

Topic #2 - Design Pattern

Education Team

PHP version: 7.4.0



LAMPART

Nội dung

❖ Tổng quan

- Inversion of Control (IoC)
- Dependency Inversion Principle (DIP)
- Dependency Injection (DI)
- IoC containers

❖ Design patterns

- | | |
|-------------|-------------|
| – Factory | - Builder |
| – Decorator | - Observer |
| – Proxy | - Composite |
| – Facade | - Singleton |
| – Command | - Template |

Tổng quan

Ví dụ 1: N-Tier Architecture



- **UI** (user interface) là giao diện tương tác với người dùng
- **Service** xử lý dữ liệu người dùng truy vấn
- **BusinessLogic** xử lý các vấn đề nghiệp vụ
- **DataAccess** truy vấn dữ liệu trực tiếp trên cơ sở dữ liệu

Tổng quan

Ví dụ 2: minh họa IoC

Cho ví dụ

```

6 class BusinessLogic {
7
8     ❶ private DataAccess $dataAccess;
9
10    public function construct() {
11        ❷ $this->_dataAccess = new DataAccess();
12    }
13
14    ❸ public function getCustomerName(int $id): string {
15        return $this->_dataAccess->getCustomerName($id);
16    }
17
18 }
19
20 //Application
21 $foo = new BusinessLogic();
22 echo $foo->getCustomerName(9);
23 //Get customer name by id = 9

```

```

3 class DataAccess {
4
5     public function __construct() {
6
7     }
8
9     // get it from DB in real app
10    public function getCustomerName(int $id): string {
11        return "Get customer name by id = {$id}";
12    }
13

```

Tổng quan

Ví dụ 2: minh họa IoC

Phân tích ví dụ

- Lớp BusinessLogic, DataAccess được thiết kế theo hướng **tightly coupled**
- Lớp BusinessLogic phụ thuộc vào lớp DataAccess
- Lớp BusinessLogic thực hiện tạo và quản lý lifetime của đối tượng từ lớp DataAccess
- DataAccess có thay đổi bên trong dẫn đến lớp BusinessLogic thay đổi (rename, remove, ...)
- DataAccess thay đổi tên lớp dẫn đến các lớp phụ thuộc phải cập nhật tên
- TDD (Test Driven Development) - không thể test 1 cách độc lập

Tổng quan

Ví dụ 2: minh họa IoC

Giải pháp

- Dùng thêm lớp Factory để quản lý việc tạo và lifetime của đối tượng

```
3 class DataAccessFactory {  
4  
5     public static function getObjectOfDataAccess(): DataAccess {  
6         return new DataAccess();  
7     }  
8  
9 }
```

```
6 class BusinessLogic {  
7  
8     private DataAccess $dataAccess;  
9  
10    public function construct() {  
11        $this->_dataAccess = DataAccessFactory::getObjectOfDataAccess();  
12    }  
13  
14    public function getCustomerName(int $id): string {  
15        return $this->_dataAccess->getCustomerName($id);  
16    }  
17  
18 }
```

Tổng quan

Ví dụ 2: minh họa IoC

Nhận xét

- Thay vì dùng từ khóa **new**, tạo đối tượng từ **DataAccessFactory**
- Đảo ngược việc tạo đối tượng từ **BusinessLogic** sang **DataAccessFactory**
- Lớp **BusinessLogic** vẫn còn dùng lớp **DataAccess**
- ???TDD

Tổng quan

Ví dụ 3: minh họa DIP (Dependency Inversion Principle)

Cho ví dụ

```
1 interface IDataAccess
2 {
3     public function getCustomerName(int $id) : string;
4 }
```

1

```
5 class DataAccess implements IDataAccess {
6
7     public function __construct() {
8
9     }
10
11     // get it from DB in real app
12     public function getCustomerName(int $id): string {
13         return "Get customer name by id = {$id}";
14     }
15 }
16
```

2

Tổng quan

Ví dụ 3: minh họa DIP (Dependency Inversion Principle)

Cho ví dụ

```
4
5 class DataAccessFactory {
6
7     public static function getObjectOfDataAccess(): IDataAccess {
8         return new DataAccess();
9     }
10
11 }
```

3

```
5 class BusinessLogic {
6
7     private IDataAccess $dataAccess;
8
9     public function __construct() {
10         $this->_dataAccess = DataAccessFactory::getObjectOfDataAccess();
11     }
12
13     public function getCustomerName(int $id): string {
14         return $this->_dataAccess->getCustomerName($id);
15     }
16
17 }
```

4

Tổng quan

Ví dụ 3: minh họa DIP (Dependency Inversion Principle)

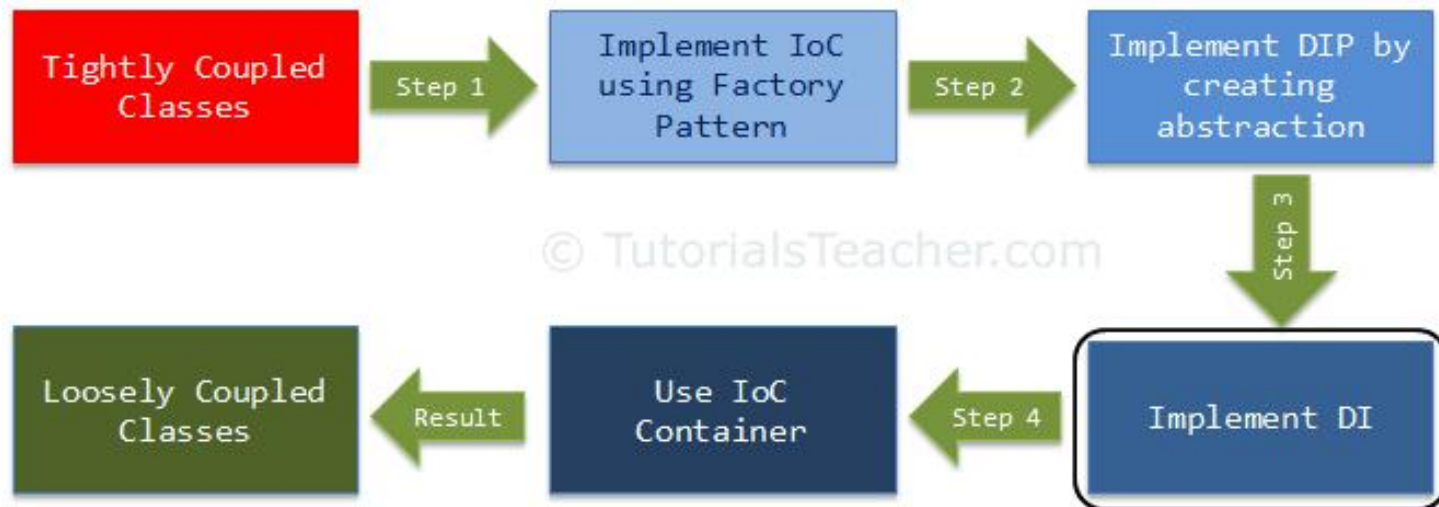
Phân tích ví dụ

- High-level: BusinessLogic
- Low-level: DataAccess
- Cả 2 lớp phụ thuộc vào interface IDataAccess
- Interface IDataAccess không phụ thuộc vào hiện thực DataAccess
- Lớp BusinessLogic không còn phụ thuộc vào chi tiết DataAccess, có thể dùng hiện thực khác của IDataAccess
- BusinessLogic vẫn còn sử dụng DataAccessFactory để lấy một đối tượng có kiểu IDataAccess
- Giả sử BusinessLogic cần dùng một hiện thực khác của IDataAccess => cần thay đổi tại BusinessLogic

Tổng quan

Ví dụ 4: minh họa Dependency Injection

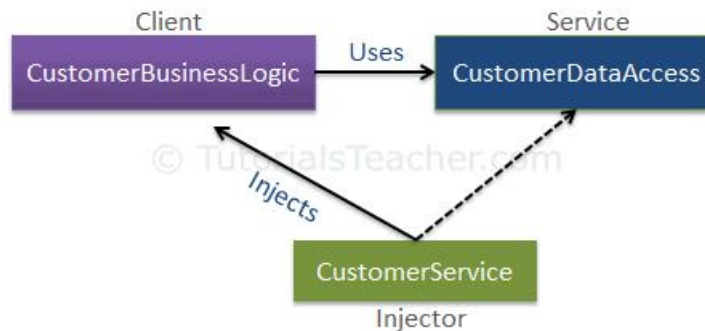
Luồng xử lý



Tổng quan

Ví dụ 4: minh họa Dependency Injection

Cho ví dụ



Tổng quan

Ví dụ 4: minh họa Dependency Injection

Cho ví dụ

```
1 interface IDataAccess
2 {
3     public function getCustomerName(int $id) : string;
4 }
```

1

```
5 class DataAccess implements IDataAccess {
6
7     public function __construct() {
8
9     }
10
11     // get it from DB in real app
12     public function getCustomerName(int $id): string {
13         return "Get customer name by id = {$id}";
14     }
15 }
16
```


2

Tổng quan

Ví dụ 4: minh họa Dependency Injection

Cho ví dụ

```
6 class BusinessLogic {
7
8     private IDataAccess $dataAccess;
9
10    public function construct(IDataAccess $dataAccess = null) {
11        if (is_null($dataAccess)) {
12            $this->_dataAccess = new DataAccess();
13        } else {
14            $this->_dataAccess = $dataAccess;
15        }
16    }
17
18    public function getCustomerName(int $id): string {
19        return $this->_dataAccess->getCustomerName($id);
20    }
21
22 }
```



Tổng quan

Ví dụ 4: minh họa Dependency Injection

Cho ví dụ

```
5 class Service {
6
7     private BusinessLogic $businessLogic;
8
9     public function __construct() {
10         $this->businessLogic = new BusinessLogic(new DataAccess());
11     }
12
13     public function getCustomerName(int $id): string {
14         return $this->businessLogic->getCustomerName($id);
15     }
16
17 }
18
19 //Application
20 $foo = new Service();
21 $foo->getCustomerName(9);
22 //Get customer name by id = 9
```

4

Tổng quan

Ví dụ 4: minh họa Dependency Injection

Nhận xét

- Lớp Service tạo và inject đối tượng của lớp DataAccess vào lớp BusinessLogic thông qua constructor (property, method)
- Lớp BusinessLogic không tạo đối tượng DataAccess
- Chương trình trở nên rời rạc hơn (more loosely coupled)

Tổng quan

Phụ thuộc

- Class BusinessLogic phụ thuộc vào class DataAccess
(Class BusinessLogi is dependent on class DataAccess)
- Class DataAccess là một phụ thuộc của class BusinessLogic
Class DataAccess is a dependency of class BusinessLogic

Tổng quan

OOP - IoC

❖ OOP

- Các lớp cần tương tác với nhau để hoàn thành tính năng của một ứng dụng
- Do đó vấn đề tạo và quản lý life time của một đối tượng do lớp phụ thuộc thực hiện

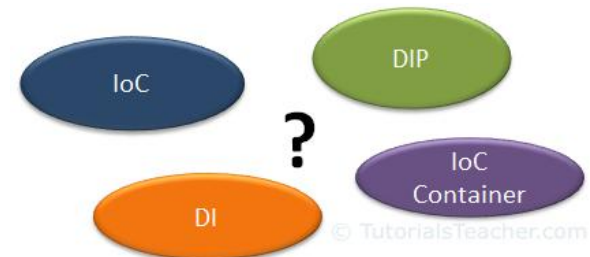
❖ IoC

- Đảo ngược luồng điều khiển trên của lập trình thủ tục ở trên
- Ủy nhiệm việc tạo và quản lý life time cho lớp khác thực hiện

Tổng quan

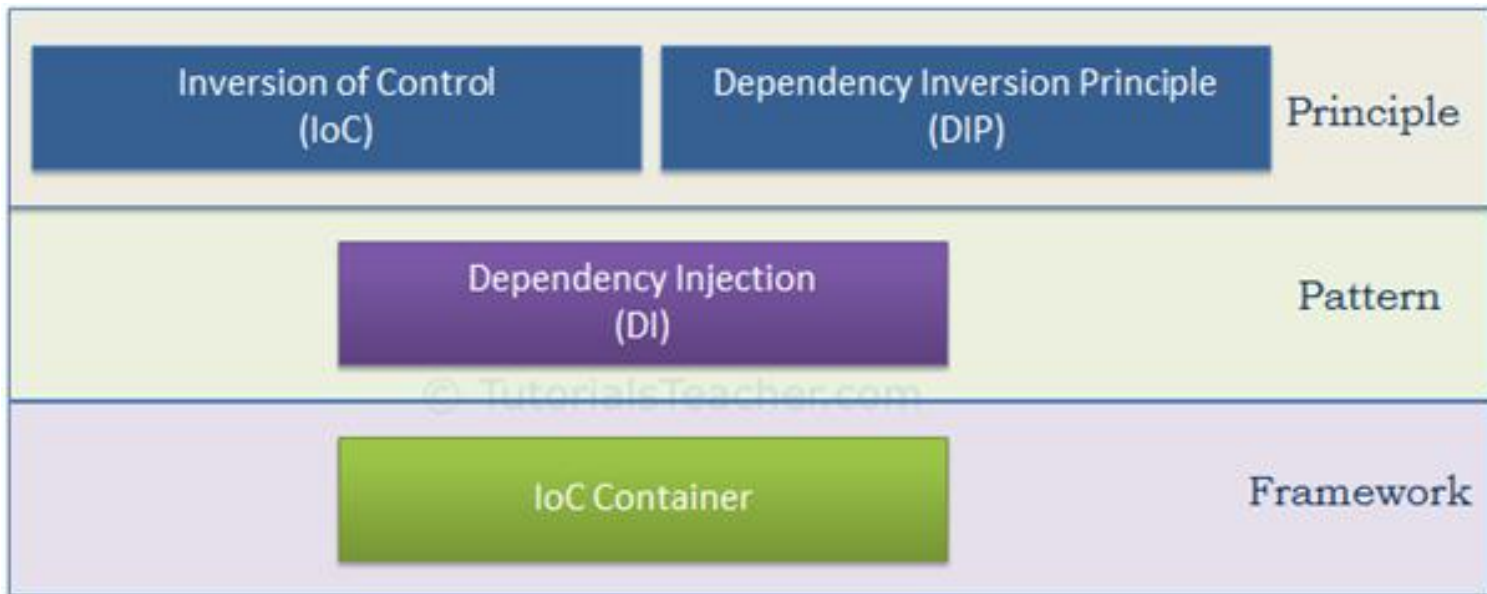
What

- (1) Inversion of Control (IoC)
- (2) Dependency Inversion Principle (DIP)
- (3) Dependency Injection (DI)
- (4) IoC containers



Tổng quan

What



Tổng quan

Design Principle - Design Pattern

❖ Design Principle

- High level - lời khuyên để thiết kế ứng dụng trở nên tốt hơn
- KHÔNG đưa ra hiện thực chi tiết hay phụ thuộc ngôn ngữ lập trình
- Ví dụ: SOLID, DRY, KISS , YAGNI

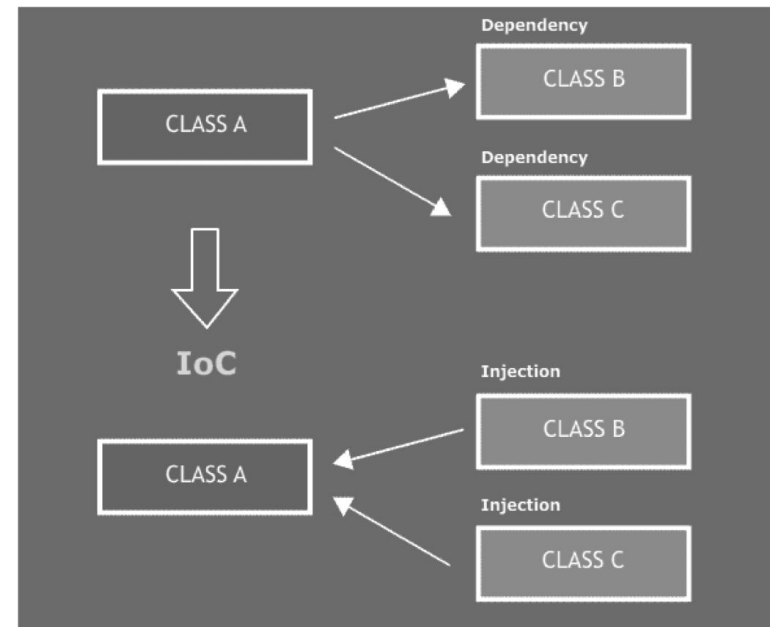
❖ Design Pattern

- Giải pháp giải quyết các vấn đề phổ dụng
- Đưa ra hiện thực cụ thể cho một vấn đề cụ thể
- Ví dụ: tạo 1 lớp chỉ có 1 đối tượng tại 1 thời điểm, Singleton
Design Pattern là 1 lựa chọn. Một số design pattern khác: factory, builder, ...

Tổng quan

IOC

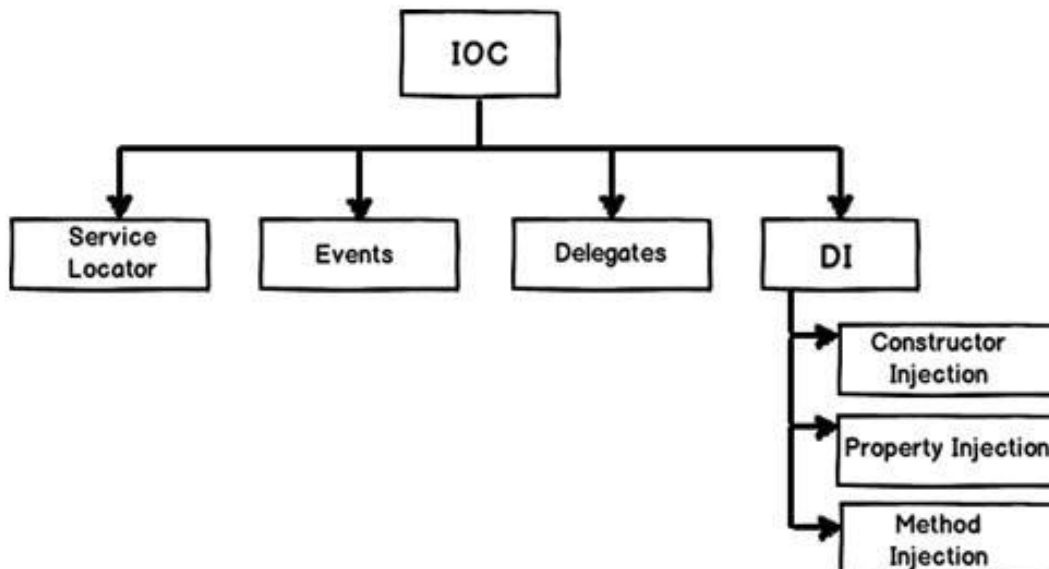
- IOC - Inversion of Control
- Là Design Principle
- Ngược với lập trình thủ tục (procedural programming)
- IOC sử dụng các đối tượng từ các lớp thông qua Injection
- Injection: setter, method, constructor



Tổng quan

IOC

- Ứng dụng
- Các design pattern sử dụng IOC



Tổng quan

Dependency Inversion Principle

- Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
- Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

Tổng quan

Dependency Injection

- Là design pattern
- Hiện thực IoC
- Cho phép tạo đối tượng phụ thuộc bên ngoài một lớp
- Cho phép gửi đối tượng bên ngoài trong lớp theo các cách khác nhau (constructor, property, method)

Tổng quan

IoC Container

- Là framework quản lý tự động Dependency Injection
- Laravel, CodeIgniter, Zend (Laminas)

Tổng quan

Design Principle

Những nguyên lý thiết kế trong OOP => code dễ đọc, dễ test, rõ ràng hơn => maintainance code sẽ dễ hơn rất nhiều => SOLID là 1 trong số các nguyên lý giúp làm điều này.

Tổng quan

Design Principle

SOLID

Single responsibility principle

- Một class chỉ nên giữ 1 trách nhiệm duy nhất

Ví dụ: Error

```
public class ReportManager()  
{  
    public void ReadDataFromDB();  
    public void ProcessData();  
    public void PrintReport();  
}
```

Tổng quan

Design Principle

SOLID

Open/closed principle

- Có thể thoải mái mở rộng 1 class, nhưng không được sửa đổi bên trong class đó
- Theo nguyên lý này, mỗi khi ta muốn thêm chức năng,... cho chương trình, chúng ta nên viết class mới mở rộng class cũ (bằng cách kế thừa hoặc sở hữu class cũ) không nên sửa đổi class cũ.

Tổng quan

Design Principle

SOLID

Liskov Substitution Principle

- Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình
- List x = new ArrayList() (???)

Tổng quan

Design Principle

SOLID

Interface Segregation Principle

- Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể

Tổng quan

Design Principle

SOLID

Dependency inversion principle

- Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
- Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại.(Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)

Tổng quan

KISS, DRY, YAGNI

KISS

KISS = Keep It Simple Stupid

- Hãy thiết kế một lớp trở nên đơn giản và dễ nhìn hơn
- Tránh kiểu tổng hợp hay lẫn lộn

DRY

Don't Repeat Yourself

- Tránh lặp lại hãy đóng gói thành phương thức
- Hay sử dụng tính kế thừa

YAGNI

You Aren't Gonna Need It

- Hãy làm tốt cho chức hiện ở thời điểm hiện tại
- Tránh lãng phí phát triển tính năng có thể sử dụng đến

Design Pattern

1. Factory Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

2. Decorator Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

3. Proxy Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

4. Repository Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

5. Command Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

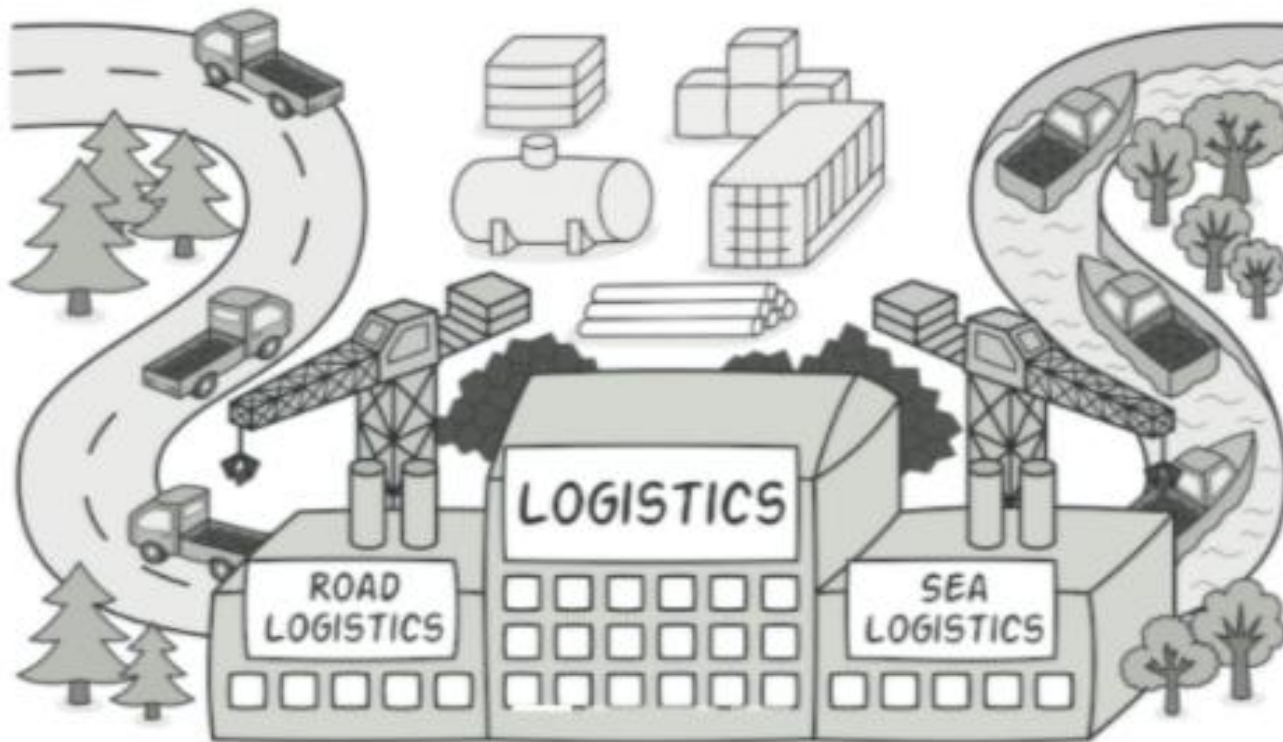
6. Bridge Pattern

- a) Vấn đề
- b) Giải pháp
- c) Áp dụng

Mục tiêu

1. Tìm hiểu rõ về các Design Pattern
2. Thiết kế được giải pháp tối ưu với các yêu cầu mở rộng, nâng cao dự án
3. Ghi nhận đóng góp từ các anh/em trong công ty
4. Chia sẻ, trao đổi, thảo luận trên tinh thần tích cực nhất

1.Factory Design Pattern



1.Factory Design Pattern

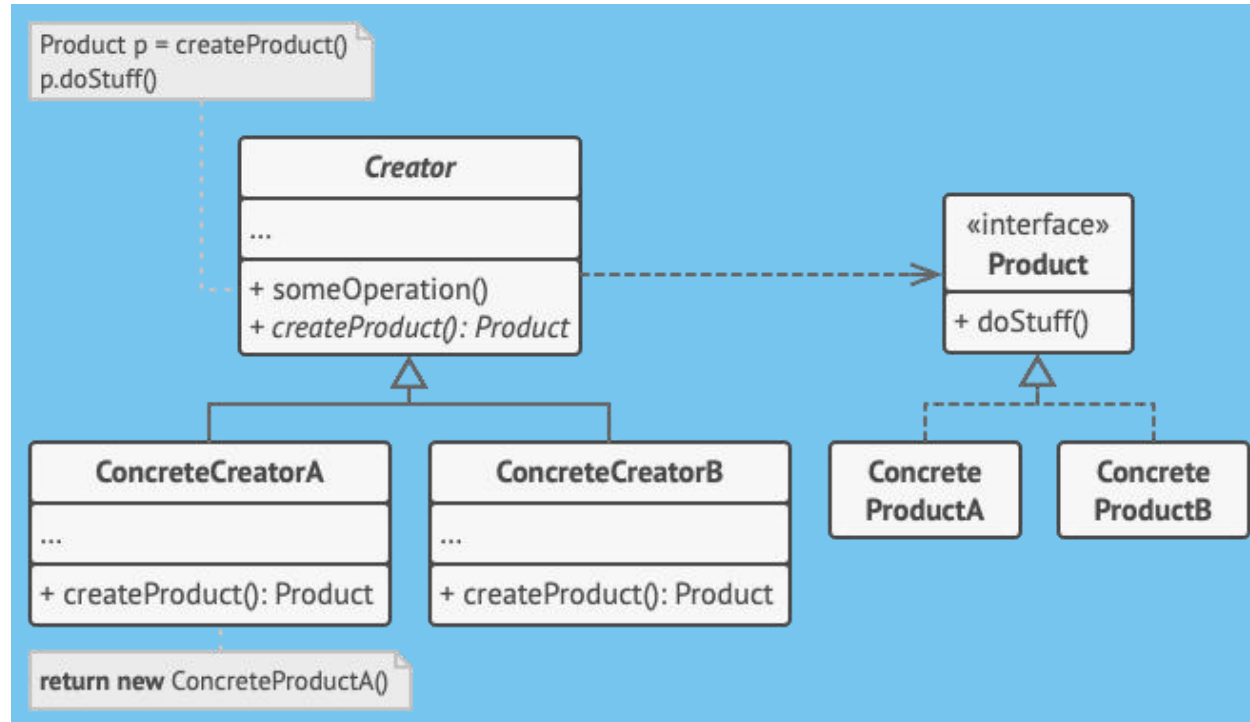
- a. Định nghĩa
 - Factory Pattern thuộc nhóm khởi tạo (Creational Pattern)
 - Nhiệm vụ chính của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đối tượng một cách linh hoạt hơn.
 - Giải quyết vấn đề tạo một đối tượng mà không cần thiết chỉ ra một cách chính xác lớp nào sẽ được tạo.

1.Factory Design Pattern

- Sử dụng khi nào?
 - Tạo ra một cách mới trong việc khởi tạo các object thông qua một interface chung
 - Bao gói được quá trình khởi tạo object, giấu đi được xử lý logic của việc khởi tạo
 - Giảm sự phụ thuộc giữa các module, logic với các class cụ thể mà chỉ phụ thuộc với interface hoặc abstract class.

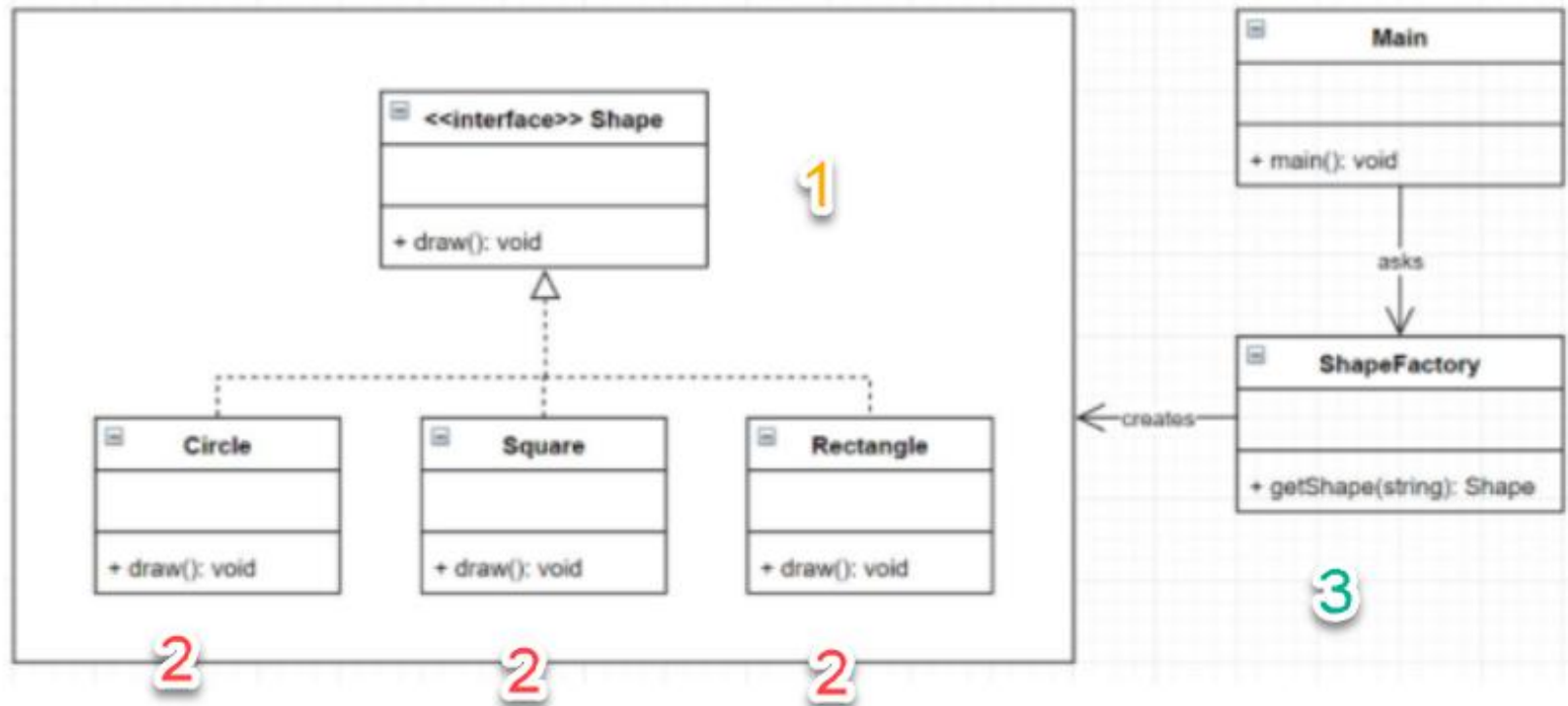
1.Factory Design Pattern

- Cấu trúc



1.Factory Design Pattern


- Thiết kế



1.Factory Design Pattern

Shape

```
1  <?php
2
3  interface Shape
4  {
5      function draw() :void;
6  }
```





1.Factory Design Pattern

Shape implement

```
1  <?php
2
3
4  class Circle implements Shape
5  {
6      function draw() :void
7      {
8          // TODO: Implement draw() method.
9          echo "Inside Circle::draw() method.";
10     }
11 }
12
```



```
1  <?php
2
3  
4  class Square implements Shape
5  {
6
7      function draw() :void
8      {
9          // TODO: Implement draw() method.
10         echo "Inside Square::draw() method.";
11     }
12 }
```



```
1  <?php
2
3  
4  class Rectangle implements Shape
5  {
6
7      function draw() :void
8      {
9          // TODO: Implement draw() method.
10         echo "Inside Rectangle::draw() method.";
11     }
12 }
```



1.Factory Design Pattern

ShapeFactory

```
1  <?php
2
3
4  class ShapeFactory
5  {
6      function getShape($shapeType) : ?Shape
7      {
8          if ($shapeType == null){
9              return null;
10         }
11         if ($shapeType == "CIRCLE"){
12             return new Circle();
13         } elseif ($shapeType == "RECTANGLE"){
14             return new Rectangle();
15         } elseif ($shapeType == "SQUARE"){
16             return new Square();
17         }
18         return null;
19     }
20 }
```

3

1.Factory Design Pattern

Demo and result

```
$shape = new ShapeFactory();  
$circle = $shape->getShape( shapeType: "CIRCLE");  
$rectangle = $shape->getShape( shapeType: "RECTANGLE");  
$square = $shape->getShape( shapeType: "SQUARE");  
  
$circle->draw();  
echo "<br>";  
$rectangle->draw();  
echo "<br>";  
$square->draw();
```

demo

Kết quả demo

Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.

1.Factory Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

- Khi bạn mở rộng một chương trình mà không làm thay đổi code của nó với Factory Pattern bạn sẽ biết rõ những thuộc tính, hành vi sẽ tồn tại trong đối tượng của bạn sẽ hoạt động
- Factory Pattern phân tách mã nguồn với mã sử dụng đối tượng từ đó bạn có thể dễ dàng phát triển sản phẩm và đối tượng một cách độc lập
- Factory Pattern giúp bạn sử dụng lại code một cách hiệu quả thay vì xây dựng lại chúng mỗi lần khởi tạo một đối tượng

1.Factory Design Pattern

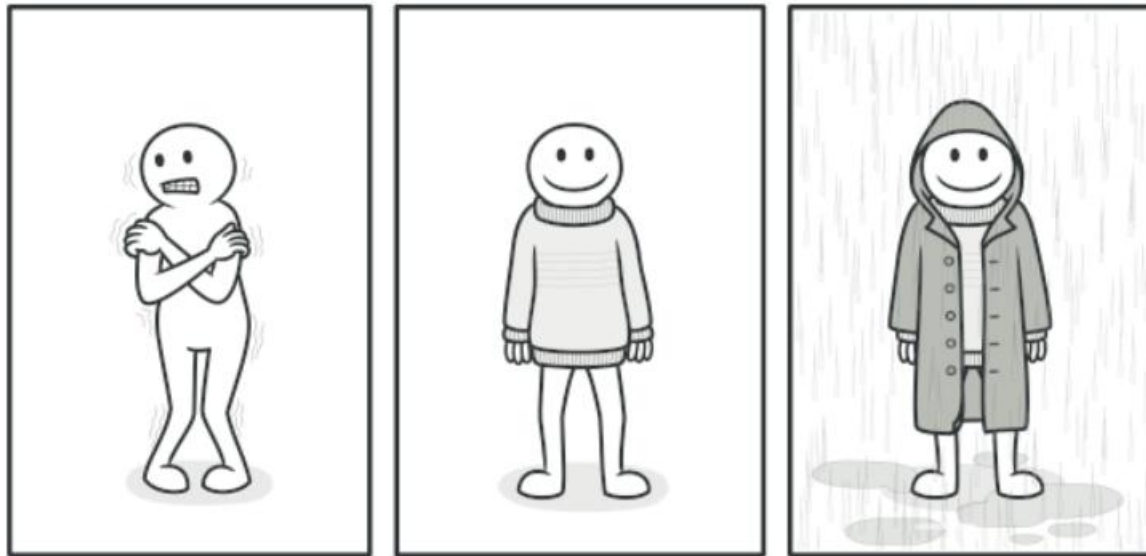
Ưu điểm

- Tạo ra một cách mới trong việc khởi tạo các Object thông qua một interface chung
- Khởi tạo các Object mà che giấu đi xử lý logic của việc khởi tạo đó
- Giảm sự phụ thuộc giữa các module, các logic với các class cụ thể, mà chỉ phụ thuộc vào interface hoặc abstract class

Nhược điểm

- Code có thể trở nên phức tạp khi có quá nhiều lớp con

2.Decorator Design Pattern



2.Decorator Design Pattern

- a. Định nghĩa

Decorator là mẫu cấu trúc cho phép thêm các tính năng mới vào một đối tượng đã có mà không làm thay đổi cấu trúc lớp của nó.

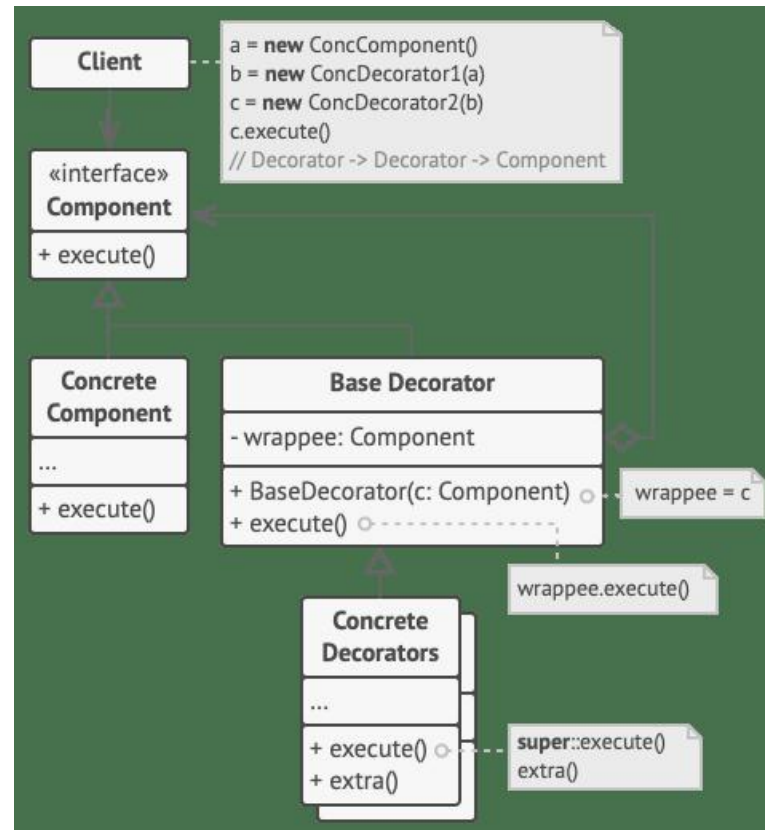
2.Decorator Design Pattern

- Sử dụng khi nào?

Khi muốn thêm các tính năng cho một đối tượng đã có một cách linh động trong thời gian chạy (run-time) mà không làm thay đổi đến các đối tượng khác của cùng lớp. Khi việc thêm tính năng mới khiến t ả phải tạo ra nhiều lớp kế thừa và mã nguồn trở nên đồ sộ

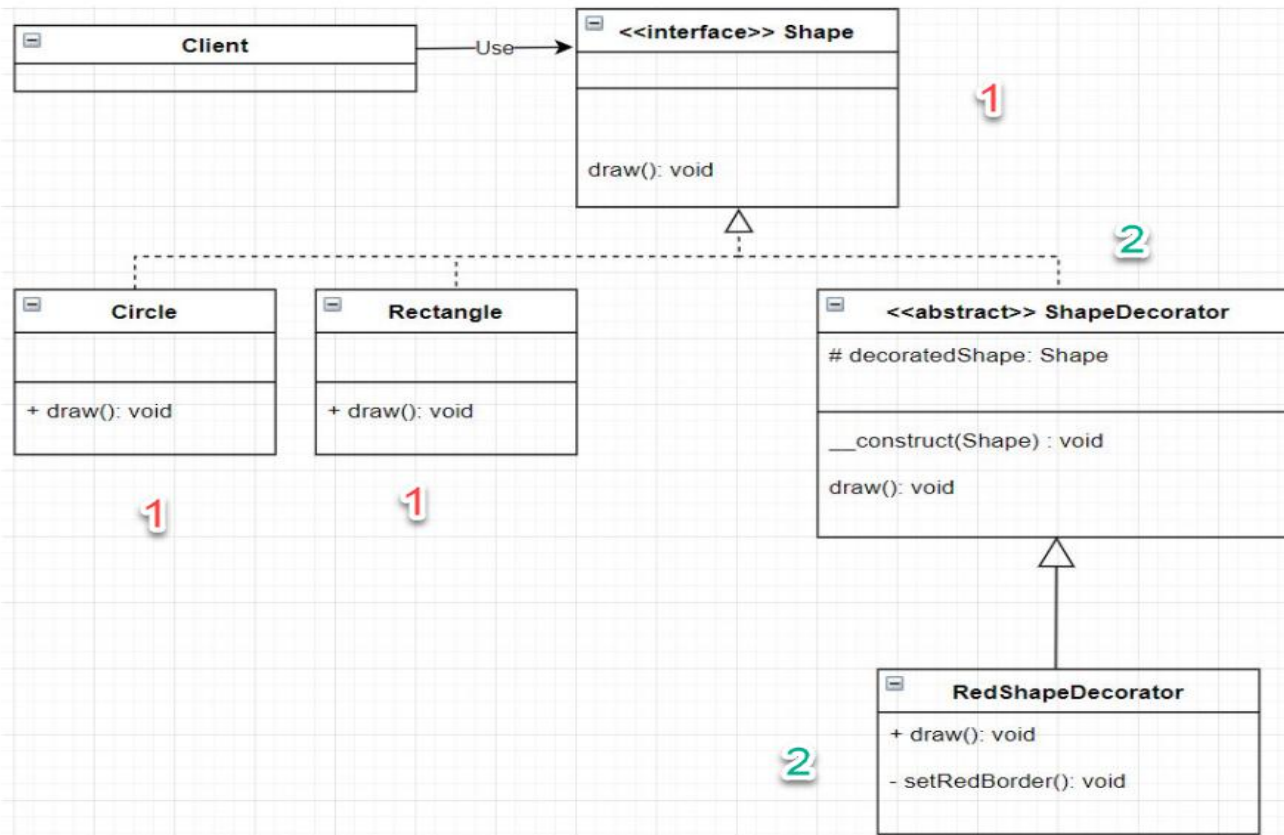
2.Decorator Design Pattern

- Cấu trúc



2.Decorator Design Pattern

- Thiết kế



2.Decorator Design Pattern

Shape

```
1  <?php
2
3  namespace Decorator;
4  interface Shape {
5      public function draw();
6  }
7
```



2.Decorator Design Pattern

Shape implement

```
1
2 namespace Decorator;
3 require_once("Shape.php");
4
5 class Circle implements Shape 1
6 {
7     public function draw()
8     {
9         echo "Shape: Circle ";
10    }
11 }
```

```
1 <?php
2 namespace Decorator;
3 require_once("Shape.php");
4
5 class Rectangle implements Shape 1
6 {
7     public function draw()
8     {
9         echo "Shape: Rectangle ";
10    }
11 }
```

2.Decorator Design Pattern

ShapeDecorator

```
1  <?php
2  namespace Decorator;
3  require_once("Shape.php");
4
5  abstract class ShapeDecorator implements Shape
6  {
7      protected $decoratedShape;
8
9      function __construct(Shape $decoratedShape){
10         $this->decoratedShape = $decoratedShape;
11     }
12
13     public function draw(){}
14 }
```

2

2.Decorator Design Pattern

ShapeDecorator Implementation

```
1  <?php
2  namespace Decorator;
3  require_once("Shape.php");
4  require_once("ShapeDecorator.php");
5
6  class RedShapeDecorator extends ShapeDecorator
7  {
8      protected $decoratedShape;
9
10     function __construct(Shape $decoratedShape){
11         parent::__construct($decoratedShape);
12     }
13
14     public function draw()
15     {
16         $this->decoratedShape->draw();
17         $this->setRedBorder($this->decoratedShape);
18     }
19     private function setRedBorder(Shape $decoratedShape){
20         echo "Border Color: Red ";
21     }
22 }
```

2

2.Decorator Design Pattern

Demo and Result

```
1 <?php
2
3 use ...
4
5
6
7 require_once("Circle.php");
8 require_once("Rectangle.php");
9 require_once("RedShapeDecorator.php");
10 $circle = new Circle();
11 echo "Circle with normal border: ";
12 $circle->draw();
13 echo "<br>";
14 echo "<br>";
15
16 $redRectangle = new RedShapeDecorator(new Rectangle());
17 echo "Rectangle of red border: ";
18 $redRectangle->draw();
19 echo "<br>";
20 echo "<br>";
21
22 echo "Circle of red border: ";
23 $redCircle = new RedShapeDecorator(new Circle());
24 $redCircle->draw();
25 echo "<br>";
26 echo "<br>";
27 ?>
```

Kết quả demo Circle with normal border: Shape: Circle

Rectangle of red border: Shape: Rectangle Border Color: Red

Circle of red border: Shape: Circle Border Color: Red

2.Decorator Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

- Khi sử dụng decorator bạn có thể thêm các hành vi bổ sung cho đối tượng mà k làm ảnh hưởng đến code
- Thay vì phải tạo ra nhiều đối tượng mới với các thuộc tính cũ và các phương thức bổ sung thì có thể dùng mẫu pattern này để kế thừa lại các thuộc tính cũ và thêm vào các thuộc tính mới
- Có thể dễ dàng thay đổi trở về kiểu decorator ban đầu mà không làm ảnh hưởng đến logic code

2.Decorator Design Pattern

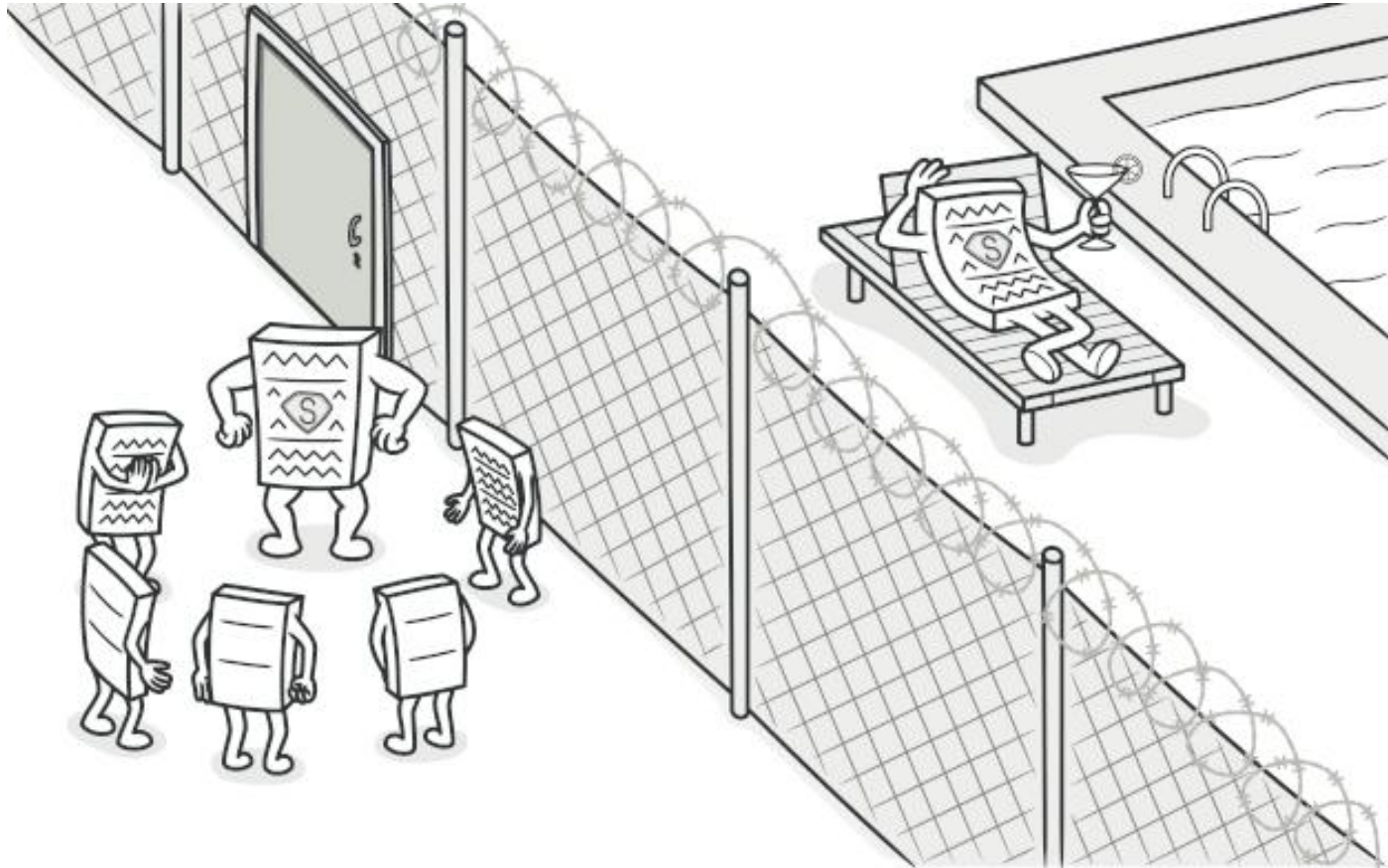
Ưu điểm

- Bạn có thể mở rộng hành vi của một đối tượng mà không cần tạo một lớp con mới.
- Bạn có thể thêm hoặc xóa các trách nhiệm khỏi một đối tượng trong thời gian chạy.
- Bạn có thể kết hợp một số hành vi bằng cách gói một đối tượng thành nhiều trình trang trí.
- Nguyên tắc Trách nhiệm Đơn lẻ. Bạn có thể chia một lớp đơn nguyên thực thi nhiều biến thể có thể có của hành vi thành một số lớp nhỏ hơn.

Nhược điểm

- Việc sử dụng kết hợp các lớp decorator với nhau có thể khiến code trở nên phức tạp
- Việc khởi tạo một đối tượng Decorator có thể trở nên phức tạp khi mà khởi tạo ta phải khởi tạo luôn cả các lớp bên trong đối tượng Decorator

3.Proxy Design Pattern



3.Proxy Design Pattern

- a. Định nghĩa

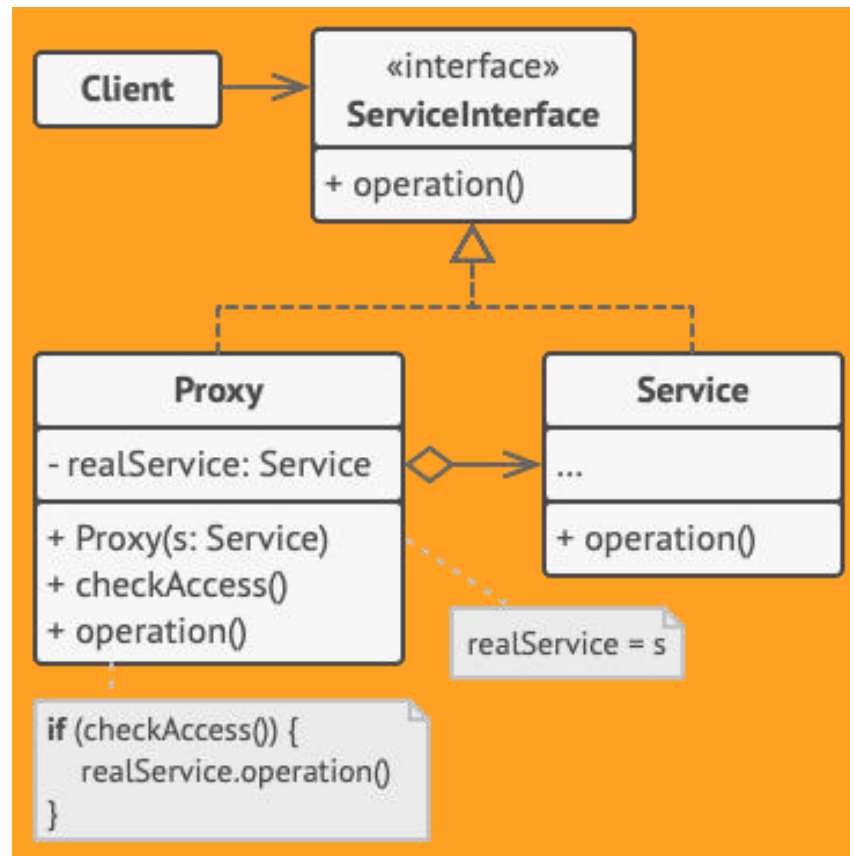
Proxy là một mẫu thiết kế thuộc nhóm cấu trúc cho phép bạn cung cấp một vật thay thế hoặc trình giữ chỗ cho một đối tượng khác. Một proxy kiểm soát quyền truy cập vào đối tượng gốc, cho phép bạn thực hiện một điều gì đó trước hoặc sau khi yêu cầu được chuyển đến đối tượng gốc.

3.Proxy Design Pattern

- Sử dụng khi nào?
 - Khi muốn bảo vệ quyền truy xuất vào các phương thức của object thực
 - Khi cần một số thao tác bổ sung trước khi thực hiện phương thức của object thực.
 - Khi tạo đối tượng ban đầu là theo yêu cầu hoặc hệ thống yêu cầu sự chậm trễ khi tải một số tài nguyên nhất định (lazy loading)
 - Khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.
 - Khi đối tượng gốc tồn tại trong môi trường từ xa (remote).
 - Khi đối tượng gốc nằm trong một hệ thống cũ hoặc thư viện của bên thứ ba.
 - Khi muốn theo dõi trạng thái và vòng đời đối tượng.

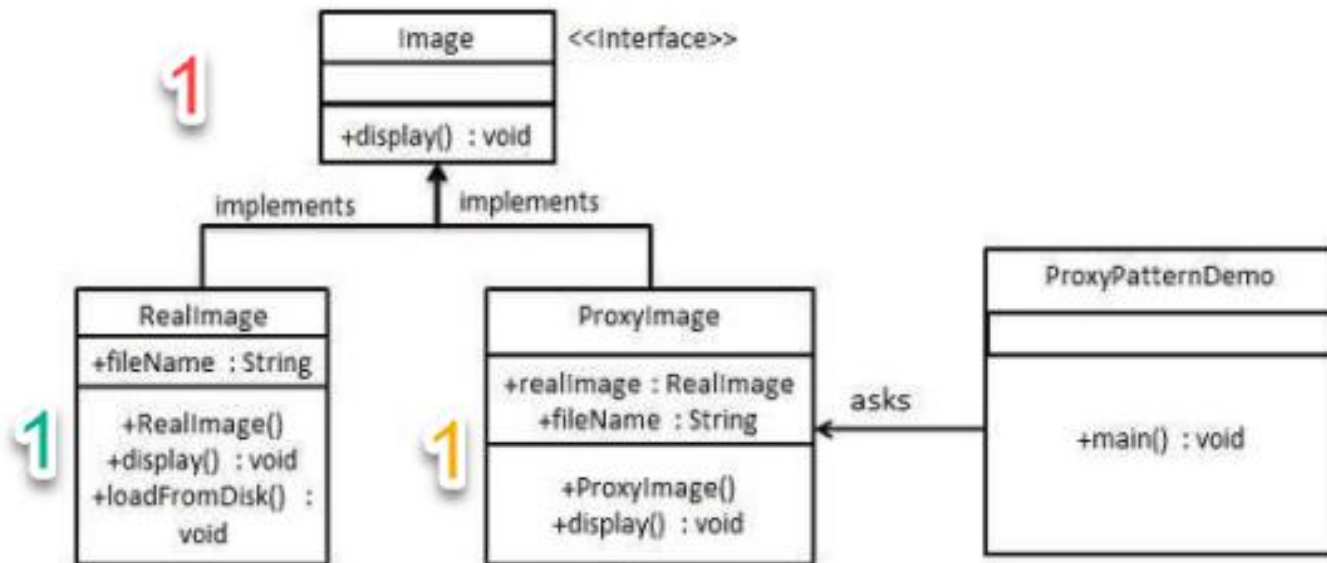
3.Proxy Design Pattern

- Cấu trúc



3.Proxy Design Pattern

- Thiết kế



3.Proxy Design Pattern

Image

```
1  <?php
2  interface Image
3  {
4      function display():void ;
5  }
```

1

3.Proxy Design Pattern

Real Image

```
1  <?php
2
3
4  class RealImage implements Image
5  {
6      private $fileName;
7
8      public function __construct($fileNameIn)
9      {
10         $this->fileName = $fileNameIn;
11         $this->loadFromDisk($fileNameIn);
12     }
13
14     function display(): void
15     {
16         // TODO: Implement display() method.
17         echo "Displaying " . $this->fileName . "<br>";
18     }
19
20     private function loadFromDisk($fileNameIn)
21     {
22         echo "Loading ".$fileNameIn . "<br>";
23     }
24 }
```

3.Proxy Design Pattern

Proxy Image

```
1  <?php
2
3
4  class ProxyImage implements Image
5  {
6      private $realImage;
7      private $fileName;
8
9      public function ProxyImage($fileNameIn)
10     {
11         $this->fileName = $fileNameIn;
12     }
13
14     function display(): void
15     {
16         // TODO: Implement display() method.
17         if($this->realImage == null)
18         {
19             $this->realImage = new RealImage($this->fileName);
20         }
21         $this->realImage->display();
22     }
23 }
```

3.Proxy Design Pattern

Demo and result

```
<?php
require_once "Image.php";
require_once "RealImage.php";
require_once "ProxyImage.php";
$image = new ProxyImage( fileName: "hinhne.jpg");
$image->display();
echo "<br>";
$image->display();
?>
```

demo

Kết quả demo

Loading hinhne.jpg
Displaying hinhne.jpg

Displaying hinhne.jpg

3.Proxy Design Pattern

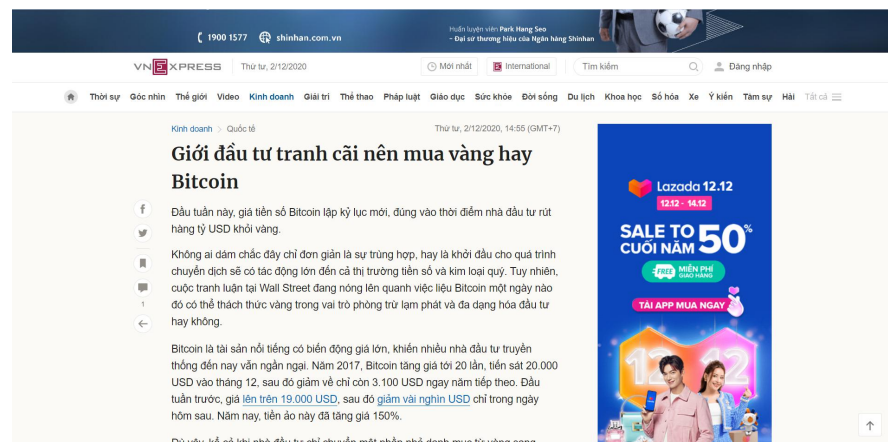
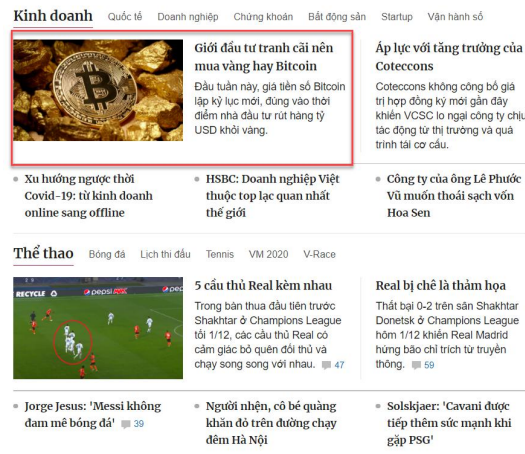
Các loại Proxy

- Virtual Proxy : Virtual Proxy tạo ra một đối tượng trung gian mỗi khi có yêu cầu tại thời điểm thực thi ứng dụng, nhờ đó làm tăng hiệu suất của ứng dụng.
- Protection Proxy : Phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu.
- Remote Proxy : Client truy cập qua Remote Proxy để chiếu tới một đối tượng được bảo vệ nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác).
- Monitor Proxy : Monitor Proxy sẽ thiết lập các bảo mật trên đối tượng cần bảo vệ, ngăn không cho client truy cập một số trường quan trọng của đối tượng. Có thể theo dõi, giám sát, ghi log việc truy cập, sử dụng đối tượng.
- Firewall Proxy : bảo vệ đối tượng từ chối các yêu cầu xuất xứ từ các client không tin nhiệm.
- Cache Proxy : Cung cấp không gian lưu trữ tạm thời cho các kết quả trả về từ đối tượng nào đó, kết quả này sẽ được tái sử dụng cho các client chia sẻ chung một yêu cầu gửi đến. Loại Proxy này hoạt động tương tự như Flyweight Pattern.
- Smart Reference Proxy : Là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu.
- Synchronization Proxy : Đảm bảo nhiều client có thể truy cập vào cùng một đối tượng mà không gây ra xung đột. Khi một client nào đó chiếm dụng khóa khá lâu khiến cho số lượng các client trong danh sách hàng đợi cứ tăng lên, và do đó hoạt động của hệ thống bị ngưng trệ, có thể dẫn đến hiện tượng “tắc nghẽn”.
- Copy-On-Write Proxy : Loại này đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn. Copy-On-Write Proxy là một thiết kế rất phức tạp.

3.Proxy Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

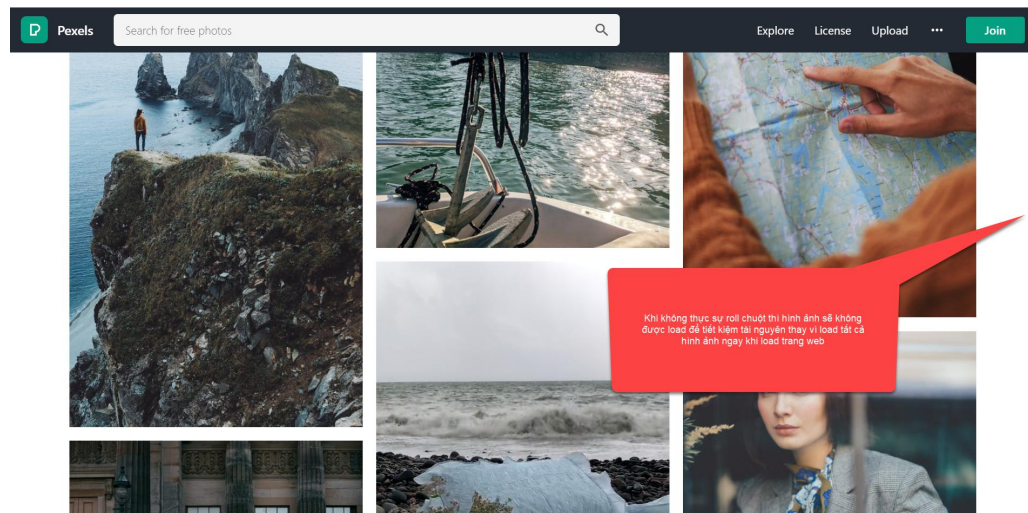
- Thay vì khi truy cập vào một đối tượng thật ta sẽ truy cập qua một đối tượng được xây dựng giống như đối tượng thật, tại đối tượng proxy này chúng ta có thể thực hiện các thao tác xác nhận, hoặc kiểm tra sau đó mới gọi đến đối tượng thật! Thay vì gọi đến đối tượng thật ngay lần đầu truy cập đến đối tượng



3.Proxy Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

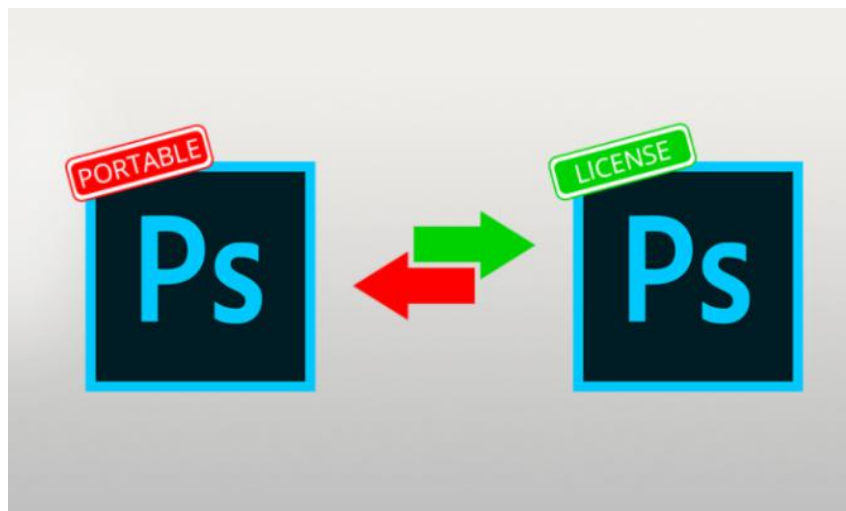
- Virtual proxy: Thay vì khởi tạo nhiều đối tượng khi ứng dụng khởi chạy, bạn có thể trì hoãn quá trình khởi tạo đối tượng đến thời điểm thật sự cần thiết



3.Proxy Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

- Protection proxy: Đây là khi bạn muốn những đối tượng thực sự có quyền mới có thể truy cập vào các đối tượng thực và sử dụng các đối tượng đó



3.Proxy Design Pattern

Lợi ích của Factory có gì hơn so với cách sử dụng bình thường?

- Proxy Caching lưu trữ các bản sao của các đối tượng web đã được truy cập ví dụ như(Tài liệu, hình ảnh, bài báo). khi người dùng truy cập lại vào các thông tin này. người dùng sẽ nhận được thông tin nhanh hơn.
- Proxy Caching sẽ lưu trữ các yêu cầu này, khi nhận được các yêu cầu của người dùng thì nó sẽ chuyển tiếp chúng đến các thông tin gốc
 - Fresh Object:Là đối tượng có thể sẵn sàng cung cấp thay cho nội dung gốc. Độ mới được xác định bằng ngày hết hạn hoặc độ tuổi tối đa của đối tượng.
 - Stale Object:Là các đối tượng hết hạn lưu trong bộ nhớ cache nhưng không còn sử dụng được nữa. Các đối tượng này phải được lưu lại từ máy chủ nguồn để làm mới trước khi cung cấp nội dung thay cho nội dung gốc
 - Nonexistent Object:Là đối tượng được lưu trên proxy nhưng không tồn tại trên máy chủ gốc. đối với các đối tượng này thì proxy sẽ tuân theo các trình tự như các đối tượng cũ.

3.Proxy Design Pattern

Ưu điểm

- Bạn có thể kiểm soát đối tượng dịch vụ mà khách hàng không biết về nó.
- Bạn có thể quản lý vòng đời của đối tượng dịch vụ khi khách hàng không quan tâm đến nó.
- Proxy hoạt động ngay cả khi đối tượng dịch vụ chưa sẵn sàng hoặc không có sẵn.
- Nguyên tắc Mở / Đóng. Bạn có thể giới thiệu proxy mới mà không cần thay đổi dịch vụ hoặc ứng dụng khách.

Nhược điểm

- Mã có thể trở nên phức tạp hơn vì bạn cần giới thiệu rất nhiều lớp mới.
- Phản hồi từ dịch vụ có thể bị trì hoãn.