به نام خدا

موضوع تحقیق:

اصول برنامه نویسی YANGI , KISS , DRY , SOLID

تفاوت constructorو deconstructor

Gc.collect

تهیه کننده:

محمد نسیمی فر

رشته تحصیلی:

مهندسی فناوری اطلاعات

درس:

برنامه نویسی سمت سرویس گیرنده

استاد:

جناب آقای میثاق یاریان

تحقیق شماره ۵

همه ی برنامه نویسان می توانند کد بزنند ولی مهم این است که کد ها بهینه باشند. شما با این دیدگاه که کد شما فقط مشکلتان را حل کند و بدون ارور اجرا شود نباید بسمت جلو حرکت کنید. بلکه با این هدف باید بسمت جلو بروید که کد شما نه تنها در حال حاضر بلکه در تمامی شرایط بخوبی کار کند همینطور در مدت زمان طولانی باید بدون هیچ مشکلی اجرا شود. ما در این مقاله به شما ده اصل آموزش می دهیم تا با استفاده از این اصول برنامه نویسی به یک برنامه نویس حرفه ای تبدیل شوید.

اصل YAGNI (You Aren’t Gonna Need It)

تعداد زیادی از توسعه‌دهندگان عادت به نوشتن کدهایی بیش از حد مورد نیاز دارند. این توسعه‌دهندگان همیشه به فکر قابلیت‌هایی هستند که ممکن است در آینده به کار بیایند. تمام این اصل نیز بر همین اساس است که « شما به آن نیاز نخواهید داشت »

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.

اصل KISS (keep it simple, stupid)

Martin Fowler در این‌باره می‌گوید:

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

این جمله تمام مفهوم KISS را در خود جای داده است. در واقع این اصل به سادگی برای ما قابل درک است اما پیاده‌سازی عملی آن دشوار است. اگر شما به یک کد را که به خوبی KISS را پیاده‌سازی کرده‌باشد بررسی کنید، این سوال برایتان پیش خواهد آمد که بقیه‌ی کدهای این پروژه کجاست؟!

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.

اصل DRY (Don’t repeat yourself)

قطعا برایتان پیش‌آمده که بخشی از کد هایتان را مشاهده کرده باشید که ظاهری آشنا دارند؛ در این لحظه متوجه می‌شوید که این قسمت از کد را باز هم در پروژه‌تان نوشته‌اید.

شما همیشه باید در هنگام برخورد به چنین مواردی بدانید که احتمالا می‌توانید کدهایتان را بهبود بخشید؛ زیرا باید کارهای تکراری را به حداقل رساند. (Don’t repeat yourself)

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.

کاربرد SOLID چیست؟

در حقیقت اصول سالید، دستورالعمل‌هایی هستند که می‌توان هنگام کار بر روی یک نرم‌افزار، آن‌ها را برای از بین بردن، عوامل نامطلوب در کد، اعمال کرد. اصول SOLID از طریق فراهم کردن چارچوبی انجام می‌گیرد که برنامه نویس با استفاده از آن می‌تواند کدهای برنامه را اصلاح و بازسازی کند تا قابلیت خوانایی و توسعه‌پذیری آن افزایش یابد.

هدف کلی اصول SOLID کاهش وابستگی‌ها و نوشتن کد ماژولار است یعنی برنامه نویسان بتوانند یک قسمت از کد را بدون اینکه روی سایر کدها تاثیری بگذارد، تغییر دهند. علاوه بر این، اصول سالید، با حذف کردن کدهای زاید و تکراری باعث می‌شود که خوانایی، قابلیت نگهداری و قابلیت توسعه کدها افزایش یابد. استفاده از اصول طراحی SOLID، مشکلات کدنویسی را کاهش می‌دهد و در ساختن نرم‌افزار تطبیقی، مؤثر و چابک، به برنامه نویسان کمک می‌کند.

مزایای اصول سالید چیست؟

مزایای اصول SOLID عبارت‌اند از:

- افزایش قابلیت نگهداری کدها
- خوانایی بهبود یافته
- افزایش مقیاس پذیری کدها
- افزایش قابلیت تست کدها
- حذف کدهای اضافی و کاهش تکرار کدها
- قابلیت استفاده مجدد از کدها
- بهبود عملکرد برنامه
- و...

معایب اصول سالید چیست؟

در قسمت قبل با مزایای اصول SOLID آشنا شدیم. اما معایب اصول SOLID چیست؟ معایب اصول SOLID عبارت‌اند از:

- افزایش پیچیدگی
- کلاس‌ها و رابطهای بیشتر
- مشکل در درک برای مبتدیان
- زمان و تلاش زیادی برای رعایت اصول SOLID نیاز است.

- و...

آشنایی با اصول پنج‌گانه SOLID در برنامه نویسی شیءگرا

در قسمت‌های قبل دانستیم که اصول طراحی SOLID چیست؟ در این قسمت به معرفی کامل ۵ اصل SOLID در برنامه نویسی شیءگرا همراه با مثال ساده آشنا خواهیم شد. اصول پنج‌گانه SOLID عبارت‌اند از:

- اصل اول: Single Responsibility Principle (اصل یگانگی مسئولیت)
- اصل دوم: Open-Closed Principle (اصل باز-بسته)
- اصل سوم: Liskov Substitution Principle (اصل جانشینی لیسکوف)
- اصل چهارم: Interface Segregation Principle (اصل تفکیک رابط‌ها)
- اصل پنجم: Dependency Inversion Principle (اصل وارونگی وابستگی)

اصل اول SOLID (Single Responsibility Principle) چیست؟

اصل اول Single Responsibility Principle مخفف SRP در SOLID به معنی «یگانگی مسئولیت» است یعنی «یک کلاس باید تنها یک وظیفه داشته باشد نه بیشتر» به عبارت دیگر «یک کلاس باید تنها یک دلیل برای تغییر داشته باشد و نه بیشتر»

بگذارید یک مثال عملی بزنیم تا بهتر متوجه این موضوع شوید:

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

در کلاس User بالا ۳ متد متفاوت داریم. متد information() اطلاعات کاربر را نشان می‌دهد، متد sendEmail()به کاربران ایمیل ارسال می‌کند و متد orders() سفارش‌های کاربران را نشان می‌دهد. کلاس Userاز اسمش مشخص است که وظیفه آن مدیریت کاربران مانند ثبت و نمایش آن‌ها را دارد.

در مثالی که مشاهده می‌کنید، User سه وظیفه متفاوت دارد و «اصل تک مسئولیتی» کلاس‌ها را نقض می‌کند. چون وظیفه ارسال ایمیل و نمایش سفارشات بر عهده کلاس User نیست. مشکل این روش این است که وقتی بعدا بخواهیم قسمت ایمیل‌ها و سفارشات را تغییر دهیم، به راحتی نمی‌توانیم آن‌ها را پیدا کنیم چون در کلاس مخصوص خود قرار ندارند و داخل کلاس کاربران قرار گرفته‌اند.

این کار باعث سردرگمی توسعه‌دهندگان می‌شود. برای رفع این مشکل از اصل اول سالید (Single Responsibility Principle) استفاده می‌کنیم. به مثال زیر توجه کنید:

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

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

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

در مثال بالا، متد اطلاعات کاربران در کلاسUser ، متد ارسال ایمیل در کلاس Email و متد نمایش سفارشات در کلاس Order قرار می‌گیرد. یعنی هر کلاس فقط و فقط وظیفه خاص خودش را دارد نه بیشتر .اصل اول SOLIDعلاوه بر اینکه باید در کلاس‌ها رعایت شود، باید در متدها و توابع نیز رعایت شود. به مثال زیر توجه کنید:

```
class Mailer {
  public send(text) {
    mailer = new Mail();
    mailer.login();

    mailer.send(text);
  }
}

mail = new Mailer;
mail.send('Hello');
```

در کلاس Mailer متد send() مسئول انجام دو وظیفه متفاوت است. احراز هویت (login) و ارسال ایمیل (send)نباید هر دو داخل یک متد باشند. یک متد نیز باید مسئول انجام یک وظیفه باشد. اگر بخواهیم از اصل اول SOLID یعنی SRP پیروی کنیم، کدهای بالا را به کدهای پایین تغییر می‌دهیم:

```
class Mailer {
  private mailer;

  public constructor(mailer) {
    this.mailer = mailer;
  }

  public send(text) {
    this.mailer.send(text);
  }
}

myEmail = new MyEmailService;
myEmail.login();

mail = new Mailer(myEmail);
mail.send('Hello');
```

حالا متد send() فقط مسئول انجام یک وظیفه (ارسال ایمیل) است و «اصل تک مسئولیتی» در متدها نیز رعایت شده است.

اصل دوم SOLID (Open-Closed Principle) **چیست؟**

اصل دوم Open-Closed Principle مخفف OCP در SOLID به معنی «اصل باز-بسته» است یعنی «اجزای نرم‌افزار باید برای توسعه باز و برای اصلاح بسته باشد» به عبارت دیگر «بتوانیم به نرم‌افزار ویژگی جدیدی اضافه کنیم (باز) بدون اینکه ویژگی جدید باعث تغییر در سایر قسمت‌های نرم‌افزار شود (بسته)»

اصل دوم سالید (Open-Closed Principle) می‌گوید که کدها باید طوری نوشته شوند که وقتی می‌خواهیم بعدا آن‌ها را توسعه دهیم و ویژگی‌های جدیدی به آن اضافه کنیم، نباید مجبور باشیم کدهای قبلی را تغییر یا دستکاری کنیم. ویژگی جدید باید به راحتی و بدون تغییر سایر کدها اضافه شود. به مثال زیر توجه کنید:

```
class Hello {
 public say(lang) {
  if (lang == 'fa') {
   return 'درود';
  } else if (lang == 'en') {
   return 'Hi';
  }
 }
}

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

کلاس Hello یک متد با نام say() دارد که با توجه به پارامتر lang (زبان)، سلام می‌کند. در این مثال ۲ زبان فارسی (fa) و انگلیسی (en) وجود دارند. حالا می‌خواهیم زبان فرانسه (fr) و آلمانی (de) را به کلاس خود اضافه کنیم:

```
class Hello {
 public say(lang) {
  if (lang == 'fa') {
   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'));
```

در کد بالا با افزودن زبان جدید، متد say() نیز تغییر کرد و اصل دوم سالید (Open-Closed Principle) نقض شد. چون با افزودن ویژگی جدید (در این مثال زبان)، متد say() تغییر داده شد. همانطور که در قسمت بالا اشاره کردیم، اصل دوم SOLID می‌گوید که» برنامه باید برای توسعه باز و برای تغییر (اصلاح) بسته باشد .« در مثال بالا با توسعه‌ی کد، متد مورد نظر تغییر کرد و این اصل نقض شد. برای رفع این مشکل از کد زیر استفاده می‌کنیم:

```
class Persian {
 public sayHello() {
  return 'درود';
```

```
  }
}

class French {
  public sayHello() {
    return 'Bonjour';
  }
}

class Hello {
  public say(lang) {
    return lang.sayHello();
  }
}

myHello = new Hello();
myHello.say(new Persian());
```

در مثال بالا هر زبانی که اضافه می‌شود، یک کلاس جدید برای آن ایجاد می‌کنیم تا کلاس Hello و متد say()دستخوش تغییر نشوند. در این صورت برنامه برای توسعه باز و برای تغییر بسته می‌شود و اصل دوم SOLIDرعایت می‌شود.

اصل سوم (Liskov Substitution Principle) SOLID چیست؟

اصل سوم Liskov Substitution Principle مخفف LSPدر SOLIDبه معنی» اصل جانشینی لیسکوف « است یعنی» کلاس‌های فرزند می‌توانند جانشین کلاس‌های والد شوند «اما به این نکته توجه کنید که »کلاس‌های فرزند نباید رفتار و ویژگی‌های کلاس والد را تغییر دهند«

به بیان ساده‌تر، اصل سوم سالید (Liskov Substitution Principle) می‌گوید که» اگر S یک زیر کلاس T باشد، آبجکت‌های نوع T باید بتوانند بدون تغییر دادن کد برنامه، با آبجکت‌های نوع S جایگزین شوند .« فرض کنید یک کلاس A مانند زیر داریم:

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

حال می‌خواهیم از کلاس A یک‌سریObject هایی بسازیم و در جاهای مختلف برنامه از آن‌ها استفاده کنیم. کد زیرObject هایی در جاهای مختلف برنامه هستند که از کلاس A استفاده می‌کنند:

```
x = new A;

// ...

y = new A;

// ...

z = new A;
```

حالا می‌خواهیم کلاس A را توسعه دهیم برای همین یک کلاس به نام B می‌سازیم که از کلاس A مشتق (ارث‌بری) شده است:

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

حالا کلاس B یک زیرنوع از کلاس A است چون کلاس B از کلاس A ارث‌بری کرده است. در کد قبل مشاهده کردیم که از کلاس A با new کردن یک سری آبجکت‌هایی ساختیم و در جاهای مختلف برنامه از آن استفاده

کردیم. چون کلاس B یک زیرنوع از کلاس A است، می‌توانیم به جای ساختن آبجکت از کلاسA ، آبجکت‌هایی از کلاس B بسازیم:

```
x = new B;

// ...

y = new B;

// ...

z = new B;
```

در کد بالا اصل سوم) SOLID اصل جانشینی لیسکوف (را انجام دادیم و کلاس‌های والد و فرزند جانشین هم شدند. توجه داشته باشید که طبق اصل سوم سالید، نباید کارکرد برنامه با جانشینی کلاس‌های والد و فرزند مختل شود، همچنین کدهای برنامه نیز نباید تغییر کنند.

اصل چهارم (Interface Segregation Principle) SOLID چیست؟

اصل چهارم Interface Segregation Principle مخفف ISP در SOLID به معنی «اصل تفکیک رابط‌ها» است یعنی «کلاس‌ها نباید مجبور باشند، متدهایی که به آن‌ها احتایج ندارند را پیاده‌سازی کنند».

به بیان ساده‌تر، اصل چهارم سالید (Interface Segregation Principle) می‌گوید که «اینترفیس (Interface)ها را باید طوری بنویسیم که وقتی یک کلاس از آن استفاده می‌کند، مجبور نباشد متدهایی که لازم ندارد را پیاده‌سازی کند». یعنی «استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است».

ایده اصل چهارم SOLID داستان جالبی دارد. زمانی که شرکت زیراکس (Xerox) تولید کنند دستگاه‌های چاپ، کپی، اسکنر، فکس و... سخت‌افزار این محصولات را ساخت، با آقای رابرت سی. مارتین تماس گرفتند و گفتند که بخش نرم‌افزاری این دستگاه‌ها را کدنویسی کند.

شرکت زیراکس یک دستگاه چندکاره ساخته بود که عملیات چاپ، کپی، اسکن و فکس همه در یک دستگاه انجام می‌شد. آقای مارتین یک کلاس (Class) ساخت و همگی متدهای چاپ، کپی، اسکن و فکس را در این کلاس قرار داد. برنامه به خوبی کار می‌کرد اما به دلیل اینکه زیراکس یک دستگاه چندکاره بود، قیمت آن بسیار زیاد بود و همه‌ی مشتری‌ها نمی‌توانستند آن را تهیه کنند.

مثلا مشتری که فقط به دستگاه کپی احتیاج داشت، لزومی نداشت که از چاپ، اسکن و فکس نیز استفاده کند اما مجبور بود هزینه آن را پرداخت کند (اصل چهارم SOLID: استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است). زیراکس برای حل این مشکل هر

عملیات را در دستگاه جداگانه قرار داد، بدین ترتیب عملیات چاپ در دستگاه چاپ، عملیات کپی در دستگاه کپی، عملیات اسکن در دستگاه اسکن و عملیات فکس در دستگاه فکس تعبیه شد.

حالا هر مشتری می‌توانست با توجه به نیاز خود دستگاه خود را با قیمت مناسب‌تر تهیه کند. اما یک مشکل در اینجا ایجاد شد. متدهایی که آقای مارتین نوشته بود، همه داخل یک کلاس بودند و با جداکردن دستگاه‌ها باید متدها نیز جدا می‌شدند و هر متد باید به دستگاه مورد نظر انتقاد داده می‌شد. مثلا دستگاه کپی نیازی نداشت متد چاپ، اسکن و فکس را در خود داشته باشد (اینترفیس (Interface)ها را باید طوری بنویسیم که وقتی یک کلاس از آن استفاده می‌کند، مجبور نباشد متدهایی که لازم ندارد را پیاده‌سازی کند).

آقای مارتین دوباره هر متد را به‌صورت جداگانه نوشت تا هر دستگاه متد خاص خودش را داشته باشد. آقای مارتین این تجربیات را در کتاب کدنویسی تمیز، با عنوان اصل چهارم از اصول SOLID بیان کرد. به مثال زیر توجه کنید:

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

اینترفیس Animal (حیوان) سه متد با نام‌های fly() برای پرواز، run()برای دویدن یا حرکت و eat() برای خوردن دارد. حالا کلاس زیر را در نظر بگیرید:

```
class Cat implements Animal {
 public fly() {
  return false;
 }

 public run() {
  // Run
 }

 public eat() {
  // Eat
 }
}
```

در کلاس بالا Cat (گربه) از کلاس Animal استفاده کرده است تا ویژگی مشترک حیوانات را دریافت کند. اما متد fly() برای پرواز کردن است و یک Cat نمی‌تواند پرواز کند. بخاطر همین متد fly() را return false قرار می‌دهیم تا نتواند از آن استفاده کند و فقط از متد run() و eat() استفاده کند .اصل چهارم سالید(ISP) ، در اینجا نقض شد. چون کلاس Cat از مجبور شد از متدی به نام fly() استفاده کند که به آن نیازی ندارد.

برای اینکه اصل چهارم SOLID را رعایت کنیم، از کد زیر استفاده می‌کنیم:

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

interface FlyableAnimal {
 fly();
}
```

همانطور که در کد بالا مشاهده می‌کنید، اینترفیس‌ها را از هم جدا کردیم. یعنی یک اینترفیس Animal داریم که متد حرکت و خوردن دارد و در تمامی حیوانات مشترک است. یک اینترفیس دیگر به نام FlyableAnimalداریم که متد پرواز دارد و فقط حیواناتی که قابلیت پرواز دارند از این ایترفیس استفاده می‌کنند. حالا می‌خواهیم مثال قبلی که قانون ISP در SOLID را نقض کرد، به شکل زیر ریفکتور کنیم:

```
class Cat implements Animal {
 public run() {
  // Run
 }

 public eat() {
  // Eat
 }
}

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

در کد بالا کلاس Cat فقط از روی کلاس Animal پیاده‌سازی شده است و کلاس Bird (پرنده) هم از کلاس Animal (ویژگی مشترک حرکت کردن و خوردن) و هم از کلاس FlyableAnimal (ویژگی پرواز) که مخصوص خودش است استفاده کرده است. در این صورت اصل چهارم سالید (کلاس‌ها نباید مجبور باشند، متدهایی که به آن‌ها احتایج ندارند را پیاده‌سازی کنند) رعایت شده و هر اینترفیس تفکیک شده است.

این نکته را در نظر داشته باشید که در شیءگرایی باید از عمومی نویسی دوری کنیم و هر کلاس وظیفه خاص خودش را انجام دهد تا برنامه منسجم و ساختاریافته‌تری داشته باشیم. این کار باعث می‌شود که خوانایی، تست قابلیت استفاده مجدد از کدها و ریفکتور کدها آسان‌تر شود.

اصل پنجم SOLID (Dependency Inversion Principle) چیست؟

اصل پنجم Dependency Inversion Principle مخفف DIP در SOLID به معنی «اصل وارونگی وابستگی» است یعنی «کلاس‌های سطح بالا نباید به کلاس‌های سطح پایین وابسته باشند؛ هر دو باید وابسته به انتزاع (Abstraction) باشند».

به بیان ساده‌تر، اصل پنجم سالید (Dependency Inversion Principle) می‌گوید که «موارد انتزاعی نباید وابسته به جزئیات باشند بلکه جزئیات باید وابسته به انتزاع باشند». بگذارید قبل از مثال برای اصل پنجم SOLID، ابتدا باید ببینیم که منظور از کلاس‌های سطح بالا و پایین و انتزاعی چیست؟

کلاس‌های سطح پایین: به کلاس‌هایی گفته می‌شوند که مسئول کارهای اساسی و پایه‌ای در نرم‌افزار هستند. مانند کلاسی که با دیتابیس ارتباط برقرار می‌کند یا کلاسی که برای ارسال ایمیل استفاده می‌شود.

کلاس‌های سطح بالا: کلاس‌هایی هستند که عملیات پیچیده و خاص‌تری انجام می‌دهند و برای این کار از کلاس‌های سطح پایین استفاده می‌کنند. برای مثال کلاس گزارش‌گیری برای ثبت و خواندن گزارش، به کلاس دیتابیس نیاز دارد و کلاس Users، برای اطلاع‌رسانی به کاربرها به کلاس ایمیل نیاز دارد.

کلاس‌های انتزاعی (Abstraction): کلاس‌هایی هستند که به خودی خود قابل پیاده‌سازی نیستند و فقط یک طرح و نقشه هستند که کلاس‌های دیگر از آن استفاده می‌کنند. برای مثال کلاس Animal (حیوان) به خودی خود قابل پیاده‌سازی نیست اما از روی آن می‌توان کلاس‌هایی به نام Cat (گربه)، Dog (سگ) و... ساخت. پس تک تک گربه، سگ و... یک کلاس انتزاعی به نام حیوان دارند که از آن تبعیت می‌کنند. برای آشنایی بیشتر با مفهوم انتزاع، در مقاله ٤ اصل شیءگرایی، ویژگی Abstraction را مطالعه کنید.

منظور از جزئیات در تعریف اصل پنجم SOLID، همان جزییات یک کلاس مثل نام، پراپرتی‌ها و متدهاست.

اکنون به سراغ مثال برای اصل پنجم سالید می‌رویم. به کد زیر توجه کنید:

```
class MySQL {
  public insert() {}
  public update() {}
  public delete() {}
}

class Log {
  private database;

  constructor() {
    this.database = new MySQL;
  }
}
```

کلاس MySQL یک کلاس سطح پایین برای اتصال به دیتابیس است. کلاس Log نیز یک کلاس سطح بالا برای گزارش‌گیری است که از کلاس سطح پایین MySQL استفاده می‌کند. حالا فرض کنید می‌خواهیم نام کلاس MySQL را تغییر دهیم، در این صورت باید در کلاس Log قسمت new MySQL نیز نام کلاس را تغییر دهیم. پس اصل پنجم SOLID در اینجا نقض می‌شود.

چون کلاس سطح پایین به کلاس سطح بالا وابسته است و هرگونه تغییر در کلاس سطح بالا، منجر به تغییر در کلاس سطح پایین می‌شود .اصل پنجم SOLID اینگونه بود که» کلاس‌های سطح بالا نباید به کلاس‌های سطح پایین وابسته باشند؛ هر دو باید وابسته به انتزاع (Abstraction) باشند «ولی در مثال بالا وابسته بود.

همچنین اگر بخواهیم برای کلاس سطح بالایLog ، به جای دیتابیس MySQL ار دیتابیس دیگری مانند MongoDBاستفاده کنیم، در این صورت دوباره باید کلاس Log را تغییر دهیم یا یک کلاس جداگانه برای هر دیتابیس بنویسیم. چون کلاس سطح بالا به کلاس سطح پایین وابسته شده است. برای رفع این مشکل از مثال زیر استفاده می‌کنیم:

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

در مثال بالا یک اینترفیس Database ساختیم که دارای متدهای insert() برای ذخیره سازی، update()برای بروزرسانی و delete() برای حذف کردن می‌باشد. حال باید کلاس‌های سطح بالا و پایین را به اینترفیس خود وابسته کنیم (انتزاع). به مثال زیر توجه کنید:

```
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() {}
}
```

در کد بالا، کلاس‌های سطح پایین MySQL و FileSystem و MongoDB از روی اینترفیس Database پیاده‌سازی شدند و وابسته به انتزاع هستند. حالا باید کلاس سطح بالای Log (گزارش‌گیری) را نیز به اینترفیس (انتزاع) وابسته کنیم تا به‌جای وابسته بودن به کلاس‌های سطح پایین، به انتزاع وابسته باشند. به مثال زیر توجه کنید:

```
class Log {
  private db: Database;

  public setDatabase(db: Database) {
    this.db = db;
  }

  public update() {
    this.db.update();
  }
}
```

همانطور که در کد بالا مشاهده می‌کنید، وابستگی کلاس سطح بالا به کلاس سطح پایین از بین رفت و وابسته به انتزاع شد. حالا می‌توانیم از هر نوع دیتابیس برای کلاس Log استفاده کنیم:

```
logger = new Log;

logger.setDatabase(new MongoDB);
// ...
logger.setDatabase(new FileSystem);
// ...
logger.setDatabase(new MySQL);

logger.update();
```

اصل پنجم SOLID نیز مانند بقیه اصولSOLID ، در جهت کاهش وابستگی بین اجزای برنامه است تا بتوانیم کدهای خوانا، قابل استفاده مجدد و قابل توسعه بنویسیم.

جمع‌بندی

اصول SOLID چیست؟ در این مقاله کاربرد اصول SOLID در برنامه نویسی شیءگرا را بررسی کردیم. به‌طور کلی اصول SOLID از ۵ اصل تشکیل شده است. اصل اول Single Responsibility Principle، اصل دوم Open-Closed Principle، اصل سوم Liskov Substitution Principle، اصل چهارم Interface Segregation Principle و اصل پنجم Dependency Inversion Principle است. اصول سالید برای افزایش قابلیت انعطاف‌پذیری، تغییرپذیری و قابلیت توسعه در برنامه‌های شیءگرا استفاده می‌شود. این نکته را در نظر داشته باشید که برنامه‌های کمی وجود دارند که هر ۵ اصل سالید را رعایت کرده‌اند، رعایت همه‌ی اصول سالید در نرم‌افزار به نوعی سخت است. یک برنامه نویس باید دانش کافی در مورد اصول SOLID داشته باشد تا بتواند یک برنامه اصولی طراحی کند. گاهی وقت‌ها ممکن است عدم تجربه‌ی توسعه‌دهنده در پیاده‌سازی اصول طراحی SOLID در برنامه نویسی شیءگرا، نه تنها باعث بهبود عملکرد نرم‌افزار نشود بلکه باعث پیچیدگی بیشتر نرم‌افزار نیز بشود.

تفاوت constructor و Deconstructor

سازنده (Constructor) و مخرب (Destructor) دو مفهوم مهم در برنامه‌نویسی شیءگرا هستند. آنها در زبان‌های برنامه‌نویسی مختلف به کار می‌روند و وظایف متفاوتی دارند. در ادامه، تفاوت‌های میان Constructor و Destructor را توضیح خواهم داد.

Constructor:

Constructor یک متد ویژه است که هنگام ایجاد یک شیء از یک کلاس صدا زده می‌شود. وظیفه اصلی Constructor، مقداردهی اولیه به ویژگی‌های شیء (متغیرهای عضو) است. یعنی وقتی یک شیء از یک کلاس ساخته می‌شود، Constructor فراخوانی می‌شود و مقادیر اولیه متغیرها تعیین می‌شوند. Constructor معمولاً نامی مشابه نام کلاس دارد و هیچ نوع مقدار برگشتی ندارد.

```
using System;

public class Person
{
    private string name;
    private int age;

    // Constructor
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
```

```
    // Destructor
    ~Person()
    {
        Console.WriteLine("Destructor called, Person object deleted");
    }

    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name}, Age: {age}");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person1 = new Person("John", 25);
        person1.DisplayInfo();

        // The destructor will be called automatically when the object is no longe
r in use
    }
}
```

در این مثال، کلاس Person دارای Constructor و Destructor است. Constructor با استفاده از کلیدواژه public و نام کلاس تعریف شده است و وظیفه‌اش مقداردهی اولیه به ویژگی‌های name و age است. Destructor با استفاده از علامت تیره معکوس (~) و نام کلاس تعریف شده است و هنگامی که شیء از حافظه حذف می‌شود، فراخوانی می‌شود و پیام "Destructor called, Person object deleted" را چاپ می‌کند.

در Main، یک شیء از کلاس Person با استفاده از Constructor ایجاد می‌شود و اطلاعات آن با فراخوانی متد DisplayInfo چاپ می‌شود. هنگامی که برنامه به پایان می‌رسد و شیء person1 دیگر در حافظه استفاده نمی‌شود، Destructor به صورت خودکار فراخوانی می‌شود و پیام مربوطه چاپ می‌شود.

بطور کلی، Constructor و Destructor در زبان C# به ترتیب هنگام ایجاد شیء (Constructor) و هنگام حذف شیء (Destructor) فراخوانی می‌شوند.

Constructor در ابتدای ایجاد یک شیء از یک کلاس فراخوانی می‌شود و وظیفه اصلی آن، مقداردهی اولیه به ویژگی‌های شیء (متغیرهای عضو) است. در مثال قبل، Constructor کلاس `Person` اطلاعات `name` و `age` را با استفاده از پارامترها دریافت کرده و به ویژگی‌های متناظرشان اختصاص می‌دهد.

Destructor همچنین یک متد ویژه است که هنگام حذف یک شیء از حافظه فراخوانی می‌شود. وظیفه Destructor، انجام عملیاتی قبل از حذف شیء (مانند پاکسازی منابع) است. در مثال C# قبلی، Destructor کلاس `Person` هنگامی که شیء از حافظه حذف می‌شود، پیام " Destructor called, Person object deleted" را چاپ می‌کند.

توجه کنید که در C#، شما نیازی به خودکار صدا زدن Destructor ندارید. زبان C# به طور خودکار مدیریت حافظه را انجام می‌دهد و Destructor هنگامی که شیء دیگر در حال استفاده نیست و باید از حافظه حذف شود، به صورت خودکار فراخوانی می‌شود.

استفاده از Destructor برای پاکسازی منابعی که توسط شیء استفاده شده‌اند (مانند فایل‌ها، اتصالات پایگاه داده و غیره) مفید است. با فراخوانی Destructor، می‌توانید این منابع را آزاد کرده و منابع مصرفی را به طور صحیح مدیریت کنید.

به طور خلاصه، Constructor در C# برای مقداردهی اولیه ویژگی‌های شیء استفاده می‌شود و Destructor برای انجام عملیات پاکسازی قبل از حذف شیء (مانند پاکسازی منابع) استفاده می‌شود.