

# DESIGN PATTERNS FOR HUMANS!

---

🧠 Ultra-simplified explanation to design patterns! 🧠

A topic that can easily make anyone's mind wobble. Here I try to make them stick in to your mind (and maybe mine) by explaining them in the *simplest* way possible.

---

Check out my [blog](#) and say "hi" on [Twitter](#).

## Introduction

---

Design patterns are solutions to recurring problems; **guidelines on how to tackle certain problems**. They are not classes, packages or libraries that you can plug into your application and wait for the magic to happen. These are, rather, guidelines on how to tackle certain problems in certain situations.

Design patterns are solutions to recurring problems; guidelines on how to tackle certain problems

Wikipedia describes them as

In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

## ⚠ Be Careful

---

- Design patterns are not a silver bullet to all your problems.
- Do not try to force them; bad things are supposed to happen, if done so.
- Keep in mind that design patterns are solutions **to** problems, not solutions **finding** problems; so don't overthink.
- If used in a correct place in a correct manner, they can prove to be a savior; or else they can result in a horrible mess of a code.

Also note that the code samples below are in PHP-7, however this shouldn't stop you because the concepts are same anyways.

## Types of Design Patterns

---

- [Creational](#)

- [Structural](#)
- [Behavioral](#)

# Creational Design Patterns

---

In plain words

Creational patterns are focused towards how to instantiate an object or group of related objects.

Wikipedia says

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

- [Simple Factory](#)
- [Factory Method](#)
- [Abstract Factory](#)
- [Builder](#)
- [Prototype](#)
- [Singleton](#)

## Simple Factory

---

Real world example

Consider, you are building a house and you need doors. You can either put on your carpenter clothes, bring some wood, glue, nails and all the tools required to build the door and start building it in your house or you can simply call the factory and get the built door delivered to you so that you don't need to learn anything about the door making or to deal with the mess that comes with making it.

In plain words

Simple factory simply generates an instance for client without exposing any instantiation logic to the client

Wikipedia says

In object-oriented programming (OOP), a factory is an object for creating other objects – formally a factory is a function or method that returns objects of a varying prototype or class from some method call, which is assumed to be "new".

### Programmatic Example

First of all we have a door interface and the implementation

```
interface Door
{
    public function getWidth(): float;
    public function getHeight(): float;
}
```

```

class WoodenDoor implements Door
{
    protected $width;
    protected $height;

    public function __construct(float $width, float $height)
    {
        $this->width = $width;
        $this->height = $height;
    }

    public function getWidth(): float
    {
        return $this->width;
    }

    public function getHeight(): float
    {
        return $this->height;
    }
}

```

Then we have our door factory that makes the door and returns it

```

class DoorFactory
{
    public static function makeDoor($width, $height): Door
    {
        return new WoodenDoor($width, $height);
    }
}

```

And then it can be used as

```

// Make me a door of 100x200
$door = DoorFactory::makeDoor(100, 200);

echo 'width: ' . $door->getWidth();
echo 'height: ' . $door->getHeight();

// Make me a door of 50x100
$door2 = DoorFactory::makeDoor(50, 100);

```

### When to Use?

When creating an object is not just a few assignments and involves some logic, it makes sense to put it in a dedicated factory instead of repeating the same code everywhere.



## Factory Method

Real world example

Consider the case of a hiring manager. It is impossible for one person to interview for each of the positions. Based on the job opening, she has to decide and delegate the interview steps to different people.

In plain words

It provides a way to delegate the instantiation logic to child classes.

Wikipedia says

In class-based programming, the factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

### Programmatic Example

Taking our hiring manager example above. First of all we have an interviewer interface and some implementations for it

```
interface Interviewer
{
    public function askQuestions();
}

class Developer implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about design patterns!';
    }
}

class CommunityExecutive implements Interviewer
{
    public function askQuestions()
    {
        echo 'Asking about community building';
    }
}
```

Now let us create our `HiringManager`

```

abstract class HiringManager
{
    // Factory method
    abstract protected function makeInterviewer(): Interviewer;

    public function takeInterview()
    {
        $interviewer = $this->makeInterviewer();
        $interviewer->askQuestions();
    }
}

```

Now any child can extend it and provide the required interviewer

```

class DevelopmentManager extends HiringManager
{
    protected function makeInterviewer(): Interviewer
    {
        return new Developer();
    }
}

class MarketingManager extends HiringManager
{
    protected function makeInterviewer(): Interviewer
    {
        return new CommunityExecutive();
    }
}

```

and then it can be used as

```

$devManager = new DevelopmentManager();
$devManager->takeInterview(); // Output: Asking about design patterns

$marketingManager = new MarketingManager();
$marketingManager->takeInterview(); // Output: Asking about community building.

```

### When to use?

Useful when there is some generic processing in a class but the required sub-class is dynamically decided at runtime. Or putting it in other words, when the client doesn't know what exact sub-class it might need.

## Abstract Factory

Real world example

Extending our door example from Simple Factory. Based on your needs you might get a wooden door from a wooden door shop, iron door from an iron shop or a PVC door from the relevant shop. Plus you might need a guy with different kind of specialities to fit the door, for example a carpenter for wooden door, welder for iron door etc. As you can see there is a dependency between the doors now, wooden door needs carpenter, iron door needs a welder etc.

In plain words

A factory of factories; a factory that groups the individual but related/dependent factories together without specifying their concrete classes.

Wikipedia says

The abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes

### Programmatic Example

Translating the door example above. First of all we have our `Door` interface and some implementation for it

```
interface Door
{
    public function getDescription();
}

class WoodenDoor implements Door
{
    public function getDescription()
    {
        echo 'I am a wooden door';
    }
}

class IronDoor implements Door
{
    public function getDescription()
    {
        echo 'I am an iron door';
    }
}
```

Then we have some fitting experts for each door type

```
interface DoorFittingExpert
{
    public function getDescription();
}

class Welder implements DoorFittingExpert
{
    public function getDescription()
    {
        echo 'I can only fit iron doors';
    }
}
```

```

    }
}

class Carpenter implements DoorFittingExpert
{
    public function getDescription()
    {
        echo 'I can only fit wooden doors';
    }
}

```

Now we have our abstract factory that would let us make family of related objects i.e. wooden door factory would create a wooden door and wooden door fitting expert and iron door factory would create an iron door and iron door fitting expert

```

interface DoorFactory
{
    public function makeDoor(): Door;
    public function makeFittingExpert(): DoorFittingExpert;
}

// Wooden factory to return carpenter and wooden door
class WoodenDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
        return new WoodenDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Carpenter();
    }
}

// Iron door factory to get iron door and the relevant fitting expert
class IronDoorFactory implements DoorFactory
{
    public function makeDoor(): Door
    {
        return new IronDoor();
    }

    public function makeFittingExpert(): DoorFittingExpert
    {
        return new Welder();
    }
}

```

And then it can be used as

```

$woodenFactory = new WoodenDoorFactory();

$door = $woodenFactory->makeDoor();
$expert = $woodenFactory->makeFittingExpert();

$door->getDescription(); // Output: I am a wooden door
$expert->getDescription(); // Output: I can only fit wooden doors

// Same for Iron Factory
$ironFactory = new IronDoorFactory();

$door = $ironFactory->makeDoor();
$expert = $ironFactory->makeFittingExpert();

$door->getDescription(); // Output: I am an iron door
$expert->getDescription(); // Output: I can only fit iron doors

```

As you can see the wooden door factory has encapsulated the `carpenter` and the `wooden door` also iron door factory has encapsulated the `iron door` and `welder`. And thus it had helped us make sure that for each of the created door, we do not get a wrong fitting expert.

### When to use?

When there are interrelated dependencies with not-that-simple creation logic involved



## Builder

### Real world example

Imagine you are at Hardee's and you order a specific deal, lets say, "Big Hardee" and they hand it over to you without *any questions*; this is the example of simple factory. But there are cases when the creation logic might involve more steps. For example you want a customized Subway deal, you have several options in how your burger is made e.g what bread do you want? what types of sauces would you like? What cheese would you want? etc. In such cases builder pattern comes to the rescue.

### In plain words

Allows you to create different flavors of an object while avoiding constructor pollution. Useful when there could be several flavors of an object. Or when there are a lot of steps involved in creation of an object.

### Wikipedia says

The builder pattern is an object creation software design pattern with the intentions of finding a solution to the telescoping constructor anti-pattern.

Having said that let me add a bit about what telescoping constructor anti-pattern is. At one point or the other we have all seen a constructor like below:

```

public function __construct($size, $cheese = true, $pepperoni = true, $tomato = false,
    $lettuce = true)
{
}

```



As you can see; the number of constructor parameters can quickly get out of hand and it might become difficult to understand the arrangement of parameters. Plus this parameter list could keep on growing if you would want to add more options in future. This is called telescoping constructor anti-pattern.

### Programmatic Example

The sane alternative is to use the builder pattern. First of all we have our burger that we want to make

```
class Burger
{
    protected $size;

    protected $cheese = false;
    protected $pepperoni = false;
    protected $lettuce = false;
    protected $tomato = false;

    public function __construct(BurgerBuilder $builder)
    {
        $this->size = $builder->size;
        $this->cheese = $builder->cheese;
        $this->pepperoni = $builder->pepperoni;
        $this->lettuce = $builder->lettuce;
        $this->tomato = $builder->tomato;
    }
}
```

And then we have the builder

```
class BurgerBuilder
{
    public $size;

    public $cheese = false;
    public $pepperoni = false;
    public $lettuce = false;
    public $tomato = false;

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

    public function addPepperoni()
    {
        $this->pepperoni = true;
        return $this;
    }

    public function addLettuce()
    {
        $this->lettuce = true;
        return $this;
    }
}
```

```

    }

    public function addCheese()
    {
        $this->cheese = true;
        return $this;
    }

    public function addTomato()
    {
        $this->tomato = true;
        return $this;
    }

    public function build(): Burger
    {
        return new Burger($this);
    }
}

```

And then it can be used as:

```

$burger = (new BurgerBuilder(14))
    ->addPepperoni()
    ->addLettuce()
    ->addTomato()
    ->build();

```

### When to use?

When there could be several flavors of an object and to avoid the constructor telescoping. The key difference from the factory pattern is that; factory pattern is to be used when the creation is a one step process while builder pattern is to be used when the creation is a multi step process.

## Prototype

### Real world example

Remember dolly? The sheep that was cloned! Lets not get into the details but the key point here is that it is all about cloning

### In plain words

Create object based on an existing object through cloning.

### Wikipedia says

The prototype pattern is a creational design pattern in software development. It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

In short, it allows you to create a copy of an existing object and modify it to your needs, instead of going through the trouble of creating an object from scratch and setting it up.

### Programmatic Example

In PHP, it can be easily done using `clone`

```
class Sheep
{
    protected $name;
    protected $category;

    public function __construct(string $name, string $category = 'Mountain Sheep')
    {
        $this->name = $name;
        $this->category = $category;
    }

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

    public function getName()
    {
        return $this->name;
    }

    public function setCategory(string $category)
    {
        $this->category = $category;
    }

    public function getCategory()
    {
        return $this->category;
    }
}
```

Then it can be cloned like below

```
$original = new Sheep('Jolly');
echo $original->getName(); // Jolly
echo $original->getCategory(); // Mountain Sheep

// Clone and modify what is required
$cloned = clone $original;
$cloned->setName('Dolly');
echo $cloned->getName(); // Dolly
echo $cloned->getCategory(); // Mountain sheep
```

Also you could use the magic method `__clone` to modify the cloning behavior.

### When to use?

When an object is required that is similar to existing object or when the creation would be expensive as compared to cloning.

# Singleton

---

## Real world example

There can only be one president of a country at a time. The same president has to be brought to action, whenever duty calls. President here is singleton.

## In plain words

Ensures that only one object of a particular class is ever created.

## Wikipedia says

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Singleton pattern is actually considered an anti-pattern and overuse of it should be avoided. It is not necessarily bad and could have some valid use-cases but should be used with caution because it introduces a global state in your application and change to it in one place could affect in the other areas and it could become pretty difficult to debug. The other bad thing about them is it makes your code tightly coupled plus mocking the singleton could be difficult.

## Programmatic Example

To create a singleton, make the constructor private, disable cloning, disable extension and create a static variable to house the instance

```
final class President
{
    private static $instance;

    private function __construct()
    {
        // Hide the constructor
    }

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

        return self::$instance;
    }

    private function __clone()
    {
        // Disable cloning
    }

    private function __wakeup()
    {
        // Disable unserialize
    }
}
```

```
}  
}
```

Then in order to use

```
$president1 = President::getInstance();  
$president2 = President::getInstance();  
  
var_dump($president1 === $president2); // true
```

## Structural Design Patterns

In plain words

Structural patterns are mostly concerned with object composition or in other words how the entities can use each other. Or yet another explanation would be, they help in answering "How to build a software component?"

Wikipedia says

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

- [Adapter](#)
- [Bridge](#)
- [Composite](#)
- [Decorator](#)
- [Facade](#)
- [Flyweight](#)
- [Proxy](#)



### Adapter

Real world example

Consider that you have some pictures in your memory card and you need to transfer them to your computer. In order to transfer them you need some kind of adapter that is compatible with your computer ports so that you can attach memory card to your computer. In this case card reader is an adapter. Another example would be the famous power adapter; a three legged plug can't be connected to a two pronged outlet, it needs to use a power adapter that makes it compatible with the two pronged outlet. Yet another example would be a translator translating words spoken by one person to another

In plain words

Adapter pattern lets you wrap an otherwise incompatible object in an adapter to make it compatible with another class.

Wikipedia says

In software engineering, the adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

## Programmatic Example

Consider a game where there is a hunter and he hunts lions.

First we have an interface `Lion` that all types of lions have to implement

```
interface Lion
{
    public function roar();
}

class AfricanLion implements Lion
{
    public function roar()
    {
    }
}

class AsianLion implements Lion
{
    public function roar()
    {
    }
}
```

And hunter expects any implementation of `Lion` interface to hunt.

```
class Hunter
{
    public function hunt(Lion $lion)
    {
        $lion->roar();
    }
}
```

Now let's say we have to add a `wildDog` in our game so that hunter can hunt that also. But we can't do that directly because dog has a different interface. To make it compatible for our hunter, we will have to create an adapter that is compatible

```
// This needs to be added to the game
class wildDog
{
    public function bark()
    {
    }
}

// Adapter around wild dog to make it compatible with our game
class wildDogAdapter implements Lion
{
    protected $dog;
```

```
public function __construct(wildDog $dog)
{
    $this->dog = $dog;
}

public function roar()
{
    $this->dog->bark();
}
}
```

And now the `wildDog` can be used in our game using `wildDogAdapter`.

```
$wildDog = new wildDog();
$wildDogAdapter = new wildDogAdapter($wildDog);

$hunter = new Hunter();
$hunter->hunt($wildDogAdapter);
```

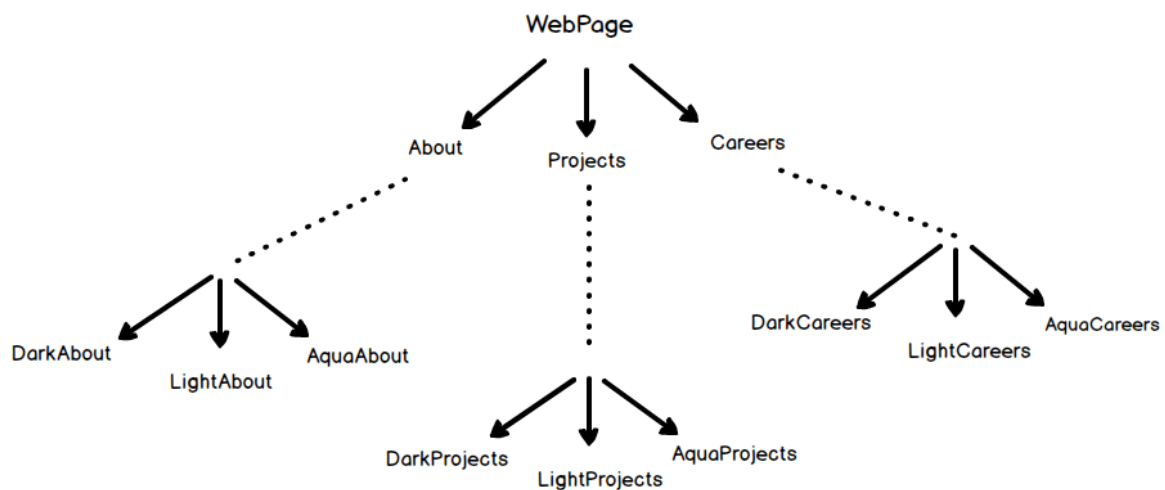
## Bridge

---

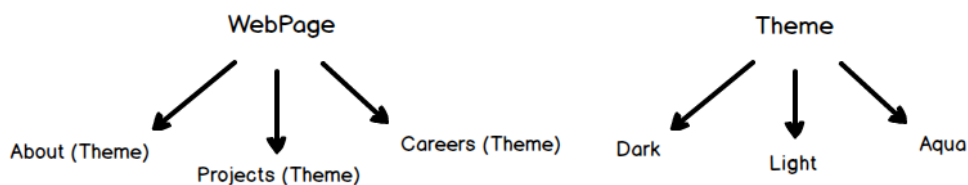
### Real world example

Consider you have a website with different pages and you are supposed to allow the user to change the theme. What would you do? Create multiple copies of each of the pages for each of the themes or would you just create separate theme and load them based on the user's preferences? Bridge pattern allows you to do the second i.e.

## Without Bridge



## With Bridge



### In Plain Words

Bridge pattern is about preferring composition over inheritance. Implementation details are pushed from a hierarchy to another object with a separate hierarchy.

### Wikipedia says

The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently"

### Programmatic Example

Translating our WebPage example from above. Here we have the `webPage` hierarchy

```
interface WebPage
{
    public function __construct(Theme $theme);
    public function getContent();
}

class About implements WebPage
{
    protected $theme;

    public function __construct(Theme $theme)
    {
```



```

        $this->theme = $theme;
    }

    public function getContent()
    {
        return "About page in " . $this->theme->getColor();
    }
}

class Careers implements WebPage
{
    protected $theme;

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

    public function getContent()
    {
        return "Careers page in " . $this->theme->getColor();
    }
}

```

And the separate theme hierarchy

```

interface Theme
{
    public function getColor();
}

class DarkTheme implements Theme
{
    public function getColor()
    {
        return 'Dark Black';
    }
}

class LightTheme implements Theme
{
    public function getColor()
    {
        return 'Off white';
    }
}

class AquaTheme implements Theme
{
    public function getColor()
    {
        return 'Light blue';
    }
}

```

And both the hierarchies

```
$darkTheme = new DarkTheme();

$about = new About($darkTheme);
$careers = new Careers($darkTheme);

echo $about->getContent(); // "About page in Dark Black";
echo $careers->getContent(); // "Careers page in Dark Black";
```

## Composite

Real world example

Every organization is composed of employees. Each of the employees has the same features i.e. has a salary, has some responsibilities, may or may not report to someone, may or may not have some subordinates etc.

In plain words

Composite pattern lets clients treat the individual objects in a uniform manner.

Wikipedia says

In software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

### Programmatic Example

Taking our employees example from above. Here we have different employee types

```
interface Employee
{
    public function __construct(string $name, float $salary);
    public function getName(): string;
    public function setSalary(float $salary);
    public function getSalary(): float;
    public function getRoles(): array;
}

class Developer implements Employee
{
    protected $salary;
    protected $name;
    protected $roles;

    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }
}
```

```

    public function getName(): string
    {
        return $this->name;
    }

    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
    }
}

```

class Designer implements Employee

```

{
    protected $salary;
    protected $name;
    protected $roles;

    public function __construct(string $name, float $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function setSalary(float $salary)
    {
        $this->salary = $salary;
    }

    public function getSalary(): float
    {
        return $this->salary;
    }

    public function getRoles(): array
    {
        return $this->roles;
    }
}

```

```
}  
}
```

Then we have an organization which consists of several different types of employees

```
class Organization  
{  
    protected $employees;  
  
    public function addEmployee(Employee $employee)  
    {  
        $this->employees[] = $employee;  
    }  
  
    public function getNetSalaries(): float  
    {  
        $netSalary = 0;  
  
        foreach ($this->employees as $employee) {  
            $netSalary += $employee->getSalary();  
        }  
  
        return $netSalary;  
    }  
}
```

And then it can be used as

```
// Prepare the employees  
$john = new Developer('John Doe', 12000);  
$jane = new Designer('Jane Doe', 15000);  
  
// Add them to organization  
$organization = new Organization();  
$organization->addEmployee($john);  
$organization->addEmployee($jane);  
  
echo "Net salaries: " . $organization->getNetSalaries(); // Net salaries: 27000
```

## Decorator

### Real world example

Imagine you run a car service shop offering multiple services. Now how do you calculate the bill to be charged? You pick one service and dynamically keep adding to it the prices for the provided services till you get the final cost. Here each type of service is a decorator.

### In plain words

Decorator pattern lets you dynamically change the behavior of an object at run time by wrapping them in an object of a decorator class.

Wikipedia says

In object-oriented programming, the decorator pattern is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. The decorator pattern is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

### Programmatic Example

Lets take coffee for example. First of all we have a simple coffee implementing the coffee interface

```
interface Coffee
{
    public function getCost();
    public function getDescription();
}

class SimpleCoffee implements Coffee
{
    public function getCost()
    {
        return 10;
    }

    public function getDescription()
    {
        return 'Simple coffee';
    }
}
```

We want to make the code extensible to allow options to modify it if required. Lets make some add-ons (decorators)

```
class MilkCoffee implements Coffee
{
    protected $coffee;

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

    public function getCost()
    {
        return $this->coffee->getCost() + 2;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', milk';
    }
}
```

```

class WhipCoffee implements Coffee
{
    protected $coffee;

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

    public function getCost()
    {
        return $this->coffee->getCost() + 5;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', whip';
    }
}

class VanillaCoffee implements Coffee
{
    protected $coffee;

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

    public function getCost()
    {
        return $this->coffee->getCost() + 3;
    }

    public function getDescription()
    {
        return $this->coffee->getDescription() . ', vanilla';
    }
}

```

Lets make a coffee now

```

$someCoffee = new SimpleCoffee();
echo $someCoffee->getCost(); // 10
echo $someCoffee->getDescription(); // Simple Coffee

$someCoffee = new MilkCoffee($someCoffee);
echo $someCoffee->getCost(); // 12
echo $someCoffee->getDescription(); // Simple Coffee, milk

$someCoffee = new WhipCoffee($someCoffee);
echo $someCoffee->getCost(); // 17

```

```
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip

$someCoffee = new VanillaCoffee($someCoffee);
echo $someCoffee->getCost(); // 20
echo $someCoffee->getDescription(); // Simple Coffee, milk, whip, vanilla
```

## Facade

### Real world example

How do you turn on the computer? "Hit the power button" you say! That is what you believe because you are using a simple interface that computer provides on the outside, internally it has to do a lot of stuff to make it happen. This simple interface to the complex subsystem is a facade.

### In plain words

Facade pattern provides a simplified interface to a complex subsystem.

### Wikipedia says

A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

### Programmatic Example

Taking our computer example from above. Here we have the computer class

```
class Computer
{
    public function getElectricShock()
    {
        echo "Ouch!";
    }

    public function makeSound()
    {
        echo "Beep beep!";
    }

    public function showLoadingScreen()
    {
        echo "Loading..";
    }

    public function bam()
    {
        echo "Ready to be used!";
    }

    public function closeEverything()
    {
        echo "Bup bup bup buzzzz!";
    }

    public function sooth()
```

```

    {
        echo "Zzzzz";
    }

    public function pullCurrent()
    {
        echo "Haaah!";
    }
}

```

Here we have the facade

```

class ComputerFacade
{
    protected $computer;

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

    public function turnOn()
    {
        $this->computer->getElectricShock();
        $this->computer->makeSound();
        $this->computer->showLoadingScreen();
        $this->computer->bam();
    }

    public function turnOff()
    {
        $this->computer->closeEverything();
        $this->computer->pullCurrent();
        $this->computer->sooth();
    }
}

```

Now to use the facade

```

$computer = new ComputerFacade(new Computer());
$computer->turnOn(); // Ouch! Beep beep! Loading.. Ready to be used!
$computer->turnOff(); // Bup bup buzzz! Haah! Zzzzz

```

## Flyweight

Real world example

Did you ever have fresh tea from some stall? They often make more than one cup that you demanded and save the rest for any other customer so to save the resources e.g. gas etc. Flyweight pattern is all about that i.e. sharing.

In plain words



It is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects.

Wikipedia says

In computer programming, flyweight is a software design pattern. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects; it is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

### Programmatic example

Translating our tea example from above. First of all we have tea types and tea maker

```
// Anything that will be cached is flyweight.
// Types of tea here will be flyweights.
class KarakTea
{
}

// Acts as a factory and saves the tea
class TeaMaker
{
    protected $availableTea = [];

    public function make($preference)
    {
        if (empty($this->availableTea[$preference])) {
            $this->availableTea[$preference] = new KarakTea();
        }

        return $this->availableTea[$preference];
    }
}
```

Then we have the `Teashop` which takes orders and serves them

```
class TeaShop
{
    protected $orders;
    protected $teaMaker;

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

    public function takeOrder(string $teaType, int $table)
    {
        $this->orders[$table] = $this->teaMaker->make($teaType);
    }

    public function serve()
    {
    }
}
```

```

        foreach ($this->orders as $table => $tea) {
            echo "Serving tea to table# " . $table;
        }
    }
}

```

And it can be used as below

```

$teaMaker = new TeaMaker();
$shop = new TeaShop($teaMaker);

$shop->takeOrder('less sugar', 1);
$shop->takeOrder('more milk', 2);
$shop->takeOrder('without sugar', 5);

$shop->serve();
// Serving tea to table# 1
// Serving tea to table# 2
// Serving tea to table# 5

```

## Proxy

### Real world example

Have you ever used an access card to go through a door? There are multiple options to open that door i.e. it can be opened either using access card or by pressing a button that bypasses the security. The door's main functionality is to open but there is a proxy added on top of it to add some functionality. Let me better explain it using the code example below.

### In plain words

Using the proxy pattern, a class represents the functionality of another class.

### Wikipedia says

A proxy, in its most general form, is a class functioning as an interface to something else. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.

### Programmatic Example

Taking our security door example from above. Firstly we have the door interface and an implementation of door

```

interface Door
{
    public function open();
    public function close();
}

class LabDoor implements Door

```

```

{
    public function open()
    {
        echo "Opening lab door";
    }

    public function close()
    {
        echo "Closing the lab door";
    }
}

```

Then we have a proxy to secure any doors that we want

```

class SecuredDoor
{
    protected $door;

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

    public function open($password)
    {
        if ($this->authenticate($password)) {
            $this->door->open();
        } else {
            echo "Big no! It ain't possible.";
        }
    }

    public function authenticate($password)
    {
        return $password === '$ecr@t';
    }

    public function close()
    {
        $this->door->close();
    }
}

```

And here is how it can be used

```

$door = new SecuredDoor(new LabDoor());
$door->open('invalid'); // Big no! It ain't possible.

$door->open('$ecr@t'); // opening lab door
$door->close(); // closing lab door

```

Yet another example would be some sort of data-mapper implementation. For example, I recently made an ODM (Object Data Mapper) for MongoDB using this pattern where I wrote a proxy around mongo classes while utilizing the magic method `__call()`. All the method calls were proxied to the original mongo class and result retrieved was returned as it is but in case of `find` or `findOne` data was mapped to the required class objects and the object was returned instead of `Cursor`.

## Behavioral Design Patterns

---

In plain words

It is concerned with assignment of responsibilities between the objects. What makes them different from structural patterns is they don't just specify the structure but also outline the patterns for message passing/communication between them. Or in other words, they assist in answering "How to run a behavior in software component?"

Wikipedia says

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

- [Chain of Responsibility](#)
- [Command](#)
- [Iterator](#)
- [Mediator](#)
- [Memento](#)
- [Observer](#)
- [Visitor](#)
- [Strategy](#)
- [State](#)
- [Template Method](#)

### Chain of Responsibility

---

Real world example

For example, you have three payment methods (**A**, **B** and **C**) setup in your account; each having a different amount in it. **A** has 100 USD, **B** has 300 USD and **C** having 1000 USD and the preference for payments is chosen as **A** then **B** then **C**. You try to purchase something that is worth 210 USD. Using Chain of Responsibility, first of all account **A** will be checked if it can make the purchase, if yes purchase will be made and the chain will be broken. If not, request will move forward to account **B** checking for amount if yes chain will be broken otherwise the request will keep forwarding till it finds the suitable handler. Here **A**, **B** and **C** are links of the chain and the whole phenomenon is Chain of Responsibility.

In plain words

It helps building a chain of objects. Request enters from one end and keeps going from object to object till it finds the suitable handler.

Wikipedia says

In object-oriented design, the chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain.

### Programmatic Example

Translating our account example above. First of all we have a base account having the logic for chaining the accounts together and some accounts

```
abstract class Account
{
    protected $successor;
    protected $balance;

    public function setNext(Account $account)
    {
        $this->successor = $account;
    }

    public function pay(float $amountToPay)
    {
        if ($this->canPay($amountToPay)) {
            echo sprintf('Paid %s using %s' . PHP_EOL, $amountToPay, get_called_class());
        } elseif ($this->successor) {
            echo sprintf('Cannot pay using %s. Proceeding ..' . PHP_EOL,
get_called_class());
            $this->successor->pay($amountToPay);
        } else {
            throw new Exception('None of the accounts have enough balance');
        }
    }

    public function canPay($amount): bool
    {
        return $this->balance >= $amount;
    }
}

class Bank extends Account
{
    protected $balance;

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

class Paypal extends Account
{
    protected $balance;
```

```

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

class Bitcoin extends Account
{
    protected $balance;

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

```

Now let's prepare the chain using the links defined above (i.e. Bank, Paypal, Bitcoin)

```

// Let's prepare a chain like below
//      $bank->$paypal->$bitcoin
//
// First priority bank
//      If bank can't pay then paypal
//      If paypal can't pay then bit coin

$bank = new Bank(100);           // Bank with balance 100
$paypal = new Paypal(200);       // Paypal with balance 200
$bitcoin = new Bitcoin(300);     // Bitcoin with balance 300

$bank->setNext($paypal);
$paypal->setNext($bitcoin);

// Let's try to pay using the first priority i.e. bank
$bank->pay(259);

// Output will be
// =====
// Cannot pay using bank. Proceeding ..
// Cannot pay using paypal. Proceeding ...
// Paid 259 using Bitcoin!

```



## Command

### Real world example

A generic example would be you ordering food at a restaurant. You (i.e. **Client**) ask the waiter (i.e. **Invoker**) to bring some food (i.e. **Command**) and waiter simply forwards the request to Chef (i.e. **Receiver**) who has the knowledge of what and how to cook. Another example would be you (i.e. **Client**) switching on (i.e. **Command**) the television (i.e. **Receiver**) using a remote control (**Invoker**).

### In plain words

Allows you to encapsulate actions in objects. The key idea behind this pattern is to provide the means to decouple client from receiver.

Wikipedia says

In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

### Programmatic Example

First of all we have the receiver that has the implementation of every action that could be performed

```
// Receiver
class Bulb
{
    public function turnOn()
    {
        echo "Bulb has been lit";
    }

    public function turnOff()
    {
        echo "Darkness!";
    }
}
```

then we have an interface that each of the commands are going to implement and then we have a set of commands

```
interface Command
{
    public function execute();
    public function undo();
    public function redo();
}

// Command
class TurnOn implements Command
{
    protected $bulb;

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

    public function execute()
    {
        $this->bulb->turnOn();
    }
}
```

```

    public function undo()
    {
        $this->bulb->turnOff();
    }

    public function redo()
    {
        $this->execute();
    }
}

class TurnOff implements Command
{
    protected $bulb;

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

    public function execute()
    {
        $this->bulb->turnOff();
    }

    public function undo()
    {
        $this->bulb->turnOn();
    }

    public function redo()
    {
        $this->execute();
    }
}

```

Then we have an `Invoker` with whom the client will interact to process any commands

```

// Invoker
class RemoteControl
{
    public function submit(Command $command)
    {
        $command->execute();
    }
}

```

Finally let's see how we can use it in our client



```
$bulb = new Bulb();

$turnOn = new TurnOn($bulb);
$turnOff = new TurnOff($bulb);

$remote = new RemoteControl();
$remote->submit($turnOn); // Bulb has been lit!
$remote->submit($turnOff); // Darkness!
```

Command pattern can also be used to implement a transaction based system. Where you keep maintaining the history of commands as soon as you execute them. If the final command is successfully executed, all good otherwise just iterate through the history and keep executing the `undo` on all the executed commands.

## Iterator

### Real world example

An old radio set will be a good example of iterator, where user could start at some channel and then use next or previous buttons to go through the respective channels. Or take an example of MP3 player or a TV set where you could press the next and previous buttons to go through the consecutive channels or in other words they all provide an interface to iterate through the respective channels, songs or radio stations.

### In plain words

It presents a way to access the elements of an object without exposing the underlying presentation.

### Wikipedia says

In object-oriented programming, the iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

### Programmatic example

In PHP it is quite easy to implement using SPL (Standard PHP Library). Translating our radio stations example from above. First of all we have `RadioStation`

```
class RadioStation
{
    protected $frequency;

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

    public function getFrequency(): float
    {
        return $this->frequency;
    }
}
```

Then we have our iterator

```
use Countable;
use Iterator;

class StationList implements Countable, Iterator
{
    /** @var RadioStation[] $stations */
    protected $stations = [];

    /** @var int $counter */
    protected $counter;

    public function addStation(RadioStation $station)
    {
        $this->stations[] = $station;
    }

    public function removeStation(RadioStation $toRemove)
    {
        $toRemoveFrequency = $toRemove->getFrequency();
        $this->stations = array_filter($this->stations, function (RadioStation $station) use
($toRemoveFrequency) {
            return $station->getFrequency() !== $toRemoveFrequency;
        });
    }

    public function count(): int
    {
        return count($this->stations);
    }

    public function current(): RadioStation
    {
        return $this->stations[$this->counter];
    }

    public function key()
    {
        return $this->counter;
    }

    public function next()
    {
        $this->counter++;
    }

    public function rewind()
    {
        $this->counter = 0;
    }
}
```

```

public function valid(): bool
{
    return isset($this->stations[$this->counter]);
}
}

```

And then it can be used as

```

$stationList = new StationList();

$stationList->addStation(new RadioStation(89));
$stationList->addStation(new RadioStation(101));
$stationList->addStation(new RadioStation(102));
$stationList->addStation(new RadioStation(103.2));

foreach($stationList as $station) {
    echo $station->getFrequency() . PHP_EOL;
}

$stationList->removeStation(new RadioStation(89)); // will remove station 89

```



## Mediator

### Real world example

A general example would be when you talk to someone on your mobile phone, there is a network provider sitting between you and them and your conversation goes through it instead of being directly sent. In this case network provider is mediator.

### In plain words

Mediator pattern adds a third party object (called mediator) to control the interaction between two objects (called colleagues). It helps reduce the coupling between the classes communicating with each other. Because now they don't need to have the knowledge of each other's implementation.

### Wikipedia says

In software engineering, the mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior.

### Programmatic Example

Here is the simplest example of a chat room (i.e. mediator) with users (i.e. colleagues) sending messages to each other.

First of all, we have the mediator i.e. the chat room

```

interface ChatRoomMediator
{
    public function showMessage(User $user, string $message);
}

```

```
// Mediator
class ChatRoom implements ChatRoomMediator
{
    public function showMessage(User $user, string $message)
    {
        $time = date('M d, y H:i');
        $sender = $user->getName();

        echo $time . '[' . $sender . ']:' . $message;
    }
}
```

Then we have our users i.e. colleagues

```
class User {
    protected $name;
    protected $chatMediator;

    public function __construct(string $name, ChatRoomMediator $chatMediator) {
        $this->name = $name;
        $this->chatMediator = $chatMediator;
    }

    public function getName() {
        return $this->name;
    }

    public function send($message) {
        $this->chatMediator->showMessage($this, $message);
    }
}
```

And the usage

```
$mediator = new ChatRoom();

$john = new User('John Doe', $mediator);
$jane = new User('Jane Doe', $mediator);

$john->send('Hi there!');
$jane->send('Hey!');

// Output will be
// Feb 14, 10:58 [John]: Hi there!
// Feb 14, 10:58 [Jane]: Hey!
```

## Memento

Real world example

Take the example of calculator (i.e. originator), where whenever you perform some calculation the last calculation is saved in memory (i.e. memento) so that you can get back to it and maybe get it restored using some action buttons (i.e. caretaker).

#### In plain words

Memento pattern is about capturing and storing the current state of an object in a manner that it can be restored later on in a smooth manner.

#### Wikipedia says

The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

Usually useful when you need to provide some sort of undo functionality.

#### Programmatic Example

Lets take an example of text editor which keeps saving the state from time to time and that you can restore if you want.

First of all we have our memento object that will be able to hold the editor state

```
class EditorMemento
{
    protected $content;

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

    public function getContent()
    {
        return $this->content;
    }
}
```

Then we have our editor i.e. originator that is going to use memento object

```
class Editor
{
    protected $content = '';

    public function type(string $words)
    {
        $this->content = $this->content . ' ' . $words;
    }

    public function getContent()
    {
        return $this->content;
    }
}
```

```

public function save()
{
    return new EditorMemento($this->content);
}

public function restore(EditorMemento $memento)
{
    $this->content = $memento->getContent();
}
}

```

And then it can be used as

```

$editor = new Editor();

// Type some stuff
$editor->type('This is the first sentence. ');
$editor->type('This is second. ');

// Save the state to restore to : This is the first sentence. This is second.
$saved = $editor->save();

// Type some more
$editor->type('And this is third. ');

// Output: Content before Saving
echo $editor->getContent(); // This is the first sentence. This is second. And this is
third.

// Restoring to last saved state
$editor->restore($saved);

$editor->getContent(); // This is the first sentence. This is second.

```

## Observer

### Real world example

A good example would be the job seekers where they subscribe to some job posting site and they are notified whenever there is a matching job opportunity.

### In plain words

Defines a dependency between objects so that whenever an object changes its state, all its dependents are notified.

### Wikipedia says

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

### Programmatic example

Translating our example from above. First of all we have job seekers that need to be notified for a job posting

```
class JobPost
{
    protected $title;

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

    public function getTitle()
    {
        return $this->title;
    }
}

class JobSeeker implements Observer
{
    protected $name;

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

    public function onJobPosted(JobPost $job)
    {
        // Do something with the job posting
        echo 'Hi ' . $this->name . '! New job posted: '. $job->getTitle();
    }
}
```

Then we have our job postings to which the job seekers will subscribe

```
class EmploymentAgency implements Observable
{
    protected $observers = [];

    protected function notify(JobPost $jobPosting)
    {
        foreach ($this->observers as $observer) {
            $observer->onJobPosted($jobPosting);
        }
    }

    public function attach(Observer $observer)
    {
        $this->observers[] = $observer;
    }

    public function addJob(JobPost $jobPosting)
```

```
{  
    $this->notify($jobPosting);  
}  
}
```

Then it can be used as

```
// Create subscribers  
$johnDoe = new JobSeeker('John Doe');  
$janeDoe = new JobSeeker('Jane Doe');  
  
// Create publisher and attach subscribers  
$jobPostings = new EmploymentAgency();  
$jobPostings->attach($johnDoe);  
$jobPostings->attach($janeDoe);  
  
// Add a new job and see if subscribers get notified  
$jobPostings->addJob(new JobPost('Software Engineer'));  
  
// Output  
// Hi John Doe! New job posted: Software Engineer  
// Hi Jane Doe! New job posted: Software Engineer
```

## Visitor

### Real world example

Consider someone visiting Dubai. They just need a way (i.e. visa) to enter Dubai. After arrival, they can come and visit any place in Dubai on their own without having to ask for permission or to do some leg work in order to visit any place here; just let them know of a place and they can visit it. Visitor pattern lets you do just that, it helps you add places to visit so that they can visit as much as they can without having to do any legwork.

### In plain words

Visitor pattern lets you add further operations to objects without having to modify them.

### Wikipedia says

In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. It is one way to follow the open/closed principle.

### Programmatic example

Let's take an example of a zoo simulation where we have several different kinds of animals and we have to make them Sound. Let's translate this using visitor pattern



```
// Visitee
interface Animal
{
    public function accept(AnimalOperation $operation);
}

// Visitor
interface AnimalOperation
{
    public function visitMonkey(Monkey $monkey);
    public function visitLion(Lion $lion);
    public function visitDolphin(Dolphin $dolphin);
}
```

Then we have our implementations for the animals

```
class Monkey implements Animal
{
    public function shout()
    {
        echo 'Ooh oo aa aa!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitMonkey($this);
    }
}

class Lion implements Animal
{
    public function roar()
    {
        echo 'Roaaar!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitLion($this);
    }
}

class Dolphin implements Animal
{
    public function speak()
    {
        echo 'Tuut tuttu tuutt!';
    }

    public function accept(AnimalOperation $operation)
    {
        $operation->visitDolphin($this);
    }
}
```

```
}  
}
```

Let's implement our visitor

```
class Speak implements AnimalOperation  
{  
    public function visitMonkey(Monkey $monkey)  
    {  
        $monkey->shout();  
    }  
  
    public function visitLion(Lion $lion)  
    {  
        $lion->roar();  
    }  
  
    public function visitDolphin(Dolphin $dolphin)  
    {  
        $dolphin->speak();  
    }  
}
```

And then it can be used as

```
$monkey = new Monkey();  
$lion = new Lion();  
$dolphin = new Dolphin();  
  
$speak = new Speak();  
  
$monkey->accept($speak);    // Ooh oo aa aa!  
$lion->accept($speak);      // Roaaar!  
$dolphin->accept($speak);   // Tuut tutt tuutt!
```

We could have done this simply by having an inheritance hierarchy for the animals but then we would have to modify the animals whenever we would have to add new actions to animals. But now we will not have to change them. For example, let's say we are asked to add the jump behavior to the animals, we can simply add that by creating a new visitor i.e.

```
class Jump implements AnimalOperation  
{  
    public function visitMonkey(Monkey $monkey)  
    {  
        echo 'Jumped 20 feet high! on to the tree!';  
    }  
  
    public function visitLion(Lion $lion)  
    {  
        echo 'Jumped 7 feet! Back on the ground!';  
    }  
}
```

```

    public function visitDolphin(Dolphin $dolphin)
    {
        echo 'walked on water a little and disappeared';
    }
}

```

And for the usage

```

$jump = new Jump();

$monkey->accept($speak); // Ooh oo aa aa!
$monkey->accept($jump);  // Jumped 20 feet high! on to the tree!

$lion->accept($speak);    // Roaaar!
$lion->accept($jump);     // Jumped 7 feet! Back on the ground!

$dolphin->accept($speak); // Tuut tutt tuutt!
$dolphin->accept($jump);  // walked on water a little and disappeared

```

## Strategy

Real world example

Consider the example of sorting, we implemented bubble sort but the data started to grow and bubble sort started getting very slow. In order to tackle this we implemented Quick sort. But now although the quick sort algorithm was doing better for large datasets, it was very slow for smaller datasets. In order to handle this we implemented a strategy where for small datasets, bubble sort will be used and for larger, quick sort.

In plain words

Strategy pattern allows you to switch the algorithm or strategy based upon the situation.

Wikipedia says

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioural software design pattern that enables an algorithm's behavior to be selected at runtime.

### Programmatic example

Translating our example from above. First of all we have our strategy interface and different strategy implementations

```

interface SortStrategy
{
    public function sort(array $dataset): array;
}

class BubbleSortStrategy implements SortStrategy
{
    public function sort(array $dataset): array
    {

```

```

        echo "Sorting using bubble sort";

        // Do sorting
        return $dataset;
    }
}

class QuickSortStrategy implements SortStrategy
{
    public function sort(array $dataset): array
    {
        echo "Sorting using quick sort";

        // Do sorting
        return $dataset;
    }
}

```

And then we have our client that is going to use any strategy

```

class Sorter
{
    protected $sorter;

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

    public function sort(array $dataset): array
    {
        return $this->sorter->sort($dataset);
    }
}

```

And it can be used as

```

$dataset = [1, 5, 4, 3, 2, 8];

$sorter = new Sorter(new BubbleSortStrategy());
$sorter->sort($dataset); // Output : Sorting using bubble sort

$sorter = new Sorter(new QuickSortStrategy());
$sorter->sort($dataset); // Output : Sorting using quick sort

```



Real world example

Imagine you are using some drawing application, you choose the paint brush to draw. Now the brush changes its behavior based on the selected color i.e. if you have chosen red color it will draw in red, if blue then it will be in blue etc.

#### In plain words

It lets you change the behavior of a class when the state changes.

#### Wikipedia says

The state pattern is a behavioral software design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass. The state pattern can be interpreted as a strategy pattern which is able to switch the current strategy through invocations of methods defined in the pattern's interface.

#### Programmatic example

Let's take an example of text editor, it lets you change the state of text that is typed i.e. if you have selected bold, it starts writing in bold, if italic then in italics etc.

First of all we have our state interface and some state implementations

```
interface WritingState
{
    public function write(string $words);
}

class UpperCase implements WritingState
{
    public function write(string $words)
    {
        echo strtoupper($words);
    }
}

class LowerCase implements WritingState
{
    public function write(string $words)
    {
        echo strtolower($words);
    }
}

class DefaultText implements WritingState
{
    public function write(string $words)
    {
        echo $words;
    }
}
```

Then we have our editor

```

class TextEditor
{
    protected $state;

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

    public function setState(WritingState $state)
    {
        $this->state = $state;
    }

    public function type(string $words)
    {
        $this->state->write($words);
    }
}

```

And then it can be used as

```

$editor = new TextEditor(new DefaultText());

$editor->type('First line');

$editor->setState(new UpperCase());

$editor->type('Second line');
$editor->type('Third line');

$editor->setState(new LowerCase());

$editor->type('Fourth line');
$editor->type('Fifth line');

// Output:
// First line
// SECOND LINE
// THIRD LINE
// fourth line
// fifth line

```

## Template Method

Real world example

Suppose we are getting some house built. The steps for building might look like

- Prepare the base of house
- Build the walls
- Add roof

- Add other floors

The order of these steps could never be changed i.e. you can't build the roof before building the walls etc but each of the steps could be modified for example walls can be made of wood or polyester or stone.

In plain words

Template method defines the skeleton of how a certain algorithm could be performed, but defers the implementation of those steps to the children classes.

Wikipedia says

In software engineering, the template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets one redefine certain steps of an algorithm without changing the algorithm's structure.

### Programmatic Example

Imagine we have a build tool that helps us test, lint, build, generate build reports (i.e. code coverage reports, linting report etc) and deploy our app on the test server.

First of all we have our base class that specifies the skeleton for the build algorithm

```
abstract class Builder
{
    // Template method
    final public function build()
    {
        $this->test();
        $this->lint();
        $this->assemble();
        $this->deploy();
    }

    abstract public function test();
    abstract public function lint();
    abstract public function assemble();
    abstract public function deploy();
}
```

Then we can have our implementations

```
class AndroidBuilder extends Builder
{
    public function test()
    {
        echo 'Running android tests';
    }

    public function lint()
    {
        echo 'Linting the android code';
    }
}
```

```

    }

    public function assemble()
    {
        echo 'Assembling the android build';
    }

    public function deploy()
    {
        echo 'Deploying android build to server';
    }
}

class IosBuilder extends Builder
{
    public function test()
    {
        echo 'Running ios tests';
    }

    public function lint()
    {
        echo 'Linting the ios code';
    }

    public function assemble()
    {
        echo 'Assembling the ios build';
    }

    public function deploy()
    {
        echo 'Deploying ios build to server';
    }
}

```

And then it can be used as

```

$androidBuilder = new AndroidBuilder();
$androidBuilder->build();

// Output:
// Running android tests
// Linting the android code
// Assembling the android build
// Deploying android build to server

$iosBuilder = new IosBuilder();
$iosBuilder->build();

// Output:
// Running ios tests
// Linting the ios code

```



```
// Assembling the ios build
// Deploying ios build to server
```


## Wrap Up Folks

---

And that about wraps it up. I will continue to improve this, so you might want to watch/star this repository to revisit. Also, I have plans on writing the same about the architectural patterns, stay tuned for it.

## Contribution

---

- Report issues
- Open pull request with improvements
- Spread the word
- Reach out with any feedback  [Follow @kamranahmedse](#)

## License

---

License **CC BY 4.0**