S.O.L.I.D. Principles

Agile Object Oriented design principles

Agenda

Problems

SOLID Principles

Workshop

Q & A

Problems

Specifications มีการเปลี่ยนแปลงได้ตลอดเวลา

ตามหลักการของ Agile โปรแกรมต้องมีความสามารถในการรับมือกับการเปลี่ยนแปลงเหล่านี้ได้

Problems

Rigidity (ความไม่ยืดหยุ่น)
การเปลี่ยนแปลงกระทบหลายจุด

Fragility (ความบอบบาง)
การเปลี่ยนแปลงกระทบ และทำให้จุดที่ไม่เกี่ยวข้องทำงานไม่ได้

Immobility (การไม่สามารถเคลื่อนที่ได้)
การไม่สามารถนำโค้ดไปใช้ซ้ำนอกเหนือจากจุดที่เขียนได้

SOLID Principles

Single responsibility principle

Open-closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

SOLID Principles

Single responsibility principle (กฎหนึ่งความรับผิดชอบ)

A class should have only one reason to change.
คลาสๆหนึ่งควรมีเหตุผลเพียงข้อเดียวเท่านั้นในการเปลี่ยนแปลง

(กลุ่มคนที่รับผิดชอบ 1 กลุ่ม = 1 เหตุผลที่จะเปลี่ยนแปลง)

SOLID Principles

```
class Employee {

…

public function calculateSalary() { … }

public function printReport() { … }

public function saveToDatabase() { … }

}
```

SOLID Principles

Open-closed principle (กฎเปิดเอ็นติตี้ที่ปิด?)

Software entities (classes and functions) should be open for extension and closed for modification.
คลาสและฟังก์ชั่นของโปรแกรมควรเปิดให้มีการต่อเติม แต่ปิดไม่ให้มีการแก้ไข

SOLID Principles

```
class Rectangle{

        private $width;
        private $height;

        public function __construct ($width, $height) { … }
        public function setWidth($width) { $this->width = $width; }
        Public function setHeight($height) { $this->height = $height; }
        Public function getWidth() { return $this->width; }
        Public function getHeight() { return $this->height; }

}
```

SOLID Principles

```
class AreaCalculator{

        public function getAreaSum($rectangles) {
                $area = 0;
                for($i = 0; $i < sizeof($rectangles); $i++){
                        $area += $rectangles[$i]->getHeight() * $rectangles[$i]-
>getWidth();
                }
                return $area;
        }

}
```

SOLID Principles

```
class Circle{

        private $radius;

        public function __construct ($radius) { $this->radius = $radius; }
        public function setRadius($radius) { $this->radius = $radius; }
        public function getRadius() { return $this->radius; }

}
```

SOLID Principles

```
class AreaCalculator{

        public function getAreaSum($shapes) {
                $area = 0;
                for($i = 0; $i < sizeof($shapes); $i++){
                        if( $shapes[$i] instanceof Rectangle ) {
                                $area += $shapes[$i]->getHeight() * $shapes[$i]-
>getWidth();
                        }
                        elseif( $shapes[$i] instanceof Circle ) {
                                $area += pi() * $shapes[$i]->getRadius * $shapes[$i]-
>getRadius;
                        }
                }
                return $area;
        }
```

SOLID Principles

```
interface Shape{
        public function getArea();
}

class Rectangle implements Shape {
        …
        public function getArea() { return $this->width * $this->height; }
}

class Circle implements Shape {
        …
        public function getArea() { return pi() * $this->radius * $this->radius; }
}
```

SOLID Principles

```
class AreaCalculator{

        public getAreaSum($shapes) {
                $area = 0;
                for($i = 0; $i < sizeof($shapes); $i++){
                        $area += $shapes[$i]->getArea();
                }
                return $area;
        }

}
```

SOLID Principles

Liskov substitution principle (กฎการแทนของลิสคอฟ)

Subtypes must be substitutable for their base type.
ถ้าที่จุดใดมีการใช้คลาสแม่ เราสามารถเปลี่ยนมาใช้คลาสลูกในจุดนั้นๆแทนคลาสแม่ได้ทันที โดยไม่เกิด Error

SOLID Principles

```php
class OrderProcessor{

    protected $orderRepo;

    public function __construct(OrderRepositoryInterface $orderRepo)
    {
        $this->orderRepo = $orderRepo;
    }

    public function process(Order $order, $userId){
        // ดำเนินการต่างๆ
        .
        .
        .

        // บันทึกรายการๆ สั่งซื้อสินค้า
        $this->orderRepo->logOrder($userId, $order, $order->price);
    }
}

interface OrderRepositoryInterface
{

    public function logOrder($userId, Order $order, $amountPaid);
}

class CsvOrderRepository implements OrderRepositoryInterface
{

    public function logOrder($userId, Order $order, $amountPaid)
    {
        // บันทึกข้อมูลลงไฟล์ในรูปแบบ CSV
    }
}
```

SOLID Principles

Shell

```
$order = new Order();
$order->id = 1;
$order->price = 200;

$csvRepo = new CsvOrderRepository();

$processor = new OrderProcessor($csvRepo);
$processor->process($order, 5);
```

SOLID Principles

```php
class DbOrderRepository implements OrderRepositoryInterface{

    protected $connection;

    public function connect($username, $password){
        // สร้าง connection
        $this->connection = new DatabaseConnection($username, $password);
    }

    public function logOrder($userId, Order $order, $amountPaid)
    {
        $this->connection->run(
            'INSERT INTO orders VALUE (?, ?, ?)',
            [
                $userId,
                $order->id,
                $amountPaid
            ]
        );
    }
}
```

SOLID Principles

```php
class OrderProcessor{

    protected $orderRepo;

    public function __construct(OrderRepositoryInterface $orderRepo)
    {
        $this->orderRepo = $orderRepo;
    }

    public function process(Order $order, $userId){
        // ดำเนินการต่างๆ
        .
        .
        .

        $this->orderRepo->connect('db_user01', 'password');

        // บันทึกรายการๆ สั่งซื้อสินค้า
        $this->orderRepo->logOrder($userId, $order, $order->price);
    }
}
```

SOLID Principles

Interface segregation principle (กฎการแยก Interface)

Classes those implement interfaces should not be forced to implement methods they do not use.

คลาสที่ต้อง implement interface ใดๆไม่ควรต้องถูกบังคับให้ implement methods ต่างๆที่คลาสนั้นไม่ได้ใช้

SOLID Principles

Dependency inversion principle (กฎการกลับการผูกติด)

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.

โมดูลระดับสูงไม่ควรผูกติดกับโมดูลระดับต่ำ ทั้งสองโมดูลควรผูกกับ Abstractions

Abstractions ไม่ควรต้องพึ่งรายละเอียดการ implement แต่รายละเอียดการ implement ต้องพึ่ง Abstractions