اصول SOLID به زبان ساده

1 - Single Responsibility Principle (SRP)

یک کلاس فقط باید به یک دلیل تغییر کنه.

2 - Open/Closed Principle (OCP)

موجودیت‌های یک نرم‌افزار (کلاس‌ها، ماژول‌ها، توابع و ...) باید برای توسعه داده شدن، باز و برای تغییر دادن، بسته باشند.

3 - Liskov Substitution Principle (LSP)

اگر S یک زیر کلاس T باشد، آبجکت‌های نوع T باید بتو انندبدون تغییر دادن کد برنامه با آبجکت‌های نوع S جایگزین شوند.

4 – Interface Segregation Principle (ISP)

کلاس‌ها نباید مجبور باشن متدهایی که به اونها احتیاجی ندارند را پیاده‌سازی کنند .

5 - Dependency Inversion Principle (DIP)

کلاس‌های سطح بالا نباید به کلاس‌های سطح پایین وابسته باشند؛ هر دو باید وابسته به انتزاع (Abstractions)باشند. موارد انتزاعی نباید وابسته به جزییات باشند. جزییات باید وابسته به انتزاع باشند .

اصول SOLID به زبان ساده - اصل اول

Single Responsibility Principleکه اولین اصل از اصول SOLID هست رو امروز با هم بررسی میکنیم SRP مخفف Single Responsibility Principleهست. با ترجمه تحت‌اللفظی یعنی "اصلی تک مسئولیتی."نقل قول زیر توضیح رسمی هست که برای SRP ارائه شده : یک کلاس فقط باید به یک دلیل تغییر کنه.
این اصل به ما میگه که هر کلاسی که توی برنامه‌ی ما وجود داره، باید یک مسئولیت خاص و مشخص داشته. در واقع این کلاس باید فقط و فقط مسئول یک عملکرد توی برنامه باشه .این جمله رو همه شنیدیم :یک کار انجام بده ولی درست انجام بده!
به مثال زیر دقت کنین:

```
class User {
public information() {}
public sendEmail() {}
public orders() {}
}
```

توی این کلاس ما سه تا متد داریم. متد information که اطلاعات کاربر رو برمیگردونه. متد sendMail برای ارسال ایمیل به کاربر و متد orders سفارش‌های کاربر رو برمیگردونه.به نظرتون اگه کلاسی به اسم User داشته باشیم، هدف این کلاس چی هست؟ احتمالا اطلاعاتی از کاربر رو ذخیره کنه یا نمایش بده. درواقع مسئولیتی در حوزه مربوط به یک کاربر. اگه به کلاس دقت کنیم، میبینیم که توی این کلاس، فقط متد informationهست که با کلاس User مرتبط هست و بقیه متدها وظایفی متفاوت با این کلاس دارن.کلاس Userنباید مسئول ارسال ایمیل و یا هندل کردن سفارشات کاربر باشه. در این صورت کلاس ما با عملکردهای ذاتی خودش محصور شده نیست. یعنی کلاس User با یک سری عملکردهای غیرمرتبط آمیخته شده.این مسئله زمانی مشکل‌ساز میشه که میخوایم کلاس رو گسترش بدیم. مثلا ایمیل‌های مختلف و اختصاصی‌تر بفرستیم. که آخر کار نمیدونیم این کلاس User هست یا! Email
راه حل چیه؟🤔
خب راه حل اینه که عمکردهای اضافی رو از کلاس User جدا و به یک کلاس اختصاصی منتقل کنیم:

```
class User {
public information() {}
}
class Email {
public send(user: User) {}
}
class Order {
public show(user: User) {}
}
```

همونطور که میبینید، کلاس User ما خلوت‌تر، تمیزتر و مرتب تر شد. همچنین توسعه این کلاس و کلاس‌های دیگه راحت‌تر انجام میشه.

اصول SOLID به زبان ساده - اصل دوم

دومین اصل از اصولSOLID ، اصل باز/بسته یا Open/Closed Principle هست که به اختصار OCPگفته میشه.
تعریف رسمی این اصل به این صورت هست:
موجودیت‌های یک نرم‌افزار (کلاس‌ها، ماژول‌ها، توابع و ...) باید برای توسعه داده شدن، باز و برای تغییر دادن، بسته باشند
توی این اصل از کلمه‌های باز و بسته استفاده شده. این کلمات با چیزی که توی ذهنمون داریم یکم متفاوت هست. اول بذارید معنی کلاس باز و بسته رو با هم بررسی کنیم و بعد به توضیح این اصل بپردازیم.

چه زمانی به یک کلاس میگیم باز؟
به کلاسی که بشه اون رو توسعه داد، بشه از اون extend کرد، متدها و پراپرتی‌ها جدید اضافه کرد و ویژگی‌ها و رفتار اون رو تغییر داد، میگن باز.

چه زمانی به یک کلاس میگیم بسته؟
کلاسی که کامل باشه. یعنی 100% تست شده باشه که بتونه توسط بقیه کلاس‌ها استفاده بشه، پایدار باشه و در آینده تغییر نکنه. توی بعضی از زبان‌های برنامه‌نویسی یکی از راه‌های بسته نگه داشتن یک کلاس، استفاده از کلمه کلیدی finalهست.
خب حالا بپردازیم به توضیح اصلOCP :
اصل OCP میگه که ما باید کد رو جوری بنویسیم که وقتی میخوایم اون رو توسعه بدیم و ویژگی‌های جدید اضافه کنیم، مجبور نشیم اون رو تغییر بدیم و دستکاری کنیم. ویژگی‌های جدید باید براحتی و بدون دستکاری کردن قسمت‌های دیگه اضافه بشن. طبق این اصل کلاس باید همزمان هم بسته باشه و هم باز! یعنی همزمان که توسعه داده میشه (باز بودن)، تغییر نکنه و دستکاری نشه (بسته بودن).
خب حالا وقتشه که با مثال درک بهتری از این اصل داشته باشم. کد زیر رو در نظر بگیرید:

```
class Hello {
public say(lang) {
if (lang == 'pr') {
return 'درود';
} else if (lang == 'en') {
return 'Hi';
}
}
}
let obj = new Hello;
console.log(obj.say('pr'));
```

این کلاس، با توجه به زبان ورودی، به ما سلام میکنه. همونطور که میبینید درحال حاضر 2 تا زبان توسط متد sayپشتیبانی میشه. اگه بخوایم زبان‌های دیگه رو اضافه کنیم چطور؟ باید متد say رو ویرایش کنیم:

```
class Hello {
public say(lang) {
if (lang == 'pr') {
return 'درود';
} else if (lang == 'en') {
return 'Hi';
} else if (lang == 'fr') {
return 'Bonjour';
} else if (lang == 'de') {
return 'Hallo';
}
}
}
let obj = new Hello;
console.log(obj.say('de'));
```

اگه بخوایم تا 150 زبان به این لیست اضافه کنیم چطور؟

همونطور که میبینید وقتی ویژگی‌های جدید اضافه میشه، کلاس ما با توجه به نیازها دستکاری میشه. این اصلا خوب نیست. چون متد say در برابر تغییرات بسته نیست و همیشه از سمت بیرون در معرض دستکاری هست.
باید چکار کرد؟🤔
خب یه راه حل بهتر اینه که ما متد say رو کلی تر و عمومی تر بنویسیم. یعنی جوری که بدون توجه به تغییرات و نیازهای جدید، مستقل و دست نخورده باقی بمونه. به اصلاح Abstract کنیم. یعنی عمومی‌تر کردن.
خب برای اینکار، من مثال رو به شکل زیر تغییر میدم:

```
class Persian {
public sayHello() {
return 'درود';
}
}
class French {
public sayHello() {
return 'Bonjour';
}
}
class Hello {
public say(lang) {
return lang.sayHello();
}
}
let obj = new Hello;
console.log(obj.say(new Persian));
```

همونطور که دیدید من هر زبان رو به یک کلاس جدید منتقل کردم. و به این صورت هر وقت که بخوایم زبان جدید اضافه کنیم، کافیه یک کلاس جدید درست کنیم. همونطور که میبینید کلاس Hello و متد say دیگه دستکاری نیمشن.

البته این مثال میتونه با استفاده از interface ها بهینه‌تر هم نوشته بشه:

```
interface
LanguageInterface
{
                    sayHello(): string;
                }

                class Persian implements LanguageInterface {
                    public sayHello(): string {
                        return 'درود';
                    }
                }

                class French implements LanguageInterface {
                    public sayHello(): string {
                        return 'Bonjour';
                    }
                }

                class Hello {
```

```
    public say(lang: LanguageInterface): string {
        return lang.sayHello();
    }
}

let obj = new Hello;
console.log(obj.say(new Persian));
```

اصول SOLID به زبان ساده - اصل سوم

سومین اصل از اصولSOLID ، اصل جایگزینی لیسکوف یا Liskov Substitution Principle هست که به اختصار LSPگفته میشه. این اصل خیلی ساده هست. هم درک کردنش و هم پیاده سازیش. تعریف آکادمیک این اصل بصورت زیر هست:

اگر S یک زیر کلاس T باشه، آبجکت‌های نوع T باید بتونن بدون تغییر دادن کد برنامه با آبجکت‌های نوع S جایگزین بشن.

فرض کنیم یک کلاس داریم به اسم:A

```
class A { ... }
```

قراره از کلاس A آبجکت‌هایی ساخته بشه که توی جاهای مختلف برنامه استفاده کنیم. فرض کنیم کد زیر قسمت‌های مختلف برنامه هست که داره از کلاس A استفاده میکنه:

```
let x = new A;
        // ...
let y = new A;
        // ...
let z = new A;
```

حالا قراره کلاس A رو توسعه بدیم. برای همین کلاسی به اسم B رو میسازیم که از کلاس A مشتق میشه:

```
class B extends A { ... }
```

پس کلاسB ، یک زیر نوع از کلاس A هست. بالاتر دیدیم که توی برنامه، از کلاس A آبجکت‌هایی ساخته و استفاده شد. چون کلاس B یک زیر نوع از کلاس A هست، میخوایم توی برنامه و جایی که از کلاس A استفاده کردیم، بجای کلاسA ، از کلاس B استفاده کنیم. یعنی:

```
let x = new A new B;
                // ...
let y = new A new B;
                // ...
let z = new A new B;
```

اینجا ما جایگزینی انجام دادیم! کلاس B رو با کلاس A عوض کردیم. طبق اصلLSP ، وقتی جایگزینی انجام میدیم، برنامه نباید بخاطر جایگزینی دچار خطا بشه. همچنین کد برنامه هم نباید تغییر کنه. این اصل به همین سادگی هست. بیاید این قانون رو نقض کنیم تا اون رو بهتر متوجه بشیم. فرض کنیم یک کلاس داریم به اسم Note.این کلاس عملیات مختلفی انجام میده، مثل خواندن، بروزرسانی و حذف یادداشت‌های شخصی:

```
class Note {
 public constructor(id) {
                // ...
                }
public save(text): void {
        // save process
                }
                }
```

حالا یک کاربر میخواد از این کلاس توی برنامه‌ی خودش استفاده کنه:

```
let note = new Note(429);
note.save("Let's do this!");
```

خب میخوایم این کلاس رو توسعه بدیم. قراره یک ویژگی اضافه کنیم که بشه یادداشت‌های فقط خواندنی ساخت. یعنی باید متد save رو رونوشت کنیم و اجازه ندیم عملیات ذخیره کردن یادداشت انجام بشه. برای این کار یک زیرکلاس از Note میسازیم و اسم اون رو میذاریم ReadonlyNoteو متد save رو رونوشت میکنیم:

```
class ReadonlyNote extends Note {
                public save(text): void {
throw new Error("Can't update readonly notes");
                }
                }
```

در حالی که متد save توی کلاس اصلی به کاربر void برمیگردوند، توی کلاس جدید یک Exception برمیگردونیم که به کاربر بگیم عملیات save ممکن نیست. خب توی برنامه، اونجایی که از Note استفاده کردیم، یک جایگزینی انجام میدیم. یعنی بجای Note از ReadonlyNote استفاده میکنیم:

```
let note = new ReadonlyNote(429);
        note.save("Let's do this!");
```

خب چه اتفاقی میوفته؟
درحالی که کاربر بی اطلاع از تغییراتِ رخ داده هست، ناگهان یک چیز غیرمنتظره و یک Exception توی برنامه‌ش رخ میده! که به ناچار باید یک سری تغییرات توی برنامه خودش اعمال کنه.اینجا اصل LSP نقض شد. چون کلاسReadonlyNote ، رفتار و ویژگی‌های کلاس والد رو تغییر داد که کاربر مجبور میشه کد برنامه‌ش رو تغییر بده .
راه بهتر👌
برای اینکه این قسمت رو بهتر بنویسیم، یک کلاس جدا میسازیم برای یادداشت‌های قابل نوشتن .اسم کلاس رو میذارم .WritableNote یعنی یادداشت‌هایی که قابلیت بروزرسانی رو دارن و بعد متد save رو از کلاس Noteبه کلاس جدید منتقل کنیم:

```
class Note {
public constructor(id) {
// ...
}
}
class WritableNote extends Note {
public save(text): void {
// save process
}
}
```

نتیجه‌گیری
پس باید در نظر داشته باشیم وقتی که میخوایم یک کلاس رو با مشتق کردن توسعه بدیم، جاهایی از برنامه که از کلاس والد استفاده شده، باید بتونه بدون مشکل با کلاس‌های فرزند هم کار کنه. یعنی کلاس فرزند نباید ویژگی‌ها و رفتار کلاس والد رو تغییر بده. مثلا اگه کلاس والد یک متد داره که خروجی اون عددی هست، کلاس فرزند نباید این متد رو جوری رونوشت کنه که خروجی آرایه باشه.

اصول SOLID به زبان ساده - اصل چهارم

اصل چهارم از SOLID اصل جداسازی اینترفیس‌ها یا Interface Segregation Principle هست که به اختصار ISPگفته میشه. توضیح رسمی و آکادمیک این اصل بصورت زیر هست:
کلاس‌ها نباید مجبور باشن متدهایی که به اونها احتیاجی ندارند را پیاده‌سازی کنند .
این اصل میگه که ما باید اینترفیس (Interface) ها رو جوری بنویسیم که وقتی یک کلاس از اون استفاده میکنه، مجبور نباشه متدهایی که لازم نداره رو پیاده‌سازی کنه. یعنی متدهای بی‌ربط نباید توی یک اینترفیس کنار هم باشن. این اصل شباهت زیادی به اصل اول SOLID داره که میگه کلاس‌ها باید فقط مسئول انجام یک کار باشن. اینترفیس زیر رو درنظر بگیرید:

```
interface Animal {
                fly();
               run();
               eat();
                  }
```

این اینترفیس سه متد داره که باید توسط کلاس‌هایی که ازش استفاده میکنن پیاده‌سازی بشه. کلاس Dolphin(دلفین) رو در نظر بگیرید که از این اینترفیس استفاده میکنه:

```
class Dolphin implements Animal {
                        public fly() {
                        return false;
                                    }
                        public run() {
                              // Run
                                    }
                        public eat() {
                               // Eat
                                    }
                                    }
```

همونطور که میدونید، دلفین‌ها نمیتونن پرواز کنن. پس ما مجبور شدیم توی متد fly بنویسیم .return false اینجا قانون ISP نقض شد. چون کلاس دلفین مجبور به پیاده‌سازی متدی شد که از اون استفاده نمیکنه.
اگه بخوایم این اصل رو رعایت کنیم باید جداسازی اینترفیس انجام بدیم. پس متد fly رو به یک اینترفیس جدا منتقل میکنیم:

```
        interface Animal {
                        run();
                        eat();
                           }
interface FlyableAnimal {
                        fly();
                           }
```

بنابراین کلاس دلفین دیگه مجبور نیست متد fly رو پیاده‌سازی کنه و کلاس‌هایی که به این متد نیاز دارن، اینترفیس FlyableAnimal رو هم پیاده‌سازی میکنن:

```
                class Dolphin implements Animal {
                                public run() {
                                      // Run
                                           }
                                public eat() {
                                       // Eat
                                           }
                                           }
class Bird implements Animal, FlyableAnimal {
                                public run() { /* ... */ }
                                public eat() { /* ... */ }
                                 public fly() { /* ... */ }
                                           }
```

نتیجه
رعایت کردن این اصل به ما کمک میکنه کدهای خواناتر و تمیزتری داشته باشیم. توی شی‌گرایی باید یک نکته رو درنظر داشته باشیم که هر چی از کلی نویسی (عمومی‌نویسی) دوری کنیم، کدهای ما منسجم‌تر و ساختار یافته تر میشن. بنابراین کدها قابل استفاده مجدد میشن، تست و Refactor هم راحت‌تر انجام میشه.

اصول SOLID به زبان ساده - اصل پنجم

اصل پنجم و آخرSOLID ، اصل وارونگی وابستگی (Dependency Inversion Principle) نام داره که به اختصار DIPگفته میشه. توضیح رسمی و آکادمیک این اصل به صورت زیر هست :

کلاس‌های سطح بالا نباید به کلاس‌های سطح پایین وابسته باشند؛ هر دو باید وابسته به انتزاع (Abstractions)باشند. موارد انتزاعی نباید وابسته به جزییات باشند. جزییات باید وابسته به انتزاع باشند .

کلاس سطح پایین

به کلاس‌هایی گفته میشه که مسئول عملیات اساسی و پایه‌ای توی نرم‌افزار هستن. مثل برقراری ارتباط با دیتابیس یا هارددیسک، کار با ایمیل و. ...

کلاس سطح بالا

کلاس‌هایی که عملیات پیچیده‌تر و خاص‌تری انجام میدن و برای انجام این کار از کلاس‌های سطح پایین استفاده میکنن. مثلا کلاس گزارش‌گیری، برای ثبت و خوندن گزارش، به کلاس دیتابیس یا هارددیسک نیاز داره. کلاسUsers ، برای اطلاع‌رسانی به کلاس ایمیل نیاز داره.

مفهوم انتزاع(Abstraction)

کلاس‌های انتزاعی کلاس‌هایی هستن که قابل پیاده‌سازی نیستن اما به عنوان یک طرح و الگو برای کلاس‌های دیگه در نظر گرفته میشوند. مثلا یک کلاس انتزاعی برای گربه، زرافه، پلنگ و پنگوئن، میشه کلاس Animal.خود Animal به خودی خود قابل پیاده‌سازی نیست. بلکه یک طرح کلی برای حیوونایی هستند که مثال زدیم. پس تک تک این حیوون‌ها یک ورژن کلی‌تر دارند که میتونیم اون رو Animal بنامیم.

جزییات

منظور از جزییات توی تعریف این اصل، جزییات یک کلاس مثل نام و ویژگی متدها و پراپرتی‌ها هست.

خب بپردازیم به بررسی این اصل. ابتدا کد زیر رو در نظر بگیرید:

```
class MySQL {
public insert() {}
public update() {}
public delete() {}
}
class Log {
private database;
constructor() {
this.database = new MySQL;
}
}
```

فرض کنیم یک کلاس سطح پایین داریم مثلا دیتابیس .MySQL و یک سری کلاس سطح بالا مثلا گزارش‌گیری (Log)از این کلاس استفاده میکنن. اگه بخوایم یک تغییر توی کلاس دیتابیس انجام بدیم، ممکنه بطور مستقیم تاثیر بذاره روی کلاس‌هایی که ازش استفاده میکنن. مثلا اگه توی کلاس MySQL اسم متد رو تغییر بدیم و یا پارامترها رو کم و زیاد کنیم، نهایتا توی کلاس Log این تغییرات رو باید اعمال کنیم.

همچنین کلاس‌های سطح بالا قابل استفاده مجدد نیستن. مثلا اگه بخوایم برای کلاس Log از دیتابیس‌های دیگه مثلا MongoDB یا هارددیسک استفاده کنیم باید کلاس Log رو تغییر بدیم یا یک کلاس جدا براساس هر نوع دیتابیس بسازیم.

خب همونطور که می‌بینید اگه یک کلاس سطح بالا وابسته به یک کلاس سطح پایین باشه این مشکلات به وجود میاد.

راه حل

برای حل این مشکل باید با اینترفیس، یک لایه انتزاعی درست کنیم. با این کار کلاس Log دیگه وابسته به یک کلاس خاص برای ذخیره‌سازی و خوندن اطلاعات نیست و میتونیم هر نوع دیتابیسی رو استفاده کنیم و برای کلاس Log اهمیتی نداره که با چه نوع دیتابیسی داره کار میکنه. چون وابسته به انتزاع هست.

ابتدا یک اینترفیس میسازیم برای اینکه کلاس‌های سطح بالا و سطح پایین رو وابسته به این اینترفیس کنیم:

```
interface Database {
insert();
update();
delete();
}
```

حالا کلاس‌های سطح پایین باید این اینترفیس رو پیاده‌سازی کنن تا وابسته به انتزاع بشن:

```
class MySQL implements Database {
    public insert() {}
    public update() {}
    public delete() {}
}
class FileSystem implements Database {
    public insert() {}
    public update() {}
    public delete() {}
}
class MongoDB implements Database {
    public insert() {}
    public update() {}
    public delete() {}
}
```

و نهایتا توی کلاس‌های سطح بالا، وابستگی به یک کلاس خاص رو به اینترفیس منتقل میکنیم. کلاس‌های سطح بالا زمانی وابسته به انتزاع میشن که بجای استفاده مستقیم از کلاس‌های سطح پایین، از یک اینترفیس (رابط) استفاده کنن:

```
class Log {
    private db: Database;
public setDatabase(db: Database) {
    this.db = db;
}
    public update() {
    this.db.update();
}
}
```

همونطور که می‌بینیم وابستگی به یک کلاس خاص از بین رفت و میتونیم هر نوع دیتابیسی رو برای کلاس Logاستفاده کنیم:

```
let logger = new Log;
logger.setDatabase(new MongoDB);
// ...
logger.setDatabase(new FileSystem);
// ...
logger.setDatabase(new MySQL);
logger.update();
```

نتیجه‌گیری

مثل بقیه اصولSOLID ، این اصل هم تلاش داره وابستگی بین اجزا رو کمتر کنه تا بتونیم کدهای قابل نگهداری، تمیزتر و قابل توسعه‌تر بنویسیم. اما در نظر داشته باشید که مثل بقیه اصول توی دنیای برنامه‌نویسی، این اصل هم باید با چشم باز اعمال بشه. گاهی وقتا اعمال کردن یک سری اصول نه تنها مشکل رو حل نمیکنه، بلکه باعث پیچیده‌تر شدن و گنگ شدن کد برنامه میشه.