KAPITOLA 2

Návrhový vzor Singleton

Jazyk PHP 5 vám pomocí klíčových slov `public`, `protected` a `private` umožňuje kontrolovat, kdo získá přístup k určitým atributům a metodám třídy. Dále vám jazyk PHP 5 umožňuje omezit to, co může být při odvozování tříd přepsané, a dokonce i to, co přepsané být musí. Jedno vám však jazyk PHP neumožňuje: nedokážete totiž omezit počet instancí dané třídy. A právě k tomuto účelu slouží návrhový vzor *Singleton*.

Návrhový vzor Singleton je jeden z nejjednodušších návrhových vzorů a zároveň i jeden z nejpoužívanějších. Možná jste jej už použili, aniž byste věděli, že se jedná o návrhový vzor.

Problém

Ve vzorovém příkladu z předchozí kapitoly jste vyčlenili kód pro ladění aplikací ze třídy `Library` a zapouzdřili jej do nové třídy. Tím jste umožnili tomuto kódu spolupracovat i s ostatními třídami. Při vytváření instance třídy `Library` jednoduše předáte implementaci rozhraní `Debugger` konstruktoru:

```
$debugger = new DebuggerEcho();
$library = new Library($debugger);
```

Chcete-li odladit i třídy `Book` a `Member`, může příslušný kód vypadat následovně:

```
$debuggerBook = new DebuggerEcho();
$book = new Book($debuggerBook, 'PC', 100);

$debuggerMember = new DebuggerEcho();
$member = new Member($debuggerMember, 1, 'Marian Böhmer');
```

Tímto způsobem jste vytvořili tři instance stejné třídy a tím spotřebovali skoro třikrát tolik paměti. Knihovna však obsahuje určitě více než jednu publikaci a má registrovaných více než jednoho člena. Spotřeba paměti tedy bude se zvyšujícím se počtem publikací a členů knihovny neustále narůstat. Když se ale na objekt `Debugger` podíváte blíže, hned vás napadne, že není potřebné pro každou publikaci a každého člena používat samostatný objekt typu `Debugger`; tento objekt se nakonec používá jen na vypsání předaných hlášení a neví nic o objektu, který jej používá.

Ideální by bylo, kdybyste mohli pro každou publikaci a každého člena knihovny používat stejný objekt typu `Debugger` – tím byste ušetřili množství paměti a vyhnuli se zbytečnému vytváření dalších objektů. S růstem knihovny budou vytvářené objekty typu `Debugger` na stále nových místech, a nikdy tak nebudete mít jistotu, který z nich byl vytvořený jak první. V takovém případě potřebujete k objektu typu `Debugger` centrální přístupový bod.

Účel návrhového vzoru

Tohoto centrálního přístupového bodu docílíte pomocí návrhového vzoru *Singleton*.

Návrhový vzor Singleton zajistí, že z určité třídy může existovat nejvíce jedna instance, a poskytne k ní globální přístupový bod.

Pro aplikování tohoto vzoru na výše popsaný problém je nutné vykonat následující kroky:

1. Poskytnout centrální bod pro přístup k instanci třídy `Debugger`.
2. Tento centrální přístupový bod musí vždy nabízet přístup ke stejnému objektu nezávisle na počtu volaní.
3. Zabránit možnosti vytvoření další instance třídy.

Implementace

Vytvoříte-li v kódu objekt typu `Debugger` na místě, kde jej chcete použít, nemáte žádnou možnost zjistit, zda už daný objekt existuje. Pro vytvoření instance objektu na centrálním místě přesuňte tento kód do nové metody. Protože k vytvoření objektu nejsou nutné žádné další informace, použijte k tomu statickou metodu, tj. metodu třídy, kterou lze zavolat bez nutnosti vytvoření instance této třídy:

```
namespace cz\k1886\debuggers;

class DebuggerEcho implements Debugger {

    public static function getInstance() {
        $debugger = new self();
        return $debugger;
    }

    public function debug($message) {
        print $message . "\n";
    }
}
```

Ve statické metodě `getInstance()` vytvoříte novou instanci třídy `DebuggerEcho`, kterou následně vrátíte. Místo vytvoření objektu typu `Debugger` pomocí operátoru `new` můžete nyní k tomuto účelu použít novou metodu:

```
use cz\k1886\debuggers\DebuggerEcho;

$debugger1 = DebuggerEcho::getInstance();
$debugger1->debug('Lorem ipsum dolor sit amet.');

$debugger2 = DebuggerEcho::getInstance();
$debugger2->debug('Proin fringilla bibendum sagittis.');

if ($debugger1 === $debugger2) {
    print '$debugger1 === $debugger2';
} else {
    print '$debugger1 !== $debugger2';
}
```

V tomto příkladu objekt nevytváříme přímo v místě, kde jej potřebujeme, nicméně počet vytvořených objektů se nezměnil, protože při každém volání metody `getInstance()` se vytvoří nový objekt typu `Debugger`. To potvrzuje i výpis z příkladu:

```
Lorem ipsum dolor sit amet.
Proin fringilla bibendum sagittis.
$debugger1 !== $debugger2
```

Z toho vyplývá, že metoda `getInstance()` by měla obsahovat více logiky a při každém volaní provést následující kroky:

1. Při volání ověřit, zda už existuje instance třídy.
2. Pokud ne, vytvořit instanci třídy pomocí operátoru `new` a uložit ji.
3. Vrátit uloženou instanci.

Aby bylo možné použít stejný objekt vícekrát, musí být uložený v rámci metody `getInstance()`. Vzhledem k tomu, že tato metoda je definovaná jako statická, musí být atribut, který bude uchovávat instanci dané třídy, také definovaný jako statický:

```
namespace cz\k1886\debuggers;

class DebuggerEcho implements Debugger {
    private static $instance = null;

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

    public function debug($message) {
        print $message . "\n";
    }
}
```

Třídu `DebuggerEcho` jste v tomto kroku doplnili o statický atribut `$instance`. Metoda `getInstance()` při jejím volání ověří, zda `$instance` již obsahuje instanci třídy. Pokud ne, vytvoří se nová instance, která se přiřadí do atributu a kterou následně metoda vrátí.

Při opětovném provedení testovacího skriptu již docílíme požadovaného efektu:

```
Lorem ipsum dolor sit amet.
Proin fringilla bibendum sagittis.
$debugger1 === $debugger2
```

V této chvíli je úplně jedno, jak často se bude metoda `getInstance()` volat, aplikace bude používat vždy stejný objekt, čímž se ušetří systémové prostředky.

Skryté problémy

Návrhový vzor Singleton s sebou nese i pár skrytých problémů. Co se stane, pokud vaši týmoví kolegové nebudou vědět, že pro získání objektu typu `Debugger` mají použít metodu `getInstance()`? Vytvoří jej klasickým způsobem:

```
use cz\k1886\debuggers\DebuggerEcho;

// váš objekt debugger1
$debugger1 = DebuggerEcho::getInstance();
$debugger1->debug('Lorem ipsum dolor sit amet.');

// objekt debugger2 vašeho kolegy
$debugger2 = new DebuggerEcho();
$debugger2->debug('Proin fringilla bibendum sagittis.');

if ($debugger1 === $debugger2) {
    print '$debugger1 === $debugger2';
} else {
    print '$debugger1 !== $debugger2';
}
```

Vytvořené objekty znovu nejsou stejné a znovu dochází k plýtvání pamětí. Musíte tedy svým týmovým kolegům zakázat možnost vytvářet instance samostatně a tím je donutit používat metodu `getInstance()`. Řešení tohoto problému je velmi jednoduché. Použití konstruktoru mimo třídu zakážete tak, že jej deklarujete jako `protected`:

```
namespace cz\k1886\debuggers;

class DebuggerEcho implements Debugger {

    // ... statický atribut a metoda getInstance()

    protected function __construct() {}

    public function debug($message) {
        print $message . "\n";
    }
}
```

Pokud se v této chvíli pokusí některý z vašich kolegů vytvořit novou instanci objektu typu `Debugger`, zareaguje na to interpret jazyka PHP následující chybou:

```
Fatal error: Call to protected cz\k1886\debuggers\DebuggerEcho::
__construct() from invalid context in test.php on line 10
```

Váš kolega bude muset místo toho použít k získání objektu metodu `getInstance()`.

Vynalézaví kolegové, kteří přesto budou chtít vytvořit novou instanci objektu, by mohli přijít na myšlenku klonovat objekt, který získají pomocí metody `getInstance()`:

```
use cz\k1886\debuggers\DebuggerEcho;

$debugger1 = DebuggerEcho::getInstance();
$debugger1->debug('Lorem ipsum dolor sit amet.');

$debugger2 = clone $debugger1;
$debugger2->debug('Proin fringilla bibendum sagittis.');

if ($debugger1 === $debugger2) {
    print '$debugger1 === $debugger2';
} else {
    print '$debugger1 !== $debugger2';
}
```

Po provedení tohoto testovacího skriptu se znovu ukáže, že se používají dvě instance třídy. I když vám nikdo takovéto kolegy nepřeje, nemůžete si být se současným řešením problému nikdy jistí, že skutečně existuje jen jedna instance třídy. Z toho vyplývá, že jste požadavek číslo tři zatím nesplnili na 100 %.

Naštěstí vám jazyk PHP také umožňuje změnit chování třídy při klonování, k čemuž stačí implementovat metodu s názvem `__clone()`. Klonování objektu zakážete tím, že zakážete volání metody `__clone()` mimo danou třídu:

```
namespace cz\k1886\debuggers;

class DebuggerEcho implements Debugger {

    // ... statický atribut a metoda getInstance()

    protected function __construct() {}

    private function __clone() {}
```

```
    public function debug($message) {
        print $message . "\n";
    }
}
```

Při opakovaném pokusu klonovat objekt typu `Debugger` obdrží váš kolega chybové hlášení, které mu oznamuje, že klonování objektu je zakázané:

```
Fatal error: Call to private cz\k1886\debuggers\DebuggerEcho::
__clone() from context '' in test.php on line 10
```

Tím jste zabezpečili, že vždy může existovat maximálně jeden objekt třídy `DebuggerEcho`, a implementovali jste tudíž svého prvého jedináčka.

POZNÁMKA

Možná se divíte, proč byl konstruktor deklarovaný jako `protected` a metoda `__clone()` jako `private`. Důvodem jsou možnosti, které poskytuje dědění. V případě konstruktoru chcete umožnit třídám, které jsou potomky třídy `DebuggerEcho`, aby jej mohly přepsat. Takto deklarovaný konstruktor není sice možné použít mimo třídu, avšak je možné jej použít na jeho obvyklé úlohy.

V případě metody `__clone()` toto není nutné a má být jen zamezení její používání.

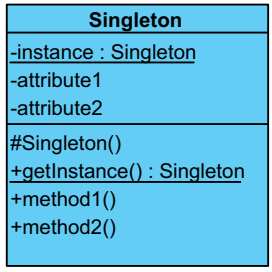
Definice

Návrhový vzor Singleton zajistí, že z určité třídy může existovat nejvíce jedna instance, a poskytne k ní globální přístupový bod.

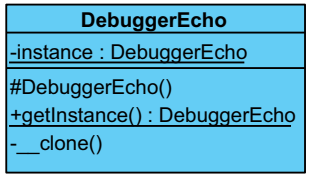
Pro implementaci tohoto návrhového vzoru jsou vždy nutné následující čtyři kroky:

1. Definovat statický atribut, který obsahuje instanci objektu třídy.
2. Implementovat statickou metodu, která vrací objekt z bodu 1 a v případě, že neexistuje, jej vytvoří.
3. Zabránit vytvoření nové instance pomocí operátoru `new` tak, že bude konstruktor deklarovaný jako `protected`.
4. Zajistit, aby objekt třídy nemohl být klonovaný, k čemuž stačí metodu `__clone()` deklarovat jako `private`.

Na obrázku 2.1 je zobrazený diagram jazyka UML pro obecný návrhový vzor Singleton a na obrázku 2.2 je jeho konkrétní implementace.



Obrázek 2.1: UML diagram návrhového vzoru Singleton



Obrázek 2.2: UML diagram konkrétní implementace

Shrnutí

Použití návrhového vzoru Singleton má pro vaši aplikaci následující dopady:

- Můžete přesně kontrolovat způsob, jakým se přistupuje ke třídě implementující tento návrhový vzor.
- Z dané třídy existuje vždy maximálně jedna instance. Druhou instanci třídy při použití tohoto návrhového vzoru není možné vytvořit. Existují však různé modifikace tohoto návrhového vzoru, u nichž je povolena i více než jedna instance.
- Návrhový vzor Singleton vám umožňuje zredukovat počet globálních proměnných nebo je ze zdrojového kódu úplně odstranit. Místo ukládání globálních instancí tříd v globálním oboru názvů můžete přistupovat k jejich instancím přes statické metody, čímž udržujete globální obor názvů volný.

Návrhový vzor Singleton je jeden z nejjednodušších návrhových vzorů, což může být i důvodem, proč je využívaný i v situacích, kdy to není nutné. Před jeho nasazením byste se proto měli zamyslet, zda skutečně povolíte jen jednu instanci třídy, nebo zda je užitečné současné využití více instancí a ve skutečnosti potřebujete jen centrální přístupový bod k objektu. V posledním případě byste k tomu mohli využít i návrhový vzor *Registry* (Registr).

Další využití

Kromě objektu typu `Debugger` existují i další možnosti, kde lze využít návrhový vzor Singleton. Často se například využívá při poskytování centrálního přístupového bodu ke konfiguraci aplikace. Při existenci jen jednoho objektu, který se stará o ukládání konfigurace, získáte hned dvě výhody:

- Pokud se změní nějaká konfigurační hodnota komponenty, platí tato změna okamžitě pro všechny komponenty.
- Konfigurační soubory stačí načíst jen jednou, což se kladně projeví na spotřebě dostupných zdrojů.

Stejně často se návrhový vzor Singleton používá, pokud aplikace přistupuje k externím zdrojům, jako jsou například databáze. Použitím jednoho centrálního objektu, který se stará o přístup k databázi, stačí otevřít jen jedno připojení, které si pak jednotlivé komponenty sdílejí.

Variace návrhového vzoru Singleton

Možná jste si říkali, jak se dá použít návrhový Singleton pro objekty, které mají být z vnějšku parametrizované. Chcete-li například znovu změnit způsob ladění aplikace na protokolování do souboru, avšak nezapisovat všechna hlášení do jednoho souboru, lze použít pro různé komponenty různé soubory.

Objekt typu `Debugger` pro protokolování do souboru by mohl vypadat následovně:

```
namespace cz\k1886\debuggers;

class DebuggerLog implements Debugger {

    protected $logfile = null;

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

    public function debug($message) {
        error_log($message . "\n", 3, $this->logfile);
    }
}
```

Při vytváření instance třídy `DebuggerLog` se konstruktoru předá název souboru, do něhož se mají předaná hlášení zapisovat. Díky tomu lze zapisovat hlášení do různých souborů.

```
use cz\k1886\debuggers\DebuggerLog;

$debugger1 = new DebuggerLog('./debugger1.log');
$debugger1->debug('Lorem ipsum dolor sit amet.');

$debugger2 = new DebuggerLog('./debugger2.log');
$debugger2->debug('Proin fringilla bibendum sagittis.');
```

Jenže v tomto případě již nemůžete využít vzor Singleton, protože chcete povolit více než jednu instanci. To ale chcete umožnit jen v případě, kdy různé instance zapisují do různých souborů. V případě zápisu do stejného protokolovacího souboru by se měla použít stejná instance. Jednoduchou změnou implementace návrhového vzoru Singleton je i toto možné. Místo jedné globální instance musíte do statického atributu uložit instanci třídy podle použitého souboru. Tento atribut tedy vyměníte za pole:

```
namespace cz\k1886\debuggers;

class DebuggerLog implements Debugger {

    protected $logfile = null;
    private static $instances = array();

    public static function getInstance($logfile) {
        if (!isset(self::$instances[$logfile])) {
            self::$instances[$logfile] = new self($logfile);
        }
        return self::$instances[$logfile];
    }

    protected function __construct($logfile) {
        $this->logfile = $logfile;
    }

    private function __clone() {}

    public function debug($message) {
        error_log($message . "\n", 3, $this->logfile);
    }
}
```

Nyní už stačí jen předat název protokolovacího souboru metodě `getInstance()`. Následující příklad vytvoří dvě instance třídy `DebuggerLog`, přestože se metoda `getInstance()` bude volat třikrát:

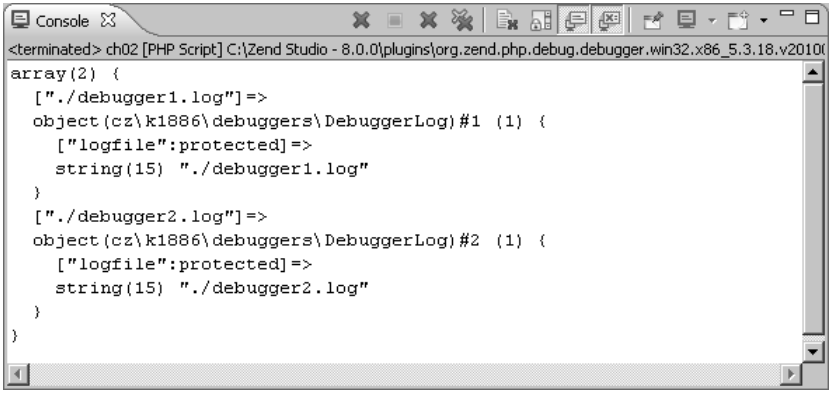
```
use cz\k1886\debuggers\DebuggerLog;

$debugger1 = DebuggerLog::getInstance('./debugger1.log');
$debugger1->debug('Lorem ipsum dolor sit amet.');

$debugger2 = DebuggerLog::getInstance('./debugger2.log');
$debugger2->debug('Proin fringilla bibendum sagittis.');

$debugger3 = DebuggerLog::getInstance('./debugger1.log');
$debugger3->debug('Mauris vitae augue dolor.');
```

První a poslední volání vrátily stejný objekt, o čemž se můžete přesvědčit i na obrázku 2.3.



Obrázek 2.3: Pole instancí upravené implementace návrhového vzoru Singleton

Sice jste zde neimplementovali vzor Singleton v klasické podobě, nicméně s touto variací se budete v praxi střetávat velmi často.