KISS

"Keep It Simple, Stupid"

اصل KISS در برنامه نویسی بسیار اهمیت دارد.سعی کنید این اصل را سعی کنید به خاطر بسپارید و برای حفظ آن تلاش کنید.هرچقدر کد ساده تر باشد نگهداری آن در آینده ساده تر است.افراد دیگری که بخواهند کد شما را مورد ارزیابی قرار دهند در آینده با استقبال بیشتری این کار را انجام میدهند.اصل KISS توسط Kelly Johnson پایه گذاری شده است و سیستم های خوب به جای پیچیدگی، به سمت ساده سازی پیش میروند.از این رو سادگی، کلید طلایی طراحی است و باید از پیچیدگی های غیرضروری دوری کرد.

Martin Fowler در این باره می‌گوید:

هر احمقی می‌تواند کدی بنویسد که ماشین آن را درک کند؛ یک برنامه‌نویس خوب کدی می‌نویسد که انسان‌ها قادر به درک آن باشند.

کد ساده‌ی مد نظر در این اصل باید سرراست و آسان و بدون هیچ هوشمندی خاصی باشد؛ البته که نوشتن کدهای ساده و غیر پیچیده یکی از نشانه‌های مهم برنامه‌نویس با کیفیت است.

کارهای متفاوتی برای ساده نگه‌داشتن کدها قابل انجام است. یکی از این موارد دوری کردن از مفاهیم انتزاعی و abstraction است.

یکی از دلایل پیچیده شدن کدها، برنامه‌نویس‌هایی اند که قصد خودنمایی دارند. این اتفاق اغلب در برنامه‌نویس‌های جوان که قصد شگفت‌زده کردن دیگران را دارند، رخ می‌دهد.

اصل KISS قصد دارد به توسعه‌دهندگان بگوید که باید کدهایتان را به ساده‌ترین و احمقانه‌ترین شکل ممکن درآورید. اما باید این نکته را مد نظر داشته باشیم که نباید بیش‌از اندازه نیز مسائل را ساده کنیم و خوانایی و قابلیت نگه‌داری و گسترش کدهایمان، قربانی این سادگی شوند.

YAGNI

“You Aren’t Gonna Need It”

گاهی اوقات تیم های توسعه و برنامه نویسان در مسیر پروژه تمرکز خود را بر روی قابلیت های اضافه ی پروژه که "فقط الان به آن نیاز دارند" یا "در نهایت به آن نیاز پیدا میکنند" میگذارند.در یک کلام :اشتباه است!در اکثر مواقع شما به آن نیاز پیدا ندارید و نخواهید داشت. "شما به آن نیازی ندارید."

اصل YAGNI قبل از کدنویسی بی انتها و بر پایه ی مفهموم "آیا ساده ترین چیزی است که می تواند احتمالا کار کند" قرار دارد.حتی اگر YAGNI را جزوی از کدنویسی بی انتها بدانیم، بر روی تمام روش ها و فرآیند های توسعه قابل اجرا است.با پیاده سازی ایده‌ی "شما به آن نیازی ندارید" میتوان از هدر رفتن وقت جلوگیری کرد و تنها رو به جلو و در مسیر پروژه پیش رفت.

هر زمان اضطراب ناشناخته ای در کد حس کردید نشانه ی یک امکان اضافی بدون مصرف در این زمان است.احتمالا شما فکر میکنید یک زمانی این امکان اضافی را نیاز دارید.آرامش خود را حفظ کنید! و تنها به کارهای موردنیاز پروژه در این لحظه نگاه کنید. شما نمیتوانید زمان خود را صرف بررسی آن امکان اضافی کنید چون در نهایت مجبور به تغییر، حذف یا احتمالا پذیرفتن هستید ولی در نهایت جزو امکانات اصلی محصول شما نیست.

اهمیت این اصل در آن است که شما نباید بیش از حد مورد نیاز کد نویسی کنید. شما نباید مفاهیم اضافی را که ممکن است در آینده نیازمندشان باشید، به پروژه‌تان اضافه کنید. شما نباید سادگی پروژه‌تان را فدای فول آپشن بودن آن کنید؛ تنها باید مواردی را که به آن‌ها نیاز دارید پیاده‌سازی کنید، نه مواردی که ممکن است در آینده به آن نیاز پیدا کنید.

پیروی از این اصل منجر به صرفه‌جویی در زمان و کاهش کدهای بلا استفاده در پروژه‌هایتان می‌شود. علاوه بر این کیفیت کدهایتان نیز افزایش می‌یابد؛ چرا که دیگر به نوشتن کدهایی که حدس می‌زنید در آینده به کارتان خواهد آمد نیازی نخواهید داشت.

DRY

“Don’t Repeat Yourself”

تا الان چندین بار به کد های تکراری در پروژه برخورد کرده اید؟ اصل DRY توسط Andrew Hunt و David Thomasدر کتاب The Pragmatic Programmer پایه گذاری ده است.خلاصه ی این کتاب به این موضوع اشاره میکنید که "هر بخش از دانش شما در پروژه باید یک مرجع معتبر، یکپارچه و منحصربفرد داشته باشد". به عبارت دیگر شما باید سعی کنید رفتار سیستم را در یک بخش از کد مدیریت کنید.

از سوی دیگر زمانی که از اصل DRY پیروی نمیکنید، در حقیقت اصل WET که به معنای Write Everything Twice یا We Enjoy Typing دامن گیر شما شده است! (لذت بردن از وقت تلف کردن)

استفاده از اصل DRY در برنامه نویسی بسیار کارآمد است. مخصوصا در پروژه های بزرگ که کد دائما در حال نگهداری و توسعه است

دلایل زیادی برای دوری از نوشتن کد تکراری وجود دارد. مهم‌ترین دلیل این است که شما برای ایجاد تغییر در کد تکراری خود باید در چندین جای مختلف کدهایتان را تغییر دهید. نتیجتا خطوط کد شما بدون دلیل بیش‌تر خواهد شد و احتمال وقوع باگ نیز افزایش می‌یابد.

اما اگر شما یک بار یک عملکرد را نوشته باشید و در مکان‌های مختلفی از پروژه‌تان از آن استفاده کنید، با این مشکل مواجه نخواهید شد: چرا که تنها با ایجاد تغییر در یک قسمت از کد، تمامی قسمت‌هایی که از آن عملکرد مشترک استفاده می‌کنند، بروز خواهند شد.

پس می‌توان گفت که DRY یک اصل مهم در کد نویسی استاندارد است. برای دستیابی به این مهارت نیاز است که منطق و کد خود را به بخش‌های قابل استفاده‌ی مجدد (reusable) تقسیم کنید و در هر کجا که به آن منطق نیاز داشتید، تنها به فراخوانی آن بپردازید.

در نهایت اگر توانستید به شکلی قابل قبولی اصل DRY را در کد نویسی خود پیاده کنید، خواهید دید که برای ایجاد تغییر در بخشی از کدتان، نیاز به ایجاد تغییراتی در دیگر بخش‌های نامربوط به آن، نخواهید داشت.

اصطلاح SOLID اولین بار توسط مایکل فیرز معرفی شد، در حالی که خود اصول در ابتدا توسط رابرت جی. مارتین، همچنین به نام عمو باب، در مقاله خود در سال 2000 ارائه شد. عمو باب دانشمند کامپیوتر مشهور، نویسنده کتاب‌های پرفروشی مانند «Clean Code» و «Clean Architecture» و یکی از شرکت‌کنندگان فعال در Agile Alliance است.

اصول SOLID در برنامه نویسی با مفاهیم کدنویسی تمیز، معماری شی گرا و الگوهای طراحی همسو هستند، زیرا همگی هدف مشترک ایجاد نرم‌افزار با کیفیت بالا را دارند. در اصل SOLID از 5 اصل اساسی تشکیل شده است که به صورت موارد زیر هستند:

اصل مسئولیت واحد(Single Responsibility Principle)

اصل باز – بسته(Open-Closed Principle)

اصل جایگزینی لیسکوف(Liskov Substitution Principle)

اصل جداسازی رابط(Interface Segregation Principle)

اصل وارونگی وابستگی(Dependency Inversion Principle)

اصول سالید را در همه زبان‌های برنامه نویسی شی گرا مانند پایتون، جاوا اسکریپت، سی شارپ، net coreو غیره می‌توان به کار گرفت.

سازنده کلاس یا Constructor

یک متد از نوع Public می باشد که دقیقا هم نام با نام کلاس می باشد. این متد دارای خروجی نمی باشد ولی میتواند ورودی داشته باشد. ورودی این متد در واقع همان پارامترها یا ورودی کلاس می باشد.

```
public class Car
{
  //Constructor
  public Car(string color, int weigth, long price, string name)
  {

  }
{
```

کاربرد سازنده کلاس چیست ؟

کاربرد اصلی سازنده کلاس این می باشد که اگر شما کلاسی داشته باشید که برای ساختن یک نمونه از آن نیاز به چند پارامتر ورودی داشته باشد، این پارامترهای ورودی را می توانید در سازنده کلاس تعریف و حین ساختن نمونه از کلاس مقدار آن را مشخص کنید.

مخرب کلاس یاDestructor

وقتی شما یک نمونه از یک کلاس را ایجاد می کنید در همان لحظه اتوماتیک سازنده کلاس اجرا می شود. حالا وقتی آبجکت می خواهد از حافظه خارج شود مخرب کلاس به صورت اتوماتیک اجرا می شود و دستورات درون آن اجرا می شود. برای تعریف مخرب کلاس می توانید به صورت زیر عمل کنید.

```
public class Car
{
   //Constructor
   ~ Car()
   {
      //Some Code
   }
{
```

gc.collect

یک تابع در زبان برنامه نویسی پایتون است که برای اجباری اجرای مجدد ماشین گرفته بندی (garbage collection)در فضای حافظه استفاده میشود.

وظیفه gc.collect این است که اشیاءی که دیگر مورد استفاده نیستند را از حافظه حذف کند.

وقتی از این تابع استفاده میکنید، بدنه جایگزین مجموعهای از شیهای غیرقابل دسترس موجود در حافظه اجرا میشود.

این کار ممکن است منجر به آزادسازی حافظه غیرضروری و افزایش عملکرد برنامه شما شود.

استفاده ی درست از gc.collect برای بهینهسازی مصرف حافظه و جلوگیری از نشتی حافظه بسیار حیاتی است، اما باید با احتیاط و هماهنگی با نیازهای واقعی برنامه شما انجام شود