احنا عارفين ان oop يعني مجموعه من ال classes واتكلمنا عن مبادئ ال oop واللي ممكن اطبقها ع ال classes ده عشان تساعدني في حاجات كتير

نتكلم بقا ع التصميم اللي بنشوف بيها ال classes ده

ال classes design بيعمل تصميم معين لل classes ده يعني بحدد مجموعه من القواعد امشي عليها في ال class ده وماينفعش اخرج عن اي قاعده زي ال architecture pattern و ال design pattern كده قولنا انهم بيدوني design او solution ل problem معينه نفس الشئ بالظبط في ال classes design بيحددلي structure معين للكلاس امشي عليه وبيكون design بينفذلي حاجه معينه عشان احسن جوده البرنامج وده يعتبر مبدأ مهم جداً في ال oop واللي هيا ال SOLID كل حرف من الكلمه ده اختصار ل design معين

S : single personality principle

O : open-closed principle

L : liskov substitution principle

I : interface segregation principle

D : dependency inversion principle

ده مجموعة من المبادئ اللي لازم اوفرها في اي كود وهدفها الأساسي انها بتقلل الترابط بين ال classes وبيقلل حدوث ال coupling بينهم (يعني بقلل فكرة ان لو عملت اي تعديل في اي كلاس مكنش مضطرة اعدل في classes ال coupling بعدين ان شاء الله) وده طبعاً بيخلي الكود بتاعي reusability و flexibility \

principle في ال S'olid' وهو ال

single responsibility principle (SSP)

يعني كل class يكون مسؤل عن وظيفه واحده بس

مثلاً عايزه اعمل عمليه بيع يبقا الكلاس ده كل الفانكشن اللي فيه هتكون بتخص عمليه البيع بس ماينفعش اجي في النص واعمل فانكشن بتحسبلي الفاتورة

وبكده لمه أطبق ال single responsibility principle اكون زودت ال cohesion لان كلاس واحد هيكون جامع كل الفانكشن وكل الداتا اللي بتخص موضوع واحد وبكده اكون عرفت اربط بينهم

وهكون قللت ال coupling لان كل الفانكشن اللي هيحتاجها الكلاس ده هتكون موجوده فيه ف انا مش مضطر اكون معتمد ع اي كلاس تاني في اي حاجه

(ال coupling && cohesion) ولمه اقلل ال coupling و ازود ال cohesion ف كده الكود بتاعي في افضل حاله بقا سهل اقراءه وافهمه واعرف اعدل عليه بكل سهوله من غير ما اتهوه في الكو

احيانا واحنا بنكتب كود بيكون في فانكشن او كلاس ماينفعش اعدل عليها او اعمل اي تعديل في هيكل ال class مثلا عندي class اسمه square فيه فانكشن بتحسب مساحه المربع ده حاجه علميه خلاص مقدرش اعدل في ال implementation وقولوا خليلي المساحه بتساوي width\*4 بدل width\*width وهنا انا بتكلم عن تاني principle في ال s'O'lid وهو

open-closed principle (OCP)

واللي بيتلخص بكل بساطة في الجمله ده

The class must be closed to modification and open to extension

معناه اني اقدر اضيف اي فانكش جديد ل اي class او اضيف اي attribute بس مقدرش اني اغير او اعدل في اي حاجه موجوده يعني مثلاً المثال اللي فوق ده عايز اضيف فانكشن جديده تحسبلي المحيط ازاي اضمن فعلاً ان اضيف الفانكشن ده بدون ماتغيرلي اي حاجه في الكود الحالي اللي بيساعدني علي اني اعمل كده ال abstract and interface class (شرحتهم قبل كده ) يعني كل ماكون عايزه اضيف اي فانكش جديده هكتبه في ال abstract class وهخلي الفانكشن الاساسيه تورث منه او اكتبها في ال interface class وهخلي الفانكشن الاساسيه ت implement منها وكده اكون قدرت اجبر الكلاس الاساسي انه يعمل implementation لفانكشن معينه وخليته يمشي بتصرف معين انا عايزه وانا اللي محدده بدون مااثر علي الكود الحالي

عندنا مثلاً اخر مثال ال operating system زي windows او Linux اقدر اضيف اي برامج جديده للنظام ده بدون مااعدل في ال operating system نفسه وبس كده

Befor:

class Shape {

public function calculateArea() {

// calculate area of shape

}

}

class Circle extends Shape {

public function calculateArea() {

// calculate area of circle

}

}

class Rectangle extends Shape {

public function calculateArea() {

// calculate area of rectangle

}

}

After:

interface Shape {

public function calculateArea();

}

class Circle implements Shape {

public function calculateArea() {

// calculate area of circle

}

}

class Rectangle implements Shape {

public function calculateArea() {

// calculate area of rectangle

}

}

class AreaCalculator {

public function calculate(Shape $shape) {

return $shape->calculateArea();

}

}

تالت مبدأ في ال so\*L\*id وهو ال liskov substitution principle (LSP) ده جي من اسم العالمه اللي عملت ال principle ده اسمها Barbara liskov واللي جمعت بعض المبادئ في ال principle ده زي ال polymorphism وال abstraction بيقول ال principle ده ان اي child يقدر يحل محل ال parent ويعمل اللي ال parent يعمله بس ده اكيد بشروط لازم ال child يكون substitution من ال parent يعني اي يكون substitution يعني كل الفانكشن اللي في ال parent لازم تكون موجودة في ال child وبتعمل نفس التصرف عشان ال child يقدر يحل محل ال parent وبكده اكون طبقت مبدأ ال abstraction واكون أجبرت الكلاسين يعملوا نفس الفانكشن او ان اعمل overriding لفانكشن موجوده في ال parent عند ال child نضرب مثال عشان نفهم نفرض مثلاً عندنا كلاس اسمه DB فيه فانكشن اسمها adddata بضيف البيانات للداتا بيز وفي class اسمه addpost وكلاس اسمه addphoto الاتنين هيعملوا inheritance من DB عشان يعملوا adddata وطبعاً كل كلاس هيعمل overriding للفانكشن اللي اسمها adddata

DB parent=new DB();

كده انا عملت object من ال parent ولمه احتاج اي حد من ال child هخليه يحل محل ال parent وخلاص مثلا عايزه اضيف صورة انادي علي كلاس addphoto واخزنه في ال parent

Parent=new addphoto ()

ولمه ااحتاج الفانكشن اللي اسمها adddata في كلاس addphoto هنادي عليها بااستخدام ال parent ولو عايزه الفانكشن اللي في addpost هقول ال parent=new addpost وهنادي ع الفانكشن وهكذا بقا لو عندي كذا كلاس بيورثو من ال parent

وكده مش هيكون عندي غير object واحد اسمه parent هو اللي هستخدمه في الكود كله

من الاخر بعمل object واحد من class parent وبتعامل معاه كانه variable وكل مااحتاج حاجه من عند ال child هخلي ال variable اللي عملته ده يساوي ال child ده

Befor:

class Rectangle {

protected $width;

protected $height;

public function setWidth($width) {

$this->width = $width;

}

public function setHeight($height) {

$this->height = $height;

}

public function getArea() {

return $this->width \* $this->height;

}

}

class Square extends Rectangle {

public function setWidth($width) {

$this->width = $width;

$this->height = $width;

}

public function setHeight($height) {

$this->width = $height;

$this->height = $height;

}

}

function calculateArea(Rectangle $shape) {

return $shape->getArea();

}

$rectangle = new Rectangle();

$rectangle->setWidth(5);

$rectangle->setHeight(10);

echo "Rectangle Area: " . calculateArea($rectangle) . "\n";

$square = new Square();

$square->setWidth(5);

echo "Square Area: " . calculateArea($square) . "\n";

After:

interface Shape {

public function getArea();

}

class Rectangle implements Shape {

protected $width;

protected $height;

public function setWidth($width) {

$this->width = $width;

}

public function setHeight($height) {

$this->height = $height;

}

public function getArea() {

return $this->width \* $this->height;

}

}

class Square implements Shape {

protected $side;

public function setSide($side) {

$this->side = $side;

}

public function getArea() {

return $this->side \* $this->side;

}

}

function calculateArea(Shape $shape) {

return $shape->getArea();

}

$rectangle = new Rectangle();

$rectangle->setWidth(5);

$rectangle->setHeight(10);

echo "Rectangle Area: " . calculateArea($rectangle) . "\n";

$square = new Square();

$square->setSide(5);

echo "Square Area: " . calculateArea($square) . "\n";

ه ان لمه class يعمل implement من كلاس تاني دع معناه انه مجبور يعمل implementation لكل الفانكشن اللي موجوده عند الكلاس والكلاس بتاعه مش هيشتغل غير لمه يعملهم كلهم implementation نتخيل كده في فانكشن انا مش محتاجها بس في نفس الوقت الكلاس مش هيشتغل غير لمه أعملها implementation اعمل اي وانا مش محتاجها؟؟

جه ال principle الرابع في ال sol'i'd وهو ال interface segregation principle (isp) بيقول اي بقا ال principle ده ان اي فانكشن انت مش محتاجها افصلها في interface (class ) لوحده واللي محتاج الفانكشن ده يبقا يعمل implement من الكلاس التاني وخلاص بس كده نضرب مثال

نفترض عندنا كلاس اسمه shop جوه الكلاس ده فانكشن اسمها buy() بتشتري اي حاجه وفانكشن اسمها sell() بتبيع اي حاجه وفانكشن اسمها free\_ice cream() ده بتدي ايس كريم مجانا دلوقت انا شخص هفتح متجر بيبيع ويشتري كتب ف هعمل implement من الكلاس اللي اسمها shop بس هيا كده بتجبرني ادي ايس كريم ببلاش وانا مش عايز اعمل كده ف الحل اننا هنعمل فانكشن buy و sell في كلاس لوحدهم و free\_ice cream في كلاس تاني ودلوقت هقدر ابيع واشتري بس واللي عايز يدي ايس كريم يبقا يعمل implement من الكلاس التاني و بكده يبقا عرفنا اي هو ال interface segregation principle بس كده

BEFor:

interface Worker {

public function work();

public function eat();

public function sleep();

}

class Developer implements Worker {

public function work() {

// Implement work method

}

public function eat() {

// Implement eat method

}

public function sleep() {

// Implement sleep method

}

}

class Manager implements Worker {

public function work() {

// Implement work method

}

public function eat() {

// Implement eat method

}

public function sleep() {

// Implement sleep method

}

}

After:

interface Workable {

public function work();

}

interface Feedable {

public function eat();

}

interface Sleepable {

public function sleep();

}

class Developer implements Workable, Feedable, Sleepable {

public function work() {

// Implement work method

}

public function eat() {

// Implement eat method

}

public function sleep() {

// Implement sleep method

}

}

class Manager implements Workable {

public function work() {

// Implement work method

}

}

اخر principle في ال \*soli\*D وهو ال dependency inversion principle (DIP)

واللي بيقول اني مخليش ال high level module معتمد علي low level module

High level module

ده الكلاس اللي بيستخدم كلاس تاني

Low level module

اكيد عكس اللي فوق مش بستخدم اي كلاس في ال implementation بتاعه

ندخل في المثال علي طول نفرض عندنا كلاس اسمه notification جو الكلاس ده بينادي علي كلاس تاني اسمه email عشان تعرضلي شعار من نوع email

يعني دلوقت لو جيت في ال main ونديت علي كلاس ال notification هيعملي شعار عن طريق ال email طب نفرض دلوقت عايز اعمل شعار SMS هعمل كلاس اسمه SMS وانادي عليه في كلاس ال notification

كده انت روحت عدلت في كلاس ال notification يبقا خالفت تاني مبدأ في ال solid اللي هو ال open closed (ال dependency قال هنعمل interface او abstract و نخلي ال SMS و ال email ي implement من ال interface ده و هنخلي ال notification مابيتعاملش غير مع ال interface ولو عوزت اعمل اي تعديلات او اضيف اي حاجه جديده غير ال SMS و ال email هضيفها لل interface بس بدون مااغير او اعمل اي حاجه في كلاس ال notification

ولمه اجي في ال main انادي علي notification هبعتلها اسم الرساله اللي عايزها زي كده

New notification(SMS)

وهيا بطريقتها هتعملي اللي انا عايزه و مش اضطر اعمل اي تعديلات في كلاس

Befor:

class Database {

public function connect() {

return 'Connected to database';

}

}

class UserRepository {

private $database;

public function \_\_construct() {

$this->database = new Database();

}

public function getUsers() {

return $this->database->connect() . ' - Fetching users';

}

}

$userRepository = new UserRepository();

echo $userRepository->getUsers();

After:

interface ConnectionInterface {

public function connect();

}

class Database implements ConnectionInterface {

public function connect() {

return 'Connected to database';

}

}

class UserRepository {

private $connection;

public function \_\_construct(ConnectionInterface $connection) {

$this->connection = $connection;

}

public function getUsers() {

return $this->connection->connect() . ' - Fetching users';

}

}

$database = new Database();

$userRepository = new UserRepository($database);

echo $userRepository->getUsers();