Jihočeská univerzitav Českých Budějovicích
Pedagogická fakulta
katedra informatiky

studijní obor: Aplikovaná výpočetní technika

Bakalářská práce

Distribuovaný systém pro správu obsahu

Vedoucí bakalářské práce
Paedr. Petr Pexa

Autor
Jan Turoň

2006

Anotace

V průběhu roku 2005 se na většině internetových serverů rozšířil hypertextový preprocesor PHP 5 umožňující plnohodnotné objektové programování. To posouvá tvorbu internetových aplikací o celou generací dále a otevírá tak prostor moderním programátorským přístupům.

Jeden z těchto přístupů je i distribuovanost: tato práce zkoumá její současné možnosti ve tvorbě systémů pro správu obsahu. Studiem aplikačně značně nedoceněných regulárních výrazů se zabývá novými cestami, které toto spojení nabízí.

Prohlašuji, že jsem tuto práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

OBSAH

ÚVOD

Motivace, která je příčinou veškeré lidské činnosti, mívá dvě složky: povinnost a zábavu (ať už ji člověk chápe jako hrátky, výzvy či snahu uplatnit se).

Ryzí povinnost znamená vykonat nějakou činnost výměnou za protihodnotu. Je pak vykonávána jen proto, abychom se dostali ke kýžené protihodnotě. U ryzí zábavy je odvedená práce také protihodnotou: toho, kdo ji vykonává, uspokojuje, má z ní radost a záleží mu na ní.

Moderní společnost je založena na soukolí povinností. Přesto jsem se snažil tuto práci pojmout jako zábavu: vložil jsem do ní upřímné úsilí, snažil se o stručnost a srozumitelnost a nepřejal nic, co bych si nevyzkoušel. Velmi bych ocenil, kdyby si čtenář z této práce něco odnesl či pokud by nalezené nedostatky pomohl odstranit.

2.1 Typografická konvence

- *kurzívou* jsou označeny nové pojmy a citace
- čárkované podtržení značí úryvky kódu v textu
- `neproporcionálně na šedém pozadí` je zapsán samostatný blok kódu
- písmem Arial Narrow jsou zapsány dokumentační URI.

2.2 Základní pojmy

K pochopení pojmu *distribuovaný systém pro správu* obsahu je nutné seznámit se s některými klíčovými pojmy.

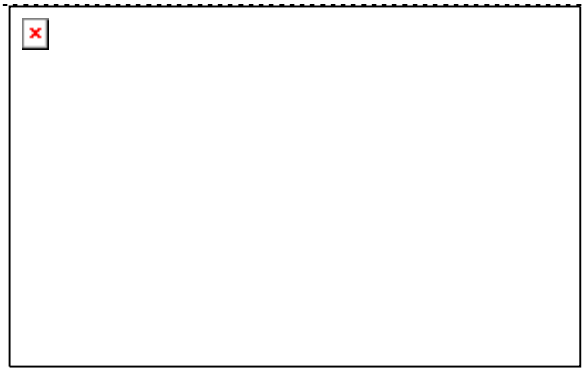
2.2.1 IS

Moderní společnost nahlíží na informaci jako na formu služby, která se stala s rozvojem Internetu snadno obchodovatelnou. Podobně jako obchodní domy pro fyzická zboží začaly pro efektivní správu informací vznikat *informační systémy* (*IS*).

IS nabízí řadu služeb dle povahy informací: jsou-li důvěrné, zabezpečuje je před neprověřenými přístupy; je-li jich mnoho, třídí je; je-li třeba, zálohuje je; také je přehledně zobrazuje a manipuluje s nimi: tyto služby jsou úkolem *systému pro správu obsahu* (z angl. *content management system*, *CMS*).

Pochopení struktury současných informačních systémů a začlenění nížezmiňovaných pojmů pomáhá následující diagram.

Obr. 1: informační systém



2.2.2 CMS

Základní úlohou CMS pro splnění výšeuvedeného úkolu je získat data od zdrojů a vytvořit přehledné a snadno ovladatelné rozhraní mezi uživateli dat a jejich zdroji.

Požadavky na CMS se většinou plynule mění. Aby bylo možné systém upravovat jinak než znovuvybudováním, je nutné vytvořit funkční strukturu jeho částí. Tento přístup se nazývá *rozvržení správy obsahu* (z angl. *content management framework*, *CMF*).

2.2.3 CMF

Idea všech úspěšnějších CMF, na které jsem dosud narazil, je *jádro*, které načítá *moduly*, které plní přesně vymezené funkce. Toto je velmi efektivní přístup: každý modul realizuje jednotlivé části CMS (např. zpracování vstupních dat či způsob jejich prezentace) a zásah do modulu teoreticky neovlivní zbytek CMS.

Tento společný jmenovatel nabádá k vyšší abstrakci, kterou se pokouší vystihnout *distribuovaný systém pro správu obsahu* (*DCMS*).

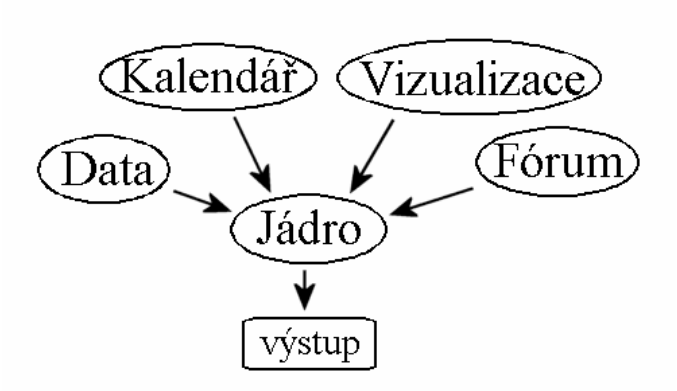
2.2.4 DCMS

I když stávající systémy používají v podstatě stejné rozvržení, jsou vzájemně nekompatibilní. Běží v rámci jednoho IS a většinou se pro okolí tváří jako černá skříňka. Nazvěme tyto systémy *kompaktní*.

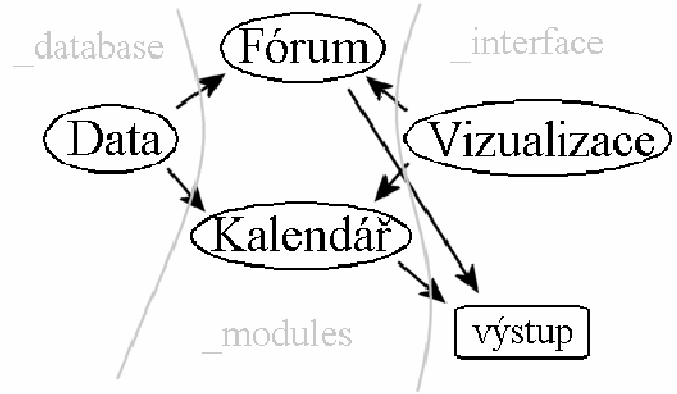
Idea distribuovaného systému nevychází z potřeby rozdělit systém na logické části z důvodu rozložení zátěže (jak je tomu například u databází), ale z důvodu integrace: postupu, kterým prochází většina IS. Ty se totiž v průběhu rozšiřování musí přizpůsobovat novým požadavkům. Dosud se to řešilo rozšířením kompaktního CMS, distribuovaný CMS se místo toho snaží využít již existující systémy.

Již v této fázi je zřejmé, že komunikace mezi částmi systému bude jedním z nejdůležitějších bodů. Rozdíl v přístupu k řešení ilustrují Obr. 2 (kompaktní řešení) a Obr. 3 (distribuované řešení).

Obr. 2: kompaktní systém



Obr. 3: distribuovaný systém



Pro další studium si rozeberme několik zavedených (kompaktních) CMS.

ROZBOR ZAVEDENÝCH CMS

Na Internetu je mnoho propracovaných CMS [2], které za sebou mají dlouhý vývoj a za kterými stojí celé týmy lidí. Silnou konkurenci však rozmělňuje mnoho možností specializací, množství přístupů k řešení správy obsahu a stále se rozvíjející technologie.

Rozvoj v této oblasti je velmi rychlý. Přesto je nezbytné učinit si základní přehled[1], aby tato práce neřešila něco, co je popsáno jinde a lépe.

3.1 Sledované vlastnosti

Každý CMS lze hodnotit podle desítek ukazatelů, například možnost doplňků, automatické členění externích dokumentů, vícevrstvou hierarchii správy, verzování pro různé prohlížeče, systémových požadavků, bezpečnosti, ovladatelnosti, výkonu, šířky či podpora jiných jazyků a systémů.

Neexistuje žádný standardizovaný srovnávací systém, proto jsem se zaměřil na čtyři ukazatele, které považuji za nejobecnější, tedy mezi různými systémy nejsnáze porovnatelné.

3.1.1 Šířka CMS

Jedním ze sledovaných ukazatelů je šířka oblasti, do které CMS zasahují. Nevýhody těch šíře zaměřených často bývají vyšší nároky na lidi i prostředky než u jednodušších CMS. Při výběru se většinou volí vhodný kompromis s ohledem na oblast, kterou příslušný IS pokrývá.

Kompromisů však existuje mnoho (s ohledem např. na rychlost, spolehlivost, zabezpečení, přizpůsobivost aj.), rychle vznikají nové požadavky a nové standardy a specializace (např. obchodní CMS či redakční systémy).

[1] vzhledem k rychlosti vývoje není účelné se tímto zabývat do přílišné hloubky: informace zastarají nejpozději během několika málo měsíců

3.1.2 Rozšiřitelnost a transparentnost

Systém by mělo být možno rozšiřovat o nové moduly, a to *čistě*, tj. přidání nového modulu nijak neovlivní další moduly a jejich přidávání, nesmí vyžadovat zásahy do jádra, aby přechod na vyšší verzi systému neznamenal nutnost všechna přizpůsobení systému provádět znovu. Čistota je vyžadována také od jádra: při pozdějších rozšiřováních nesmí být závislé na žádném modulu, tj. odinstalování libovolného modulu nesmí ovlivnit ostatní části systému. Čistá instalace se ukázala být u sledovaných CMS problematická.

Transparentností rozumíme členění kódu na logické části, aby bylo zřejmé, která část je za co zodpovědná. Kód, který transparentní není, se rozšiřuje těžce.

3.1.3 Dokumentace

Velmi rozšířený omyl tvůrců CMS je, že nejen moderátor, ale ani správce nemusí znát HTML ani jiné internetové nástroje. Ještě jsem totiž nepotkal člověka, který by neuměl ani HTML a pustil se do správy systému (takový by neuměl řešit ani jednoduché problémy, kterých při nejlepší vůli bývá při provozu mnoho).

Racionální je u správce alespoň elementární znalost PHP předpokládat a dokumentovat zejména kód a architekturu, ne administrátorské nástroje (v těch by se měl každý vyznat intuitivně).

Kód modulů by neměl být psán jako černá skříňka, ale přehledně a srozumitelně.

3.1.4 Systémové nároky

CMS lze nahlížet jako nástroj pro datovou abstrakci, tvůrci pak často předpokládají příliš obecná data. Obecnost však znamená také vyšší složitost a pomalost, modul se často nepíše datům na míru. Nemělo by se zapomínat blíže prozkoumat data (zejména jejich datové toky a množství změn v čase) dříve, než se začne modul navrhovat.

Kupříkladu CMS využívající databáze mívají sklon ukládat do nich každou drobnost. Je třeba si ale uvědomit, že přístup k databázovému serveru zpomaluje načítání a jeho výpadky bývají častější než u HTTP serveru. Do databází je vhodné ukládat rozsáhlá data se kterými je nutno bezpečně manipulovat a která se mají třídit či jinak vyhodnocovat. Ostatní perzistentní data se mi osvědčilo ukládat do souborů na HTTP či FTP serveru, nejsou-li nezbytná, pak jako COOKIES na klientském počítači.

3.2 Testování CMS

Všechny níže testované CMS jsou z kategorie Open-source. Volil jsem co nejširší pole ve sledovaných vlastnostech. Zkoumal jsem i některé další systémy (např. MediaWiki, který podporuje Wikipedii), které se mi však nepodařilo testovat v plném rozsahu z důvodu vyžadování některých choulostivých funkcí (např. system, socketové funkce), které bývají poskytovateli zakázány.

Testování probíhalo na serveru webzdarma [1] (mnohé systémy měly problémy s automatickým vkládáním reklamy), na placeném serveru [3], na školním serveru [4] a na domácím odříznutém (standalone) serveru, všechny s podporou PHP 4 (příp. 5) a MySQL, tedy technologie většinou postačující pro běh sledovaných CMS.

3.2.1 CMSimple

Autor Peter Harteg, Dán. Heslo Small, Simple, Smart výstižně popisuje hlavní rysy systému: je velmi jednoduchý, po celou dobu testování jsem se nesetkal s problémovou situací.

Z nížeuvedených důvodů jej hodnotím jako jednoznačně nejlepší ze všech sledovaných CMS.

Zdrojový kód: Velmi úsporný a zhuštěný, autor pochází ze staré programátorské školy, například používá jednopísmenné řídící proměnné či více příkazů na řádku: slovy autora spaghetti scripting. Důsledně však dodržuje adresářovou strukturu, odděluje moduly, vzhled, obsah. Narozdíl od ostatních systémů jsem nikde nebyl nucen kód editovat.

Systém zabírá pouhých 70kB, jeho jádro pak kolem 30kB.

Dokumentace: Nejlepší ze všech sledovaných CMS: na prezentačních stránkách je podrobný uživatelský, instalační i vývojářský manuál, popisující systém na více úrovních od uživatelského ovládání po architekturu systému a význam jednotlivých proměnných. Obdivuhodné je, že dokumentace je rozsáhlejší než samotný kód.

Rozšiřitelnost: Obsahuje skript pro instalaci jisté třídy modulů, a to přímo z rozhraní systému. Velmi intuitivní rozšiřování textového obsahu: efektivní interní WYSIWYG editor (7kB kód) přímo kdekoli zobrazitelný. Přechod na vyšší verzi je bezproblémový: stačí zálohovat adresáře s daty a překopírovat na server nový systém.

Systémové nároky: Nevyžaduje databázi (bez rozšiřujících modulů), nenárokuje si celý HTTP server, pouze svůj adresář. Jednoduchá instalace: stačí kamkoliv nakopírovat, není nutno nic nastavovat.

Zdroj: www.cmsimple.dk [5]

3.2.2 Plone

Vysoce sofistikovaný CMS běžící pod vlastním serverem *ZOPE*. Vhodný pro spolupráci rozsáhlých týmů: alespoň jeden člověk musí systému rozumět do hloubky, ostatní vystačí s uživatelskou znalostí *ZMI [7]*: internetového rozhraní, které ZOPE nabízí.

Tento systém využívá i JČU. Jeho možnosti jsou asi nejširší ze všech sledovaných CMS, přesto všude, kde jsem se s ním setkal (CIT, ZSF, domácí server), bylo nutno řešit relativně obtížné problémy nesouvisející s prezentovanými daty.

Vyvíjí jej celé týmy lidí, jedná se o nejsložitější systém nabízený zdarma, se kterým jsem se dosud setkal, příliš složitý na jednoduché projekty [7].

Zdrojový kód: Jádro ZOPE je tvořeno zejména objekty jazyka Python, pro které ZOPE využívá interní objektovou databázi *ZODB*. Jeho textová definice vyžaduje asi 100MB diskového prostoru.

Dokumentace: Velice všestranná. Na prezentačních stránkách se nachází detailní dokumentace všech částí tohoto složitého systému, včetně školení, tipů a rozborů jeho výhod a nevýhod.

Rozšiřitelnost: Vzhledem k objektové struktuře je rozšiřování velmi efektivní a teoreticky bezproblémové. Každý kus kódu má svůj vlastní prostor, díky dědění a polymorfismu může být týž kód velmi efektivně využíván ve složitých strukturách vyšších vrstev, které se (také díky UML) rozšiřují velmi snadno. V tom spočívá dle mého největší síla Plone.

Síla tohoto rysu se však projeví až při instalaci více různorodých modulů. Pro malé a střední projekty je struktura systému příliš obecná.

Systémové nároky: Systém využívá pomalou objektovou databázi prakticky při každé operaci. Navíc negeneruje prostý kód stránek, ale obecné objekty, které se převádí pomocí ZPublisheru – součásti serveru ZOPE. V pomalosti a vysokých serverových nárocích vidím největší slabinu Plone: objekty nejsou vždy nejvýhodnější datovou strukturou, při vysoké zátěži snáze server kolabuje.

Zdroj: www.plone.org [6]

3.2.3 PHP Nuke

Prodělává relativně nerovnoměrný a pomalý vývoj, vyvíjí ho prakticky jeden člověk (Francisco Burzi).

Přestože některé verze PHP Nuke jsou nabízeny komerčně, ani v jedné sledované oblasti nevyniká nad ostatní testované systémy: jeho heslo The future of the web vidím jako silně zavádějící z následujících důvodů:

Zdrojový kód: moduly tvoří dynamicky načítané soubory, které obsahují pouze funkce (bez jakéhokoli jmenného prostoru nebo alespoň pravidel pro pojmenování). Funkce jednotlivých modulů se volají podle globální proměnné $op, což je velmi jednoduchý a efektivní přístup, umožňující „lepení modulů“, viz níže.

Dokumentace: Špatná. Zcela chybí dokumentace kódu a architektury, dokumentována pouze triviální nastavení.

Rozšiřitelnost: Kladně hodnotím, že se systém nevyvíjí pouze záplatami, ale na úrovni modulů, což umožňuje jeho vysokou přizpůsobivost. Díky tomu se phpNuke stal vzorem několika dalším CMS (např. *openPHPNuke*, *postNuke*, *UNITED-NUKE*).

Čistou instalaci umožňují takzvané *bloky*, což jsou graficky ucelené oblasti ve sloupcovém návrhu. Jejich struktura je velmi jednoduchá: obsahují kód podle této šablony:

```
if (eregi("filename.php", $_SERVER['PHP_SELF'])) {
    Header("Location: index.php");
    die();
}
$content="HTML obsah";
```

Podmínka zajistí, že modul nebude volán samostatně. HTML obsah proměnné $content potom systém obalí tabulkou a vloží do sloupce.

Některé části se však instalují i nečistě (a styl, kterým je projekt napsán prozrazuje, že se na čistotu moc nehledí), jak dokládají pokyny k instalaci NukeCalendar [9]

1. Install a FRESH PHP Nuke v5.6 and NukeCalendar 1.1a to your server base directory

Hint: if you don't use a fresh installation it should also work, but you could damage your current installation. So be careful !

2a. Replace the following...

Systémové nároky: Systém příliš využívá HTML tabulky, které se v prohlížečích zobrazují různě a jejich načítání je pomalé. Nehospodárné využívání databáze: desítky SQL tabulek s jednotkami triviálních záznamů, např. tabulka blocks obsahující seznam bloků: data mění pouze administrátor při změně systému, umístění do souboru by bylo mnohem efektivnější.
Zdroj: www.phpnuke.org [8]

3.2.4 phpRS

Český CMS, který vyvíjí Jiří Lukáš, původně pro svůj elektronický časopis Supersvět.

Zdrojový kód: Je to příklad systému vyvíjeného pomocí záplat. Časem se kód stal značně chaotický, moduly pro vyšší verze jsou dokonce občas nepoužitelné pro verze nižší. Přechod na vyšší verzi je možný pouze pro ty, kteří mají záznam o tom, co kde v kódu změnili.

Rozšiřitelnost a dokumentace: Nevyhovující. Při každé instalaci nového modulu je nutno měnit soubory config.php a css.css, mnohdy dokonce i adresářovou strukturu. Následující větu z postupu při instalaci modulu galerie považuji za dostatečně ilustrativní, bez nutnosti dalších komentářů: [11]

…Nyní si otevřeme soubor css.css (např F3 v Total Commandrovi) a jeho obsah zkopírujeme do svého CSS souboru. …

Zdroj: www.phprs.cz [10]

3.3 Vytyčení vlastní cesty

Z výšeuvedených systémů jsem se snažil přidržet dobře fungujících věcí a vymyslet elegantnější cestu tam, kde zaostávaly. Předsevzal jsem si zejména následující body:

- Instalaci jádra musí postačovat prosté zkopírování na místo, odkud má běžet, složitější nastavení může být součástí nepovinných modulů. Server totiž může mít různá omezení, se kterými složitější instalace mnohdy nepočítají. Zvláště je nutno brát v úvahu, že na serveru mohou běžet i jiné aplikace. Instalace modulů musí být čisté (viz 1.2.2).
- Rozvržení (framework) musí být takové, aby se systém s narůstajícím počtem doplňujících modulů nestával z hlediska architektury a kódu nepřehlednější (což krom phpRS splňují všechny sledované systémy).
- Velkou pozornost je vhodné věnovat dokumentaci (viz 1.2.3) a přehlednému psaní kódu. Příliš zhuštěný kód je netransparentní, ale i v příliš rozvleklém programátorském vyjadřování je nesnadné se vyznat.
- Objektová struktura je vhodná pouze u těch částí, u nichž se předpokládá rozšiřování a vývoj, tj. u modulů. Jádro takové být nemusí, jeho ceněné vlastnosti nejsou vývojové (např. transparentnost, otevřenost), ale provozní (např. rychlost, spolehlivost).

Idea je, že jádro nemusí plnit funkci dispečera (tj. mít přehled o všem, co se v systému děje), postačí funkce vrátného (tj. korektně začlenit kód modulů). Moduly pak lze přirovnat k pracovníkům firmy: většina z nich je důležitějších než vrátný. Není bezpečnostní díra přenechat co největší možnosti řízení modulům, neboť snahou jejich tvůrců není do systému se nabourat, ale vylepšit jej.

REALIZACE

Aby bylo možno bez zásahu do jádra a modulů systém rozšiřovat, je důležité zvolit dostatečně obecnou a přenosnou technologii.

4.1 Komunikace

Dlouhou dobu jsem experimentoval s XML, a nakonec jej zavrhl. Je to univerzální jazyk a stalo se módní jej chválit. Je zvláštní, že jsem v literatuře nenarazil na kritiku jeho nedostatků, které ilustruje tento příklad.

```
<FRIENDS>
<PERSON>
  <IDENTIFICATION>
  007
  </IDENTIFICATION>
  <NAME>
  James Bond
  </NAME>
</PERSON>

<PERSON>
  <IDENTIFICATION>
  008
  </IDENTIFICATION>
  <NAME>
  James Watt
  </NAME>
  </PERSON>
</FRIENDS>
```

Obrovské množství metadat. Tak velké, že jsou data samotná pro člověka obtížně čitelná a zápis je možný pouze automaticky. XML a přidružené technologie nabízí možnost formátovat data, třídit je, exportovat do nejrůznějších programů a formátů, ale vše je nutno zapsat explicitně, informace se rozkládají na atomy bez jakýchkoliv souvislostí. Informace má však smysl pouze v kontextu ostatních informací a konetxt XML nezná.

Inspirací mi byl jazyk sázecího programu TeX. V něm by byl zápis mnohem jednodušší:

```
\person{007} James Bond
\person{008} James Watt
```

Většinu metainformací lze totiž určit *kontextuálně*: první parametr ve složených závorkách je číslo, druhý parametr končí nakonci řádku a skládá se ze jména a příjmení. Tuto metainformaci je možno uložit zvlášť a dále ji v zápise neopakovat.

Kupodivu existuje nástroj, který se dá použít v PHP, je mnohem obecnější a mocnější než XML a dá se zužovat bez zásahů do serveru. Tímto nástrojem jsou *regulární výrazy*, pomocí nichž lze i pohodlně převádět ze speciálních formátů do rozšířeného a těžkopádnějšího XML.

Ptal jsem se tedy známých z oboru, proč místo XML nepoužívají regulární výrazy. Odpovědí bylo, že regulární výrazy jsou sice mocný nástroj, ale jejich zápis je příliš složitý a nejsou pro ně programátorské nástroje, které by jej usnadňovaly. Pro jejich využití jsem tedy vytvořil samostatný modul, který toto usnadnění poskytuje a věnoval jim celou kapitolu o komunikaci.

4.1.1 Regulární výrazy

Regulární výrazy má mnoho lidí v podvědomí jako nástroje pro výběr a úpravu textu, osobně jsem se s nimi poprvé setkal v Linuxovém bashi jako s nástrojem pro hromadné zpracování. Tato nejjednodušší třída regulárních výrazů se v PHP nazývá *POSIX Extended Regular Expressions* a je dostupná pomocí funkcí ereg, ereg_replace a split, případně jejich verzí nerozlišujících velikost písma (case-insensitive). Tyto funkce jsou z hlediska možností použití pouze podmnožinou funkcí složitějších tříd, které budou popsány níže.

Složitější třída regulárních jazyků se v PHP nazývá PCRE[2]: Perl-Compatible Regular Expressions a rozšiřuje předchozí třídu zejména o nehladové testování, zpětné reference, podmínky a další rysy vysvětlené v následující podkapitole.

Bylo zajímavé porovnat, jestli vyšší efektivita výpočtu rychlostně vyrovná vyšší náročnost výpočtu.

```
function getmicrotime(){
   list($usec,$sec)=explode(" ",microtime());
   return ((float)$usec+(float)$sec);
}

$fu="/A(.*)Z/U"; //nehladové vyhodnocování
$fg="/A([^Z]*)Z/"; //hladové vyhodnocování
$x="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
$loops=10000;

$start=getmicrotime();
for($i=0;$i<$loops;$i++) ereg_replace("a","E",$x);
$time=getmicrotime()-$start;
echo "POSIX: $time<br/>";

$start=getmicrotime();
for($i=0;$i<$loops;$i++) preg_replace("/a/","E",$x);
$time=getmicrotime()-$start;
echo "PCRE: $time<br/>";
```

Výpočet je 2-3x rychlejší ve prospěch PCRE.

[2] V názvu je *Perl-Compatible*, ale výrazy se od těch perlovských v některých detailech liší: je to dáno jinými možnostmi obou jazyků. Tyto detaily zde nejsou využívány, a tudíž zde nejsou popsány.

Podpora regulárních výrazů však končí u elementárních funkcí umožňujících jejich prostý zápis. CMS však lze nahlížet jako nástroj na přetváření dat: od dat v čisté podobě (jak je definuje zadavatel) k poskytovaným datům (jak by je měli vidět uživatelé) přes systémová data (jak jsou uloženy v IS). Regulární výrazy pak dokáží tuto transformaci provádět průhledně v podobě přehledné tabulky, která abstraktním způsobem reprezentuje všechna metadata a tento pohled jsem použil ve vytvoření zmiňovaného modulu. K vysvětlení tohoto principu je nutno nejprve popsat PCRE syntax a příslušné PHP funkce. Ti, kterým je toto důvěrně známé, mohou přeskočit k podkapitole 3.3, která rozebírá onu tabulku a její zápis.

4.1.2 PCRE Syntax

Materiály, na které jsem narazil, byly buď to příliš povrchní a končily u POSIXu, nebo byly příliš úplné, a tedy složité a nepřehledné [13][14]. Proto jsem tuto podkapitolu pojal stylem stručné příručky, které vše shrnuje přehledně do hloubky, která je zde využita.

Každý PCRE se skládá z *funkčního výrazu* a *modifikátorů*. Funkční výraz je uzavřen mezi zvolené neabecední a nečíselné znaky (většinou se používá /), následován modifikátory, které určují způsob jeho zpracování.

Funkční výrazy (dále označováno F či podmínky) jsou řetězce, které určují způsob testování *zkoumaného řetězce* (dále označováno X či proměnná) zleva doprava. Každý znak X je testován svou identitou v F kromě metaznaků, které mají svůj zvláštní význam většinou sloužící k popisu určité šablony znaků. X se standardně testuje hladově, tj. každá šablona F testuje co největší počet znaků odpovídající šabloně (toto chování lze změnit).

F si lze představit jako funkci či program, X jako zdrojová data. X splňuje F, odpovídají-li všechny šalbony F jejím vyhodnocením v X, jak je uvedeno v příkladech

```
F              X                 výsledek
abba           al jabbar         odpovídá (všechny šablony F
mají vzor v X)
abba           jab baj           neodpovídá
```

Postup při testování prvního případu je přibližně následující[3]*:*

```
f=a, x=a       odpovídá, pokračuje se v testování
f=b, x=l       neodpovídá, pokus o testování F ve zbytku X
f=a, x=_       neodpovídá, pokus o testování F ve zbytku X
f=a, x=j       neodpovídá, pokus o testování F ve zbytku X
f=a, x=a       odpovídá, pokračuje se v testování
f=b, x=b       odpovídá, pokračuje se v testování
f=b, x=b       odpovídá, pokračuje se v testování
f=a, x=a       odpovídají všechny šablony F, testováno
úspěšně, konec
```

Pomocí **metaznaků** lze tvořit složitější šablony. Seznam metaznaků je následující

\ znak nezpracování (escape character)

^ začátek řádku

$ konec řádku

. libovolný znak kromě konce řádku

? předchozí šablona se vyskytuje jednou nebo vůbec

* předchozí šablona se vyskytuje libovolněkrát nebo vůbec

+ předchozí šablona se vyskytuje alespoň jednou

{ začátek kvantifikátoru

} konec kvantifikátoru

[začátek definice třídy

] konec definice třídy

(začátek podvýrazu

) konec podvýrazu

| alternativa

Výraz uvnitř () se nazývá podvýraz, uvnitř {} je kvantifikátor a uvnitř [] třída. Ve třídě se jako metaznaky chápou pouze

\ znak nezpracování

[3] Reálně se používají složitější postupy (např. Knuth-Morris-Prattův algoritmus) umožňující při vyhledávání občas přeskočit dopředu u složitějších šablon, princip sekvenčního vyhledávání však zůstává.

^ negace třídy (pouze na začátku třídy)
- rozmezí třídy
] konec definice třídy

Znak uzavírající F je také metaznakem. Je-li zvolen např. |, nelze už jej použít jako znak alternativy.

Znak nezpracování má několik kontextuálních významů:

- následuje-li metaznak, testuje se jako obyčejný znak. Chceme-li například otestovat ^ jako obyčejný znak, v F to zapíšeme jako \^, podobně znak \ vyhodnotíme jako \\
- následuje-li znak, testuje se jako funkční znak nebo třída znaků:

\n (new line) konec řádku
\r (carriage return) návrat vozíku (konec řádku ve Windows je \r\n)
\d (decimal) libovolný znak čísla
\D libovolný znak kromě čísla
\s (whitespace) znak bílého místa (mezera, nový řádek, tabulátor)
\S libovolný znak kromě bílého místa
\w (word) znak abecedy (malé i velké)
\W libovolný znak kromě znaku abecedy
\b (boundary) konec slova (\w\W nebo \W\w)
\B libovolný znak krom konce slova
\A začátek X
\Z konec X nebo konec řádku v X
\a (alarm) pípnutí
\z konec X

- následuje-li číslo 0-99, označuje odkaz na dříve odpovídající podvýraz podle pořadí

následuje-li xhh, kde h je šestnáctkové číslo, odpovídá znak příslušného šestnáctkového kódu

Třídy slouží ke specifikaci množiny znaků, například samohlásky lze definovat jako [aeiouy]. Znak ^ na první pozici má význam negace třídy, tedy [^aeiouy] odpovídá čemukoliv krom samohlásek. Uvnitř třídy je možno použít i pomlčku jako znak rozsahu, třeba [\dA-F] odpovídá šestnáctkovému číslu. Zápis [^\Waeiouy] odpovídá všem souhláskám. Mají-li se uvedené metaznaky testovat jako normální znaky, je nutné jim dát předponu znaku nezpracování.

Kvantifikátory udávají počet opakování. Nejčastěji používaná konstrukce je asi .* znamenající libovolný počet libovolných znaků. Vzhledem k hladovému testu jej lze vyložit jako vše zbývající do konce výrazu. Krom výše vysvětlených kvantifikátorů *, + a ? lze ještě použít výčtové kvantifikátory, kupříkladu [aeiouy]{2} odpovídá dvěma samohláskám. Specifikovat lze i rozsah, třeba (ha){2,5} odpovídá výrazům *haha* až *hahahahaha*. Je-li druhý parametr vynechán, bere se jako nekonečno: kupříkladu au{3,} odpovdá *auuu*, *auuuuuu* atd. Hladovost odpovídá všem následným po sobě jdoucím u.

Alternativa určuje více možností. Tento výraz (jedničč|dvoj|troj|čtyř|pět|šest)ka odpovídá třeba slovu *dvojka* nebo *šestka*. Alternativu je nutno uzavírat do závorek, aby se poznalo, jak velká část má být alternativní. Díky uzávorkování se testuje jako podvýraz.

Podvýrazy jsou části F uzavřené v kulatých závorkách. Odpovídající části X jsou potom vráceny v některých funkcích (viz 2.1.2). Toto chování není vždy žádoucí (např. u alternativy). Nechceme-li uzávorkovaný výraz zahrnout do výsledku, uvedeme ho znaky ?:, například (?:kouzelník|mág) (\S+) vrátí (vyhodnotí) jako první podvýraz pro X="mág zvedl ruce" slovo *zvedl*, zatímco pro X="bojovník zvedl meč" nevrátí nic. Podvýraz nezahrnutý do výsledku nelze použít pro nížeuvedené zpětné odkazy.

Podvýrazy lze vnořovat, tj. vytvářet podvýrazy uvnitř podvýrazů. V tom případě se při vyhodnocování postupuje zvenku dovnitř a zleva doprava.

Zpětné odkazy vrací otestovaný uzávorkovaný podvýraz, čísluje se od jedničky. Chceme-li například vypsat obsah mezi dvěmi HTML značkami, zapíšeme to jako <([^>]+)>([^<]*)</\1> Obsah je vrácen jako druhý podvýraz, první byl použit pro zpětný odkaz.

Prohledávání testuje znaky nacházející se před výrazem nebo po něm. Je nutno uzávorkovat jako podvýraz, aby bylo jednoznačné, kolik znaků se má testovat, nicméně toto uzávorkování se nikdy jako podvýraz nevyhodnotí!

a) ?= znaky se nacházejí za výrazem. Chceme-li například testovat všechna slova končící středníkem, ale nezahrnout onen středník do výsledku, zapíšeme to jako (\w+)(?=;)

b) ?! znaky se nenacházejí za výrazem. F ve tvaru (\w+)(?!;) kladně otestuje všechna slova, která nekončí středníkem.

c) ?<= znaky se nacházejí před výrazem. Slovo, před kterým se nachází slovo *Bc.* nebo *Mgr.*, testuje tento výraz: (?<=Bc\.|Mgr\.)\s(\w+). Znak tečky je nutno uvést předponou znaku nezpracování, neboť se jinak chápe jako metaznak.

d) ?<! znaky se nenacházejí před výrazem. Kupříkladu otestování slov, před kterými není slovo Frau, se provede touto šablonou: (?<!Frau)\s(\w+).

Testované znaky před řetězcem musí mít pevnou délku, jinak se v PHP vyhodnotí jako chyba při kompilaci, jako třeba v tomto případě (?<=goo+)(gle) nedokáže otestovat *gle* předcházeno znaky *goo*, *gooo*, *gooooo* atd. U alternativ je však různá délka povolena, výraz (?<=Velký|Malý)\s(vůz) kladně otestuje slovo *vůz* v názvu souhvězdí, ale neotestuje jej v jiném kontextu, například *starý vůz*.

Testování pouze jednou se zapisuje podobně jako při prohledávání: uvozující znaky jsou ?>. Při psání dokumentačního modulu jsem se setkal se situací, kdy se měl otestovat výraz obsahující libovolný počet mezer volitelně následovanými hvězdičkami volitelně následovanými mezerami, po kterých se měl vyhodnotit výraz, pokud jeho první znak není zavináč.
Na první pohled správný zápis ^\s***\s*([^@].*) nefunguje. Při vyhodnocování [^@] kompilátor totiž zkusí vyhodnotit znak před ním, jelikož kvantifikátor * znamená libovolný počet znaků (tedy i o znak méně, než hladově otestoval). Tím se splní podmínka a je vrácen řetězec s jedním znakem před zavináčem až do konce.

Testování pouze jednou dává kompilátoru příkaz *už se zpátky nevracej*, požadovaný vzor je tedy ^(?>\s***\s*)([^@].*).

Modifikátory určují nestandardní zpracování F. Uvedeny pouze nejdůležitější.

a) i (*case-insensitive*) nerozlišování velikosti písma abecedních znaků: znaku b tak například odpovídají znaky B a b.

b) m (*multiline*) víceřádkový výraz: standardně je X jednořádkový uvozený znakem ^ a uzavřený $. Víceřádkový výraz takto uvozuje a uzavírá každou řádku. Metaznakový vzor konce řádky zůstává \Z, vzor pro konec výrazu je \z.

c) s (*space extended*) rozšířený význam tečky: metaznak tečka odpovídá i koncům řádků (standardně konec řádku tečce neodpovídá.

d) x (*extra space*) ignoruj bílá místa: bílá místa v F (mezera, tabulátor) jsou ignorována, užitečné pro zpřehlednění složitých výrazů.

e) e (*PHP extended*) PHP rozšíření: Y (viz preg_replace níže) je vyhodnocen jako PHP kód, který provede nahrazení v F (standardně je provedeno nahrazení textovým významem Y)

f) U (ungreedy) nehladové vyhodnocení: vzor v F splňuje co nejkratší úsek v X (standardně u hladového vyhodnocování co nejdelší úsek). Často se uvádí, že tento modifikátor zpomaluje vyhodnocování. Provedl jsem tedy podobný test jako u srovnání POSIX a PCRE, hladové vyhodnocování bylo o 8-20% rychlejší.

4.1.3 Vyhodnocování řetězců

int preg_match (string F, string X [, array R]) provádí vyhodnocování, jak je popsáno výše, výsledek odkazem je vrácen v poli R.

Jestliže podmínka F není splněna, pole R je prázdné a funkce vrátí 0. Je-li F splněna, funkce vrátí 1 a pole R obsahuje na pozici 0 testovaný řetězec, na dalších pozicích s číselnými klíči všechny testované výrazy podle pořadí.

```
$index=<<<INDEX                                                    1
  WWW1, 1                                                          2
  JAVA1, 1                                                         3
  JAVA2, 1                                                         4
INDEX;                                                             5
preg_match("/(\S+),\s?(\d)/m",$index,$results);                    6
print_r($results);                                                 7
výstup:\> Array ( [0] => WWW1, 1 [1] => WWW1 [2] => 1 )            8
```

Výstup neodpovídá očekávání: na pozici 1 je sice vrácen sledovaný předmět a na pozici 2 zapsaná známka, je však vrácena pouze první položka. Kýženého výsleku dosáhneme s využitím následující funkce.

int preg_match_all (string F, string X, array R) Je podobná výšeuvedené funkci s tím rozdílem, že vrátí pole, jehož prvky jsou pole – jednotlivé výsledky předchozí funkce. Výsledek je v tomto případě.

```
výstup:\> Array (                                                  1
 [0] => Array ( [0] => WWW1, 1 [1] => JAVA1, 1 [2] => JA-
VA2, 1 )                                                           2
 [1] => Array ( [0] => WWW1 [1] => JAVA1 [2] => JAVA2 )            3
 [2] => Array ( [0] => 1 [1] => 1 [2] => 1 )                       4
)                                                                  5
```

V prvním prvku je vráceno pole předmětů, ve druhém pole známek.

Je-li žádoucí tvar výsledku pole s prvky polí ve tvaru (*1 => předmět, 2=> známka)*, lze toho docílit nepovinným třetím parametrem int F. Je-li (implicitně) nastaven na PREG_PATTERN_ORDER, je výsledek vrácen ve výšeuvedeném tvaru, je-li nastaven na PREG_SET_ORDER, je výsledek v nížeuvedeném tvaru:

```
Array (
 [0] => Array ( [0] => WWW1, 1 [1] => WWW1 [2] => 1 )
 [1] => Array ( [0] => JAVA1, 1 [1] => JAVA1  [2] => 1 )
 [2] => Array ( [0] => JAVA2, 1 [1] => JAVA2 [2] => 1 )
)
```

Obě funkce mohou mít jako nepovinný čtvrtý parametr int F také PREG_OFFSET_CAPTURE, který způsobí, že jednotlivé prvky výsledku nejsou řetězce podvýrazů, ale pole obsahující dva prvky: pozici nalezeného podvýrazu v X a podvýraz samotný[4].

4.1.4 Úprava řetězců

mixed preg_replace (mixed F, mixed Y, mixed X [, int L])
v řetězci X nahradí šablonu F řetězcem Y. Nepovinný parametr L udává maximální počet nahrazení. Na uzávorkované podvýrazy v F je možné v Y odkazovat pomocí \\n nebo $n, kde *n* je pořadí uzávorkovaného podvýrazu[5].

```
$f="/(\s\d)/";                                                  1
$g='známka$1';                                                  2
print_r(preg_replace($f,$g,"WWW1, 1;JAVA1, 2;JAVA2 3",2);       3
výstup:\> WWW1, známka 1;JAVA1, známka 2;JAVA2 3                4
```

X může být zadán i ve formě pole řetězců: pak je nahrazení provedeno v každém prvku podobně jako u funkce preg_match_all.

[4] v případě preg_match_all je vrácený výsledek ve tvaru pole polí s prvky polí, což je nejkomplexnější tvar ze všech standardních PHP funkcí.

[5] Způsobem \\n není možné rozlišit případy typu \\11: vzniká nejednoznačnost, jedná-li se o jedenáctý podvýraz, nebo o první podvýraz následovaný jedničkou. Proto se preferuje způsob $n, který je možné ohraničit kvantifikátorem, např. ${1}1 znamená první podvýraz následovaný jedničkou.

```
$f="/(\s)(\d)/";                                                    1
$g='$1známka $2';                                                   2
$x=array("WWW1, 1", "JAVA1, 2", "JAVA2, 3");                        3
print_r(preg_replace($f,$g,$x));                                    4

výstup:\> Array (                                                   5
 [0] => WWW1, známka 1                                              6
 [1] => JAVA1, známka 2                                             7
 [2] => JAVA2, známka 3                                             8
)                                                                   9
```

Pokud jsou F a Y pole, provede se nahrazení odpovídající každé dvojice prvků v odpovídajícím pořadí. Toto pořadí obecně není shodné s hodnotou klíče, proto je nutné pro korektní nahrazení obě pole uspořádat podle klíče standardní PHP funkcí ksort.

```
$f=array(1=>"/1/", 2=>"/2/", 3=>"/3/");                             1
$g=array(3=>"trojka", 2=>"dvojka", 1=>"jednička");                  2
$x=array("1 2 3", "3 2 1");                                         3
print_r(preg_replace($f,$g,$x));                                    4
ksort($f); ksort($g);                                               5
print_r(preg_replace($f,$g,$x));                                    6

výstup:\>                                                           7
Array ( [0] => trojka dvojka jednička [1] => jednička dvoj-
ka trojka )8
Array ( [0] => jednička dvojka trojka [1] => trojka dvojka
jednička )9
```

V případě modifikátoru \e se obsah G bere jako PHP kód, který vrátí řetězec, který se použije jako nahrazení. Následující skript opraví text v proměnné $string tak, aby na začátku vět byla velká písmena.

```
$f="/(^|\.(?:\s|\n))(\w)/me";                                       1
$g="'\\1'.strtoupper('\\2')";                                       2
print_r(preg_replace($f,$g,$string));                               3
```

mixed preg_replace_callback (mixed F, callback C, mixed X [, int L]) v řetězci X nahradí šablonu F voláním C, což je funkce s parametrem pole, na jejíž vstup jsou předány odpovídající výstupy F. Jinak se chová jako předchozí funkce. Podrobněji rozvinuto v 2.1.3.

4.1.5 Ostatní funkce

array preg_grep (string F, array X) provádí výběr pole. Za X se dosazuje pole řetězců, výsledek je vrácen jako pole prvků, které splňují podmínku F.

```
$olympionics2006=array(
        "běžky01"=>"Neumannová",
        "běžky02"=>"Bauer",
        "brusle01"=>"Bémová",
        "něco01"=>"Kraus");
print_r(preg_grep('/(\w+á$/',$olympionics2006));
```

Vrátí pole všech žen v seznamu se zachováním klíčů, podobně jako v Linuxové funkci grep.

string preg_quote (string S [, string D]) uvede všechny metaznaky v řetězci znakem nezpracování a vrátí takto upravený řetězec. Nepovinný druhý parametr je uvozovač výrazu, který bude také uvozen znakem nezpracování (implicitní uvozovač je /).

array preg_split (string F, string X [, int L [, int P]]) vrátí pole, jehož prvky vzniknou rozpadem X na hranicích popsaných F. Maximální počet hranic, které jsou určovány od počátku řetězce, lze udat v nepovinném parametru L. Nakonec je možno uvést nepovinný parametr P, který má podle hodnoty následující význam:

PREG_SPLIT_NO_EMPTY: do výsledku nebudou zahrnuty prázdné prvky

PREG_SPLIT_DELIM_CAPTURE: uzávorkovaný podvýraz je také vrácen ve výsledku

PREG_SPLIT_OFFSET_CAPTURE: prvky výsledku nebudou podřetězce, ale dvouprvkové pole obsahující pozici prvku v původním řetězci a podřetězec, který vznikne stejným způsobem, jako v ostatních případech. Hodnoty je možné kombinovat operátorem &.

4.1.6 RET

Využívání regulárních výrazů voláním uvedených PHP funkcí je programování ad hoc. Při aplikaci v CMS je zapotřebí obecnějších struktur, které by se daly univerzálně využít.

Lze využít vlastností funkce preg_replace, viz 3.2: F (vstup), Y (nahrazení) a X (program) mohou být pole. To lze nahlížet jako příkazy interpretu: F pak obsahuje definici příkazů, Y jejich vykonání.

Tuto vyšší vrstvu lze oddělit například tak, že definice F a Y jsou v samostatném souboru, nezávisle na X.

```
soubor fy.ret                                                    1
"/(\w+(ová))/"   "paní $1"                                       2
"/bysme/"        "bychom"                                        3
soubor programu                                                  4
$fy=file("fy.ret");                                              5
foreach($fg as $line) list($f[],$y[])=explode("\t",$line);       6
$x="já, Nováková a Neumannová bysme zítra rádi přišli na
oběd.";                                                          7
echo preg_replace($f,$y,$x);                                     8
výstup:\> já, paní Nováková a paní Neumannová bychom zítra
rádipřišli na oběd.                                              9
```

Pomocí PCRE lze zapisovat podmínky ve tvaru (?(podmínka)ano|ne), cykly lze vytvářet pomocí kvantifikátorů. Tyto konstrukce lze však použít pouze na zpracovávaný řetězec, nikoli na proměnné, neboť PCRE mají *referenční transparentnost*[6]. Bylo by tedy výhodné začlenit PHP funkce a objekty přímo do regulárních výrazů. Smyslem tohoto snažení je vytvořit univerzální rozhraní, přes které si objekty přizpůsobují vstup, který tak může být v obecném formátu.

Pro názornost následující příkazy využívají tyto testovací funkce:

[6] to znamená, že jednou použitá paměťová místa nelze změnit až do konce programu. To má za následek vyšší spotřebu paměti (například hodnotu zvýšenou o jedničku nelze prostě přepsat, je nutno vytvořit nové paměťové místo), ale také vyšší rychlost: žádná operace se neprovádí dvakrát, na cokoli jednou spočítané program už jen odkazuje. Tento přístup se nazývá funkcionální programování, neboť jde vlastně o předávání mezivýsledků funkcí dalším funkcím, proměnné tak nejsou zapotřebí.

```
// mějme nějakou základní funkci a třídu                    1
                                                            2
function test_function($name) {return "zdraví ".$name;}    3
                                                            4
class Foo { var $salut;                                     5
  function Foo($salut) {$this->salut=$salut;}               6
  function test_class($name="třída") {return "$this->salut
$name";}                                                    7
} $class= new Foo("zdraví");                                8
                                                            9
echo test_function("funkce");                               10
echo $class->test_class("třída");                           11
                                                            12
výstup:\> zdraví funkce zdraví třída                        13
```

Volání test_function($parametr) vrátí jak u obyčejné funkce (3), tak u metody objektu (5-8) řetězec *zdraví $parametr*, (10, 11) na výstup (13). U objektu je navíc podřetězec *zdraví* jako objektová proměnná, což testuje možnosti přístupu k ní.

K manipulaci s funkcemi slouží zmíněná PHP funkce preg_replace_callback. Uživatelsky viděno je stejná jako preg_replace, akorát místo řetězce Y používá přímo PHP funkci C, na jejíž vstup je předáno pole M, které obsahuje otestované výrazy v F.

```
echo preg_replace_callback(
  "/@pozdrav\s+(\S+)/",                                     //=f
  "\$class->test_class,"                                    //=c
  "@pozdrav třída"                                          //=x
  );

výstup:\> Warning: preg_replace_callback() requires argu-
ment 2, '$class->test_class', to be a valid callback in
c:\www\pokus.php
```

Zdá se, že kompilátor nerozpozná objektovou funkci (obyčejné funkce jsou volány úspěšně). Vzhledem k omezení parametru nelze metodu volat ani nepřímo (pomocí call_user_method).

Jako C však může být obyčejná funkce, která zavolá funkci objektovou. Aby byl zachován *jmenný prostor* skriptu, je tato funkce volána anonymně konstruktem create_function, aby proměnná existovala i mimo jmenný prostor anonymní funkce, je nutno ji uvést konstruktem global:

```
echo preg_replace_callback(                                    1
  "/@pozdrav\s+(\S+)/",                                        2
  create_function('$m',                                        3
    'global $class; return $class->test_class($m[1]);'         4
  )                                                            5
  ,"@pozdrav třída"                                            6
);                                                             7

výstup:/> zdraví třída                                         8
```

Tento způsob má tu nevýhodu, že parametr volané funkce musí být pouze pole obsahující hodnoty uzávorkovaných výrazů a nic jiného. Toto omezení lze obejít tím, že je tato funkce volána ve funkci složitější, PHP však nabízí pohodlnější vkládání PHP kódu do regulárních výrazů pomocí modifikátoru \e. Je však bezpečnostní nutností[7] složitějších programů (jakým je například CMS) mít je uzavřeny ve jmenných prostorech, které lze v PHP vytvořit pouze pomocí objektů.

```
// nyní pomocí nich nechme zpracovat příkaz @pozdrav pomocí
PCRE                                                           1
$f="/@pozdrav\s+(\S+)/e";                                      2
echo preg_replace($f,"test_function('\\1')","@pozdrav funk-
ce");                                                          3
echo preg_replace($f,"\$class->test_class('\\1')","@pozdrav
třída");                                                       4

výstup:\> zdraví funkce zdraví třída                           6
```

Skript má ukázat použití PHP funkce v regulárním výrazu. Na řádku 3 a 4 jsou volány podřetězcem @pozdrav, jako parametr je jim přidán první uzávorkovaný regulární výraz.

Na řádku 4 je nutno uvést znakem nezpracování *$*, aby se nezpracoval jako metaznak regulárního výrazu, ale jako proměnná PHP[8].

[7] krom bezpečnosti vnější, (útoky zvnějšku) je tím míněna také bezpečnost vnitřní (konzistentní struktura)

[8] aby se nezpracoval ani jako metaznak ani jako proměnná, je nutno ho uvést třemi zpětnými lomítky: \\\$ po zpracování regulárním výrazem dá na mezivýstup \$, který zpracuje PHP jako znak $.

4.1.7 Rušení

Některé objekty potřebují komunikovat se serverem a hrozí jejich případná kolize. Aby k ní nedocházelo, musí být všechny externí proměnné uvozeny předponou: unikátním klíčovým slovem mezi moduly.

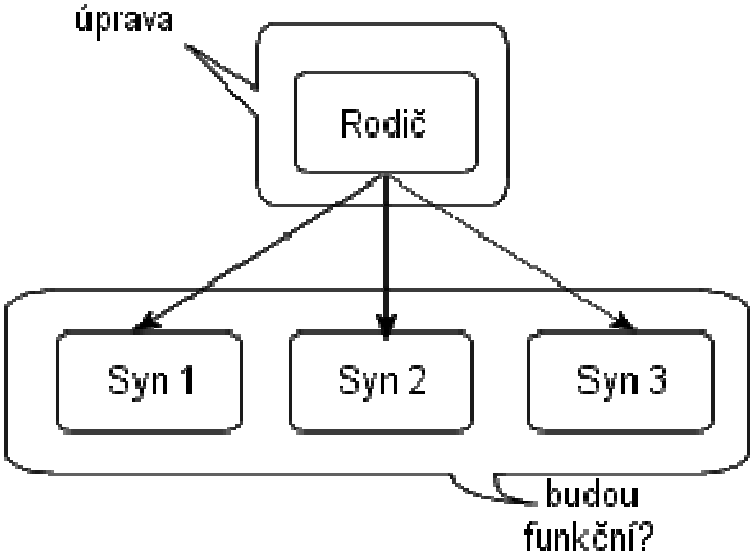
Zajištění unikátnosti je prozatím ponecháno na správci serveru. Pro kontrolu lze využít funkce `get_names_of_vars`. Vytvoření globální tabulky unikátních jmen je totiž proti filosofii omezených funkcí jádra. Pro tyto účely by s nárůstem složitosti systému bylo vhodné rozšířit modul, který kolize kontroluje. Vzhledem k doposud únosné složitosti systému toto dosud nebylo zapotřebí, kontrolní modul zatím pouze informuje o provázanosti tříd.

4.2 Architektura

V klasickém objektovém programování se objekty volají navzájem. Tento chaotický přístup působí rostoucí nepřehlednost vazeb a jedna zdánlivě nevinná změna může rozhodit celý systém. Tento problém lze vyřešit kvalitním *UML* návrhem, který je však vhodný pouze pro objekty, což nejsou vždy nejvhodnější datové struktury. Je tedy žádoucí vymyslet návrh obecnější. Jedno z řešení je neobjektové struktury do objektu zapouzdřit jako statické funkce.

U CMS, který není navrhován pro konkrétní IS, však neznáme aplikační pohled, tj. nevíme, kterým směrem se bude jeho vývoj ubírat. Rozšiřování obvykle funguje až do okamžiku, kdy je třeba provést změnu. Problém znázorňuje Obr. 4.

Obr. 4: úprava tříd



Je tedy vhodné vyhnout se obecným strukturám závislostí a aplikovat pouze strukturu stromovou: není pak nutné pátrat po závislostech v celém systému, ale pouze po potomcích.

Dále by se nemělo dědit z tříd modulů, jelikož ty mají být nezávislé. Nebylo-li by tomu tak, zdánlivě nevinná úprava modulu by mohla ohrozit funkčnost na všech systémech, které ho využívají. Nezávislost zaručuje, že se na všech systémech změní pouze onen modul, což je naopak žádoucí rys: provozujeme-li více systémů a potřebujeme-li změnit modul, který se nachází na více z nich, stačí změny provést na jednom místě, což zároveň zaručuje jejich konzistenci.

4.2.1 Zvláštnosti PHP

Jazyk PHP se stal plně objektovým až od verze 5.0, kdy v něm byly plně implementovány polymorfismus a zapouzdření. Do té doby bylo možno využívat pouze dědičnosti, což snižovalo objekty na úroveň šablon a jmenných prostorů.

Důležitým rozdílem PHP 5 oproti jiným objektovým jazykům je nedodržením zásady identity objektů (každý objekt má mít svůj vlastní paměťový prostor). Odkazy v PHP nejsou totiž ukazatelé.

Zatímco ukazatel ukazuje na paměťový prostor cíle, *odkazy* pouze ukazují do vnitřní tabulky přejmenování (aliasů), která odkazuje jedno paměťové místo. To znamená, že provázání je narozdíl od ukazatele obousměrné: úprava odkazu změní cíl a naopak. V tabulce se navíc nerozlišuje typ odkazu, jak dokládá nížeuvedený příklad, který pochází ještě z dob vývoje této práce pod PHP 4, kde nebyla rozhraní.

```
class interfaceBar {                                          1
   var $var;                                                  2
   function interfaceBar($var) {$this->var=$var;}             3
}                                                             4
                                                              5
class Bar extends interface {                                 6
   function Bar(&$obj) {                                      7
    if(is_subclass_of($obj,"interfaceBar"))                   8
     $this->var=&$obj->var;                                   9
    else                                                      10
     $this->var=$obj;                                         11
   }                                                          12
}                                                             13
                                                              14
$a=new Bar(create_function('',''));                           15
$b=new Bar($a);                                               16
$a->var="y";                                                  17
echo $b->var; //y                                             18
```

Provázání je na řádku 9 obousměrné! Pokud bychom změnili sdílenou proměnnou v objektu `$b`, změna by se promítla i do `$a`. Tento rys může být nežádoucí, ale jednosměrné odkazy v PHP nelze zapsat, což je asi nejzávažnější omezení PHP objektů.

Řádek 15 ukazuje netypovost odkazů: místo ukazatele na proměnnou je zde ukazatel na anonymní funkci (anonymní proměnné v PHP 4 vytvářet nelze).

PHP nabízí i funkce pro manipulaci s objekty, jak je vidět v příkladu na řádku 8. To však není žádné rozšíření oproti jiným jazykům, jsou například funkčně ekvivalentní s podmnožinou třídy Object jazyka Java.

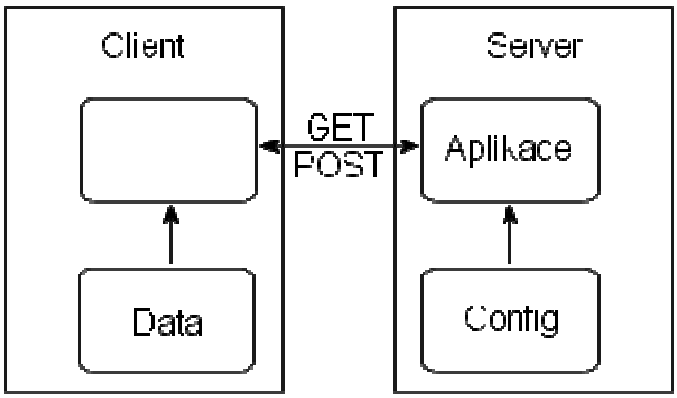
4.2.2 Distribuovanost

Základ systému tvoří třídy v adresáři classes a rozhraní v adresáři interfaces. Ty jsou využívány pro moduly v adresáři modules, kde má každý svůj vlastní podadresář.

Ke vzájemné komunikaci je využíváno regulárních transformací, což objekty činí na sobě nezávislé (výstup jednoho objektu lze přizpůsobit vstupu druhého objektu). Tohoto rysu lze s výhodou využít právě pro distribuovanost, například využívat moduly i v IS bez PHP serveru.

Pro tyto účely obsahují moduly (které jsou umístěny na serveru) soubory config.csv, kde jsou ukládána nastavení pro klienty, toto nastavení je možné uživatelským skriptem souborů config.php. Klient pak umístí modul do svého systému přes HTML rám a obsluhuje jej krom serverového nastavení také GET a POST proměnnými a soubory na svém serveru, které modul využívá.

Obr. 5: komunikace server-klient

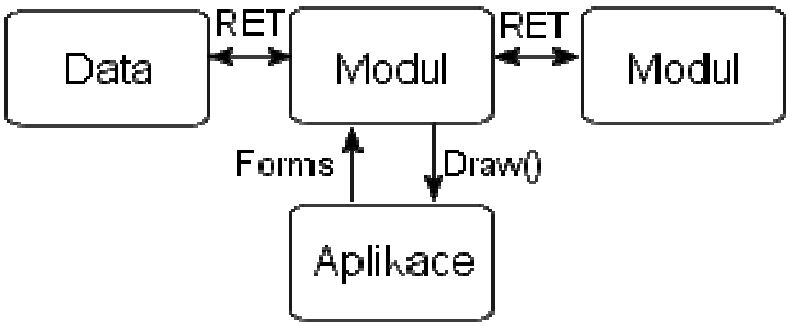


4.2.3 Jádro

Jak bylo vysvětleno v 2.2.4, systém má fungovat na základě spolupráce nezávislých modulů, které mohou být umístěny na různých serverech. Vstupy a výstupy modulů lze přizpůsobit pomocí *RET (regular expression table)*, viz 4.1.6.

Z tohoto pohledu systém nemá žádné jádro jako kód, ale pouze jako komunikační protokol RET.

Obr. 6: komunikace mezi moduly



Základy pokročilé práce s regulárními výrazy byly vysvětleny v 4.1. Nenachází-li se ve zpracovávaném textu žádné kontextuální informace, je možné jej přímo transformovat, v opačném případě je schůdné načíst jej do PHP proměnné a tu dále podle kontextu zpracovat.

Definujme si tedy RET jako posloupnost uspořádaných dvojic (F, X), kde F je vzor a X obraz transformace. Nechť se jedná o načítání do paměti, pokud X začíná na *$*. Tento metaznak ovšem musíme rozlišit od znaku *$*, stejnětak znaky ″ a ′, kterými se uzavírají víceslovné řetězce. Znak *$* je chápán jako metaznak pouze na začátku řetězce, uvozovky jsou metaznaky, není-li před nimi znak nezpracování zpětné lomítko, což lze zapsat výrazem

```
"/(\"|\')(.*)(?<!\\\)\\1"                                    f
```

Objektové možnosti nabízejí schůdnou cestu, jak načíst data do proměnné bez rizika kolize: modul jádra načte proměnné do asociativního pole, které je chráněnou proměnnou třídy. Cílový objekt se pak z jádra zdědí.

Mějme kupříkladu RET, která nahradí bšechny výskyte *bysme* slovem *bychom* a načte do proměnné všechna slova na konci vět, která obsahují *e*.

```
soubor template.ret
"/bysme/"       "bychom"                                    1
"/(\w*e\w*)\./"  "$eslova"                                  2
```

Výstup by mohl vypadat nějak takto:

```
class A extends RET {                                       1
  function tell() {print_r($this->data);}                   2
}                                                           3
$a= new A('template.ret');                                  4
echo $a->run('Včera. Dnes. Zítra. Měli bysme se připra-
vit.');                                                     5
$a->tell();                                                 6
Výstup\:>                                                   7
Včera. Dnes. Zítra. Měli bychom se připravit.               8
Array ( [eslova] => Array ( [0] => Včera [1] => Dnes ) )    9
```

4.2.4 Modul Kalendář

Tento modul názorně využívá distribuovanosti systému, viz **Chyba! Nenalezen zdroj odkazů.**. Data na klientovi jsou soubory HTML zobrazující události k určitým dnům. Pomocí skriptu nebo zásahem správce serveru, na kterém běží kalendář se změní soubor config.csv, jehož struktura je například:

```
CalendarId=klubzsf.ic.cz; TargetFrame=main; SourceFile=http://klubzsf.ic.cz/cal.csv;
CalendarId=turonj00; CSSFile=cal.css
```

Klient použije modul tím, že zavolá server s GET proměnnou CalendarId, která mu byla serverem přidělena. Do stránek je může začlenit například HTML příkazem:

```
<iframe src=http://adresaserveru.cz/calendar.php?CalendarId=klubzsf.ic.cz/>
```

Server pak u sebe spustí instanci modulu kalendář s parametry zadanými na příslušném řádku. Nastavuje se:

- **TargetFrame** cílový rám u klienta, kam se zobrazují odkazy v kalendáři
- **SourceFile** zdrojový soubor (zpravidla u klienta, nebo i jinde) obsahující údaje o událostech kalendáře
- **CSSFile** CSS, který se má použít při zobrazení kalendáře

Uvedený HTML příkaz vykreslí kalendář ve formě tabulky s klientem definovanými styly podle souboru CSSFile a funkčními odkazy podle souboru SourceFile, jehož struktura jsou řádky ve tvaru den;název;cíl;styl, například:

```
31.12;Silvestr;klubzsf.ic.cz/data/silvestr.html;1
4.6.2007;narozeniny; klubzsf.ic.cz/data/narozeniny.html;2
```

kde:

- **den** je den, ke kterému se vztahuje událost. Je-li vynechán rok, událost se zobrazuje každoročně, je-li vynechán měsíc, událost se zobrazuje každý měsíc v daný den.
- **název** je typ události: zobrazí se jako titulek po přejetí příslušněh odne v kalendáři myší.
- **cíl** je soubor, který se zobrazí při kliknutí na odkaz v příslušném dni
- **styl** jeden ze dvou stylů zobrazování odkazů, způsob, jak zvýraznit důležité události. Není-li uveden, zobrazí se prvním stylem.

Na klientském serveru tedy nemusí běžet PHP a klient nemusí znát kód modulu, přesto má nad ním plnou kontrolu. Ukazuje se také další výhoda: klient u sebe může zobrazovat i jiné kalendáře a naopak, může svůj kalendář sdílet s jinými servery.

Otázka bezpečnosti se zde neřeší: logicky nespadá do kompetencí kalendáře, zabezpečení by měl obstarávat samostatný modul.

Klient by však neměl být od kódu odstíněn. Je totiž možné, že:

- potřebuje si modul upravit pro svůj systém
- chce do modulu vidět z provozních či bezpečnostních důvodů

4.2.5 Modul Dokumentace

Tento modul je ilustrací komunikace mezi jednotlivými částmi systému. Vstupem jsou zde komentáře zdrojového kódu, požadovaným výstupem je přehledná HTML dokumentace. To zahrnuje dvojí zpracování:

- oddělení dokumentačních komentářů od ostatních částí kódu
- porozumění obsahu komentářů

První část realizuje následující RET:

```
"/\/\*\*\r?\n(.*)\*\/\r?\n(.*)\r?\n/sU"  "$functions"
"/^(.*)\/\*\*(.*)\*\//mU"                "$variables"
```

Dokumentační komentáře jsou uvozeny /**.

Do pole $functions se načtou komentáře funkcí: jsou to všechny víceřádkové dokumentační komentáře, za kterými následuje hlavička funkce, která se také načte (kontaxtuálně se tak určí, který komentář patří které funkci.

Do pole $variables se načtou komentáře proměnných: jsou to všechny jednořádkové dokumentační komentáře, před kterými je deklarace proměnné. Opět se zde využívá kontextu.

Porozumění obsahu realizuje tato RET:

```
"/@author (.*)/"                         "$author"
"/@version (.*)/"                        "$version"
"/@codepage (.*)/"                       "$codepage"
"/@return (.*)/"                         "$return"
"/@var (\$\w+)\s+(.*)/"                  "$var"
"/^(?>\s*\**\s*)([^@].*)/m"              "$text"

"/\$([^\s=<]+)/"                         "\$<span
class=\"Cvar\">\1</span>"
```

Inspiraci jsem načerpal z tvorby dokumentace jazyka Java [12], význam jednotlivých parametrů je stejný jako tam. Parametry jsou načteny do proměnných, viz 4.2.3.

Poslední řádek této RET je příkladem přímé transformace: komentář proměnné je rovnou vrácen obalen CSS a nezpracovává se jako PHP proměnná.

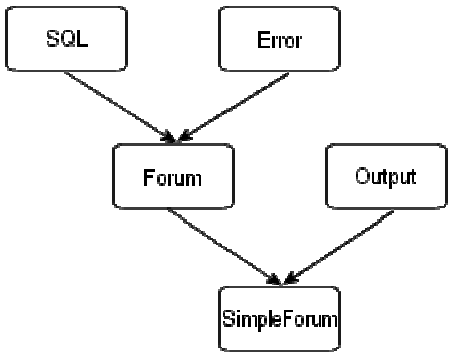
Programátorsky zajímavé je, že u obou druhů zpracování se s výhodou používají různé instance jádra (jinými slovy: modul běží na dvou jádrech): je tak zachován jmenný prostor a máme přehled o tom, který parametr se vztahuje ke které části.

4.2.6 Modul Fóra

Tento modul ukazuje složitější než jednogenerační dědění modulů distribuovaného systému.

Základem je třída SQL, která obhospodařuje připojení k MySQL databázi. Z ní dědí třída Forum, která se stará o všechny operace s tabulkou fóra v databázi. Z ní potom dědí modul SimpleForum, který vykreslí uživatelské rozhraní a stará se o komunikaci mezi uživatelem a serverem.

Obr. 7: schéma fóra



Toto logické členění má oproti komplexnímu řešení tyto výhody:

- **Znovuvyužitelný kód**. Čím výše v diagramu se třída či rozhraní nachází, tím vyšší abstrakci představuje, tím vzdálenější operace od výsledku nejspodnější třídy provádí. Pro podobné moduly lze tak využít kód již dříve napsaný, není nutno začínat od začátku.
- **Vymezení**. Každá část se zabývá vymezenou částí řešení. V třídě SimpleForum kupříkladu není žádná manipulace s tabulkami databáze (tím se zabývá třída Forum) a naopak: v třídě Forum se negeneruje žádná část výstupu (tím se zabývají pouze třídy implementující rozhraní Output).
- **Ladění**. Nastane-li chyba, je díky této hierarchii většinou jasné, která část ji způsobila: není pak nutno prohledávat celý kód, ale jen jeho logickou část. Známé a z modulu neodstranitelné chyby (například nenavázání spojení s databázovým serverem) jsou jednotně zpracovávány přes rozhraní Error.

ZÁVĚR

5.1 Realizační

5.1.1 Rozšiřitelnost

Systém je vyvíjen několik měsíců, což je v porovnání s fungujícími systémy příliš krátká doba na odstranění všech nedostatků. Přesto zažil změnu technologií: přechod od PHP 4 k PHP 5, což zahrnuje změnu od procedurálního k objektovému programování. Nesnáze byly naznačeny v kapitole 4.2.1 a nejsem si jist, bylo-li by starou technologií rozšiřování systému únosné. Díky PHP 5 se snadno programují a ladí i složitější struktury, jak bylo ukázáno v 4.2.6.

Začal-li by počet tříd systému narůstat, bylo by zapotřebí mezi nimi zavést hierarchii, pravděpodobně podobnou struktuře balíčků jazyka Java. Vzhledem k tomu, že servery mají být specializované, nemělo by k přílišnému nárůstu tříd dojít. Jména tříd pak lze přehlédnout jedním pohledem, je pak snadné pojmenovat je unikátně.

5.1.2 Distribuovanost

Ukázalo se (např. 4.2.4), že distribuovaný systém pro správu obsahu lze využívat v mnoha systémech. Díky velmi přizpůsobivé komunikaci jednotlivých částí (4.1.6) a samostatnosti modulů si může každý zvolit složitost systému, jehož řízení má plně pod kontrolou správce klientského systému.

Brojí se tak proti dogmatu, že *správce má být od kódu odstíněn a má se soustředit na aplikace* (3.1.3). Domnívám se totiž (nepodloženě), že argument, že správce nemusí být programátor, je pouze zástěrka toho, že se tvůrce CMS za svůj kód stydí (například kvůli nepřehlednosti či primitivnosti jednotlivých částí), což se snaží zakrýt vizuálním efektem výsledku. Místo toho předpokládám, že správce je inteligentní programátor, a tedy je účelné mu zkoumání (a případný rozvoj) systému usnadnit. Zavedené dogma jsem se snažil nahradit větou, že *správce má právo být od kódu odstíněn, má však také právo na pochopení kódu*.

5.1.3 Šířka

V době prezentace jsem systém vyvíjel a využíval pouze na svých projektech. Možná jsem trochu přecenil své síly, když jsem se domníval, že systém dotáhnu do té míry, že bude pro ostatní inspirující na něm pokračovat.

Doufám však, že jsem dostatečně názorně vysvětlil potenciál myšlenky distribuovanosti na CMS. Bude-li někdo v práci pokračovat či pomůže-li někomu překonat stereotyp rozvíjení systému od začátku, budu rád.

5.2 Cíle

Předsevzetí cílů je popsáno v 3.3.

5.2.1 Instalace jádra

Jako komunikační protokol stačí zavést třídu RET.php, pokud má systém sloužit jako distribuovaný server. Třída pomocí RET zajišťuje přizpůsobení vstupu všem objektů, které požadují formátovaný vstup.

5.2.2 Rozvržení

Je požadována nezávislost modulů na ostatních modulech (4.2). Modul může využívat pouze třídy, které se (už z omezení PHP) musí nacházet na serveru, kde se nachází sám modul.

Jinými slovy, funkčnost modulu je plně určena serverem, na kterém se nachází, lze tedy určit, na kterém serveru nastala chyba.

5.2.3 Dokumentace

Dokumentace je vyřešena samostatným modulem automatickým procesem za využití RET. Tím je zajištěna jednotnost dokumentace všech částí vygenerovaných tímto modulem.

Krom jednotnosti je výraznou výhodou rychlost automatického procesu: v praxi totiž tvorba dokumentace obvykle trvá relativně dlouho.

5.2.4 Objektová struktura

Ukazuje se, že objekty jsou nejvýhodnější struktura všech částí systému. Důvody jsou zejména:

- možnosti použití UML do budoucna
- bezpečnost (jmenný prostor a přístup k datům)
- od PHP 5 mohou mít objekty i statické části, které není nutné zavádět dynamicky

5.3 Přínos

- Práce se vydala ve své oblasti značně průkopnickou cestou.
- Upozorňuje na možnosti regulárních výrazů, využívá je a nabízí vyšší funkce pro manipulaci s nimi (4.1.6).
- Aplikuje a široce využívá objektové možnosti PHP 5 i v oblasti CMS, které se doposud vyvíjely procedurálním stylem bez důvodné potřeby objektů.

REJSTŘÍK A SEZNAMY

6.1 Rejstřík

6.2 Seznam obrázků

6.3 Seznam použité literatury

Všechny uvedené odkazy jsou citovány z 26.11.2006.

[1] *phpMyAdmin 2.6.0-pl3* [databáze on-line].
Dostupné z URL: <http://mysql.webzdarma.cz/>.

[2] *The CMS matrix*.
URL: <http://www.cmsmatrix.org/>

[3] *pipni.cz*

[4] *home.pf.jcu.cz*

[5] *CMSimple Content management*.
URL: <http://cmsimple.dk/>

[6] *plone.org*.
URL: <http://plone.org/>

[7] *definitive guide to plone*. s. 8-9, 74, 205-206
URL: <http://plone.org/documentation/manual/definitive-guide/definitive_guide_to_plone.pdf>

[8] *PHP-Nuke*.
URL: <http://phpnuke.org/>

[9] *EZ's NukeCalendar 1.1a Modification Beta 3: soubor install.txt*. [textový soubor ZIP archivu] Dostupné z URI:
<http://phpnuke.org/modules.php?name=Downloads&d_op=getit&lid=380>

[10] *Redakční systém a publikační systém phpRS*.
URL: <http://www.supersvet.cz/phprs/>.

[11] *phpRS 2.8.0: soubor info_cz_od_2-0-0_vyse*. [textový soubor v ZIP archivu]
Dostupné z URI:
<http://www.supersvet.cz/download.php?soubor=98>

[12] *javadoc - The Java API Documentation Generator*.
URI
<http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html#javadoctags>

[13] *PHP: Pattern Syntax - Manual*.
URL: <http://cz.php.net/manual/cs/reference.pcre.pattern.syntax.php>

[14] *PHP: Pattern Modifiers - Manual*.
URL: <http://cz.php.net/manual/cs/reference.pcre.pattern.modifiers.php>