





---------

**Major Unit: Design Patterns**

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. A general definition could be: In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.

Types of Design Patterns

* Creational
* Structural
* Behavioral

**CREATIONAL DESIGN PATTERNS**

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

Some Creational design patterns are: Simple Factory, Factory Method, Singleton, Abstract Factory, Builder, Prototype,

**Simple Factory**

Consider, you are building a house and you need doors. It would be a mess if every time you need a door, you put on your carpenter clothes and start making a door in your house. Instead you get it made from a factory.

In plain words, Simple factory simply generates an instance for client without exposing any instantiation logic to the client.

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

$door = DoorFactory::makeDoor(100, 200);

echo 'Width: ' . $door->getWidth();

echo 'Height: ' . $door->getHeight();

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**

Defines an interface for creating an object, but let’s classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.

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.

**Singleton Patterns**

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

public class SingleObject {

//create an object of SingleObject

private static SingleObject instance = new SingleObject();

//make the constructor private so that this class cannot be

//instantiated

private SingleObject(){}

//Get the only object available

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

----

Get the only object from the singleton class.

SingletonPatternDemo.java

public class SingletonPatternDemo {

public static void main(String[] args) {

//illegal construct if you write this

//SingleObject object = new SingleObject();

//Get the only object available

SingleObject object = SingleObject.getInstance();

//show the message

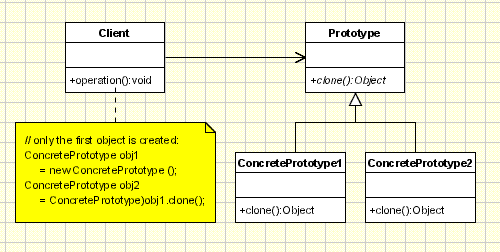
object.showMessage();

}

}

**Prototype Pattern**

The purpose of the prototype pattern is to create object based on an existing object through cloning.



For example: The classes participating to the Prototype Pattern are:

* Client - creates a new object by asking a prototype to clone itself.
* Prototype - declares an interface for cloning itself.
* ConcretePrototype - implements the operation for cloning itself.

The process of cloning starts with an initialized and instantiated class. The Client asks for a new object of that type and sends the request to the Prototype class. A ConcretePrototype, depending of the type of object is needed, will handle the cloning through the Clone() method, making a new instance of itself. Here is a sample code for the Prototype pattern:

|  |
| --- |
| Public interface Prototype  {  public abstract Object clone ( );  }  public class ConcretePrototype implements Prototype {  public Object clone() {  return super.clone();  }  }  public class Client {  public static void main( String arg[] )  {  ConcretePrototype obj1= new ConcretePrototype ();  ConcretePrototype obj2 = (ConcretePrototype)obj1.clone();  }  } |

**Builder 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();

**Structural Patterns**

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?"

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. Some examples: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

**Bridge**

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.



|  |  |  |
| --- | --- | --- |
| 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();  }  } | 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';  }  } | $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"; |

**Decorator Pattern**

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

|  |  |  |
| --- | --- | --- |
| interface Coffee  {  public function getCost();  public function getDescription();  }  class SimpleCoffee implements Coffee  {  public function getCost()  {  return 10;  }  public function getDescription()  {  return 'Simple coffee';  }  } | 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';  }  } | $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.

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

**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.

|  |  |  |
| --- | --- | --- |
| 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";  }  } | 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();  }  } | $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 |

**Behavioral Design Patterns**

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?". Example of behavioral design patterns are: Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, Visitor, Strategy, State, Template Method.

Chain of Responsibility design pattern

The Chain of Responsibility design pattern allows an object to send a command without knowing what object will receive and handle it. The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both. The most usual example of a machine using the Chain of Responsibility is the vending machine coin slot: rather than having a slot for each type of coin, the machine has only one slot for all of them. The dropped coin is routed to the appropriate storage place that is determined by the receiver of the command.

switch (n) {

case label1:

code to be executed if n=label1;

break;

case label2:

code to be executed if n=label2;

break;

case label3:

code to be executed if n=label3;

break;

...

default:

code to be executed if n is different from all labels;

}

Iterator Design Pattern

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. For 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.

Mediator Design Pattern

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. 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.

Memento Design Pattern

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