Tervezési minták

Az informatikában a programtervezési mintának nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.

A tervminták nem nyújtanak kész tervet, amit közvetlenül le lehet kódolni, habár vannak hozzájuk példakódok, amiket azonban meg kell tölteni az adott helyzetre alkalmas kóddal. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

A legtöbb tervminta objektumorientált környezetre van kidolgozva. Mivel a funkcionális programozás kevéssé ismert és használt, arra a környezetre még csak kevés tervminta ismert, például a monád. Az objektumorientált minták közül nem mindegyiket lehet, és nem mindegyiket érdemes itt használni. Van, amit módosítani kell.

A tervezési minták kategóriái

- Létrehozási minták
- Szerkezeti minták
- Viselkedési minták

Létrehozási minták

Ezek a minták objektumok létrehozására valók. Bizonyos helyzetekben az objektumok közvetlen létrehozása elbonyolíthatja a terveket vagy a kódot. A létrehozási minták ezt egyszerűsítik, és a létrehozás felelősségét a tervmintában szereplő objektumokra ruházza át. Ezek magukba zárják a folyamatot, így azzal a kliensnek nem kell foglalkoznia. A másik alapelv szerint elrejtik a termék konkrét típusát, a létrehozás folyamatát, és azt, hogy hogyan kombinálódnak.

Ebbe a csoportba a következő minták tartoznak: * Simple Factory * Factory Method * Abstract Factory * Builder * Prototype * Singleton

Abstract Factory

Az abstract factory programtervezési minta módot nyújt arra, hogy egységbe zárjuk közös témához kapcsolódó egyedi gyártó metódusok egy csoportját anélkül, hogy specifikálnák azok konkrét osztályait. Normál használatban, a kliens szoftver létrehozza az absztrakt gyár egy konkrét implementációját, és aztán a gyár általános interfészét használja a témához kapcsolódó konkrét objektumok létrehozásához. A kliens nem tudja (vagy nem törődik vele), milyen konkrét objektumokat kap ezekből a belső gyárakból, mivel csak a termékeik általános interfészét használja. Ez a tervezési minta szétválasztja egymástól objektumok egy csoportjának implementációját azok általános használatától és objektum összetételre hagyatkozik, mivel az objektumok létrehozása olyan metódusokban van implementálva, amik a gyár interfészén vannak ismertté téve számára.

Egy gyár, egy konkrét osztály helye a kódban, ahol az objektumok létrejönnek. Az absztrakt gyár minta használatának szándéka arra irányul, hogy elszigetelje egymástól az objektumok létrehozását azok használatától, és hogy egymással összefüggő objektumok családjait hozza létre anélkül, hogy azok konkrét osztályaitól függene. Ez lehetővé teszi új származtatott típusok bevezetését, anélkül, hogy az ős osztályokat használó kódot meg kellene változtatni.

Pédakód

Először készítsünk egy `Door` interfészt valamint néhány implementációt.

```
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';
    }
}
```

Ezután minden egyes ajtótípushoz rendeljünk különböző ajtószerelő szakértőket:

```
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';
    }
}
```

Majd az absztrakt gyárunk lehetővé teszi, hogy az összekapcsolódó objektumainkat egységbe zárjuk.

```
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();
    }
}
```

A következőképp használható:

```
$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
```

Prototype

A minta lényege a klónozás, azaz az eredeti objektummal megegyező új példány létrehozása. Az egyszerű értékadás erre nem alkalmas, mivel az csak az objektum hivatkozását másolja le, melynek eredményeképpen az eredeti példány és másolata ugyanoda hivatkozik. Két típust különböztetünk meg, a sekély és a mély klónozást. A sekély klónozás esetében az osztály által hivatkozott objektumokat ugyanúgy másoljuk, mint elemi típusú tulajdonságait. A mély klónozásnál az osztály által hivatkozott objektumokat is klónozzuk.

Példakód

PHP-ban egyszerű a megvalósítása a `clone` használatával.

```
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;
    }
}
```

Ezután a klónozása a következőképp történik:

```
 $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
```

A `__clone` metódus felüldefiniálásával tudunk módosítsani a klónozás működésén.

Mikor használjuk?

Ha egy olyan objektumra van szükség, amely egy meglévőhöz nagyon hasonlít, vagy amikor a létrehozás a klónozáshoz képest drágább folyamat lenne.

Singleton

A singleton design pattern egy olyan programtervezési minta, amely egy objektumra korlátozza egy osztály létrehozható példányainak a számát. Gyakran előfordul, hogy egy osztályt úgy kell megírni, hogy egyetlen példány lehet belőle. Az objektumorientált paradigmából jól ismert, hogy egy osztályból példányt a konstruktorán keresztül lehet készíteni.

Ha van publikus konstruktor az osztályban, akkor akárhány példány készíthető belőle. Tehát publikus konstruktora nem lehet a singletonnak. Viszont ha nincs konstruktor, akkor nem hozható létre a példány, amin keresztül meghívhatjuk a metódusait. A megoldást az osztályszintű (statikus) metódusok jelentik. Ezeket akkor is meg lehet hívni, ha nincs példány.

A singletonnak van egy olyan osztályszintű metódusa, amely minden hívójának ugyanazt az objektumot adja vissza. Ennek a neve konvenció szerint `getInstance` szokott lenni.

Természetesen ezt a példányt is létre kell hozni. Ehhez egy privát konstruktort kell készíteni, amit a `getInstance` metódus meghívhat.

Pédakód

```


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
    }
}
```

Hívása a következőképp történik:

```
$president1 = President::getInstance();
$president2 = President::getInstance();

var_dump($president1 === $president2); // true
```

Szerkezeti minták

A szerkezeti minták lényege az osztályokból és objektumokból nagyobb szerkezetek létrehozása. Amennyiben osztályokkal dolgozunk, öröklléssel vagy felületek megvalósításával összetételeket hozunk létre, például több interfész implementálásával egy többféleképpen is viselkedő osztályt. Objektumok esetében dinamikus „összeragasztásukkal" érjük el az olyan összetételek létrejöttét, amelyek kellően rugalmasak.

Ebbe a csoportba a következő minták tartoznak: * Adapter * Bridge * Composite * Decorator * Facade * Flyweight * Proxy

Decorator pattern

A díszítő programtervező minta egy olyan programtervezési minta, amely lehetővé teszi adott objektumokhoz más viselkedések hozzáadását akár statikusan, akár dinamikusan anélkül, hogy hatással lenne az azonos osztályból származó többi objektumra. A díszítő gyakran alkalmas arra is, hogy a program megfeleljen az egyértelmű felelősség elvének, mivel lehetővé teszi a felelősségek egyértelmű felosztását különböző osztályok között.

Pédakód

A következő példa a díszítők használatát szemlélteti a kávékészítés esetén. Ebben a példában az esetünk csak a költségeket és a leírásokat tartalmazza.

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

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

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

A következő osztályok tartalmazzák a díszítőket az összes `Coffee` osztályra, beleértve a díszítő osztályokat magukat.

```
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';
    }
}
```

Hívása a következőképp történik:

```
$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 pattern

// TODO

Flyweight pattern

A pehelysúlyú objektum egy olyan objektum, amely minimalizálja memória használatot azzal, hogy annyi adatot oszt meg, amennyi csak lehetséges más hasonló objektumokkal. Ez a nagyszámú objektumok használatának az a módja, mikor egy egyszerű ismételt reprezentáció használna fel nem elfogadható mennyiségű memóriát. Gyakran az objektum állapotának egyes részei megoszthatók, gyakorlatilag külső adat struktúrákban tároljuk őket, és csak ideiglenesen adjuk át a pehelysúlyú objektumoknak a felhasználás során.

Példakód

A következő példa teák rendelését és felszolgálását teszi lehetővé. Először készítsünk tea típusokat és tea készítőt.

```
// 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];
    }
}
```

Ezután a `TeaShop` végzi a megrendelések felvételét és kiszolgálását.

```
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;
        }
    }
}
```

Használata a következőképp történik:

```
$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 pattern

// TODO

Viselkedési minták

A viselkedési minták elsősorban algoritmusokkal, illetve az osztályok és objektumok közötti kommunikációval, a felelősségi körök kijelölésével foglalkoznak. Bonyolult vezérlési folyamatokat modelleznek az osztályok és objektumok közötti interakciókra lebontva, így könnyebben átláthatóvá téve azokat. Az osztályminták öröklést, az objektumminták aggregációt alkalmaznak ennek érdekében.

Ebbe a csoportba a következő minták tartoznak: * Chain of Responsibility * Command * Iterator * Mediator * Memento * Observer * Visitor * Strategy * State * Template Method

Mediator pattern

A programtervezésben a mediátor minta egy olyan objektum, mely megszabja, hogy hogyan viselkedjen objektumok egy csoportja. Ezt a mintát a viselkedési minták közé soroljuk annak köszönhetően, hogy képes megváltoztatni a program futási viselkedését.

Általában egy program nagyszámú csoportot tartalmaz, a logika és a számítás ezek között az osztályok között oszlik meg. Azonban ahogy újabb és újabb osztályokat készítünk egy programhoz, főleg karbantartás vagy refactoring közben, sokkal összetettebbé válhat a kommunikáció ezen osztályok között. A program ezáltal nehezen olvashatóvá és kezelhetővé válik. Továbbá nehezebb lesz megváltoztatni a programot, mivel bármilyen változás további osztályok kódjának megváltoztatását eredményezheti.

A Mediátor minta alkalmazásakor az objektumok közötti kommunikációt mediátor objektum végzi. Az objektumok ezentúl nem közvetlenül egymással kommunikálnak, hanem a Mediátoron keresztül. Ez csökkenti a kommunikáló objektumok közötti függőséget, ezáltal csökkenti a csatolás mértékét.

Példakód

```
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;
    }
}

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);
    }
}
```

Használata a következőképpen történik:

```
$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 pattern

A Memento programtervezési minta biztosítja, hogy egy objektum visszaállítható legyen az előző állapotába.

A Memento minta három objektummal implementálható: originator (kezdeményező), caretaker (gondnok), memento (emlékeztető). Az originator egy objektum, aminek van valamilyen belső állapota. A caretaker valamit módosít az originator belső állapotán, azonban azt akarja, hogy lehetősége legyen visszavonni a változásokat. A caretaker először kér az originator-tól egy memento objektumot. Ezután valamilyen műveletet (vagy műveleteket) végez az originator-on, majd, ha vissza akarja állítani az előző állapotot, akkor visszaadja az originato-nak a memento objektumot. A memento objektum önmagában egy átlátszatlan objektum (a caretaker nem változtathatja meg, vagy nem ajánlott megváltoztatnia). Amikor ezt a mintát használjuk, ügyelni kell arra, hogy ez csak egy objektummal foglalkozik, az originator eközben megváltoztathat más objektumokat vagy erőforrásokat.

Példakód

```
class EditorMemento
{
    protected $content;

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

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

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();
    }
}
```

Használata a következőképpen történik:

```
$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 pattern

Az Observer, vagy Megfigyelő minta egy olyan szoftvertervezési minta, melyben egy objektum, melyet alanynak hívunk, listát vezet alárendeltjeiről, akiket megfigyelőknek hívunk és automatikusan értesíti őket bármilyen állapotváltozásról, többnyire valamely metódusuk meghívásán keresztül. Többnyire elosztott eseménykezelő rendszerek kialakításakor használjuk. A Megfigyelő minta kulcsfontosságú része az ismert Model-View-Controller (MVC, Modell-Nézet-Vezérlő) architekturális modellnek. A Megfigyelő mintát számos programozási könyvtár és rendszer alkalmazza, többek között szinte minden GUI toolkit.

A Megfigyelő minta memóriaszivárgást okozhat, melyet elévülő hallgató problémaként is ismerhetünk, mivel az alap implementáció során explicit regisztrációt és explicit deregisztrációt is végrehajtunk, mint a megsemmisítési mintában, mivel az alany erősen kötődik a megfigyelőkhöz, hogy azok aktívak maradjanak. Ezt megelőzhetjük, ha az alanyok csak gyengén kötődnek a megfigyelőkhöz.

A megfigyelőt gyakran úgy implementálják, hogy az alany annak az objektumnak a része, aminek az állapotát megfigyelik, és ha szükséges, akkor értesítse a változásról a megfigyelőket. Ez szoros csatolást eredményez, így az alany és a megfigyelők tudnak egymásról, és ami még rosszabb, turkálhatnak egymásban. Ez rontja a sebességet, a skálázhatóságot, a karbantarthatóságot, a biztonságot és a rugalmasságot.

Egy másik megvalósítási mód a fel- és leiratkozási minta, ahol üzenetekkel kommunikálnak. Ez tartalmaz egy üzenetsor szervert és egy üzenetkezelőt is. Ez lehetővé teszi, hogy a megfigyelők és az alanyok lazábban kapcsolódjanak, és még csak ne ismerjék egymást, hiszen a üzenetszerver közvetít közöttük. De a fel- és leiratkozás minta máshogy is használható, amivel jelek és értesítések küldhetők; ezzel hasonló eredmény érhető el, a megfigyelő minta használata nélkül.

Példakód

```
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();
    }
}

class JobPostings 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);
    }
}
```

Egy példa a használatára:

```
// Create subscribers
$johnDoe = new JobSeeker('John Doe');
$janeDoe = new JobSeeker('Jane Doe');

// Create publisher and attach subscribers
$jobPostings = new JobPostings();
$jobPostings->attach($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 pattern

Az objektumorientált programozásban és a szoftverfejlesztésben a látogató tevezési minta segítségével tudjuk szétválasztani az algoritmust és az objektum szerkezetét. A gyakorlati eredménye ennek a szétválasztásnak az, hogy képessé válik a program arra, hogy új műveleteket adjunk hozzá a létező objektumstruktúrához anélkül, hogy módosítanánk annak szerkezetét. Ez az egyik módja az Open-Closed tervezési alapelv megvalósításának.

A lényege, hogy lehetővé teszi, hogy egy új virtuális funkciót adjunk az osztályokhoz anélkül, hogy az osztályok szerkezetét meg kellene változtatni. Helyette létrejön egy látogató osztály, amely implementálja az összes létező megvalósítását az adott virtuális funkciónak. A látogató tartalmazza a példány referencia beviteli értékét, és lehetővé teszi a dupla küldést.

Pédakód

```
// 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);
}
```

Ezután készítsünk implementációkat az Animal interfészből:

```
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);
    }
}
```

Ezután implementáljuk a visitort:

```
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();
    }
}
```

Egy példa a használatára:

```
$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!
```

Ha most hozzá akarnánk egy új viselkedést adni az Animal osztályhoz, azt a következőképpen tehetjük meg:

```
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';
    }
}
```

Egy példa a használatára:

```
$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 pattern

A számítógép-programozásban a stratégia minta (vezérelv mintaként is ismert) egy szoftvertervezési minta, amely lehetővé teszi, hogy egy algoritmus viselkedését a futás során válasszuk meg. A stratégia minta meghatározza az algoritmusok egy családját, egységbe foglal minden algoritmust, és a családon belül cserélhetővé teszi ezeket az algoritmusokat. A stratégia segítségével az algoritmus az őt használó kliensektől függetlenül változhat, miután az megtette a beállításokat.

Példakód

```
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;
    }
}

class Sorter
{
    protected $sorter;

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

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

Egy példa a használatára:

```
$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
```

Template method pattern

A számítógép-programozásban a sablonfüggvény programtervezési minta egy olyan viselkedési tervezési minta, amely egy sablonfüggvény nevű metódus algoritmusával definiálja a program vázát. Ezek némely lépése felüldefiniálható alosztályokban. Lehetővé teszi egyes algoritmusok lépéseinek az újradefiniálását anélkül, hogy az algoritmus struktúrája megváltozna.

Az objektumorientált programozásban az elsőnek létrehozott osztály biztosítja a tervezési algoritmus alaplépéseit. Ezek a lépéseket az absztrakt metódusok valósítják meg. Később, az alosztályokban az absztrakt metódusokat megváltoztatva jönnek létre a valódi változások. Így az általános algoritmus egy helyen van definiálva, de a konkrét lépések változtathatóak az alosztályokon keresztül.

A sablonfüggvény tervezési minta gyakran előfordul, legalábbis a legegyszerűbb esetben, ahol a metódus csak egyetlenegy absztrakt metódust hív meg egy objektumorientált nyelvben. Ha egy szoftverprogramozó többalakú metódust használ végig, lehet, hogy ez a tervezési minta lesz a természetes velejárója. Ez azért van, mert egy absztrakt vagy többalakú függvény hívása maga az indoka az absztrakt vagy többalakú metódusnak. A sablonfüggvény mintát arra is lehet használni, hogy azonnal plusz értéket adjunk egy szoftvernek.

A sablonfüggvény minta implementációk valósítják meg a védett változók GRASP elvet, mint ahogy az illesztő minta teszi. A különbség annyi, hogy az illesztő minta ugyanazt az interfészt adja néhány operációval, míg a sablonfüggvény minta csak egyet ad egynek.

Példakód

Először is megvan a sablonfüggvényünk, ami meghatározza az algoritmus vázát.

```
 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();
}
```

Ezután megvalósíthatjuk az implementációkat:

```
 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';
    }
}
```

Egy példa a használatára:

```
$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
```