مقدمه ای بر Design Pattern

با عرض سلام خدمت دوستان عزیز و گرامی. در این مقاله ، تصمیم دارم که شما را با مفاهیم Design Pattern آشنا نمایم. دوستان عزیز، در صورتیکه که با زبان های .NET. به عنوان زبانهای کاملا شیء گرا آشنایی داشته باشید، مطمئنا با کلاس های متعددی آشنا شده و یا شخصا نسبت به ایجاد آنها اقدام کرده اید. حال اگر به زبان ساده Member های یک کلاس را Fields, Property, Method, Event بنامیم، شاید شما تاکنون با صدها و یا هزاران کلاسی که با یک و یا ترکیبی از این Member ها ایجاد می شوند، آشنا شده و یا ایجاد کرده باشید. حال با توجه به مطالب عنوان شده، سوال این است که یادگیری Pattern ها چه جایگاهی در طراحی و یا پیاده سازی کلاس ها دارد؟

اجازه دهید که با مثال ساده، این مساله را بررسی نماییم:

دوستان، شاید در زمان طراحی و یا پیاده سازی برنامه های خود، با کلاسهایی روبرو شده اید که اصطلاحا کلاسهای کلاسیک نبوده اند! و در زمان طراحی و یا پیاده سازی آنها، شما را کمی به فکر فرو برده و یا بعضا دچار سردرگمی کرده باشند!. در اینگونه موارد Pattern ها به یاری شما آمده و مشکلاتتان را تا حجم قابل قبولی حل می نمایند. اجازه دهید نمونه ای را با هم بررسی نماییم. تصور کنید که تصمیم دارید در پروژه خود، کلاسی طراحی نمایید که بتوان از آن تنها یک شیء ایجاد نمود! ممکن است کمی تعجب کنید!! ولی باور کنید که در یک پروژه واقعی به کرات اتفاق می افتد که شما تمایل به ایجاد چنین کلاس هایی داشته باشید. اجازه دهید نمونه هایی از این دست را برای شما ذکر نمایم:

۱- کلاس مدیرعامل. (در یک سازمان بیش از یک مدیرعامل وجود ندارد)

۲- کلاس رئیس هیات مدیره. (در یک سازمان بیش از یک رئیس هیات مدیره وجود ندارد)

۳- کلاس Supervisor.

٤- کلاس Connection. با توجه به اینکه در اکثر موارد پیشنهاد می شود که در هر پروژه، خصوصا در پروژه های Windows Based بیش از یک Connection به بانک اطلاعاتی نداشته باشیم، ایجاد کلاسی که تنها امکان یک شیء Connection داشته باشد، بسیار اهمیت خواهد داشت.

با توجه به نمونه های فوق، ایجاد کلاسی که بتوان از آن فقط یک شیء ایجاد نمود، اهمیت زیادی پیدا می کند. ولی چگونه چنین کلاسی ایجاد نماییم:

۱- در صورتیکه خودمان می خواهیم از چنین کلاسی استفاده نماییم، تا آخر پروژه حواسمان را جمع کنیم که خدای نکرده، بیش از یک شیء از آن ایجاد نکنیم!
۲- در صورتیکه در پروژه تیمی کار می کنیم، همکارانمان را قسم دهیم! که از این کلاس بیش از یک شیء ایجاد نکنند!!
۳- ساعتها فکر کنیم تا یک راه حل نسبتا قابل قبولی پیدا کنیم!!!

صبر کنید! این روشها را فقط برای مزاح عرض کردم!! مطمئن باشید که بکارگیری هر یک از این سه روش فوق، پروژه شما را با بحران جدی مواجه می کند. پس راه حل چیست؟

در واقع طراحی چنین کلاس هایی، به مرور زمان، گریبانگیر هر برنامه نویسی شده است و به مرور زمان، برنامه نویسان خبره با توجه به تجربیات شخصی خود و دیگران، تصمیم به ایجاد الگوهایی کرده اند که راه حل اینگونه مشکلات خواهد بود. در حال حاضر شاید در حدود یکصد الگو یا Pattern در این رابطه وجود داشته باشد که تنها بیست و پنج مورد از آنها استاندارد شده و مورد استفاده طراحان و برنامه نویسان قرار می گیرد. شاید یکی از ساده ترین و پرکاربرد ترین این الگوها، الگوی Singleton بوده که با استفاده از آن، نمونه های فوق به راحتی قابل طراحی و پیاده سازی می باشند. نکته قابل توجه این است که در برنامه Rational XDE 2003، تمامی این الگوهای معروف به صورت Built in وجود داشته و طراحان می توانند تنها با یک Drop & Drag ساده، از این الگوها در طراحی خود استفاده نمایند. لذا یادگیری این الگوها را برنامه نویسان عزیز و خصوصا طراحان گرامی پیشنهاد می کنم.

مثال :

یک برنامه شبیه ساز عملیات نجات می خواهیم بنویسیم که دارای اشیای مختلفی می باشد: ماشین ها، انسانها، ماشین های آتش نشانی، ماشین پلیس، ساختمان های آتش گرفته، جاده های مسدود، افراد مصدوم، ...

نکته مهم آن است که می خواهیم یک کلاس با نام Simulator داشته باشیم که عملا شبیه سازی را انجام دهد و اطلاعات مربوط به کل عناصر فوق را ذخیره نماید. تمام اشیای موجود در دنیای برنامه باید یک ارجاع به یک Simulator داشته باشند و اطلاعاتی را که لازم دارند از آن درخواست نمایند و ضمنا اطلاعات وضعیت خود را نیز به آن اطلاع دهند. بعنوان مثال، ماشین های آتش نشانی، لیست ساختمان های آتش گرفته را از Simulator می پرسند، ماشین پلیس لیست جاده های مسدود را از Simulator می گیرد.

طبیعی است که اگر بیش از یک شیء از Simulator در برنامه وجود داشته باشند، مشکلاتی پیش می آید و عملکرد برنامه با مشکل مواجه می شود.

راه حل ابتدایی: از برنامه نویسان خواسته شود که حواسشان را جمع کنند که بیش از یک شیء از کلاس Simulator ایجاد نشود.

معایب این روش مشخص است و در نتیجه من اصلا به ذکر آنها نمی پردازم.

راه حل مبتنی بر الگوی طراحی: استفاده از الگوی طراحی Singleton

طبق این الگوی طراحی، خود کلاس Singleton باید دارای این قابلیت باشد که اصلا امکان ایجاد بیش از یک شیء از آن، وجود نداشته باشد. بدین ترتیب، کنترل، محلی می شود و همین که در داخل کلاس، کنترلهای لازم انجام شده باشد، کافی است و پیچیدگی به بیرون منتقل نمی شود.

کلاس Simulator بر اساس این الگوی طراحی بدین ترتیب می باشد: (البته لازم به ذکر است که این الگوی طراحی را، در سطح کد، به شکل دیگری هم (با تغییرات جزئی) می توان پیاده سازی کرد)

```
public class Simulator {
     private static Simulator simulator = new
                Simulator();

    private Simulator() { ... }
    public static Simulator getSimulator() {
                return simulator;
   }
}
```

توضیح: سازنده کلاس دارای سطح دسترسی خصوصی است. در نتیجه امکان ایجاد شیء از این کلاس، در خارج کلاس وجود ندارد. به بیان دیگر، فقط خود کلاس Simulator می تواند اشیایی از این کلاس ایجاد کند.

مشتریان این کلاس، هیچ راهی برای بدست آوردن یک ارجاع به یک شیء از نوع Simulator ندارند مگر آنکه متد ایستای getSimulator را فراخوانی کنند. این متد هم یک ارجاع به یک شیء ایستا برمی گرداند. فیلد simulator بصورت ایستا تعریف شده است. در نتیجه زمانی که کلاس Simulator در حافظه بارگذاری می شود این فیلد برای یک بار ایجاد و مقداردهی می شود. هر بار که متد getSimulator فراخوانی شود، یک ارجاع به همین شیء برگردانده می شود. در نتیجه شرط مورد نظر، تضمین می شود.

خوب این الگوی طراحی، وابسته به مثال شبیه سازی نمی باشد و در مثالهای دیگری نیز قابل اعمال است.

بعنوان مثال می خواهیم در یک برنامه یک عنصر مدیر صف داشته باشیم که تقاضاهایی را دریافت کرده و در صف قرار داده و این تقاضا ها را به نوبت سرویس دهد. ممکن است بخواهیم مطمئن شویم که در هر لحظه بیش از یک مدیر صف نمی تواند وجود داشته باشد. در این صورت می توانیم از الگوی طراحی Singleton استفاده کنیم. بنابراین همانطور که گفتیم یک الگوی طراحی را می توانیم در مسائل متعددی که مشابه هم هستند استفاده کنیم.

شاد و پیروز باشید