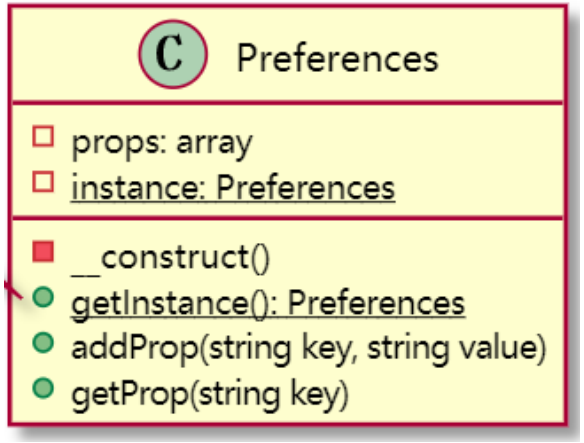
Singleton



- replace global variables
- better encapsulation
- perform associated operations within the class
- Singleton is considered bad by someone.
- Use with caution and do not overuse.

```php
$option = Preferences::getInstance();
$option->addProp('username', 'zhang');
var_dump($option->getProp('username'));
```

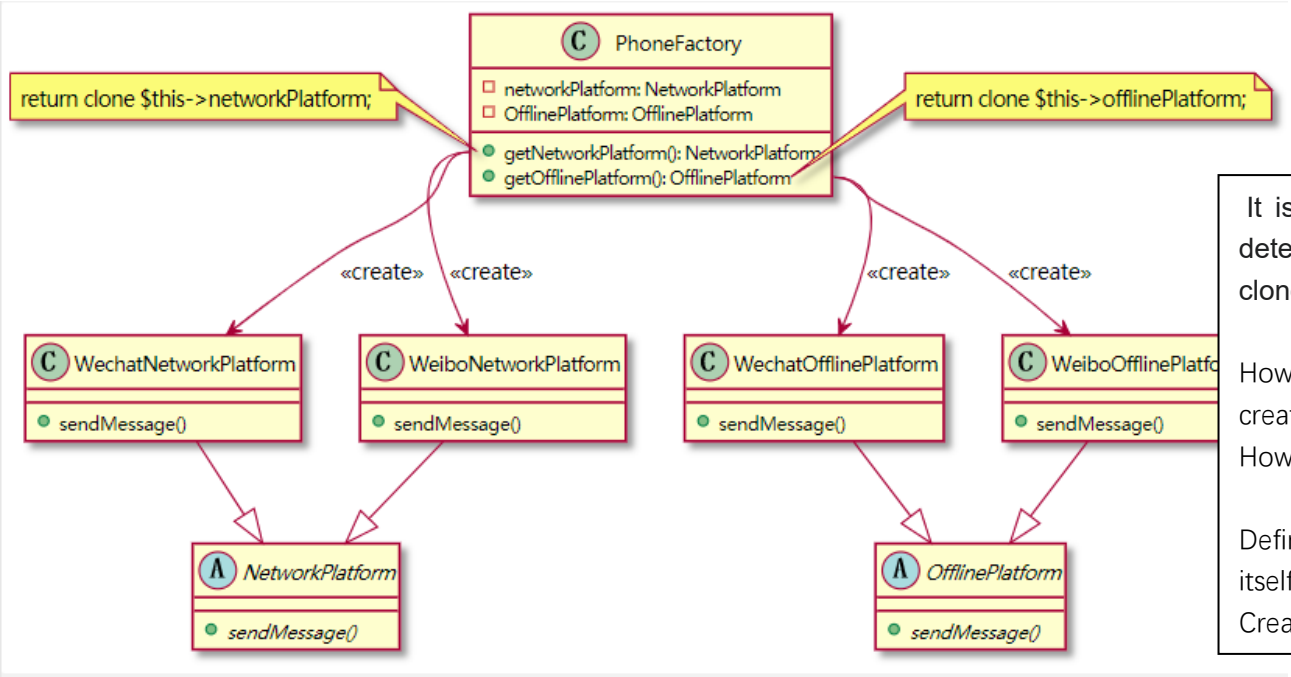
```php
class Preferences
{
    private array $props;
    private static Preferences $instance;

    public static function getInstance()
    {
        // ATTENTION THE LOGIC
        if (empty(self::$instance)) {
            self::$instance = new Preferences();
        }
        return self::$instance;
    }
```

restricts the instantiation of a class to one "single" instance.

Ensure that a class only has one instance
Easily access the sole instance of a class
Control its instantiation
Restrict the number of instances
Access a global variable

Hide the constructors of the class.
Define a public static operation (getInstance()) that returns the sole instance of the class.

Prototype



It is used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects.

How can objects be created so that which objects to create can be specified at run-time?
How can dynamically loaded classes be instantiated?

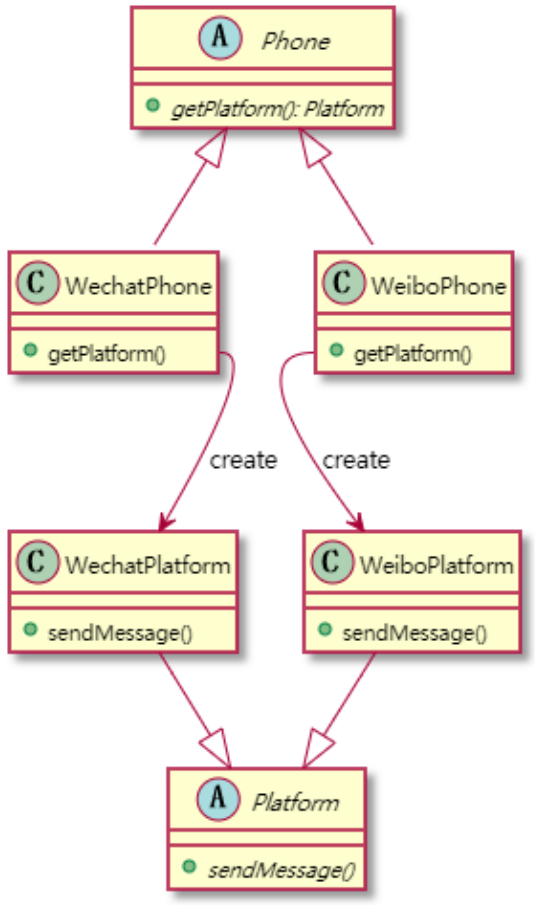
Define a Prototype object that returns a copy of itself.
Create new objects by copying a Prototype object.

- avoid too many subclass
- avoid the cost of new a class
- may be need deep clone (__clone) in php

```php
$phone = new PhoneFactory(
    new WeiboNetworkPlatform(),
    new WeiboOfflinePlatform()
);
var_dump($phone->getNetworkPlatform()->sendMessage())
;
```

```php
class PhoneFactory
{
    private NetworkPlatform $networkPlatform;

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

    public function getNetworkPlatform(): NetworkPlatform
    {
        return clone $this->networkPlatform;
    }
```

Factory Method



- separate creator and product
- use subclass to create object
- but may cause too many subclass
- less advantage

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. rather than by calling a constructor.

How can an object be created so that subclasses can redefine which class to instantiate?
How can a class defer instantiation to subclasses?

Define a separate operation (factory method) for creating an object.
Create an object by calling a factory method.

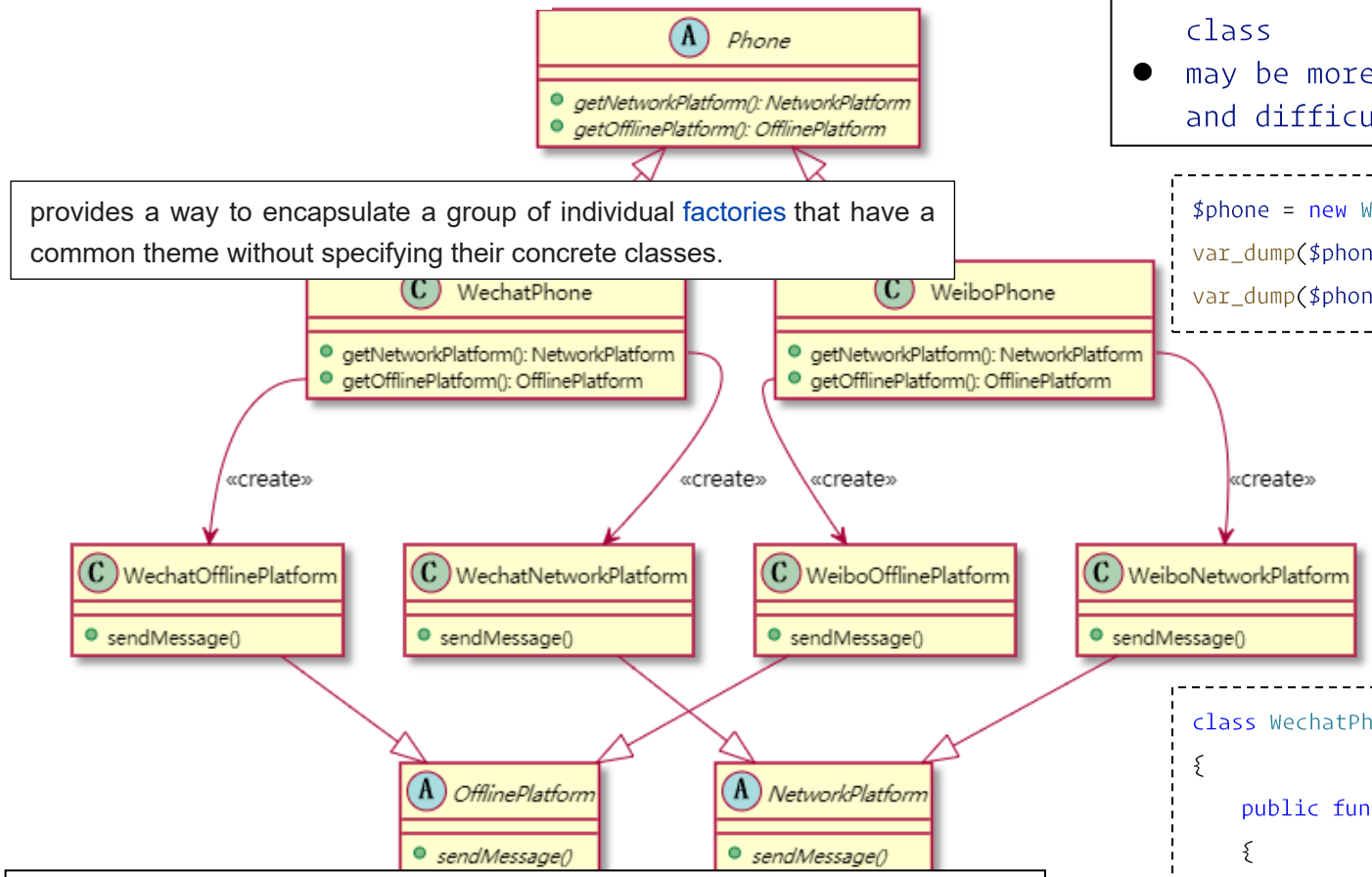
```php
$phone = new WechatPhone();
var_dump($phone->getPlatform()->sendMessage()
```

```php
abstract class Phone
{
    abstract public function getPlatform():
    Platform;
    // Attention here, the return type is Platform
    // here is factory method
}

class WechatPhone extends Phone
{
    public function getPlatform()
    {
        return new WechatPlatform();
    }
}
class WechatPlatform extends Platform
{
    public function sendMessage()
    {
        return 'sent by wechat';
    }
```

Abstract Factory

- evolution of Factory Method
- separate creator and implement
- extend new type without changing base class
- may be more complicated, too many files and difficult to maintain

provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.



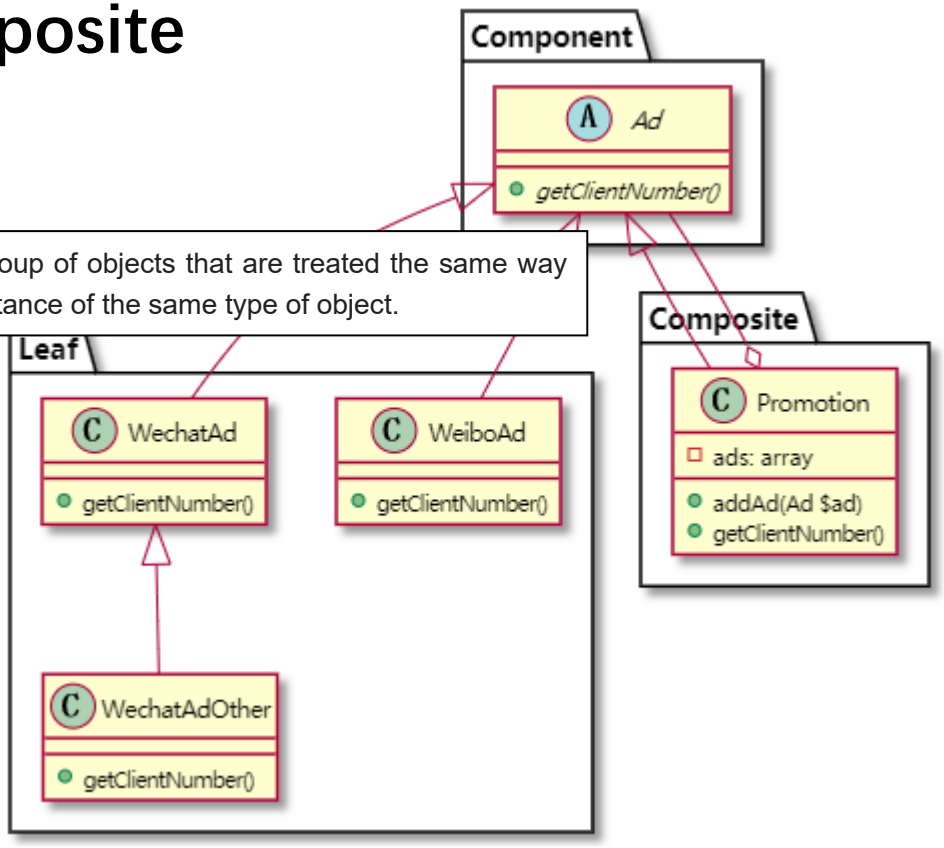

```php
$phone = new WechatPhone();
var_dump($phone->getNetworkPlatform()->sendMessage());
var_dump($phone->getOfflinePlatform()->sendMessage());
```

How can an application be independent of how its objects are created?
How can a class be independent of how the objects it requires are created?
How can families of related or dependent objects be created?

Encapsulate object creation in a separate (factory) object. That is, define an interface (AbstractFactory) for creating objects, and implement the interface.
A class delegates object creation to a factory object instead of creating objects directly.

```php
class WechatPhone extends Phone
{
    public function getNetworkPlatform(): NetworkPlatform
    {
        return new WechatNetworkPlatform();
    }

    public function getOfflinePlatform(): OfflinePlatform
    {
        return new WechatOfflinePlatform();
    }
```

Composite



describes a group of objects that are treated the same way as a single instance of the same type of object.

- good for complex tree structure
- easy to extend tree
- difficult to provide common interface (aka component)

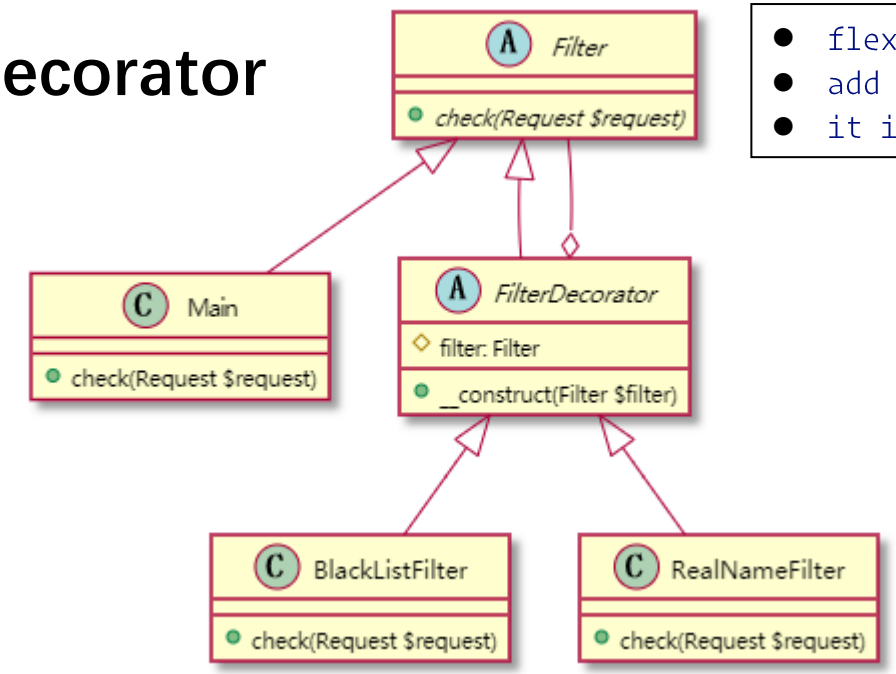
```php
$innerTeam = new Promotion();
$innerTeam->addAd(new WeiboAd());
$outerTeam = new Promotion();
$outerTeam->addAd(new WechatAd());

$innerTeam->addAd($outerTeam);
var_dump($innerTeam->getClientNumber());
```

```php
class Promotion extends Ad
{
    private array $ads = [];
    public function addAd(Ad $ad)
    {
        $this->ads[] = $ad;
    }
    public function getClientNumber()
    {
        $number = 0;
        foreach ($this->ads as $ad) {
            $number += $ad->getClientNumber();
        }
        return $number;
    }
}
```

A part-whole hierarchy should be represented so that clients can treat part and whole objects uniformly.
A part-whole hierarchy should be represented as tree structure.

Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects. Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

Decorator



- flexible way to replace subclass
- add objects dynamically at runtime
- it is not a language feature...

```
/*
    Filter +check()
    RealNameFilter
    BlackListFilter
    Main()
*/
$filter = new RealNameFilter(new BlackListFilter(new Main()));
$filter->check(new Request());
```

```
class Main extends Filter
{
    public function check(Request $request)
    {
        echo 'do something useful...';
    }
```

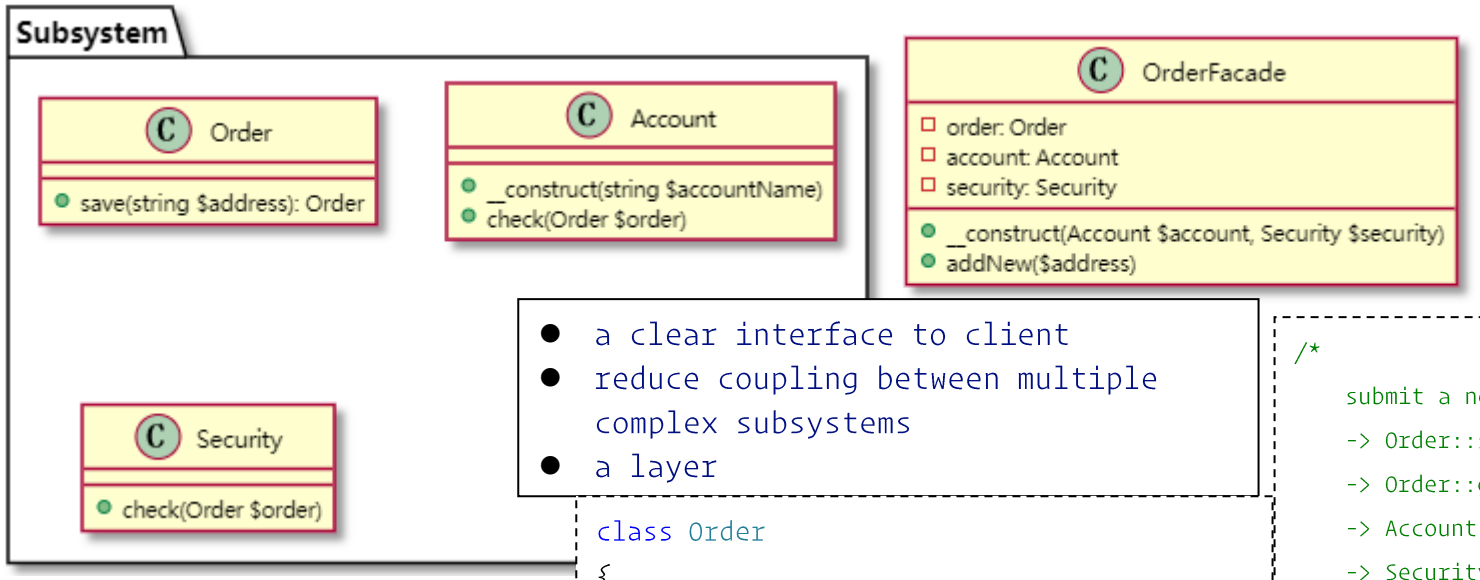
allows behavior to be added to an individual object, dynamically, without affecting the behavior of other objects from the same class

Responsibilities should be added to (and removed from) an object dynamically at run-time.[4]
A flexible alternative to subclassing for extending functionality should be provided.

implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it
perform additional functionality before/after forwarding a request.

```
abstract class FilterDecorator extends Filter
{
    protected Filter $filter;
    public function __construct(Filter $filter)
    {
        $this->filter = $filter;
    }
}
class RealNameFilter extends FilterDecorator
{
    public function check(Request $request)
    {
        echo 'real name check. ';
        $this->filter->check($request);
    }
}
```

Facade



- a clear interface to client
- reduce coupling between multiple complex subsystems
- a layer

```php
class Order
{
    public function save(string $address)
    {
        echo 'new order object created.';
        return $this;
    }
}
```

```php
/*
    submit a new order
    -> Order::save($address)
    -> Order::getOrder()
    -> Account::check(Order $order)
    -> Security::check(Order $order)
*/
$account = new Account('account name');
$security = new Security();
$order = new OrderFacade($account, $security);
$order->addNew('address');
```

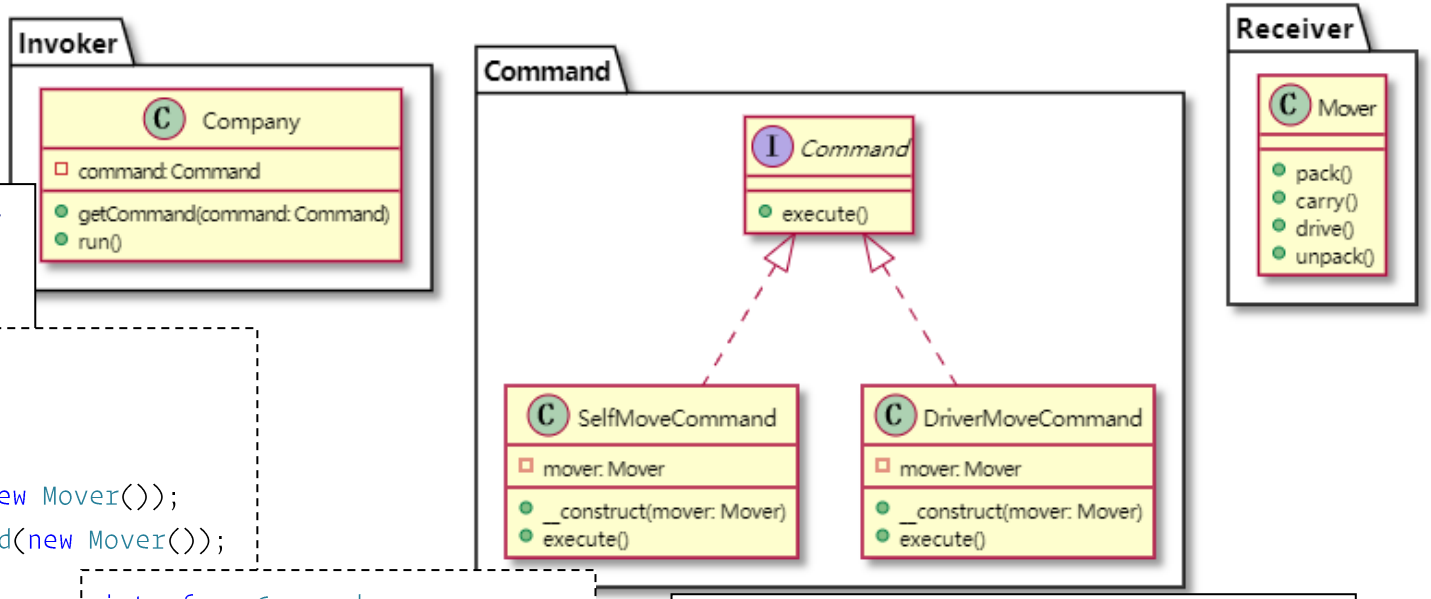
a facade is an object that serves as a front-facing interface masking more complex underlying or structural code

To make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem.
The dependencies on a subsystem should be minimized.

implements a simple interface in terms of (by delegating to) the interfaces in the subsystem and may perform additional functionality before/after forwarding a request.

```php
class OrderFacade
{
    public function addNew($address)
    {
        $this->order = (new Order())->save($address);
        $this->account->check($this->order);
        $this->security->check($this->order);
        echo 'added successfully';
    }
}
```

Command



- decouple Invoker and Receiver
- SCP+OCP
- commonly used

```php
// company = Invoker
$company = new Company();
// mover = Receiver
$command = new SelfMoveCommand(new Mover());
$command2 = new DriverMoveCommand(new Mover());
$company->getCommand($command2);
$company->run();
```

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

```php
class DriverMoveCommand implements Command
{
    private $mover;

    public function __construct(Mover $mover)
    {
        $this->mover = $mover;
    }
    public function execute()
    {
        $this->mover->pack();
        $this->mover->carry();
    }
}
```

Extend

```php
$company = new Company();
$context = $company->getContext();
// $context->setCommand('SelfMove');
$context->setCommand('DriverMove');
$company->run();
```

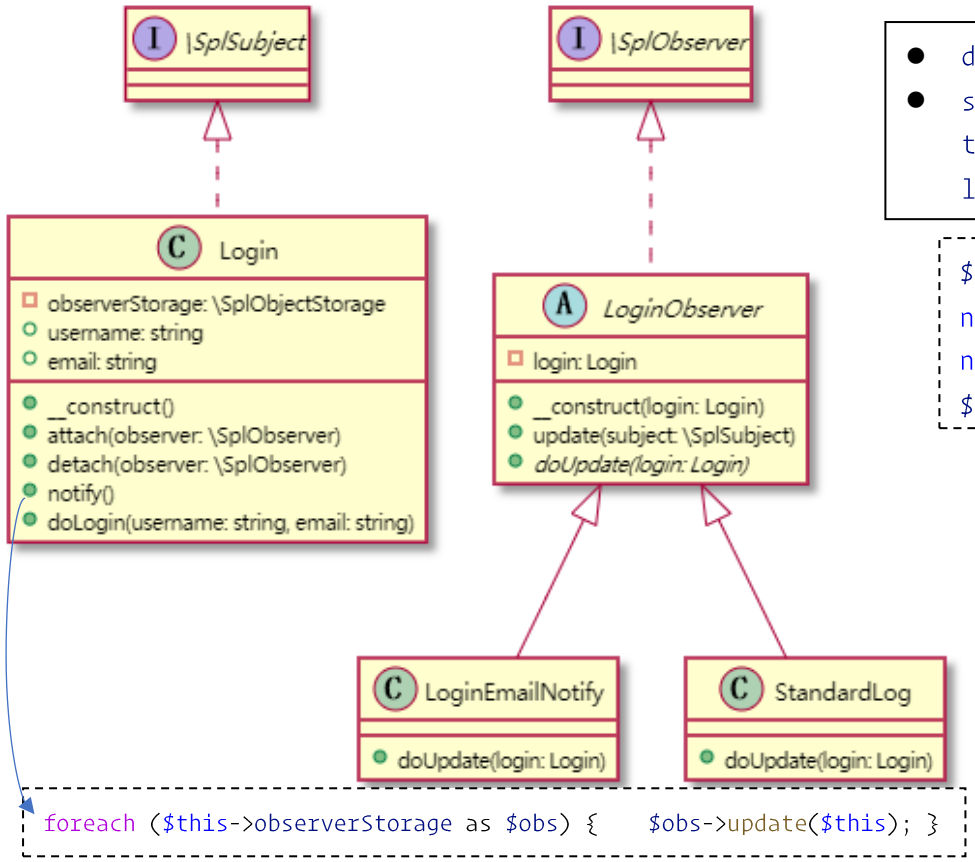
- no receiver
- CommandFactory = Client = Front Controller

an object is used to encapsulate all information needed to perform an action or trigger an event at a later time

Coupling the invoker of a request to a particular request should be avoided. That is, hard-wired requests should be avoided.
It should be possible to configure an object (that invokes a request) with a request.

Define separate (command) objects that encapsulate a request.
A class delegates a request to a command object instead of implementing a particular request directly.

Observer




```
foreach ($this->observerStorage as $obs) {    $obs->update($this); }
```

A one-to-many dependency between objects should be defined without making the objects tightly coupled.
It should be ensured that when one object changes state, an open-ended number of dependent objects are updated automatically.
It should be possible that one object can notify an open-ended number of other objects.

Define Subject and Observer objects.
so that when a subject changes state, all registered observers are notified and updated automatically (and probably asynchronously).

- decouple objects
- subject holds strong references to the observers, may cause memory leaks. use weak references.

```
$login = new Login();
new LoginEmailNotify($login);
new StandardLog($login);
$login->doLogin('UserName', 'admin@aspirantzhang.com');
```

```
abstract class LoginObserver implements \SplObserver
{
    private Login $login;
    public function __construct(Login $login) // here, only Login type parameter is allowed
    {
        $this->login = $login;
        $weakObj = \WeakReference::create($this);
        $login->attach($weakObj->get());
    }
    public function update(\SplSubject $subject)
    {
        if ($subject === $this->login) {    // and here, check types are equal, to ensure that correct class type is used
            $this->doUpdate($subject);
        } else {      echo 'wrong type';    }
    }
    abstract public function doUpdate(Login $login);
}
```

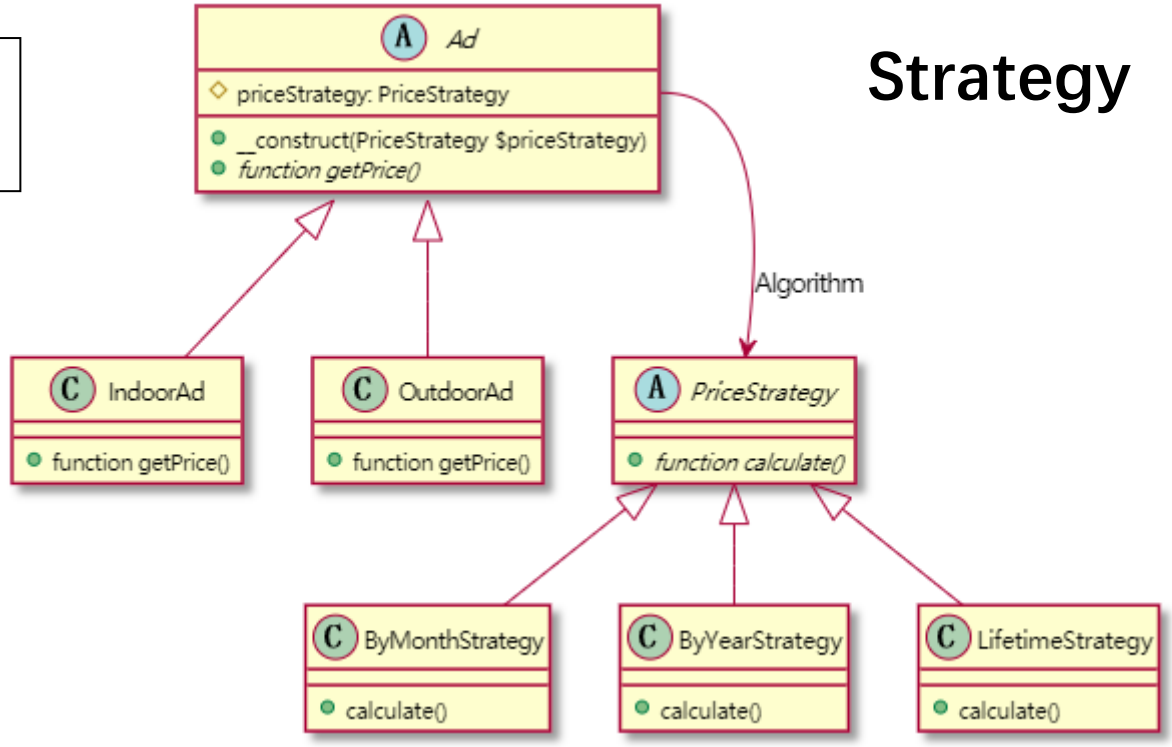
Strategy

- select algorithm at runtime
- instead of inheritance
- good example for open/closed principle

```php
/*
    Ad department        Strategy
    - Indoor Ad          - by Year
    - Outdoor Ad         - by Month
                         - by Lifetime
*/
$ad = new IndoorAd(new ByMonthStrategy());
var_dump($ad->getPrice());
```

```php
abstract class Ad
{
    public function __construct(protected
PriceStrategy $priceStrategy) {}
    abstract public function getPrice();
}
class IndoorAd extends Ad
{
    public function getPrice()
    {
        return
$this->priceStrategy->calculate();
    }
}
```

```php
abstract class PriceStrategy
{abstract public function calculate();}
class ByMonthStrategy extends
PriceStrategy
{
public function calculate()
    {       return 500;    }
}
```



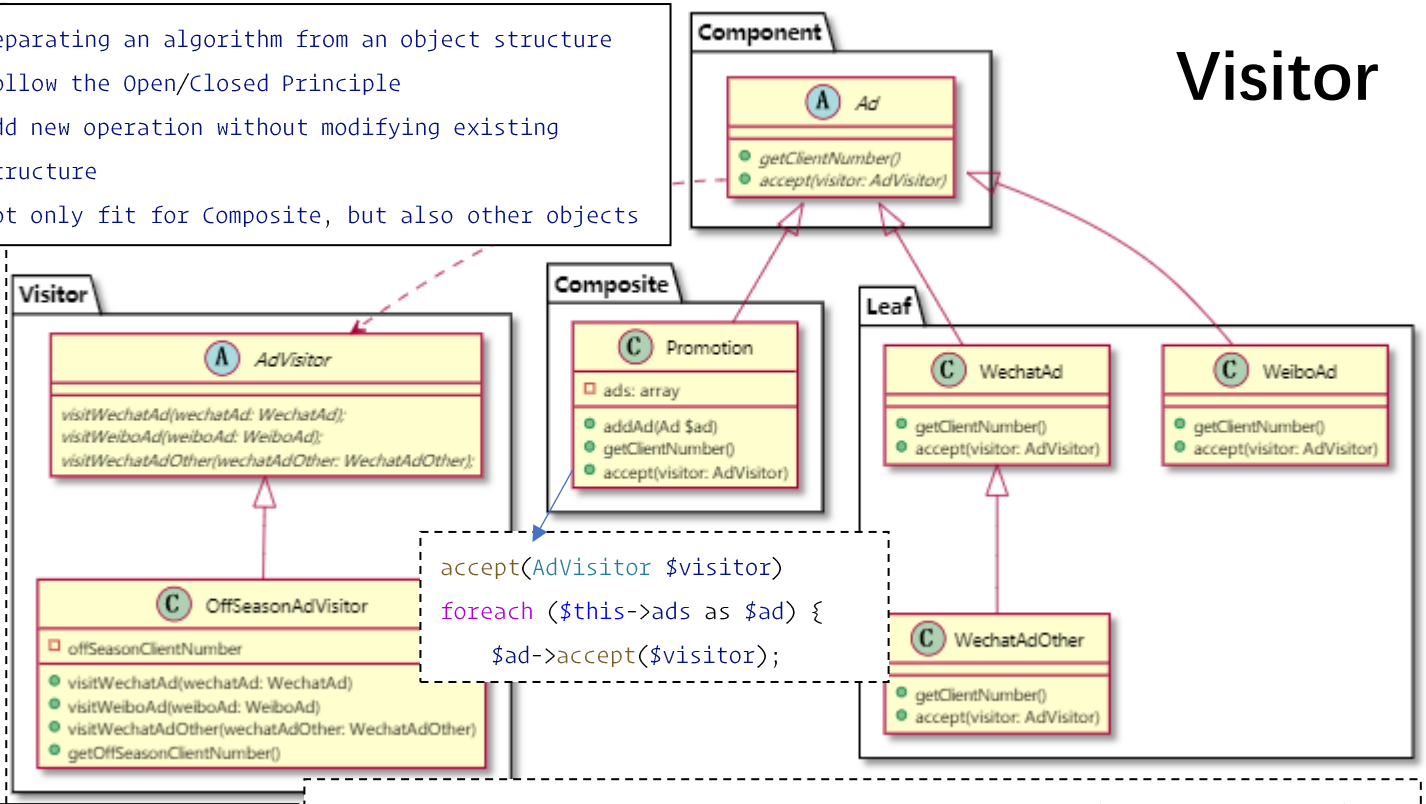
enables selecting an algorithm at runtime

The strategy pattern uses composition instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.

Visitor

```php
$innerTeam = new Promotion();
$innerTeam->addAd(new WeiboAd());
$outerTeam = new Promotion();
$outerTeam->addAd(new WeiboAd());
$innerTeam->addAd($outerTeam);
var_dump($innerTeam->getClientNumber()
);
/*
    Once you give him a key, he can
come unlimited times.
    key  => accept() in leaf
    come => visit() in visitor
 */
$offSeasonVisitor = new
OffSeasonAdVisitor();
$innerTeam->accept($offSeasonVisitor);
var_dump($offSeasonVisitor->getOffSeas
onClientNumber());
```

- separating an algorithm from an object structure
- follow the Open/Closed Principle
- add new operation without modifying existing structure
- not only fit for Composite, but also other objects




```php
accept(AdVisitor $visitor)
foreach ($this->ads as $ad) {
    $ad->accept($visitor);
```

It should be possible to define a new operation for (some) classes of an object structure without changing the classes.

Define a separate (visitor) object that implements an operation to be performed on elements of an object structure.
Clients traverse the object structure and call a dispatching operation accept (visitor) on an element — that "dispatches" (delegates) the request to the "accepted visitor object". The visitor object then performs the operation on the element ("visits the element").

a way of separating an algorithm from an object structure on which it operates.

```php
abstract AdVisitor{ abstract public function visitWechatAd(WechatAd $wechatAd); }
class OffSeasonAdVisitor extends AdVisitor
{   private $offSeasonClientNumber;
    public function visitWechatAd(WechatAd $wechatAd)
    {
        echo 'visitWechatAd';
        echo "\n";
        $this->offSeasonClientNumber += $wechatAd->getClientNumber() * 0.8;
    }
    ......
    public function getOffSeasonClientNumber()
        return $this->offSeasonClientNumber;    }}
```