به نام خدا

میلاد کلاته

درس برنامه نویسی سمت سرور

استاد آقای میثاق یاریان

شماره دانشجویی:**01221033720030**

موضوع **gc.collect SOLID , DRY , KISS , YAGNI و , constructor , deconstructor**

KISS

“**K**eep **I**t **S**imple, **S**tupid”

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.

YAGNI

“**Y**ou **A**ren’t **G**onna **N**eed **I**t”

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.

DRY

“**D**on’t **R**epeat **Y**ourself”

تا الان چندین بار به کد های تکراری در پروژه برخورد کرده اید؟ اصل DRY توسط Andrew Hunt و David Thomas در 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

ممکن است علاقه ای به رعایت اصل DRY نداشته باشید ولی اصول قبلی (YAGNI, KISS) را حتما بخاطر بسپارید.

SOLID

SOLIDمخفف پنج اصل طراحی شی گرا است که توسط Martin .C Robert در سال 2000 معرفی شد. این اصول برای بهبود کیفیت کد و سهولت توسعه و نگهداری آن طراحی شده اند .

اصول SOLID عبارتند از:

1 :Single Responsibility Principle (SRP)هر کلاس باید یک مسئولیت واحد داشته باشد .

2 :(OCP) Open-Closed Principleکلاس ها باید باز برای گسترش باشند، اما بسته برای اصالح باشند .

3 :(LSP) Liskov Substitution Principleزیر کلاس ها باید بتوانند به جای کلاس های پایه خود استفاده شوند

4 :(ISP) Interface Segregation Principle .رابط ها باید به کوچکترین رابط های ممکن تقسیم شوند .

5 :(DIP) Dependency Inversion Principle وابستگی ها باید به سمت انتزاع ها هدایت شوند، نه جزئیات پیاده سازی

Single Responsibility Principle (SRP)

اصل SRP بیان می کند که هر کالس باید یک مسئولیت واحد داشته باشد. این بدان معناست که هر کلاس باید تنها یک کار را انجام دهد و به هیچ چیز غیر ضروری وابسته نباشد.

open-Closed Principle (OCP)

اصل OCP بیان می کند که کالس ها باید باز برای گسترش باشند، اما بسته برای اصلاح باشند. این بدان معناست که کلاس ها باید به گونه ای طراحی شوند که بتوان ویژگی های جدیدی را به آنها اضافه کرد، بدون اینکه نیاز به تغییر کد موجود باشد.

Liskov Substitution Principle (LSP)

اصل LSP بیان می کند که زیر کلاس ها باید بتوانند به جای کلاس های پایه خود استفاده شوند. این بدان معناست که زیر کلاس ها باید رفتار کلاس پایه را حفظ کنند.

Interface Segregation Principle (ISP)

اصل ISP بیان می کند که رابط ها باید به کوچکترین رابط های ممکن تقسیم شوند. این بدان معناست که هر رابط باید فقط یک مجموعه کوچک از مسئولیت ها را تعریف کند.

Dependency Inversion Principle (DIP)

اصل DIP بیان می کند که وابستگی ها باید به سمت انتزاع ها هدایت شوند، نه جزئیات پیاده سازی. این بدان معناست که کلاس ها باید به انتزاع ها وابسته باشند، نه جزئیات پیاده سازی.

Constructor

Constructor یا تابع سازنده یک متد خاص یا ویژه از کلاس است یا ساختاری در برنامه نویسی شئی گرا است و به منظور مقدار دهی اولیه یا initializes مورد استفاده قرار می گیرد Constructor.یک Instanceاز متد است که همانند instance از کلاس می ماند و امکان دسترسی به member یا اعضا آن Constructor یا کلاس امکان پذیر خواهد بود که می تواند به صورت پیشفرض تعریف شده یا توسط کاربر تکمیل شود Constructor یا تابع سازنده در دوره اجرا زنده ماندشان (Life Time) یکبار اجرا می شوند و از آن به بعد قابل دسترسی خواهد بود هر Constructor یا تابع سازنده باید با access modifiersیا تعیین کننده دسترسی تعریف شوند تا محدوده دسترسی به آنها معلوم شود Constructor تابعی است که در هنگام ایجاد کلاس به صورت اتوماتیک فراخوانی می شود و می توان توسط آن تنظیمات اولیه همانند ایجاد ارتباط با دیتابیس و یا کارهای مشابه را انجام داد.

از این تابع بیشتر برای مقداردهی کردن متغییر ها و فیلدهای یک کلاس استفاده می شود.

Destroctor

در مقابلConstructor ، مفهوم دیگری داریم تحت عنوان .Destroctor گفتیم کانستراکتور زمانی فراخوانی می‌شود که یک شیئ از روی کلاسی ساخته می‌شود؛ اما در مقابل، دیستراکتور زمانی فراخوانی می‌شود که یک آبجکت یا بهتر بگوییم یک شیئ از بین می‌رود و از جمله مواقعی که یک شیئ ساخته شده از روی کلاسی از بین می‌رود) یا اصطلاحاً Destroy می‌شود (می‌توان به زمانی اشاره کرد که اسکریپت ما به اِتمام رسیده که بالتبع بخشی از حافظهٔ اِشغال‌شده توسط آن اسکریپت نیز آزاد می‌شود) همچنین زمانی که از متدی تحت عنوان unset() استفاده می‌کنیم، آبجکت ساخته‌شده از بین خواهد رفت (.برای ساخت متد دیستراکتور نیز همواره باید آن را به صورت __destruct() بنویسیم و نکته‌ای که همواره در مورد دیستراکتورها باید مد نظر قرار دهیم این است که این گروه از متدها، بر خلاف کانستراکتورها و سایر متدها، به هیچ وجه نمی‌توانند پارامتر ورودی دریافت کنند.

gc.collect

هر زبان برنامه نویسی **Garbage Collection**را به‌طور متفاوتی پیاده سازی می‌کند، اما از لحاظ **خودکار** بودن این عمل، همگی شبیه به یکدیگر هستند. در بعضی موارد، **قابلیت های GC** را می‌توان از طریق **کتابخانه** یا **ماژول** به یک زبان اضافه کرد **GC** .یک جزء نسبتا رایج در اکثر **زبان های برنامه نویسی مدرن** است و این تنها رویکرد **مدیریت حافظه** است که بیشتر توسعه‌دهنده‌ها آن را می‌شناسند و به‌طور کلی **GC**یک قابلیت مهم برای توسعه‌دهندگان به‌شمار می‌رود **GC** .یک فرآیند مداوم است که به پردازنده (CPU) نیاز دارد و می‌تواند عملکرد کلی برنامه را تحت‌تأثیر قرار دهد یا حتی عملکرد آن را مختل کند. به همین دلیل، برخی از توسعه‌دهندگان هنوز در مورد **مزایای GC** بحث می‌کنند و معتقدند که آنها می‌توانند **حافظه** را بهتر از یک فرآیند خودکار **مدیریت** کنند.