



IBEROAMERICANA COORPORACIÓN UNIVERSITARIA

FACULTAD DE INGENIERIA

FUNDAMENTOS DE DISEÑO

ACTIVIDAD 2 - IMPLEMENTACIÓN DE PRINCIPIOS SOLID
Y GRASP

NESTOR OSORIO CASTIBLANCO

TUTOR: MARY RUBIANO

NOVIEMBRE 12 DE 2021

1.Implementación de principios SOLID

Cuando se combinan, facilitan que un programador desarrolle software que sea fácil de mantener, entender y sea menos costoso en el momento de crear un nuevo módulo en su aplicativo. todos sabemos que la POO (Programación Orientada a Objetos) nos permite agrupar entidades con funcionalidades parecidas o relacionadas entre sí, pero esto no implica que los programas no se vuelvan confusos o difíciles de mantener. Del desarrollo ágil de software.

S: Single responsibility principle o Principio de responsabilidad única

O: Open/closed principle o Principio de abierto/cerrado

L: Liskov substitution principle o Principio de sustitución de Liskov

I: Interface segregation principle o Principio de segregación de la interfaz

D: Dependency inversion principle o Principio de inversión de dependencia

2. Indique la relación de estos principios con el diseño de la arquitectura de un sistema y las consecuencias de su uso o la omisión de ellos.

En el ámbito del software cada vez es más común escuchar el término “arquitectura de software”, y encontrar oportunidades de empleo para “arquitectos de software”. Aun así, este concepto tiende a ser malentendido y la falta de comprensión al respecto de sus principios frecuentemente repercute de manera negativa en la construcción de sistemas de software

Link repositorio GITHUB (<https://github.com/nosorio95/Nestor-Osorio-Castiblanco.git>)

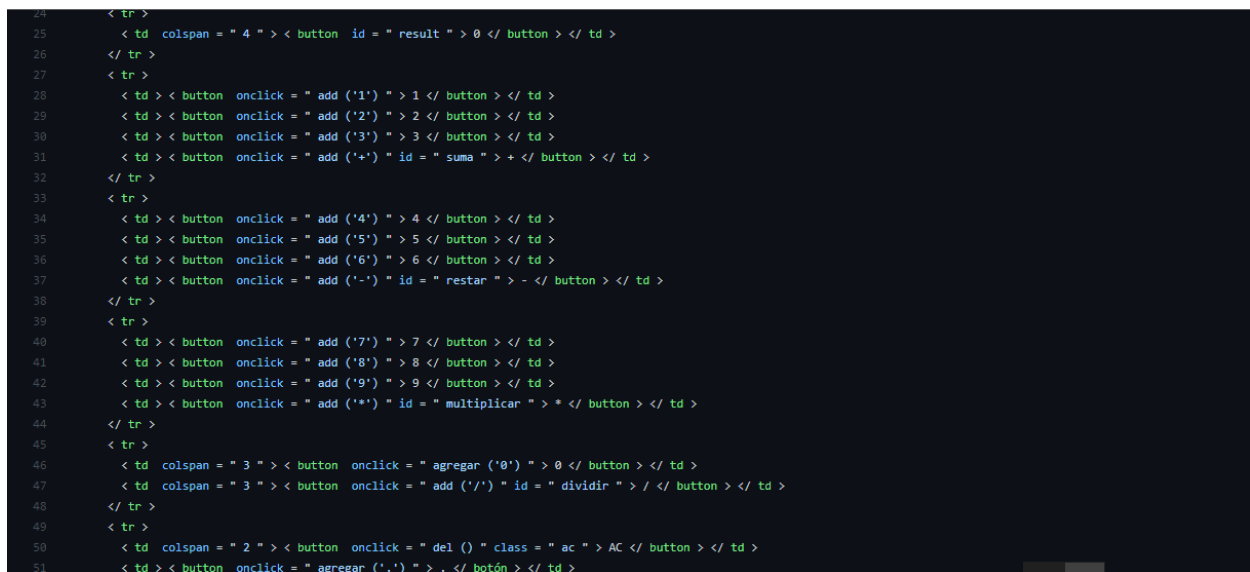
Pantallazos:

Se crea calculadora sencilla, se evidencia implementación con la plataforma GITHUB



The screenshot shows the GitHub web interface for a file named 'index.html' in the repository 'Nestor-Osorio-Castiblanco'. The file was created by user 'nosorio95' and has 111 lines of code (100 SLOC) with a size of 3,04 KB. The code is a simple HTML document for a calculator. It includes a meta viewport tag, a jQuery script, and a table with buttons for digits 0-9, arithmetic operators (+, -, *, /), and a 'result' button. The code is displayed in a dark-themed editor with line numbers on the left.

```
1 <!doctype html >
2 < html lang = " en " >
3   < cabeza >
4     < meta charset = " utf-8 " >
5     < meta name = " viewport " content = " viewport-fit = cover, width = device-width, initial-scale = 1.0, minimum-scale = 1.0, maximum-scale = 1.0, user-scalable = no " >
6
7   < Guión src = " https://code.jquery.com/jquery-3.5.1.js " integridad = " sha256-Qmo7LDvxbWT2tbbQ97853yJnYU3MhH / C8ycbRAKjPDc = " crossorigin = " Anónimo " > </ script de >
8   < titulo > calculadora! </ titulo >
9   </ cabeza >
10  < estilo >
11    mesa {
12      ancho : 100 % ;
13    }
14    botón {
```



This block continues the HTML code from the previous screenshot, showing the table structure for the calculator buttons. The code is displayed in a dark-themed editor with line numbers on the left.

```
24 < tr >
25   < td colspan = " 4 " > < button id = " result " > 0 </ button > </ td >
26 </ tr >
27 < tr >
28   < td > < button onclick = " add ('1') " > 1 </ button > </ td >
29   < td > < button onclick = " add ('2') " > 2 </ button > </ td >
30   < td > < button onclick = " add ('3') " > 3 </ button > </ td >
31   < td > < button onclick = " add ('+') " id = " suma " > + </ button > </ td >
32 </ tr >
33 < tr >
34   < td > < button onclick = " add ('4') " > 4 </ button > </ td >
35   < td > < button onclick = " add ('5') " > 5 </ button > </ td >
36   < td > < button onclick = " add ('6') " > 6 </ button > </ td >
37   < td > < button onclick = " add ('-') " id = " restar " > - </ button > </ td >
38 </ tr >
39 < tr >
40   < td > < button onclick = " add ('7') " > 7 </ button > </ td >
41   < td > < button onclick = " add ('8') " > 8 </ button > </ td >
42   < td > < button onclick = " add ('9') " > 9 </ button > </ td >
43   < td > < button onclick = " add ('*') " id = " multiplicar " > * </ button > </ td >
44 </ tr >
45 < tr >
46   < td colspan = " 3 " > < button onclick = " agregar ('0') " > 0 </ button > </ td >
47   < td colspan = " 3 " > < button onclick = " add ('/') " id = " dividir " > / </ button > </ td >
48 </ tr >
49 < tr >
50   < td colspan = " 2 " > < button onclick = " del () " class = " ac " > AC </ button > </ td >
51   < td > < button onclick = " agregar ('.') " > . </ botón > </ td >
```

principal ▾

Nestor-Osorio-Castiblanco / Controlador.php / <> Salta a ▾

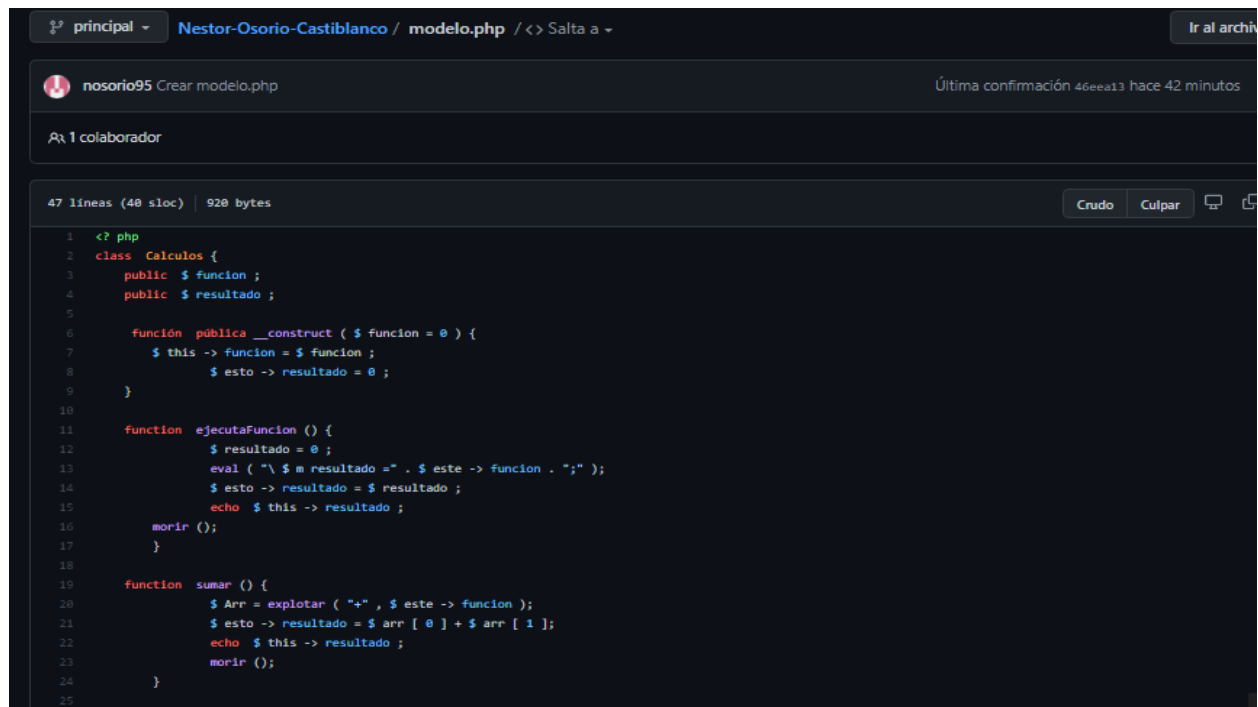


nosorio95 Cambiar el nombre de Controlador a Controlador.php

1 colaborador

20 líneas (15 sloc) | 411 bytes

```
1  <? php
2      require_once ( "modelo.php" );
3      $ Calculos = new Calculos ( $_POST [ 'resultado' ] );
4
5          if ( strpos ( $_POST [ 'resultado' ], '+' ) ) {
6              morir ( $ Calculos -> sumar ());
7          }
8
9          if ( strpos ( $_POST [ 'resultado' ], '-' ) ) {
10             morir ( $ Calculos -> restar ());
11         }
12
13         if ( strpos ( $_POST [ 'resultado' ], '/' ) ) {
14             morir ( $ Calculos -> dividir ());
15         }
16
17         if ( strpos ( $_POST [ 'resultado' ], '*' ) ) {
18             morir ( $ Calculos -> multiplicar ());
19         }
```



The screenshot shows a code editor interface with a dark theme. At the top, there's a header bar with a user profile icon, the name 'Nestor-Osorio-Castiblanco', the file name 'modelo.php', and a 'Salta a' dropdown. Below this, a sub-header shows the user 'nosorio95' and the action 'Crear modelo.php', along with a confirmation message 'Última confirmación 46ee13 hace 42 minutos'. The main area displays PHP code for a class named 'Calculos'. The code includes a constructor, a method 'ejecutaFuncion()', and a method 'sumar()'. The code is syntax-highlighted and includes line numbers from 1 to 25. On the right side of the code editor, there are buttons for 'Crudo', 'Culpar', and icons for a terminal and a copy function.

```
1  <? php
2  class Calculos {
3      public $ funcion ;
4      public $ resultado ;
5
6      función pública __construct ( $ funcion = 0 ) {
7          $ this -> funcion = $ funcion ;
8          $ esto -> resultado = 0 ;
9      }
10
11     function ejecutaFuncion () {
12         $ resultado = 0 ;
13         eval ( "\ $ m resultado =" . $ este -> funcion . ";" );
14         $ esto -> resultado = $ resultado ;
15         echo $ this -> resultado ;
16         morir ();
17     }
18
19     function sumar () {
20         $ Arr = explotar ( "+" , $ este -> funcion );
21         $ esto -> resultado = $ arr [ 0 ] + $ arr [ 1 ];
22         echo $ this -> resultado ;
23         morir ();
24     }
25
```

1. Principio de responsabilidad única

Como puedes observar, este principio dice que un objeto/clase debería únicamente tener una responsabilidad completamente encapsulada por la clase. Aquí, cuando hablamos de responsabilidad, nos referimos a una razón para cambiar. Este principio nos dirige hacia una cohesión más fuerte en la clase y un encaje más flojo entre la dependencia de clases, una mayor facilidad de lectura y un código con una complejidad menor.

2. Principio abierto-cerrado

Aquí la idea es que una entidad permite que comportamiento se extienda, pero nunca modificando el código de fuente. Cualquier clase (o cualquier cosa que escribas) debe de estar escrito de una manera que puede utilizarse por lo que es. Puede ser extensible, si se necesita, pero nunca modificado.

3. Principio de sustitución de Liskov

Como su nombre indica, este principio fue definido por Barbara Liskov. La idea aquí es que los objetos deberían ser reemplazados por ejemplos de su subtipo, y ello sin que la funcionalidad del

sistema desde el punto de vista de los clientes se vea afectada. Básicamente, en vez de utilizar la implementación actual, deberías ser capaz de utilizar una clase base y obtener el resultado esperado.

4. Principio de segregación de la interfaz

Se basan en cómo escribir interfaces. Entonces, ¿qué significa? Básicamente, una vez la interfaz se convierte en larga, se necesita absolutamente de separarla en pequeñas partes más específicas. Una interfaz será definida por el cliente que lo utilice, lo que significa que el será el único que tenga conocimientos de los métodos relacionados con ellos.

5. Principio de inversión de la dependencia

Este principio está principalmente basado en reducir las dependencias entre los módulos del código. Básicamente, este principio será de gran ayuda para entender cómo atar correctamente sistemas juntos. Si los detalles de tu implementación dependen de los altos niveles de abstracciones, te ayudará a conseguir un sistema bien acoplado. También, influirá en la encapsulación y cohesión de ese sistema.

Implementación de principios GRASP

En diseño orientado a objetos, GRASP son patrones generales de software para asignación de responsabilidades, es el acrónimo de "GRASP (object-oriented design General Responsibility Assignment Software Patterns)". Aunque se considera que más que patrones propiamente dichos, son una serie de "buenas prácticas" de aplicación recomendable en el diseño de software.

1.Creador

El patrón creador nos ayuda a identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases.

2.Controlador

El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado.

3.Alta cohesión y bajo acoplamiento

Nos dice que la información que almacena una clase debe ser coherente y debe estar (en la medida de lo posible) relacionada con la clase.

4.Bajo acoplamiento

Es la idea de tener las clases lo menos ligadas entre sí que se pueda. De tal forma que, en caso de producirse una modificación en alguna de ellas, se tenga la mínima repercusión posible en el resto de clases, potenciando la reutilización, y disminuyendo la dependencia entre las clases

5.Polimorfismo

El uso del polimorfismo, cuando las alternativas o comportamientos relacionados varían según el tipo (clase), asigne la responsabilidad para el comportamiento- utilizando operaciones polimórficas- a los tipos para los que varía el comportamiento. Asigna el mismo nombre a servicios en diferentes objetos.

6.Variaciones Protegidas

Es el principio fundamental de protegerse del cambio, de tal forma que todo lo que preveamos en un análisis previo que es susceptible de modificaciones, lo envolvamos en una interfaz, utilizando el polimorfismo para crear varias implementaciones y posibilitar implementaciones futuras, de manera que quede lo menos ligado posible a nuestro sistema, de forma que cuando se produzca la variación, nos repercuta lo mínimo. Forma parte de los patrones Grasp avanzados.

Conclusión:

- Cuando desarrollas cualquier software, hay dos conceptos muy importantes: cohesión (cuando dos partes distintas de un sistema trabajan juntas para tener mejor resultado que ambas partes trabajando de manera individual) y acoplamiento (puede verse como un grado de dependencia en una clase, método u otra entidad de software)
- El acople se presenta normalmente en muchos códigos y como he mencionado anteriormente, la situación más óptima sería tener un bajo acoplamiento y alta cohesión. Con esta pequeña introducción a los 5 principios SOLID, has podido ver que nos ayudan a lograrlo.

