2

OOP prostřednictvím návrhových vzorů

Jasně největší a neproklamovanější změnou v PHP5 je kompletní přepracování objektového modelu a značně vylepšená podpora standardních objektově orientovaných (OO) metodologií a technik. Tato kniha není zaměřena na techniku objektově orientovaného programování (OOP) a ani není o návrhových vzorech. Těmto tématům se věnuje mnoho jiných knih (některé jsou uvedeny na konci této kapitoly). Místo toho se v této kapitole budeme věnovat přehledu rysů OOP v PHP5 a některým obecným návrhovým vzorům.

Já mám spíše agnostický pohled na využití OOP v PHP. Pro mnohé problémy je použití metod OOP jako použití kladiva k zabití mouchy. Úroveň abstrakce, která je zde použita, se mi jeví zbytečná pro zvládnutí jednoduchých úkolů. Metody OOP jsou vhodné pro řešení komplexnějších systémů. Již jsem pracoval na několika rozsáhlých projektech, které opravdu těžily z toho, že byly navrženy modulárně s využitím technik OOP.

V této kapitole je popsán přehled pokročilých rysů OOP, které jsou nyní podporovány v PHP. Některé zde uvedené příklady budou použity i v dalších částech knihy a budou – doufejme – sloužit jako ukázky toho, jak lze prostřednictvím OOP úspěšně řešit určité problémy.

Objektově orientované programování znamená zásadní změnu od procedurálního programování, které je tradiční technikou programátorů PHP. U procedurálního programování máte data (uložená v proměnných), která předáváte funkcím a ty pak provádí s těmito daty různé operace, mohou je modifikovat, či vytvářet z nich data nová. Procedurální programování je tradičně seznam instrukcí, které se vykonávají v daném pořadí pomocí výrazů řízení toku, funkcí apod. Zde následuje příklad takovéhoto procedurálního kódu:

```
<?php
function hello($name)
```

```
{
  return "Hello $name!\n";
}
function goodbye($name)
{
  return "Goodbye $name!\n";
}
function age($birthday) {
  $ts = strtotime($birthday);
  if($ts === -1) {
  return "Unknown";
  }
  else {
  $diff = time() - $ts;
  return floor($diff/(24*60*60*365));
  }
}
$name = "george";
$bday = "10 Oct 1973";
print hello($name);
print "You are ".age($bday)." years old.\n";
print goodbye($name);
? >
```

Objektově orientované programování

Úvodem je důležité poznamenat, že v procedurálním programování jsou funkce a data od sebe navzájem odděleny. V OOP jsou data a funkce, které s těmito daty operují, navzájem svázány do objektů. Objekty obsahují jak data (nazývané attributy nebo vlastnosti), tak i funkce, které těmito daty operují (nazývané metody).

Objekt je definován třídou, jejíž je instancí. Třída definuje atributy, které objekt obsahuje a metody, které může použít. Objekt vytvoříte vytvořením instance třídy. Vytvořením instance se vytvoří nový objekt, incializují se všechny jeho atributy a je volán jeho konstruktor, což je funkce, která provádí všechny operace prvotního nastavení. Konstruktor třídy bývá v PHP 5 nazýván `__construct()`, což umožňuje jeho jednoduchou identifikaci. Následující příklad definuje jednoduchou třídu `User`, vytváří její instanci a volá její dvě metody:

```
<?php
class User {
  public $name;
  public $birthday;
  public function __construct($name, $birthday)
```

```
  {
    $this->name = $name;
    $this->birthday = $birthday;
  }
  public function hello()
  {
    return "Hello $this->name!\n";
  }
  public function goodbye()
  {
    return "Goodbye $this->name!\n";
  }
  public function age() {
    $ts = strtotime($this->birthday);
    if($ts === -1) {
      return "Unknown";
    }
    else {
      $diff = time() - $ts;
      return floor($diff/(24*60*60*365)) ;
    }
  }
}
$user = new User('george', '10 Oct 1973');
print $user->hello();
print "You are ".$user->age()." years old.\n";
print $user->goodbye();
?>
```

Spuštění tohoto kódu vyvolá následující:

```
Hello george!
You are 29 years old.
Goodbye george!
```

Konstruktor je v tomto příkladu velmi jednoduchý – pouze inicializuje dva atributy, jméno (`name`) a datum narození (`birthday`). Metody jsou rovněž triviální. Poznamenejme, že parametr `$this` je automaticky vytvořen uvnitř metod třídy a reprezentuje samotný objekt `User`. Pro přístup k vlastnostem nebo metodám objektu použijte notaci `->`.

Na první pohled zde není vidět mnoho rozdílů mezi asociačním polem a kolekcí funkcí, které ji reprezentují. Nicméně, existují některé další důležité vlastnosti, které si popíšeme v následujících částech:

- **Dědičnost** – Dědičnost je schopnost odvozovat novou třídu ze stávající a zdědit nebo přepsat její atributy a metody.

- **Zapouzdření** – Zapouzdření je schopnost skrýt data před uživatelem třídy.
- **Speciální metody** – Jak uvidíte dále, třídy umožňují, v okamžiku, kdy je nový objekt vytvářen, pomocí konstruktoru vykonat prvotní inicializaci (např. nastavení atributů). Dále jsou rovněž generovány další události, které nastávají při určitých příležitostech, jakými jsou např. kopírování, rušení objektu apod.
- **Polymorfismus** – Když dvě třídy implementují stejné externí metody, je možné ve funkcích použít zaměnitelnost. Protože k úplnému pochopení polymorfismu je třeba větší znalostní báze než v této chvíli máme, popíšeme si ho později v této kapitole v části věnované polymorfismu.

Dědičnost

Dědičnost můžete využít, pokud chcete vytvořit novou třídu, která má mít podobné vlastnosti nebo chování jako již existující třída. K zajištění dědičnosti podporuje PHP schopnost třídy rozšířit se o již existující třídu. Když třídu rozšíříte, nová třída zdědí všechny vlastnosti a metody svého rodiče (se dvěma výjimkami, které si popíšeme později v této kapitole). Dále pak můžeme buď přidat nové vlastnosti a metody nebo přepsat již existující. Dědičný vztah je definován klíčovým slovem `extends`. Nyní si rozšíříme třídu `User`, nová třída bude reprezentovat uživatele s právy administrátora. Třídu rozšíříme o vybrání uživatelského hesla ze souboru `NDBM` a přidáme funkci umožňující porovnání tohoto hesla se zadaným heslem:

```
class AdminUser extends User{
  public $password;
  public function __construct($name, $birthday)
  {
    parent::__construct($name, $birthday);
    $db = dba_popen("/data/etc/auth.pw", "r", "ndbm");
    $this->password = dba_fetch($db, $name);
    dba_close($db);
  }
  public function authenticate($suppliedPassword)
  {
    if($this->password === $suppliedPassword) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Třebaže je to úplně krátké, třída `AdminUser` automaticky zdědí všechny metody třídy `User`, takže můžete volat metody `hello()`, `goodbye()` a `age()`. Poznamenejme, že musíte ručně volat konstruktor

rodičovské třídy: `parent::__constructor()`; PHP5 automaticky nevolá konstruktor rodiče. Parent je klíčové slovo, které vyjadřuje, že se jedná o třídu rodiče.

Zapouzdření

Uživatelé přecházející z procedurálního jazyka nebo PHP 4 si mohou myslet, že všechno kolem je veřejné. PHP verze 5 umožňuje rozdělit viditelnost dat atributů a metody na veřejnou, chráněnou a privátní. Tyto typy jsou normálně označovány zkratkou PPP (public, protected, private) a mají standardní sémantiku:

- **Public** – Veřejná proměnná nebo metoda, která může být přímo přístupná jakémukoli uživateli třídy.
- **Protected** – Chráněná proměnná nebo metoda, která nemůže být přístupná uživatelům třídy, ale může být přístupná uvnitř podtřídy, která tuto třídu dědí.
- **Private** – Privátní proměnná nebo metoda, která může být přístupná pouze uvnitř třídy, ve které je definována. To znamená, že privátní proměnná nebo metoda nemůže být volána z potomka dané třídy.

Zapouzdření umožňuje definovat veřejné rozhraní, které určuje způsob, jakým může uživatel s třídou komunikovat. Metody, které nejsou veřejné můžete předělat nebo změnit bez toho, abyste se znepokojovali tím, že poškodíte kód, který jste zdědili z nějaké třídy. Beztrestně rovněž můžete přepsat privátní metody. Přepsání chráněných metod vyžaduje více péče, abyste se vyvarovali přerušení vazby podtříd.

Zapouzdření není v PHP nezbytné (pokud není použito, metody a proměnné jsou považovány za veřejné), ale pokud je to jenom trochu možné, je dobré zapouzdření využít. V jednoprogramátorském prostředí a zvláště pak při týmové práci je velkou chybou vyhýbat se veřejným rozhraním objektu a provádět přímý přístup k proměnným a k metodám. Takové přístupy rychle vedou k neudržitelnosti kódu, protože místo jednoduchého veřejného rozhraní se dostanete do stavu, kdy jsou všechny metody třídy neschopné přepracování bez toho, aniž byste nezpůsobili chyby v třídě, která tyto metody používá. Použití PPP vás zavazuje používat tuto dohodu a zajišťuje, že v externím kódu jsou použity pouze veřejné metody bez ohledu na lákadlo přímého přístupu.

Statické (třídní) atributy a metody

V PHP mohou být navíc metody a vlastnosti deklarovány jako statické. Statická metoda je vázána přímo na třídu, ne k instanci této třídy (objektu). Statické metody jsou volány pomocí syntaxe `ClassName::method()`. Použití proměnné `$this` uvnitř statické metody není možné.

Statická vlastnost je proměnná třídy, která je spojena se třídou, ne s instancí třídy. Tzn., že když je vlastnost změněna, projeví se tato změna ve všech instancích dané třídy. Statické proměnné jsou deklarovány klíčovým slovem `static` a jsou přístupné prostřednictvím syntaxe `ClassName::$property`. Následující příklad ukazuje, jak taková statická proměnná funguje:

```
class TestClass {
  public static $counter;
```

```
}
$counter = TestClass::$counter;
```

Pokud chcete ke statické proměnné přistupovat uvnitř třídy, můžete rovněž použít kouzelná klíčová slova `self` a `parent`, která rozlišují, zdali se jedná o danou třídu nebo jejího rodiče. Použití klíčových slov `self` a `parent` umožňuje vyhnout se explicitní referenci na název třídy. Zde je jednoduchý příklad, který využívá statickou vlastnosti k přiřazení unikátního celočíselného `ID` pro každou instanci třídy:

```
class TestClass {
  public static $counter = 0;
  public $id;
  public function __construct()
  {
    $this->id = self::$counter++;
  }
}
```

Speciální metody

Třídy v PHP mají vyhrazeny určité názvy metod jako speciální volání pro zpracování určitých událostí. Už jste se setkali s metodou `__construct()`, která je automaticky volána, když je vytvářena instance objektu. Třídy využívají dalších pět speciálních metod: `__get()`, `__set()` a `__call()` ovlivňující způsob, jakým jsou vlastnosti a metody objektu volány a jsou popsány později v této kapitole. Zbylé dvě metody jsou `__destruct()` a `__clone()`.

`__destruct()` je metoda volaná při rušení objektu. Destruktory jsou vhodné pro uvolnění zdrojů (např. napojení na soubor nebo databázi), které třída vytvořila. V PHP jsou proměnné referenčně počítány. Když je počítadlo referencí proměnné sníženo na nulu, je proměnná automaticky odstraněna ze systému. Pokud je proměnnou objekt, je volána jeho metoda `__destruct()`.

Následující jednoduché zabalení funkcí pracujících v PHP se souborem ukazuje využití destruktoru:

```
class IO {
  public $fh = false;
  public function __construct($filename, $flags)
  {
    $this->fh = fopen($filename, $flags);
  }
  public function __destruct()
  {
    if($this->fh) {
      fclose($this->fh);
    }
  }
  public function read($length)
```

```
  {
    if($this->fh) {
      return fread($this->fh, $length);
    }
  }
  /* ... */
}
```

V mnoha případech je vytváření destruktoru zbytečné, protože PHP na konci požadavku uvolňuje všechny použité zdroje. Pro dlouhotrvající skripty nebo skripty, které otevírají velký počet souborů, je však uvolňování zdrojů důležité.

V PHP 4 jsou objekty předávány svou hodnotou. Tzn. pokud v PHP 4 napíšete:

```
$obj = new TestClass;
$copy = $obj;
```

vytvoříte aktuálně tři kopie dané třídy: Jedna instance objektu je vytvořena při deklaraci, druhá během přiřazení návratové hodnoty z konstruktoru do `$obj` a třetí, když přiřadíte `$copy` proměnné `$obj`. Tato sémantika je úplně odlišná od sémantiky ve všech ostatních objektově orientovaných jazycích a v PHP 5 je již od ní upuštěno.

Když v PHP 5 vytváříte objekt, vracíte handle na tento objekt, který je obdobný v pojetí s referencí s C++. Když spustíte předchozí kód v PHP 5, vytvoříte pouze jednu instanci daného objektu; nevytvoříte žádnou jeho další kopii.

Pro vytvoření aktuální kopie objektu musíte v PHP 5 použít metodu `__clone()`. Takže předcházející příklad, ve kterém vytváříme v `$copy` aktuální kopii objektu `$obj` (a nikoliv jenom další referenci na stejný objekt), bude vypadat následovně:

```
$obj = new TestClass;
$copy = $obj->__clone();
```

Pro některé třídy není možné použít předpřipravenou metodu `__clone()`, ale PHP dovoluje její přepsání. Uvnitř metody `__clone()` máte proměnnou `$this`, která je novým objektem se všemi původními vlastnostmi objektu, které byly zkopírovány. Např. ve třídě `TestClass`, definované již dříve v této kapitole, byste použitím defaultní metody `__clone()`, zkopírovali i její vlastní proměnnou `id`. Proto musíte upravit tuto třídu následovně:

```
class TestClass {
  public static $counter = 0;
  public $id;
  public $other;
  public function __construct()
  {
    $this->id = self::$counter++;
  }
```

```
  public function __clone()
  {
    $this->id = self::$counter++;
  }
}
```

Stručný úvod do návrhových vzorů

Určitě jste již o návrhových vzorech někdy slyšeli, ale nevěděli jste co tento pojem vlastně znamená. Návrhové vzory jsou zevšeobecněná řešení tříd problémů, se kterými se programátoři setkávají nejčastěji.

Když se programování věnujete už nějaký delší dobu, určitě máte potřebu přizpůsobit si danou knihovnu tak, aby byla přístupná prostřednictvím různých API. Nejste sami. Toto je obecný problém. Je sice pravda, že neexistuje nějaké obecné řešení, které řeší všechny problémy – nicméně, lidé tento typ problému znají a řeší jej stále znovu a znovu. Základní myšlenkou návrhových vzorů je, že problémy a jejich odpovídající řešení mají sklon k následování pomocí opakujících se šablon.

Návrhové vzory však bývají někdy nedoceňovány. Před lety jsem odmítl využít návrhové vzory bez toho, aniž bych o nich reálně uvažoval. Moje problémy byly natolik specifické a složité, že jsem si myslel, že se na ně nehodí žádná šablona. To bylo ode mně opravdu krátkozraké.

Návrhové vzory poskytují slovník pro identifikaci a klasifikaci problémů. V Egyptské mytologii měly božstva a další entity tajná jména a až když jste tato jména objevili, mohli jste ovládat jejich "božskou sílu". Problémy návrhu jsou v přírodě velmi podobné. Když můžete rozeznat správné řešení problému v přírodě a ztotožnit ho se známou množinu obdobných (řešených) problémů, dostanete se na správnou cestu k jejich řešení.

Tvrdit, že jedna kapitola o návrhových vzorech vám bude stačit, je směšná. V následujících částech si popíšeme několik vzorů, hlavně jako prostředek pro popsání některých pokročilých objektově orientovaných technik, které jsou v PHP k dispozici.

Adaptér vzoru

Adaptér vzoru poskytuje přístup k objektu prostřednictvím specifického rozhraní. V čistě objektově orientovaném jazyku umožňuje adaptér vzoru provázání alternativního API s objektem; ale v PHP se často spíše setkáme s tím, jak takový adaptér vzoru poskytuje alternativní rozhraní k sadě procedur.

Poskytnutí rozhraní s objektem prostřednictvím specifického API může být užitečné z těchto dvou hlavních důvodů:

- Když různé třídy poskytují podobné služby a implementují stejné API, můžete se mezi nimi v průběhu programu přepínat. Toto je známo jako polymorfismus. Toto slovo pochází z latiny: Poly znamená "mnoho" a morph znamená "tvar" čili "mnohotvárnost".

- Může být obtížné změnit předdefinovaný systém pro využívání sady objektů. Když do projektu začleňujete třídu třetí strany, která není kompatibilní s použitým API systému, je často nejjednodušším řešením použít adaptér poskytující přístupu k danému API.

Nejběžnější způsob použití adaptérů v PHP není při vytváření alternativního rozhraní pro spojení jedné třídy s druhou (protože to je v komerčním kódu PHP limitováno placenou částkou a v otevřeném kódu můžete takovéto rozhraní změnit přímo). PHP má svůj původ v procedurálním jazyku, a proto jsou mnohé předpřipravené funkce PHP jsou ze své podstaty procedurální. Když je potřeba, aby byly funkce spouštěny postupně, v daném pořadí (např. když pracujete s databází, musíte použít postupně funkce `mysql_pconnect()`, `mysql_select_db()`, `mysql_query()` a `mysql_fetch()`), jsou pro uchování dat pro napojení na databázi použity určité zdroje a vy je musíte předávat do všech těchto funkcí. Zabalením celého takového procesu do třídy se můžete zbavit stále se opakující činnosti a vyhnete se generování následných chyb.

Řekněme, že bychom chtěli vytvořit objektové rozhraní pro dvě základní zdrojové funkce MySQL: pro napojení se na databázi a pro získání výsledných dat. Naším cílem není napsat úplnou abstrakci, ale jednoduše vytvořit dostatečně zabalený kód, který bude zpřístupňovat všechny rozšiřující funkce MySQL objektově orientovaným způsobem a přidá navíc několik dalších praktických věcí.

Zde je první pokus, jak zabalit třídu:

```
class DB_Mysql {
  protected $user;
  protected $pass;
  protected $dbhost;
  protected $dbname;
  protected $dbh; // Database connection handle
  public function __construct($user, $pass, $dbhost, $dbname) {
    $this->user = $user;
    $this->pass = $pass;
    $this->dbhost = $dbhost;
    $this->dbname = $dbname;
  }
  protected function connect() {
    $this->dbh = mysql_pconnect($this->dbhost, $this->user, $this->pass);
    if(!is_resource($this->dbh)) {
      throw new Exception;
    }
    if(!mysql_select_db($this->dbname, $this->dbh)) {
      throw new Exception;
    }
  }
  public function execute($query) {
    if(!$this->dbh) {
```

```
    $this->connect();
  }
  $ret = mysql_query($query, $this->dbh);
    if(!$ret) {
      throw new Exception;
    }
    else if(!is_resource($ret)) {
      return TRUE;
    } else {
      $stmt = new DB_MysqlStatement($this->dbh, $query);
      $stmt->result = $ret;
      return $stmt;
    }
  }
}
```

Pomocí tohoto rozhraní můžete vytvořit nový objekt `DB_Mysql`, při inicializaci mu předat přístupové údaje k databázi MySQL a tím se na ní napojit (uživatelské jméno, heslo, stanice a název databáze):

```
$dbh = new DB_Mysql("testuser", "testpass", "localhost", "testdb");
$query = "SELECT * FROM users WHERE name = '".mysql_escape_string($name)."'";
$stmt = $dbh->execute($query);
Tento kód vrací objekt DB_MysqlStatement, což je objekt, který zabaluje výstup
z MySQL databáze:
class DB_MysqlStatement {
  protected $result;
  public $query;
  protected $dbh;
  public function __construct($dbh, $query) {
    $this->query = $query;
    $this->dbh = $dbh;
    if(!is_resource($dbh)) {
      throw new Exception("Not a valid database connection");
    }
  }
  public function fetch_row() {
    if(!$this->result) {
      throw new Exception("Query not executed");
    }
    return mysql_fetch_row($this->result);
  }
  public function fetch_assoc() {
    return mysql_fetch_assoc($this->result);
```

```
  }
  public function fetchall_assoc() {
    $retval = array();
    while($row = $this->fetch_assoc()) {
      $retval[] = $row;
    }
    return $retval;
  }
}
```

K vypsání jednotlivých řádků výsledku dotazu musíte místo použití funkce `mysql_fetch_assoc()`, použít následující konstrukci:

```
while($row = $stmt->fetch_assoc()) {
  // process row
}
```

Následuje několik poznámek k uvedené implementaci:

- Vyhýbá se manuálnímu volání funkcí `connect()` a `mysql_select_db()`.
- Při chybě vyvolá výjimku. Výjimky jsou v PHP 5 novým rysem. Zde o nich ještě nebudeme mluvit, ale podrobně si je popíšeme v druhé polovině kapitoly 3 popisující zpracování chyb.
- Tento kód není moc praktický. Neexistuje způsob, jak jednoduše znovu použít nebo upravit dotaz – pro jiný dotaz musíte znovu všechna data kódu přepsat.

Na základě těchto tří problémů můžete toto rozhraní rozšířit tak, aby umožňovalo automaticky zabalit všechna data, která mu pošlete. Nejjednodušší cestou, jak to udělat, je provést emulaci všech připravených dotazů. Když spouštíte znovu dotaz na databázi, hrubé SQL, které zde vytváříte, musí být nejprve přeloženo do formátu, kterému databáze vnitřně rozumí. Tento krok vyžaduje zvýšené nároky na výkon a mnohé databázové systémy tento problém řeší cachováním. Uživatel si může dotaz předpřipravit, což způsobí, že databáze si provede rozbor tohoto dotazu a vrací jakýsi předkompilovaný zdroj, který je pak použit k vlastní reprezentaci dotazu. Takováto vlastnost je často spojována s pojmem vázání SQL. Vázání SQL umožňuje provést rozbor dotazu v souladu s umístěním dat, se kterými budou vaše proměnné následně svázány. Následně pak můžete svázat parametry s předkompilovanou verzí dotazu ještě před jeho spuštěním. V mnoha databázových systémech (zejména v Oracle) je použití vázání dat SQL významnou výkonnostní výhodou.

Verze MySQL před verzí 4.1 neposkytuje samostatné uživatelské rozhraní k přípravě dotazů před jejich spuštěním, ani neumožňuje vázání SQL. To pro nás znamená posílat všechna měnící se data ke zpracování samostatně, zajistit vhodné místo k uložení proměnných a uchovávat je až do doby, než budou vloženy do dotazu. Rozhraní nové verze MySQL 4.1 je vybudováno na základě rozšíření `mysqli` od Georga Richtera a poskytuje nové funkčnosti.

Abychom těchto nových vlastností mohli využít, musíme rozšířit třídu `DB_Mysql` o metody `prepare` a do třídy `DB_MysqlStatement` vložit metody `bind` a `execute`:

```
class DB_Mysql {
/* ... */
  public function prepare($query) {
    if(!$this->dbh) {
      $this->connect();
    }
    return new DB_MysqlStatement($this->dbh, $query);
  }
}
class DB_MysqlStatement {
  public $result;
  public $binds;
  public $query;
  public $dbh;
  /* ... */
  public function execute() {
    $binds = func_get_args();
    foreach($binds as $index => $name) {
      $this->binds[$index + 1] = $name;
    }
    $cnt = count($binds);
    $query = $this->query;
    foreach ($this->binds as $ph => $pv) {
      $query = str_replace(":$ph", "'".mysql_escape_string($pv)."'", $query);
    }
    $this->result = mysql_query($query, $this->dbh);
    if(!$this->result) {
      throw new MysqlException;
    }
    return $this;
  }
  /* ... */
}
```

Metoda `prepare()` v tomto příkladu neprovádí nic, jen jednoduše vytváří nový objekt `DB_Mysql-Statement`, který specifikuje dotaz. Skutečná práce je prováděná třídou `DB_MysqlStatement`. Pokud nemáme vázané parametry, můžete tyto objekty volat takto:

```
$dbh = new DB_Mysql("testuser", "testpass", "localhost", "testdb");
$stmt = $dbh->prepare("SELECT *
                       FROM users
                       WHERE name = '".mysql_escape_string($name)."'");
$stmt->execute();
```

Skutečnou výhodou použití zabaleného objektu oproti použití normálního procedurálního volání je, že k dotazu můžete navázat parametry. To provedete tak, že do textu dotazu vložíte výraz začínající dvojtečkou (`:`), který naváže daná data dotazu až v době provádění kódu:

```
$dbh = new DB_Mysql("testuser", "testpass", "localhost", "testdb");
$stmt = $dbh->prepare("SELECT * FROM users WHERE name = :1");
$stmt->execute($name);
```

Výraz `:1` v dotazu označuje, že se jedná o umístění první svázané proměnné. Když zavoláte metodu `execute()` objektu `$stmt`, tato metoda tento výraz rozpozná a přiřadí mu hodnotu prvního předaného argumentu (`$name`) a doplní hodnotu této proměnné za výraz `:1` v dotazu.

Dokonce, i když toto navázané rozhraní nemá normální výkonnostní výhodu vázaného rozhraní, poskytuje vhodný způsob, jak se jednoduše vyhnout přepisování všech vstupů v dotazu.

Šablona vzoru

Šablona vzoru popisuje třídu, jež modifikuje logiku podtřídy, čímž ji rozšiřuje.

Šablonu vzoru můžete např. využít ke skrytí všech parametrů specifikujících napojení na databázi, které jsme používali v předchozích třídách. Při použití tříd z předchozích částí musíte specifikovat konstanty parametrů pro připojení:

```
<?php
require_once 'DB.inc';
define('DB_MYSQL_PROD_USER', 'test');
define('DB_MYSQL_PROD_PASS', 'test');
define('DB_MYSQL_PROD_DBHOST', 'localhost');
define('DB_MYSQL_PROD_DBNAME', 'test');
$dbh = new DB::Mysql(DB_MYSQL_PROD_USER, DB_MYSQL_PROD_PASS,
                     DB_MYSQL_PROD_DBHOST, DB_MYSQL_PROD_DBNAME);
$stmt = $dbh->execute("SELECT now()");
print_r($stmt->fetch_row());
?>
```

Abychom se vyhnuli specifikaci konstant parametrů připojení, můžeme definovat podtřídu `DB_Mysql` a v ní natvrdo zadat parametry pro připojení k databázi `test`:

```
class DB_Mysql_Test extends DB_Mysql {
  protected $user = "testuser";
  protected $pass = "testpass";
  protected $dbhost = "localhost";
  protected $dbname = "test";
  public function __construct() { }
}
```

Obdobně můžeme to samé udělat pro ostrou databázi:

```
class DB_Mysql_Prod extends DB_Mysql {
  protected $user = "produser";
  protected $pass = "prodpass";
  protected $dbhost = "prod.db.example.com";
  protected $dbname = "prod";
  public function __construct() { }
}
```

Polymorfismus

Objekty určené pro práci s databází, popsané v této kapitole, jsou dobře obecně použitelné. Ale ve skutečnosti, když se podíváte na jiné databázové rozšíření vytvořené v PHP, najdete v nich vždy znovu a znovu stejnou základní funkčnost – napojení se na databázi, příprava dotazů, spuštění dotazů a zpracování výsledků. Když budete chtít, můžete napsat obdobné třídy `DB_Pgsql` nebo `DB_Oracle`, které zabalí knihovny pro PostgreSQL, respektive Oracle a v nich můžete použít úplně stejné základní metody.

Je ale vždy důležité použít pro metody, které provádějí stejný druh operace vždy totožné názvy metod, a to i v případě, že tyto metody jsou vnitřně nekompatibilní. To následně umožňuje polymorfismus, což je schopnost transparentně zaměnit jeden objekt za jiný, jejichž přístupové API jsou shodné.

Prakticky: polymorfismus znamená, že můžete například napsat funkci:

```
function show_entry($entry_id, $dbh)
{
  $query = "SELECT * FROM Entries WHERE entry_id = :1";
  $stmt = $dbh->prepare($query)->execute($entry_id);
  $entry = $stmt->fetch_row();
  // display entry
}
```

Tato funkce nepracuje pouze, když je proměnná `$dbh` objekt typu `DB_Mysql`, ale bude pracovat správně s jakýmkoli typem objektu `$dbh`, který má implementovanou metodu `prepare`() a tato metoda vrací objekt, který má implementovány metody `execute()` a `fetch_assoc()`.

Abychom daný databázový objekt předali do všech volaných funkcí, můžeme využít princip delegace. Delegace je objektově orientovaný model, kdy objekt má jako atribut jiný objekt, který používá k provádění daných úloh.

Knihovny zapouzdřující databázi jsou ideálním příkladem objektu, který využívá právě delegace. V normálních aplikacích mnohé třídy potřebují provádět databázové operace. Při tvorbě takovýchto tříd máte dvě možnosti:

- Všechna databázová volání můžete implementovat nativně. To je směšné. Toto řešení je jen pro nouzovou práci, kterou pak musíte dělat stále znovu a znovu a takovéto zabalení databáze je vlastně zbytečné.

- Můžete použít zabalené API databáze, ale vlastní objekty inicializovat mimo. Zde je příklad, který takovouto možnost využívá:

```
class Weblog {
  public function show_entry($entry_id)
  {
    $query = "SELECT * FROM Entries WHERE entry_id = :1";
    $dbh = new Mysql_Weblog();
    $stmt = $dbh->prepare($query)->execute($entry_id);
    $entry = $stmt->fetch_row();
    // display entry
  }
}
```

  Jak můžete vidět, inicializace objektu napojení na databázi mimo, se jeví jako dobrý nápad. Můžete zde použít zabalenou knihovnu, což je dobré. Problém ale nastane, když chcete změnit databázi, která tuto třídu používá, to pak musíte vytvořit a změnit všechny funkce, které provádějí operace s databází.

- Použijete delegaci, a to tím, že třída `Weblog` obsahuje atribut, kterým je objekt zabalující databázi. Když je tato třída vytvářena, je vytvořen objekt zabalující databázi, který pak můžete použít pro všechny vstupně/výstupní operace. Zde je přepsána původní třída `Weblog`, která používá tuto techniku:

```
class Weblog {
  protected $dbh;
  public function setDB($dbh)
  {
    $this->dbh = $dbh;
  }
  public function show_entry($entry_id)
  {
    $query = "SELECT * FROM Entries WHERE entry_id = :1";
    $stmt = $this->dbh->prepare($query)->execute($entry_id);
    $entry = $stmt->fetch_row();
    // display entry
  }
}
```

Nyní můžete nastavit databázi do objektu následovně:

```
$blog = new Weblog;
$dbh = new Mysql_Weblog;
$blog->setDB($dbh);
```

Samozřejmě můžete místo nastavení delegace databáze použít šablonu vzoru:

```
class Weblog_Std extends Weblog {
  protected $dbh;
  public function __construct()
  {
    $this->dbh = new Mysql_Weblog;
  }
}
$blog = new Weblog_Std;
```

Delegace je výhodná vždy, když potřebujete provádět komplexní službu nebo službu, kterou je vhodné provádět uvnitř třídy. Dalším místem, kde se delegace běžně využívá, jsou třídy, které potřebují generovat výstup. Když je potřebné provádět výstup různými způsoby (např. v HTML, RSS [Rich Site Summary nebo také Really Simple Syndication] nebo prostém textu), vyplatí se vytvořit registr delegací umožňující generovat výstup, který chcete.

Rozhraní a kontrola typu

Klíčem k úspěšné delegaci objektů je zaručení toho, že všechny třídy, které mají být použity jsou polymorfické. Když v objektu `Weblog` nastavíte jako parametr `$dbh` třídu, která nemá implementovánu metodu `fetch_row()`, vyvoláte při běhu programu fatální chybu. Takováto chyba při běhu programu se detekuje velice těžce a proto bychom měli již dopředu zajistit, že všechny objekty mají implementovány všechny požadované funkce.

K předcházení takovýchto chyb zavádí PHP 5 koncept rozhraní. Rozhraní (interface) je vlastně kostra třídy. Definuje všechny metody třídy, ale neobsahuje žádný jejich kód – pouze vzor toho, jaké má daná metoda argumenty.

Zde je základní rozhraní s názvem `interface`, které specifikuje metody potřebné pro napojení na databázi:

```
interface DB_Connection {
  public function execute($query);
  public function prepare($query);
}
```

Vždy, když pak budete chtít děděním rozšířit danou třídu, použijete toto rozhraní a protože neobsahuje kód, jednoduše rozpoznáte všechny funkce, které jsou v něm definovány a které budete muset implementovat.

Např. pokud má třída `DB_Mysql` implementovat všechny funkce specifikované v rozhraní `DB_Connection`, musíte ji deklarovat následovně:

```
class DB_Mysql implements DB_Connection {
  /* definice třídy */
}
```