Prof. Williams

Andy Nguyen

Homework: **Explain OOP principles**

**Due: 2/21/2021**

**I) What is Single Responsibility principle?**

A single Responsibility principle is one of the parts of five object-oriented programming design of (S.O.L.I.D). A single Responsibility principle is nothing, but it is a computer-programming principle that states every module or class should have responsibility over a single part of the functionality provided by the software. and that responsibility should be entirely encapsulated by the class, module, or function.

this is simply 'A class should have one, and only one, reason to change'.

By the example, we will know about how important the Single Responsibility principle is :

* **“Every function you write should do exactly one thing. It should have one clearly defined goal.”**
* Code:

**//Below we use the function name Different() to calculate the different between 2 values 1 and 2**

test('Calculator difference between two numbers', () => {

//assigning the results of the calc different method to an object, calculator returns an object

let result = Calculator.Difference(1,2);

//expect is used to test if the results are correct

expect(result).toBe(-1);

**//Below we use the function name Product() to calculate the product for 2 values 1 and 2**

test('Calculator product of two numbers', () => {

//assigning the results of the calc product method to an object, calculator returns an object

let result = Calculator.Product(1,2);

//expect is used to test if the results are correct

expect(result).toBe(2);

});

**Benefits of following the Single Responsibility principle :-** easier to understand the class or module:-

When the class only does "one thing", its interface usually has a small number of methods that are self-explanatory. It should also have a small number of member variables (less than seven-is).

more reusable the class or module :

* If a class has multiple responsibilities, and only one of those is needed in another area of the software, then the other unnecessary responsibilities hinder reusability. Having a single responsibility means the class should be reusable without modification.

**II) What is Open-Closed Principle :** JavaScript Modules should be open for extension but closed for modification.

Tutorial:

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification".

**Explanation**: It tells you to write your own code so that you will be able to add new functionality(code) without changing the existing code. It prevents the situations in which a change to one of your code in the classes also requires you to adapt all depending on code in the classes.

**Example:** The best way to implement this by using the Interfaces.

* In this Class Diagram no changes required when a new shape is added (Follow the rules of open closed principle properly)

//getresults is a method so that it can return the results of the calculation

GetResults() {

return this.op(this.a,this.b)

}

**Add(values)**

**Operations. Push(values)**

// where I can the values can be added without using manually input.

**III) The Liskov Substitution Principle**

" Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T."

**Explanation:** The principle defines that objects of the super-class shall be replaceable with objects of the sub-classes without breaking the application. It requires the objects of your sub-classes to behave in the same way as the objects of your super- class. we need to follow two rules.

1. Method Override (Sub-class Input Parameter values accepted by the Super-class parameters)

2. Return value of the method (return value of a method of the sub-class needs to comply with the same rules as the return value of the method of the super-class.)

**Example:** This can be explained by using the concept of the Inheritance(Parent-Child Relationship). Generally, we can use this get the property values from parent to child. Override will happen between super-class and sub-class.

**Some code to make this clear:**

class Calculation {

//the constructor is the first method called after instantiation and usually sets the properties of the object

constructor(a, b, op) {

//"this" is the internal reference of the object to access its internal methods and properities

this.a = a;

this.b = b;

this.op = op;

}

//getresults is a method so that it can return back the results of the calculation

GetResults() {

return this.op(this.a,this.b)

}

}

module.exports = Calculation;

**IV) Interface Segregation Principle**

The interface segregation principle states that no client should be forced to depend on methods that it does not use. This way of working splits large interfaces that aren’t very succinct, into much smaller, manageable interfaces. This approach aims to segregate logic into a decoupled approach that allows for ease of refactoring in the future. Take the following snippet for example. It strictly defines the methods that are available but takes no care for the logic that is contained within that method. This approach allows developers to write succinct and easy to follow code solutions.

For example:

function resetCalculator() {

calculator.displayValue = '0';

calculator.firstOperand = null;

calculator.waitingForSecondOperand = false;

calculator.operator = null;

console.log(calculator);

}

**V)The Dependency Inversion Principle**

The High-level modules, which contains a complex logic, should be easily reusable and not affected by changes in low-level modules, it provides the utility features. to achieve that , the experts introduce an abstraction that decouples the high-level and low-level modules from each other.

Based on this idea, the Dependency Inversion Principle consists of two parts:

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 abstraction Explanation: That high-level & low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So, in the end, you get two dependencies 1. The high-level module depends on the abstraction, and 2. The low-level depends on the same abstraction.

**Example:** In order to achieve all the above principles, we must and should familiar with the concepts of the Interface, Inheritance, Abstraction mechanisms.

function handleOperator(nextOperator) {

const { firstOperand, displayValue, operator } = calculator

const inputValue = parseFloat(displayValue);

if (operator && calculator.waitingForSecondOperand) {

calculator.operator = nextOperator;

console.log(calculator);

return;

}

if (firstOperand == null && !isNaN(inputValue)) {

calculator.firstOperand = inputValue;

} else if (operator) {

const result = calculate(firstOperand, inputValue, operator);

calculator.displayValue = String(result);

calculator.firstOperand = result;

}

calculator.waitingForSecondOperand = true;

calculator.operator = nextOperator;

console.log(calculator);

}