**SOLID PRINCIPLES PRESENTATION TEXT**

**Slide 1:**

Hello every one. My name is Vladislav , and I’ll introduce the presentation about so called SOLID principles.

**Slide 2:**

SOLID is five design principles intended to make software designs more understandable, flexible, and maintainable. It helps us to think about the right way to build a software system

SOLID is a mnemonic acronym that refers to a set of five principles at the base of a good software design:

* Single Responsibility Principle
* Open/Closed Principle
* Liskov Substitution Principle
* Interface Segregation Principle
* Dependency Inversion Principle

**Slide 3-1:**

So, let start from the first one. S – ***Single Responsibility Principle.***

*The single responsibility principle says that each of our classes has to be only used for one purpose.*

*We need this so that we don’t have to change code as often when something changes. It’s also hard to understand what the class is doing if it’s doing many things.*

*Unrelated concepts in the same class also make comprehending the purpose of the code harder.*

**Slide 3-2:**

For example, we can write something like the following to follow the single responsibility principle:

class Rectangle {  
 constructor(length, width) {  
 this.length = length;  
 this.width = width;  
 } get area() {  
 return this.length \* this.width;  
 }  
}

The Rectangle class above only has the length and width of a rectangle as members and lets us get the area from it.

It does nothing else, so it follows the single responsibility principle.

**---------**

A bad example would be:

class Rectangle {  
 constructor(length, width) {  
 this.length = length;  
 this.width = width;  
 } get area() {  
 return this.length \* this.width;  
 }

}

We should have a createCircle method in a Rectangle class since they’re unrelated concepts.

**Slide 4-1:**

The second SOLID principle concerns the extensibility of components and is called the ***Open/Closed Principle***. Its focus is on avoiding changes when we need to extend a component's feature. The principle states:

*Software entities like classes, modules and functions should be open for extension but closed for modifications.*

In the design of the components of our application, we have to take into account these two aspects:

* **Open for extension**: The components should be adjustable to the changing needs of the application
* **Closed for modifications**: The required changes should not involve the original component itself

If we apply this principle, we can get more easily adaptable and maintainable applications.

**Slide 4-2:**

To illustrate how to apply this principle, let's take the example class *Rectangle*

class Rectangle {  
 constructor(length, width) {  
 this.length = length;  
 this.width = width;  
 } get area() {  
 return this.length \* this.width;  
 }  
}

**Slide 4-2:**

Then if we want to add a function to for calculating its perimeter, we can do it by adding a method to do it as follows:

class Rectangle {  
 constructor(length, width) {  
 this.length = length;  
 this.width = width;  
 } get area() {  
 return this.length \* this.width;  
 } get perimteter() {  
 return 2 \* (this.length + this.width);  
 }  
}

As we can see, we didn’t have to change existing code to add it, which satisfies the open/closed principle.

**Slide 5-1:**

The third SOLID principle, the Liskov Substitute Principle, is somehow an extension of the Open/Closed Principle.

The principle says: if we have a parent class and a child class, then we can interchange the parent and child class without getting incorrect results.

This means that the child class must implement everything that’s in the parent class. The parent class serves the class has the base members that child classes extend from.

For example, if we want to implement classes for a bunch of shapes, we can have a parent Shape class, which are extended by all classes by implementing everything in the *Shape* class.

**Slide 5-2:**

We can write the following to implement some shape classes and get the area of each instance:

class Shape {  
 get area() {  
 return 0;  
 }  
}

class Rectangle extends Shape {  
 constructor(length, width) {  
 super();  
 this.length = length;  
 this.width = width;  
 } get area() {  
 return this.length \* this.width;  
 }  
}

class Square extends Shape {  
 constructor(length) {  
 super();  
 this.length = length;  
 } get area() {  
 return this.length \*\* 2;  
 }  
}

class Circle extends Shape {  
 constructor(radius) {  
 super();  
 this.radius = radius;  
 } get area() {  
 return Math.PI \* (this.radius \*\* 2);  
 }  
}

const shapes = [  
 new Rectangle(1, 2),  
 new Square(1, 2),  
 new Circle(2),  
]

for (let s of shapes) {  
 console.log(s.area);  
}

Since we override the area getter in each class that extends Shape , we get the right area for each shape since the correct code is run for each shape to get the area.

**Slide 6-1:**

I – Interface Segregation principle. Official version says: Clients should not be forced to depend on methods they do not use. In other words, the interface should provide minimal funcitionality, and we shouldn’t impose the implementation of something if it’s not needed.

Although JavaScript does not support interfaces as abstract types. In any case, this principle does not refer to the interfaces as a pure syntactic element, but to the whole set of public properties and methods of an object.

~~This is probably the simplest Principle to understand. It simply states, you shouldn't force a class to implement a method it has no use for. In other words, when creating your interfaces, make sure the interfaces are like classes, and have a single responsibility.~~

**Slide 6-2:**

Lets jump to the example:

We have class *Animal* :

class Animal {  
 constructor(name){  
 this.name = name;  
 }

walk(){

console.log(`${this.name} can walk`)

}

swim(){

console.log(`${this.name} can swim`)

}

fly(){

console.log(`${this.name} can fly`)

}

}

**Slide 6-2:**

class Dog extends Animal {

}

class Eagle extends Animal {

}

class Whale extends Animal {

}

const dog = new Dog(“Ralph”);

dog.walk(); // Ralph can walk

dog.swim(); // Ralph can swim

dog.fly(); // incorrect Ralph can fly

const eagle = new Eagle(“Kenny”);

eagle.walk(); // Kenny can walk

eagle.swim(); // incorrect Kenny can swim

eagle.fly(); // Kenny can fly

const whale = new Whale(“Jack”);

whale.walk(); // incorrect Jack can walk

whale.swim(); // Jack can swim

whale.fly(); // incorrect Jack can fly

**Slide 6-3:**

To fix incorrect examples we can overload necessary methods:

class Dog extends Animal {

fly(){

return null;

}

}

class Eagle extends Animal {

swim(){

return null;

}

}

class Whale extends Animal {

fly(){

return null;

}

walk(){

return null;

}

}

All is fine, but we can see, that our class Animal has too common things, because of this, we have to cancel unnecessary methods. We will implement IS principle in other way, that will be correct

**Slide 6-4:**

We have the same base class *Animal*:

class Animal {  
 constructor(name){  
 this.name = name;  
 }

}

**Slide 6-5:**

Then we create some simple objects with similar construction:

const swimmer = {

swim(){

console.log(`${this.name} can swim`)

}

}

const flier = {

fly(){

console.log(`${this.name} can fly`)

}

}

const walker = {

walk(){

console.log(`${this.name} can walk`)

}

}

So, we have objects, that add specific behavior.

Of course we have specific classes as at the previous example.

class Dog extends Animal {

}

class Eagle extends Animal {

}

class Whale extends Animal {

}

And now the magic. In Javascript we can use method Object.assign() to implement the IS principle.

Object.assign(Dog.prototype,swimmer,walker);

Object.assign(Eagle.prototype,walker,flier);

Object.assign(Whale.prototype,swimmer);

**Slide 6-6:**

And in the end we have objects with correct methods

const dog = new Dog(“Ralph”);

dog.walk(); // Ralph can walk

dog.swim(); // Ralph can swim

const eagle = new Eagle(“Kenny”);

eagle.walk(); // Kenny can walk

eagle.fly(); // Kenny can fly

const whale = new Whale(“Jack”);

whale.swim(); // Jack can swim

**Slide 7-1:**

The last one is D – Dependency Inversion Principle. The bases of principle:

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.
* DIP is about binding classes behind the interfaces consumed by client code.
* DIP states that classes that implement interfaces are not visible to the client code.

**Slide 7-2:**

It could be difficult to understand for listening, but lets jump to the example, and I guess, it will be clear.

We have class *Database*, that helps to manipulate with database

class Fetch{

request(url){

return fetch(url).then(res=>res.json());

}

}

class Database {

constructor(){

this.fetch = new Fetch();

}

getData(){

return this.fetch.request(“example.com”);

}

}

const db = new Database();

db.getData(); // data from database

**Slide 7-2:**

Then our customer decides to change our storage to localstorage. Now we should write again new API to get data from localstorage.

class LocalStorage{

get(key){

return “data from local storage”;

}

}

And now we should change the code in our *Database* class.

class Database {

constructor(){

//this.fetch = new Fetch();

this.localStorage = new LocalStorage();

}

getData(){

//return this.fetch.request(“example.com”);

return this.localStorage.get(“key”);

}

}

const db = new Database();

db.getData(); // data from local storage

It works, but it turns, that we should change our Database class every time, because it depends on concrete implementation of data storage. It is not very comfortable, and it is difficult to maintain.

**Slide 7-2:**

Therefore, for this case we can implement our Dependency Inversion Principle.

We need to write an interface to manipulate with our entities.

First, we create a wrapper for each storage entity.

class FetchData {

constructor(){

this.fetch = new Fetch();

}

getClientData(key){

return this.fetch.request(key)

}

}

class LocalStorageData {

constructor(){

this.localstorage = new LocalStorage();

}

getClientData(key){

return this.localstorage.get(key)

}

}

**Slide 7-2:**

We have the same methods name, but with different implementations.

And now, we need to rewrite Database class to correct way.

class Database {

constructor(data){

//this.fetch = new Fetch();

//this.localStorage = new LocalStorage();

this.data = data;

}

getData(key){

//return this.fetch.request(“example.com”);

//return this.localStorage.get(“key”);

return this.data.getClientData(key);

}

}

const db = new Database(new FetchData());

db.getData(); // data from database

const db1 = new Database(new LocalStorageData());

db1.getData(); // data from localstorage

We can see, that we don’t need to change storage implementation, but just putting another class to the constructor. And in the end, we have that our database not depends on low level abstractions, but depends on that concrete interface that we put. We have changed our dependencies order and DIP has implemented.

**Slide 8:**

**Thanks for your attention, and I’m waiting for your questions if they are ☺**