Design Patterns in .NET

Abbas Rezaei Aliabad

The SOLID Design Principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

الگوهای طراحی (Design Patterns) مجموعه‌ای از راهکارهای استاندارد و اثبات‌شده برای حل مشکلات متداول در طراحی نرم‌افزار هستند. این الگوها نه تنها باعث افزایش کیفیت کد می‌شوند، بلکه ارتباط بین اجزای مختلف سیستم را نیز تسهیل می‌کنند. یادگیری و به‌کارگیری الگوهای طراحی به توسعه‌دهندگان کمک می‌کند تا کدی تولید کنند که:

- به‌راحتی قابل تغییر و توسعه باشد.
- از تکرار کد جلوگیری کند.
- انعطاف‌پذیری سیستم را افزایش دهد.

برای درک بهتر الگوهای طراحی، ابتدا باید اصول پایه‌ای طراحی نرم‌افزار را بشناسیم. یکی از مهم‌ترین این اصول، مجموعه‌ای از مفاهیم به نام **SOLID** است.

اصول SOLID

اصول SOLID پایه و اساس طراحی خوب نرم‌افزار هستند که شامل پنج اصل می باشد:

۱- اصل مسئولیت واحد (Single Responsibility Principle - SRP): هر کلاس باید تنها یک وظیفه داشته باشد و تنها یک دلیل برای تغییر آن وجود داشته باشد.

۲- اصل باز-بسته (Open-Closed Principle - OCP): کلاس‌ها و توابع باید برای توسعه باز اما برای تغییر بسته باشند.

۳- اصل جایگزینی لیسکوف (Liskov Substitution Principle - LSP): کلاس‌های فرزند باید بتوانند جایگزین کلاس والد شوند بدون اینکه رفتار سیستم مختل شود.

۴- اصل تفکیک رابط‌ها (Interface Segregation Principle - ISP): رابط‌ها باید به گونه‌ای طراحی شوند که پیاده‌سازی آن‌ها فقط شامل موارد مورد نیاز باشد.

۵- اصل وارونگی وابستگی (Dependency Inversion Principle - DIP): کلاس‌های سطح بالا نباید به جزئیات وابسته باشند، بلکه باید به انتزاعات وابسته شوند.

۱- Single Responsibility Principle

اصل مسئولیت واحد (Single Responsibility Principle - SRP) یکی از اصول پایه‌ای در طراحی شی‌گرایی است که می‌گوید:

"هر کلاس فقط باید یک وظیفه داشته باشد و تنها یک دلیل برای تغییر آن وجود داشته باشد".

در این متن، اصل مسئولیت واحد با یک مثال ساده و کاربردی توضیح داده شده است:

مثال: دفترچه یادداشت(Journal)

فرض کنید می‌خواهید یک کلاس برای نگهداری یادداشت‌های شخصی طراحی کنید. این کلاس به صورت زیر شروع می‌شود:

```
public class Journal
{
  private readonly List<string> entries = new List<string>();
  // just a counter for total # of entries
  private static int count = 0;
}
```

این کلاس وظیفه‌اش مدیریت یادداشت‌ها است:

- **افزودن یادداشت‌ها** (AddEntry)
- **حذف یادداشت‌ها** (RemoveEntry)

تا اینجا کلاس وظیفه خودش را به درستی انجام می‌دهد.

حالا می‌توانیم از آن استفاده کنیم:

```
public void AddEntry(string text)
{
  entries.Add($"{++count}: {text}");
}

public void RemoveEntry(int index)
{
  entries.RemoveAt(index);
}
```

حالا می‌توانیم از آن استفاده کنیم:

```
var j = new Journal();
j.AddEntry("I cried today.");
j.AddEntry("I ate a bug.");
```

چه مشکل بوجود خواهد آمد؟

فرض کنید تصمیم بگیرید که دفترچه را در یک فایل ذخیره کنید. اگر این متد را به کلاس Journal اضافه کنید، کدی شبیه به این خواهید داشت:

```
public void Save(string filename, bool overwrite = false)
{
  File.WriteAllText(filename, ToString());
}
```

این کار **اصل مسئولیت واحد را نقض می‌کند**، زیرا حالا کلاس Journal دو وظیفه متفاوت دارد:

۱- نگهداری و مدیریت یادداشت‌ها (مسئولیت اصلی).

۲- ذخیره‌سازی داده‌ها در فایل (مسئولیتی اضافی).

مشکل اینجاست که اگر بعداً روش ذخیره‌سازی را تغییر دهید (مثلاً بخواهید یادداشت‌ها را در فضای ابری ذخیره کنید)، باید تغییرات زیادی در کلاس Journal انجام دهید. این موضوع کدی شکننده و پیچیده ایجاد می‌کند.

راه‌حل: جداسازی مسئولیت‌ها

برای رعایت اصل SRP ، باید **ذخیره‌سازی** را از کلاس Journal جدا کنید. می‌توانید یک کلاس جدید به نام PersistenceManager بسازید:

```
public class PersistenceManager
{
  public void SaveToFile(Journal journal, string filename,
                         bool overwrite = false)
  {
    if (overwrite || !File.Exists(filename))
      File.WriteAllText(filename, journal.ToString());
  }
}
```

حالا:

- کلاس Journal فقط مسئول مدیریت یادداشت‌ها است.
- کلاس PersistenceManager مسئول ذخیره‌سازی داده‌ها است.

این طراحی انعطاف‌پذیر است و تغییرات مربوط به هر وظیفه در کلاس مرتبط خودش انجام می‌شود.

مفاهیم تکمیلی

۱. شیء خدا (God Object): یک مثال افراطی از نقض اصلSRP ، شیء خدا است. این شیء، کلاسی است که تلاش می‌کند همه کارها را انجام دهد و در نتیجه بسیار بزرگ، پیچیده و سخت‌فهم می‌شود. کار با چنین کلاسی سخت است و تغییرات در آن باعث خطاهای غیرمنتظره می‌شود.

۲. بوی بد کد (Code Smell) : اگر کدی دارید که نیازمند تغییرات کوچک و مکرر در بخش‌های مختلف است، این معمولاً یک نشانه (Code Smell) است که طراحی شما مشکل دارد.

نتیجه‌گیری

- اصل SRP تاکید می‌کند که هر کلاس باید فقط یک وظیفه مشخص داشته باشد.
- جداسازی وظایف (مانند مدیریت داده‌ها و ذخیره‌سازی) باعث می‌شود کد تمیزتر، قابل فهم‌تر و نگهداری‌پذیرتر شود.
- با استفاده از این اصل، می‌توانید از پیچیدگی و خطاهای احتمالی جلوگیری کنید و سیستم‌های نرم‌افزاری پایدارتر و انعطاف‌پذیرتری بسازید.