Singleton (unikat)

Pri razvoju web aplikacija, često ima smisla konceptualno i arhitekturalno omogućiti korišćenje samo jedne instance određene klase. Singleton patern omogućava upravo tako nešto.

TODO: NEED NEW SINGLETON CODE EXAMPLE

Ovaj kôd implementira singleton patern koristeći statičku promenljivu i statički getInstance() metod. Obratite pažnju i na sledeće:

Konstruktor \_\_construct() je deklarisan kao protected kako bi izvan klase bilo onemogućeno kreiranje Singleton instance preko new operatora.

Magični metod \_\_clone() je deklarisan kao private kako bi bilo onemogućeno kloniranje instance preko clone operatora.

Magični metod \_\_wakeup() je deklarisan kao private kako bi bila onemogućena deserijalizacija instance preko globalne funkcije unserialize().

Nova instanca se kreira preko late static binding mehanizma u statičkom metodu getInstance() putem ključne reči static. Upravo ovo omogućava nasleđivanje klase Singleton u primeru.

Signleton patern je koristan u situacijama kada treba da obezbedimo da imamo samo jednu instancu neke klase u toku jednog kompletnog ciklusa u web aplikaciji. Tipičan primer su globalni objekti (kao što je neka Configuration klasa) ili deljeni resursi (kao što je event queue).

Treba da budete oprezni pri korišćenju Singleton paterna, jer sama njegova priroda uvodi globalno stanje u vašu aplikaciju, čime se smanjuje njena testabilnost. U većini slučajeva, dependency injection princip može (i treba) da se koristi umesto Singleton klasa. Korišćenjem dependency injection-a ne uvodimo nepotrebne direktne zavisnosti u dizajn naše aplikacije, jer objekat koji bude koristio taj neki deljeni ili globalni resurs neće imati znanja o tome o kojoj se tačno klasi radi.

Strategy

Primenom Strategy paterna enkapsulirate grupu određenih algoritama, pri čemu je klijentska klasa odgovorna za instanciranje konkretnog algoritma, bez znanja o načinu na koji je on implementiran. Postoji nekoliko varijacija ovog paterna, a najjednostavniji od njih će biti demonstriran u nastavku.

Prvi deo prikazuje grupu algoritama za ispis nekog niza podataka: jedan vrši nativnu, drugi radi JSON serijalizaciju, a treći ga ostavlja netaknutim:

<?php

interface OutputInterface

{

public function load();

}

class SerializedArrayOutput implements OutputInterface

{

public function load()

{

return serialize($arrayOfData);

}

}

class JsonStringOutput implements OutputInterface

{

public function load()

{

return json\_encode($arrayOfData);

}

}

class ArrayOutput implements OutputInterface

{

public function load()

{

return $arrayOfData;

}

}

Enkapsuliranjem ovih algoritama u zasebne klase, na elegantan i čist način stavljate do znanja drugim programerima da lako mogu da dodaju novu output strategiju, bez uticaja na klijenstki kôd.

Primetićete da svaka ‘output’ klasa implementira određeni OutputInterface. Ovaj interfejs ima za cilj ima da definiše jednostavan “ugovor” koji svaka nova implementacija mora da ispoštuje. Takođe, implementiranjem jednog zajedničkog interfejsa, kao što ćete i videti u nastavku, biće omogućena primena Type Hinting-a, kako bi se obezbedilo to da kôd koji koristi ovu funkcionalnost radi sa ispravnim tipovima klasa, u ovom slučaju ‘OutputInterface’ implementacijama.

Sledeći primer kôda demonstrira kako poziv klijentske klase može da koristi neki od ovih algoritama tako što će ga zahtevati prilikom izvršavanja:

<?php

class SomeClient

{

private $output;

public function setOutput(OutputInterface $outputType)

{

$this->output = $outputType;

}

public function loadOutput()

{

return $this->output->load();

}

}

Ova klijentska klasa ima privatno svojstvo koje mora da bude prosleđeno prilikom instanciranja, pri čemu to mora da bude ‘OutputInterface’ implementacija. Nakon što je ovo svojstvo postavljeno, poziv loadOutput() metoda će pozvati load() metod neke konkretne Output klase.

<?php

$client = new SomeClient();

// Želite niz?

$client->setOutput(new ArrayOutput());

$data = $client->loadOutput();

// Želite JSON?

$client->setOutput(new JsonStringOutput());

$data = $client->loadOutput();

Factory (fabrika)

Jedan od najčešće korišćenih dizajn paterna je Factory. U njegovom slučaju, sve što neka klasa radi jeste kreiranje objekta kojeg nameravate da koristite. Pogledajmo sledeći primer Factory paterna:

<?php

class Automobile

{

private $vehicleMake;

private $vehicleModel;

public function \_\_construct($make, $model)

{

$this->vehicleMake = $make;

$this->vehicleModel = $model;

}

public function getMakeAndModel()

{

return $this->vehicleMake . ' ' . $this->vehicleModel;

}

}

class AutomobileFactory

{

public static function create($make, $model)

{

return new Automobile($make, $model);

}

}

// Factory kreira Automobile objekat

$veyron = AutomobileFactory::create('Bugatti', 'Veyron');

print\_r($veyron->getMakeAndModel()); //ispisuje "Bugatti Veyron"

Ovaj kôd koristi Factory za kreiranje instance Automobile klase. Postoje dva moguća benefita pisanja kôda na ovaj način. Prvi je to što ćete u slučaju izmene, preimenovanja ili zamene Automobile klase, morati da menjate jedino kôd u njenom factory-u, umesto na svakom mestu gde se koristi klasa Automobile. Druga prednost je ta što u slučaju da je sâmo kreiranje objekta kompleksna operacija, sav taj posao možete obaviti u factory klasi, umesto da ga ponavljate svaki put kada vam treba nova instanca.

Korišćenje Factory paterna nije uvek neophodno (ispravno). Prethodni primer je toliko prost da bi korišćenje factory-a bio samo nepotrebna komplikacija. Ipak, ako radite na velikom i kompleksnom projektu, korišćenje factory-a vas možete lišiti dosta problema.