه یکسانه؛ چیزی که داره تغییر میکنه رفتار گوشی هست (که در این حالت داره وضعیت رو هم تغییر میده)
   2. یه نمونه از class diagram این پترن رو داشته باشیم:  
      
   3. این پترن شباهت های زیادی به Strategy داره. در واقع انگار عین همدیگه ان اما تفاوتی هم دارن. در state، متد داخل اینترفیس یه شئ از نوع context رو داره میگیره. در حالی که داخل strategy اینطور نیست. از طرفی، در state، هر ConcreteState میتونه از state های دیگه باخبر باشه در حالی که در strategy اینطوری نیست و هر ConcreteStrategy داره به صورت مستقل عمل می کنه.  
      در واقع، strategy داره میگه یه عمل با چه روش ها و مدل هایی میتونه انجام بشه اما state میگه چطور میتونم از یه وضعیت به یه وضعیت دیگه برم و در هر وضعیت چه کار هایی میتونم انجام بدم.  
      توی strategy، بیشتر داره external state رو هندل می کنه (مثلا کاربر میخواد یه کاری رو به شکل های مختلف انجام بده) اما state سمت internal state هست (یه درخواست در چه مرحله ای از flow مورد نظر هست).
   4. مزایا:
      1. حفظ single responsibility: هر state، خودش داره کار های مربوط به خودشو هندل می کنه. کس دیگه ای کارای اونو براش انجام نمیده.
      2. جلوگیری از کثیف شدن کد و استفاده بیش از اندازه از condition ها
      3. جلوگیری از branching (یک حالت داخل خودش یه حالت دیگه صدا کنه و اونم دوباره به همین شکل و تمام این کار ها داخل یه scope اتفاق بیفته در صورتی که راحت میشه هر کدوم رو جداگونه توی کلاس خودش نوشت)
      4. به مرور زمان، سمت کلاینت تغییری ایجاد نمیشه. تنها جایی که تغییر میتونه بکنه ConcreteState ها هستن که یا خودشون تغییر می کنن یا state های جدید اضافه میشه اما سمت کلاینت دست نخورده میمونه و حتی اینترفیسمون هم احتمال پایینی داره که تغییر کنه.
   5. معایب:
      1. تعریف تعداد زیادی کلاس (برای هر State یه کلاس تعریف میشه)
      2. در سناریو های ساده، استفاده از این پترن پیچیدگی کد رو افزایش میده.
      3. ممکنه جابجایی بین state ها پیچیده باشه و سختی هایی ایجاد کنه (که البته من مخالفم چون وقتی تمام کار های یه state خاص داره توی یه جا انجام میشه، کار ما راحتتره و اگه اشتباهی هم پیش میاد یا چیزی جا می مونه دیگه این تقصیر برنامه نویسه نه پترن. در واقع یا بیزنس مشکل داره یا رعایت نشده یا نکته فنی ای رو برنامه نویس از قلم انداخته)
      4. با بزرگتر شدن پروژه هم ممکنه پیچیدگی بالا بره؛ مثلا یه سری validation به state ها اضافه بشه که قبلا نیاز نبوده.
      5. تست کردنش میتونه سخت باشه چون سناریو های زیادی رو ممکنه نیاز باشه بررسی کنیم.