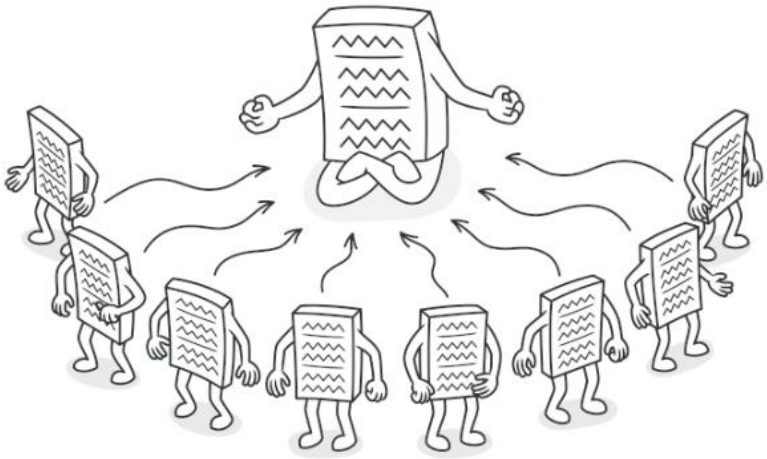
Singleton در Design Pattern

مفهوم

Singleton یک design pattern خلاقانه ای است که با کمک آن می توانید مطمئن شوید که یک کلاس تنها یک نمونه دارد و در عین حال یک نقطه ی دسترسی سراسری را برای این نمونه در نظر بگیرید.



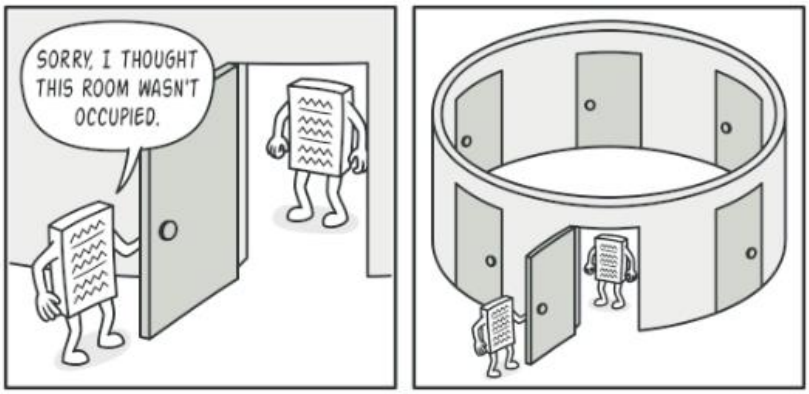
مشکل

این pattern به طور همزمان دو مشکل زیر را حل می کند و اصل مسئولیت واحد را نقض می کند.

1. تضمین این مطلب که یک کلاس تنها یک نمونه داشته باشد. چرا اصلا کنترل تعداد نمونه های یک کلاس اهمیت دارد؟ رایج ترین دلیل برای این امر کنترل دسترسی به برخی از منابع مشترک مثل دیتابیس ها یا فایل ها است.

   این روند به این صورت کار می کند که فرض کنید شیئی را ایجاد کرده اید، اما بعد از مدتی می خواهید شیء جدیدی را ایجاد کنید. به جای اینکه شیء جدیدی را دریافت کنید، شما آن شیئی را دریافت می کنید که قبلا ایجاد شده است.

   توجه داشته باشید که این رفتار را نمی توان به کمک یک سازنده ی معمولی پیاده سازی کرد. چرا که فراخوان های این گونه سازنده ها همیشه باید شیء جدیدی را برگشت دهند.



کلاینت ها ممکن است اصلا متوجه این مطلب نشوند که همیشه با شیء یکسانی در حال کار هستند.

2. ارائه ی یک نقطه ی دسترسی سراسری به این نمونه. آن دسته از متغیرهای سراسری که شما (و البته من) برای ذخیره کردن برخی از اشیاء اساسی استفاده کردیم را یادتان هست؟ با وجود اینکه این اشیاء تا حد زیادی کارآمد هستند، اما تا در عین حال بسیار خطرناک هستند، چرا که هر کد می تواند محتوای این متغیرها را بازنویسی کند و برنامه خراب شود.

Singleton pattern درست مانند متغیرهای سراسری این امکان را به شما می دهد تا از هرجایی در برنامه به برخی از اشیاء دسترسی پیدا کنید. با این حال این pattern از بازنویسی این نمونه توسط کد دیگر جلوگیری می کند.

وجه دیگر این مشکل این است که شما نمی خواهید کدی که مشکل شماره 1 را حل می کند، در سراسر برنامه ی شما پخش شود، بلکه ترجیح می دهید این کد داخل یک کلاس قرار گیرد؛ مخصوصا اگر بخش های باقی مانده ی کد شما از قبل به آن وابسته باشند.

امروزه Singleton pattern به حدی محبوبیت پیدا کرده است که مردم حتی چیزی که تنها یکی از مشکلات بالا را حل می کند را Singleton می نامند.

راه حل

تمامی پیاده سازی های Singleton مشترکا دارای مراحل زیر هستند:

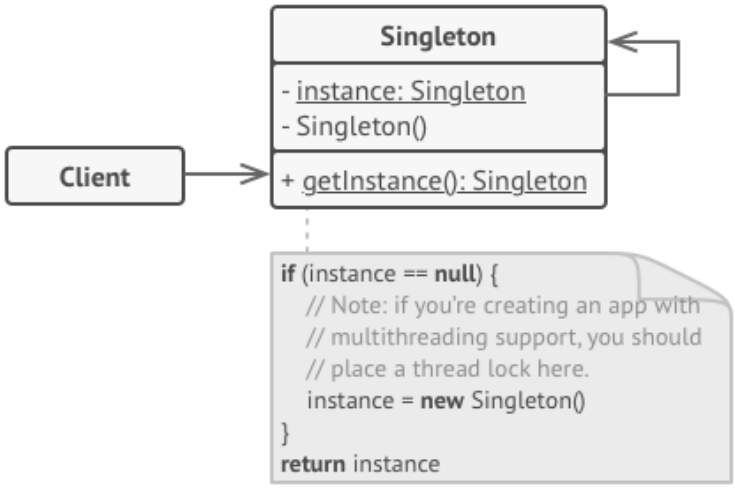
- جهت جلوگیری از استفاده از عملگر new در کنار کلاس Singleton توسط اشیاء دیگر سازنده ی پیش فرض را خصوصی کنید.
- یک متد creation استاتیک را ایجاد کنید تا به عنوان سازنده عمل کند. این متد به صورت مخفیانه، این سازنده ی خصوصی را فراخوانی می کند تا این سازنده شیئی را ایجاد کند و آن را داخل یک فیلد استاتیک ذخیره کند. تمامی فراخوان های پس از این متد این شیء کش شده را برگشت می دهند.

اگر کد شما به کلاس Singleton دسترسی داشته باشد، در این صورت می تواند متد استاتیک Singleton را فراخوانی کند. بنابراین هر زمان که این متد فراخوانی شود، همیشه همین شیء برگشت داده می شود.

مقایسه با دنیای واقعی

دولت ها می توانند نمونه ی خوبی از این pattern باشند. کشورها می توانند تنها یک دولت رسمی داشته باشند، صرف نظر از هویت های فردی افرادی که دولت را شکل می دهند، عبارت «دولت X» یک نقطه ی دسترسی سراسری بوده که بیانگر گروهی از افرادی است که منصبی را برعهده دارند.

ساختار



1. کلاس Singleton متد استاتیک getInstance که نمونه ی یکسانی از کلاس خود را برگشت می دهد را اعلان می کند.

2. سازنده ی Singleton باید از کد کلاینت مخفی باشد. فراخوانی متد getInstance باید تنها روش دریافت شیء Singleton باشد.

شبه کد

در این مثال کلاس ارتباطی دیتابیس به عنوان یک Singleton عمل می کند.

این کلاس دارای سازنده ی عمومی نیست. بنابراین تنها راه دریافت اشیاء آن، فراخوانی متد getInstance است. این متد اولین شیء ایجاد شده را در جایی ذخیره می کند (کش می کند) و تمامی فراخوان های پس از آن را برگشت می دهد.

```
// The Database class defines the `getInstance` method that lets
// clients access the same instance of a database connection
// throughout the program.
class Database is
  private field instance: Database

  // The singleton's constructor should always be private to
  // prevent direct construction calls with the `new`
  // operator.
  private constructor Database() is
    // Some initialization code, such as the actual
    // connection to a database server.
    // ...

  // The static method that controls access to the singleton
  // instance.
  static method getInstance() is
    if (this.instance == null) then
      acquireThreadLock() and then
        // Ensure that the instance hasn't yet been
        // initialized by another thread while this one
        // has been waiting for the lock's release.
        if (this.instance == null) then
          this.instance = new Database()
    return this.instance

  // Finally, any singleton should define some business logic
  // which can be executed on its instance.
  public method query(sql) is
    // For instance, all database queries of an app go
    // through this method. Therefore, you can place
    // throttling or caching logic here.
    // ...
```

```
class Application is
  method main() is
    Database foo = Database.getInstance()
    foo.query("SELECT ...")
    //...
    Database bar = Database.getInstance()
    bar.query("SELECT ...")
    // The variable `bar` will contain the same object as
    // the variable `foo`.
```

کاربرد

زمانی که یکی از کلاس های برنامه ی شما باید تنها یک نمونه را در اختیار تمامی کلاینت ها قرار دهد، از Singleton pattern استفاده کنید. مثلا زمانی که یک شیء دیتابیس واحد توسط بخش های مختلفی از برنامه به اشتراک گذاشته شده باشد.

Singleton pattern تمامی ابزارهای ایجاد اشیاء یک کلاس را غیرفعال می کند و تنها متد creation ویژه ای را فعال نگه می دارد. این متد یا شیء جدیدی را ایجاد می کند، یا اگر شیئی قبلا ایجاد شده باشد، آن را برگشت می دهد.

زمانی که می خواهید به صورت صریح تر متغیرهای سراسری را کنترل کنید، از Singleton pattern استفاده کنید.

Singleton pattern برخلاف متغیرهای سراسری تضمین می کند که تنها یک نمونه از کلاس موجود باشد. هیچ چیزی غیر از خود کلاس Singleton نمی تواند جای نمونه ی کش شده را بگیرد.

توجه داشته باشید که همیشه می توانید این محدودیت را اصلاح کنید و به مقدار لازم نمونه های Singleton را ایجاد کنید. برای این کار تنها بخشی از کد که باید تغییر کند، بدنه ی متد getInstance() است.

چگونگی اجرا

1. جهت ذخیره سازی نمونه ی Singleton یک فیلد استاتیک خصوصی را به کلاس اضافه کنید.

2. جهت دریافت این نمونه یک متد creation استاتیک و عمومی را ایجاد کنید.
3. داخل این متد استاتیک "lazy initialization" را پیاده سازی کنید. سپس در اولین فراخوان این متد شیء جدیدی ایجاد می شود که آن را باید داخل فیلد استاتیک قرار دهید. این متد همیشه باید این نمونه را در تمامی فراخوان های بعدی بازگشت دهد.
4. سازنده ی این کلاس را خصوصی کنید. متد استاتیک این کلاس همچنان قادر خواهد بود تا تنها این سازنده را برگشت دهد و نه بیشتر.
5. به کد کلاینت بروید و جای تمامی فراخوان های مستقیم به سازنده ی Singleton را با فراخوان های متد استاتیک creation آن عوض کنید.

مزایا و معایب

- ✓ می توانید مطمئن شوید که یک کلاس تنها یک نمونه دارد.
- ✓ جهت دسترسی به این نمونه دارای یک نقطه ی سراسری می شوید.
- ✓ شیء singleton تنها زمانی مقداردهی اولیه می شود که برای بار اول به آن نیاز باشد.
- × اصل مسئولیت واحد را نقض می کند. این pattern به صورت همزمان دو مشکل را حل می کند.
- × این pattern می تواند طراحی نامناسب را بپوشاند. مثلا زمانی که اجزای برنامه در رابطه با همدیگر بیش از حد اطلاعات دارند.
- × این pattern در یک محیط چند ریسه ای به رسیدگی ویژه ای نیاز دارد، به گونه ای که ریسه های متعدد برای چندین بار نمی توانند یک شیء singleton را ایجاد کنند.
- × تست واحد کد کلاینت singleton ممکن است دشوار باشد، چرا که بسیاری از فریمورک های تست در زمان ایجاد اشیاء ساختگی به وراثت متکی هستند. با توجه به این که سازنده ی کلاس singleton خصوصی است و در بسیاری از زبان ها اوررایید کردن متدهای استاتیک غیرممکن است، برای ساختن singleton و تقلید از آن نیاز به یک راه خلاقانه دارید و یا اینکه تست ها را کلا ننویسید و یا از Singleton pattern استفاده نکنید.

رابطه با Pattern های دیگر

- کلاس Facade را اغلب می توان به یک Singleton تبدیل کرد. چرا که یک شیء Facade واحد در اغلب موارد کفایت می کند.
- اگر به گونه ای بتوانید تمامی حالت های مشترک اشیاء را تقلیل دهید و به یک شیء Flyweight تبدیل کنید، در این صورت Flyweight شبیه به Singleton خواهد شد. با این حال بین این pattern ها دو اختلاف اساسی وجود دارد:
  1. تنها یک نمونه Singleton باید وجود داشته باشد. این در حالی است که یک کلاس Flyweight می تواند چندین نمونه با حالت های ذاتی مختلف داشته باشد.
  2. شیء Singleton قابل تغییر و اشیاء Flyweight غیرقابل تغییر هستند.
- Abstract Factory ها، Builder ها و Prototype ها را تماما می توان به صورت Singleton پیاده سازی کرد.