Persian + Sana :)

جمعه, ۲۷ تیر ۱۳۹۹ ۰۴:۳۲ ق.ظ

اصول SOLID :

حرف اول	مخفف	مفهوم
S	Single responsibility principle	**اصل تک‌وظیفه‌ای** یک کلاس باید تنها یک وظیفه داشته باشد (یک کلاس باید تنها یک دلیل برای تغییر داشته باشد و نه بیشتر)
O	Open–closed principle	**اصل باز. بسته** اجزای نرم‌افزار باید نسبت به توسعه باز (یعنی پذیرای توسعه باشد) و نسبت به اصلاح بسته باشند (یعنی پذیرای اصلاح نباشد). (یعنی مثلاً برای افزودن یک ویژگی جدید به نرم‌افزار نیاز نباشد که بعضی از قسمت‌های کد را بازنویسی کرد، بلکه بتوان آن ویژگی را مانند پلاگین به راحتی به نرم‌افزار افزود)
L	Liskov substitution principle	**اصل جانشینی لیسکو** اشیاءِ یک برنامه که از یک کلاس والد هستند، باید به راحتی و بدون نیاز به تغییر در برنامه، قابل جایگزینی با کلاس والد باشند.
I	Interface segregation principle	**اصل تجزیهٔ (تفکیک) رابط** استفاده از چند رابط که هر کدام، فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است.
D	Dependency inversion principle	**اصل وارونگی وابستگی** بهتر است که برنامه به تجرید(abstraction) وابسته باشد نه پیاده‌سازی

From <https://sadra.me/2018/solid>

اصل Single Responsibility

یک کلاس تنها باید یک دلیل برای تغییر داشته باشد.
یک وظیفه -> یک دلیل برای تغییر (میتواند باشد)
اگر یک کلاس بیش از یک وظیفه داشته باشد -> coupling اتفاق می افتد -> برای تغییر نیاز است که هر دو (یا هر چند وظیفه که در آن کلاس وجود دارد) تغییر کند.

فواید SRP
کد کوچک میشود.
کد خوانا تر است.
کد قابل فهم تر است
نگهداری کد آسان تر است
پتانسیل بیشتری برای نوشتن تست برای کد وجود دارد (تست کد آسان تر است)
مدیریت تغییرات کد آسان تر است.
جایگزینی کد آسان تر است

مثال ها : صفحه ی ۹۳ تا ۹۵

اصل Open/Closed Principle (OCP)

ایده اصلی : اجزای نرم افزار (مانند کلاس ها، ماژول ها، فانکشن ها و غیره) باید نسبت به توسعه باز و نسبت به تغییر/اصلاح بسته باشد.
باز برای گسترش: امکان افزودن رفتار جدید در آینده وجود داشته باشد.
بسته برای تغییر: نیازی به تغییر کدهای منبع یا باینری نباشد.

کلاس ها باید به گونه ای نوشته شده باشند که بتوانند توسعه پیدا کنند بدون آنکه نیاز به modification داشته باشند.

زمانی که implementation یک کلاس کامل شد تنها زمانی باید modify بشه که برای برطرف کردن یک باگ باشه.
فیچر های جدید یا تغییر در فیچر ها نیازمند ساخته شدن یک کلاس دیگر هستند.
مکانیزم ؟
Abstraction (انتزاع) و Polymorphism (چند ریختی) بودن، درون مایه اصل OCP هستند.
معرفی کلاس هایabstract base
Implementکردن interfaceهای متداول (common)
Deriveکردن (استخراج کردن | اشتقاق کردن) کلاس ها از common interfaceها و یا کلاس ها

مثال ها : صفحه‌ی ۹۸ تا ۱۰۲

فواید ocp :
طراحی پایدار تر
کدهای موجود و فعلی تغییر نمی کنند.

تغییرات با استفاده از «اضافه کردن» کد اتفاق می افتند و نه با modify کردن کد ها.

تغییرات ایزوله هستند و باقی کد رو تحت تاثیر و تغییر قرار نمی دهند.

پتانسیل تست پذیری کد بیشتر است.
مدیریت تغییرات آسان تر است
جایگزینی کد آسان تر است
طراحی قابل توسعه است
کد، احتمالا قابلیت دوباره استفاده شدن خواهد داشت
انتزاع کد، بیشتر است.

اصل Liskov Substitution Principle (LSP)

ایده اصلی: زیر کلاس ها باید بتوانند بدون هیچ تغییری با کلاس والد (پایه) جایگزین شوند.

این اصل میگوید که یک ماژول از برنامه از یک base class استفاده کند باید refrenceای که به base classدارد به راحتی بتواند با یک derived class جایگزین شود و مشکلی در functionality آن ماژول ایجاد نگردد.

کلاس مشتق شده (derived class) باید یک مدل رفتار خاص داشته باشد (باید سرویس های یکسانی با super classاش داشته باشد اما باید حداقل از آنها به شکلی متفاوت provideشده باشند.

Contract کلاس پایه باید در کلاس های مشتق شده مورد احترام باشد.

اگر این اصل نقض گردد، اصل ocp هم نقض میگردد (چرا ؟)

فواید LSP:
طراحی flexibleتر
یک کلاس پایه با یک کلاس مشتق میتواند جایگزین شود (به راحتی)
پتانسیل تست پذیری کد بیشتر است
نیاز است که اصل open/closed رعایت شود.

مثال ها : مثالی از آن در جزوه نبود.

اصل Interface Segregation Principle (ISP)

ایده اصلی : استفاده از چند رابط که هر کدام فقط یک وظیفه را بر عهده دارد بهتر از استفاده از یک رابط چند منظوره است.
این اصل به ما میگوید : حق نداریم client را (منظور کلاسی که از این تایپ یا کلاس والد استفاده میکند) مجبور کنیم تا به متدهایی وابستگی داشته باشد که حقیقتاً هیچ استفاده از آن ها نمیکند.
به جای اینترفیس های چاق از رابط های منسجم کوچک تک وظیفه ای استفاده کنید.
اگر کلاسی با استفاده های مختلف دارید برای هر کدام از استفاده ها یک اینترفیس محدود جداگانه درست کنید.

چرا از ISP استفاده کنیم؟
هدف اصلی این است که clientها را وادار کنیم که تا جای ممکن از اینترفیس های کوچک و منسجم استفاده کنند.
اینترفیس های چاق به coupling ناخواسته منجر خواهند شد و dependency های تصادفی بین کلاس ها ایجاد میکنند.

مثال ها: صفحه ی ۱۰۸ تا ۱۱۰

فواید ISP :
انسجام بیشتر
Clientها میتوانند تقاضای اینترفیس های منسجم تر داشته باشند.
طراحی پایدار تر
تغییرات ایزوله هستند و تمام کد را تحت تاثیر قرار نمی دهند.
اصل Liskov Substitution را حمایت می کند.

اصل Dependency Inversion Principle (DIP)

ایده اصلی : ماژول های سطح بالا نباید به ماژول های سطح پایین وابسته باشند. بلکه هردوی آنها باید به انتزاعات (کلاس abstractionیا اینترفیس) وابستگی داشته باشند. در واقع ابسترکشن نباید به جزییات وابستگی داشته باشد، بلکه جزییات (سطوح پایین تر برنامه) باید به ابسترکشن وابسته باشند.

کلاس های سطح بالا باید شامل business logic باشند و کلاس های سطح پایین باید detail های implementation فعلی رو داشته باشند.

مثال ها : صفحه ی ۱۱۳ تا ۱۱۵

فواید DIP :
طراحی طبق contract را ممکن میکند.
مدیریت تغییرات آسان تر است
جایگزینی کد آسان تر است
طراحی قابل توسعه است
کد پتانسیل بیشتری برای تست شدن دارد.

Design Practice

صفحات ۱۱۷ تا ۱۲۵ : خلاصه ای برای این قسمت نوشته نشده.

Design Patterns:

دیزان پترن چیست ؟

یک دیزان پترن، یک general reusable solution است برای یک مشکل که به طور معمول در طراحی نرم افزار پیش می آید.

یک دیزاین پترن یک طراحی تکمیل شده نیست که به کد تبدیل میشه.

یک دیزان پترن در واقع یک توضیح یا یک template است برای چگونگی حل یک مشکل که میتواند در موقعیت های مختلف مورد استفاده قرار گیرد.

دیزاین پترن های شی گرا در واقع نشان دهنده‌ی relationshipها و interaction های بین کلاس ها و objectها هستند بدون specify کردن final application classes یا objectهایی که درگیر هستند.

طبقه بندی:

Creational:

با مکانیزم object creation سروکار دارد.

Structural:

با شناسایی یک راه ساده برای پیدا کردن relationship بین موجودیت ها، طراحی را آسان می کند.

Behavioral :

Communication patternهای متداول بین objectها را شناسایی میکند (با هدف افزایش flexibility در carryکردن این communication

Architectural :

برای معماری نرم افزار استفاده می شوند.

Classification

Creational	Structural	Behavioural	Architecture
Factory method Abstract Factory Builder Lazy Loading Object pool Prototype Singleton Multiton Resource acquisition is initialization	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Null Object Null Object Command Interpreter Iterator Mediator Memento Observer State Chain of responsibility Strategy Specification Template method Visitor	Layers Presentation-abstraction-control Three-tier Pipeline Implicit invocation Blackboard system Peer-to-peer Model-View-Controller Service-oriented architecture Naked objects

4

Singleton Design Pattern :

Instantiation یک کلاس را به یک instance محدود میکند.

زمانی که فقط یک object برای هماهنگی در بین سیستم نیاز است، به کار می آید.

ایده اصلی : خود کلاس رو مسئول کنترل کردن instantiation کردن.

Singleton

```
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

- **Factory Method :**
  - هدف

پیاده‌سازی مفهوم کارخانه

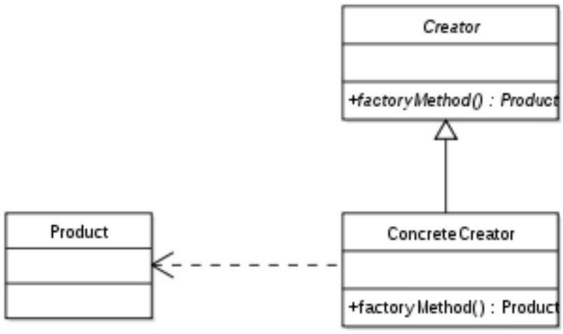
intertfaceای برای ساختن یک شیء تعریف می‌کند اما client تصمیم می‌گیرد که کدام کلاس سازنده‌است

به کلاس اجازه‌ی موکول کردن سازنده به client را می‌دهد

  - کاربرد

زمانی که Frameworkای نیاز دارد مدل معماری را برای بازه‌ای از applicationها، استاندارد کند اما به applicationها این اجازه را بدهد که اشیاء دامنه‌ی خود را تعریف کرده و سازنده‌یشان را فراهم کنند

- ساختار



مثال ها : صفحه ی ۱۳۴ و ۱۳۵

Factory Method

Creational DP

```
public interface ImageReader {
   public DecodedImage getDecodedImage();
}

public class GifReader implements ImageReader {
   public DecodedImage getDecodedImage() {
      // ...
      return decodedImage;
   }
}

public class JpegReader implements ImageReader
      {
   public DecodedImage getDecodedImage() {
      // ...
      return decodedImage;
   }
}
```

```
public class ImageReaderFactory {
   public static ImageReader
            getImageReader(InputStream is) {
      int imageType = determineImageType(is);
      switch (imageType) {
         case ImageReaderFactory.GIF:
            return new GifReader(is);
         case ImageReaderFactory.JPEG:
            return new JpegReader(is);
         // etc.
      }
   }
}
```

Factory Method

Creational DP

```
//Empty vocabulary of actual object
public interface IPerson
{
    string GetName();
}

public class Villager : IPerson
{
    public string GetName()
    {
        return "Village Person";
    }
}

public class CityPerson : IPerson
{
    public string GetName()
    {
        return "City Person";
    }
}

public enum PersonType
{
    Rural,
    Urban
}
```

```
/// <summary>
/// Implementation of Factory - Used to create objects
/// </summary>
public class Factory
{
    public IPerson GetPerson(PersonType type)
    {
        switch (type)
        {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
            default:
                throw new NotSupportedException();
        }
    }
}
```