Gang of Four Design Patterns: A Comprehensive Guide

Introduction

The Gang of Four (GoF) design patterns are a set of 23 design patterns introduced in the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These patterns provide solutions to common software design problems and promote code reuse, flexibility, and maintainability.

Core Design Principles

1. Open/Closed Principle (OCP)

- Software entities should be open for extension but closed for modification
- Example: Use inheritance or interfaces to extend functionality without changing existing code

```
class PaymentGateway {
    public function processPayment($amount) {
        // Default processing logic
    }
}

class PayPalGateway extends PaymentGateway {
    public function processPayment($amount) {
        // Custom PayPal processing logic
    }
}
```

2. Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules
- Both should depend on abstractions (interfaces)
- Use dependency injection to inject interfaces rather than concrete classes

3. Composition Over Inheritance

- Prefer composition over inheritance to create flexible objects
- Use services or traits for modular functionality instead of large monolithic classes

4. Interface Segregation Principle (ISP)

- Clients should not be forced to depend on interfaces they don't use
- Create specific interfaces for each type of service

5. Principle of Least Knowledge (Law of Demeter)

- An object should only communicate with its immediate neighbors
- Use services or repositories to hide internal dependencies

Design Pattern Categories

1. Creational Patterns

Singleton Pattern

- **Purpose**: Ensures a class has only one instance with global access
- **Example**:

```
public function register() {
    $this->app->singleton(MyService::class, function ($app) {
        return new MyService();
    });
}
```

Factory Method Pattern

- **Purpose**: Defines interface for object creation, letting subclasses decide which class to instantiate
- **Example**:

```
$user = UserFactory::create(['name' => 'John Doe']);
```

Abstract Factory Pattern

- **Purpose**: Creates families of related objects without specifying concrete classes

```
interface PaymentGatewayFactory {
    public function createPaymentProcessor();
    public function createRefundProcessor();
}

class StripeFactory implements PaymentGatewayFactory {
    public function createPaymentProcessor() {
        return new StripePayment();
    }

    public function createRefundProcessor() {
        return new StripeRefund();
    }
}
```

2. Structural Patterns

Facade Pattern

- **Purpose**: Provides simplified interface to complex subsystem
- **Example**: Laravel's DB, Cache, and Route facades

```
Cache::put('key', 'value', 10);
```

Adapter Pattern

- **Purpose**: Makes incompatible interfaces work together

```
class ThirdPartyAdapter implements PaymentInterface {
    protected $thirdParty;
```

```
    public function __construct(ThirdPartyPaymentGateway $thirdParty) {
        $this->thirdParty = $thirdParty;
    }

    public function processPayment($amount) {
        return $this->thirdParty->pay($amount);
    }
}
```

Composite Pattern

- **Purpose**: Composes objects into tree structures for part-whole hierarchies

```
interface FileComponent {
    public function getSize();
}

class File implements FileComponent {
    public function getSize() {
        return 10; // size in KB
    }
}

class Directory implements FileComponent {
    protected $files = [];

    public function addFile(FileComponent $file) {
        $this->files[] = $file;
    }

    public function getSize() {
        return array_sum(array_map(function($file) {
            return $file->getSize();
        }, $this->files));
    }
}
```

Proxy Pattern

- **Purpose**: Provides a surrogate or placeholder for another object

```
class UserDataProxy {
    protected $realUser;

    public function __construct(User $user) {
        $this->realUser = $user;
    }

    public function getUserData() {
        return Cache::remember('user_data_' . $this->realUser->id, 60,
function () {
            return $this->realUser->load('profile');
```

```
        });
    }
}
```

Decorator Pattern

- **Purpose**: Adds functionality to objects dynamically

```
interface UserInterface {
    public function getName();
}

class User implements UserInterface {
    public function getName() {
        return 'John Doe';
    }
}

class UserDecorator implements UserInterface {
    protected $user;

    public function __construct(UserInterface $user) {
        $this->user = $user;
    }

    public function getName() {
        return 'Mr. ' . $this->user->getName();
    }
}
```

3. Behavioral Patterns

Template Method Pattern

- **Purpose**: Defines algorithm skeleton, letting subclasses override specific steps

```
abstract class DataProcessor {
    public function processData() {
        $this->loadData();
        $this->transformData();
        $this->saveData();
    }

    abstract protected function loadData();
    abstract protected function transformData();
    abstract protected function saveData();
}
```

Chain of Responsibility Pattern

- **Purpose**: Passes request along handler chain until handled

```
interface Handler {
    public function handle(Request $request);
}
```

```php
class AuthenticationHandler implements Handler {
    public function handle(Request $request) {
        if (!$request->isAuthenticated()) {
            return "Authentication failed.";
        }
        return $next($request);
    }
}
```

State Pattern

- **Purpose**: Alters object behavior when internal state changes

```php
interface OrderState {
    public function handle(Order $order);
}

class PendingState implements OrderState {
    public function handle(Order $order) {
        echo "Order is pending.";
    }
}

class ShippedState implements OrderState {
    public function handle(Order $order) {
        echo "Order is shipped.";
    }
}
```

Command Pattern

- **Purpose**: Encapsulates request as object

```php
interface Command {
    public function execute();
}

class CreateUserCommand implements Command {
    protected $name;

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

    public function execute() {
        return User::create(['name' => $this->name]);
    }
}
```

Observer Pattern

- **Purpose**: Objects notify observers of state changes

```php
class UserRegistered {
    public $user;

    public function __construct(User $user) {
        $this->user = $user;
    }
}

class SendWelcomeEmail {
    public function handle(UserRegistered $event) {
        Mail::to($event->user)->send(new WelcomeMail());
    }
}
```

4. MVC Pattern Implementation

- **Model**: Represents data and business logic

```php
class User extends Model {
    protected $fillable = ['name', 'email'];
}
```

- **View**: Displays data to user

```php
<!-- user.blade.php -->
<h1>{{ $user->name }}</h1>
```

- **Controller**: Handles logic between Model and View

```php
class UserController extends Controller {
    public function show($id) {
        $user = User::find($id);
        return view('user', compact('user'));
    }
}
```

Common Anti-Patterns and Code Smells

Code Organization Issues

1. **Large Classes**
   - Symptom: Classes with too many responsibilities
   - Solution: Break into smaller, focused classes
2. **Long Methods**
   - Symptom: Methods performing multiple actions
   - Solution: Extract into smaller, single-purpose methods
3. **Duplicate Code**
   - Symptom: Repeated code blocks
   - Solution: Extract into reusable methods or services

Design Issues

1. **Primitive Obsession**

- Symptom: Overuse of primitive types instead of objects
- Solution: Create value objects

```
class Email {
 private $email;

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

 public function getEmail() {
     return $this->email;
 }
}
```

2. **Feature Envy**
   - Symptom: Class too interested in another class's details
   - Solution: Move behavior to data-owning class
3. **Shotgun Surgery**
   - Symptom: Single change requires multiple class modifications
   - Solution: Centralize functionality, reduce dependencies
4. **Data Clumps**
   - Symptom: Groups of related data passed together
   - Solution: Create dedicated classes for related data
5. **Message Chains**
   - Symptom: Long chains of method calls
   - Solution: Use method chaining or introduce intermediate methods
6. **Inappropriate Intimacy**
   - Symptom: Classes know too much about each other
   - Solution: Reduce coupling through proper encapsulation
7. **Switch Statements**
   - Symptom: Complex conditional logic
   - Solution: Use polymorphism and inheritance
8. **Refused Bequest**
   - Symptom: Subclass doesn't use inherited methods
   - Solution: Reorganize inheritance hierarchy

Best Practices for Implementation

1. **Start Simple**
   - Don't over-engineer initially
   - Apply patterns when complexity warrants them
2. **Document Intentions**
   - Explain why a pattern was chosen
   - Document any constraints or assumptions

3. **Maintain Flexibility**
   - Design for change
   - Use interfaces and abstractions appropriately
4. **Consider Context**
   - Not every problem needs a design pattern
   - Choose patterns based on specific requirements
5. **Review and Refactor**
   - Regularly review implementation
   - Refactor when patterns no longer fit
6. **Test Coverage**
   - Ensure comprehensive testing
   - Test pattern behavior and edge cases

Conclusion

Design patterns are powerful tools for creating maintainable and flexible software, but they should be used judiciously. Understanding both patterns and anti-patterns helps in making better architectural decisions and writing cleaner code. The key is to apply patterns where they add value and solve specific problems, rather than using them simply because they exist.

Remember that patterns are guidelines, not rules. They should be adapted to fit your specific context and requirements. Regular code reviews and refactoring help ensure patterns remain effective as your application evolves.