Padrões de Projetos

Padrões de Projetos

Padrões de projeto são soluções reutilizáveis para problemas comuns que os desenvolvedores enfrentam durante o processo de design e desenvolvimento de software. Eles são abstrações de experiências comuns, representando as melhores práticas para resolver determinados tipos de problemas. Os padrões de projeto ajudam a criar código mais organizado, flexível, reutilizável e de fácil manutenção, promovendo a consistência e a eficiência no desenvolvimento de software.

Padrões de Projetos

Padrões de Projetos tem sua origem no trabalho de um arquiteto chamado Christopher Alexander que escreveu dois livros de bastante sucesso onde ele exemplificava o uso e descrevia seu raciocínio para documentar os padrões para a arquitetura. Porém em 1995 um grupo de quatro profissionais basearam-se em Christopher Alexander e escreveram o livro "Design Patterns: Elements of Reusable Object-Oriented Software" [Gamma95] que continha um catálogo com 23 padrões de projetos (Design Patterns) orientados a software. A idéia dos autores do livro era documentar problemas recorrentes que acontecia nos softwares.

Padrões de Projetos - Criação

Singleton: Usado para garantir que uma classe tenha apenas uma instância e forneça um ponto global de acesso a essa instância.

Factory Method: Utilizado para criar objetos de várias subclasses de uma classe base, permitindo que a classe base delegue a responsabilidade de criar objetos para suas subclasses.

Abstract Factory: Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Factory

```php
You, last month | 1 author (You)
interface Connection {
    public function connect();
}

You, last month | 1 author (You)
class MySQLConnection implements Connection {
    public function connect() {
        return "Conectado ao MySQL";
    }
}

You, last month | 1 author (You)
class PostgreSQLConnection implements Connection {
    public function connect() {
        return "Conectado ao PostgreSQL";
    }
}
```

```php
You, last month | 1 author (You)
abstract class ConnectionFactory {
    abstract public function createConnection(): Connection;
}

You, last month | 1 author (You)
class MySQLFactory extends ConnectionFactory {
    public function createConnection(): Connection {
        return new MySQLConnection();
    }
}

You, last month | 1 author (You)
class PostgreSQLFactory extends ConnectionFactory {
    public function createConnection(): Connection {
        return new PostgreSQLConnection();
    }
}

// Uso:
$factory = new MySQLFactory();
$connection = $factory->createConnection();
echo $connection->connect(); // "Conectado ao MySQL"
```

Padrões de Projetos - Estruturais

Adapter: Permite que classes com interfaces incompatíveis trabalhem juntas, envolvendo uma interface em outra para fazer com que elas funcionem juntas.

Decorator: Permite adicionar responsabilidades a objetos de forma dinâmica, oferecendo uma alternativa à criação de subclasses para estender funcionalidades.

Facade: Fornece uma interface simplificada para um conjunto de interfaces mais complexo em um subsistema, facilitando a interação do cliente com o sistema.

Adapter

```php
// Classe antiga com interface incompatível
You, last month | 1 author (You)
class OldEmailSender {
    public function send($to, $subject, $body) {
        return "Email enviado para $to: $subject - $body";
    }
}

// Nova interface desejada
You, last month | 1 author (You)
interface EmailService {
    public function sendEmail($recipient, $message);
}
```

```php
// Nova interface desejada
You, last month | 1 author (You)
interface EmailService {       You, last month • Uncommitted changes
    public function sendEmail($recipient, $message);
}

// Adaptador
You, last month | 1 author (You)
class EmailAdapter implements EmailService {
    private $oldSender;

    public function __construct(OldEmailSender $oldSender) {
        $this->oldSender = $oldSender;
    }

    public function sendEmail($recipient, $message) {
        return $this->oldSender->send($recipient, "Notificação", $message);
    }
}

// Uso:
$adapter = new EmailAdapter(new OldEmailSender());
echo $adapter->sendEmail("user@test.com", "Olá!"); // Adapta a chamada
```

Decorator

```
You, last month | 1 author (You)
interface Coffee {
    public function getCost(): float;
    public function getDescription(): string;
}

You, last month | 1 author (You)
class SimpleCoffee implements Coffee {
    public function getCost(): float {
        return 5.0;
    }

    public function getDescription(): string {
        return "Café simples";
    }
}
```

```
You, last month | 1 author (You)
class MilkDecorator implements Coffee {
    private $coffee;

    public function __construct(Coffee $coffee) {      You, last m
        $this->coffee = $coffee;
    }

    public function getCost(): float {
        return $this->coffee->getCost() + 2.0;
    }

    public function getDescription(): string {
        return $this->coffee->getDescription() . ", com leite";
    }
}

// Uso:
$coffee = new SimpleCoffee();
$coffee = new MilkDecorator($coffee);
echo $coffee->getDescription(); // "Café simples, com leite"
```

Facade

```php
// Subsistemas complexos
...
class Estoque {
    public function verificarDisponibilidade(string $produto): bool {
        // Lógica complexa de verificação
        return true; // Simulado
    }
}

...
class Pagamento {
    public function processarPagamento(float $valor): bool {
        // Lógica complexa de pagamento
        return true; // Simulado
    }
}

...
class Notificacao {
    public function enviarEmail(string $cliente, string $mensagem): void {
        echo "Email para $cliente: $mensagem\n";
    }
}
```

```php
// Facade (interface simplificada)
You, last month | 1 author (You)
class SistemaPedidosFacade {
    private $estoque;
    private $pagamento;
    private $notificacao;

    public function __construct() {
        $this->estoque = new Estoque();
        $this->pagamento = new Pagamento();
        $this->notificacao = new Notificacao();
    }

    public function processarPedido(string $produto, float $valor, string $cliente): void {
        if ($this->estoque->verificarDisponibilidade($produto)) {
            if ($this->pagamento->processarPagamento($valor)) {
                $this->notificacao->enviarEmail(
                    $cliente,
                    "Pedido de $produto (R$$valor) processado!"
                );
            }
        }
    }
}

// Uso pelo cliente (simples!)
$facade = new SistemaPedidosFacade();
$facade->processarPedido("Livro PHP", 99.90, "joao@email.com");
```

Padrões de Projetos - Comportamentais

Observer: Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

Strategy: Define uma família de algoritmos, encapsulando-os de forma que sejam intercambiáveis. Isso permite que o algoritmo seja selecionado e alterado durante a execução do programa.

Chain of Responsibility: Cria uma cadeia de objetos que podem processar solicitações sequencialmente, passando a solicitação de um objeto para outro até que ela seja tratada.

Observer

```
You, last month | 1 author (You)
class Newsletter {
    private $subscribers = [];

    public function subscribe($subscriber) {
        $this->subscribers[] = $subscriber;
    }

    public function notify($message) {
        foreach ($this->subscribers as $subscriber) {
            $subscriber->update($message);
        }
    }
}

You, last month | 1 author (You)
class User {
    public function update($message) {
        echo "Usuário notificado: $message\n";
    }
}

// Uso:
$newsletter = new Newsletter();
$newsletter->subscribe(new User());
$newsletter->notify("Novo artigo publicado!");
```

Strategy

```php
You, last month | 1 author (You)
interface PaymentMethod {
    public function pay($amount);
}

You, last month | 1 author (You)
class CreditCardPayment implements PaymentMethod {
    public function pay($amount) {
        return "Pagamento de R$$amount via Cartão de Crédito";
    }
}

You, last month | 1 author (You)
class PayPalPayment implements PaymentMethod {
    public function pay($amount) {
        return "Pagamento de R$$amount via PayPal";
    }
}
```

```php
You, last month • Uncommitted changes
You, last month | 1 author (You)
class Checkout {
    private $paymentMethod;

    public function setPaymentMethod(PaymentMethod $method) {
        $this->paymentMethod = $method;
    }

    public function finishPayment($amount) {
        return $this->paymentMethod->pay($amount);
    }
}

// Uso:
$checkout = new Checkout();
$checkout->setPaymentMethod(new PayPalPayment());
echo $checkout->finishPayment(100); // "Pagamento de R$100 via PayPal"
```

Chain of Responsibility

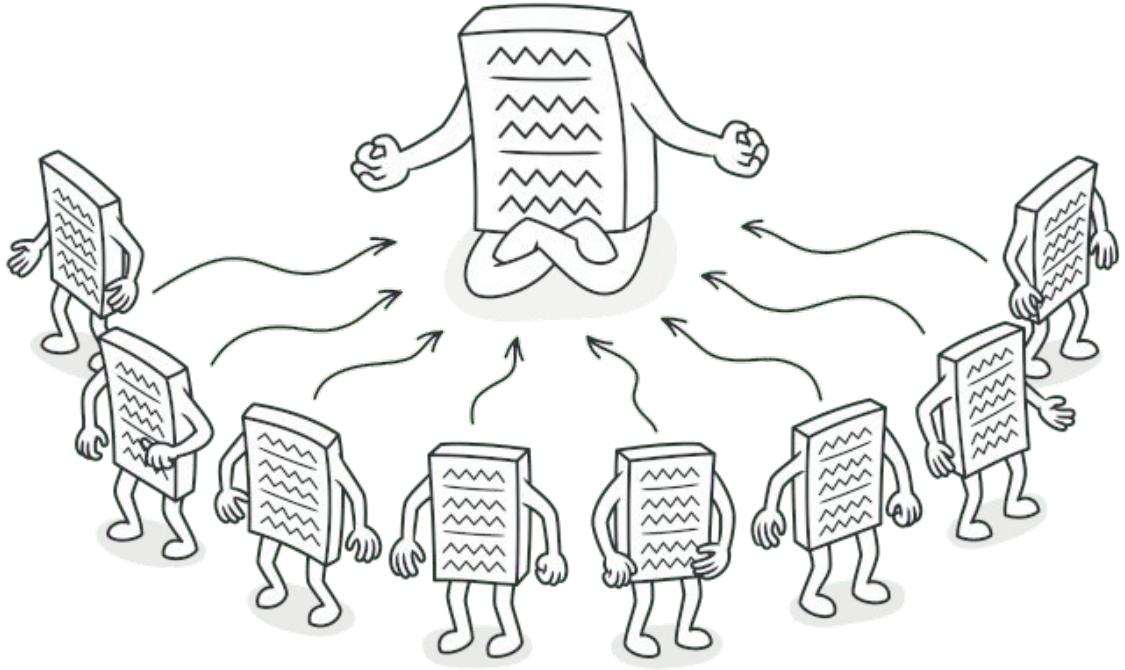
```php
You, last month | 1 author (You)
abstract class Middleware {       You, last month • U
    private $next;

    public function setNext(Middleware $next) {
        $this->next = $next;
    }

    public function handle($request) {
        if ($this->next) {
            return $this->next->handle($request);
        }
        return true;
    }
}
```

```php
You, last month | 1 author (You)
class AuthenticationMiddleware extends Middleware {
    public function handle($request) {
        if ($request !== "autenticado") {
            return "Erro: Autenticação falhou!";
        }
        return parent::handle($request);
    }
}

You, last month | 1 author (You)
class LoggingMiddleware extends Middleware {
    public function handle($request) {
        echo "Log: Requisição recebida\n";
        return parent::handle($request);
    }
}
```

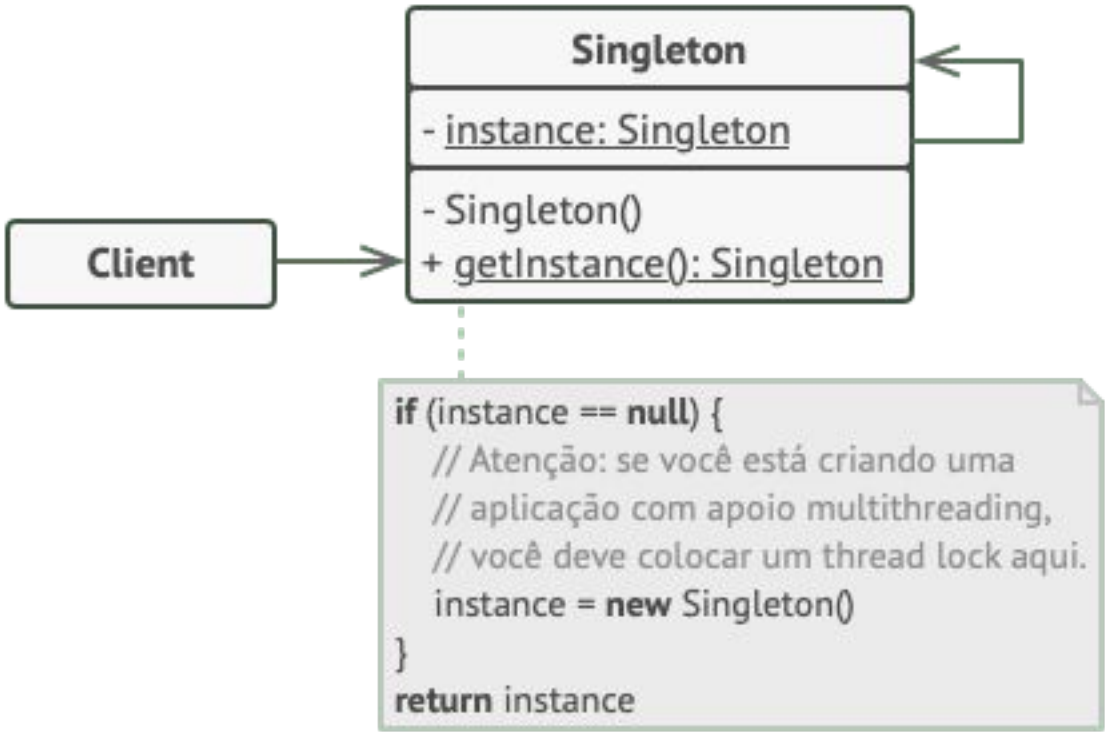
```php
// Uso:
$chain = new AuthenticationMiddleware();
$chain->setNext(new LoggingMiddleware());

echo $chain->handle("autenticado"); // Passa pela cadeia
```

Padrão Singleton

Permite a você garantir que uma classe tem apenas uma instância, enquanto provê um ponto de acesso global para esta instância.



Padrão Singleton - Estrutura



Padrão Singleton - Código

```php
class Singleton {
  private static $instance = null;

  private function __construct() {
    // Construtor privado para impedir a criação de instâncias fora da classe.
  }

  public static function getInstance() {
    if (self::$instance == null) {
      self::$instance = new Singleton();
    }

    return self::$instance;
  }
}

$singleton = Singleton::getInstance();        You, 1 second ago • Uncommitted changes
```

Padrão Singleton - Código

```php
class Logger {
  private static $instance = null;
  private $logFile = null;

  private function __construct() {
    $this->logFile = fopen('log.txt', 'a');
  }

  public static function getInstance() {
    if (self::$instance == null) {
      self::$instance = new Logger();
    }

    return self::$instance;
  }

  public function log($message) {
    fwrite($this->logFile, $message . "\n");
  }

  public function __destruct() {
    fclose($this->logFile);
  }
}
```

```php
$logger = Logger::getInstance();
$logger->log("Primeira mensagem de log");
$logger->log("Segunda mensagem de log");
```