Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informatika

Návrhové vzory v PHP

BAKALÁŘSKÁ PRÁCE

Student : Jan Havel

Vedoucí : Ing. Rudolf Pecinovský CSc.

Oponent : Ing. Marko Ristič

2016

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 4. května 2016 . . . . . .

Jan Havel

Poděkování

Chci poděkovat především vedoucímu mé práce Ing. Rudolfu Pecinovskému, CSc. za velmi vstřícný přístup a mnoho užitečných rad k postupu zpracování této práce.

Abstrakt

Cílem práce je podat základní přehled nad oblastmi podpory objektově orientovaného programování a návrhových vzorů v kontextu programovacího jazyka PHP. Krom rešeršní části se shrnutím vývoje podpory objektového programování v jazyce PHP a shrnutím vývoje a významu návrhových vzorů byla v další části práce vytvořena příručka vybraných návrhových vzorů, které jsou v tomto jazyce využitelné. Vybrané návrhové vzory jsou v příručce doplněny o ukázkové kódy implementace na jednoduchých a výstižných situacích, které usnadní pochopení vzoru. V závěru práce se nachází část, která obsahuje doporučení pro úpravu vstupních kurzů programování v jazyce PHP na Vysoké škole Ekonomické.

Klíčová slova

Návrhové vzory, programovací jazyk PHP, příručka, doporučení, objektově orientované programování.

Abstract

The aim of this bachelor thesis is to give a basic overview of the areas of support object-oriented programming and design patterns in the context of the programming language PHP. This bachelor thesis includes recherché part with summary of the historic support of object-oriented programming in programming language PHP and a summary of the development and importance of design patterns. Moreover, a handbook with selected design patterns, which are useful in this programming language, was created in another part of the work. Selected design patterns in handbook are amend with sample codes on implementation of simple and cogent situations which facilitate the understanding of the pattern. In conclusion, there is a section that contains recommendations for modification of introductory courses in programming in programming language PHP at the University of Economics.

Keywords

Design patterns, programming language PHP, Handbook, recommendation, object-oriented programming.

Obsah

1 Úvod

Ačkoli je dnes objektové programování v jazyce PHP nejenom díky velkým společnostem jako je Facebook značně rozšířené, mnoho učebnic i různých internetových kurzů pro začátečníky se bohužel téma využití objektového programování natož téma návrhových vzorů v tomto jazyce věnuje jen velmi omezeně nebo vůbec. Dle mého názoru je vhodné seznamovat se s objektově orientovaným stylem programování co nejdříve, již v počátcích vlastního programování.

Pro začátečníka v oblasti tvorby webových stránek není obtížné vložit do svého HTML dokumentu pár řádků kódu PHP a docílit tak oživení svého webu o zajímavé dynamické funkce. Z vlastní zkušenosti vím, že tato skutečnost často vede k situaci, kdy se začínající programátor se složitějšími konstrukcemi seznamuje právě prostřednictvím jazyka PHP.

Bohužel situace na poli výukových zdrojů (například [5]) může způsobit, že mírně pokročilý programátor zná, popřípadě využívá, pouze základní principy objektového programování a techniky návrhových vzorů, kterým se v této práci věnuji, mohou zůstat neobjeveny.

Přiznám se, že sám jsem toho příkladem a s návrhovými vzory jsem se setkal až díky výuce programovacího jazyka Java.

Proto se hlavní myšlenka této bakalářské práce týká co nejjednoduššího předání znalostí ohledně využití návrhových vzorů. Především doufám, že tuto snahu ocení studenti, kteří se již v rámci jazyka PHP setkali s objektově orientovaným programováním, nicméně návrhové vzory dosud nepoznali, nebo je v jazyce PHP nepoužívají.

V práci nejprve komentuji vybrané informační zdroje, ze kterých jsem čerpal znalosti problematiky. Dále osvětluji základní pojmy a definice, které jsou důležité pro pochopení základních souvislostí problematiky. Osvětluji pojmy jako paradigma nebo objektově orientované programování (OOP). Také popisuji historický vývoj jazyka PHP, se zaměřením na jeho podporu OOP a historický vývoj samotných návrhových vzorů v OOP.

V druhé části práce předkládám příručku s vybranými návrhovými vzory, které jsou využitelné v jazyce PHP. Příručka nemá sloužit jako katalog návrhových vzorů a proto při popisu těchto vzorů kladu důraz především na jednoduché a přímočaré vysvětlení, které pomůže začátečníkům rychle proniknout do základů problematiky.

1.1 Cíle práce

Stěžejní části práce, jejichž vytvoření si kladu za cíl, jsem shrnul v následujících bodech:

- Osvětlit čtenáři základní pojmy a souvislosti paradigma objektově orientovaného programování a jeho využití v programovacím jazyku PHP.
- Představit obecnou funkci a význam návrhových vzorů v dříve osvětleném kontextu objektově orientovaného programování.
- Vytvořit příručku popisující konkrétní návrhové vzory využitelné v jazyku PHP, především jejich účel a způsob implementace (nasazení).
- V závěru práce sepsat vhodná doporučení pro úpravu vstupních kurzů programování v jazyce PHP na Vysoké škole Ekonomické.

1.2 Komu je práce určena

Práce je koncipována tak, aby byla užitečná především studentům začátečnických kurzů programování v jazyce PHP. Pro dostatečně komplexní pochopení problematiky návrhových vzorů je třeba mírně pokročilá znalost technik objektového programování, nicméně jak již bylo řečeno, v práci uvádím problematiku co nejjednodušeji, aby bylo snadné pochopit základní myšlenky.

Pro zkušené programátory může být popis některých vzorů nedostatečně rozsáhlý, a proto jim doporučuji po přečtení této práce využít podrobněji psané doporučené literatury, kterou je možné najít v části Rešerše informačních zdrojů.

2 Rešerše informačních zdrojů

Pro počáteční vstup do problematiky a prohloubení znalostí jsem využil především klasické publikace v knižní formě.

Nejspíše nejpřínosnějším informačním zdrojem pro tvorbu této bakalářské práce je kniha Návrhové vzory v PHP [1] slovenského programátora Mariana Böhmera. Tato publikace z roku 2012, ve které autor popisuje různorodé návrhové vzory společně s příklady využití a ukázkami vzorového kódu jednotlivých vzorů v jazyce PHP mi velmi pomohla při počátečním pochopení zpracovávané problematiky návrhových vzorů a byla mi užitečnou inspirací při tvorbě vlastních ukázkových kódů. Kromě získaných znalostí jsem se značně inspiroval způsobem řazení návrhových vzorů.

Neméně důležitým zdrojem je kniha vedoucího mé bakalářské práce Rudolfa Pecinovského: 33 vzorových postupů pro objektové programování [3]. V této knize je velice čtivým způsobem osvětleno mnoho návrhových vzorů, které rozebírám v této práci. Kniha není zaměřena přímo na využití návrhových vzorů v jazyce PHP, nicméně jsem často využil obecně platné znalosti. Také učebnice Java 8: Úvod do objektové architektury pro mírně pokročilé [4] od stejného autora mi byla velmi užitečnou, především pro inspiraci překladem terminologie a pro získávání znalostí o objektovém programování.

Při čerpání znalostí ohledně vývoje jazyka PHP a vývoje objektově orientovaného programování jsem využil vícero zdrojů, především elektronických knih dostupných z internetu. Stručně osvětlit základní pojmy objektově orientovaného programování (OOP) není snadný úkol. Značně mi ho ulehčila učebnice *PHP 5: začínáme programovat* [5], ve které autor Jiří Bráza důležité pojmy OOP vysvětluje v kontextu jazyka PHP, což pro mě bylo při získávání počátečních znalostí velmi užitečné. Bohužel návrhové vzory autor vůbec nezmiňuje, tudíž informace o možnostech jejich využití v jazyce PHP jsem musel hledat jinde. Především v zahraničních výukových publikacích. Velmi užitečným zdrojem informací o vztahu jazyka PHP s OOP a návrhovými vzory byla kniha Matta Zandstry: *PHP objects, patterns, and practice* [25] a také kniha Williama Sanderse: *Learning PHP design patterns* [27].

Pro pochopení provázanosti vývoje jazyka PHP a vývoje jeho funkcí směřujícím vstříc objektově orientovanému programování byl velmi přínosný také článek *History of PHP* [2], zveřejněný na webu php.net. Další důležité informace o vývoji objektově orientovaného programování jsem čerpal ze článku programátora Zeeva Suraskiho: *The Object-Oriented Evolution of PHP* [8], který napsal v roce 2002 pro web DevX.com.

Nemohu nezmínit také velmi přínosný informační zdroj, webovou stránku Sourcemaking.com [11]. Z tohoto webu jsem čerpal důležité informace o řadě rozebíraných návrhových vzorů a mimo to jsem se inspiroval ukázkovými zdrojovými kódy, které jsou na webu zpracovány v jazyce PHP.

3 PHP a jeho využití pro objektové programování

3.1 Paradigma objektově orientovaného programování

Objektově orientované programování je programovací paradigma, které představuje způsob uvažování nad psaním kódu. Tento způsob uvažování spočívá především ve vytváření několika samostatných objektů - částí aplikace, které jsou díky svému rozhraní opakovaně využitelné i v jiných aplikacích [9].

Toto paradigma je v současné době často upřednostňované oproti procedurálnímu, především kvůli snazšímu vývoji a údržbě aplikací. Nevýhodou může být náročnost na výpočetní výkon, který je potřeba pro správu objektů. Ze zkušenosti vlastní i ze zkušenosti mých vyučujících však vím, že prakticky žádná větší aplikace nebude ve stejně krátkém čase a za stejný objem prostředků procedurálně napsaná tak dokonale, aby se vyrovnala struktuře objektově orientovaného programování.

Pro většinu z Vás, kteří se již setkali s objektovým programováním, jsou základní pojmy jako třída, objekt nebo rozhraní dobře známé. Přesto ty nejdůležitější stručně shrnu. Snažíme se s jejich využitím přenést reálné objekty ze světa kolem nás do světa programování.

3.1.1 Základní pojmy OOP

Pro počáteční porozumění objektově orientovanému paradigma je třeba překonat překážku v podobě porozumění vazbě mezi třídami a objekty. Při programování s pomocí tohoto paradigma musíme na programy nahlížet jako na simulaci ať už reálného nebo smyšleného světa [4], který představují jednotlivé objekty a jejich vzájemné vazby.

Objekt

V objektově orientovaném programování jsou části modelovaného světa reprezentovány pojmenovanými daty – neboli objekty. Jako objekty jsou v programech označovány kolekce dat, které reprezentují skutečný objekt z modelovaného světa [4]. Tyto objekty představují obecně všechny části programu, které lze pojmenovat. Ve světě objektového programování jsou objektem i děje a vlastnosti [4]. Tudíž i třídy a metody jsou typy objektů se specifickou funkčností.

Každý objekt má podle [5] tyto části: Vnitřní stav (což jsou vlastnosti objektu - proměnné, tzv. atributy) a Vnitřní chování (což představují jednotlivé metody, které manipulují s daty objektu).

Nový objekt (instance) třídy Inzerát lze vytvořit následujícím způsobem:

```
$inzerat1 = new Inzerat();
```

Třída, datový typ, metody a rozhraní

Pojem třída ve zkratce představuje šablonu, která slouží pro generování (tvorbu) konkrétních objektů. Objekty se stejnými charakteristikami se v aplikacích často mnohokrát opakují. Tyto charakteristiky (struktury dat) nazýváme datové typy [4], a konkrétní datové typy nazýváme třídy.

Říkáme, že objekt s konkrétními charakteristikami je instancí jeho třídy. Instance je datový typ v paměti, vytvořený podle určité třídy.

Každá taková třída musí být v jazyce PHP uvozena klíčovým slovem *Class* a pojmenována, přičemž lze využít libovolnou kombinaci písmen číslic mimo číslice na začátku. Dále musí být její tělo ohraničeno složenými závorkami. Tělo třídy obsahuje konkrétní atributy (pro ukládání dat) a metody. Metody slouží k manipulaci s daty objektu [5]. Programátor může využívat metody statické, které lze zavolat i bez existence objektu třídy, v případě kdy stačí získat statické hodnoty třídy. V případech kdy je třeba vytvořit objekt pro provedení určité metody, které nestačí statická data třídy, je třeba nejprve vložit speciální metodu Konstruktor. Tato metoda *__construct()* slouží pro inicializaci třídy, jak je nazýváno vytvoření objektu třídy. Volání této metody obstarává jazyk PHP automaticky, když programátor v kódu použije operátor *new* [5].

Příklad kódu třídy může vypadat následovně:

```
Class Inzerat {
// tělo třídy (atributy a metody)
public function __construct();
}
```

Speciálním typem datového typu (třídy) je *Rozhraní (Interface)*. Jedná se o rozhraní instancí tříd, které ho implementují (využívají) a obsahuje obecnou definici veřejně přístupných metod. Pomocí těchto veřejně přístupných metod potom lze komunikovat se všemi třídami, které rozhraní využívají, a přitom každá třída může stejnou metodu z rozhraní vykonávat po svém. Uvození rozhraní se v jazyce PHP děje prostřednictvím klíčového slova *interface* a pro implementaci rozhraní určitou třídou musí třída využívat klíčové slovo *implements*, jak osvětluje následující příklad:

```
interface NastenkaCasti
{
    public function vymaz();
}
Class Inzerat implements NastenkaCasti{
Private function vymaz();
}
```

Dědění

Pojem dědění [4] představuje možnost vytvářet v objektově orientovaných programovacích jazycích hierarchii mezi jednotlivými třídami. Výše postavená třída se nazývá předek a níže postavená třída potomek. Při vhodném použití dědění je možné dosáhnout úspory v množství napsaného kódu, protože je možné ve třídě potomka uvádět pouze změny oproti třídě předka.

V jazyce PHP se k určení předka třídy využívá klíčové slovo *extends*.

```
Class InzeratReality extends Inzerat{
// tělo třídy (atributy a metody)
}
```

Polymorfismus

Jako polymorfismus označujeme schopnost objektů na stejné zprávy reagovat různými způsoby [4].

Polymorfismus bývá často názorně vysvětlen na obrázku, kde různá zvířata mohou odpovídat na stejnou zprávu vykonáním stejné metody (mluv), ale každé posvém: „Haf“, „Bůů“ atd. [9]. Tato metoda může mít pro každé zvíře stejnou základní strukturu a změna je jen v konkrétním způsobu vykonání funkce.

Obrázek 1:

Názorný obrázek situace, ve které se využívá polymorfismus. (Zdroj: [9]*)*

3.2 Vývoj a pojmy jazyka PHP

Ačkoli je dnes metodika objektově orientovaného programování užívaná v naprosté většině PHP projektů, nebylo tomu tak hned od počátku vývoje tohoto jazyka [8]. Tento postupný vývoj objektově orientovaných funkcí v jazyce PHP stručně popíši.

V této kapitole zmíním následující verze jazyka [25]:

- PHP Tools, FI, Construction Kit, PHP/FI 2.0: Jazyk PHP, ale ne takový, jako dnes.
- PHP 3: Poprvé přidána základní podpora pro objekty.
- PHP 4: Růst zájmu o objektové programování.
- PHP 5: Definitivní zakomponování podpory objektů do srdce jazyka.
- PHP 7: Velmi rychlé nové jádro a úklid jazyka.

3.2.1 PHP Tools, FI, Construction Kit, PHP/FI 2.0

Vývoj jazyka PHP, jak jej známe dnes, započal u produktu pojmenovaného PHP/FI, jehož tvorbu započal v roce 1994 [2] dánsko-kanadský programátor Rasmus Lerdorf [10]. Na počátku to byl pouze soubor rozhraní Common Gateway Interface (CGI) napsaný v jazyce C, který dokázal automaticky generovat HTML stránky. Lerdorf program původně využíval ke sledování návštěvnosti na svých webových stránkách. Sadu těchto skriptů později pojmenoval „Personal Home Page Tools“, což bylo často zkracováno na „PHP Tools.“ V průběhu následujícího roku Lerdorf tyto nástroje postupně přepsal a přidal jim další funkcionality jako například možnost interakce s databází. Tento nový nástroj umožnil uživatelům vytvářet jednoduché dynamické webové aplikace jako například návštěvní knihu. V červnu roku 1995 pak Lerdorf dokonce zveřejnil kompletní zdrojový kód nástroje, což umožnilo uživatelům využívat jej zcela bez omezení a také to umožnilo uživatelské opravy chyb a mnoho nových uživatelsky vyvynutých funkcionalit kódu. Lerdorf dále pracoval na rozvoji kódu nástroje, který na krátkou dobu přejmenoval na FI, což je zkratka pro „Forms Interpreter.“ V této verzi již přidal některé základní funkce, které známe z dnešního jazyka PHP – například syntaxi založenou na jazyce Perl a vloženou do HTML souboru pomocí komentářů. Popularita FI postupně rostla, stále ještě jako CGI nástroje, ne jako jazyka.

V říjnu 1995 Lerdorf znovu kompletně přepsal zdrojové kódy a znovu se vrátil k názvu PHP jako „Personal Home Page Construction Kit.“ Toto skriptovací rozhraní se konečně rozvíjelo jako programovací jazyk, který byl záměrně navržen s podobnou konstrukcí jako jazyk C, což zajistilo rychlé přijetí vývojáři, kteří byli zvyklí na jazyk C, Perl nebo další obdobné.

Další kompletní přestavbou prošel kód v Dubnu 1996, po které ho Lerdorf pojmenoval kombinací předešlých vydání PHP/FI. Po tomto vydání se PHP opravdu přeměnilo z pouhé sady nástrojů na skutečný programovací jazyk, což zahrnovalo nově podporu pro funkce definované uživatelem, cookies, různé druhy databázových systémů a mnoho dalšího. Ještě v červnu toho roku tato verze PHP/FI dostala první číselné označení 2.0 [2].

3.2.2 PHP 3.0

Teprve po dalším vývoji se PHP ve verzi 3.0 velmi přiblížilo dnešní podobě jazyka. V roce 1997 se kvůli nedostatkům verze PHP/FI 2.0 programátoři Andi Gutmans a Zeev Suraski z univerzity v Tel Avivu rozhodli k dalšímu kompletnímu přepsání jazyka.

Tato verze PHP 3.0 byla velice úspěšná a přitáhla k PHP pozornost mnoha nových vývojářů.

Velkou výhodou této verze byla především snadná možnost rozšíření, která přilákala desítky vývojářů, kteří přišli s řadou modulů. Další klíčový prvek této verze jazyka PHP byla podpora pro objektově orientované programování. Přítomnost objektů však nebyla zamýšlena jako nutná součást nové syntaxe a podpora pro třídy byla přidána až v roce 1997.

Podpora tříd měla v této verzi jazyka stále několik omezení. Například nebyl umožněn přístup k přepsání metody děděné od třídy předka.

Přepracovaná podoba jazyka byla úspěšná takovým způsobem, že byla po devíti měsících testování v červnu roku 1998 vyhlášena nově vzniklým Vývojovým týmem PHP jako oficiální nástupce PHP/FI 2.0. V tu dobu už PHP 3.0 běželo zhruba na 70 000 doménách, což představovalo asi 10% všech tehdejších webserverů [2].

3.2.3 PHP 4.0

Podle [2] se krátce po oficiálním představením PHP 3.0, v zimě roku 1998, programátoři Andi Gutmans a Zeev Suraski rozhodli pracovat na dalším přepsání jádra PHP. Cílem nového návrhu bylo zlepšit výkon PHP pro běh složitých aplikací, které sice byli díky modularitě schopné běžet v prostředí PHP 3.0, ale tato verze nebyla navržena pro efektivní práci s tak náročnými aplikacemi. Nové jádro, implementované v jazyce C a pojmenované Zend Engine (podle křestních jmen autorů), bylo představeno v polovině roku 1999.

Verze PHP 4.0, která byla založena na tomto jádře, byla vydána v květnu roku 2000. Kromě významného vylepšení výkonu přináší nová verze PHP další důležitá vylepšení. Mezi nimi je například podpora více webových serverů, podpora http sessions, nebo zlepšení bezpečnosti zpracování předávaných dat [2].

Právě Zend Engine [6] umožnil lepší využití programových konstrukcí, jako jsou třídy. Z hlediska objektového programování spočívala hlavní výhoda v možnosti přepsání metody rodičovské třídy, ke které byl nově umožněn přístup i ze třídy potomka.

V té době se stupňoval zájem o paradigma objektového programování, což naznačovaly především různé online webové články. V oficiálním PHP manuálu byla rozšířena syntaxe založená na objektovém paradigma [8].

Za zmínku stojí i fakt, že objektové paradigma vznikalo již v 70. letech, avšak mezi běžné programátory se začínalo prosazovat až v letech 80. a masivní popularizace se dočkalo až od roku 1995, kdy významně pomohlo vydání knihy, jasně definující základní zásady paradigma, *Design Patterns: Elements of Reusable Object-Oriented Software [18]* a také s příchodem jazyka Java. Také procedurální jazyk PHP prokázal schopnost vyslyšet požadavky svých uživatelů, právě pozoruhodně rychlým implementováním podpory pro objekty [8]. Objektové programování také převzal projekt PEAR, což je oficiální softwarové úložiště PHP. Mnoho balíčků, které PEAR nabízel, využívalo objektových návrhových vzorů.

Objektové programování v PHP se začalo dostávat stále více do popředí, což dokazuje i článek [8] Zeeva Suraskiho, který napsal v roce 2002 pro DevX.com.

Suraski v článku píše, že jedním z největších zvratů ve vývoji PHP bylo to, že i přes velmi omezenou funkčnost a mnoho problémů při implementaci objektového paradigma v jazyce PHP se toto paradigma stalo nejpopulárnějším v rostoucím počtu aplikací napsaných v PHP.

3.2.4 PHP 5.0

S další verzí jazyka PHP 5.0 přišla definitivní podpora objektového programování. Jazyk si však stále drží určitý zdravý odstup a ponechává uživateli volbu, zda bude objekty využívat či nikoli.

Vývojový tým PHP, který už čítá desítky vývojářů, vypustil oficiální verzi PHP 5.0 v červenci roku 2004 po dlouhém vývoji s řadou testovacích verzí [2]. Krom řady nových funkcí tvůrci rozpoznali výjimečný význam objektů především pro rozsáhlé podnikové systémy a umožnili jejich plnou podporu v novém jádře Zend Engine 2.0 novým objektovým modelem a dalšími novými funkcemi objektů. Velmi důležitou změnou bylo defaultně nastavené chování objektů na „předávání pomocí reference“ („pass by reference“) místo jejich kopírování, což velmi ulehčilo psaní kódu [1].

Dalšímu růstu zájmu o jazyk PHP pravděpodobně pomohlo jeho využití velkými společnostmi jako je Facebook nebo Yahoo, které ho díky svým novým vlastnostem začali hojně využívat pro svoje platformy [8].

S verzí jazyka 5.0 se PHP postupně stalo jedním ze základních standardů pro vývoj webových technologií.

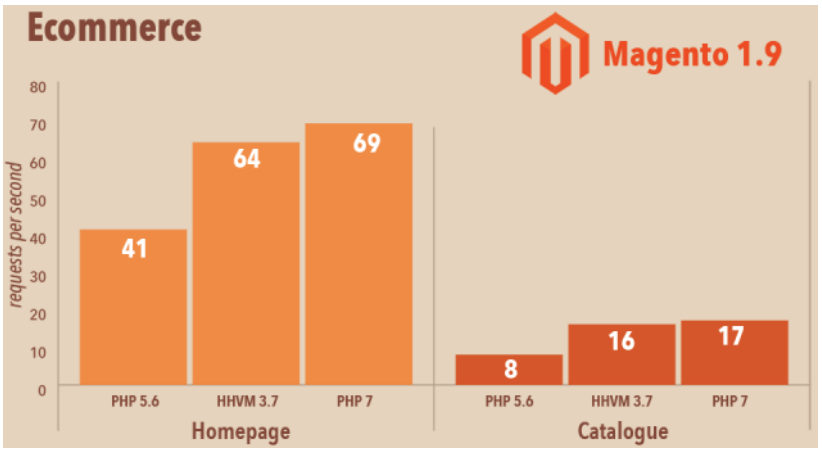
Vývoj další verze PHP 6.0 se potýkal s mnohými problémy a vydání oficiální verze se dlouho odkládalo. Před vydáním další verze jazyka se rozhořeli ostré debaty napříč komunitou vývojového týmu PHP i dalších vývojářů ohledně označení příští hlavní verze. Někteří namítali, že většina nových změn již byla kvůli zpoždění implementována do verzí 5.3, 5.4, 5.5 a 5.6, což bylo poslední stabilní vydání představené v roce 2014. Nakonec převládl a byl v červenci 2014 odhlasován názor podporující označení 7.0 místo nepovedeného projektu 6.0 [8].

3.2.5 PHP 7.0

Po dlouhém čekání, na začátku prosince 2015, konečně představil vývojový tým PHP novou verzi 7.0 [12].

Nejčastěji zmiňovanou výhodou nové verze PHP je znatelné vylepšení výkonu, které umožnilo nově přepsané jádro Zend Engine 3.0. Dle prohlášení na webu společnosti Zend technologies Ltd [6] se podařilo zvýšit výkon oproti verzi 5.6 až dvojnásobně a vylepšený systém využití paměti uspoří až 50% paměti.

Zend Technologies Ltd dokonce zveřejnila výsledky výkonových testů (tzv. benchmarků) pro některé známé PHP aplikace.



Obrázek 2:

Obrázek znázorňující výsledky výkonového testu systému Magento 1.9 v nové i předchozí verzi jazyka PHP. (Zdroj: [6]*)*

Jak je vidět například na tomto výkonovém testu systému Magento ve verzi 1.9, který je využíván především pro řízení internetových obchodů, nová verze PHP tu opravdu znamená nezanedbatelný nárůst výkonu.

Toto znatelné navýšení výkonu se stejným hardwarem je velmi lákavé i pro korporátní uživatele, kteří jinak s přechodem na novou verzi jazyka nespěchají. Spolu s rozsáhlou zpětnou podporou kódu pro verzi 5.6 je možné očekávat rychlé rozšíření této verze.

Společně s vylepšením výkonu přišli také další důležité změny. PHP 7.0 zavádí konzistentní podporu pro 64 bitovou architekturu, což umožňuje užití jazyka na 64 bitových systémech Windows.

Mezi cíle nové verze jazyka patří také úklid. Jedná se o odstranění různých zastaralých funkcí a SAPI (zkratka pro „Server Application Programming Interface“, což jsou rozhraní umožňující komunikaci mezi PHP a různými webovými servery).

PHP v předchozí verzi obsahovalo SAPI pro servery které už nejsou podporovány a jejich vydavatelé je už mnohdy ani nekomerčně nenabízí. Také některé starší funkce a doplňky přestali být k dispozici a zůstali jenom jejich novější podporované verze. (Například rozšíření mysql a mysqli.)

Anatol Belski k tomu ve svém článku „*Removal of dead or not yet PHP7 ported SAPIs and extensions*“ [13] doslova napsal:

„*This leads to the situation where the repository contains a lot of unmaintained code. The code which doesn't become any updates for years or relies on long unmaintained libraries. Besides that code being not maintained, it is also kind of security risk* [13].“

Ve volném překladu tím chtěl říci, že PHP obsahuje množství již léta neudržovaného kódu, nebo kódu, který se spoléhá na neudržované knihovny a to znamená mimo zastaralé funkce také bezpečnostní riziko.

Pro uvolnění prostoru pro další zdokonalování bylo třeba k tomuto pročištění přistoupit i přes možné problémy. Zastaralé odstraněné položky již pravděpodobně většina vývojářů nepoužívá, nicméně je nutné brát na vědomí, že některé aplikace psané pro starší verze jazyka mohou být pro běh v PHP ve verzi 7 nekonzistentní a jejich běh tedy může být narušen.

Pro podrobnou kontrolu odstraněných funkcí můžete využít článek Nikota Popova: *Remove Deprecated functionality in PHP 7* [14].

Mezi další nové funkcionality této verze jazyka patří například možnost využití anonymních tříd, které mohou urychlovat kódování i vykonávání kódu, nebo možnost deklarovat návratový typ funkce a to i pomocí čtyř nových skalárních datových typů *boolean, integer, float a string*. Vývojář tak může vyznačit požadovaný návratový typ jako booleovskou hodnotu, celé číslo, číslo s pohyblivou čárkou nebo řetězec [12].

3.3 Návrhové vzory

Některé problémy, které PHP vývojář musí řešit, se velmi často opakují napříč různými projekty. Opakovaně musíme řešit například otázky, jakým způsobem budeme prezentovat a jak ukládat různé druhy dat, nebo jak vytvoříme webu navigaci.

Na tyto a jim podobné (opakované) postupy si postupem času najde každý vývojář svoje vlastní lépe či hůře navrhnuté odpovědi (řešení). Vytvoří si tak vlastní soubor zvyklostí, technik a dalších zažitých postupů, který můžeme pojmenovat jako osobní návrhové vzory.

Návrhové vzory (anglicky Design Patterns) popisují určitý častý problém softwarového inženýrství (návrhu počítačových programů) a předkládají doporučený postup jeho řešení. Většina vzorů je, nebo by měla být, popsána na základě praktických zkušeností a pozorování programátorů, nikoli na základě teorie.

Zkušenější vývojář proto může po prostudování konkrétního návrhového vzoru rozpoznat techniky, které sám používá, ke kterým již dospěl na základě vlastních zkušeností.

Návrhové vzory poskytují přiblížení postupu vhodného řešení, nejsou to však hotová řešení ve formě knihoven či částí zdrojového kódu. Často jsou sice pro ilustraci uváděny příklady implementace jednotlivých vzorů, nicméně vzory samotné nabízejí pouze koncept elegantního řešení obecného problému, které může být využito pro řešení konkrétní situace.

Jak trefně uvádí Rudolf Pecinovský ve své učebnici *Java 8: Úvod do objektové architektury pro mírně pokročilé,* návrhové vzory jsou programátorské ekvivalenty matematických

vzorečků, v nichž místo čísel vystupují objekty a datové typy – neboli doporučení pro řešení často se opakujících problémů [4].

Objektově orientované návrhové vzory, na které s v této práci zaměřím, tedy popisují vztahy mezi třídami a objekty ale nepředkládají konkrétní implementaci (kromě drobných příkladů).

3.3.1 Význam návrhových vzorů

Návrhové vzory především umožňují méně zkušeným vývojářům využít dlouhodobé poznatky dalších vývojářů a efektivně (rychleji) tak vyvíjet funkce, které se v různých aplikacích často opakují.

Velmi mě oslovilo následující shrnutí v přednášce Miroslava Beneše, Návrhové vzory pro J2EE: „Návrhové vzory zabraňují objevování již objeveného [15].“

3.3.2 Stručná historie návrhových vzorů

Inspirace užití návrhových vzorů pro obor výpočetní techniky přišla z oblasti mimo počítačový průmysl, z architektury. Vznik konceptu vzorů se datuje do sedmdesátých let 20. století, ke knize A Pattern Language: Towns, Buildings, Construction [19]. Autor knihy, architektonický akademik Christopher Alexander (a několik dalších) zde rozebírá 253 architektonických vzorů. Tyto vzory dohromady vytváří „pattern language“ (v překladu „jazyk vzorů“, jak jej pojmenoval autor.) Tento koncept vzorů, které popisují základní schéma praxí ověřené-ho návrhu budov a městských částí v obecném smyslu, dokázal Alexander popsat velmi originálně a zaujmout tak mnoho odborníků.

Díky této publikaci postupně začínali být komunitě softwarových vývojářů zřejmé některé podobnosti architektonického návrhářství a softwarovém návrhářství.

Softwarový návrháři Ward Cunnigham a Kent Beck po čase použili Alexanderův koncept vzorů a vytvořili „programming pattern language“, který pomocí pěti návrhových vzorů sloužil jako vstupní příručka objektově zaměřeného programovacího jazyka Smalltalk [16].

Tuto práci představili v roce 1987 na Object-Oriented Programming, Systems, Languages & Applications Conference (OOPSLA, což je konference zaměřená na objektově orientované programování) a tím podnítili zájem o vzory u dalších vývojářů [7].

Programátoři Erich Gamma a Richard Helm v roce 1990 zjistili, že mají společné zájmy ohledně téma návrhových vzorů a začali společně vytvářet první katalog vzorů využitelných v softwarovém návrhu. Identifikovali čtyři takové vzory: Composite, decider, observer a constrainer [16].

Na stejné konferenci objektového programování (OOPSLA) v roce 1991 se vzory staly velmi diskutovaným tématem. K Erichovi Gammovi a Richardu Helmovi se připojili

programátoři Ralph Johnson a John Vlissides a společně s dalšími zaujatými týmy vývojářů definitivně započali výzkum objektově orientovaných návrhových vzorů.

Skupina těchto čtyř programátorů (Erich Gamma, Richard Helm, Ralph Johnson a John Vlissides), později známá jako „Gang of Four“ (GoF, neboli „Banda čtyř“), napsala v průběhu několika let knihu *Design Patterns: Elements of Reusable Object-Oriented Software* [18], která popisuje 23 návrhových vzorů rozdělených do tří kategorií ve formě katalogu.

Zkratkou GoF bývá tato skupina a přeneseně i jejich katalog návrhových vzorů často označována i v odborné literatuře [3]. V této práci se této zvyklosti budu držet také a zkratkou GoF se budu na tuto literaturu odvolávat.

Zájem o vzory kulminoval na konferenci OOPSLA v roce 1994, kde skupina GoF prezentovala právě svou novou knihu a získala tak většinu pozornosti.

Kniha byla vydána v roce 1995 a stala se velmi úspěšnou průkopnickou prací v oblasti objektově orientovaných návrhových vzorů a v oblasti objektového programování obecně. Od představení knihy zájem o návrhové vzory v oblasti softwarového návrhu exponenciálně rostl a mnoho vývojářů pokračovalo v dalším vývoji.

V této práci budu využívat formu rozdělení návrhových vzorů podle zmiňované knihy autorů GOF: Design Patterns: Elements of Reusable Object-Oriented Software, tedy tyto tři základní kategorie:

- Creational Patterns (návrhové vzory vytváření)
- Structural Patterns (strukturální návrhové vzory)
- Behavioral Patterns (návrhové vzory chování)

3.3.3 Popis a struktura návrhových

Různé publikace popisují návrhové vzory různými způsoby, některé velmi formálně a některé méně. Nicméně většina se shoduje přinejmenším v základním obsahu. Tím jsou tyto sekce: Název vzoru, problém na který se vzor zaměřuje a popis řešení daného problému.

Návrhový vzor může být mimo tyto základní sekce popisován také vzorovými příklady implementace na nějaké konkrétní situaci a často jsou také diskutovány výhody a nevýhody následků užití vzoru.

Mnoho knih uvádí příklady implementace návrhových vzorů v jazyce Java nebo C. V této práci budu u některých vzorů uvádět příklady implementace v jazyce PHP. Tyto příkladové využití vzorů v jazyce PHP nebudou sloužit k přímému využití, ale budou díky jednoduchým situacím sloužit k co nejsnadnějšímu pochopení podstaty vzoru.

Pro zápis struktury objektově orientovaných návrhových vzorů jsou nejčastěji využívány grafické UML diagramy tříd, které umožňují definici tříd a jejich vazeb na obecné úrovni. UML, neboli Unified Modeling Language (v překladu unifikovaný modelovací jazyk), je jazyk využívaný v softwarovém inženýrství především pro návrh softwaru a jeho vizualizaci. Stojí za ním neziskové konsorcium pro technologickou standardizaci Object Management Group (OMG). Jedná se o standardizační organizaci mnoha technologických firem, přičemž členství v organizaci je otevřené a zdarma. Více informací najdete na webových stránkách organizace [17].

3.3.4 Proč používat návrhové vzory v jazyce PHP

Návrhové vzory definují objekty s jejich vazbami obecně při využívání objektově orientovaného programování, což z principu umožňuje jejich využití v různých programovacích jazycích, které podporují toto paradigma. Mnoho vzorů je možné aplikovat mezi různými jazyky pouze s drobnými změnami, nicméně neplatí to tak vždy. Některé návrhové vzory pracují dobře s jazyky, které umožňují ucelený běh aplikace na serveru po delší dobu.

Jazyk PHP takto nefunguje. Aplikace se načítá opakovaně s každým http požadavkem. Vždy to neznamená, že takovéto vzory nemůžeme v PHP použít s dobrými výsledky. Musíme však zajistit že budeme brát v úvahu problémy, které mohou při použití PHP nastat. [1]

Jazyk PHP má velikou výhodu v jednoduchosti a rychlosti s jakou mohou začátečníci proniknout do jeho základů. Pro začátečníka v oblasti tvorby webových prezentací není obtížné vložit do svého HTML dokumentu pár řádků kódu PHP a změnit koncovku dokumentu z *.html* na *.php*, čímž může snadno docílit oživení své prezentace o zajímavé funkce. Skromné začátky tohoto jazyka ho sice nevyvedli na výsluní korporátních programovacích jazyků, nicméně pozdější velmi zdařilá práce týmu Zend [6] a převzetí jazyka velkými společnostmi ukázali, že jazyk PHP je připraven i pro korporátní sféru. Především vyspělá podpora moderního paradigma objektově orientovaného programování pomohla rozšíření jazyka do náročných a rozsáhlých projektů, kde se toto paradigma stává stále užívanější [8]. Díky tomu dnes můžeme při objektově orientovaném návrhu softwaru v PHP využít potenciál návrhových vzorů, čímž se můžeme vyvarovat zpátečnických řešení pomocí testovaných a praxí prověřených postupů [25].

Uživatelsky příjemnému využívání OOP a návrhových vzorů v jazyce PHP také významně pomáhá trend poslední doby a to Aplikační rámce (anglicky Frameworky) [25]. Jsou to velmi užitečná rozhraní pro programování aplikací, využívající různé podpůrné knihovny a často obsahující zabudovanou podporu pro snadné nasazení návrhových vzorů. Tyto Aplikační rámce jsou často velmi užitečné – pomáhají především s tvorbou mnoha jednoduchých nebo často se opakujících funkcionalit. Vhodným příkladem Aplikačního rámce s podporou pro návrhové vzory je například Zend Framework [30].

4 Příručka vybraných návrhových vzorů

V této části práce budou představeny základní informace o vybraných návrhových vzorech, které jsou více či méně užitečné pro využití v jazyce PHP. U popisovaných návrhových vzorů bude uvedena základní pointa, která stručně popisuje jejich význam, účel vzoru a postup řešení daného problému. Vybrané vzory budou doplněny o ukázkovou implementaci zpracovanou v jazyce PHP.

Následující tabulka zachycuje strukturu vzorů, které budou v této příručce uvedeny, a navíc obsahuje stručnou pointu jednotlivých vzorů.

Tabulka 1: *Tabulka zachycuje popisované vzory a krátkou charakteristiku jejich pointy.*

Název vzoru	**Pointa vzoru**
Vzory pro vytváření (creational patterns)	
Jedináček (Singleton)	Návrhový vzor Jedináček umožňuje vytvořit pouze jedinou instanci třídy a zajistit k ní globální přístup.
Tovární metoda (Factory Method)	Návrhový vzor Tovární metoda definuje třídu s rozhraním pro vytváření (a předávání) objektů, přičemž vlastní tvorbu objektů již přenechává svým potomkům, kteří rozhodují o konkrétní implementaci.
Stavitel (Builder)	Návrhový vzor Stavitel umožňuje oddělit konstrukci komplexních objektů od jejich reprezentace a to takovým způsobem, aby se dal způsob vybudování objektu opakovaně využít pro různé instance.
Prototyp (Prototype)	Návrhový vzor Prototyp spočívá ve způsobu vytváření objektů pomocí klonování již existujícího prototypu objektu.
Strukturální vzory (structural patterns)	
Strom (Composite)	Základem tohoto vzoru je stromová struktura, která umožňuje uspořádání jednotlivých objektů do větších celků, složených (neboli kompozitních) objektů. Pro přístup k jednotlivým i složeným objektům se využívá stejný způsob, což usnadňuje případné manipulace.
Adaptér (Adapter)	Použití tohoto návrhového vzoru je vhodné, pokud je potřeba propojit dvě třídy s rozhraními neschopnými vzájemné spolupráce.

Most (Bridge)	Tento návrhový vzor odděluje rozhraní a implementaci (zavádění metod) třídy a umožňuje tak jejich nezávislé změny.
Dekorátor (Decorator)	Návrhový vzor slouží pro přidávání funkcionalit již existujícím třídám, které nechceme z různých důvodů upravovat. Tyto nové funkce či změny stávajících metod lze provádět za běhu programu.
Fasáda (Facade)	Základní myšlenkou tohoto vzoru je poskytnout zjednodušené rozhraní (fasádu) k určité logické skupině rozhraním systému a umožnit tak zjednodušený přístup k částem aplikace bez nutnosti znát kompletní funkci systému za fasádou.
Muší váha (Flyweight)	Tento návrhový vzor se využívá pro zajištění úspory další paměti při využívání velkého počtu téměř shodných objektů.
Vzory chování (behavioral patterns)	
Pozorovatel (Observer)	Pomocí návrhového vzoru Pozorovatel lze vytvořit mezi subjektem (pozorovaným objektem) a libovolným počtem pozorovatelů závislost typu 1:n. Všichni pozorovatelé jsou automaticky informováni o změně stavu subjektu.
Prostředník (Mediator)	Základní myšlenkou vzoru je způsob interakce (spolupráce) objektů pomocí objektu prostředníka, který mění vazby objektů z typů m:n na 1:n. Struktura odstraňuje vzájemné (přímé) vazby mezi objekty a zároveň umožňuje komunikaci objektů, které nemají přímé komunikační vazby.
Šablonová metoda (Template Method)	Návrhový vzor Šablonová metoda vytváří základní rozhraní algoritmů pomocí konkrétní metody. Rozhraní algoritmu rozděleného v jednotlivých krocích, které poté implementují potomci. Tito potomci slouží jako zaměnitelné části algoritmu.
Příkaz (Command)	Vzor slouží k zapouzdření vykonání příkazů do podoby objektů, což umožní přidávat tyto požadavky do fronty požadavků jiného objektu a využívat operace prováděné zpětně.

Návštěvník (Visitor)	Tento návrhový vzor umožňuje přidávání nových operací do struktur objektů, jejichž struktura nelze měnit. Nové operace jsou zapouzdřeny do samostatné třídy.
Iterátor (Iterator)	Návrhový vzor Iterátor poskytuje možnost postupně přistupovat k prvkům složených objektů bez znalosti jejich základní struktury.
Stav (State)	Tento návrhový vzor slouží pro správu různorodých stavů objektů. Použití vzoru Stav poskytne možnost současně měnit chování objektů při změně jejich vnitřních stavů.
Strategie (Strategy)	Návrhový vzor Strategie zapouzdřuje skupinu algoritmů a umožnuje tak výměnu právě používaného algoritmu nezávisle na objektu který algoritmus využívá.
Zřetězení zodpovědnosti (Chain of Responsibility)	Návrhový vzor Zřetězení zodpovědnosti odděluje původce požadavku od jeho příjemce, což umožnuje, aby požadavek obdrželo a provedlo více objektů. Pomocí vzoru lze vytvářet zřetězené struktury objektů, které postupně obdrží konkrétní požadavek, dokud ho některý objekt nezpracuje.

4.1 Vzory pro vytváření

V této kapitole popisuji následující čtyři vzory využívané při tvorbě objektů:

- Jedináček (Singleton)
- Tovární metoda (Factory Method)
- Stavitel (Builder)
- Prototyp (Prototype)

4.1.1 Návrhový vzor Jedináček (Singleton)

Pointa

Návrhový vzor Jedináček umožňuje vytvořit pouze jedinou instanci třídy a zajistit k ní globální přístup.

Účel vzoru

Vzor využíváme v případě, kdy potřebujeme, aby v aplikaci existovala třída s pouze jednou globální instancí.

Globální proměnné jsou pro PHP programátory i pro objektově zaměřené programátory obecně častým zdrojem problémů. V případě spoléhání se na globální proměnné se v rozsáhlejších aplikacích často stává, že globální proměnná je přepsána určitou třídou zatímco jiná třída zde měla uložena důležitá data. Podobný problém může PHP vývojář zažít, pokud nepoužívá jmenné prostory a dojde ke kolizi dvou tříd se stejným jménem [1].

Jazyk PHP neumí vývojáře před kolizí globálních proměnných varovat a její projevy se obvykle odhalují velmi těžko.

Jsou však určité situace, kdy stojí zato globální proměnné použít i přes možná rizika.

Dobrým příkladem je například třída, která slouží pro připojení k databázi. Postup připojení k určité databázi je zpravidla stále stejné a není třeba zbytečně vytvářet stále nové připojení při každém požadavku. Pro dosažení takového znovupoužití objektů využijeme právě návrhový vzor Jedináček.

Jedná se o jeden z vůbec nejpoužívanějších návrhových vzorů. Třída vytvořená podle tohoto vzoru při počátečním požadavku na objekt tento objekt vytvoří, uloží a předá jeho reference (odkaz na místo uložení). Při každém dalším požadavku na nový objekt, třída prověří svou paměť, a pokud najde již vytvořený objekt, předá dál jeho reference.

Využití tohoto návrhového vzoru může poskytovat základní obranu právě proti problémům, které přicházejí s použitím globálních proměnných.

Programátor Aaron Saray ve své knize *Professional PHP Design Pattern* [20] napsal:

„*Many PHP packages have been guilty of having a strong object - oriented core but stillmaking use of the $GLOBALS array. The most common reason they implement the global variable usage is for configuration options. Instead, a configuration object could be created as a Singleton* [20].“

Zjednodušeně přeloženo: Mnoho PHP projektů, které jsou objektově navržené, stále využívá globální pole pro konfigurační možnosti aplikace. Namísto tohoto řešení by vývojáři měli využít možnost vytvoření konfiguračních objektů podle návrhového vzoru Jedináček.

Výhodou využití objektu místo klasické globální proměnné v této situaci poskytuje možnost přidat dodatečná bezpečnostní opatření, která znemožní úpravu konfiguračních objektů po počátečním vytvoření.

Velmi často je tento vzor využíván právě pro již zmiňovanou tvorbu objektů sloužících pro připojení k databázi. Vzhledem k tomu, že připojení k databázi může být náročné například na čas, je vhodné navrhnout objekt sloužící pro připojení k databázi jako Jedináček. Při každém dalším požadavku na novou instanci se tak znovu využije první připojení a ušetří se tak prostředky pro tvorbu nových databázových připojení.

Řešení problému

Řešení tohoto problému pomocí vzoru Jedináček představuje určitá třída, která se stará o to, aby mohla mít maximálně jednu centrálně spravovanou instanci. Základními kameny takovéto funkčnosti jsou: privátní konstruktor, statická proměnná *$instance* a statická metoda *getInstance()*.

V určené třídě vytvoříme soukromou statickou proměnnou, která bude sloužit pro uložení oné jediné instance třídy, kterou budeme v aplikaci sdílet. Pro omezení vytváření dalších instancí mimo třídu je potřeba skrýt konstruktor pomocí příznaku *private*.

Nakonec je třeba vytvořit veřejnou statickou metodu, pomocí které budeme zvenku přistupovat k instanci třídy. Statickou proto, že takovou metodu lze zavolat bez vytvoření instance třídy, která metodu obsahuje. Metoda musí při volání ověřit, zda již instance třídy existuje, vrátit tuto instanci pokud ji najde a v případě prvního volání musí instanci vytvořit a uložit.

Vytvoření instance může být implementováno dvěma způsoby. Uvedený způsob vytvoření až v případě potřeby (při prvním zavolání) bývá označován jako „Lazy loading“ (v překladu „líné načítání“). Druhý způsob se nazývá „eager loading“ (v překladu „dychtivé načítání“) a spočívá ve vytvoření instance hned při iniciaci třídy.

Problémy při využití vzoru

Při použití tohoto vzoru na rozsáhlý projekt, na kterém pracuje více programátorů, mohou vznikat při neinformování ostatních problémy.

Aby ostatní programátoři využívali pro získání instance metodu getInstance() je klasický konstruktor deklarovaný jako *private* a pokud se tedy někdo pokusí vytvořit novou instanci objektu, dostane v jazyce PHP následující hlášení: „*Fatal error: Call to protected*.“ Někteří vytrvalí programátoři se však mohou pokoušet získat další instanci pomocí metody clone (klonování). Tomuto klonování objektu lze zabránit zakázáním zavolání metody *__clone()* mimo třídu, tedy nastavením příznaku *private* [1].

Ukázkový kód (PHP)

Tento příklad předvádí využití vzoru Jedináček pro zmiňované připojení k databázi pomocí funkce *mysqli*.

Výpis 1: *Ukázková implementace návrhového vzoru Jedináček v jazyce PHP.*

```
<?php
/*
*Třída která umožní pouze jediné Mysqli databázové připojení
*/
class DatabaseConnection
{
//Proměnná, která slouží pro uložení jedinečné instance třídy
private static $instance = NULL;
private $pripojeni;
private $host = "localhost";
private $username = "uzivatel";
private $password = "heslo";
private $database = "databaze";

/*
*Následuje funkce getInstance pro získání instance databázového připojení
*Funkce vrací proměnnou $instance
*/
public static function getInstance() {
if (self::$instance === NULL) { //Funkce prověří, zda již existuje instance
self::$instance = new self();
//Pokud instance neexistuje, funkce ji vytvoří
}
return self::$instance;
}

//Konstruktor
private function __construct() {
$this->pripojeni = new mysqli($this->host, $this->username, $this->password,
$this->database);  //Vytvoří nové Mysqli připojení

//Chybové hlášení při neúspěšném připojení
if(mysqli_connect_error()) {
trigger_error("Připojení k MySQL serveru se nezdařilo: " .
mysqli_connect_error() . " (" . mysqli_connect_errno() . ")" );
}

}

//Dále je třeba zamezit klonování objektu
private function __clone() {}
```

```
//Funkce pro získání mysqli připojení
public function getConnection() {
return $this->pripojeni;
}

}
?>
```

4.1.2 Návrhový vzor Tovární metoda (Factory Method)

Pointa

Návrhový vzor Tovární metoda definuje třídu s rozhraním pro vytváření (a předávání) objektů, přičemž vlastní tvorbu objektů již přenechává svým potomkům, kteří rozhodují o konkrétní implementaci. Vytváření instancí objektů je tak lépe kontrolovatelné.

Variace vzoru

Tovární metoda je obecný vzor popsaný v GoF [18]. Tato obecná tovární metoda je nejčastěji deklarována jako abstraktní metoda nějakého rozhraní. Kromě této obecné formy existuje i několik různých odvozených variant (Tovární Metoda, Jednoduchá Tovární Metoda, Instanční Tovární Metoda, Abstraktní Tovární Metoda a další), které se různě liší, kromě faktu, že všechny jsou určeny pro pře-dávání odkazů instancí.

V této práci se zaměřím především na variantu s názvem Jednoduchá Tovární metoda. Tato metoda bývá někdy také nazývána jako Statická tovární metoda. To proto, že nejčastěji bývá implementována jako statická metoda třídy, na jejíž instanci vrací odkaz [1]. Statická je metoda z důvodu možnosti zavolání i bez vytvořené instance třídy, která metodu obsahuje. S využitím tohoto návrhového vzoru se můžeme setkat i u základní podoby vzoru Jedináček.

Účel vzoru

Při různých situacích je výhodné neposkytovat mimo konkrétní třídu její konstruktor, který pokaždé při požadavku na odkaz instance bezpodmínečně vytvoří instanci novou, ať se to hodí nebo ne. Pokud nějaká část aplikace potřebuje pro svůj chod odkaz na instanci určité třídy, je možné definovat tzv. Tovární metodu, která bude užívána namísto konstruktoru a bude předávat odkaz instance. Tovární metoda má nad konstruktorem výhodu především v možnosti využití již existující instance, což konstruktor nedokáže. Pouze pokud požadovaná instance ještě neexistuje, Tovární Metoda ji vytvoří.

Další velká výhoda Tovární Metody oproti konstruktoru je možnost vracet instance různých návratových typů – například instanci potomka třídy.

I co se týče syntaxe, přináší použití této metody výhody. Především se jedná o možnost tovární metody (na rozdíl od konstruktoru) pojmenovat a pro konstrukci instancí také není třeba využívat operátor *new*.

Tovární metoda bývá často využívána také pro uchovávání obecných údajů potřebných pro vytvoření objektu. Při opakované tvorbě nějakého objektu na různých částech aplikace často dochází k situaci, kdy je zbytečně opakován stejný kód pro vytvoření a nastavení instance.

Příkladem mohou být například objekty tlačítek. Vytvořit instanci takového tlačítka může znamenat i několik řádků kódu – je třeba nastavit parametry jako rozměry, text, barvu a další.

Pokud je takových tlačítek v aplikaci více, vyplatí se oddělit kód od metody pomocí Tovární Metody. V případě, že jsou tlačítka v aplikaci stejná až na některé atributy, např. popisný text, lze s použitím Tovární Metody upravovat pouze tento popisný text. Ostatní vlastnosti uchovává tato metoda a není třeba je opakovat.

Ukázkový kód (PHP)

Pokud budeme v určité aplikaci opakovaně vytvářet instance třídy *MobilniTelefony*, které mají automaticky nastavené mnoho parametrů, vyplatí se využít návrhový vzor Tovární Metoda. V případě, že budeme v naší aplikaci vytvářet mnoho mobilních telefonů s kompletními specifikacemi, tato metoda poskytne úsporu kódu [1].

Výpis 2: *Ukázková implementace návrhového vzoru Tovární metoda v jazyce PHP.*

```
<?php
/*
*Třída, která obsahuje Tovární Metodu pro vytváření instancí mobilních telefonů
*/
class MobilniTelefon
{
        private $Vyrobce;
        private $Model;
    private $Kapacita;

//Konstruktor
private function __construct($Vyrobce, $Model, $Kapacita) {
$this->Vyrobce = $Vyrobce;
$this->Model = $Model;
$this->Kapacita = $Kapacita;

}


        public static function TovarnaNaMobilniTelefon Iphone6s()
        {
```

```
                $iphone6s = new MobilniTelefon("Apple", "6s", "32");
            return $iphone6s;
        }
}
?>
```

4.1.3 Návrhový vzor Stavitel (Builder)

Pointa

Návrhový vzor Stavitel umožňuje oddělit konstrukci komplexních objektů od jejich reprezentace a to takovým způsobem, aby se dal způsob vybudování objektu opakovaně využít pro různé instance.

Účel vzoru

Jedná se o další vzor vytváření objektů. Vzor Stavitel se využívá tam, kde je třeba nejenom vytvořit určité objekty, ale také jim nastavit další vlastnosti (typicky mnoho vlastností). Obecně je výhodné vzor využít téměř pokaždé když nějaká třída potřebuje k vytváření objektů jakékoli nastavení [1]. Naopak pokud je třeba vytvořit objekt bez dalšího nastavení, je vhodnější využít jiných vzorů – například vzor Tovární metoda.

Tento vzor proto najde (jako většina vzorů) své opodstatnění především v rozsáhlejších projektech [1].

Řešení problému

Návrhový vzor stavitel stojí na třech základních kamenech:

- Produkt (Product) - představuje určitou výslednou instanci, kterou je třeba vytvořit
- Řídící objekt (Director) - tento objekt se stará o postupné vytváření jednotlivých částí produktu ve správném pořadí pomocí výkonných objektů
- Výkonný objekt (Builder) - Neobsahuje logiku posloupnosti využití svých metod, ale využívá tyto metody pro vytvoření konkrétních částí produktu. Výkonný objekt by měl mít několik metod, každou pro nastavení jedné určité záležitosti a také metodu *getResult()*, která vrátí připravený objekt. Výkonných objektů může být více, ale všechny by měli implementovat stejné rozhraní. Výkonný objekt nevolá přímo svoje metody, ale předá svou instanci řídícímu objektu, který by měl mít tvořící metodu (*build* metodu) pro postupné volání jednotlivých funkcí výkonného objektu, což je umožněno díky stejnému rozhraní které implementují.

Možné problémy při použití vzoru

I použití tohoto návrhové vzoru může být v některých případech kontraproduktivní (působící nevhodně).

Například již zmiňované využití vzoru v jednoduchých aplikacích v případech, kdy je třeba vytvořit instanci nějaké třídy bez dalšího nastavení.

Využití vzoru Stavitel také způsobí obtížnější ladění kódu, s čímž je třeba počítat při rozhodování, zdali vzor použít [1].

Ukázkový kód (PHP)

Výpis 3: *Ukázková implementace návrhového vzoru Stavitel v jazyce PHP.*

```
class Oprava
{
public $povaha;
 const ZARUKA   = "Záruční oprava";
 const NEZARUKA = "Mimozáruční oprava";
}

interface OpravaBuilderInterface
{
public function setPovaha();
public function getResult();
}

//Výkoný objekt (Builder) pro vytvoření Záruční opravy
class ZarucniOpravaBuilder implements OpravaBuilderInterface
{
private $oprava;
 public function __construct()
    {
        $this->oprava = new Oprava();
    }

    public function setPovaha()
    {
        $this->oprava->povaha = Oprava::ZARUKA;
    }

    public function getResult()
    {
        return $this->oprava;
    }

}
```

```
//Výkoný objekt (Builder) pro vytvoření Mimozáruční opravy
class MimoZarucniOpravaBuilder implements OpravaBuilderInterface
{
private $oprava;
 public function __construct()
    {
        $this->oprava = new Oprava();
    }

    public function setPovaha()
    {
        $this->oprava->povaha = Oprava::NEZARUKA;
    }

    public function getResult()
    {
        return $this->oprava;
    }

}


//Řídící objekt (Director) zavolá všechny metody Výkoného objektu a vrátí objekt
třídy Oprava
class OpravaDirector
{

 public function build(OpravaBuilderInterface $builder)
    {
        $builder->setPovaha();
    return $builder->getResult();
    }

}

$director                    = new OpravaDirector();
$ZarucniOpravaBuilder        = new ZarucniOpravaBuilder();
$MimoZarucniOpravaBuilder    = new MimoZarucniOpravaBuilder();
//Použití
$MimoZarucniOprava = $director->build($MimoZarucniOpravaBuilder );
```

4.1.4 Návrhový vzor Prototyp (Prototype)

Pointa

Návrhový vzor Prototyp spočívá ve způsobu vytváření objektů pomocí klonování již existujícího prototypu objektu.

Účel vzoru

Tento vzor umožní zamezit vzniku nadměrného počtu tříd. Místo toho aby měli všechny objekty svojí třídu, vytvářejí se z prototypu jedné třídy.

Vzor je oproti ostatním vzorům pro vytváření objektů mnohem flexibilnější pro pozdější úpravy [1].

Na rozdíl od návrhového vzoru Tovární metoda umožňuje vzor Prototyp i vkládání a odstraňování nových produktů za běhu programu tím, že lze různé prototypy u správce prototypů registrovat zavoláním příslušné metody [1].

Řešení problému

Pro řešení problému pomocí návrhového vzoru Prototyp je třeba vytvořit třídu správce prototypů (dle vzoru Tovární metoda), která pomocí metod klonování vytvoří nové instance podle prototypů, které jsou třídě předány.

Dále se musí vytvořit zmiňované prototypy a metody jejich předávání výše zmíněné třídě [1].

Možné problémy při použití vzoru

Pro správné fungování vzoru Prototyp je nutné,aby všechny třídy, podle kterých se vytvářejí prototypy bylo možné klonovat. Problém nastává v případě, že nějaká třída, která má sloužit jako prototyp obsahuje v atributech kromě klasických hodnot i reference na další objekty. V tomto případě je potřeba vytvořit v těchto třídách dodatečné metody *__clone()*, které obstarají vytvoření kopií objektů uložených v atributech.

Standardní implementace operátoru *clone* vytvoří jen kopii objektu a převezme všechny atributy. V případě, že se u atributů objektu jedná o další objekty, nebude z těchto objektů vytvořena žádná kopie. V takovém případě mluvíme o plytké kopii objektu [1]. Jazyk PHP umožňuje pomocí metody *__clone()* určit způsob, jakým má být daný objekt klonovaný. Tímto způsobem je možné vytvářet i hluboké kopie objektů. To znamená kopie všech objektů, na které v daném objektu odkazuje [1].

4.2 Strukturální vzory

V této kapitole se zaměřím na další skupinu návrhových vzorů, které mají své využití především při situacích kdy je třeba objekty (vytvořené pomocí návrhových vzorů vytváření) propojovat do vzájemně propojených struktur.

V kapitole popisuji následující návrhové vzory:

- Strom (Composite)
- Adaptér (Adapter)
- Most (Bridge)
- Dekorátor (Decorator)
- Zástupce (Proxy)
- Fasáda (Facade)
- Muší váha (Flyweight)

4.2.1 Návrhový vzor Strom (Composite)

Pointa

Základní myšlenka tohoto vzoru spořívá ve stromové struktuře, která umožňuje uspořádání jednotlivých objektů do větších celků, složených (neboli kompozitních) objektů. Pro přístup k jednotlivým i složeným objektům se využívá stejný způsob, což takovou strukturu zpřehledňuje a usnadňuje případné manipulace.

Účel vzoru

Tento návrhový vzor může být velmi užitečný, pokud aplikace využívá struktury vícero jednoduchých i složených objektů.

Složené objekty jsou objekty, které obsahují referenční odkazy na další objekty. Tyto další objekty mohou být opět složené, nebo jednoduché bez dalších referencí k jiným.

Využití návrhového vzoru spočívá v seskupení jednotlivých a složených objektů do stromové hierarchické struktury. Každá větev této struktury může být dále rozvětvena nebo může být konečná a obsahovat pouze jednotlivé objekty [21].

Řešení problému

Pro využití struktury popisované návrhovým vzorem Strom, je důležité vytvořit třídu, která slouží jako jednotné rozhraní pro přístup ke všem objektům této struktury. V této třídě jsou

využívány metody jako například *přidat()*, *odebrat()*, *ziskatPotomka()* nebo *nastavAtribut()*, které umožňují jednotnou manipulaci s objekty ve struktuře. Při využití tohoto návrhového vzoru je hlavní výhodou právě tento přístup ke všem objektům, který probíhá stejným způsobem. Uživatel se tedy dále nemusí zajímat o strukturu objektů, zdali jsou složené nebo jednoduché. V případě složených objektů je konkrétní metoda vykonána na všech větvích struktury [21].

Ukázkový kód (PHP)

V tomto ukázkovém případě je potřeba pracovat se strukturou objektů vinotéky, kde jsou uloženy krabice s několika lahvemi vína i jednotlivé lahve samostatně. Složené objekty Krabice i jednoduché objekty Lahve mají stejnou funkci pro vrácení názvu vína. Zatímco třída *Lahve* vrací hodnotu pouze jednoho konkrétního vína, třída *Krabice* vrací několik názvů všech lahví, které obsahuje. Stejně tak metoda pro přidání a odstranění lahve vína z krabice je stejná pro třídy *Lahve* i *Krabice*, ovšem u objektů třídy *Lahve* vrací hodnotu „*FALSE*“ a je tedy využitelná jen u objektů třídy *Krabice*.

Výpis 4: *Ukázková implementace návrhového vzoru Strom v jazyce PHP*.

```
<?php

abstract class Vinoteka {
    abstract function getNazevLahve($lahevPredavana);
    abstract function getPocetLahvi();
    abstract function setPocetLahvi($novyPocetLahvi);
    abstract function pridejLahev($lahev);
    abstract function odstranLahev($lahev);
}
class Lahev extends Vinoteka {
    private $nazev;
    function __construct($nazev) {
      $this->nazev = $nazev;
    }
    function getNazevLahve($lahevPredavana) {
      if (1 == $lahevPredavana) {
        return $this->nazev;
      } else {
        return FALSE;
      }
    }
    function getPocetLahvi() {
      return 1;
    }
    function setPocetLahvi($novyPocetLahvi) {
      return FALSE;
    }
    function pridejLahev($lahev) {
```

```
      return FALSE;
    }
    function odstranLahev($lahev) {
      return FALSE;
    }
}

class Krabice extends Vinoteka {
    private $lahve = array();
    private $pocetLahvi;
    public function __construct() {
      $this->setPocetLahvi(0);
    }
    public function getPocetLahvi() {
      return $this->pocetLahvi;
    }
    public function setPocetLahvi($novyPocetLahvi) {
      $this->pocetLahvi = $novyPocetLahvi;
    }
    public function getNazevLahve($lahevPredavana) {
      if ($lahevPredavana <= $this->pocetLahvi) {
        return $this->lahve[$lahevPredavana]->getNazevLahve(1);
      } else {
        return FALSE;
      }
    }
    public function pridejLahev($lahev) {
      $this->setPocetLahvi($this->getPocetLahvi() + 1);
      $this->lahve[$this->getPocetLahvi()] = $lahev;
      return $this->getPocetLahvi();
    }
    public function odstranLahev($lahev) {
      $pocet = 0;
      while (++$pocet <= $this->getPocetLahvi()) {
        if ($lahev->getNazevLahve(1) ==
          $this->lahve[$pocet]->getNazevLahve(1)) {
          for ($x = $pocet; $x < $this->getPocetLahvi(); $x++) {
            $this->lahve[$x] = $this->lahve[$x + 1];
          }
          $this->setPocetLahvi($this->getPocetLahvi() - 1);
        }
      }
      return $this->getPocetLahvi();
    }
}
?>
```

4.2.2 Návrhový vzor Adaptér (Adapter)

Pointa

Použití tohoto návrhového vzoru je vhodné, pokud je potřeba propojit dvě třídy s rozhraními neschopnými vzájemné spolupráce [1].

Účel vzoru

Návrhový vzor Adaptér najde využití především tam, kde je v aplikaci třeba používat jakékoli třídy které mají odlišné rozhraní, než to které používají ostatní třídy. Může se jednat například o třídy zastaralé, nebo třídy přejaté. Přepsání takových tříd do podoby vhodné pro architekturu dané aplikace může být velmi náročné a často je mimo reálné prostředky, nicméně jejich využití může být nezbytné [22]. Tyto třídy je mnohdy nutné využívat pouze dočasně, a také proto je namístě využít tento návrhový vzor, který je v takové situaci velmi užitečný.

Často bývá vzor využíván v případě, kdy aplikace pracuje s vícero databázemi. Pro metody připojení k databázi je vytvořeno rozhraní obsluhující konkrétní databázový nástroj a v ostatních místech aplikace je tak možné používat stále stejné, všeobecně platné metody jako například *insert (), update(),* či *delete()* [22].

Řešení problému

Pro vyřešení problému komunikace mezi nekompatibilními třídami (třídami neschopnými vzájemné spolupráce), se již existující třída se svým nevyhovujícím rozhraním zabalí do rozhraní nového. Toto nové rozhraní komunikaci převede do potřebné formy [1].

Uživatel aplikace tak může zavolat metody v tomto rozhraní a rozhraní požadavky přepošle připojované třídě v potřebné formě.

Pro vytvoření rozhraní (adaptéru) k určité nové (například převzaté) třídě je potřeba nejprve porovnat rozdíly mezi rozhraním této třídy a rozhraním navrhnutým pro náš systém.

Poté je třeba vytvořit novou třídu, která bude sloužit jako rozhraní pro připojení např. převzaté třídy, podle závěrů ze zmiňovaného porovnání. V této třídě *Adapter* (rozhraní) musí být implementovány všechny metody vyžadované rozhraním našeho systému a tyto metody musí být upravené pro přeposílání požadavků na původní metody připojované třídy [1].

V systému se dále využívá objekt adaptéru, kterým je připojovaný objekt obalen. V případě vyžádání údajů od připojované třídy aplikace volá metody rozhraní.

Možné problémy při použití vzoru

Při použití tohoto vzoru je v mnoha případech namístě očekávat chyby způsobené metodami adaptéru (rozhraní), které nemusí předpokládat všechny stavy připojované třídy. Vhodné je implementovat zaznamenávání chyb a provést testování nového adaptéru [1].

Vytvoření nového kompletního adaptéru pro určitou třídu může být velmi nákladné a je proto velice důležité důsledně provést počáteční porovnání rozhraní a například vybrat pouze využitelné metody, čímž se mohou ušetřit značné prostředky při tvorbě rozhraní.

Ukázkový kód (PHP)

V tomto jednoduchém ukázkovém případě disponuje restaurace aplikací, která slouží pro zobrazování informací o naskladněných lahvích vína. Tato aplikace očekává metodu *getNazevARok()*. Pro skladové potřeby je nově využíván systém, jehož třída *Lahev* místo původní metody *getNazevARok()* nabízí metody *getNazev()* a *getRok()*.

Pro třídu *Lahev* je vytvořen adaptér - třída *AdapterLahve*, která převezme instanci třídy *Lahev* a spojí metody *getNazev()* a *getRok()* v jedinou metodu *getNazevARok()*.

Výpis 5: *Ukázková implementace návrhového vzoru Adaptér v jazyce PHP*.

```
<?php

class Lahev {
    private $nazev;
    private $rok;
    function __construct($nazev, $rok) {
        $this->nazev = $nazev;
        $this->rok  = $rok;
    }
    function getNazev() {
        return $this->nazev;
    }
    function getRok() {
        return $this->rok;
    }
}
class AdapterLahve {
    private $lahev;
    function __construct(Lahev $lahev) {
        $this->lahev = $lahev;
    }
    function getNazevARok() {
        return $this->lahev->getNazev().' z roku  '.$this->lahev->getRok();
    }
}
?>
```

4.2.3 Návrhový vzor Most (Bridge)

Pointa

Tento návrhový vzor odděluje rozhraní a implementaci (zavádění metod) třídy a umožňuje tak jejich nezávislé změny [1].

Účel vzoru

Návrhový vzor Most se podobá návrhovému vzoru Adaptér. Vzor Adaptér se však používá v situaci kdy je potřeba vytvořit vhodné rozhraní mezi hotovými třídami, na rozdíl od vzoru Most, který své využití najde především již při návrhu aplikace. Návrhový vzor Most slouží pro oddělení abstrakce (rozhraní) od její konkrétní implementace, čímž pomáhá předcházet nadměrnému počtu tříd při předávání implementací. Využití oddělení abstrakce od implementace se hodí například při jakémkoli současném využití části kódu společného pro více míst aplikace a části kódu jedinečného.

Řešení problému

Pro oddělení rozhraní třídy od její implementace je nutné postupovat od vytvoření rozhraní pro vybrané implementace, které uchovávají konkrétní údaje. Dalším krokem je vytvoření nejméně dvou tříd, které implementují dříve vytvořené rozhraní a ukládají konkrétní údaje. Poté je ještě nutné vytvořit možnost předat instanci vytvořených tříd (abstrakci) exitující třídě [1].

V případě potřeby pracovat s konkrétními údaji z implementace se v existující třídě použije rozhraní vytvořené na začátku postupu.

Ukázkový kód (PHP)

Následující kód slouží k předvedení základní funkčnosti návrhového vzoru Most. Třída *BridgeLahev* využívá dvě rozdílně zpracované implementace, které jsou od této třídy odděleny a následně jsou umístěny do nových tříd s názvem *BridgeLahevStarsImplementace* a *BridgeLahevCapslockImplementace*. Třída *BridgeLahev* při každém vytváření instance přiřadí jednu ze zmiňovaných implementací.

Výpis 6: *Ukázková implementace návrhového vzoru Most v jazyce PHP.*

```
<?php

abstract class BridgeLahev {
    private $nazevBridgeLahev;
    private $rokBridgeLahev;
    private $implementaceBridgeLahev;
    function __construct($nazev, $rok, $vyberImplementace) {
```

```
      $this->nazevBridgeLahev = $nazev;
      $this->rokBridgeLahev  = $rok;
      if ('STARS' == $vyberImplementace) {
        $this->implementaceBridgeLahev = new BridgeLahevStarsImplementace();
      } else {
        $this->implementaceBridgeLahev = new BridgeLahevCapslockImplementace();
      }
    }
    function getNazev() {
      return $this->implementaceBridgeLahev->getNazev($this->nazevBridgeLahev);
    }
    function getRok() {
      return $this->implementaceBridgeLahev->getRok($this->rokBridgeLahev);
    }
}
class BridgeLahevNazevRok extends BridgeLahev {
    function getNazevRok() {
      return $this->getNazev() . " z roku: " . $this->getRok();
    }
}
class BridgeLahevRokNazev extends BridgeLahev {
    function getRokNazev() {
      return $this->getRok() . " - " . $this->getNazev();
    }
}
abstract class BridgeLahevImplementace {
    abstract function getNazev($nazev);
    abstract function getRok($rok);
}
class BridgeLahevCapslockImplementace extends BridgeLahevImplementace {
    function getNazev($nazev) {
      return strtoupper($nazev);
    }
    function getRok($rok) {
      return strtoupper($rok);
    }
}

class BridgeLahevStarsImplementace extends BridgeLahevImplementace {
    function getNazev($nazev) {
      return Str_replace(" ","*",$nazev);
    }
    function getRok($rok) {
      return Str_replace(" ","*",$rok);
    }
}
?>
```

4.2.4 Návrhový vzor Dekorátor (Decorator)

Pointa

Návrhový vzor Dekorátor slouží pro přidávání funkcionalit již existujícím třídám, které nechceme z různých důvodů upravovat. Tyto nové funkce či změny stávajících metod lze provádět za běhu programu [1].

Účel vzoru

V případě využití tohoto vzoru jsou přístupny mnohem větší možnosti různých (průběžných) změn v aplikaci než je tomu u dědění. Dekorátory, jak jsou nazývány třídy, které obdržené instanci jiné třídy přidají novou funkcionalitu nebo provedou úpravu stávajících funkcionalit, jsou totiž schopny provádět změny i za běhu programu. Metody, které najdou své využití jen střídmě, mohou být obstarávány pomocí dekorátorů a hlavní třídy tak mohou zůstat přehlednější.

Řešení problému

Pro užití struktury tohoto návrhového vzoru je nejdříve potřeba vytvořit novou třídu pro dekorátory, která bude stejného typu jako upravovaný (dekorovaný) objekt. V této třídě je definován konstruktor, kterému bude předáván upravovaný objekt.

Dalším krokem je v této třídě vytvořit všechny potřebné (nové i staré) metody a jejich volání přesměrovat na upravovaný objekt. Po tomto postupu už je možné přepsáním vhodných metod vytvořit konkrétní dekorátory jako potomky dříve vytvořené obecné třídy pro dekorátory [1].

Možné problémy při použití vzoru

Použití dekorátorů musí být pečlivě promyšleno. Zatímco správné použití může pomoci zpřehlednit kód aplikace, nevhodné používání velkého množství dekorátorů může naopak zapříčinit komplikovaný kód, kde není na první pohled jasná úplná celková funkce některých tříd.

Ukázkový kód (PHP)

Ukázka užití vzoru v jazyce PHP využívá (upravovanou) třídu Lahev, která poskytuje metody pro zjištění názvu a roku uskladněné lahve vína. V případě kdy je potřeba přidat funkcionalitu která umožní zobrazovaný název lahve vína zvýraznit pomocí vykřičníků, ale není vhodné měnit původní třídu, je vytvořena nová základní třída pro dekorátory *LahevNazevDecorator* a také potomek této třídy dekorátorů - konkrétní dekorátor (třída)

ZvyrazneniNazevuDecorator, kde byla vytvořena metoda sloužící pro zvýraznění názvu *zvyrazniNazev()*.

Místo metody *getNazev()* třídy *Lahev* bude využíváno metody *getNazev()*, kterou obsahuje třída *LahevNazevDecorator*.

Výpis 7: *Ukázková implementace návrhového vzoru Dekorátor v jazyce PHP*.

```
<?php

class Lahev {
    private $nazev;
    private $rok;
    function __construct($nazev, $rok) {
        $this->nazev = $nazev;
        $this->rok  = $rok;
    }
    function getNazev() {
        return $this->nazev;
    }
    function getRok() {
        return $this->rok;
    }
}

class LahevNazevDecorator {
    protected $lahev;
    protected $nazev;
    public function __construct(Lahev $lahev) {
        $this->lahev = $lahev;
$this->nazev = $this->lahev->getNazev();
    }

    function getNazev() {
        return $this->nazev;
    }
}

class ZvyrazneniNazevuDecorator extends LahevNazevDecorator {
    private $dekorator;
    public function __construct(LahevNazevDecorator $dekorator) {
        $this->dekorator = $dekorator;
    }
    function zvyrazniNazev() {
        $this->dekorator->nazev = "!" . $this->dekorator->nazev . "!";
    }
}
?>
```

4.2.5 Návrhový vzor Zástupce (Proxy)

Pointa

Návrhový vzor Zástupce slouží především k řízení přístupů k původnímu objektu, pomocí zástupného objektu používaného místo objektu původního.

Účel vzoru

Anglický název návrhového vzoru Proxy lze v tomto kontextu přeložit jako zástupce nebo náhradník. Návrhový vzor je velmi podobný návrhovému vzoru Dekorátor, ale na rozdíl od něho neslouží k přidávání dalších funkcí původnímu objektu, ale k řízení přístupu k původnímu objektu.

Původní (tedy zastupovaný) objekt je nahrazen jiným objektem, který slouží jako zástupce původního. Zástupce je pro ostatní části systému stejný jako původní objekt, ale může řídit a upravovat veškerou komunikaci.

Využití tento vzor najde v řadě případů, přičemž rozlišujeme tyto čtyři obecné typy využití vzoru [23]:

- Virtuální zástupce (Virtual Proxy): Slouží pro zastoupení objektů náročných na paměť. Takový zástupce vytvoří původní objekt pouze v případě, když je to nezbytně nutné a to s pouze nezbytnými daty.
- Vzdálený zástupce (Remote Proxy): Je využíván pro zastupování vzdálených objektů, kdy zástupný objekt obstarává síťovou komunikaci a zbytek aplikace s ním může komunikovat jako by to byl přímo vzdálený objekt.
- Ochranný zástupce (Protection Proxy): V tomto případě zástupný objekt chrání objekt původní pomocí řízení přístupů (ověřování oprávnění uživatelů) a sledování či zaznamenávání přístupů.
- Chytrý zástupce (Smart Proxy): Najde využití, pokud je vhodné doplnit dodatečné operace o funkce jako je ukončení připojení při překročení limitu připojení, počítání odkazů na určitý objekt, či další funkce vhodné pro vylepšení výkonu aplikace.

Řešení problému

Princip implementace rozhraní původního (zastupovaného) objektu platí i v tomto případě.

Nejdříve je potřeba vytvořit novou třídu *Zástupce*, která využívá veškerá rozhraní původní třídy. Ve všech metodách je volání přesměrováno na původní objekt, který je vykoná.

Nakonec je vytvořen potomek třídy *Zástupce* a v potřebných metodách přidat kontrolní funkcionality [1].

Jak již bylo řečeno, vzor se velmi podobá vzoru Dekorátor, který rozšiřuje objekty o nové metody. Vzor Zástupce se oproti tomu využívá jako zástupce objektu bez dalších metod. Tyto dva vzory se liší ve způsobu užití [1].

Ukázkový kód (PHP)

V tomto ukázkovém případě je návrhový vzor Zástupce předveden ve formě Virtuálního zástupce (Virtual Proxy). Byla vytvořena třída *SeznamLahvi*, pro která má svého zástupce, třída *ProxySeznamLahvi*, který slouží pro zastoupení původní třídy *SeznamLahvi*, která by mohla být náročná na paměť. Objekt původní třídy tak může být vytvořen pouze v nezbytných případech.

Výpis 8: *Ukázková implementace návrhového vzoru Zástupce v jazyce PHP.*

```
<?php
class ProxySeznamLahvi {
    private $seznamLahvi = NULL;
    //SeznamLahvi není vytvořen hned
    function __construct() {
    }
    function getPocetLahvi() {
        if (NULL == $this->seznamLahvi) {
            $this->makeSeznamLahvi();
        }
        return $this->seznamLahvi->getpocetLahvi();
    }
    function pridejLahev($lahev) {
        if (NULL == $this->seznamLahvi) {
            $this->makeSeznamLahvi();
        }
        return $this->seznamLahvi->pridejLahev($lahev);
    }
    function getLahev($lahevHledana) {
        if (NULL == $this->seznamLahvi) {
            $this->makeSeznamLahvi();
        }
        return $this->seznamLahvi->getLahev($lahevHledana);
    }
    function odstranLahev($lahev) {
        if (NULL == $this->seznamLahvi) {
            $this->makeSeznamLahvi();
        }
        return $this->seznamLahvi->odstranLahev($lahev);
    }
    //make metoda pro vytvoření instance
```

```
    function makeSeznamLahvi() {
        $this->seznamLahvi = new seznamLahvi();
    }
}

class SeznamLahvi {
    private $lahve = array();
    private $pocetLahvi = 0;
    public function __construct() {
    }
    public function getPocetLahvi() {
      return $this->pocetLahvi;
    }
    public function setPocetLahvi($novyPocetLahvi) {
      $this->pocetLahvi = $novyPocetLahvi;
    }
    public function getLahev($lahevHledana) {
       if ( (is_numeric($lahevHledana)) && ($lahevHledana <= $this-
>getPocetLahvi())) {
            return $this->lahve[$lahevHledana];
        } else {
           return NULL;
        }
    }
    public function pridejLahev(Lahev $lahev) {
      $this->setPocetLahvi($this->getPocetLahvi() + 1);
      $this->lahve[$this->getPocetLahvi()] = $lahev;
      return $this->getPocetLahvi();
    }
    public function odstranLahev(Lahev $lahev) {
      $pocet = 0;
      while (++$pocet <= $this->getPocetLahvi()) {
        if ($lahev->getNazev() ==
          $this->lahve[$pocet]->getNazev()) {
          for ($x = $pocet; $x < $this->getPocetLahvi(); $x++) {
            $this->lahve[$x] = $this->lahve[$x + 1];
          }
          $this->setPocetLahvi($this->getPocetLahvi() - 1);
        }
      }
      return $this->getPocetLahvi();
    }
}

class Lahev {
    private $nazev;
    private $rok;
    function __construct($nazev, $rok) {
```

```
        $this->nazev = $nazev;
        $this->rok  = $rok;
    }
    function getNazev() {
        return $this->nazev;
    }
    function getRok() {
        return $this->rok;
    }
}
?>
```

4.2.6 Návrhový vzor Fasáda (Facade)

Pointa

Základní myšlenkou tohoto vzoru je poskytnout zjednodušené rozhraní (fasádu) k určité logické skupině rozhraním systému a umožnit tak zjednodušený přístup k částem aplikace bez nutnosti znát kompletní funkci systému za fasádou.

Účel vzoru

Návrhový vzor Fasáda je vhodné použít v případě, kdy Vaše aplikace obsahuje stále rostoucí počet tříd a stává se stále náročnější vysvětlovat podrobně novým kolegům, jak spolu třídy souvisí. Vytvořením rozhraní (fasády) je možné tento problém chytře vyřešit. Vnitřní provázanost tříd nějakého menšího celku aplikace je tak možné odstínit a všichni vývojáři tak mohou používat funkce těchto tříd, aniž by je museli dopodrobna znát. Stačí se seznámit s rozhraním.

Časté využití najdeme při připojování k databázi. Funkce využívající jazyka SQL jsou skryty za fasádou s příslušnými metodami, což klientům zjednodušuje práci s databází a správci databázového nástroje to tak umožňuje i přechod na nový databázový systém bez dalších úprav mimo fasádu. Další výhody využití vzoru se mohou projevit centralizací některých úloh, snížení počtu potřebných tříd může pomoci odstranit duplicitu kódu.

Tento návrhový vzor se částečně shoduje se vzorem Adaptér. Na rozdíl od něj však rozhraní fasáda neumožňuje pouze předávání informací mezi rozhraními, ale komunikaci s určitou menší částí aplikace zjednodušuje.

Řešení problému

Pro nasazení tohoto návrhového vzoru pro určitou část aplikace je důležité nejdříve identifikovat všechny třídy a objekty, které by měli být skryté za rozhraním (fasádou) a také identifikovat všechny operace, které musí fasáda podporovat.

V dalším kroku se musí vytvořit tyto identifikované třídy fasády a naplnit je metodami, pomocí kterých bude fasáda komunikovat se schovanými částmi aplikace. Nakonec je třeba vytvořit metody, které využijí klienti pro zjednodušený přístup k funkčnosti původního systému [1].

Ukázkový kód (PHP)

Tento velmi jednoduchý ukázkový příklad využívá rozhraní *LahevFacade* pro přejmenování nepřehledně pojmenovaných metod ve třídě *Lahev*.

Výpis 9: *Ukázková implementace návrhového vzoru Fasáda v jazyce PHP.*

```
<?php
class LahevFacade
{
    private $lahev;

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

    public function setNazev($nazev)
    {
        $this->lahev->setNazevZobrazovaneLahve($nazev);
        return $this;
    }

    public function setRok($rok)
    {
        $this->lahev->setRokZobrazovaneLahve($rok);
        return $this;
    }

    public function setTyp($typ)
    {
        $this->lahev->setTypNapojeZobrazovaneLahve($typ);
        return $this;
    }

    public function getNazev()
    {
```

```
        $this->lahev->getNazevZobrazovaneLahve();
    return $this;
    }

    public function getRok()
    {
        $this->lahev->getRokZobrazovaneLahve();
    return $this;
    }

    public function getTyp()
    {
        $this->lahev->getTypNapojeZobrazovaneLahve();
    return $this;
    }
}

class Lahev {
    public $nazev;
    public $rok;
    public $typNapoje;

    public function getNazevZobrazovaneLahve() {
        return $this->nazev;
    }
    public function getRokZobrazovaneLahve() {
        return $this->rok;
    }
    public function getTypNapojeZobrazovaneLahve() {
        return $this->typNapoje;
    }
    public function setNazevZobrazovaneLahve($nazev) {
           $this->nazev = $nazev;
    }
    public function setRokZobrazovaneLahve($rok) {
           $this->rok = $rok;
    }
    public function setTypNapojeZobrazovaneLahve($typ) {
           $this->typNapoje = $typ;
    }
}
?>
```

4.2.7 Návrhový vzor Muší váha (Flyweight)

Pointa

Tento návrhový vzor se využívá pro zajištění úspory další paměti při využívání velkého počtu téměř shodných objektů.

Účel vzoru

Hlavním principem každého návrhového vzoru Muší váha, je rozdělení třídy na část vnitřního stavu a část vnějšího stavu. Pokud je vnitřní stav stejný pro mnoho instancí, není nutné, aby si ho všechny instance ukládaly, ale stačí, když budou odkazovat na (sdílený) vnitřní stav a ukládat mohou jenom jedinečná data, tedy vnější stav. Identické instance pak nemusí být vytvářeny opakovaně, ale stačí předávat první vytvořenou instanci [1].

Použití tohoto vzoru tedy umožňuje udržet nároky na paměť při vytváření mnoha instancí co nejníže.

Řešení problému

Po zjištění potřeby třídy, která vytváří velký počet podobných instancí, je vhodné třídu oprostit od objektů, které uchovávají společné stavy pro mnoho objektů. Tyto stavy je vhodné uložit mimo instance do nové třídy, která slouží jako zástupce původní třídy [1].

Ukázkový kód (PHP)

Tato jednoduchá ukázka využití návrhového vzoru Muší váha je založena na třídě *FlyweightLahev* a třídě *FlyweightKrabice*, přičemž však využívá i třídu *FlyweightFactory* (napsanou podle návrhového vzoru Tovární metoda), která se stará o vytvoření instancí třídy *FlyweightLahev* až v momentě kdy jsou potřeba.

Restaurace, která využívá aplikaci pro zobrazování naskladněného vína nakupuje pět odrůd vína a proto je nabízeno pět metod pro vytvoření těchto lahví. V případě, kdy bude vytvořena již jednou vytvořená láhev, místo další instance bude předán pouze odkaz na tu první, která obsahuje stejné informace.

Výpis 10: *Ukázková implementace návrhového vzoru Muší váha v jazyce PHP.*

```
<?php
class FlyweightLahev {
    public $nazev;
    public $rok;
    public $typNapoje;

    function __construct($nazev, $rok, $typ) {
        $this->nazev = $nazev;
```

```
        $this->rok  = $rok;
    $this->typNapoje  = $typ;
    }

    function getNazev()
    {
    return $this->nazev;
    }
    function getRok()
    {
       return $this->rok;
    }

    public function getTyp()
    {
        return $this->typNapoje;
    }

}

class FlyweightFactory {
    private $lahve = array();
    function __construct() {
        $this->lahve[1] = NULL;
        $this->lahve[2] = NULL;
        $this->lahve[3] = NULL;
        $this->lahve[4] = NULL;
        $this->lahve[5] = NULL;
    }
    function getLahev($lahev) {
        if (NULL == $this->lahve[$lahev]) {
            $makeFunkce = 'makeLahev'.$lahev;
            $this->lahve[$lahev] = $this->$makeFunkce();
}
        return $this->lahve[$lahev];
    }
    function makeLahev1() {
        $lahev = new FlyweightLahev('Sauvignon','2014','polosuché');
        return $lahev;
    }
    function makeLahev2() {
        $lahev = new FlyweightLahev('Dornfelder','2013','suché');
        return $lahev;
    }
    function makeLahev3() {
        $lahev = new FlyweightLahev('Chardonnay','2012','polosladké');
        return $lahev;
    }
```

```
    function makeLahev4() {
        $lahev = new FlyweightLahev('Veltlínské zelené','2013','suché');
        return $lahev;
    }
    function makeLahev5() {
        $lahev = new FlyweightLahev('Château Petrus','1994','polosuché');
        return $lahev;
    }
}

class FlyweightKrabice {
    private $lahve = array();
    function pridejLahev($lahev) {
        $this->lahve[] = $lahev;
    }
    function zobrazLahve() {
        $nazvyLahvi = NULL;
        foreach ($this->lahve as $lahev) {
            $nazvyLahvi .= 'Název odrůdy: "'.$lahev->getNazev();"
 }
        return $nazvyLahvi;
    }
}
?>
```

4.3 Vzory chování

Tato skupina návrhových vzorů popisuje především řešení situací, kdy je třeba efektivně řešit spolupráci objektů.

V kapitole popisuji tyto návrhové vzory:

- Pozorovatel (Observer)
- Prostředník (Mediator)
- Šablonová metoda (Template Method)
- Příkaz (Command)
- Návštěvník (Visitor)
- Iterátor (Iterator)
- Stav (State)
- Strategie (Strategy)
- Zřetězení zodpovědnosti (Chain of Responsibility)

4.3.1 Návrhový vzor Pozorovatel (Observer)

Pointa

Pomocí návrhového vzoru Pozorovatel lze vytvořit mezi subjektem (pozorovaným objektem) a libovolným počtem pozorovatelů závislost typu 1:n. Všichni pozorovatelé jsou automaticky informováni o změně stavu subjektu [1].

Účel vzoru

Vzor se využívá v případech, kdy je třeba konkrétní objekt informovat o změně stavu jiného objektu. Při návrhu větší aplikace je velmi náročné pracovat se závislostmi. Objekty často souvisejí s mnoha dalšími a v případě vzájemné nutnosti upozornění na změny jejich stavů není vhodné tyto funkce vkládat přímo do těchto objektů. Díky návrhovému vzoru Pozorovatel lze funkce vzájemných upozornění vytvářet mimo sledované objekty, což pomůže přehlednosti kódu. Výhodou tohoto přístupu je možnost měnit funkčnost za běhu programu.

Konkrétním příkladem může být aukční aplikace, která vzor využívá pro upozorňování objektu dražitele, který pozoruje změny stavu objektů nabízejících. Nabízející mohou „zvednout“ očíslovanou kartu, přijmout tak dražitelovu cenu, čímž změní svůj stav [24].

Řešení problému

Vytvořit strukturu podle vzoru Pozorovatel lze provést následujícím postupem:

Ve většině případů je potřeba začít s vytvořením rozhraní pozorovatele (*Observer*), které bude obsahovat metodu *update()*, určenou pro sledování určeného parametru. V případě změny stavu pozorovaného objektu tato funkce informuje o změně objekt pozorovatele. Metoda *update()* očekává jako parametr instanci pozorované třídy (celý objekt), což umožní jednomu pozorovateli pozorovat více instancí a přináší obecně větší flexibilitu (například v možnosti sledovat jakékoli parametry instance).

Některé postupy uvádějí pro metodu *update()* jen určitý sledovaný parametr, což však není ve většině případů výhodné. Tyto implementace nabízejí jistotu typu předávaných objektů, ale nejsou výhodné především pro další úpravy a možnost sledovat různé subjekty.

Dalším krokem je vytvoření rozhraní pozorovaného objektu (*Observable*), které obsahuje metody pro správu pozorovatelů (přidání a odstranění) a také metodu *notify()* pro upozornění pozorovatele. Toto rozhraní *Observable* je třeba použít v pozorovaném objektu a do všech metod které ovlivňují stav objektu vložit volání metody *notify()*.

Konkrétní pozorovatelé s (využitím rozhraní *Observer*) jsou vytvořeni až nakonec.

Knihovna SPL od verze 5.1 jazyka PHP nabízí předpřipravené rozhraní, které může uživatelům ulehčit od tvorby vlastních rozhraní potřebných pro využívání návrhového vzoru Pozorovatel [1].

Knihovna SPL rozhraní *Observer* nazývá *SplObserver* a rozhraní *Observable* nazývá *SplSubject*. Tyto názvy byly zvoleny z důvodu možných konfliktů s rozhraními vytvořenými uživatelem. Více informací o knihovně SPL se čtenář může dočíst přímo v manuálu jazyka PHP [28].

Ukázkový kód (PHP)

Jednoduchý příklad využití návrhového vzoru Pozorovatel pracuje se třídou *PozorovanyObjekt*, která implementuje rozhraní *Observable*.

Pro pozorovatele je zde použita třída *Pozorovatel*, implementující rozhraní *Observer*.

Na konci příkladu jsou vytvořeny modelové objekty, přidán pozorovatel a po změně parametru *OblibeneVzory* bude výstupem programu, díky funkci *update()* výpis:

„Pozor! Změna oblíbených vzorů: Bridge, Proxy, Observer.“.

Výpis 11: *Ukázková implementace návrhového vzoru Pozorovatel v jazyce PHP.*

```
<?php

Interface Observer {
    public function update(Observable $observable);
}

Interface Observable {
    public function pridatObserver(Observer $observer);
    public function odebratObserver(Observer $observer);
    public function notify();
}

function writeln($upozorneni) {
    echo $upozorneni."<br/>";
}

class Pozorovatel implements Observer {
    public function __construct() {
    }
    public function update(Observable $observable) {
      writeln('Pozor! změna oblíbených vzorů: '.$subject->getOblibeneVzory());
    }
}

class PozorovanyObjekt implements Observable {
    private $oblibeneVzory = NULL;
    private $observers = array();
    function __construct() {
    }
    function pridatObserver(Observer $observer) {

      $this->observers[] = $observer;
    }
    function odebratObserver(Observer $observer) {

      foreach($this->observers as $okey => $ovalue) {
        if ($ovalue == $observer) {
          unset($this->observers[$okey]);
        }
      }
    }
    function notify() {
      foreach($this->observers as $obs) {
        $obs->update($this);
      }
```

```
    }
    function updateOblibeneVzory($upraveneOblibeneVzory) {
      $this->oblibeneVzory = $upraveneOblibeneVzory;
      $this->notify();
    }
    function getOblibeneVzory() {
      return $this->oblibeneVzory;
    }
}

//modelové objekty pro testování
$objekt = new PozorovanyObjekt();
$pozorovatel1 = new Pozorovatel();
$objekt->pridatObserver($pozorovatel1);
$objekt->updateOblibeneVzory('Bridge, Proxy, Observer');

?>
```

4.3.2 Návrhový vzor Prostředník (Mediator)

Pointa

Základní myšlenkou vzoru je způsob interakce (spolupráce) objektů pomocí objektu prostředníka, který mění vazby objektů z typů m:n na 1:n [1]. Struktura odstraňuje vzájemné (přímé) vazby mezi objekty a zároveň umožňuje komunikaci objektů, které nemají přímé komunikační vazby [1].

Účel vzoru

Návrhový vzor Prostředník určuje efektivní způsob komunikace většího počtu objektů. Místo vzájemné (přímé) komunikace objekty využívají komunikaci skrz objekt prostředníka. Vzor bývá často demonstrován (předváděn) na příkladu kontrolní věže na letišti, kdy posádky letadel v dosahu kontrolní věže místo přímé komunikace komunikují s kontrolní věží, která zapouzdří všechna spojení a sama dále rozhoduje, komu a jak zprávu předá [29]. Zprávu může například předat i pracovníkům na přistávací dráze, se kterými nemá posádka letadla přímé spojení.

Řešení problému

Způsob vytvoření vhodné struktury začíná u vytvoření rozhraní prostředníka. Dále jsou vytvořeny rozhraní ostatních komunikujících objektů s metodami, které bude volat prostředník. Dle rozhraní vytvořeného na počátku je třeba vytvořit konkrétního prostředníka s funkcionalitou zabezpečující všechna spojení. V případech s rozsáhlou funkcionalitou prostředníka lze využít i další pomocné třídy prostředníka.

Na závěr jsou dle připravených rozhraní upraveny nebo vytvořeny konkrétní objekty, které využívají ke komunikaci prostředníka [1].

Ukázkový kód (PHP)

V tomto ukázkovém kódu je jako prostředník použita třída *LahevProstrednik*, která převezme upozornění od tříd *LahevNazev* a *LahevNazevPuvodu*, pokud některá z těchto dvou tříd změní svůj text na zobrazení velkými nebo naopak malými písmeny. Pokud změna velikosti textu proběhla jen u jedné ze zmiňovaných tříd, metoda *change()* zajistí stejnou změnu i u druhé třídy.

Výpis 12: *Ukázková implementace návrhového vzoru Prostředník v jazyce PHP.*

```
<?php
class LahevProstrednik {
    private $nazev;
    private $zemePuvodu;
    function __construct($nazev, $rok) {
      $this->nazev = new LahevNazev($nazev,$this);
      $this->zemePuvodu  = new LahevZemePuvodu($zemePuvodu,$this); }
    function getNazev() {return $this->nazev;}
    function getZemePuvodu() {return $this->zemePuvodu;}
/*
*pokud je u nazvu nebo zemePuvodu provedena zmenena velikost pismen, tato funkce
zajisti jeji zmenu i u ostatních
*/
    function zmena(Lahev $menenaTrida) {
      if ($menenaTrida instanceof LahevNazev) {
        if ('upper' == $menenaTrida->getStav()) {
          if ('upper' != $this->getZemePuvodu()->getStav()) {
            $this->getZemePuvodu()->nastavZemePuvoduUpperCase();
          } } elseif ('lower' == $menenaTrida->getStav()) {
          if ('lower' != $this->getZemePuvodu()->getStav()) {
            $this->getZemePuvodu()->nastavZemePuvoduLowerCase();
          }}
      } elseif ($menenaTrida instanceof LahevZemePuvodu) {
        if ('upper' == $menenaTrida->getStav()) {
          if ('upper' != $this->getNazev()->getStav()) {
            $this->getNazev()->nastavNazevUpperCase();
          }
      } elseif ('lower' == $menenaTrida->getState()) {
          if ('lower' != $this->getNazev()->getStav()) {
            $this->getNazev()->nastavNazevLowerCase();
          }}  } }}
abstract class Lahev {
    private $prostrednik;
    function __construct($prostrednik) {
        $this->prostrednik = $prostrednik;
```

```
    }
    function getProstrednik() {return $this->prostrednik;}
    function zmeneno($menenaTrida) {
        getProstrednik()->zemePuvoduChanged($menenaTrida);
    }}
class LahevNazev extends Lahev {
    private $nazev;
    private $stav;
    function __construct($nazev, $prostrednik) {
      $this->nazev = $nazev;
      parent::__construct($prostrednik);
    }
    function getNazev() {return $this->nazev;}
    function setNazev($nazev) {$this->nazev = $nazev;}
    function getStav() {return $this->stav;}
    function setStav($stav) {$this->stav = $stav;}
    function nastavNazevUpperCase() {
      $this->setNazev(strtoupper($this->getNazev()));
      $this->setStav('upper');
      $this->getProstrednik()->zmena($this);
    }
    function nastavNazevLowerCase() {
      $this->setNazev(strtolower($this->getNazev()));
      $this->setStav('lower');
      $this->getProstrednik()->zmena($this);
    }}
class LahevZemePuvodu extends Lahev {
    private $zemePuvodu;
    private $stav;
    function __construct($zemePuvodu, $prostrednik) {
        $this->zemePuvodu = $zemePuvodu;
        parent::__construct($prostrednik);}
    function getZemePuvodu() {return $this->zemePuvodu;}
    function setZemePuvodu($zemePuvodu) {$this->zemePuvodu = $zemePuvodu;}
    function getStav() {return $this->stav;}
    function setStav($stav) {$this->stav = $stav;}
    function nastavZemePuvoduUpperCase() {
        $this->setZemePuvodu(strtoupper($this->getZemePuvodu()));
        $this->setStav('upper');
        $this->getProstrednik()->zmena($this);}
    function nastavZemePuvoduLowerCase() {
        $this->setZemePuvodu(strtolower($this->getZemePuvodu()));
        $this->setStav('lower');
        $this->getProstrednik()->zmena($this);
    }}
?>
```

4.3.3 Návrhový vzor Šablonová metoda (Template Method)

Pointa

Návrhový vzor Šablonová metoda vytváří základní rozhraní algoritmů pomocí konkrétní metody. Rozhraní algoritmu rozděleného v jednotlivých krocích, které poté implementují potomci. Tito potomci slouží jako zaměnitelné části algoritmu.

Účel vzoru

Většina algoritmů se skládá z několika postupně jdoucích kroků, které se někdy dají provádět více způsoby. Použití rozdílných způsobů může ovlivnit řadu skutečností.

Například může mít pozitivní či negativní dopady na výkon aplikace případně i na výsledky algoritmu. Pomocí návrhového vzoru Šablonová metoda je možné vytvořit strukturu snadno zaměnitelných, jednotlivých kroků algoritmu s pevně určeným pořadím [1]. Vytvoření takové struktury je poměrně nenáročné a značně zvýší flexibilitu kódu.

Řešení problému

V abstraktní základní třídě je třeba vytvořit minimálně jednu abstraktní metodu. Potřeba je také vytvořit metodu pro algoritmus. Tato metoda musí být deklarována jako *final* a volat minimálně dvě další metody, přičemž alespoň jedna metoda musí být abstraktní.

Poté už je třeba jenom vytvořit potomky základní třídy a vytvořit v nich abstraktní metody [1].

Ukázkový kód (PHP)

V tomto jednoduchém ukázkovém případě je třída *SablonaAbstraktni* základní třídou, která obsahuje abstraktní metodu *getLahevInfo*.

Tato abstraktní metoda volá funkci *zpracujNazev()*, která musí být překryta a funkci *zpracujRok()*, která překryta být nemusí.

Výpis 13: *Ukázková implementace návrhového vzoru Šablonová metoda v jazyce PHP.*

```
<?php
abstract class SablonaAbstraktni {
    public final function LahevInfo($lahev) {
        $nazev = $lahev->getNazev();
        $rok = $lahev->getRok();
        $zpracovanyNazev = $this->zpracujNazev($nazev);
        $zpracovanyRok = $this->zpracujRok($rok);
        if (NULL == $zpracovanyNazev) {
            $zpracovaneInfo = $zpracovanyNazev;
        } else {
            $zpracovaneInfo = $zpracovanyNazev.' by '.$zpracovanyRok;
```

```
        }
        return $zpracovaneInfo;
    }
    //Tato funkce musí být překryta (overridden)
    abstract function zpracujNazev($nazev);
        //Tato funkce může být překryta (overridden)
    function zpracujRok($rok) {
        return NULL;
    }
}
class SablonaKonkretniMezeryRimsky extends SablonaAbstraktni {
    function zpracujNazev($nazev) {
      return Str_replace(' ',' ',$nazev);
    }
    function zpracujRok($rok) {
      return Str_replace('1','I',$rok);
    }
}
class SablonaKonkretniPomlcky extends SablonaAbstraktni {
    function zpracujNazev($nazev) {
        return Str_replace(' ','-',$nazev);
    }
}
class Lahev {
    private $nazev;
    private $rok;
    function __construct($nazev, $rok) {
        $this->nazev = $nazev;
        $this->rok  = $rok;
    }
    function getNazev() {return $this->nazev;}
    function getRok() {return $this->rok;}
    }
}
?>
```

4.3.4 Návrhový vzor Příkaz (Command)

Pointa

Návrhový vzor Příkaz slouží k zapouzdření vykonání příkazů do podoby objektů. Díky tomuto zapouzdření je možné přidávat tyto požadavky do fronty požadavků jiného objektu a využívat operace prováděné zpětně.

Účel vzoru

Tento návrhový vzor nabízí elegantní způsob, jak převést vykonávání příkazů na samostatné objekty, což sebou nese řadu výhod. Přizpůsobit aplikaci změnám je tak mnohem jednodušší. Protože objekt který vykonává požadavek nemusí vědět, jakým způsobem se požadavek provede. Způsob provedení požadavku je zapouzdřen v jiném objektu. Samostatné objekty jednotlivých příkazů lze snadno rozšiřovat a upravovat.

Často je tento vzor využíván pro návrh *undo* (zpětných) operací, které vrací provedené změny do původního stavu [1]. Dobrým demonstrativním příkladem jsou instalační programy.

Společné využití se vzorem Strom přináší možnost vytvářet makra mnoha typů řízení posloupností příkazů.

Řešení problému

Pro zapouzdření jednotlivých příkazů (požadavků) je nutné pro tyto příkazy vytvořit jednotlivá rozhraní. Jednotlivé příkazy je poté třeba vytvořit s využitím vytvořených rozhraní.

Poté je možné vytvořit funkčnost klienta, umožňující přijímat, řadit a postupně vykonávat dříve vytvořené příkazy.

Ukázkový kód (PHP)

V této ukázkové situaci je objekt třída *LahevHvezdyPrikaz* vytvořen při vzniku instance třídy *Lahev*. Objekt *LahevHvezdyPrikaz* využívá metodu *setHvezdy()* objektu Lahev vždy, když je použita metoda *proved(),*

Výpis 14: *Ukázková implementace návrhového vzoru Příkaz v jazyce PHP*.

```
<?php

class Lahev {
    private $nazev;
    private $zemePuvodu;
    function __construct($nazev, $zemePuvodu) {
        $this->setNazev($nazev);
        $this->setZemePuvodu($zemePuvodu);
```

```
    }
    function getNazev() {
        return $this->nazev;
    }
    function setNazev($nazev) {
        $this->nazev = $nazev;
    }
    function getZemePuvodu() {
        return $this->zemePuvodu;
    }
    function setZemePuvodu($zemePuvodu) {
        $this->zemePuvodu = $zemePuvodu;
    }
    function setHvezdy() {
        $this->setNazev(Str_replace(' ','*',$this->getNazev()));
        $this->setZemePuvodu(Str_replace(' ','*',$this->getZemePuvodu()));
    }
    function setBezHvezd() {
        $this->setNazev(Str_replace('*',' ',$this->getNazev()));
        $this->setZemePuvodu(Str_replace('*',' ',$this->getZemePuvodu()));
    }
    function getNazevAZemePuvodu() {
        return $this->getNazev().' z '.$this->getZemePuvodu();
    }
}

abstract class LahevPrikaz {
    protected $lahev;
    function __construct($lahev) {
        $this->lahev = $lahev;
    }
    abstract function proved();
}

class LahevHvezdyPrikaz extends LahevPrikaz {
    function proved() {
        $this->Lahev->setHvezdy();
    }
}

class LahevBezHvezdPrikaz extends LahevPrikaz {
    function proved() {
        $this->Lahev->setBezHvezd();
    }
}
?>
```

4.3.5 Návrhový vzor Návštěvník (Visitor)

Pointa

Tento návrhový vzor umožňuje přidávání nových operací do struktur objektů, jejichž struktura nelze měnit. Nové operace jsou zapouzdřeny do samostatné třídy.

Účel vzoru

Návrhový vzor Návštěvník pomáhá v situacích, kdy by změna funkčnosti určitých tříd mohla způsobit nestabilitu aplikace nebo jejich částí. Nové operace jsou proto zapouzdřeny do samostatné třídy. Pomocí vzoru Návštěvník lze přidávat i takové nové funkce, které by bez samostatné třídy nebylo možné provádět, kvůli nepřístupným údajům zvenčí. Je třeba si však uvědomit že použití návštěvníka vyžaduje veřejné metody, které mu zpřístupní potřebné parametry, čímž se poruší zapouzdření původních tříd.

Řešení problému

Vytvoření popisované struktury musí začínat analýzou typů prvků ve struktuře, které mají dostat novou funkčnost. Na základě této analýzy je třeba vytvořit rozhraní pro objekt návštěvníka, které bude obsahovat konkrétní metodu *visit()* pro každý prvek s novou funkcionalitou.

Do původních tříd objektů v rozšiřované struktuře je potřeba přidat metodu *acceptVisitor(),* pro komunikaci se třídou návštěvníka a v této metodě volat příslušnou metodu *visit().*

Nakonec je potřeba projít u každé třídy s upravovanou funkčností odkazované objekty a jejich nové metodě *acceptVisitor()* předat objekt návštěvníka [1].

4.3.6 Návrhový vzor Iterátor (Iterator)

Pointa

Návrhový vzor Iterátor poskytuje možnost postupně přistupovat k prvkům složených objektů bez znalosti jejich základní struktury [1].

Účel vzoru

Návrhový vzor Iterátor poslouží především v situacích, kdy je pro získání různých informací důležité procházet konkrétní strukturu objektů opakovaně různými způsoby. Použití (většinou externího (vnějšího)) iterátoru zjednoduší rozhraní objektu agregátoru. Rozhraní agregátoru musí obsahovat pouze metodu *getIterator()* pro vrácení konkrétního iterátoru, což umožnuje paralelní iterace (procházení) struktury.

Různé způsoby procházení jsou užitečné v případech, kdy je potřeba získávat ze struktury objektů různé mnohotvárné informace.

Řešení problému

Využití tohoto vzoru musí předcházet úvaha nad typem použitého rozhraní pro určenou třídu. Využití tohoto vzoru je v jazyce PHP značně usnadněno možností využít knihovnu SPL (Standard PHP Library), která nabízí mimo jiné i předpřipravené rozhraní *Iterator* a rozhraní pro externí iterace *IteratorAggregate*.

Rozhraní *Iterator* slouží obecně pro jednodušší funkčnosti. V jednu chvíli může probíhat pouze jedna iterace, ale není třeba porušovat zapouzdření agregátu. Další možností je použít rozhraní *IteratorAgregate*, které umožnuje více paralelních iterací (více iterátorů pro stejné údaje), ale v určitých případech je nutné porušit zapouzdření agregátu kvůli přístupu iterátoru k potřebným atributům.

Po zvolení vhodnějšího rozhraní je třeba implementovat metody, které zvolené rozhraní vyžaduje [1].

4.3.7 Návrhový vzor Stav (State)

Pointa

Tento návrhový vzor slouží pro správu různorodých stavů objektů. Použití vzoru Stav poskytne možnost současně měnit chování objektů při změně jejich vnitřních stavů [1].

Účel vzoru

Při použití tohoto návrhového vzoru se při změně vnitřního stavu objektu tato změna projeví změnou třídy objektu. Použití vzoru přidává (pomocí nových tříd) další úroveň zapouzdření pro jednotlivá chování objektů. Takto rozdělené třídy pomáhají přehlednosti kódu, ale zvětšují celkový počet tříd aplikace. Před použitím vzoru je proto důležité zhodnotit rozsáhlost jednotlivých metod a určit jestli se vyplatí jejich rozdělení za cenu existence většího počtu tříd.

Často je vzor využíván také při programování síťových aplikací a programů, které vykonávají úlohu syntaktické analýzy, což znamená kontrolu syntaktické správnosti dokumentu [31].

Řešení problému

Při odhalení třídy, která se chová dle aktuálního stavu instance a využívá k tomu rozsáhlé metody, se vyplatí využít návrhový vzor Stav. Počátečním krokem je vytvoření rozhraní,

které obsahuje vybrané metody ovlivňující stav instance sledované třídy. V této původní třídě musí být vytvořena metoda ukládání aktuálního stavu a metoda pro změnu současného stavu.

Původní sledovaná třída může mít funkčnost pro správu svých stavů, nebo mohou být tyto stavy spravovány pomocí vzoru Tovární metoda či Jedináček [1].

Třídy jednotlivých stavů a funkčnost jejich vzájemné záměny jsou vytvořeny nakonec.

4.3.8 Návrhový vzor Strategie (Strategy)

Pointa

Návrhový vzor Strategie zapouzdřuje skupinu algoritmů a umožnuje tak výměnu právě používaného algoritmu nezávisle na objektu který algoritmus využívá.

Účel vzoru

Tento vzor najde využití v případech, kdy je v aplikaci potřeba opakovaně využívat různé postupy (strategie) operací. S využitím tohoto vzoru se stávají základní třídy (které vzor využívají) přehlednější a zlepšuje se možnost jejich znovupoužití. Nevýhodou může být skutečnost, že se zvyšujícím se počtem strategií může docházet k nadměrnému růstu celkového počtu tříd.

Řešení problému

Nejprve je nutné připravit obecné rozhraní pro všechny třídy jednotlivých algoritmů a vytvořit nejméně dvě konkrétní třídy využívající toto vytvořené rozhraní. Dalším krokem je vytvoření funkcionality, která umožní využít v základní třídě dříve vytvořené konkrétní třídy s algoritmy [1]. Základní třída je třída, která bude využívat prvky tohoto vzoru.

V základní třídě je nakonec potřeba vytvořit metodu, která svůj požadavek předá konkrétní třídě s právě využívaným algoritmem.

4.3.9 Návrhový vzor Zřetězení zodpovědnosti (Chain of Responsibility)

Pointa

Návrhový vzor Zřetězení zodpovědnosti odděluje původce požadavku od jeho příjemce, což umožnuje, aby požadavek obdrželo a provedlo více objektů [1]. Pomocí vzoru lze vytvářet zřetězené struktury objektů, které postupně obdrží konkrétní požadavek, dokud ho některý objekt nezpracuje.

Účel vzoru

Pokud je v aplikaci potřeba zpracovávat nějaký požadavek postupně různými způsoby, využije se právě tento návrhový vzor. Vzor Zřetězení zodpovědnosti nabízí možnost kontroly komunikace mezi odesilatelem a příjemcem požadavku, přičemž tak umožní, aby měl požadavek vícero postupných příjemců (řešitelů). Využití této funkcionality může být žádoucí v mnoha konkrétních situacích. Řešení požadavku tak například může začínat u těch nejjednodušších metod a v případě, že se s jejich pomocí požadavek nepodaří vyřešit, odešle se dalšímu objektu v řetězci, který může využívat složitější metody nebo vyššího oprávnění.

Problémy při využití vzoru

Orientace v kódu je při použití tohoto návrhového vzoru může být mnohem obtížnější, než při pevné vazbě mezi původcem požadavku a jeho příjemcem.

S použitím tohoto vzoru je třeba mít na paměti, že chování takové zřetězené struktury objektů může vyústit v situaci, kdy není požadavek zpracován žádným objektem v řetězci. V některých případech je takové chování vhodné, ale v některých situacích je nutné požadavek vyřídit a je třeba implementovat funkčnost, která tento problém bude řešit.

Řešení problému

Prvním krokem pro vytvoření struktury objektů podle tohoto vzoru je vytvoření rozhraní s metodami pro zpracování konkrétních požadavků.

Toto rozhraní musí využívat všechny třídy schopné zpracovávat požadavky.

Dalším krokem je přidání funkcionality, která objektům přidá možnost (v případě že nemůžou požadavek vyřešit) odeslat požadavek dalšímu objektu v řetězci, jehož odkaz má uložen.

Poté je již možné sestavit konkrétní řetězec uložením odkazů na další objekt v objektech předchozích [1].

4.4 Další návrhové vzory, které obvykle nejsou v jazyce PHP využívané.

Pro tuto příručku byly vybrány vzory, které mají v jazyce PHP využití. Existují i další návrhové vzory, které jsou v tomto jazyce využívány jen velmi zřídka, popřípadě vůbec. V této kapitole dva z nich krátce popíšu.

Návrhový vzor Interpret (Interpreter)

Tento návrhový vzor je jeden z nejméně využívaných v jazyce PHP [1]. Vzor slouží pro vytváření reprezentací gramatických pravidel jazyka pomocí tříd a určuje způsob jak tyto gramatická pravidla využívat.

Implementace jazyka dle tohoto vzoru má tu výhodu, že lze díky struktuře abstraktního syntaktického stromu snadno přidávat nová klíčová slova. V případech se složitou gramatikou však může být problém strukturu abstraktního syntaktického stromu vytvořit [1].

Návrhový vzor Pamětník (Memento)

Situace je obdobná jako u vzoru Interpreter. Také se jedná o jeden z nejméně používaných návrhových vzorů v jazyce PHP.

Už autoři knihy GOF poznamenali, že vytvořit potřebná rozhraní může být obtížné v jazycích, které nemají dvě úrovně statické (na úrovni třídy) ochrany, což PHP zatím nemá [32].

Pomocí tohoto vzoru je možné odchytit a uložit aktuální stav objektu bez porušení jeho zapouzdření, což umožní využívat zpětné funkce, jimiž lze objekt vrátit do původního stavu [1].

5 Závěr práce

Hlavním bodem zájmu práce jsou návrhové vzory v kontextu jazyka PHP, do jejichž problematiky byl čtenář zasvěcen s využitím historických a současně i novodobých souvislostí. V počáteční části jsem osvětlil skutečnosti pronikání paradigma objektově orientovaného programování do jazyka PHP, což mi umožnilo vhodně navázat popisem historického vývoje a podstaty návrhových vzorů.

V další části práce jsem vytvořil kapitolu Příručka vybraných návrhových vzorů, kde objasňuji účel návrhových vzorů, které jsou vhodné pro použití v jazyce PHP. Příručka cílí především na programátory, kteří s návrhovými vzory začínají. Přehled jednotlivých vzorů je proto zpracován s důrazem na co nejpřímočařejší vysvětlení jejich podstaty.

5.1 Doporučení pro úpravu kurzů zabývajících se výukou PHP

Na závěr bych chtěl navrhnout následující doporučení pro úpravu kurzů zabývajících se výukou PHP na Vysoké Škole Ekonomické, které vycházejí ze znalostí získaných převážně díky tvorbě této práce:

Za velmi důležité považuji především přidat mezi témata probíraná na cvičeních a především na přednáškách v rámci výuky tvorby dynamicky generovaných stránek v PHP část věnovanou výuce možnosti využití paradigma objektově orientovaného programování a následně výuce možností, které přináší nejvyužívanější návrhové vzory.

Jak jsem již uvedl v rešeršní části, mnoho výukových publikací pro jazyk PHP objektově orientované programování (OOP) přehlíží a návrhové vzory často vůbec nezmiňuje.

Z vlastní zkušenosti vím, že na vstupních kurzech zabývajících se výukou PHP se málo kdy nesejdou jak lidé, kteří už mají s tvorbou v tomto jazyce obsáhlejší zkušenosti, tak lidé kteří jsou téměř naprostí začátečníci. Neměli bychom mít za zlé snahu vyučujících věnovat v takových případech více času osvětlováním základních znalostí na úkor zkušenějších studentů, nicméně je důležité předat těm pokročilejším minimálně základy znalostí o možnostech využití OOP a Návrhových vzorů. Pokročilejší studenti by se měli dle mého názoru dozvědět zmiňované znalosti na přednáškách v poslední třetině semestru a vyzkoušet si na cvičení s pomocí vyučujícího implementovat minimálně jeden návrhový vzor na konkrétní situaci.

5.2 Dosažené cíle práce

Na počátku tvorby této práce jsem si stanovil určité cíle, kterých bych v práci rád dosáhl. S využitím zamýšlených postupů se mi podařilo zpracovat vše, co jsem dopředu určil. Hlavní cíle práce, kterých jsem dosáhl, sem opět shrnul v následujících bodech:

- Byly identifikovány důležité pojmy a oblasti objektově orientovaného programování, které byli následně stručně a jasně představeny.
- Podařilo se osvětlit vývoj podpory paradigma objektově orientovaného programování v jazyce PHP od historických počátků vývoje jazyka až do současnosti.
- Byl představen koncept návrhových vzorů jak z obecného hlediska, tak se zaměřením na možnosti podpory návrhových vzorů v jazyce PHP.
- V další části práce vznikla příručka s vybranými návrhovými vzory, které je vhodné v jazyce PHP v určitých situacích využívat.
- Nakonec byly navrženy doporučení pro změny ve výuce kurzů zabývajících se výukou PHP.

Terminologický slovník

Termín	Význam [zdroj]
Objektově orientované programování (OOP)	Objektově orientované programování je programovací paradigma, které představuje způsob uvažování nad psaním kódu. Způsob spočívá především ve vytváření několika samostatných objektů - částí aplikace, které jsou díky svému rozhraní opakovaně využitelné i v jiných aplikacích [autor],[9].
Objekt	Jako objekty jsou v programech označovány kolekce dat, které reprezentují skutečný objekt z modelovaného světa [4].
Třída	Pojem třída ve zkratce představuje šablonu, která slouží pro generování (tvorbu) konkrétních objektů [autor].
Dědění	Pojem dědění [4] představuje možnost vytvářet v objektově orientovaných programovacích jazycích hierarchii mezi jednotlivými třídami [autor].
Polymorfizmus	Jako polymorfismus označujeme schopnost objektů na stejné zprávy reagovat různými způsoby [4].
Datový typ	Stejné charakteristiky určitých objektů nazýváme datové typy [4].
Metody	Metody slouží k manipulaci s daty objektu [5].
Rozhraní	Rozhraní je speciálním druh datového typu, který obsahuje veřejně přístupné metody [autor].
Zend Engine	Jádro jazyka PHP, implementované v jazyce C a pojmenované Zend Engine podle křestních jmen autorů [autor],[2].
Návrhový vzor	Pojem návrhový vzor v programování znamená popis často se opakujícího problému návrhu počítačových programů a předkládají doporučený postup jeho řešení [autor].
Gang of Four (GoF)	Zkratkou GoF bývá označována skupina autorů knihy *Design patterns: elements of reusable object-oriented software* a přeneseně i jejich kniha je tak často označována i v odborné literatuře [3].

Seznam literatury

[1] BÖHMER, Marian. *Návrhové vzory v PHP: [23 vzorových postupů pro rychlejší vývoj]*. 1. vyd. Brno: Computer Press, 2012. ISBN 978-80-251-3338-5.

[2] History of PHP. *Php.net* [online]. The PHP Group [cit. 2016-04-24]. Dostupné z: http://php.net/manual/en/history.php.php

[3] PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Vyd. 1. Brno: Computer Press, 2007. ISBN 978-80-251-1582-4.

[4] PECINOVSKÝ, Rudolf. Java 8: úvod do objektové architektury pro mírně pokročilé. První vydání. Praha: Grada Publishing, 2014. Knihovna programátora (Grada). ISBN 978-80-247-4638-8.

[5] BRÁZA, Jiří. PHP 5: začínáme programovat. 1. vyd. Praha: Grada, 2005. Průvodce (Grada). ISBN 80-247-1146-X.

[6] PHP 7. Zend.com [online]. Zend Technologies Ltd [cit. 2016-04-24]. Dostupné z: http://www.zend.com/en/resources/php-7

[7] The Origins of Pattern Theory, the Future of the Theory, And The Generation of a Living World. Patternlanguage.com [online]. Christopher Alexander, 1996 [cit. 2016-04-24]. Dostupné z: https://www.patternlanguage.com/archive/ieee/ieeetext.htm

[8] SURASKI, Zeev. The Object-Oriented Evolution of PHP. In: Devx.com [online]. 2002 [cit. 2016-04-24]. Dostupné z: http://www.devx.com/webdev/Article/10007/0/page/2

[9] ČÁPKA, David. Objektově orientované programování a evoluce vývoje softwaru. In: Itnetwork.cz [online]. [cit. 2016-04-24]. Dostupné z: http://www.itnetwork.cz/navrhove-vzory/objektove-orientovane-programovani-a-evoluece-vyvoje-softwaru

[10] Rasmus Lerdorf. Lerdorf.com [online]. [cit. 2016-04-24]. Dostupné z: http://lerdorf.com/bio/

[11] Design Patterns. Sourcemaking.com [online]. [cit. 2016-04-24]. Dostupné z: https://sourcemaking.com/design_patterns

[12] MONUS, Anna. PHP 7: 10 Things You Need to Know. In: *Hongkiat.com* [online]. 2015 [cit. 2016-04-24]. Dostupné z: http://www.hongkiat.com/blog/php7

[13] BELSKI, Anatol. PHP RFC: Removal of dead or not yet PHP7 ported SAPIs and extensions. In: Php.net/ [online]. 2015 [cit. 2016-04-24]. Dostupné z: https://wiki.php.net/rfc/removal_of_dead_sapis_and_exts

[14] POPOV, Nikita. PHP RFC: Remove deprecated functionality in PHP 7. In: Php.net/ [online]. 2014 [cit. 2016-04-24]. Dostupné z: https://wiki.php.net/rfc/remove_deprecated_functionality_in_php7

[15] Návrhové vzory pro J2EE. In: Vsb.cz [online]. [cit. 2016-04-24]. Dostupné z: http://www.cs.vsb.cz/benes/vyuka/tis/prednasky/14-j2eedp-4.pdf

[16] KHOSROW-POUR, Mehdi. Encyclopedia of information science and technology. 2nd ed. Hershey, PA: Information Science Reference, c2009. ISBN 9781605660271.

[17] About OMG®. Omg.org [online]. [cit. 2016-04-24]. Dostupné z: : http://www.omg.org/gettingstarted/gettingstartedindex.htm

[18] GAMMA, Erich. Design patterns: elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley, c1995. ISBN 0201633612.

[19] ALEXANDER, Christopher., Sara ISHIKAWA a Murray SILVERSTEIN. A pattern language: towns, buildings, construction. New York: Oxford University Press, 1977. ISBN 0195019199.

[20] SARAY, Aaron. Professional PHP design patterns. Indianapolis, IN: Wiley, c2009. Wrox professional guides. ISBN 978-0-470-49670-1.

[21] Composite Pattern. Univerzitní informační systém Mendelovy univerzity v Brně [online]. [cit. 2016-05-01]. Dostupné z: https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=32230

[22] Adapter Design Pattern. Sourcemaking [online]. [cit. 2016-05-01]. Dostupné z: https://sourcemaking.com/design_patterns/adapter

[23] Proxy (zástupce). Voho.cz [online]. [cit. 2016-05-01]. Dostupné z: http://voho.cz/wiki/proxy/

[24] Observer Design Pattern. Sourcemaking [online]. [cit. 2016-05-01]. Dostupné z: https://sourcemaking.com/design_patterns/observer

[25] ZANDSTRA, Matt. PHP objects, patterns, and practice. Fourth edition. Berkeley, CA: Apress, 2013. ISBN 9781430260318.

[26] HAYDER, Hasin. Object-oriented programming with PHP5: learn to leverage PHP5's OOP features to write manageable applications with ease. [Online-Ausg.]. Birmingham [u.a.]: Packt Publ, 2007. ISBN 9781847192561.

[27] SANDERS, William B. Learning PHP design patterns. Sebastopol, CA: O'Reilly Media, 2013. ISBN 1449344917.

[28] Standard PHP Library (SPL). Php.net [online]. [cit. 2016-05-02]. Dostupné z: http://php.net/manual/en/book.spl.php

[29] Mediator Design Pattern. Sourcemaking [online]. [cit. 2016-05-02]. Dostupné z: https://sourcemaking.com/design_patterns/mediator

[30] Zend Framework. Framework.zend.com [online]. [cit. 2016-05-02]. Dostupné z: http://framework.zend.com/about/

[31] Parsery: XML & aplikace. Kosek.cz [online]. [cit. 2016-05-03]. Dostupné z: http://www.kosek.cz/clanky/swn-xml/ar02s30.html

[32] PHP Memento Design Pattern Part I: Wide & Narrow Interfaces. Php5dp.com [online]. [cit. 2016-05-03]. Dostupné z: http://www.php5dp.com/php-memento-design-pattern-part-i-wide-narrow-interfaces/

Seznam obrázků a tabulek

Seznam obrázků

Seznam tabulek