

# 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)

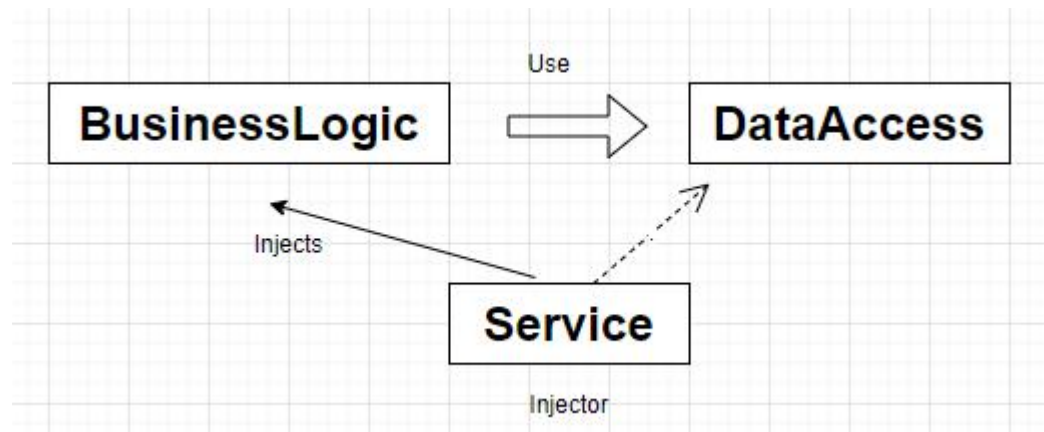
### Phân tích ví dụ

- 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

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

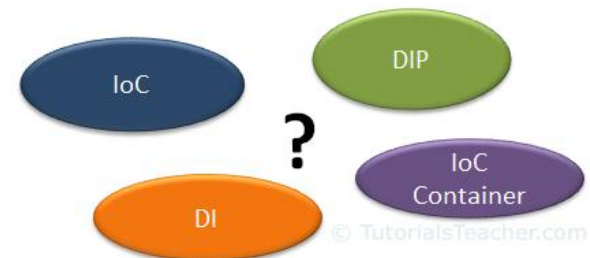
## Một số khái niệm

(1) Inversion of Control (IoC)

(2) Dependency Inversion Principle (DIP)

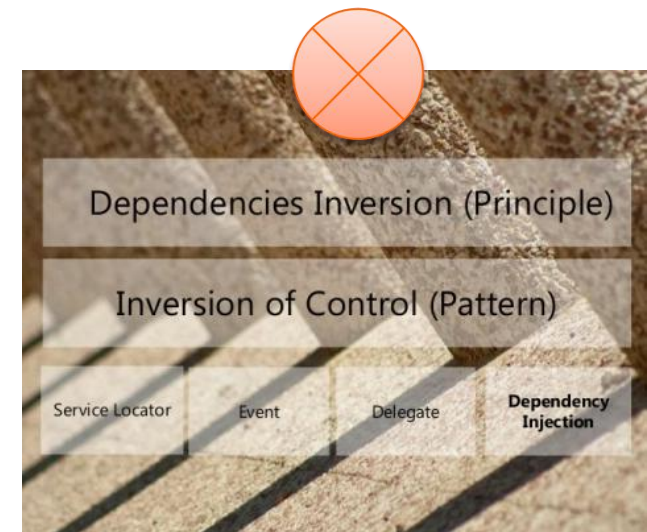
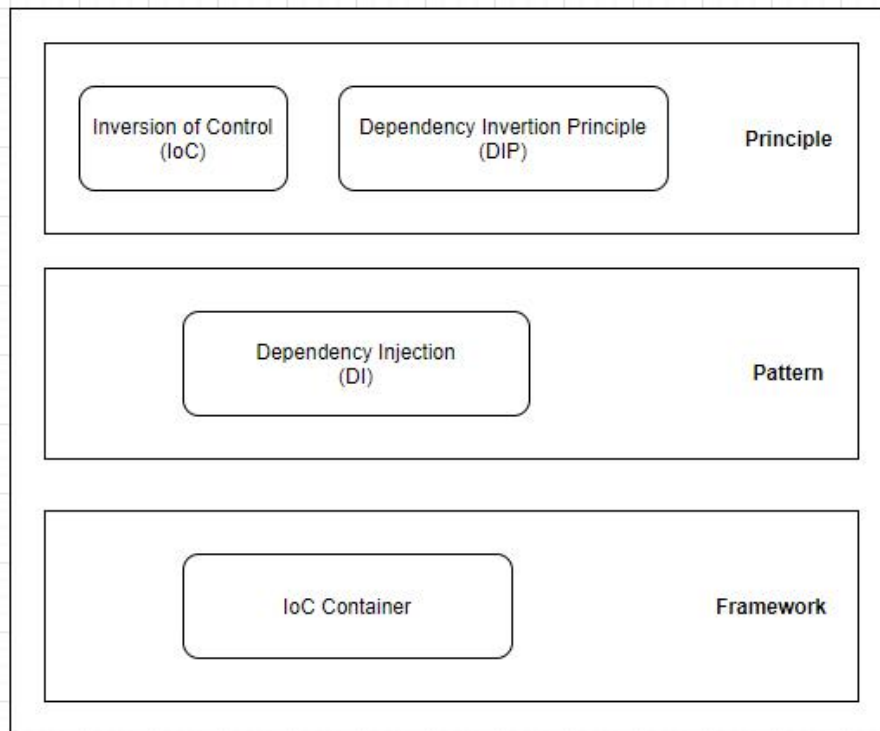
(3) Dependency Injection (DI)

(4) IoC containers



# Tổng quan

## Kiến trúc



# 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

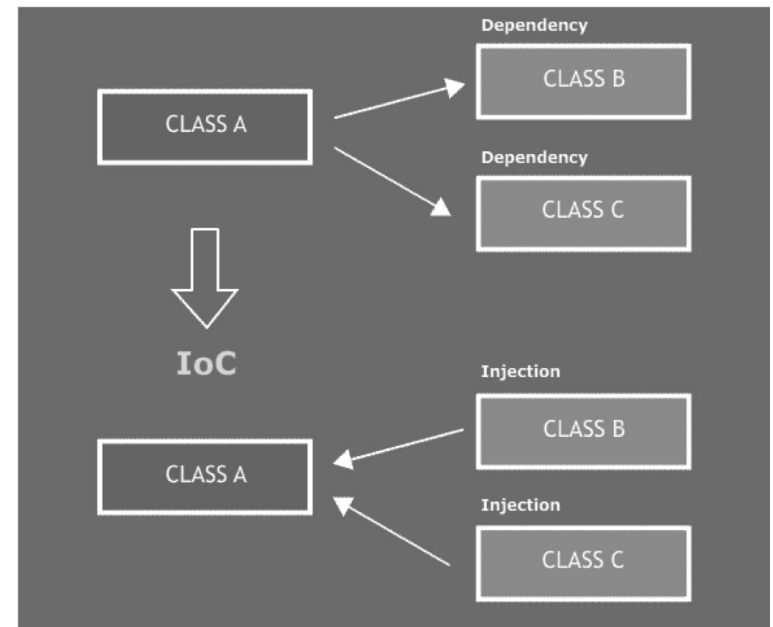
### ❖ Design Pattern

- 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

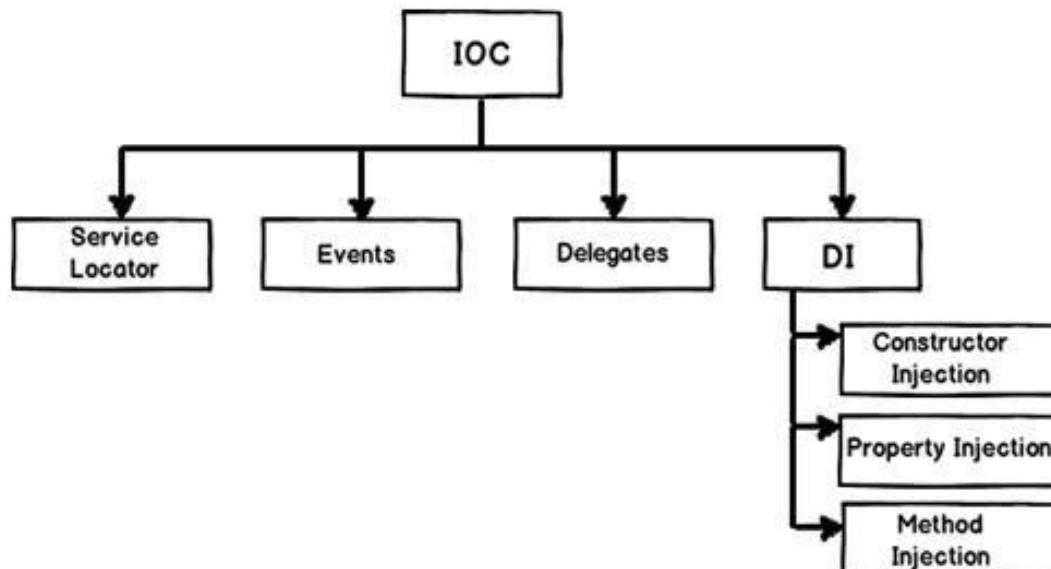
- IOC - Inversion of Control
- Là Design Principle
- Ngược với lập trình thủ tục (procedural programming)
- IOC sử dụng các đối tượng từ các lớp thông qua Injection
- Injection: setter, method, constructor



# 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



# Nội dung

## 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

## ❖ Ví dụ 1: ứng dụng vận tải LogicticsApp

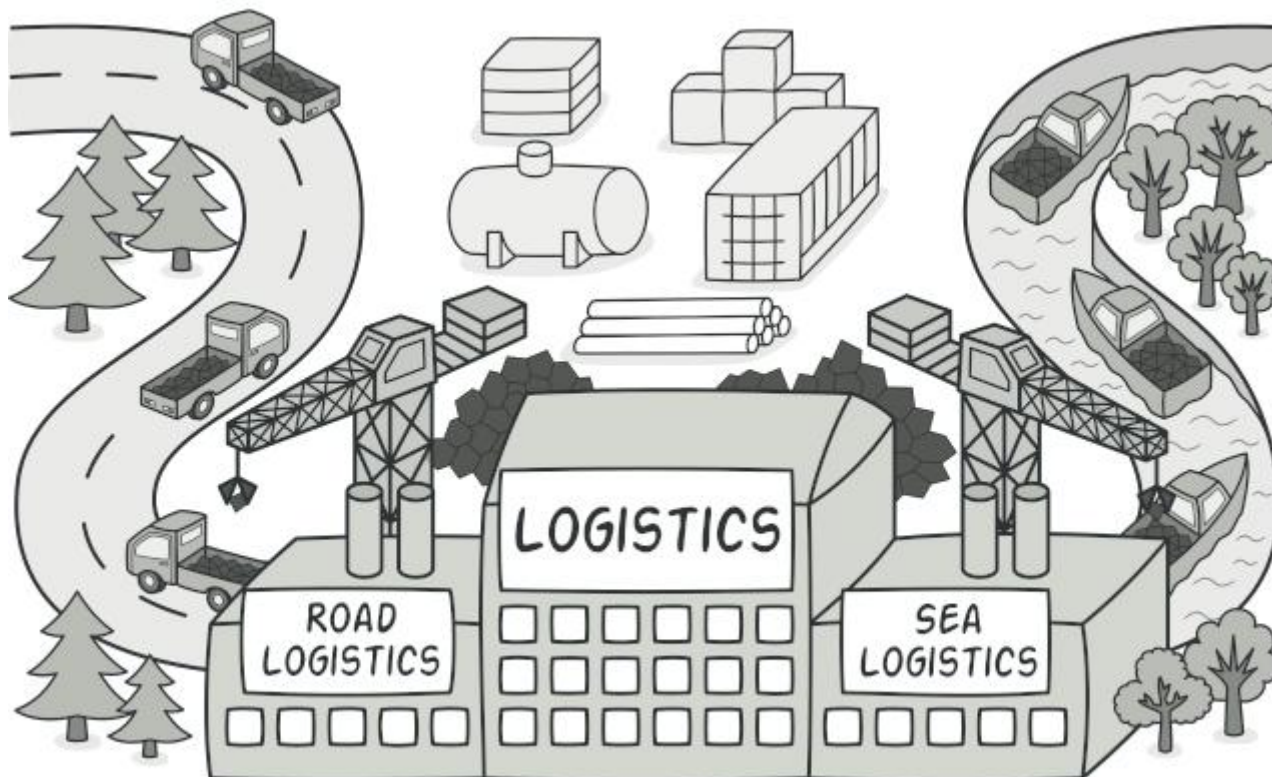
Vận chuyển bằng Truck



# 1.Factory Design Pattern

## ❖ Ví dụ 1: ứng dụng LogisticsApp

Mở rộng ứng dụng vận chuyển bằng Rail, Ship, Airline



# 1.Factory Design Pattern

## ❖ Vấn đề

- Mở rộng ứng dụng hiện có gặp khó khăn khi thay đổi codebase
- Thêm vào một hiện thực của một interface lại thay đổi basecode

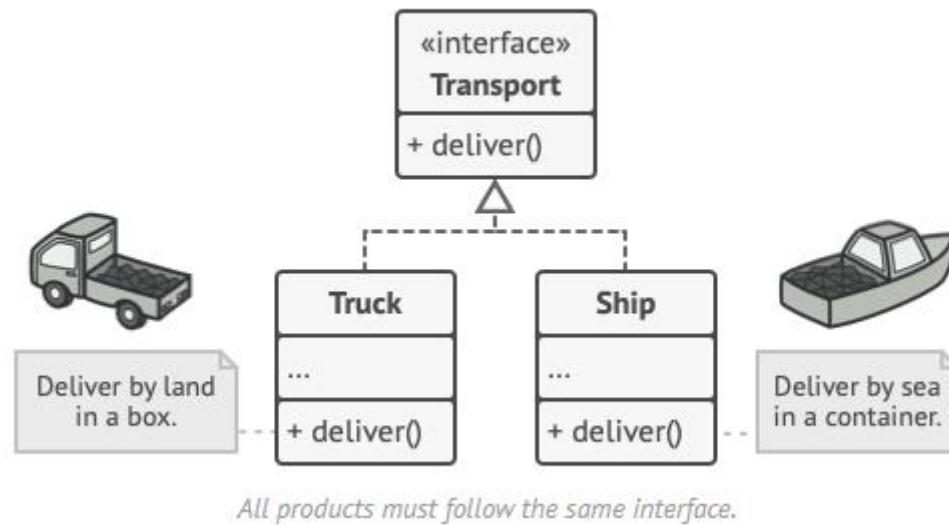
## ❖ Ví dụ

- Ứng dụng vận tải **LogisticsApp** chỉ có thể quản lý vận chuyển bằng **Truck**
- Yêu cầu mở rộng vận chuyển bằng **Ships, Rail, Aairline**
- Vấn đề gặp phải khi mở rộng: mã code xử lý liên quan đến **Truck** => cần thay đổi code nhiều => Khi thêm hình thức vận chuyển => lại thay đổi code

# 1.Factory Design Pattern

## ❖ Ví dụ 1: ứng dụng LogicticsApp

Giải pháp mở rộng phương thức vận chuyển Ship



# 1.Factory Design Pattern

## ❖ Factory

### Mô tả

- 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

## ❖ Factory

### 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.

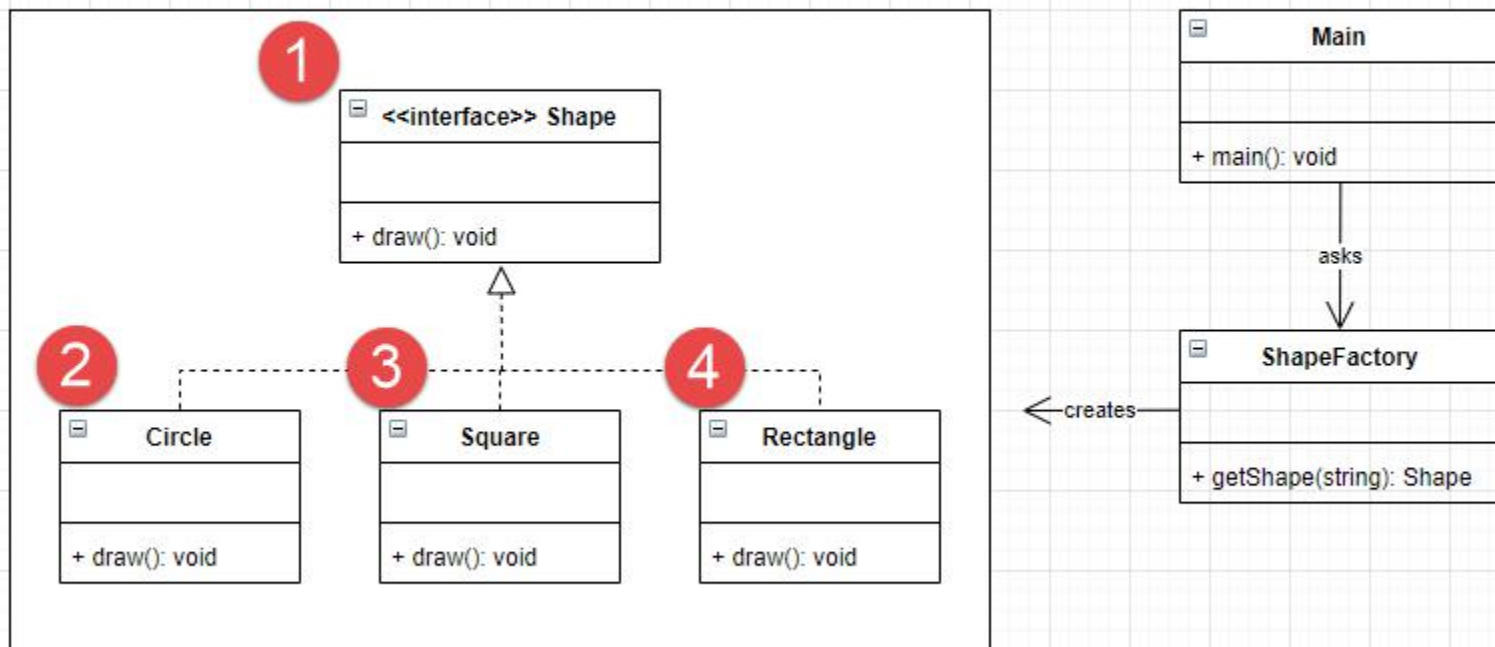


# 1.Factory Design Pattern

## ❖ Ví dụ 2: ứng dụng Shape

Vẽ đa giác

### 1. Factory



# 1.Factory Design Pattern

## ❖ Ví dụ 2: ứng dụng Shape

### Interface Shape và các implements

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

1

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

2

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

3

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

4

# 1.Factory Design Pattern

## ❖ Ví dụ 2: ứng dụng Shape

### FactoryShape

```
3  class ShapeFactory {  
4  
5      public function getShape($shapeType) : ?Shape {  
6          if ($shapeType == null) {  
7              return null;  
8          }  
9          if ($shapeType == "CIRCLE") {  
10             return new Circle();  
11         }  
12         elseif ($shapeType == "RECTANGLE") {  
13             return new Rectangle();  
14         }  
15         elseif ($shapeType == "SQUARE") {  
16             return new Square();  
17         }  
18         return null;  
19     }  
20 }  
21 }
```

# 1.Factory Design Pattern

## ❖ Ví dụ 2: ứng dụng Shape

### Demo

```
16  $shape = new ShapeFactory();
17  $circle = $shape->getShape("CIRCLE");
18  $rectangle = $shape->getShape("RECTANGLE");
19  $square = $shape->getShape("SQUARE");
20
21  $circle->draw() . EOL;
22  $rectangle->draw() . EOL;
23  $square->draw() . EOL;
24
25  //Inside Circle::draw() method.
26  //Inside Rectangle::draw() method.
27  //Inside Square::draw() method.
```

# 1.Factory Design Pattern

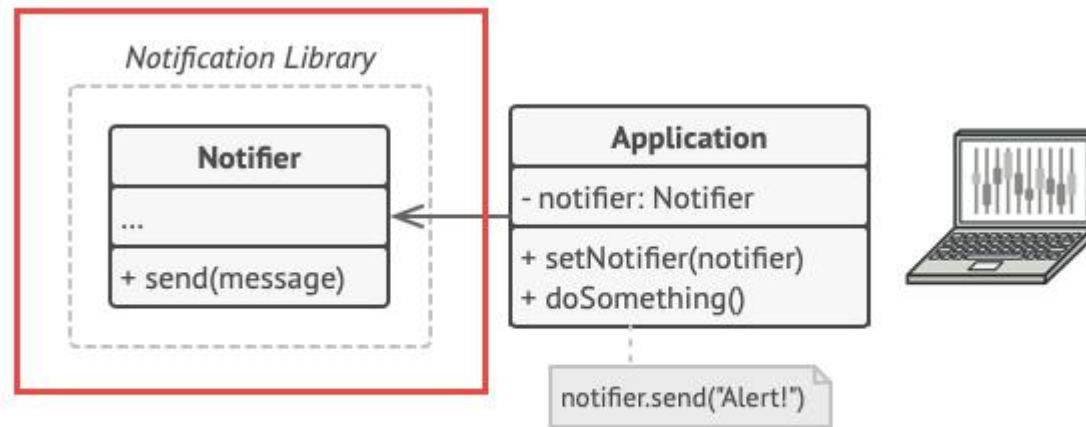
## ❖ Nhận xét: lợi ích khi dùng factory pattern

- Khi mở rộng một chương trình mà không làm thay đổi code của nó
- Factory Pattern phân tách mã nguồn với mã sử dụng đối tượng => phát triển sản phẩm và đối tượng một cách độc lập
- **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

### ❖ Ví dụ 1: ứng dụng gửi tin Notifier

Ứng dụng đang hoạt động với tính năng gửi email tới khách hàng

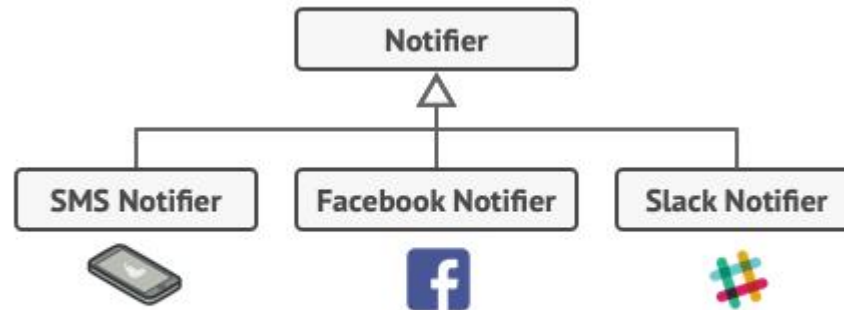


*A program could use the notifier class to send notifications about important events to a predefined set of emails.*

## 2.Decorator Design Pattern

### ❖ Ví dụ 1: ứng dụng gửi tin Notifier

Thông qua các hình thức gửi tin khác



*Each notification type is implemented as a notifier's subclass.*

## 2.Decorator Design Pattern

### ❖ Ví dụ 1: ứng dụng gửi tin Notifier

Giải pháp mở rộng hình thức gửi tin khác

- Subclass (SMS, Facebook, Slack...) extends Notifier
  - Hiện thực phương thức trong các subclass
- => Sẵn sàng để ứng dụng sử dụng
- Phát sinh yêu cầu: dùng vài hay tất cả phương pháp gửi tin cùng 1 lúc?



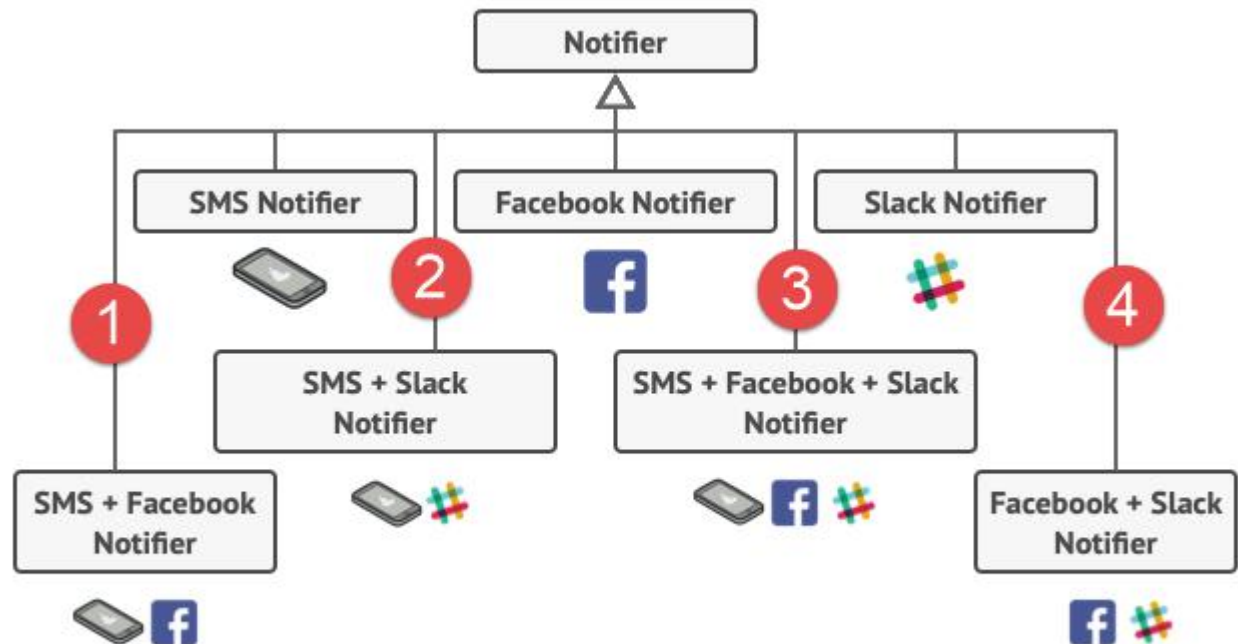
## 2.Decorator Design Pattern

### ❖ Ví dụ 1: ứng dụng gửi tin Notifier

Giải pháp mở rộng hình thức gửi tin khác - Kết hợp (Combinatorial)

Tạo ra các subclasses kết hợp với vài phương thức notification  
=> nhanh chóng đáp ứng

!!!  
Làm cho mã trở nên nhiều hơn



*Combinatorial explosion of subclasses.*

## 2.Decorator Design Pattern

### ❖ Ví dụ 1: ứng dụng gửi tin Notifier

Giải pháp mở rộng hình thức gửi tin khác - thừa kế (inheritance)

- Inheritance là static => không thể thay đổi hành vi của đối tượng lúc runtime => thay bằng object khác từ một subclass khác
- Subclass chỉ có 1 parentclass

=> Sử dụng giải pháp **Aggregation** hay **Composition**

- 1 object có 1 tham khảo đến 1 đối tượng khác
- Ủy quyền thực hiện 1 số hành vi
- Dễ dàng thay đổi đối tượng liên kết bằng 1 đối tượng khác
- Một object có thể sử dụng hành vi của nhiều lớp khác nhau

## 2.Decorator Design Pattern

### ❖Decorator (Wrapper)

- structural design pattern
- 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.
- Tạo ra một object có thể linked với nhiều object khác

## 2.Decorator Design Pattern

### ❖Decorator (Wrapper)

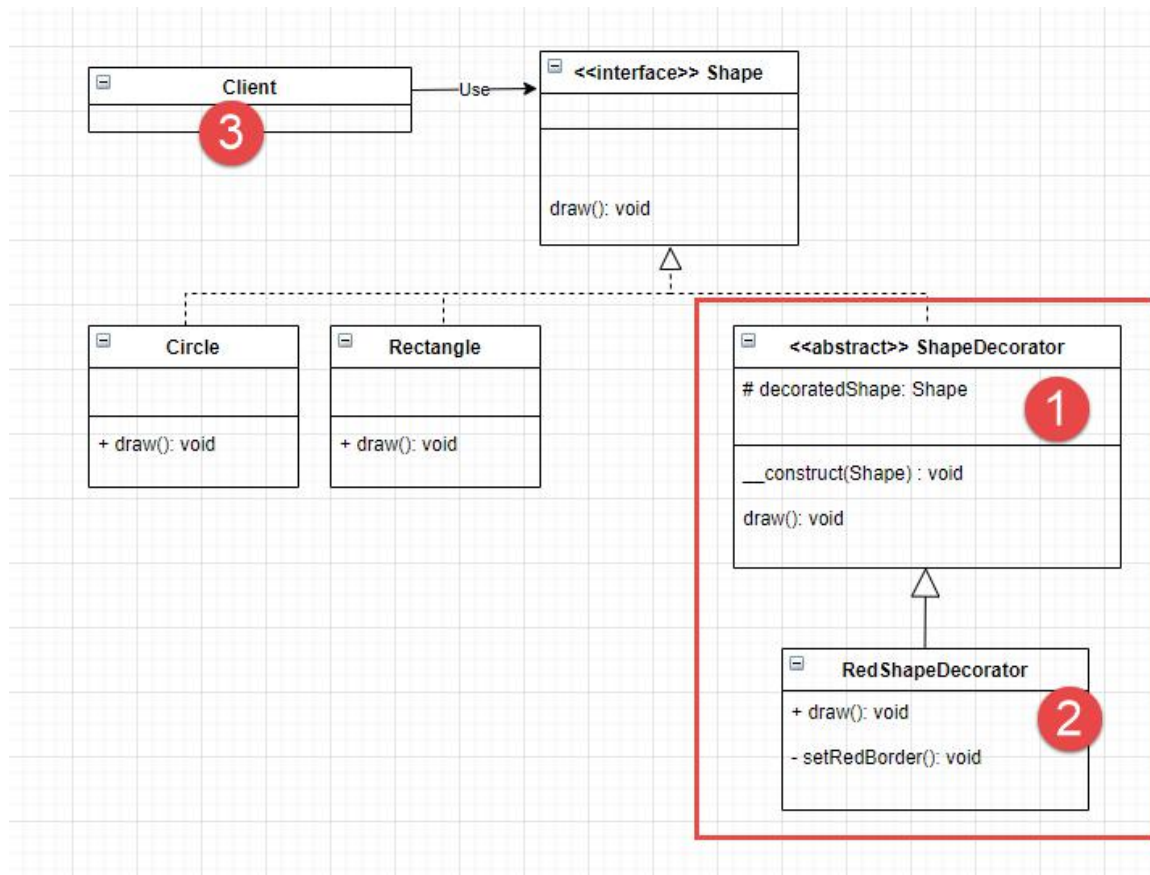
#### Sử dụng khi nào?

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 ta phải tạo ra nhiều lớp kế thừa và mã nguồn trở nên đồ sộ

## 2.Decorator Design Pattern

### ❖ Ví dụ 2: ứng dụng Shape

Vẽ đa giác và tô màu



## 2.Decorator Design Pattern

### ❖ Ví dụ 2: ứng dụng Shape

#### Abstract ShapeDecorator và class RedShapeDecorator

```
6  abstract class ShapeDecorator implements Shape {
7
8
9  protected $decoratedShape;
10
11  function __construct(Shape $decoratedShape) {
12      $this->decoratedShape = $decoratedShape;
13  }
14
15  }
16
17  class RedShapeDecorator extends ShapeDecorator {
18
19
20  function __construct(Shape $decoratedShape) {
21      parent::__construct($decoratedShape);
22  }
23
24  public function draw() {
25      $this->decoratedShape->draw();
26      $this->setRedBorder($this->decoratedShape);
27  }
28
29  private function setRedBorder(Shape $decoratedShape) {
30      echo "Border Color: Red";
31  }
32
33  }
```

1

2

## 2.Decorator Design Pattern

### ❖ Ví dụ 2: ứng dụng Shape

#### Demo

```
8  $circle = new Circle();
9  echo "Circle with normal border: ";
10 echo $circle->draw() . EOL;
11 //Circle with normal border: Shape: Circle
12
13 $redCircle = new RedShapeDecorator(new Circle());
14 echo "Circle of red border: ";
15 echo $redCircle->draw() . EOL;
16 //Circle of red border: Shape: CircleBorder Color: Red
17
18 $redRectangle = new RedShapeDecorator(new Rectangle());
19 echo "Rectangle of red border: ";
20 echo $redRectangle->draw() . EOL;
21 //Rectangle of red border: Shape: RectangleBorder Color: Red
```

## 2.Decorator Design Pattern

### ❖ **Nhận xét: lợi ích khi dùng decorator pattern**

- Có thể thêm các hành vi bổ sung cho đối tượng mà không làm ảnh hưởng đến code
- Có thể dễ dàng thay đổi trở về kiểu decorator ban đầu mà không làm ảnh hưởng đến logic code
- **Ưu điểm**
  - Mở rộng hành vi của một đối tượng mà không cần tạo một lớp con mới.
  - Liên kết với các object trở nên rời rạc nhờ đó có thể dễ dàng liên kết hay tách rời
- **Nhược điểm**
  - Code có thể trở nên phức tạp khi có quá nhiều lớp con